RQF LEVEL 4

GENFC401

COMPUTER SYSTEM AND ARCHITECTURE

Fundamentals Of C++ Programming

TRAINEE'S MANUAL

October, 2024

RTB | RWANDA TVET BOARD

FUNDAMENTALS OF C++ PROGRAMMING

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate IV in Computer System And Architecture, specifically for the module **"GENFC401:FUNDAMENTALS OF C++ PROGRAMMING."**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

PRODUCTION TEAM

Authoring and Review

HAKIZIMANA Evariste

MUSABYIMANA Xavier

Validation

NDABAKURANYE Pierre Claver

NISHIMIRWE Liliane

UWAMAHORO Bonaventure

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

BYUKUSENGE Protais

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

MANISHIMWE Marc

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

CLI: Command-Line Interface.

CPU: Central Processing Unit

CSS: Cascading StyleSheet

CSV: Comma-Separated Values

Dev-C++: Developer-C++.

ECU: Engine Control Unit

GUI: Graphical User Interface.

HTML: Hypertext MarkUP Langauge

I/O: Input/Output

IDE: Integrated Development Environment.

Ios: Input/Output Stream.

JSON: Javascript Object Notation

LNN: LongNamespaceName

MinGW: Minimalist GNU for Windows1**.**

MSVC: Microsoft Visual C++

MySQL: My Structured Query Language.

OOP: Object-Oriented Programming

PHP: Hypertext Preprocessor

ptr: Pointer

RAM: Random Access Memory

RQF: Rwanda Qualification Framework

RTB: Rwanda TVET Board

std: Standard

TQUMProject: TVET Quality Management Project

XML: Extensible Mark Up Language

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Computer System and Architecture specifically for the module of **"Fundamentals of C++ programming".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

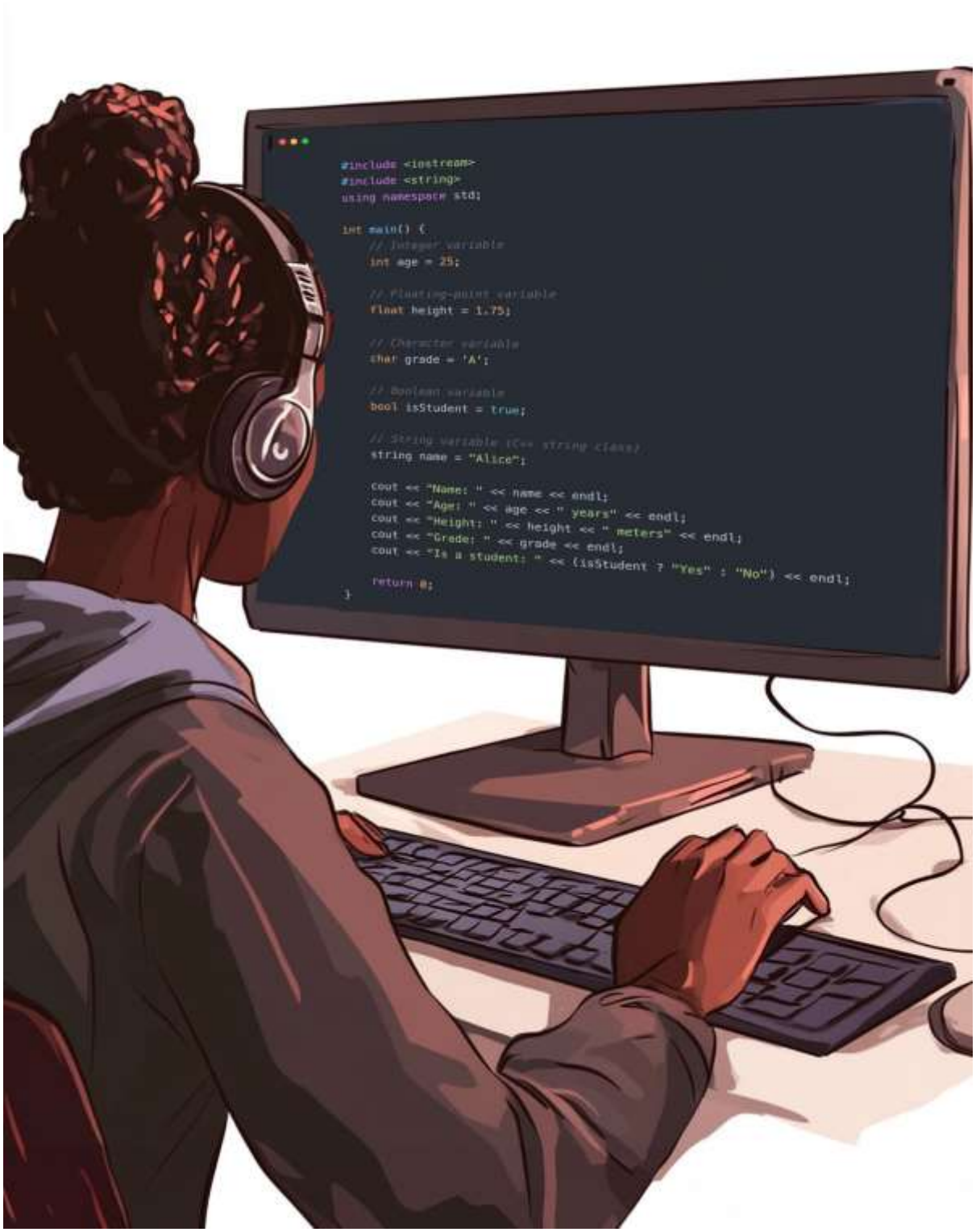
The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: GENFC401-FUNDAMENTALS OF C++ PROGRAMMING

Learning Outcome 1: Apply Basic C++ concepts
Learning Outcome 2: Apply OOP concepts
Learning Outcome 3: Perform CPU optimization


#include <iostream>
#include <string>
using namespace std;
int main() {
// Integer variable
int age = 25;
// Floating-point variable
float height = 1.75;
// Character variable
char grade = 'A';
// Boolean variable
bool isStudent = true;
// String variable (C++ string class)
string name = "Alice";
cout << "Name: " << name << endl;
cout << "Age: " << age << " years" << endl;
cout << "Height: " << height << " meters" << endl;
cout << "Grade: " << grade << endl;
cout << "Is a student: " << (isStudent ? "Yes" : "No") << endl;
return 0;
}

Indicative contents
1.1. Preparation of Development Environment **1.2 .Apply variables.** **1.3.Apply of Operators** **1.4. Apply control structure** **1.5.Apply function** **1.6.Apply Array**

Key Competencies for Learning Outcome 1: Apply basic C++ Concepts

Knowledge	Skills	Attitudes
• Description of C++ Development Environment • Description of C++ programming Tools • Description of IDE menus and Icons • Description of variable • Description of operators • Description of C++ control structures • Description of C++ function • Descriptions of C++ arrays	• Installing C++ programming tools (IDE, text editor) • Testing the Development environment • Executing C++ program • Applying Variable • Applying C++ operators in program • Applying control structures in C++ • Applying C++ function • Applying arrays in C++	• Being self-motivated • Being critical thinker • Being detailed oriented • Being skilful • Being creative

Duration:20 **hrs**

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe properly the C++ Environment based on programming standards.
2. Describe properly C++ programming tools based on their usage.
3. Install correctly C++ programming tools.
4. Test correctly the installed C++ Development environment.
5. Describe properly IDE Menus and Icons in C++
6. Execute correctly C++ program
7. Describe properly variable based on C++ programming standards.
8. Apply correctly variable in C++ program.
9. Describe properly operator and its types based on C++ standards and sementics defined.
10. Apply correctly operators in C++ program.
11. Describe correctly control structures based on C++ rules and syntax defined.
12. Apply correctly control structures based on C++ rules and syntax defined
13. Describe C++ function based on the task to be done.
14. Apply properly function in C++ program.
15. Describe properly arrays in C++ programming language.
16. Apply correctly arrays based on C++ program.

Resources

Equipment	**Tools**	**Materials**
• Computer • Storage device	• Integrated Development Environment (IDE)	• Internet

Indicative content 1.1: Preparation of Development Environment

Duration: 2 hrs

Theoretical Activity 1.1.1: Description of C++ programming language.

Tasks:

1: Answer the questions reflecting the description of C++ programming language;

I. What do you understand by:
   a) C++ programming language
   b) debugging in C++

II. Describe the following:
   a) C++ file extensions
   b) Features of C+ programming language
   c) Applications of C++ programming language

2: Note the essential of the key findings

3: Presentation of the findings

4: Asks questions for more clarifications or provide concerns

5: Read through key **readings 1.1.1**. in trainee manual.

Key readings 1.1.1.

Description of C++ programming language.

. Introduction to C++ programming language

1. **Definitions**

C++ language: is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a compiled language.C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

2. **C++ extensions**

Two most common C++ file extensions are **.cc** and **.cpp** both store C++ codes.

Note:

A file with a **.cc** extension is a C++ source code file. The .cc file extension is commonly used in Unix-based systems for C++ source files.

Difference Between .cc and .cpp File Extensions

The below table illustrates the primary differences between .cc and .cpp file extensions:

Feature	.cc File Extension	.cpp File Extension
Usage	It is generally used in Unix/Linux based system	It is commonly used in windows.
Typing Speed	It is faster to type due to fewer characters.	It takes slightly longer to type.
Compatibility	This file extension is supported by Unix, GNU C++, Clang, Microsoft Visual C++, and Metrowerks CodeWarrior.	This file extension is supported by GNU C++ , Clang, Digital Mars, Borland C++, Watcom, Microsoft Visual C++, and Metrowerks CodeWarrior.
External Factor	.cc extension is used by Google.	LLVM libc++ use the .cpp file

3. The features of C++ programming language

C++ programming language exerts the following features:

a. Object-Oriented Programming

C++ is an Object-Oriented Programming Language, unlike C which is a procedural programming language. This is the most important feature of C++. It can create/destroy objects while programming. Also, It can create blueprints with which objects can be created.

b. Machine Independent

A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however, they are machine-independent. Let us understand this feature of C++ with the help of an example. Suppose you have

written a piece of code that can run on Linux/Windows/Mac OSx which makes the C++ Machine Independent the executable file of the C++ cannot run on different operating systems.

c. Simple

It is a simple language in the sense that programs can be broken down into logical units and parts, has rich library support and has a variety of data types. Also, the Auto Keyword of C++ makes life easier.

d. High-Level Language

C++ is a High-Level Language, unlike C which is a Mid-Level Programming Language. It makes life easier to work in C++ as it is a high-level language it is closely associated with the human-comprehensible English language.

e. Popular

C++ can be the base language for many other programming languages that supports the feature of object-oriented programming. **Bjarne Stroustrup** found Simula 67, the first object-oriented language ever, lacking simulations, and decided to develop C++.

f. Case-sensitive

It is clear that C++ is a case-sensitive programming language. For example, **cin** is used to take input from the input stream. But if the **"Cin"** won't work. Other languages like HTML and MySQL are not case-sensitive languages.

g. Compiler Based

C++ is a compiler-based language, unlike Python. That is C++ programs used to be compiled and their executable file is used to run them. C++ is a relatively faster language than Java and Python.

h. Dynamic Memory Allocation

When the program executes in C++ then the variables are allocated the dynamical heap space. Inside the functions, the variables are allocated in the stack space. Many times, We are not aware in advance how much memory is needed to store particular information in a defined variable and the size of required memory can be determined at run time.

i. Memory Management

C++ allows us to allocate the memory of a variable or an array in run time. This is known as Dynamic Memory Allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, the memory must be de-allocated dynamically allocated memory manually after it is of no use. The allocation and deallocation of the memory can be done using the new and delete operators respectively.

j. Multi-threading

Multithreading is a specialized form of multitasking and multitasking is a feature that allows your system to execute two or more programs concurrently.
C++ doesn't contain any built-in support for multithreaded applications. Instead, it relies entirely upon the OS to supply this feature.

4. Application of C++

C++ is a versatile programming language with a wide range of applications across various domains. Here are some key areas where C++ is commonly used:

a. **Game Development**:

C++ is widely used in game development due to its high performance and ability to handle complex graphics and real-time simulations. Popular game engines like Unreal Engine are built using C++.

b. **GUI-Based Applications**:

Many graphical user interface (GUI) applications are developed using C++. Examples include Adobe Photoshop and Illustrator.

c. **Operating Systems**:

C++ is used in the development of operating systems due to its efficiency and control over system resources. Parts of Windows, macOS, and Linux are written in C++.

d. **Browsers**

Web browsers like Google Chrome and Mozilla Firefox use C++ for rendering engines and other performance-critical components.

e. **Database Software**:

C++ is used in the development of database management systems like MySQL and MongoDB

f. **Embedded Systems**:

C++ is used in embedded systems for its ability to provide low-level hardware control while maintaining high performance.

g. **Financial Systems**:

Many banking and financial applications use C++ for its reliability and performance in handling large-scale transactions and data processing.

h. **Cloud/Distributed Systems**:

C++ is used in cloud computing and distributed systems for its scalability and efficiency.

i. **Compilers**:

C++ is used to develop compilers for other programming languages, leveraging its speed and control over system resources.

j. **Libraries**

Many standard libraries and frameworks are written in C++ due to its robustness and performance.

Theoretical Activity 1.1.2: Description of C++ programming Tools

Tasks:

1: Read and answer the following questions

I. What do you understand by:

a) Text editor

b) interpreter

c) debugger

II. Describe the debugging process

III. Differentiate compiler from interpreter.

2: Write your key findings on papers /flipcharts

3: Present your findings to the whole class

4: Ask clarifications if any.

5: Read key **readings 1.1.2** in trainee manual

Key readings 1.1.2.

Description of C++ tools

. **Integrated Development Environments (IDEs)**

C++ supports the following IDEs:

DevC++: is an open-source Integrated Development Environment (IDE) designed for programming in C and C++. It provides tools for writing, compiling, and debugging C/C++ code using the MinGW (Minimalist GNU for Windows) compiler, which is based on the GNU Compiler Collection (GCC).

1. **Visual Studio**

   A comprehensive IDE for Windows, offering extensive features for C++ development, including debugging, code analysis, and more.

2. **CLion**

   A cross-platform IDE from Jet Brains, known for its powerful code analysis and refactoring tools

3. **Eclipse**

   An open-source IDE with a robust set of tools for C++ development

4. **CodeBlocks**

   A free, open-source IDE that is highly extensible and customizable

5. **NetBeans**

   Another open-source IDE that supports multiple languages, including C++

6. **Text Editors**

   Most popular text editors for C++ programming language are

7. **Visual Studio Code**

   A lightweight but powerful source code editor with support for C++ through extensions.

8. **Sublime Text**

   A versatile text editor with a wide range of plugins available for C++ development

9. **Atom**

An open-source text editor developed by GitHub, which can be customized for C++ development

I. Compilers

1. **GCC (GNU Compiler Collection)**

   A widely-used open-source compiler that supports C, C++, and other languages

2. **Clang**

   A compiler front end for the C, C++, and Objective-C programming languages, part of the LLVM project

3. **Microsoft Visual C++ Compiler**

   Part of the Visual Studio suite, offering robust support for C++ development on Windows

II. Difference between compiler and interpreter

Compiler

Translation Method: A compiler translates the entire source code into machine code before execution.

Execution Speed: Programs compiled into machine code generally run faster since the translation is done beforehand.

Error Detection: Errors are detected after the entire program is compiled, making debugging potentially more complex.

Output: Produces an executable file that can be run independently of the source code

Memory Usage: Requires more memory as it generates object code.

Compilers: Used by languages like C, C++, and Java

Interpreter

Translation Method: An interpreter translates code line by line during execution.

Execution Speed: Interpreted programs run slower because each line is translated on the fly.

Error Detection: Errors are detected immediately as each line is executed, which can make debugging easier.

Output: Does not produce an independent executable file; the source code is needed each time the program runs.

Memory Usage: Generally uses less memory since it does not generate object code.

Interpreters: Used by languages like Python, JavaScript, and Ruby

V. Debugging

Overview

Debugging in C++ is an essential part of the development process, as it helps identify and resolve errors (bugs) in the code. Debugging tools and techniques allow developers to trace the execution flow, inspect variables, and diagnose problems. Here are some common techniques and tools for debugging C++ programs:

Common Debugging Techniques

1. Manual Debugging:

Using Print Statements

The simplest and most straightforward method is inserting print statements (std::cout) to trace program execution and inspect variable values at different points in the code.

Example:

```
int x = 5;

std::cout << "Value of x: " << x << std::endl;
```

This method is quick but becomes cumbersome in large programs and doesn't offer much control over runtime behavior.

Assertions

Assertions are used to ensure that certain conditions hold true during runtime. If an assertion fails, the program will terminate and report the error.

Example:

```
#include <cassert>

int divide(int a, int b) {

   assert(b != 0 && "Division by zero!");

   return a / b; }
```

Use assertions to catch logic errors during development. They can be disabled in release builds by defining NDEBUG (#define NDEBUG). For larger projects, logging frameworks can be more useful than print statements. These frameworks

allow you to control the level of detail, format logs, and write output to files. Popular logging libraries include **spdlog** or **Boost.Log**.

Using a Debugger

A debugger is a tool that allows you to control the execution of your program, step through code, set breakpoints, inspect variables, and more.

- **Common Debuggers**

GDB (GNU Debugger): A popular debugger for C++ programs, especially on Linux and macOS. It allows stepping through code, setting breakpoints, examining stack traces, and more.

LLDB: Part of the LLVM project, it's another powerful debugger, particularly on macOS.

Visual Studio Debugger: On Windows, Visual Studio offers an integrated debugging environment for C++ with a user-friendly interface.

Xcode Debugger: For macOS developers, Xcode provides a built-in debugger that integrates with LLDB.

v. **Version Control**

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It is essential for collaborative software development, allowing multiple developers to work on the same project without overwriting each other's changes. Popular version control systems include Git, Subversion, and Mercurial.

Practical Activity 1.1.3: Installing C++ programming tools

Task:

1: As computer technician, you are requested to install Dev-C++ in your computer

2: Read installation steps from **key readings 1.1.3.**

3: Install Dev-C++ setup following installation steps demonstrated by the trainer and ask assistance whether needed.

4: Run installed setup to check if it is well installed.

Key readings 1.1.3: Installing C++ programming tools

Steps of installing Dev C++ IDE and setting Setup Environment Variable Path

1. **Download Dev-C++**

The below steps show how to download Dev C++ from its official website – Bloodshot.net.

Step 1: Open your web browser and go to the official Dev-C++ website using the link above.

Step 2: Click the "Download Original Dev-C++ 5" button. Your browser will prompt you to save the file. The download should start automatically after a few seconds.

The file should be stored in your default download location.

Installing Dev-C++

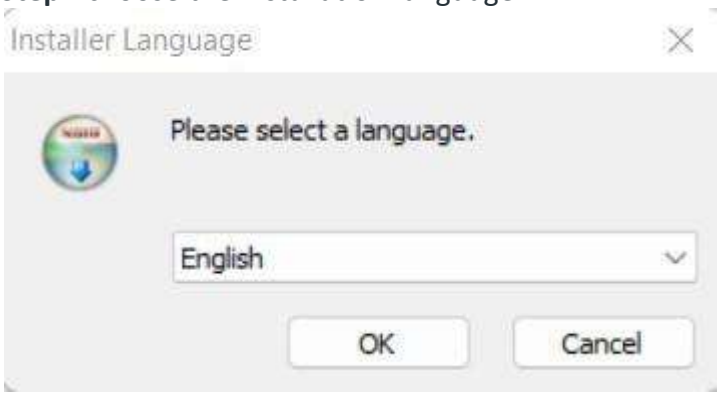
Once you have downloaded the setup, we can install the Dev-C++ using the following steps.

Step 1: Getting Started. Click the OK button to start the installation

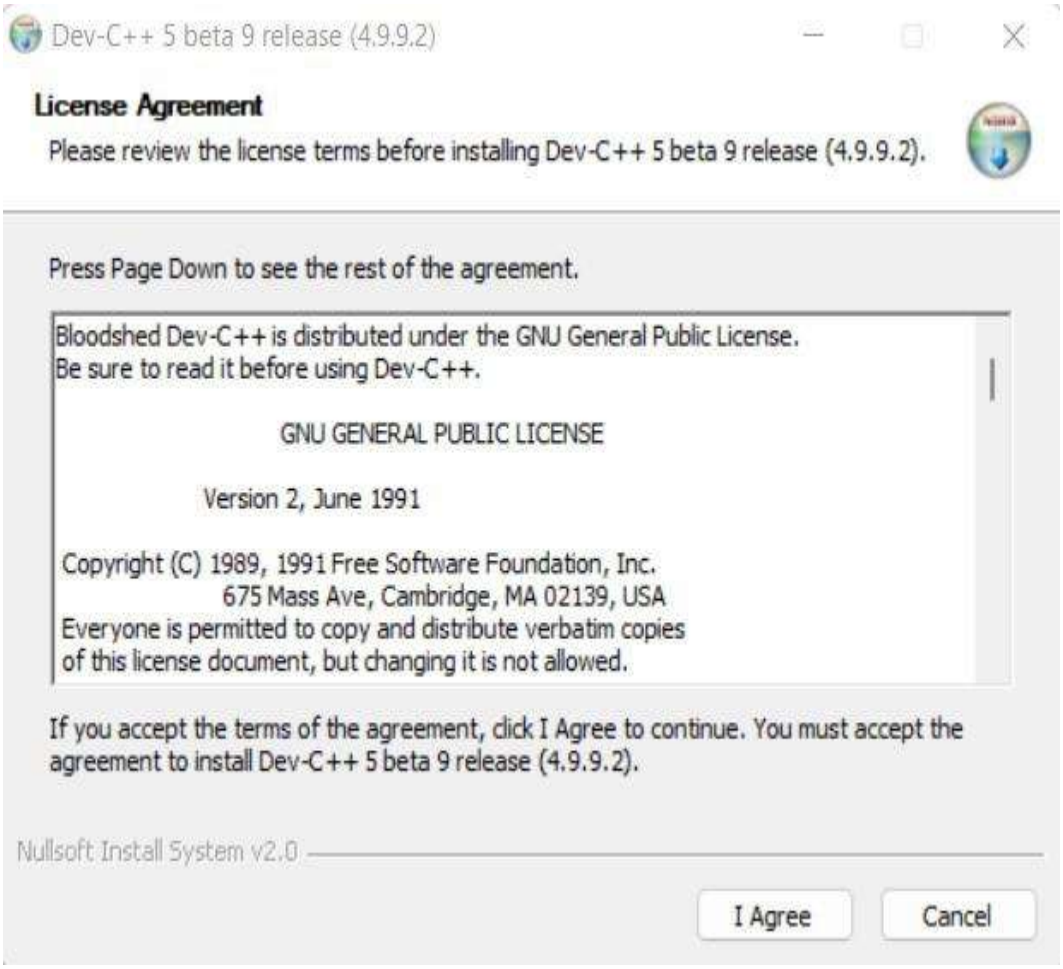
***Note**: Please make, there is no any other version of Dev-C++ is already installed. If any other version of Dev-C++ is already installed then we need to first uninstall it and then install the new version.*

Set up installation

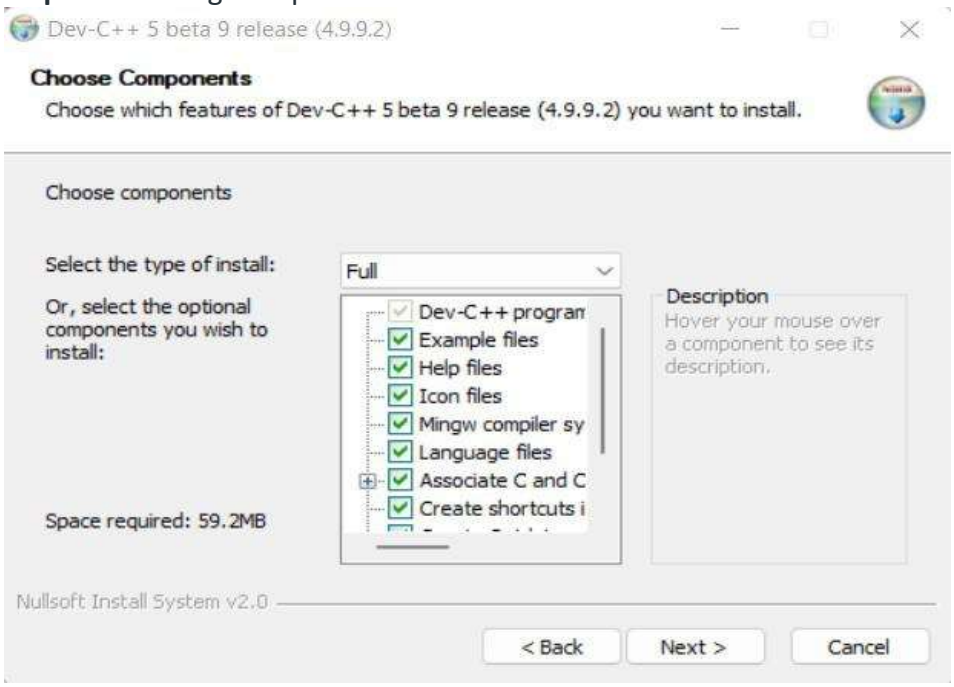
Step1:choose the installation language



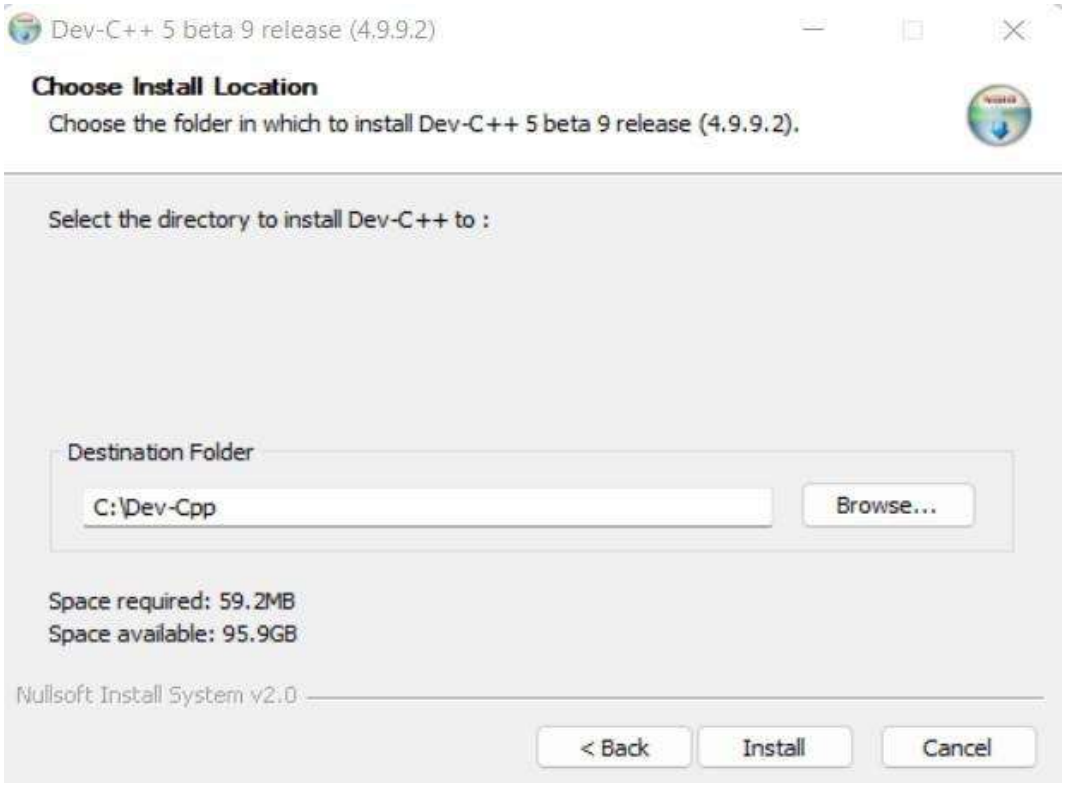
Step 2: Now, accept the License Terms & Agreement by clicking “I Agree”.



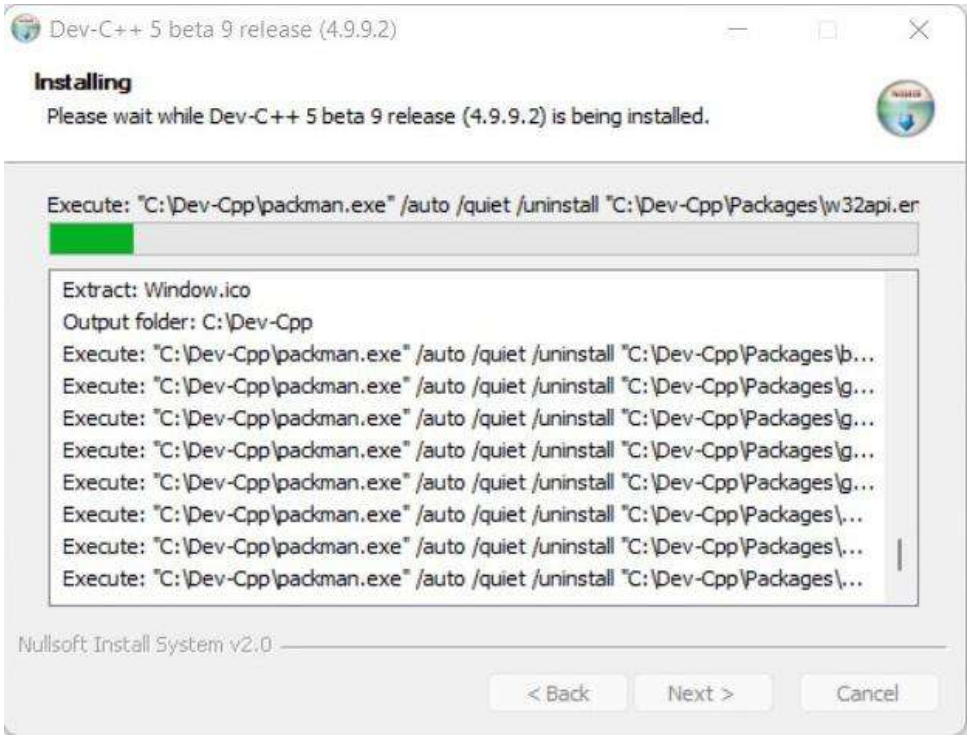
Step 3: Choosing Components To Be Installed.



Step 4: Specify the Location.

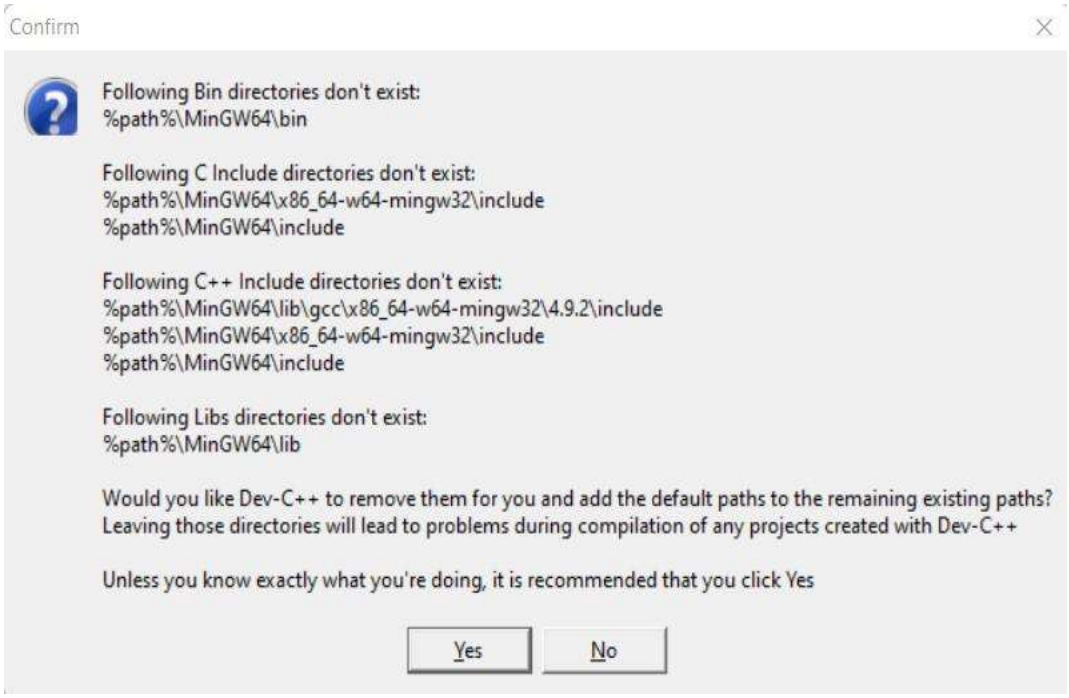


Step 5: Installing Libraries.



Step 6: Installation Completion.

Step 7: Define Path & Directories.



Note: *We need to click 'Yes', Dev-C++ will automatically create the bins for us.*
The following are steps to set the variable path for Dev-C++ on Windows:

1. **Open System Properties**:

- ✓ Right-click on "My Computer" or "This PC" on your desktop or in File Explorer.
- ✓ Select "Properties".
- ✓ Click on "Advanced system settings" on the left.
- ✓ In the System Properties window, click on the "Advanced" tab.
- ✓ Click on the "Environment Variables" button at the bottom.

2. **Edit the PATH Variable**:

✓ In the Environment Variables window, find the “Path” variable in the “System variables” section and select it.
✓ Click on “Edit”.

3. **Add Dev-C++ Path**:

✓ In the Edit Environment Variable window, click “New” and add the path to your
✓ Dev-C++bindirectory. This is usually **C:\Dev-Cpp\bin.**
✓ Click “OK” to close all windows.

4. **Verify the Path**:
✓ Open Command Prompt and type g++ --version to check if the path is set correctly.

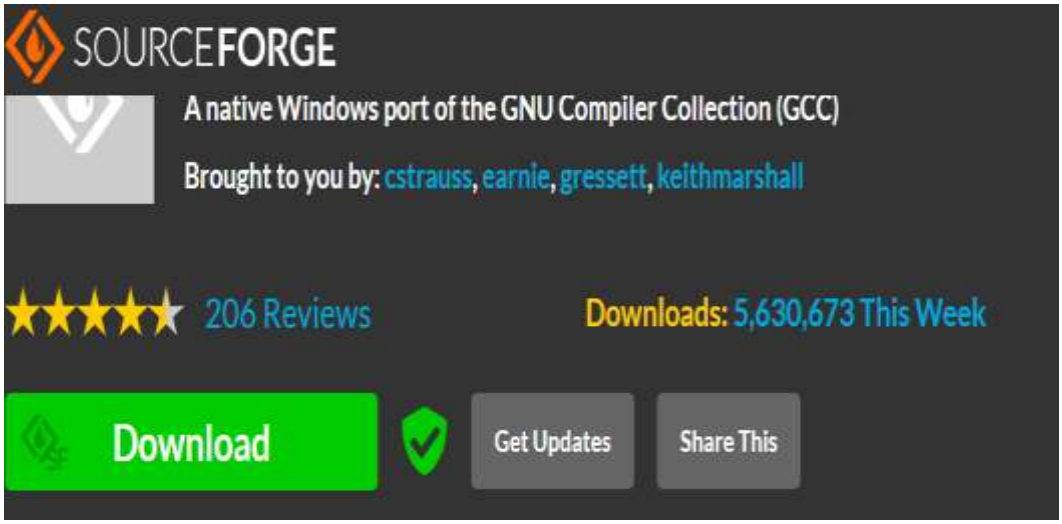
Compiler Installation

Steps to install compiler (MinGW) :

1. Open your browser and type **MinGW** and click on download

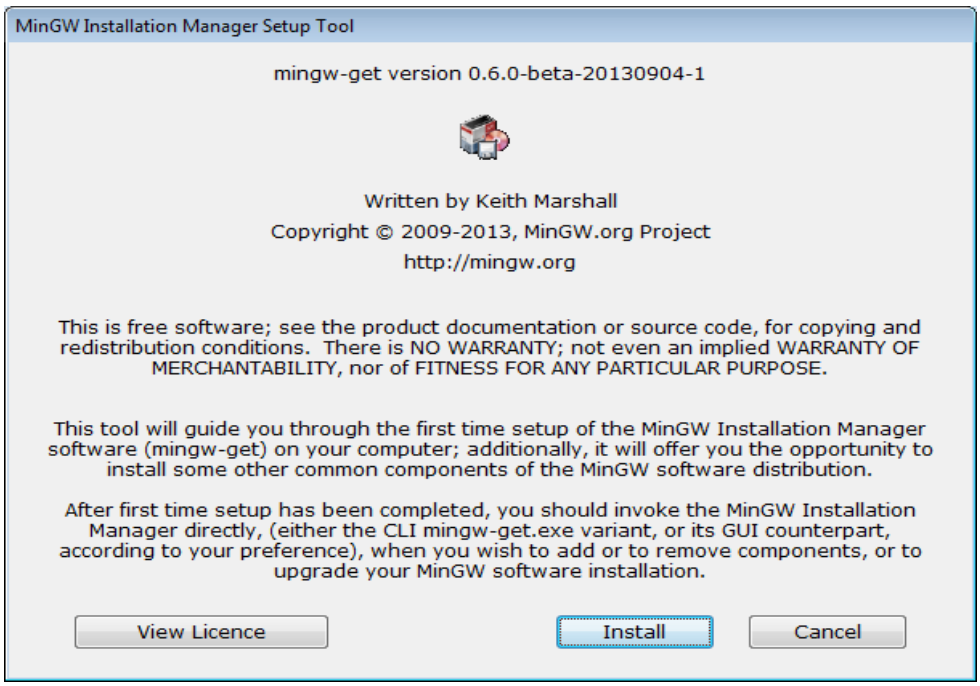
MinGW: A native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications; includes extensions to the MSVC runtime to support C99 functionality. All of MinGW's software will execute on the 64bit Windows platforms.

2. Click on download

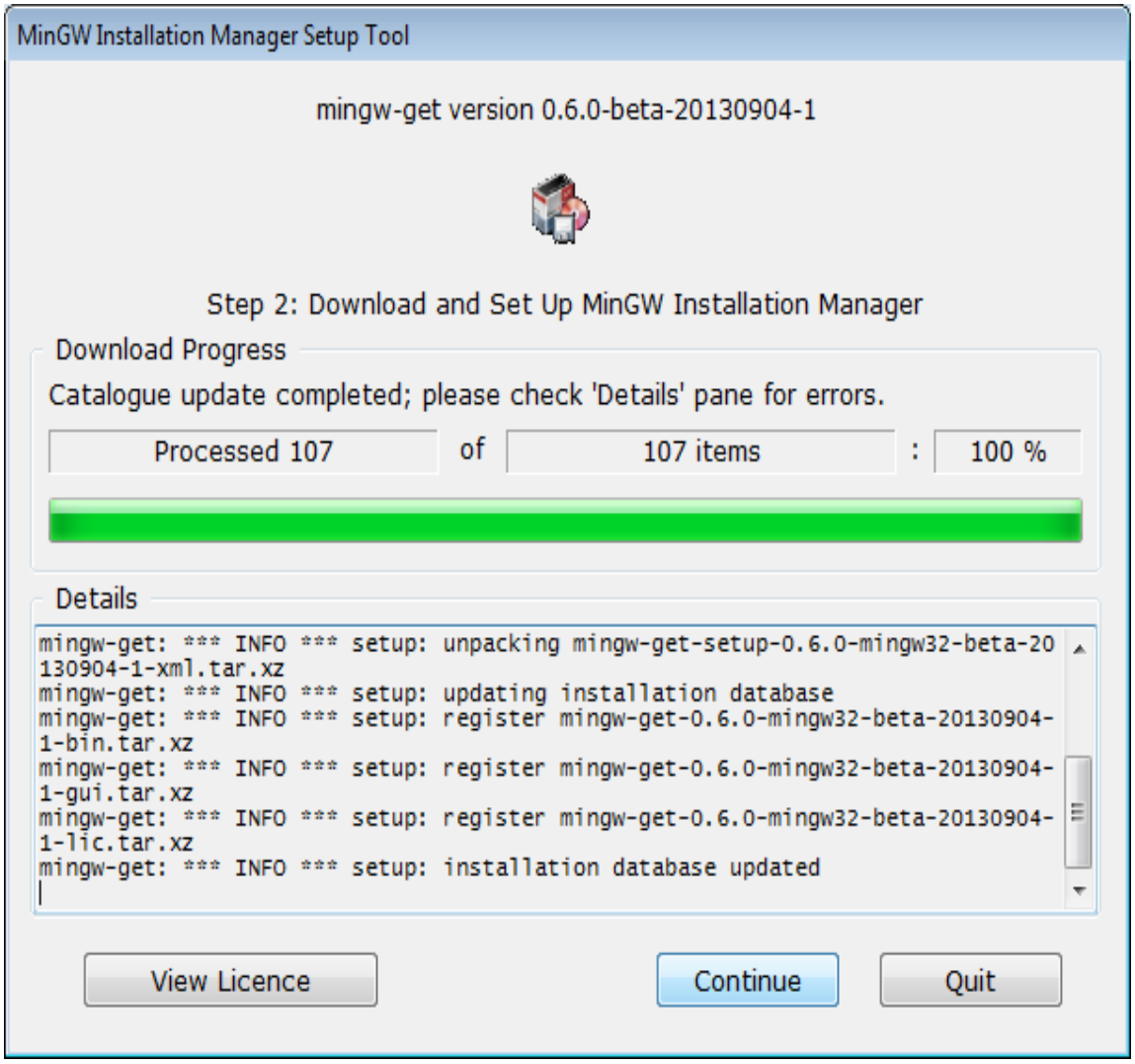


3. **Go in download and double click on its set up**

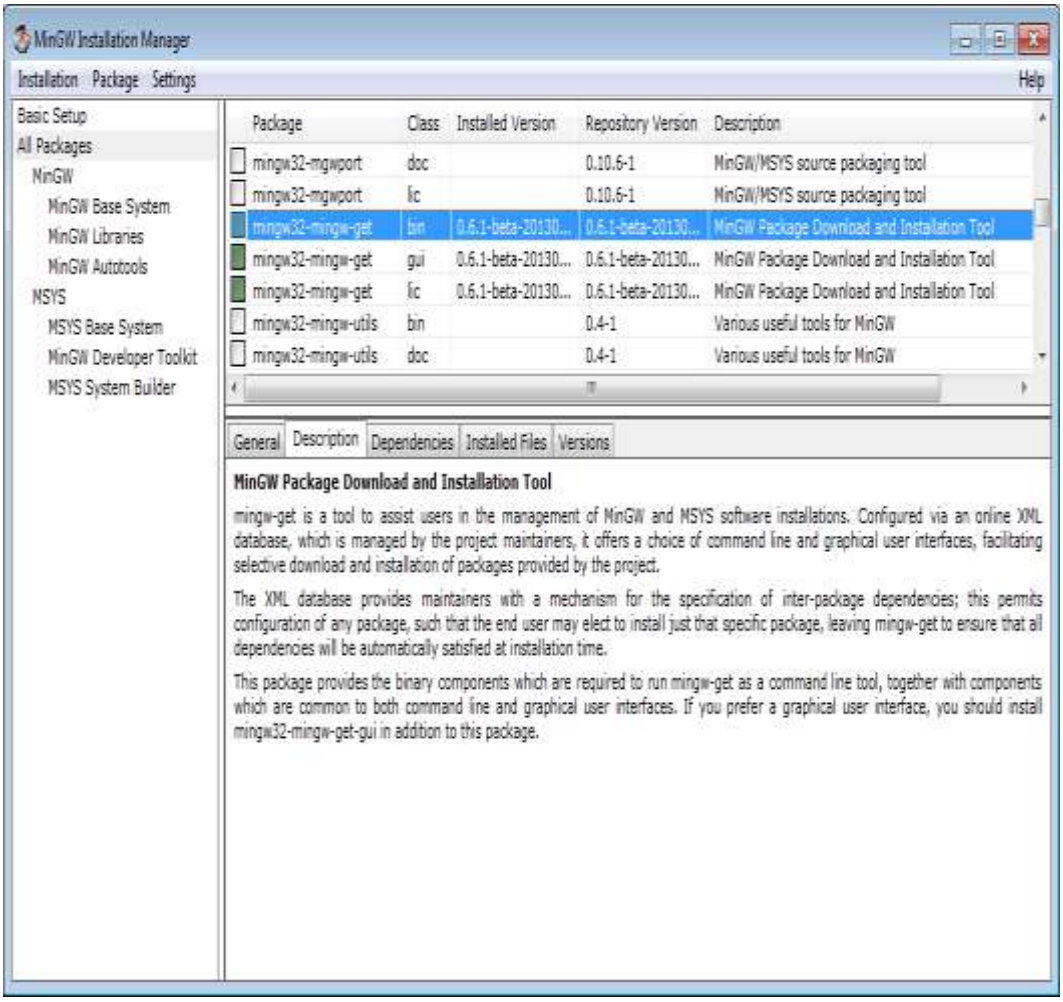
4. **Click on install**



5. **Click on continue**

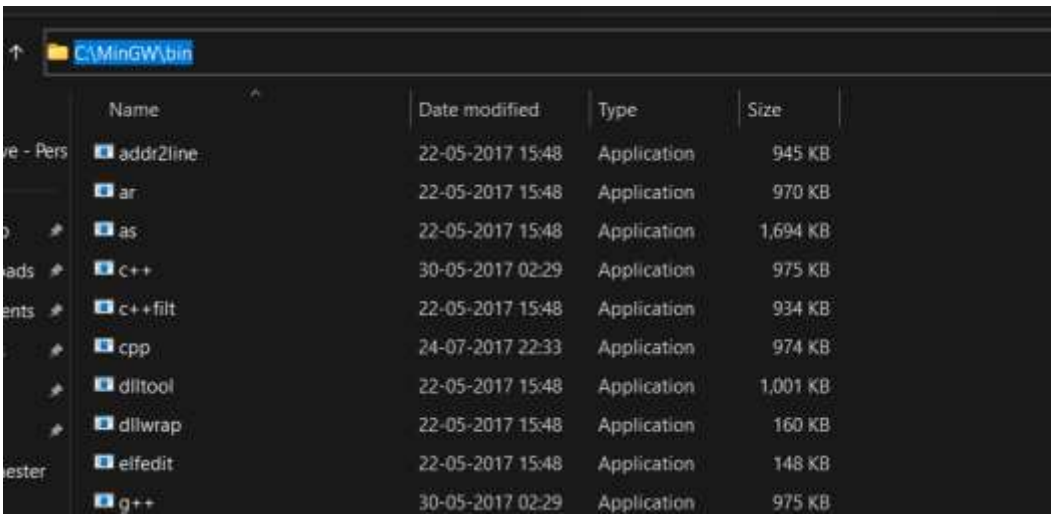


6. See MinGW Installer -- Package View

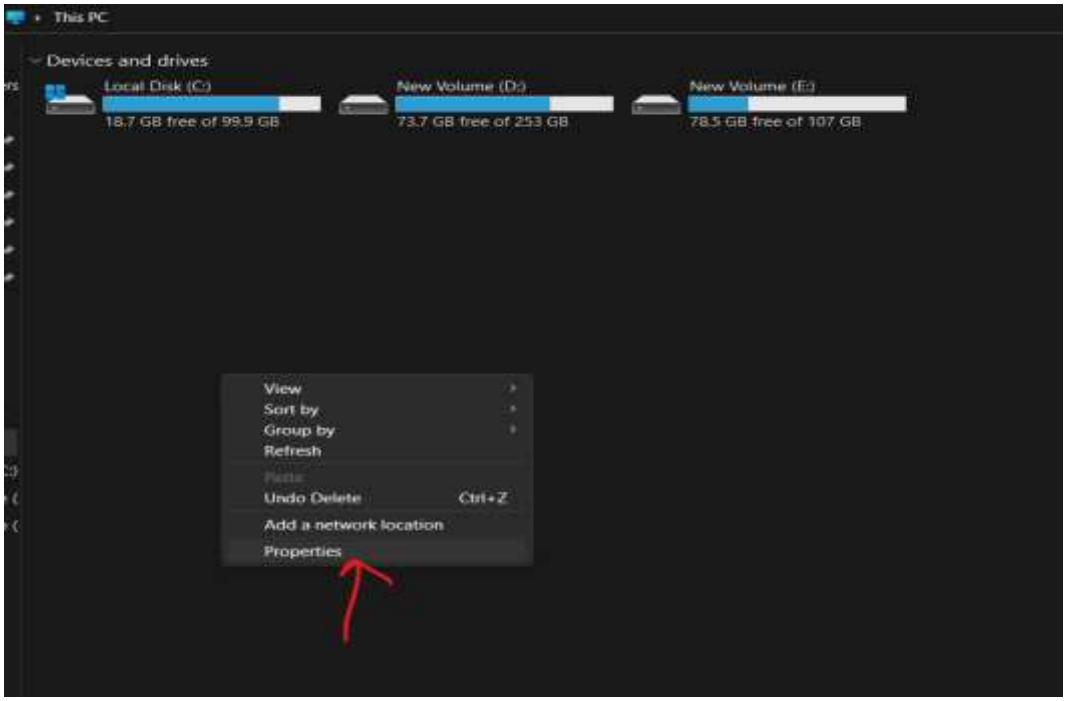


Another method to change Environment Variable for MinGw.

Step 1: Go to the C drive on your device and search for the MinGW folder. And in the MinGW folder go to the bin folder and copy its path.

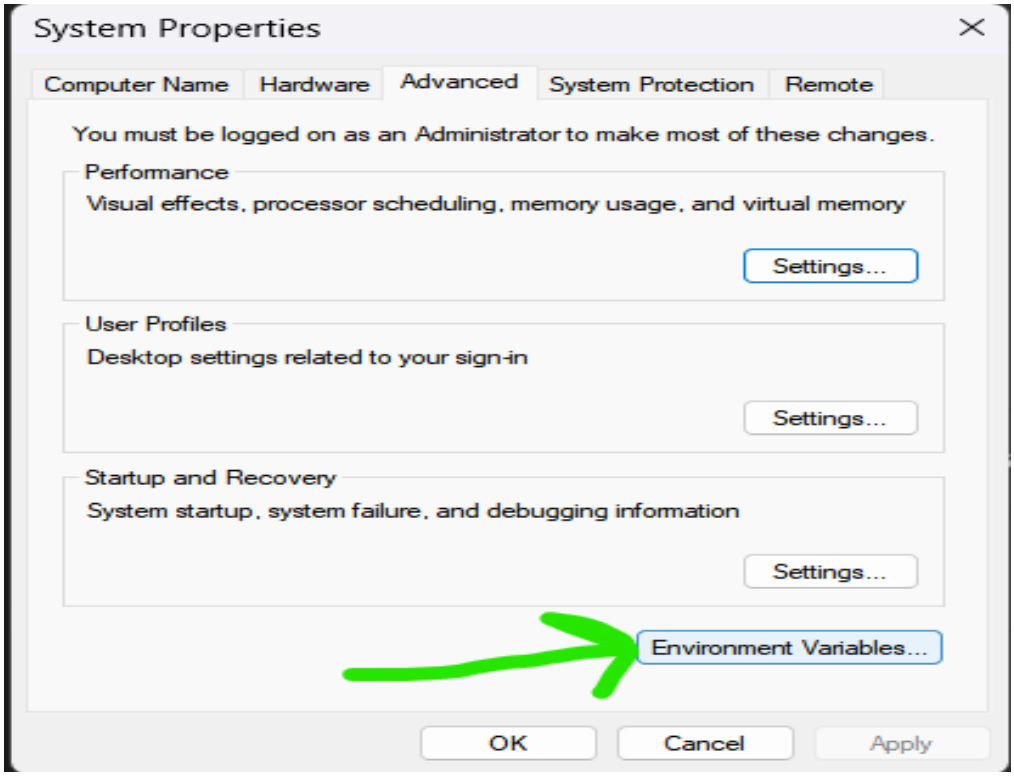


Step 2: Now go to this pc and right click there and click on show more properties and click on properties pointed by the arrow.

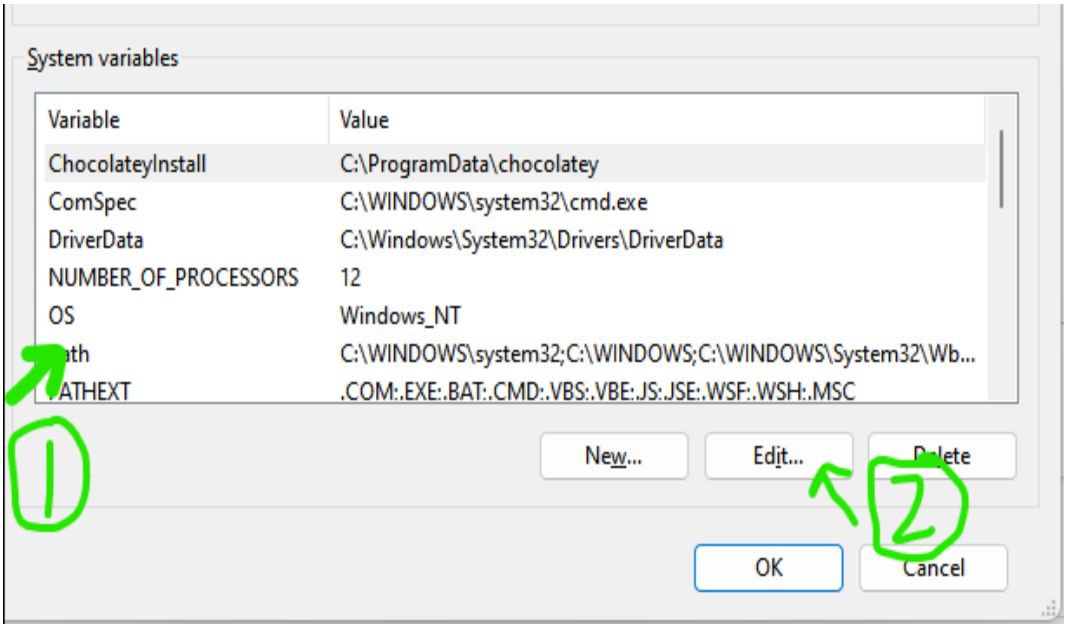


Step 3: Now click on Advanced system setting.

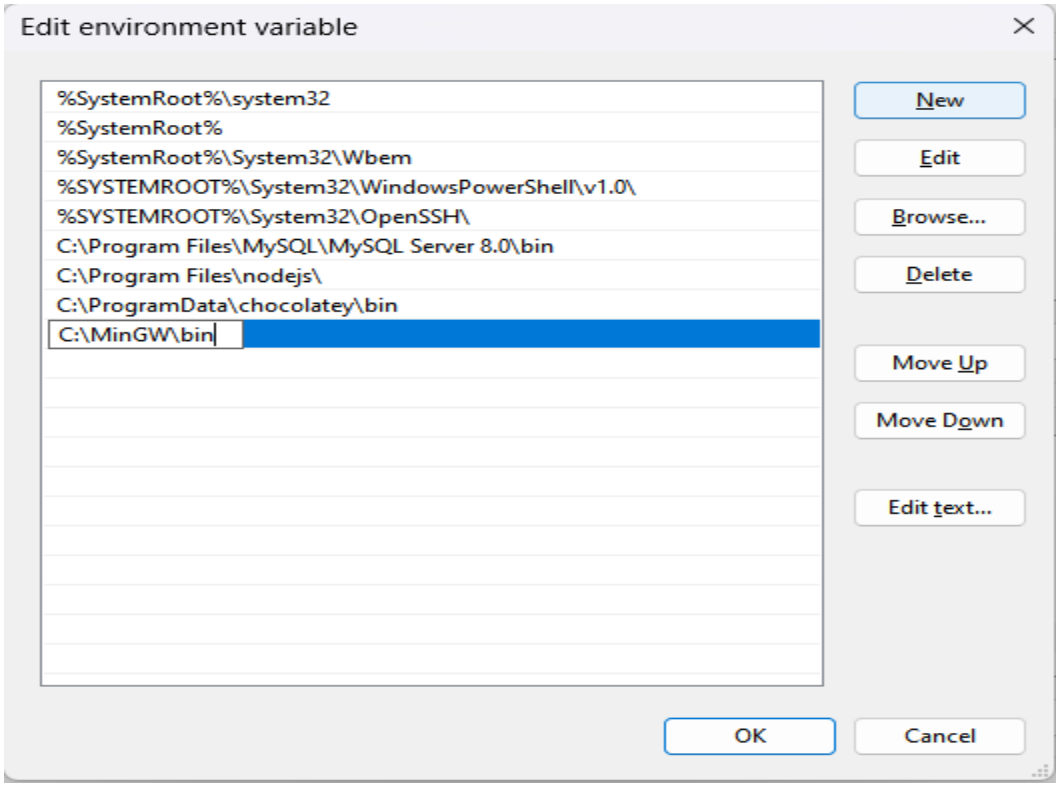
Step 4: Now click on environmental variables.



Step 5: Now in system variables click on path and click on edit button.



Step 6: Now click on new button and paste the copied path there as shown in photo.



Now click on “ok” button of step 6, 5 and 4 and then close all the windows and restart your pc. This marks the complete installation of MinGW in the system.

1. **Test the installed C++ IDE**

- **Description of IDE menus and Icons for C++(DevC++)**

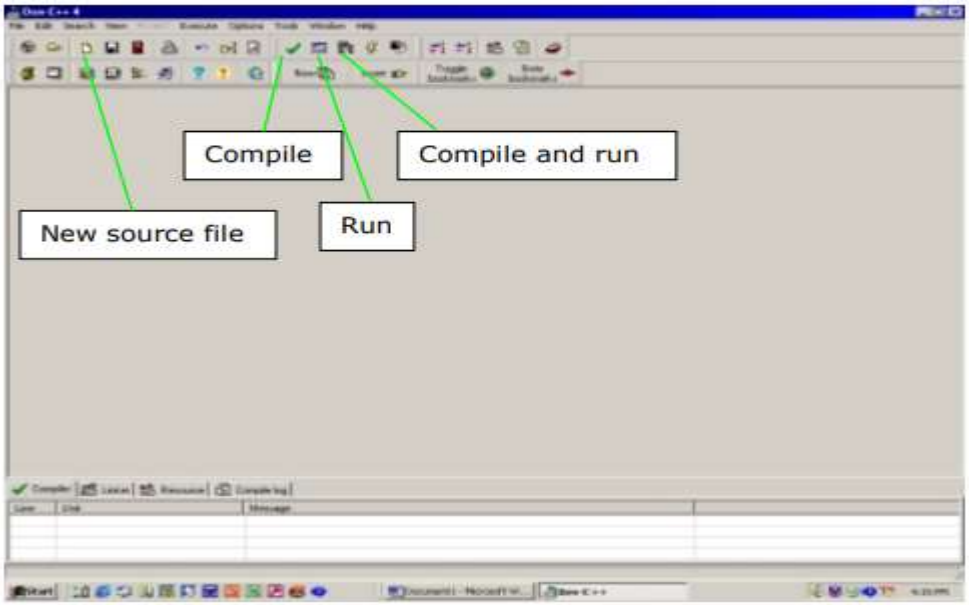
  Dev-C++ provides an integrated environment for writing programs. "Integrated environment" means Dev-C++ is a combination program, consisting of a **text editor** and a **C++ compiler**.

  A text editor is a limited word processing program that allows you to type in your program, make corrections and changes, and save and retrieve your program from disk storage.

  The editor is integrated with the Dev-C++ compiler so that you can easily switch between editing your program and compiling and running it.

- **Dev-C++ interface**

  When you click on the Dev-C++ icon on your desktop, the program window opens the following interface:



- **Navigate through Dev-C++ basics**

1. **To enter Dev-C++**

   Double-click the Dev-C++  icon.

   Dev-C++.exe

   Select File | New | **Source File**

   In the right hand lower window, type in and edit your program using the Dev-C++ editor.

2. **Save the file**

   Clicking on the  Save icon:

   When you save your file for the first time, you must specify three things: The drive name (nothing, if saving on the hard drive; d: or e: or something else if you have a flash drive)

   The name of the program itself, which should be something like prog1 or first or assn1

   The extension, which must be .cpp for a C++ program. For example, you might name your first assignment (saved on a flash drive) as follows: d:prog1.cpp

3. **Compile your program by selecting Execute**

   Click on the  icon above the edit window

4. **Execute your program by selecting Execute**

   Run or by clicking  on the icon

5. **Print your program by selecting File**

    Click the icon.

   In the window that pops up, remove any check mark in the box that says “Line numbers.” (With line numbers, your printed lines may run off the page.) Click OK.

6. **Exit from Dev-C++ by selecting File**

   Click  the exit tab or clicking on the Close icon at the top right corner of the Dev-C++ window

Practical Activity 1.1.4: Executing a C++ program

Task:

1. As technician in Computer system and achitecture run a C++ program to display the text “Hello World” to ensure your Development environment is working correctly
2. Read steps involved in running a C++ program found in Key readings **1.1.4**
3. Run C++ program described in task 1 and ask assistance if needed.
4. Verify the output if they match as the one expected.

Key readings 1.1.4

Executing a C++ program

Steps to execute a C++ program

1. **Open your Dev-C++:**

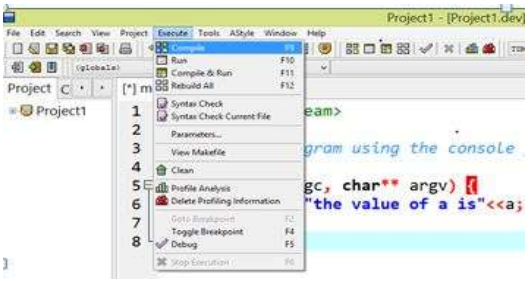
   Click the Dev-C++ icon on the desktop/TaskBar or find it on the start menu

2. **Write Your C++ Program:**

   with your open interface for Dev-C++ type in the following codes

```
#include <iostream>

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

3. **Compile the Program:**



- ✓ Go to execute tab on the menu bar
- ✓ In the pop up that appears choose compile

4. **Run the Program:**

- ✓ Simply click on the "Run" or "Build and Run" button from execute menu
- ✓ The IDE will compile and execute the program automatically.

5. **Check Output:**

   When the program runs successfully, the console or terminal will display:

   Hello, World!

Points to Remember

Description of C++ programming language

- **C++** is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- Debugging is the process of identifying errors and correct them
- The most common extensions in C++ are **.cc and .cpp** both store C++ codes.
- C++ programming language has many features including Object-Oriented programming, machine independent, simple to learn as well as multi-threading.
- **C**++ gets its applications in development of operating system, browser, embedded system, distributed system and libraries
- Development tools for C++ are : IDE ,text editors ,translator and debugger

Steps to install and test C++ tools.

- Main steps to install and test Dev C++ on windows system:
  - ✓ Have your Dev C++ setup.
  - ✓ Run the Installer
  - ✓ Choose Installation Language
  - ✓ Accept License Agreement
  - ✓ Choose Installation Location
  - ✓ Select Components
  - ✓ Create Start Menu Folder
  - ✓ Installation Process
  - ✓ Finish Installation
  - ✓ Setup the environment path
  - ✓ Test the installed C++ IDE

Application of learning 1.1

XYZ Co Ltd, a software development Company, has a college project that requires integration of C++ programs. As a technician, you are asked to provide help to XYZ Co Ltd by installing Dev C++ in their computers.

Indicative content 1.2: Apply variables

Duration: 3hrs

Theoretical Activity 1.2.1: Description of variable

Tasks:

1: Answer the questions reflecting to the definition of variable and data type

1. Describe the following terms:
    a. Variable
    b. Variable name/ identifier
    a. Reserved words /keyword

2: Write the findings on paper / flipchart

3: Present your findings to the whole class

4: Asks clarification if any.

5: Read through Key readings 1.2.1 in trainee manual

Key readings 1.2.1.

Description of variable and data type

Definition of key terms

A variable: is a place in memory that has a name and can hold data. It is a memory location where data are stored

Data type refers to the type of value a variable has. It serves to indicate kind of value to hold within a variable, size of the memory to store that value as well as type of operation to perform over a certain value

Identifier is the name of a variable

Key word/reserved word is the word that has a special meaning in a given programming language

Identification of Reserved Keywords

Rules for naming an identifier (variable naming rules)

- ✓ Variable name cannot be a reserved word
- ✓ Variable name Cannot be started with a number
- ✓ Variable name can't contain a space instead use underscore
- ✓ Variable name can be started with a letter or underscore
- ✓ Special characters(+ , -, /, *, &,% ,#) are not allowed

- ✓ A variable name is case sensitive

  Common reserved words/keywords in C++

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

  Some operators cannot be used as identifiers since they are reserved words under some circumstances: **and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq**

  Examples:

  Valid variable names(identifier):

- first
- conversion
- Pay_Rate
- counter1

  Invalid variable names(identifiers)

- Hello!
- One+two
- Pay rate
- 2ndyear

Theoretical Activity 1.2.2: Description of Data Types

Tasks:

1: Answer the questions reflecting to data type in C++:

1. What do you understand by data type in C++?
2. Discuss different categories of data types in C++
3. Outline range of data values for each data type in C++

2: Write the findings on paper / flipchart

3: Presentation of the findings

4: Asks questions for more clarifications or provide concerns

5: Read through Key readings 1.2.2 for additional clarifications

Key readings 1.2.2:

Description of C++ data types

. Introduction

In C++, a data type specifies the type of data that a variable can hold. It determines the size of the variable in memory, the values that can be stored, and the operations that can be performed on it. C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application.

I. Categories of data types

The following are data types supported by C++ language:

Integer: The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.

Character: Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.

Boolean: Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true* or *false*. The keyword used for the Boolean data type is **bool**.

Floating Point: Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.

Double Floating Point: Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.

void: Void means without any value. **void** data type represents a valueless entity. A void data type is used for those functions which do not return a value.

Array: A collection of elements of the same data type.

Pointer: A variable that stores the memory address of another variable.

Function: A function that returns a value of a specific data type.

Structure: A composite data type that groups variables of different data types under one name.

Class: A blueprint for creating objects with properties and behaviours.

Enumeration: A user-defined data type used for defining constants with meaningful names.

II. Classification of data types in C++

Data Types in C++ are mainly divided into 3 types:

a. **Primitive Data Types**:

These data types are built-in or predefined data types and can be used directly by the user to declare variables. Primitive data types available in C++ are:

- ✓ Integer
- ✓ Character
- ✓ Boolean
- ✓ Floating Point
- ✓ Double Floating Point
- ✓ Valueless or Void

b. **Derived Data Types:**

Derived data types that are derived from the primitive or built-in data types are referred to as Derived Data Types. These can be of four types namely:

- ✓ Function
- ✓ Array
- ✓ Pointer
- ✓ Reference

c. **Abstract or User-Defined Data Types**

Abstract or User-Defined data types are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined data types:

- ✓ Class
- ✓ Structure
- ✓ Union
- ✓ Enumeration
- ✓ Typedef defined Data type

Summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

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

The values of the columns Size and Range depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer types char, short, int and long must each one be at least as large as the one preceding it, with char being always 1 byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

Theoretical Activity 1.2.3: description of variable declaration

Tasks:

1: Answer the following questions reflect to declaration of variable:

1. What do you understand by
    a. Variable declaration?
    b. Variable initialization?
2. Identify the syntax to declare variables in C++
3. Outline various ways of initialization in C++
4. Discuss scope of variable in C++

2: Write the findings on paper / flipchart

3: Present your findings to the whole class.

4: Asks questions clarifications if any

5: Read through Key readings 1.2.3 in trainee manual.

Key readings 1.2.3:

Variable declaration

It refers to the process of reserving memory space to hold a value. It is the process of telling the compiler about the existence of the variable.

a. **How to declare a variable**

It involves specifying the **name** of the variable and the **data type** of the value it stores.

b. Syntax to declare a variable

To declare a variable in C++ use the following syntax:

data type identifier;

Where

- ✓ **Data type:** may be one of primitive data types: integer, float, string, Boolean…
- ✓ **Identifier:** is the name of the variable

Examples

- ✓ int MyNumber;
- ✓ string FirstName;
- ✓ bool Decision;

Variable initialization

Variable initialization is the process of assigning an initial value to a variable.

a. Ways to initialize variables in C++

C++ supports three (3) ways to initialize a variable depending on type of variable and version of C++ being used:

- ✓ **Copy initialization**

copy initialization also known as **c-like initialization** , consists of appending an equal sign followed by the value to which the variable is initialized

syntax:

VariableName=Value;

Or

data type VariableName=Value;

Examples

MyNumber=20; // initializing MyNumber variable to 20

char gender='M'; // initializes gender variable to 'M'

int Num1=87; // initializing Num1 variable to 87(inline initialization)

- ✓ **Direct initialization**

Direct initialization also known as **constructor initialization**, encloses the initial value between parentheses (()) and it is done during declaration.

Syntax

data type VariableName(Value);

Examples

1. int MyNumber(129);
2. char Gender('F');
3. float Average(34);

✓ **Uniform initialization**

Uniform initialization also known as list/brace initialization uses braces ({}) instead of parentheses. Introduced by C++11, has many advantages to narrowing conversion.

Syntax:

data type VariableName{Value};

Examples

1. int x{10}; // Uniform initialization of an integer
2. double y{3.14}; // Uniform initialization of a double
3. char z{'A'}; // Uniform initialization of a character
4. bool flag{true}; // Uniform initialization of a Boolean
5. const int a{100}; // Uniform initialization of a const variable
6. const double pi{3.14159}; // Uniform initialization of a const double

The Scope of variable

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, While a local variable is one declared within the body of a function or a block.

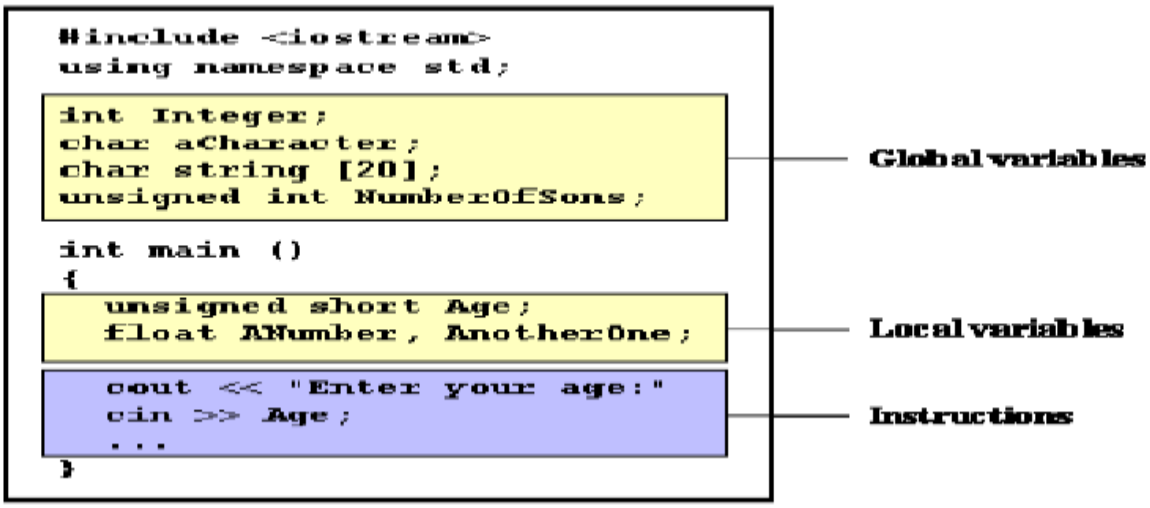


Figure 2: This picture illustrates the scope of variable where the variables like Age , ANumber and AnotherOne which declared inside main function are local variables and others like Integer ,aCharacter ,String and NumberOfsons which are declared outside of main function are called global variables.

Introduction to strings

Variables that can store non-numerical values that are longer than one single character are known as strings. C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: and have access to the std namespace (which we already had in all our previous programs thanks to the using namespace statement).

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

int main ()
{
  string mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

```
This is a string
```

As you may 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. Both initialization formats are valid with strings:

```
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 being assigned values during execution:

```
// 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;
}
```

```
This is the initial string content
This is a different string content
```

Constants

Constants are expressions with a fixed value.

✓ **Literals**

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

- **Integer Numerals**

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal

numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type int. However, we can force them to either be unsigned by appending the u character to it, or long by appending l:

```
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
```

In both cases, the suffix can be specified using either upper or lowercase letters.

- **Floating Point Numbers**

They express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is double. If you explicitly want to express a float or long double numerical literal, you can use the f or l suffixes respectively:

```
3.14159L   // long double
6.02e23f   // float
```

Any of the letters that can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters without any difference in their meanings.

- **Character and string literals**

  There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

  The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string (which generally consists of more than one character) we enclose it between double quotes ("). When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

  x alone would refer to a variable whose identifier is x, whereas 'x' (enclosed within single quotation marks) would refer to the character constant 'x'.

  Character and string literals have certain peculiarities, like the escape codes.

  These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line

```
"string expressed in \
two lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"this forms" "a single" "string" "of characters"
```

Finally, if we want the string literal to be explicitly made of wide characters (wchar_t), instead of narrow characters (char), we can precede the constant with the L prefix:

```
L"This is a wide character string"
```

Wide characters are used mainly to represent non-English or exotic character sets.

- **Boolean literals**

  There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

✓ **Defined constants (#define)**

You can define your own names for constants that you use very often without having to resort to memoryconsuming variables, simply by using the #define preprocessor directive.

Its format is:

#define identifier value;

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

  return 0;
}
```

```
31.4159
```

In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

- **Declared constants (const)**

  With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

  Here, pathwidth and tabulator are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

Practical Activity 1.2.4: Declaring and initializing variable

Task:

1: Read and perform the following task:
Write a C++ program that stores and display information about student marks in C++ programming fundamentals

2: Read Key readings 1.2.4. in trainee manual

3: Write, compile and execute C++ program described in task 1.

4: Verify the output if they match as expected.

Key readings 1.2.4:

Declaring and initializing variables

steps to run C++ program that stores and display information about student marks in C++ programming fundamentals:

1. **Open your Dev-C++:**

   Click the Dev-C++ icon on the desktop/TaskBar or find it on the start menu

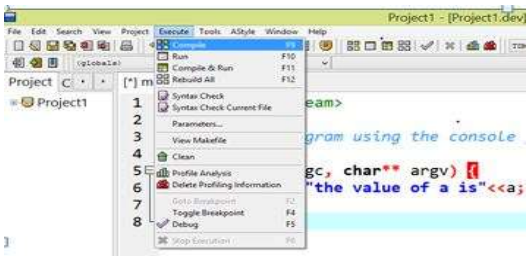
2. **Write Your Program**

   with your open5 interface for Dev-C++ type in the following codes:

   - include the <iostream> and <string> header file
   - define main function and
     - declare variables: StudentName, CourseName and Marks
     - initialize variables: StudentName, CourseName and Marks
   - close the main function

```
#include <iostream>

#include<string.h>

using namespace std;

/* run this program using the console pauser or add your own getch, system("pause") or input loop */

int main(int argc, char** argv) {
```

```
//variable declaration
string StudentName;
string CourseName;
int Marks;
// variable initialization
StudentName="MUKABIYI Emmanueline";
CourseName="C++ Programming";
Marks=90;
// displaying values of variables
cout<<"Student name:"<<StudentName<<endl;
cout<<"Course name:"<<CourseName<<endl;
cout<<"Marks obtained:"<<Marks;
                                    return 0;
}
```



3. **Compile the Program:**

✓ Go to execute tab on the menu bar

✓ In the pop up that appears choose compile

4. **Run the Program:**

✓ Simply click on the “Run” or “Build and Run” button from execute menu

✓ The IDE will compile and execute the program automatically.

5. **Check Output:**

When the program runs successfully, check if the output is as the following

```
Student name:MUKABIYI Emmanueline
Course name:C++ Programming
Marks obtained:90
--------------------------------
Process exited after 0.6655 seconds with return value 0
Press any key to continue . . .
```

Application of learning 1.2:

INEZA deposited **FRW 200000** in a Micro-finance company at an interest rate of 20% per annum. At the end of each year, the interest earned is added to the deposit and the new amount becomes the deposit for that year. Write a C++ program that would track the growth of deposits over a period of seven yeaars.

Indicative content 1.3: Apply operators

Duration: 3hrs

Theoretical Activity 1.3.1: Description of operators

Tasks:

1: Answer the questions reflecting to the definition of operator

1) What do you understand by the terms:
    a. Operator
    b. Operator overloading
    c. Precedence and associativity.
2) Discuss about the types of operators

2: Write your key findings on paper/flipchart

3: Presentation of the findings

4: Ask clarifications if any

5: Read through Key **readings 1.3.1** for additional clarification

Key readings 1.3.1.

Description of operator

1. **Definition**

An operator is a symbol that is applied on the operand in order to perform an operation. In C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

2. **Type of operators and their description**

a. **Assignment operator (=)**

The assignment operator assigns a value to a variable.

Example a = 5

This statement assigns the integer value **5** to the variable **a.**

Program example

```
// C++ program to illustrate the use of assignment operator
#include <iostream>
using namespace std;
int main()
{
   // Declare an integer variable
   int x;
   // Assign the value 20 to variable x using assignment
   // operator
   x = 20;
   cout << "The value of x is: " << x << endl;
   return 0;
}
```

b. **Arithmetic Operators:** These operators perform basic arithmetic operations.

- **Addition (+)**: Adds two operands.
- **Subtraction (-)**: Subtracts the second operand from the first.
- **Multiplication (*)**: Multiplies two operands.
- **Division (/)**: Divides the first operand by the second.
- **Modulo (%)**: Returns the remainder of an integer division.

The table below gives a summary of the five arithmetic operators supported in C++:

Operator	Name	Description	Example (A=10, B=20)
+	Addition	Adds two operands	A+B returns 30
-	Subtraction	Subtract right operand from left	A-B returns -10
*	Multiplication	Multiplies binary operands	A*B returns 200
/	Division	Divides numerator by denominator	B/A returns 2
%	Modulus	Gives remainder of integer division	B%A returns 0

c. **Relational Operators**: These operators compare two operands and return a boolean value.

There are six relational operators supported in C++: equals **(==),**less than **(<)**,greater than **(>)**, less than or equal to (<=), greater than or equal to (>=), and not equals (!=). Like arithmetic operators, , relational operators are also binary operators because they e.g. 5>3 to return true or false.

Table below shows summary of relational operator in their order of precedence from highest to lowest.

Operator	Name	Description	Example (A=10, B=20)
= =	Equal to	Checks two operands are equal, if yes it returns true.	A = =B; returns false
<	Less than	Checks if operand on left is less than that on the right.	A<B; returns true
>	Greater than	Checks if operand on left is greater than that on the right.	A>B; returns false
<=	Less than or equals to	Checks if operand on left is less than or equal to that on the right.	A<=B; returns true
>=	Greater than or equals to	Checks if operand on left is greater than or equal to that on the right.	A>=B; returns false
!=	Not equal to	Checks if operand on left is not equal to that on the right.	A!=B; returns true

d. **Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)**
When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

Table below gives a summary of the five self-assigned operators supported in C++:

Operator	Name	Description	Example (A=10, B=20)
+=	Conditional Addition	Adds to itself value on the right operator	A+=B; assigns A=30 (A=A+B; A=10+20)
-=	Conditional Subtraction	Subtract from itself value on the right of operator	A-=B; assigns A=-10 (A=A-B; A=10-20)
*=	Conditional Multiplication	Multiplies itself with value on the right of operator	A*=B assigns A=200 (A=A*B; A=10*20)
/=	Conditional Division	Divides itself by value on the right of operator	B/=A assigns A=2 (B=B/A; A=20/10)
%=	Conditional Modulus	Gives remainder of integer division	B%=A assigns B=0 (A=A%B; A=10%20)

e. Increment and decrement operators

In C++, increasing a value by 1 is referred to as incrementing while decreasing it by 1 is decrementing. C++ supports a unary (++) operator as a shortcut to incrementing a value by 1 and decrementing (--) by 1. Note that the term unary means that the operator takes only one operand. **For example**, the following statements increases and decreases value of count by 1 respectively:

count++; // equivalent to count=count+1.

Count--; // equivalent to count=count-1.

A characteristic of this operator is that it can be used both as **a prefix** and as **a suffix.**

That means that it can be written either before the variable identifier **(++a)** or after it **(a++).**

Although in simple expressions like **a++ or ++a** both have exactly the same meaning, in other expressions in which the result of the **increase** or **decrease** operation is evaluated as a value in an outer expression they may have an important difference in their meaning:

In the case that the increase operator is used as a prefix **(++a)** the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix **(a++)** the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Notice the difference:

Example 1	Example 2
B=3; A=++B; // A contains 4, B contains 4	B=3; A=B++; // A contains 3, B contains 4

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

f. Logical operators

In C++, there are three logical operators used to form complex relational conditions. These are: && (AND), || (OR), and ! (NOT) also called negation. Whereas the && and || operators are binary, ! is a unary operator that takes only one operand on its right. Consequently, the operator negates the value or expression on its right to return opposite Boolean value.

Table below gives a summary of the three operators. Operator Name Description Example **(A=10, B=20)**

Operator	Name	Description	Example (A=10, B=20)
&&	AND	Checks if two operands or expressions are true, if one is false it returns false.	A<5&& B>17; returns false
||	OR	Checks if one of the operand or expressions is true, if either is true it returns true.	A<5|| B>17; returns true
!	NOT	Unary operator that negates its operand or expression. If true, it returns false.	!(A>=B); returns true

g. Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false.

Its format is: **condition ? result1 : result2**

h. Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

<<	Bitwise left shift	The operator shifts the bits of an *expression* left by the number of bits specified.	If A=00001110 then *A<<2 returns 00111000*
>>	Bitwise right shift	The operator shifts the bits of an *expression* right by the number of bits specified.	If A=00111000 then A>>2 *returns* 00001110

i. Bitwise operators

Unlike other operators mostly used to manipulate decimal (base 10) numbers, bitwise operators are used to manipulate binary numbers.

The table below gives a summary of bitwise operators supported by C++ namely: AND (&) , inclusive OR (|), exclusive OR (^) one's complement (~),binary left shift << and binary right shift>>.

Bitwise operator	Name	Description	Example
&	Bitwise AND	Checks if both A and B are true to return true. If either or both are false, the expression returns false (0).	If A= 1, B=0 then A&B *returns 0*
|	Bitwise OR	Checks if either A or B is true to return true. If both are false, the expression returns false (0).	If A= 1, B=0 then A|B *returns 1*
^	Bitwise XOR	Checks if either A or B is true to return true. If both are true or false, the expression returns false (0).	If A= 0, B=1 then A^B *returns 1*
~	One's complement	Unary inversion of 0's to 1 and 1's to 0s in a binary number.	If A= 1, B=0 then ~A *returns 0, ~B returns 1*

h. Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was **(int).**

Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

i. sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to **a,** because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

3. Precedence of operators

Operator Precedence is a set of rules that defines the priority for all the operators present in a programming language. The operation which has an operator with the highest priority will be executed first.

Example: a= 4+5/10;

In the above given example if we solve the addition first then the answer will be **9/10=0.9** .
Now if we solve the division first then the answer will be **4 + 0.5 = 4.5 .**
Out of this the second answer is a valid answer because the precedence of division is more than addition.
Therefore **a=4.5**

4. Operator associativity

Associativity defines whether you should go from left to right or you should go to right to left. If associativity says that you should go left to right then

operation on the left side should be performed first and then only we should move to right side.

a= 5*5/10

If we solve the multiplication operation first then answer will be **25/10= 2.5**
If we solve the division operation first then the answer will be **5*0.5=2.5**
Out of this the first answer is correct because the given two operators have same precedence therefore we should look at the associativity which is left to right.

5. **The difference between precedence and associative operator is describe in the table below**

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	Left-to-Right
	[]	Array Subscript (Square Brackets)	
	.	Dot Operator	
	->	Structure Pointer Operator	
	++ , —	Postfix increment, decrement	
2	++ / —	Prefix increment, decrement	Right-to-Left

	+ / –	Unary plus, minus	
	! , ~	Logical NOT, Bitwise complement	
	(type)	Cast Operator	
	*	Dereference Operator	
	&	Addressof Operator	
	Sizeof	Determine size in bytes	
3	*,/,%	Multiplication , division, modulus	Left-to-Right
4	+/-	Addition, subtraction	Left-to-Right
5	<< , >>	Bitwise shift left, Bitwise shift right	Left-to-Right
6	< , <=	Relational less than, less than or equal to	Left-to-Right

	> , >=	Relational greater than, greater than or equal to	
7	== , !=	Relational is equal to, is not equal to	Left-to-Right
8	&	Bitwise AND	Left-to-Right
9	^	Bitwise exclusive OR	Left-to-Right
10	|	Bitwise inclusive OR	Left-to-Right
11	&&	Logical AND	Left-to-Right
12	||	Logical OR	Left-to-Right
13	?:	Ternary conditional	Right-to-Left
14	=	Assignment	Right-to-Left
	+= , -=	Addition, subtraction assignment	

	*= , /=	Multiplication , division assignment	
	%= , &=	Modulus, bitwise AND assignment	
	^= , |=	Bitwise exclusive, inclusive OR assignment	
	<<=, >>=	Bitwise shift left, right assignment	
15	,	comma (expression separator)	Left-to-Righ

Practical Activity 1.3.2: Applying operators

Task:

1: Read and perform the following task:

As a computer technician, Create a simple calculator program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) based on user input

2: Read the **key readings 1.3.2** in tranee manual.

3: Write, compile and run C++ program described in task 1.

4: Verify if the output match the one expected.

Key readings 1.3.2

Applying operators

To create and execute a C++ program that simulates simple calculator, follow these steps:

1. **Create a New C++ File**:

   Start a new file and save it with a .cpp extension (e.g., calculator.cpp).

2. **Write program codes:**

- Include Necessary Headers: At the top of your file, include the iostream library:

  #include <iostream>
- **Define the main Function**: Start the main function

  int main() {
- **Declare Variables**: Declare variables to store the operands, the result, and the operation:

  double num1, num2, result;

  char operation;
- **User Input**: Prompt the user to enter the first number, the operation, and the second number:
- **Perform Calculations**: perform the appropriate arithmetic operation based on user input:
- **Display the Result**: Output the result to the user:

3. **Compile and Run**:

- Compile your code and run the program to test its functionality.

4. **Test Different Operations**:

- Try different combinations of numbers and operations to ensure the calculator works as expected.

Points to Remember

Description of operators

- **Arithmetic Operators:** These perform basic mathematical operations. Examples include + (addition), - (subtraction), * (multiplication), / (division), and % (modulus).
- **Relational Operators:** These compare two values and return a boolean result. Examples are == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).
- **Logical Operators:** Used to perform logical operations. Examples include && (logical AND), || (logical OR), and ! (logical NOT).
- **Bitwise Operators**: These perform operations on the binary representations of numbers. Examples are & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), and >> (right shift).
- **Assignment Operators:** Used to assign values to variables. The basic assignment operator is =, but there are compound assignment operators like +=, -=, *=, /=, and %=.
- **Ternary Operator**: This is a short-hand for an if-else statement. It uses the syntax condition ? expression1 : expression2.
- **Operator Precedence and Associativity**: Operators have a defined precedence which determines the order in which operations are performed. For example, multiplication and division have higher precedence than addition and subtraction. Associativity determines the order of operations when operators have the same precedence, typically left-to-right or right-to-left

Applying operators:

- To create and run a C++ program that simulates simple calculator, follow these steps:

1. Create a new C++ file with a .cpp extension (e.g., calculator.cpp).
2. Include the iostream library at the top of the file.
3. Declare variables for operands, result, and operation type.
4. Prompt the user for input (two numbers and an operator).
5. Perform the selected arithmetic operation based on user input.
6. Compile and run the program, testing various operations for functionality.

Application of learning 1.3:

As computer technician write a program that will ask a user to enter two numbers and calculate its sum , product, difference, quotient ,remainder and tells the user if both numbers are equal and if Is the sum is positive and non-zero.

Indicative content 1.4: Apply control structure

Duration: 4hours

Theoretical Activity 1.4.1 Description of control structure

Tasks:

1: Read the following below and answer to the following question

1. Define the following terms
    a. Control Structures
    b. Iteration Logic (Repetitive Flow)
    c. Selection Logic (Conditional Flow)
    d. Conditional Statement
    e. Switch Case Statement
    f. Jump statement
2. Differentiate four types of jump statements
3. Describe the basic control structure key concepts

2: Write your findings on paper/flipchart

3: Present your findings to the whole class

4: 5: Ask for clarification if any

6: Read the **Key readings 1.4.1.:**

Key readings 1.4.1

Description of control structure

1. **Introduction to control structure**

In C++, program control structures are essential for directing the flow of execution in a program. They determine how and when certain parts of code are executed based on specific conditions or iterations. Here's a detailed breakdown of the primary control structures in C++:

2. **Sequential Control**

This is the default mode of execution where statements are executed one after another in the order they appear in the code. There are no special conditions or iterations involved in this structure.

Example

```
#include <iostream>

int main() {
   std::cout << "This is the first line.\n";
   std::cout << "This is the second line.\n";
   return 0;
}
```

3. Selection Control Structures

These structures allow the program to choose different paths of execution based on certain conditions.

a. If Statement

The **if statement** executes a block of code if a specified condition is true.

Syntax of if statement:

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

Example

```
int a = 5;
if (a > 0) {
   std::cout << "a is positive.\n";
}
```

b. If-Else Statement

The if-else statement provides an alternative path if the condition is false.

Syntax of if… else statement:

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

Else

{

// Code to execute if the condition is false

}

Example

```
int a = -5;
if (a > 0) {
   std::cout << "a is positive.\n";
} else {
   std::cout << "a is non-positive.\n";
}
```

c. if-Else-If else statement

This structure allows checking multiple conditions.

Syntax of if….elseif ….else statement:

```
if (condition1) {
   // Code to execute if condition1 is true
} else if (condition2) {
   // Code to execute if condition2 is true
} else {
   // Code to execute if none of the conditions are true
}
```

Example

```
int a = 0;
if (a > 0) {
   std::cout << "a is positive.\n";
} else if (a < 0) {
   std::cout << "a is negative.\n";
} else {
   std::cout << "a is zero.\n";
```

```
}
```

d. A nested if

A **nested if** statement in C++ is an if statement placed inside another if statement. This allows you to check multiple conditions in a hierarchical manner.

Syntax of nested if statement:

```
// if base_condition is true control goes to base_condition1
if ( base_condition)
    {
   // if base_condition is true control goes to base_condition2
   if(base_condition1)
       {
      if(base_condition2)
         .....
      .....
   }
}
```

Here's a simple example to illustrate:

```
#include <iostream>
using namespace std;
int main() {
   int number;
   cout << "Enter an integer: ";
   cin >> number;
   if (number > 0) { // outer if statement
      if (number % 2 == 0) { // nested if statement
         cout << "The number is positive and even." << endl;
      } else {
         cout << "The number is positive and odd." << endl;
```

```
        }
    } else {
        cout << "The number is not positive." << endl;
    }
    return 0;
}
```

In this above example, the program checks if number is positive and even , checks if number is positive and odd and also check if the number is not positive then it executes the corresponding block of code.

4. Switch Statement

The switch statement is used for selecting one of many code blocks to execute, based on the value of a variable.

a. Syntax of Switch … case statement:

```
switch (expression) {
    case constant1:
        // Code to be executed if expression equals constant1
        break; // Optional, but usually used to exit the switch
    case constant2:
        // Code to be executed if expression equals constant2
        break;
    // You can have any number of case statements.
    default:
        // Code to be executed if expression doesn't match any case
        break; // Optional
}
```

Example:

```
int day = 3;
switch (day) {
    case 1:
        std::cout << "Monday\n";
```

```
        break;
    case 2:
        std::cout << "Tuesday\n";
        break;
    case 3:
        std::cout << "Wednesday\n";
        break;
    default:
        std::cout << "Invalid day\n";
}
```

b. Rules for switch statements

Here are the key rules for using switch statements in programming languages like C++

- ✓ **Expression Evaluation**:

  The switch statement evaluates an expression once and compares its value against multiple case labels.

- ✓ **Case Labels**:

  Each case label must be a constant and unique. They represent the possible values the expression can take.

- ✓ **Colon After Case Labels**:

  Each case label must end with a colon (:).

- ✓ **Break Statement**:

  Typically, each case block ends with a break statement to prevent fall-through to the next case. If break is omitted, the program continues to execute the subsequent case blocks.

- ✓ **Default Case**:

  A default label can be included to handle any values not explicitly handled by the case labels. There can be only one default label.

- ✓ **Nesting**:

  Switch statements can be nested within each other.

- ✓ **No Overlapping Cases**:

Case labels must not overlap; each value should be handled by only one case.

c. **Difference between switch and multiple if statements**

Here are the differences between switch and if statements:

- ✓ **Expression Type**:
- **Multiple if Statement:** Can evaluate a variety of expressions, including integers, characters floating-point numbers, and boolean values.
- **switch Statement**: Typically evaluates only integer and character expressions.
- ✓ **Condition Testing**:
- **Multiple if Statement:** Can test for a range of conditions, including equality, inequality, and other logical expressions**.**
- **switch Statement: Tests only for equality against constant values.**
- ✓ **Syntax and Readability**:
- **Multiple if Statement:** Can become complex and harder to read with multiple nested conditions.
- **switch Statement:** Generally easier to read and manage when dealing with multiple discrete values.
- ✓ **Execution Flow:**
- **Multiple if Statement:** Executes the block of code associated with the first true condition. If no conditions are true, it can execute an optional else block.
- **switch Statement:** Executes the block of code associated with the matching case label and continues until a break statement is encountered or the end of the switch block is reached.
- ✓ **Default Handling**:
- **Multiple if Statement**: Uses an else block to handle cases where none of the conditions are true.
- **Switch Statement:** Uses a default case to handle values not explicitly matched by any case labels.
- ✓ **Performance**:
- **Multiple if Statement** May be slower for a large number of conditions due to sequential evaluation.
- **Switch Statement**: Can be more efficient for a large number of discrete values due to jump table optimization in some compilers

5. Iteration control structures

Iteration control structures are essential in programming for performing repetitive tasks efficiently.

Let's discuss the main types of iteration control structures in C++ namely for loop, while loop, do…while loop as well as nested loop:

a. for Loop

The **for… loop** is used when the number of iterations is known beforehand. It consists of an initialization, a condition, and an increment/decrement operation. Its format is:

```
for (initialization; condition; increase)
{
Statement
}
```

Example:

```
for (int i = 0; i < 5; i++) {
   std::cout << "Iteration " << i << std::endl;
}
```

In this example, the loop runs five times, printing the iteration number each time.

b. **while Loop**

The while loop continues to execute a block of code as long as the specified condition is true. It is useful when the number of iterations is not known in advance.

Example:

```
int i = 0;
while (i < 5) {
   std::cout << "Iteration " << i << std::endl;
   i++;
}
```

Here, the loop runs until i is no longer less than 5.

c. do-while Loop

The do-while loop is similar to the while loop, but it guarantees that the block of code is executed at least once, as the condition is checked after the execution. Its format is:

```
do
{
Statement
```

```
}
 while (condition);
```

Example:

```
int i = 0;
do {
   std::cout << "Iteration " << i << std::endl;
   i++;
}
 while (i < 5);
```

In this example, the loop runs at least once and continues as long as i is less than 5.

d. Nested Loops

Nested loops are loops within loops. They are useful for working with multi-dimensional data structures, such as matrices or tables.

Example:

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      std::cout << "i = " << i << ", j = " << j << std::endl;
   }
}
```

In this example, the outer loop runs three times, and for each iteration of the outer loop, the inner loop runs three times. This results in a total of nine iterations, printing the values of i and j for each combination.

e. **Use Cases of loops:**

for Loop: Iterating over arrays or collections where the number of elements is known.

while Loop: Reading data from a file until the end of the file is reached.

do-while Loop: Displaying a menu to the user at least once and repeating until the user chooses to exit.

Nested Loops: Processing elements in a 2D array or matrix, such as performing matrix multiplication.

f. Jump statements:

✓ **The break statement**

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

✓ **The continue statement**

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration

✓ **The goto statement:**

go to allows to make an absolute jump to another point in the program.

You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement.

A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it.

Practical Activity 1.4.2: Applying control structure

Task:

1: Read and and do the task below:

In athletics, runners are awarded medals depending on position as follows: position 1: gold; position 2: silver and position 3: bronze. The rest of the runners are not awarded any medal but receives appreciation message saying "Thank you for your participation". Using nested if...else and switch statements, write a C++ program that determines the medal to be awarded to runners depending on time each athlete touches the finish line

2: Read the **key readings 1.4.2** in the trainee manual.

3: Write, compile and run C++ program described in task 1.

4: Verify the output if it is the same as the one expected

Key readings 1.4.2

Applying control structures

The following are steps to create and run a C++ program that that awards medals to runners based on their finish positions

Step 1: Create a New C++ File

Open your IDE and create a new project or a new file.

Step 2: Write the C++ Code

Type in the the following code into your file:

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::sort
#include <iomanip>   // For std::setprecision
using namespace std;
int main() {
```

```
    int numRunners;
    cout << "Enter the number of runners: ";
    cin >> numRunners;
    vector<double> finishTimes(numRunners);
    cout << "Enter the finish times of the runners (in seconds):" << endl;
    for (int i = 0; i < numRunners; i++) {
        cin >> finishTimes[i];
    }
    // Create a copy to sort the times and get positions
    vector<double> sortedTimes = finishTimes;
    sort(sortedTimes.begin(), sortedTimes.end());
    // Determine the medal positions
    for (int i = 0; i < numRunners; i++) {
        // Find the position of the current runner's time in the sorted list
        int position = find(sortedTimes.begin(), sortedTimes.end(), finishTimes[i]) -
sortedTimes.begin() + 1;
        // Using nested if...else to determine the medal
        if (position == 1) {
            cout << "Runner " << (i + 1) << " (Time: " << setprecision(2) << fixed <<
finishTimes[i] << " seconds): Gold Medal" << endl;
        } else if (position == 2) {
            cout << "Runner " << (i + 1) << " (Time: " << setprecision(2) << fixed <<
finishTimes[i] << " seconds): Silver Medal" << endl;
        } else if (position == 3) {
            cout << "Runner " << (i + 1) << " (Time: " << setprecision(2) << fixed <<
finishTimes[i] << " seconds): Bronze Medal" << endl;
        } else {
            cout << "Runner " << (i + 1) << " (Time: " << setprecision(2) << fixed <<
finishTimes[i] << " seconds): Thank you for your participation" << endl;
```

```
        }


        // Alternatively, you can implement the same logic using a switch statement
        /*
        switch (position) {
            case 1:
                cout << "Runner " << (i + 1) << " (Time: " << finishTimes[i] << " seconds): Gold Medal" << endl;
                break;
            case 2:
                cout << "Runner " << (i + 1) << " (Time: " << finishTimes[i] << " seconds): Silver Medal" << endl;
                break;
            case 3:
                cout << "Runner " << (i + 1) << " (Time: " << finishTimes[i] << " seconds): Bronze Medal" << endl;
                break;
            default:
                cout << "Runner " << (i + 1) << " (Time: " << finishTimes[i] << " seconds): Thank you for your participation" << endl;
                break;
        }
        */
    }
    return 0;
}
```

Step 3: **Compile the and Run the Program**

Step 4: **Observe the Output**

The program will display the medal awarded (gold, silver, bronze) or a thank-you message for each runner based on their finishing position.

Points to Remember

Description of control structure

While using any control structure in C++ focus on the following:

- **Proper Syntax**: Ensure you use the correct syntax for each control structure. For example, always use curly braces {} to define the scope of if, else, for, while, and do-while blocks, even if they contain only one statement. This helps avoid errors and improves readability.
- **Indentation and Formatting**: Consistently indent your code to make it more readable. Proper formatting helps you and others understand the flow of the program.
- **Condition Evaluation**: Be mindful of the conditions you write. Ensure they are logical and correctly evaluate to true or false. For example, use == for comparison and = for assignment.
- **Avoid Deep Nesting**: Try to avoid deeply nested control structures as they can make your code hard to read and maintain. Consider breaking complex logic into functions.
- **Use `switch` for Multiple Conditions**: When you have multiple conditions based on the same variable, consider using a switch statement instead of multiple if-else statements. This can make your code cleaner and more efficient.
- **Break and Continue**: Use break to exit loops early and continue to skip the current iteration and proceed to the next one. These can help control the flow within loops effectively.
- **Default Case in `switch`**: Always include a default case in a switch statement to handle unexpected values. This ensures your program can handle all possible inputs gracefully.
- **Error Handling**: Implement error handling within your control structures to manage unexpected inputs or conditions. This can prevent your program from crashing and provide useful feedback to users.
- **Logical Operators**: Use logical operators (&&, ||, !) correctly to combine multiple conditions. Ensure you understand the precedence and associativity of these operators to avoid logical errors.
- **Exit Points**: Be aware of where your control structures exit. For example, in nested loops, a break statement will only exit the innermost loop

Application of learning 1.4:

SANUKI is a cooperative BANK that needs to include technology in their daily working in order to help clients to make transaction ,As CSA Technician Create a simple ATM simulation program that allows a user to: Check their balance ,Deposit money ,Withdraw money and Exit the program where the minimum balance for a client to withdraw should be RwF 1000.

Indicative content 1.5: Apply function

Duration: 4 hrs

Theoretical Activity 1.5.1: Description of functions

Tasks:

1: Answer the questions below:

- ✓ i. Describe the following :
  a. Function
  b. Function definition
  c. Function declaration
  d. Function call
  e. function parameters/arguments
- ✓ ii. List Benefits of using Function in C++

2: Write key findings on the paper/flipchart

3: Present your findings to the whole class

4: Ask clarifications if any.

5: Read the key **readings 1.5.1** in trainee manual.

Key readings 1.5.1.:

Description of function

1. **Introduction**

   A function is a block of code designed to perform a specific task.

2. **Purpose**

   Functions are used to perform certain actions, and they are important improves code organization, reusability, and readability.

3. **Function Definition**

   The function definition is where the actual implementation of the function resides. It contains the code that gets executed when the function is called. It consists of a return type, the function name, parameters (if any), and a block of code inside {}.

Syntax:

```
return_type function_name(parameter_list) {
   // Code to be executed
}
```

Example:

```
int add(int a, int b) {
   return a + b;
}
```

4. **Function Declaration (Prototype)**

   The function declaration, also known as the function prototype, tells the compiler about the function's name, return type, and parameters, but does not contain the function body. This is useful when the function is defined after it is called in the code, allowing the compiler to know about the function beforehand.

   Syntax:

   return_type function_name(parameter_list);

   Example:

   ```
   int add(int a, int b); // Declaration
   ```

5. **Function Call**

   A function call is the point in the program where the function is invoked. When called, the program jumps to the function definition, executes the code, and returns the result (if any) to the caller.

   Syntax

   ```
   function_name(argument_list);
   ```

   Example:

   ```
   int result = add(5, 10);  // Calling the function
   ```

   Here, the add function is called with arguments 5 and 10. The return value is stored in the variable result.

6. **Function Parameters**

   Function parameters are variables declared in the function definition that receive the values (arguments) when the function is called. They act as placeholders for the actual values (arguments) passed during the function call.

   Parameters are defined inside the parentheses of the function definition, and their data types must be specified.

   Example:

```
int add(int a, int b) {
   return a + b;
}
```

In this case, int a and int b are parameters. When the function is called, arguments like 5 and 10 are passed to these parameters.

Note:

C++ allows default values for parameters. If no argument is passed, the default value is used.

```
int multiply(int a, int b = 2) {
   return a * b;
}
```

Theoretical Activity 1.5.2: Description of Types of functions

Tasks:

1: Answer the questions bellow:

1) Describe the following:
    a. Built-in function
    b. User defined function
    c. Inline function
    d. Lambda function

2: write findings on paper/flichart

3: Present of the findings to the whole class

4: Asks for clarifications if any.

5: Read key reading 1.5.2 trainee manual

Key readings 1.5.2.

Description of Types of functions

C++ supports several types of functions and the main ones are described here bellow:

- ✓ Library function
- ✓ User defined function

1. Library function

Library functions are pre-defined functions provided by the C++ Standard Library. These functions are part of various libraries included in the C++ language and are designed to help developers with common tasks like mathematical calculations, input/output operations, string manipulation, and data handling.

Examples: Sqrt(), sin(), cosin(), len(), strlen(), pow()…

a. Characteristics

- ✓ **Efficient and optimized:**

Library functions are typically optimized for performance and efficiency since they are written by experts and have undergone extensive testing

They are often faster and more reliable than custom-written equivalents because they take advantages of low level optimization

- ✓ **Portable**

Since they are part of standard libraries, they are available across all compliant C++ compilers and platforms

- ✓ **Ease of use**

Library functions reduce the need for developers to write functions from scratch saving time and effort

They provide high-level abstraction for complex tasks allowing programmers to focus on the core logic of their applications

- ✓ **Reliable**

Library functions are well tested thoroughly documented and maintained by the community or standard organization (ISO). They are less prone to errors compared to user-defined functions that may not be rigorously tested.

- ✓ **Wide range of functionality**

  The C++ functions cover multiple domains such as mathematics, string manipulation, I/O operations, algorithm, memory management.

- ✓ **Safe and secure**

  Most library functions are designed to handle edge cases and error gracefully. Some of them are equipped with error-checking mechanisms to prevent common issues like buffer overflow

b. Using Library Functions

The following are steps to implement library functions in C++

- ✓ **Include the relevant header file:**

  At the beginning of your program using the `#include` directive.

- ✓ **Call the function:**

  Simply write the function name in your code with the appropriate arguments to invoke its action.

c. Program examples

1. Functions for manipulating C++ **std::string** objects, such as **length**(), **substr**(), **find**(), **append**(), and more.

```
 #include <iostream>
#include <string>
 using namespace std;
  int main() {
     string str = "Hello, World!";
     cout << "Length of string: " << str.length() << endl;
     return 0;
  }
```

2. Collection of functions for performing algorithms such as **sort**(), **find**(), **max**(), **min**(), etc.

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

  int main() {
     vector<int> numbers = {3, 1, 4, 1, 5};
     sort(numbers.begin(), numbers.end());  // Sort the vector
```

```
    for (int n : numbers) {
        cout << n << " ";
    }
    return 0;
}
```

3. Functions for manipulating C++ style strings. Common functions include `strcpy()`, `strlen()`, `strcmp()`, etc.

```
#include <iostream>

#include <cstring>

using namespace std;

int main() {
    char str1[20] = "Hello";
    char str2[20];
    strcpy(str2, str1);  // Copy str1 into str2
    cout << "Copied string: " << str2 << endl;
    return 0;
}
```

d. Common C++ library functions with their respective header files

Library/header file	**Function**
<**iostream**>:Input/Output Stream	**cin:** Standard input stream **cout:** Standard output stream **cerr**: Standard error stream **clog**: Standard logging stream
<**cmath**>: Math Functions	**sqrt**(): Square root of a number **pow**(): Power of a number **sin**(): Sine of an angle (in radians) **cos**(): Cosine of an angle (in radians) **tan**(): Tangent of an angle (in radians) **log**(): Natural logarithm (base `e`) **exp**(): Exponential function

	fabs(): Absolute value of a floating-point number **ceil**(): Ceiling of a number **floor**(): Floor of a number
<**cstdlib**>: General Utilities	**rand**(): Generates a random number **srand**(): Seeds the random number generator **abs**(): Absolute value of an integer **atoi**(): Converts a string to an integer **atof**(): Converts a string to a floating-point number **system**(): Executes a system command malloc(): Allocates dynamic memory **free**(): Deallocates dynamic memory
<**cstring**>:C-Style String Functions	**strlen**(): Returns the length of a C-style string **strcpy**(): Copies one string to another **strcmp():** Compares two strings **strcat():** Concatenates two strings **strchr():**Finds the first occurrence of a character in a string **strstr():** Finds the first occurrence of a substring in a string **memcpy():**Copies a block of memory
<string>: String Class	**length():** Returns the length of a string **substr():** Extracts a substring **find():** Finds a substring within a string **append():**Appends to the string **compare():** Compares two strings

	c_str(): Returns a C-style string equivalent of a string object
<vector>: Dynamic Array (Vector) Class	push_back(): Adds an element to the end of the vector **pop_back**(): Removes the last element of the vector **size**(): Returns the number of elements in the vector **clear**(): Removes all elements from the vector **at**(): Accesses an element at a specific index **begin**(): Returns an iterator to the first element **end**(): Returns an iterator to one past the last element
<**algorithm**>: Algorithms	**sort**(): Sorts elements in a range **reverse**(): Reverses elements in a range **find():** Finds an element in a range **count():** Counts the number of occurrences of a value in a range **min():**Returns the smaller of two values **max():** Returns the larger of two values **accumulate():** Computes the sum of elements in a range **copy():** Copies elements from one range to another
<map>: Associative Container (Map)	**insert():** Inserts elements into the map **find():** Finds an element with a specific key **erase():**Erases elements by key or iterator

	clear(): Clears all elements from the map **size():**Returns the number of elements in the map **begin():**Returns an iterator to the first element **end():** Returns an iterator to one past the last element
<list>: Doubly Linked List	**push_back():** Adds an element to the end of the list **push_front():**Adds an element to the beginning of the list **pop_back**(): Removes the last element **pop_front():** Removes the first element **insert():** Inserts an element at a specified position **erase():** Erases elements from a specified position **sort():** Sorts the elements in the list
<queue>: Queue Container	**push():** Adds an element to the back of the queue **pop():** Removes the front element from the queue **front():** Accesses the front element **back():** Accesses the last element **empty():** Checks whether the queue is empty **size():** Returns the number of elements in the queue
<stack>: Stack Container	**push():** Pushes an element onto the stack **pop():** Removes the top element of the stack

	top(): Returns the top element **empty():** Checks if the stack is empty **size():** Returns the number of elements in the stack
<deque>: Double-Ended Queue	**push_back():**Adds an element to the end of the deque :**push_front():**Adds an element to the beginning of the deque **pop_back():** Removes the last element of the deque. **pop_front():** Removes the first element of the deque. **at():** Accesses an element by index **size():**Returns the number of elements in the deque
<set>: Associative Container (Set)	**insert():** Inserts elements into the set **find():** Finds an element in the set **erase():** Erases elements from the set **clear():** Clears all elements from the set **size():** Returns the number of elements in the set **begin():** Returns an iterator to the first element **end():** Returns an iterator to one past the last element
<conio>: The conio.h header file provides several console input/output functions(primarily used in older DOS-based systems or specific environments	**getch():**Reads a single character from the keyboard without displaying it on the screen (no echo) **getche():** Reads a single character from the keyboard and displays (echoes) it on the screen.

like Turbo C++ or Borland C++.	**kbhit():** Checks if a key has been pressed. If a key has been pressed, it returns a non-zero value; otherwise, it returns 0. It does not wait for the key press. **clrscr():** Clears the console screen. **textcolor(int color):** Sets the color of the text in the console **textbackground(int color**): Sets the background color of the text in the console. **gotoxy(int x, int y):** Moves the cursor to a specified position (`x`, `y`) in the console window. Coordinates start at (1, 1). **cputs(const char *str**): Outputs a string at the current cursor position. **cprintf(const char *format, ...):** Works like `printf`, but outputs the formatted string to the console at the current cursor position. **putch(int char):** Outputs a single character at the current cursor position. **ungetch(int char**): Puts the character back into the input buffer so that it can be read again by subsequent input functions. **wherey():** Returns the current Y-coordinate of the cursor. **delline():** Deletes the current line where the cursor is positioned. **insline():** Inserts a blank line at the current cursor position and shifts the existing lines down. window(int left, int top, int right, int bottom): Defines a window within the

	console screen for further input/output operations. **highvideo():** Sets high-intensity text (bright colors). **lowvideo():** Sets low-intensity text (dim colors). **normvideo():**Resets the text to normal video intensity.

2. User-defined function

In C++, a user-defined function is a function that the programmer creates, as opposed to pre-defined functions provided by libraries.

Examples

AddTwo()// to add two numbers

FactorialNumber()// to return a factorial value of a number

a. **Function Declaration**

The declaration of the function (also known as a prototype) tells the compiler the brief information about function's name, return type, and the types of its parameters

The function must be declared before it is used in the program.

Syntax:

return_type function_name(ArgList);

Where

Return Type: Specify the data type the function will return

Function Name: Give the function a unique name. It should reflect to what function is used for and follows all rules for identifier.

Argument list /Parameters (ArgList): List any inputs the function will take. They indicate data type for each input (parameter) of the function.

; indicates the end of each statement.

Examples

✓ int MultplyTwo(int, int); // returns the product of two integer numbers

- ✓ float Average(int a, int b); //returns the average of two integer numbers
- ✓ void SetValues(int, int); // used to intialze two integer variables to two integer numbers
- ✓ void PrintInfo(string); // used to display an information as string value

Note:

Function declaration is just locate above main() function. This is optional if the function definition appears before its usage. However, using declarations is a good practice, especially in large programs

b. Function definition

Function definition tells the compiler the details of what function is to do and how it must work. You need to define the function, which involves writing the actual code that the function will execute.

It consists of the function's **name, return type, parameter list, and the body** of the function where the logic is implemented

Syntax:

Return type function_name(ArgList){

// function body

Return value; // (if return type is not void)

}

Where:

Function body: is the block of code enclosed in curly braces {} that defines what the function does and how it does it. It contains the statements and logic that perform the task and, if necessary, returns a value.

Return value/statement: it specifies the value that will be returned to the caller if the return type is not **void**. If the function's return type is `void`, no return statement is needed.

Notes: function definition take place outside the main function either before main or after main

Examples

1. function to add two numbers and return their sum

```
int addNumbers(int a, int b) {
      return a + b;  // The function body
   }
```

2. function to accept two numbers and display the greatest

```
void Great(int m, int n){
if (m>n){
cout<<”the greatest is “<<m<<endl;
}
Else
{
Cout<<”he greatest is:”<<n<<endl;
}
}
```

c. Function call

After defining the function, it must be called from within the **main** function or another function in the program. Function is to invoke the task that the function is to do. The call is where the function is executed, and arguments are passed to it if it takes parameters.

Syntax:

Function_name(ArgList);

Examples:

- ✓ int Result= MultplyTwo(12,45);
- ✓ double Mean=Average (7, 8);

Note: function call take place in the main function

Example of full programs:

1. A C++ program to implement the user defined function

```
#include <iostream>
// Function declaration (prototype)
int addNumbers(int, int);
// Main function
int main() {
   int num1, num2;
   std::cout << "Enter two numbers: ";
   std::cin >> num1 >> num2;
   // Function call
   int result = addNumbers(num1, num2);
```

```
    std::cout << "The sum is: " << result << std::endl;
    return 0;
}
// Function definition
int addNumbers(int a, int b) {
    return a + b;  // Returning the sum of two numbers
}
```

2. Implementing 4 different types of functions as per return types and argument list

```
#include <iostream>

#include <conio.h>

using namespace std;

//Declaration of 4 User-Defined Functions:

int SumTwoNumbers(void); //return value function without arguments

void MultiplyTwoNumbers(int,int); // Non return value with argument

float DivideTwoNumbers(int,int);// a return value function with argument

void CallAboveThreeFunctions(void);// non return value function without
argument

inta,b;

int main()

{

cout<< "\n\tWelcome to User-Defined Functions Operations:\n";

cout<< "\n\tPlease Enter Any Two Numbers to Perform Operations";

cout<< "\n\tAddition\n\tMultiplication\n\tDivide";

cout<< "\nEnter First Number: ";

cin>> a;

cout<< "\nEnter Second Number: ";

cin>> b;

CallAboveThreeFunctions(); //Calling of 4 User-defined Functions

getch();
```

```
   return 0;
}
//Definition of 4 User-Defined Functions:
intSumTwoNumbers(void)
{
int sum = 0;
   sum = a+b;
   return (sum);
}
void MultiplyTwoNumbers(int x, int y)
{
int multiply = 0;
   multiply = x*y;
cout<< "Multiplication Of Your Two Entered Numbers is:" << multiply <<endl;
}
float DivideTwoNumbers(intx,int y)
{
floatdivide = 0.0;
divide = x/y;
return(divide);
}
void CallAboveThreeFunctions(void)
{
int sum = SumTwoNumbers();
   float divide = DivideTwoNumbers(a,b);
cout<< "Addition Of Your Two Entered Numbers is: " << sum <<endl;
cout<< "Division Of Your Two Entered Numbers is:" << divide <<endl;
```

MultiplyTwoNumbers(a,b);

}

Notes:

A function can be:

- ✓ A return value function with argument
  - **Return type funct_name(arglist);**
- ✓ A return value function without arguments
  - **Return type Funct_name(void);**
- ✓ A non return value function with argument
  - **Void funct_name(arglist);**
- ✓ A non return value without argument
  - **Void funct_name(void);**

Recursive function

A recursive function in C++ is a function that calls itself to solve smaller instances of the same problem.

Components of recursive function

Recursive functions often have two main components.

- ✓ **Base Case**:

  It is the condition that stops the recursion to prevent infinite loops.

- ✓ **Recursive Case**:

  It is the part where the function calls itself with a modified parameter that gradually approaches the base case.

Examples:

- ✓ Factorial Function Using Recursion

```
#include <iostream>
// Recursive function to calculate factorial
int factorial(int n) {
   if (n <= 1) {
      return 1;  // Base case: factorial of 0 or 1 is 1
   } else {
      return n * factorial(n - 1);  // Recursive case
   }
}
```

```
int main() {
   int num = 5;
   std::cout << "Factorial of " << num << " is " << factorial(num) << std::endl;
   return 0;
}
```

Explanation:

Base Case: When n <= 1, the recursion stops, and the function returns **1**.

Recursive Case: If n is greater than 1, the function calls itself with the argument **n - 1**. This continues until **n** is reduced to 1, at which point the base case is satisfied.

Steps:

a. **Initial Call:** factorial(5) is called from main().
b. Recursive Calls:

- factorial(5) calls factorial(4)
- factorial(4) calls factorial(3)
- factorial(3) calls factorial(2)
- factorial(2) calls factorial(1)

c. Base Case Reached: factorial(1) returns 1, and each recursive call starts returning back up the stack:

- factorial(2) returns 2 * 1 = 2
- factorial(3) returns 3 * 2 = 6
- factorial(4) returns 4 * 6 = 24
- factorial(5) returns 5 * 24 = 120

✓ Function to find Fibonacci number

```
int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

✓ Function to add digits of a given number

```
int sumOfDigits(int n) {
   if (n == 0) return 0;
   return n % 10 + sumOfDigits(n / 10);
}
```

✓ Function to calculate the exponentiation(power) of number

```
int power(int base, int exp) {
   if (exp == 0) return 1;
```

```
    return base * power(base, exp - 1);
}
```

✓ Function to find the greatest common divisor of two numbers

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

✓ Function for binary search

```
int binarySearch(int arr[], int low, int high, int key) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key) return mid;
    else if (arr[mid] > key) return binarySearch(arr, low, mid - 1, key);
    else return binarySearch(arr, mid + 1, high, key);
}
```

✓ Function to return the sum of first N natural numbers

```
int sum(int n) {
    if (n == 0) return 0;
    return n + sum(n - 1);
}
```

✓ Function to count digits of a given integer number

```
int countDigits(int n) {
    if (n == 0) return 0;
    return 1 + countDigits(n / 10);
}
```

✓ Function to reverse a string

```
void reverseString(string &str, int start, int end) {
    if (start >= end) return;
    swap(str[start], str[end]);
    reverseString(str, start + 1, end - 1);
}
```

✓ Function to check palindrome string

```
bool isPalindrome(string str, int start, int end) {
    if (start >= end) return true;
    if (str[start] != str[end]) return false;
```

```
    return isPalindrome(str, start + 1, end - 1);
}
```

✓ Function for merge sort

```
void mergeSort(int arr[], int l, int r) {
    if (l >= r) return;
    int mid = l + (r - l) / 2;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid + 1, r);
    merge(arr, l, mid, r);
}
```

✓ Function for quick sort

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

Inline function

It is a function that is expanded in line when it is invoked, meaning that the compiler replaces the function call with the actual code of the function. This can result in faster execution because it avoids the overhead of function calls.

a. Characteristics of Inline Functions

✓ **No Function Call Overhead:**

The function's code is inserted at the point of the call, avoiding the overhead of a function call (such as saving registers and returning control).

✓ **Faster Execution:**

Since there's no function call, execution may be faster, especially for small functions.

✓ **Binary Size:**

Inline functions may increase the size of the binary if they are invoked many times because the function code is copied each time it's called.

Examples:

- Computing the square of a number

```
#include <iostream>
using namespace std;
inline int square(int x) {
   return x * x;
}
int main() {
   int num = 5;
   cout << "Square of " << num << " is " << square(num) << endl;
   return 0;
}
```

- **Program to add two Numbers**

```
#include <iostream>
using namespace std;
inline int add(int a, int b) {
   return a + b;
}

int main() {
   int x = 10, y = 20;
   cout << "Sum: " << add(x, y) << endl;
   return 0;
}
```

- **Program to find the maximum of two numbers**

```
#include <iostream>
using namespace std;
inline int max(int a, int b) {
   return (a > b) ? a : b;
}
int main() {
   int x = 30, y = 20;
   cout << "Maximum: " << max(x, y) << endl;
   return 0;
}
```

- **Program to calculate the square of a number**

```
#include <iostream>
using namespace std;
inline int square(int x) {
   return x * x;
```

```
}
int main() {
   int num = 5;
   cout << "Square of " << num << " is " << square(num) << endl;
   return 0;
}
```

- **Program to check if a number is Even**

```
#include <iostream>
using namespace std;
inline bool isEven(int n) {
   return n % 2 == 0;
}
int main() {
   int num = 42;
   if (isEven(num)) {
      cout << num << " is even." << endl;
   } else {
      cout << num << " is odd." << endl;
   }   return 0;
}
```

Notes:

- ✓ Inline functions must be defined in the header file (or in the same file where they are used) because the compiler needs to replace the function call with the function's code.
- ✓ Inline functions cannot have recursion, as this would prevent inlining due to the repetitive nature of recursive calls.
- ✓ The `inline` keyword is generally not necessary in modern C++, as the compiler is quite good at optimizing small functions to be inlined without explicit hints

3. Lambda function

Lambda function (also known as lambda expression), is a concise way to define an anonymous function directly in your code. It provides a way to create inline functions, which are especially useful for short and simple tasks, such as in algorithms or callbacks. Lambda function allows write functions without explicitly naming them, which can make the code more readable and maintainable.

Syntax:

[capture_List](Parameters)->return_type{

//function body

}

Where

[capture_list]: Specifies which variables from the surrounding scope are accessible inside the lambda function. You can capture variables by value or by reference.

(parameters): Lists the parameters that the lambda function takes, similar to a regular function.

-> **return_type** (optional), Specifies the return type of the lambda function. If omitted, the return type is deduced automatically by the compiler.

Function body: The code that will be executed when the lambda is called.

Examples

Program to add two numbers using lambda function

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

int main() {
   // Define a lambda function that adds two numbers
   auto add = [](int a, int b) -> int {
      return a + b;
   };
   // Call the lambda function
   cout << "Sum: " << add(5, 10) << endl;  // Output: Sum: 15
   return 0;
}
```

In this example:

- ✓ The lambda function is defined inline using [].
- ✓ It takes two integer parameters, a and b, and returns their sum.
- ✓ The return type is explicitly stated as int using -> int.

Capturing Variables in Lambda Functions

A lambda function can capture variables from its surrounding scope. This is done by specifying a **capture list** inside the square brackets [].

There are two main ways to capture variables:

1. **By Value (=):** The lambda makes a copy of the variables from the outer scope.
2. **By Reference (&):** The lambda function works with references to the variables, allowing modifications to affect the original variables.

Example of Capturing by Value

```
#include <iostream>
using namespace std;
int main() {
   int x = 10;
   auto captureByValue = [x]() {
      return x * 2;
   };
   cout << "Captured by value: " << captureByValue() << endl;  // Output: 20
   return 0;
}
```

Example of Capturing by reference

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

int main() {
   int x = 10;
   auto captureByReference = [&x]() {  x *= 2;
   };
   captureByReference();
   cout << "Captured by reference: " << x << endl;  // Output: 20
   return 0;
}
```

Mutable Lambdas

By default, variables captured by value inside a lambda are **const**. However, you can modify captured values by adding the `mutable` keyword.

Example: Mutable Lambda

```
#include <iostream>
using namespace std;
int main() {
   int x = 10;
   auto mutableLambda = [x]() mutable {
      x += 5;
```

```
        return x;
    };
    cout << "Modified in lambda: " << mutableLambda() << endl;  // Output: 15
    cout << "Original x: " << x << endl;                        // Output: 10
    return 0;
}
```

Lambda Usage in STL Algorithms

Lambda functions are often used with algorithms in the Standard Template Library (STL). For example, you can use a lambda function with s**td::for_each**, **std::sort**, or **std::find_if**

Example: Using Lambda with **std::for_each**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    // Use lambda with for_each to print each element
    for_each(numbers.begin(), numbers.end(), [](int n) {
        cout << n << " ";
    });
    return 0;
}
```

Note: Lambda expressions provide powerful and concise ways to define small functions directly at the point of use in C++

Practical Activity 1.5.3: Applying function in C++

Task:

1: Read and perform the following task:

As a firmware development technician you are tasked to write and run C++ program that takes two integer numbers, finds their sum, and calculates one number raised to the power of the other using **pow**() function.

2: Read Key readings **1.5.3.**

3: Apply functions in C++ by Performing above task

4: Present your work to the trainer

Key readings 1.5.3

Applying function in C++

Running a C++ Program implements functions, follow these steps to run C++ program that takes two integer numbers, finds their sum, and calculates one number raised to the power of the other using functions:

1. **Open your Dev-C++:**
   Click the Dev-C++  icon on the desktop/TaskBar or find it on the start menu

2. **Write Your Program**

   with your open interface for Dev-C++ type in the following codes:

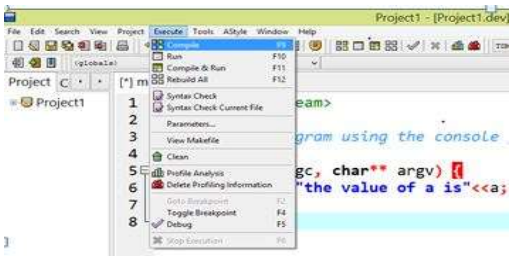
✓ Define the functions for sum and power calculation

- Write a function `calculateSum` that will take two integers as input and return their sum
- Write a function `calculatePower` that will take two integers as input and return the result of the first number raised to the power of the second number

✓ Write the `main` function:

- In the `main` function, prompt the user to enter two integers
- Call the `calculateSum` function and the `calculatePower` function to perform the required calculations
- Display the results

```
#include <iostream>
#include <cmath>  // for pow function
using namespace std;
// Function to calculate the sum of two numbers
int calculateSum(int a, int b) {
   return a + b;
}
// Function to calculate the power of one number to another
int calculatePower(int base, int exponent) {
   return pow(base, exponent);
}
int main() {
   int num1, num2;
   // Input two integers
   cout << "Enter the first integer: ";
   cin >> num1;
   cout << "Enter the second integer: ";
   cin >> num2;
   // Calculate the sum
   int sum = calculateSum(num1, num2);
   // Calculate the power (num1 raised to the power of num2)
   int power = calculatePower(num1, num2);
   // Display the results
   cout << "The sum of " << num1 << " and " << num2 << " is: " << sum << endl;
   cout << num1 << " raised to the power of " << num2 << " is: " << power <<
endl;
   return 0;
}
```



3. **Compile the Program:**
   - ✓ Go to execute tab on the menu bar
   - ✓ In the pop up that appears choose compile
4. **Run the Program:**
   - ✓ Simply click on the “Run” or “Build and Run” button from execute menu
   - ✓ The IDE will compile and execute the program automatically.

5. **Check Output:**
   When the program runs successfully, the console or terminal will display:

```
Enter the first integer: 6
Enter the second integer: 3
The sum of 6 and 3 is: 9
6 raised to the power of 3 is: 216

--------------------------------

Process exited after 8.028 seconds with return value 0
Press any key to continue . . .
```

Points to Remember

Description of function

- C++ functions can be found in various types such as
  - Library function; those which are predefined in the system language. Ex: sqrt(), pow()…
  - User defined functions; those functions that are defined by the user. Ex: AddTwo(), PrintString(),…
- A user defined function may be classified as:
  - Inline function
  - Lambda function

Applying function in C++

- For a function to be implemented it must:
  - Be declared
  - Be defined
  - Be called
- To call a user defined function two ways are provided:
  - Call by value
  - Call by reference
- To run a program that implements C++ function follow these steps
  - Start the IDE(Dev-C++)
  - Write codes
  - Test the written program codes

Application of learning 1.5

XYZ SACCO is creating a desktop banking application to help customers manage their savings accounts offline. The program enables users to deposit funds, withdraw money (with insufficient funds checks), check account balances, and calculate interest over a time at a fixed rate. Your task is to create a C++ program than should simulate this banking system.

Indicative content 1.6: Apply Array

Duration: 4hrs

Theoretical Activity 1.6.1.: Descriptions of array

Tasks:

1: Answer the questions that follow:

1. Describe the following :
   a) Array in C++
   b) Array declaration
   c) Array initialization
   d) Access array elements in C++
   e) Iterating Over Array
2. Identify types of array in C++

2: Write the findings on paper / flipchart

3: Presentation your findings to the whole class

4: Asks clarifications if any

5: Read Key **readings 1.6.1** in trainee manual

Key readings 1.6.1

DESCRIPTION OF AN ARRAY

1. Definition

In C++, an **array** is a collection of elements of the same data type that are stored in contiguous memory locations.

b. Example:

Array of numbers containing marks of students in a certain course

Marks					
25	67	14	78	34	90
0	1	2	3	4	5

c. Components of an array

An array consists of the following components:

✓ **Elements**

It is the actual data items stored in the array. All elements are of the same data type (e.g., integers, floats, characters).

✓ **Index**

It is a numerical representation of the position of an element in the array. The index typically starts at 0 in most programming languages and increases sequentially.

✓ **Array Name**

This is the identifier used to reference the array in the code. It serves as a reference point to access or manipulate the array elements by differentiating it from others.

✓ **Size/Length**

It is meant by the total number of elements that the array can hold. The size of the array is usually fixed at the time of creation in the case of static arrays.

✓ **Memory Location**

It refers to the location in the computer's memory where the array's elements are stored. Specifically, it is the memory address of the first element (also called the **base address**) of the array, since the array elements are stored in contiguous memory locations.

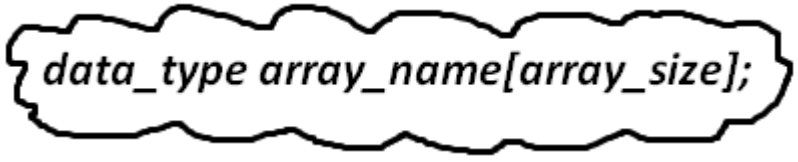
Note:

These components together allow an array to store, access, and manipulate a collection of homogeneous data efficiently.

2. Array declaration

Array declaration refers to the definition of the array by specifying the **data type**, the **array name**, and **the number of elements** (size) it will hold. The size of the array must be a constant expression that defines how many elements the array can store.

Syntax:



Where

- **data_type**: is the type of elements the array will hold (e.g., int, float, char, etc.).
- **array_name:** The identifier (name) for the array.
- **array_size:** The number of elements that the array will contain.

Examples

int numbers[5]; // Declares an integer array of size 5

float Weight[20]; // declares a float array of size 20

char Name[100]; // declares a character array of size 100

int Matrix[3][4];// declares a 2D array with 3 rows and 4 columns

3. Array initialization

Different ways are employed to initialize array.

- **Initializing individual item:**

number[0]=34;// only initializes the first element by 34

number[1]=5; ;// only initializes the second element by 5

number[3]=8; ;// only initializes the fourth element by 8

- **Initializing at the time of declaration**

```
int numbers[5] = {1, 2, 3, 4, 5};  // Initializes an array of 5 integers with specified
//values
int numbers[5] = {1, 2};  // Initializes the first two elements; remaining elements
are set to 0
// Array: {1, 2, 0, 0, 0}
int numbers[] = {1, 2, 3, 4, 5};  // The size is automatically determined as 5
int numbers[5] = {};  // All elements are initialized to 0
char name[6] = "Alice";  // Initializes with the string "Alice" including the null
// terminator Array: {'A', 'l', 'i', 'c', 'e', '\0'}
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};  // Initializes a 2x3 array with specified values
int tensor[2][2][2] = {
   {
      {1, 2},
      {3, 4}
   },
   {
```

```
            {5, 6},
            {7, 8}
        }
    }; // Initializes a 3D array with specified values
```

4. Accessing Array Element

Accessing elements in an array involves using index to refer to a specific element within array.

Examples

- **Accessing elements**

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

int first = numbers[0];  // Accesses the first element, which is 10

int second = numbers[1]; // Accesses the second element, which is 20

int matrix[2][3] = {
            {1, 2, 3},
            {4, 5, 6}
        };

int value = matrix[1][2];  // Accesses the element in the second row, third column, which is 6
```

- **Modifying Elements**

```
numbers[2] = 35;  // Changes the third element from 30 to 35

matrix[0][1] = 10;
```

5. Iterating Over Array

To iterate over an array in C++, you can use various methods depending on whether you're working with a single-dimensional or multidimensional array. Here are some common techniques

✓ **Using a For Loop**

```
int numbers[5] = {10, 20, 30, 40, 50};
        for (int i = 0; i < 5; ++i) {
            std::cout << numbers[i] << " ";  // Access and print each element
        }
        std::cout << std::endl;
```

✓ **Using a Range-Based For Loop (C++11 and Later)**

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

        for (int num : numbers) {
           std::cout << num << " ";  // Access and print each element
        }
        std::cout << std::endl;
```

Note:

The above techniques is applied in C++11 an later versions.

✓ **Using a While Loop**

```
int numbers[5] = {10, 20, 30, 40, 50};
        int i = 0;
        while (i < 5) {
           std::cout << numbers[i] << " ";  // Access and print each element
           ++i;
        }
        std::cout << std::endl;
```

✓ **Using Nested For Loops**

```
 int matrix[2][3] = {
           {1, 2, 3},
           {4, 5, 6}
        };

        for (int i = 0; i < 2; ++i) {  // Iterate over rows
           for (int j = 0; j < 3; ++j) {  // Iterate over columns
              std::cout << matrix[i][j] << " ";  // Access and print each element
           }
           std::cout << std::endl;
        }
```

6. TYPES OF ARRAY

In C++, arrays can be categorized into several types based on their dimensions and whether they are fixed or dynamic.

k. Single-Dimensional Arrays

A single-dimensional array, also known as a one-dimensional array, is a list of elements of the same type arranged in a linear order

Example

```
int numbers[5] = {1, 2, 3, 4, 5};  // An array of 5 integers
```

l. Multi-Dimensional Arrays

Multi-dimensional arrays are arrays of arrays. They are used to represent data in more than one dimension, such as matrices.

These include:

- **Two-Dimensional Arrays**: Representing tables or matrices with rows and columns.

Example

```
int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };  // A 3x4 matrix
```

- **Three-Dimensional Arrays:** Extend the concept of a 2D array into three dimensions.

Example

```
int tensor[2][3][4] = {
        {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        },
        {
            {13, 14, 15, 16},
            {17, 18, 19, 20},
            {21, 22, 23, 24}
        }
    };  // A 2x3x4 tensor
```

m. Dynamic Arrays

Dynamic arrays are allocated at runtime, allowing the size to be determined during program execution. They are implemented using pointers and can be resized if needed

- **Dynamic Array Using `new`**

Example

```
int* dynamicArray = new int[10];  // Allocates an array of 10 integers
```

- **Dynamic Array with Standard Library Containers (e.g., `std::vector`):**

```
#include <vector>
```

```
std::vector<int> vec = {1, 2, 3, 4, 5};  // A dynamic array with initial values
```

n. Static Arrays

Static arrays are arrays with a fixed size determined at compile time. They are allocated on the stack.

Example

```
int staticArray[10];  // A static array of 10 integers.
```

Practical Activity 1.6.2: Applying Arrays

Task:

1: Read and the task below:

- ✓ Develop a C++ program to manage student grades using both single and multi-dimensional arrays. This program will store students' names and grades, calculate the class average, and display the grade in a tabular format.

2: Read Key readings **1.6.2** in trainee manual

3: Apply array in C++ by performing task 1

4: Verify the out of your program.

Key readings 1.6.2

Applying array

Write and run a C++program that creates, initializes, and manipulates arrays (both single-dimensional and multi-dimensional arrays) to manage and process student grades for a class.

This program will:

- ✓ Store students' names and their grades
- ✓ Calculate the average grade
- ✓ Display the grades in a tabular format.

Steps

Step 1:Create a New C++ Source File

✓ File Creation

Create a new file named `grades_management.cpp` or any name you prefer.

✓ Open the File

Open the file in a text editor or IDE.

Step 2:Write or Copy the Program Code

Type in the following code into your source codes:

```
#include <iostream>
#include <iomanip>  // For std::setw

const int numStudents = 3;
const int numSubjects = 4;

int main() {
   // 1. Define and initialize arrays
   std::string studentNames[numStudents] = {"Alice", "Bob", "Charlie"};
   int grades[numStudents][numSubjects] = {
      {85, 90, 78, 92},
      {88, 76, 95, 89},
      {91, 84, 77, 85}
   };

   // 2. Calculate and display average grades
```

```
        std::cout << "Student Grades and Averages:\n";
        std::cout << std::setw(12) << "Name" << std::setw(10) << "Math" <<
    std::setw(10) << "Science" << std::setw(10) << "English" << std::setw(10) <<
    "History" << std::setw(10) << "Average\n";
        std::cout << std::string(60, '-') << '\n';

        for (int i = 0; i < numStudents; ++i) {
           int sum = 0;
           std::cout << std::setw(12) << studentNames[i];

           for (int j = 0; j < numSubjects; ++j) {
              std::cout << std::setw(10) << grades[i][j];
              sum += grades[i][j];
           }

           double average = static_cast<double>(sum) / numSubjects;
           std::cout << std::setw(10) << std::fixed << std::setprecision(2) << average
    << '\n';
        }

        return 0;
     }
```

Step 3:Compile and run the Program

✓ Using Command Line:

- Navigate to the directory where your source file is saved.
- Run the following command to compile the program:

```bash
g++ -o grades_management grades_management.cpp
```

- This command compiles `grades_management.cpp` and produces an executable named `grades_management`.

✓ Using an IDE:

- Open the project or source file in your IDE.
- Use the build or compile option provided by the IDE (usually found in the "Build" or "Compile" menu).

Running only after compilation

Step 4:Run the Program

✓ Using Command Line:

- Execute the compiled program with the following command:

```bash
./grades_management
```

- This runs the program and displays the output in the terminal.

✓ Using an IDE:

- Run the program using the IDE's run option (usually a "Run" or "Execute" button).

Step 5:Review the Output

Verify that the output matches your expectations, with student names, grades, and average grades displayed correctly in a tabular format.

```
Student Grades and Averages:
        Name      Math   Science   English   History  Average
-------------------------------------------------------------

       Alice        85        90        78        92    86.25
         Bob        88        76        95        89    87.00
     Charlie        91        84        77        85    84.25

--------------------------------
Process exited after 1.054 seconds with return value 0
Press any key to continue . . .
```

Points to Remember

Description of array

- An array is a collection of more than one item with similar data types stored in contiguous memory location.
- Array is basically made of array element, array index, array name, array memory address as the basic components that allow to store, access, and manipulate its data efficiently
- Before using array you must first declare it and can be initialized through various method such as individual item initialization or declaration time initialization
- Index is an array component that serves for its data access and manipulation.
- To classify array refer to its size and the way it is declared. Hence we can find:
    - Single dimension arrays
    - Multi dimensions array (2D, 3D arrays)
    - Static arrays
    - Dynamic arrays

Applying array

- The following are steps involved to run a program that applies arrays in C++

- Start the IDE(Dev-C++)
- Write codes
- Test the written program codes

Application of learning 1.6

At XYZ TSS, a C++ trainer for a class of 25 trainees needs to automate the management of trainees' test grades. The task involves creating a C++ program that allows entering marks, calculating the average grade, identifying the highest and lowest grades, and displaying all grades in tabular format.

Learning outcome 1 end assessment

Theoretical assessment

Written assessment

I. Choose the correct answer about apply C++ concepts

1. Who invented C++?

   a) Dennis Ritchie
   b) Ken Thompson
   c) Brian Kernighan
   d) Bjarne Stroustrup

2. Which of the following is used for comments in C++?

   a) /* comment */
   b) // comment */
   c) // comment
   d) Both // comment and /* comment */

3. Which of the following is a correct identifier in C++? a) VAR_1234
   b) $var_name
   c) 7VARNAME
   d) 7var_name

4. Which of the following is not a type of constructor in C++?

   a) Default constructor
   b) Parameterized constructor
   c) Copy constructor
   d) Friend constructor

5. What is the correct syntax for including a user-defined header file in C++?

   a) #include [userdefined]
   b) #include "userdefined"
   c) #include <userdefined.h>
   d) #include <userdefined>

6. What is the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main() {
   int a = 5;
  int b = 10;
 cout << a + b;
return 0;
      }
```

a) 5
b) 10
c) 15
d) Compilation error

7. Which of the following is used to define a constant in C++? a) #define
   b) const
   c) static
   d) Both #define and const

8. What is the default access specifier for members of a class in C++? a) Public
   b) Private
   c) Protected
   d) None of the above

9. Which of the following is the correct syntax to declare a pointer in C++?

   a) int *ptr;
   b) int ptr*;
   c) int* ptr;
   d) Both a and c

10. What is the purpose of the return statement in C++? a) To exit a loop
    b) To exit a function and return a value
    c) To define a constant
    d) To allocate memory
    Answer: b) To exit a function and return a value

II. Choose the correct answer about setting up a development environment for C++ along with their answers:

1. Which of the following is an essential component for setting up a C++ development environment?

   a) Text Editor
   b) Compiler
   c) Debugger
   d) All of the above

2. Which IDE is commonly used for C++ development on Windows?
3. a) Xcode
   b) Visual Studio
   c) Eclipse
   d) Code::Blocks

4. What is the purpose of a C++ compiler?

   a) To edit the source code
   b) To convert source code into machine code
   c) To debug the program
   d) To run the program directly

5. What is the file extension for C++ source files?

   a) .c
   b) .cpp
   c) .java
   d) .py

6. Which of the following is an online IDE for C++?

   a) Geany
   b) ide.geeksforgeeks.org
   c) Notepad++
   d) Sublime Text

7. What is the role of a debugger in a C++ development environment?

   a) To write code
   b) To compile code
   c) To find and fix errors in the code

d) To execute code

8. Which command is used to compile a C++ program using the GCC compiler?

   a) gcc program.cpp
   b) g++ program.cpp
   c) compile program.cpp
   d) run program.cpp

9. What is the purpose of setting the PATH environment variable for a compiler?

   a) To specify the location of source files
   b) To specify the location of the compiler executable
   c) To specify the output directory for compiled files
   d) To specify the location of libraries

10. Which of the following is a cross-platform IDE for C++ development?

    a) Visual Studio
    b) Xcode
    c) Code::Blocks
    d) Notepad++

III. Choose the correct answer about about applying variables in C++

1. Which of the following is the correct way to declare an integer variable in C++?

   a) int var;
   b) integer var;
   c) int: var;
   d) var int;

2. What is the default value of a local variable in C++? a) 0
   b) NULL
   c) Undefined
   d) 1

3. Which of the following is a valid variable name in C++? a) 1variable
   b) variable_1
   c) variable-1
   d) variable 1

4. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main() {
 int a = 5;
  int b = 2;
  int c = a + b;
 cout << c;
  return 0;
        }
```

a) 5
b) 2
c) 7
d) Compilation error

5. Which of the following is used to declare a constant variable in C++? a) const
   b) constant
   c) final
   d) static

6. What is the size of an int variable in C++?

   a) 2 bytes
   b) 4 bytes
   c) 8 bytes
   d) Depends on the system/compiler

7. Which of the following is the correct syntax to initialize a variable at the time of declaration? a) int var = 10;
   b) int var : 10;
   c) int var == 10;
   d) int var <- 10;

8. What will be the output of the following C++ code?

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

```
int main() {
  int x = 10;
int y = 20;
x = y;
 cout << x;
 return 0;
        }
```

a) 10
b) 20
c) 30
d) Compilation error

9. Which of the following is not a valid variable declaration in C++? a) int var1;
   b) float var2;
   c) double var3;
   d) char var4 = 'A';
   What is the scope of a variable declared inside a function in C++? a) Global
   b) Local
   c) Static
   d) Dynamic
10. What is the scope of a variable declared inside a function in C++? a) Global
    b) Local
    c) Static
    d) Dynamic

IV. Choose the correct answer about about applying operators in C++

1. Which of the following is the assignment operator in C++?

   a) ==
   b) =
   c) ===
   d) :=

2. Which operator is used for checking equality? a) :=
   b) =
   c) ==
   d) !=

3. What is the result of 5 % 2?

   a) 2.5
   b) 2
   c) 1
   d) 0

4. Which operator increases the value of a variable by 1?

   a) ++
   b) --
   c) +=
   d) *=

5. What is the purpose of the != operator?

   a) Multiplication
   b) Modulus
   c) Not equal to
   d) Division

6. What does the && operator represent? a) Bitwise AND
   b) Logical AND
   c) Bitwise OR
   d) Logical OR

7. Which operator has the highest precedence? a) +
   b) /
   c) %
   d) ()

8. What is the result of 10 | 5? a) 15
   b) 5
   c) 2
   d) 10

9. Which operator is used to allocate dynamic memory in C++? a) malloc
   b) new
   c) alloc
   d) create

10. What does the ^ operator do in C++?

    a) Exponentiation
    b) Logical XOR
    c) Bitwise XOR
    d) None of the above

V. Choose the correct answer about applying control structures in C++

1. Which control structure is used for executing a block of statements repeatedly based on a condition?
   a) if
   b) switch
   c) loop
   d) goto

2. Which keyword is used to test a condition in C++?

   a) test
   b) switch
   c) decide
   d) if

3. Which of the following is NOT a loop in C++?

   a) for
   b) while
   c) do
   d) check

4. How many times is the body of a do-while loop guaranteed to execute?

   a) 0
   b) 1
   c) Until the condition is true
   d) Infinitely

5. What will be the output of the following code snippet?

```
int x = 5;
if (x == 5)
   cout << "Yes";
else
  cout << "No";
```

   a) No
   b) Yes
   c) Error
   d) None of the above

6. Which control structure allows you to choose between multiple alternatives?

   a) if-then
   b) for
   c) switch-case
   d) while

7. What is the purpose of the break statement in loops?

   a) To start the loop
   b) To exit the loop immediately
   c) To skip one iteration
   d) None of the above

8. Which loop is best suited for iterating over arrays when you know the number of iterations in advance?

   a) if
   b) do-while
   c) while
   d) for

9. How can you execute a block of code irrespective of whether a condition in an if statement is true or false?

   a) else
   b) elseif
   c) then
   d) finally

10. Which of the following statements will run indefinitely?

    a) for(; ;) { }
    b) while(1) { }
    c) do { } while(1);
    d) All of the above

VI. Choose the correct answer about applying functions in C++

1. Which of the following is the default return value of functions in C++ if no return type is specified?

   a) int
   b) char
   c) float
   d) void

2. What happens to a function defined inside a class without any complex operations (like looping, a large number of lines, etc.)?

   a) It becomes a virtual function of the class
   b) It becomes a default calling function of the class
   c) It becomes an inline function of the class
   d) The program gives an error

3. What is an inline function?

   a) A function that is expanded at each call during execution
   b) A function that is called during compile time
   c) A function that is not checked for syntax errors
   d) A function that is not checked for semantic analysis

4. When are inline functions expanded?

   a) Compile-time
   b) Run-time
   c) Never expanded
   d) End of the program

5. In which of the following cases may inline functions not work?

   i) If the function has static variables.
   ii) If the function has global and register variables.
   iii) If the function contains loops
   iv) If the function is recursive
   a) i, iv
   b) iii, iv
   c) ii, iii, iv
   d) i, iii, iv

6. When do we define the default values for a function?

   a) When a function is defined
   b) When a function is declared
   c) When the scope of the function is over
   d) When a function is called

7. Where should default parameters appear in a function prototype?

   a) To the rightmost side of the parameter list
   b) To the leftmost side of the parameter list
   c) Anywhere inside the parameter list
   d) Middle of the parameter list

8. If an argument from the parameter list of a function is defined as constant, then:

   a) It can be modified inside the function
   b) It cannot be modified inside the function
   c) Error occurs
   d) Segmentation fault

9. Which of the following features is used in function overloading and functions with default arguments?

   a) Encapsulation
   b) Polymorphism
   c) Abstraction

10. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
int fun(int x = 0, int y = 0, int z) {
    return (x + y + z);
}
int main() {
 cout << fun(10);
   return 0;
            }
```

   a) 10
   b) 0
   c) Error
   d) Segmentation fault

VII. Choose the correct answer about applying arrays in C++

1. Which of the following correctly declares an array in C++?

   a) int array[10];
   b) int array;
   c) array{10};
   d) array array[10];

2. What is the index number of the last element of an array with 9 elements?

   a) 9
   b) 8
   c) 0
   d) Programmer-defined

3. What is the correct definition of an array?

   a) An array is a series of elements of the same type in contiguous memory locations
   b) An array is a series of elements of different types in contiguous memory locations
   c) An array is a series of elements of the same type placed in non-contiguous memory locations
   d) An array is an element of different types

4. Which of the following accesses the seventh element stored in an array? a) array[6];
   b) array[7];
   c) array(7);
   d) array;

5. Which of the following gives the memory address of the first element in an array?

   a) array[0];
   b) array[1];
   c) array(2);
   d) array;

6. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main() {
 int array[] = {0, 2, 4, 6, 7, 5, 3};
int n, result = 0;
for (n = 0; n < 7; n++) {
result += array[n];
  }
cout << result;
     return 0;
```

```
}
```

a) 25
b) 26
c) 27
d) 21

7. How do you pass an array to a function in C++?

   a) By copying
   b) By value
   c) By pointer
   d) By name

8. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
void modifyArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
    arr[i] += 5;
  }
}
int main() {
  int arr[3] = {1, 2, 3};
  modifyArray(arr, 3);
  for (int i = 0; i < 3; i++) {
     cout << arr[i] << " ";
  }
  return 0;
}
```

a) 1 2 3
b) 6 7 8
c) 1 7 3
d) Compilation error

9. Which of the following are true about multidimensional arrays in C++?

   a) They are arrays of arrays
   b) The syntax to declare a 2D array is type arrayName[rows][columns];
   c) They can have two or more dimensions
   d) All of the above

10. What is the size of the array char arr[] = "geeksforgeeks";?

    a) 12
    b) 13
    c) 15
    d) 14

VIII. Answer the questions below by filling the missing terms in the blank space

1. The keyword used to define a constant variable in C++ is _____.
2. The _____ loop is guaranteed to execute at least once.
3. The _____ statement is used to exit a loop prematurely.
4. A function that calls itself is known as a _____ function.
5. The return type of a function that does not return a value is _____.
6. The _____ statement is used to exit a loop prematurely.
7. A function that calls itself is known as a _____ function.
8. The return type of a function that does not return a value is _____.
9. An array is a collection of elements of the same _____ stored in contiguous memory locations.
10. Question: The index of the first element in a C++ array is _____.
11. Question: The correct way to declare an array of 10 integers is _____.
12. To initialize an array of 5 integers with values 1, 2, 3, 4, 5, you write _____.
13. The syntax to access the third element of an array named arr is _____.
14. To change the value of the fifth element of an array arr to 10, you write _____.
15. Question: A two-dimensional array can be declared as _____.
16. Question: To access the element in the second row and third column of a 2D array arr, you write _____.
17. The function _____ can be used to get the size of an array in C++.
18. Question: To pass an array to a function, you typically pass the _____ of the array.
19. To find the maximum value in an array, you need to iterate through the array using a
20. The keyword used to declare a variable in C++ is _____.
21. To initialize a variable x of type int with the value 10, you write _____.
22. A variable declared inside a function is called a _____ variable.
23. A variable declared outside all functions is called a _____ variable.
24. A variable that can hold a sequence of characters is of type _____.

25. A variable that can hold a true or false value is of type _____.
26. A constant variable must be initialized at the time of _____.
27. The data type used to store a sequence of characters is _____.
28. The data type used to store a single character is _____.
29. The data type used to store a true or false value is _____.
30. The data type used to store a floating-point number is _____.
31. The operator used to add two numbers is _____.
32. The operator used to subtract one number from another is _____.
33. The operator used to multiply two numbers is _____.
34. The operator used to divide one number by another is _____.
35. The operator used to find the remainder of a division is _____.
36. The operator used to check if two values are equal is _____.
37. The operator used to check if one value is greater than another is _____.
38. The operator used to check if one value is less than or equal to another is
39. The operator used to perform a logical AND operation is _____.
40. The operator used to perform a logical OR operation is _____.
41. The operator used to perform a logical NOT operation is _____.
42. The operator used to assign a value to a variable is _____.
43. The operator used to add and assign a value to a variable is _____.
44. The operator used to subtract and assign a value to a variable is _____.

IX. In the table below ,Match every C++ terms with its corresponding description by writing the letter of C++ term after the number of its description in the Answers column

Answers	**Description**	**C++ terms**
……..	**1.** The keyword used to declare a constant variable is	**A.** Poinnter
……..	**2.** A variable that stores the address of another variable is called a ___	**B.** constant
……..	**3.** The operator used to add two numbers is _____.	**C.** = =
……..	**4.** The operator used to check if two values are equal is _____.	**D.** Recursive
……..	**5.** A function that calls itself is known as a _____ function.	**E.** +
……..	**6.** The keyword _____ is used to suggest to the compiler to expand the function inline.	**F.** 0
……..	**7.** The index of the first element in a C++ array is _____.	**G.** Inline

……	**8.** To initialize an array of 5 integers with values 1, 2, 3, 4, 5, you write _____	**H.** Local
……	**9.** A variable declared inside a function is called a _____ variable.	**I.** int arr[5] ={1, 2, 3, 4, 5};
……	**10.** The operator used to find the remainder of a division is _____	**J.**
……	**11.** The return type of a function that does not return a value is _____.	**K.** %
……	**12.** A two-dimensional array can be declared as _____.	**L.** Void

X. Read the following statement and answer by true if it is correct or false if it is not correct.

1. The #include directive is used to include standard libraries in a C++ program.
2. Variables in C++ must be declared before they are used.
3. The int data type in C++ can store decimal values.
4. The = operator is used for comparison in C++.
5. The if statement is used to execute a block of code only if a specified condition is true.
6. A for loop in C++ can be used to iterate over a range of values.
7. Functions in C++ can return multiple values.
8. Arrays in C++ can hold elements of different data types.
9. The main function is the entry point of a C++ program.
10. The && operator is used for logical AND operations in C++.
11. The **||** operator is used for logical OR operations in C++.
12. The **!** operator is used for logical NOT operations in C++.
13. The switch statement can be used to select one of many code blocks to be executed.
14. The break statement is used to exit a loop or switch statement prematurely.
15. The continue statement is used to skip the current iteration of a loop and proceed to the next iteration.
16. The return statement is used to exit a function and return a value to the caller.
17. The void keyword is used to declare a function that does not return a value.
18. The sizeof operator returns the size of a variable or data type in bytes.
19. The cin object is used to output data to the console.

20. The cout object is used to input data from the console.

Practical assessment

Create a C++ program to calculate a student's average grade from scores in five subjects. Use variables for the student's name and scores, store the scores in an array, and define a function to compute the average. Implement loops for input and total calculation, then display the student's name and average score.

END

References

Babb, J. (2017). *C++: A Beginner's Guide* (4th ed.). McGraw-Hill.

Barlow, J., & McGowan, C. (2016). *C++ Programming for Beginners*. CreateSpace Independent Publishing Platform.

Bjarne Stroustrup. (2014). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Bjarne Stroustrup. (2019). *The C++ Programming Language* (5th ed.). Addison-Wesley.

C++ Institute. (2020). *C++ Programming Essentials*. Retrieved from https://www.cppinstitute.org/

Cplusplus.com - C++ Language Tutorial: This tutorial explains C++ from its basics to more advanced features with practical examples.

Deitel, P. J., & Deitel, H. M. (2016). *C++: How to Program* (10th ed.). Pearson Education.

Dromey, R. G. (2018). *How to Solve It by Computer*. Prentice Hall.

Eckel, B. (2010). *Thinking in C++* (2nd ed.). Prentice Hall.

Gaddis, T. (2018). *Starting Out with C++: From Control Structures through Objects* (9th ed.). Pearson.

GeeksforGeeks - C++ Basics: This tutorial covers the fundamental concepts of C++ with examples and explanations.

Gibbons, J. (2015). *Data Structures and Algorithms in C++*. Cambridge University Press.

Griffiths, D. (2019). *C++ Programming: From Problem Analysis to Program Design* (5th ed.). Cengage Learning.

Here are some useful links to help you learn and apply basic C++ concepts:

Kochan, S. G. (2014). *Programming in C++* (4th ed.). Sams Publishing.

Lambert, K. (2018). *Fundamentals of C++ Programming* (3rd ed.). Cengage Learning.

LearnCpp.com: A free website dedicated to teaching modern C++ with detailed lessons and best practices.

Lippman, S. B., Lajoie, J. & Moo, B. E. (2013). *C++ Primer* (5th ed.). Addison-Wesley.

Savitch, W. J. (2016). *Absolute C++* (6th ed.). Pearson.

Schildt, H. (2018). *C++: The Complete Reference* (4th ed.). McGraw-Hill.

Sebesta, R. W. (2019). *Concepts of Programming Languages* (11th ed.). Pearson Education.

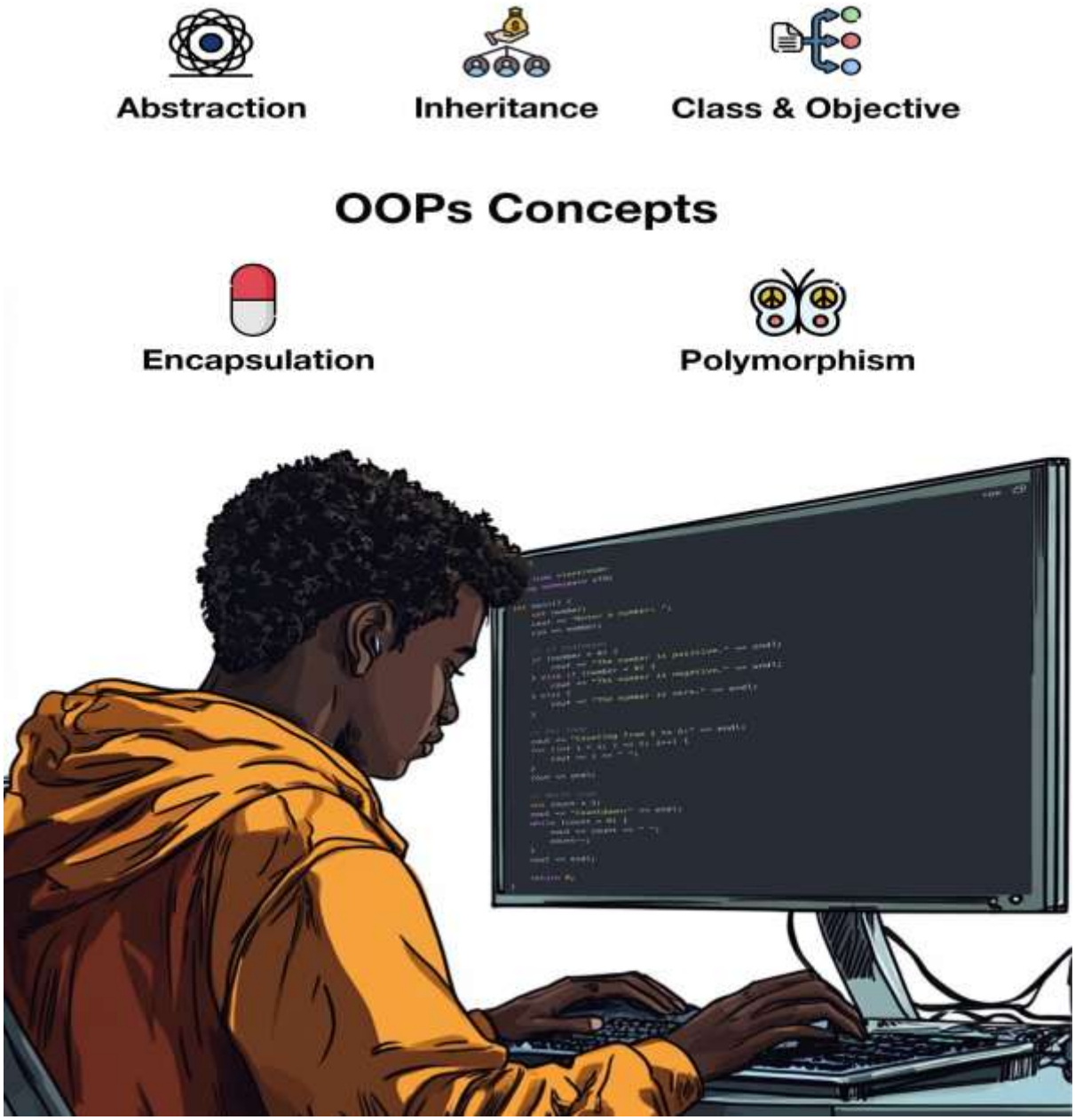
Simplilearn - C++ Basics: This guide provides an easy-to-understand introduction to C++ concepts.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

W3Schools - C++ Tutorial: A comprehensive guide to learning C++ with interactive examples and exercises.

Walther, C. (2020). *C++ Programming: A Practical Approach*. Independently published.

West, C. (2018). *C++ in 21 Days* (5th ed.). Sams Publishing.


Abstraction
Inheritance
Class & Objective
OOPs Concepts
Encapsulation
Polymorphism

Indicative contents

2.1. Apply Class and object
2.2. Apply Inheritance and polymorphism
2.3. Apply Namespaces
2.4. Errors and Exceptions handling

Key Competencies for Learning Outcome 2: APPLY OOP CONCEPTS

Knowledge	Skills	Attitudes
• Description of OOP key terms • description of advantages of using OOP in C++ • Description of class and object • Description of constructor and destructor • Description of Inheritance and polymorphism • Description of Errors and Exceptions handling • Description of namespaces	• Applying class and object • Implementing Inheritance and polymorphism • Performing of Errors and Exceptions handling • Applying namespace	• Being cooperative • Being Hard working • Being creative and innovative •

Duration: 30 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe properly OOP key terms as used in C++ programming language
2. Apply properly Class and object based on the principles of object-oriented programming.
3. Describe correctly Inheritance and polymorphism based on the rules and principles of object-oriented programming.
4. Apply properly Inheritance and polymorphism based on the rules and principles of object-oriented programming.
5. Describe correctly Namespaces based on the rules and conventions
6. Apply properly Namespaces based on the rules and conventions
7. Describe correctly Errors and Exceptions according to the rules and practices of error handling
8. Apply effectively Errors and Exceptions according to the rules and practices of error handling

Resources

Equipment	**Tools**	**Materials**
• Computer	• Integrated ○ Development Environment (IDE)	• Internet

Indicative content 2.1.: Apply Class and object

Duration: 10 hrs

Theoretical Activity 2.1.1: Description of OOP Key Terms

Tasks:

1: Read the questions below and answer them

I. Define the following terms as used in C++ OOP:

a. Class

b. Object

c. Encapsulation

d. Data Abstraction

e. Constructors

f. Destructors

g. Inheritance

h. Polymorphism

i. Function Overloading

j. Function Overriding

II. Describe Class and Object

2: Write the findings on paper /flipchart

3: Present your findings

4: Ask for clarification if any.

5: Read the Key readings **2.1.1**.for more clarification.

Key readings 2.1.1

Description of OOP Key Terms

1. Definition of key terms

OOP stands for Object-Oriented Programming.

✓ **Class**

A class is a blueprint for creating objects in C++. It defines a data type by bundling data (attributes) and methods (functions) that operate on the data. A class encapsulates related properties and behaviors into a single unit.

✓ **Object**

An object is an instance of a class. It represents a specific entity with the characteristics and behaviors defined by its class. Each object can hold its own data and state.

✓ **Encapsulation**

Encapsulation is the concept of restricting access to certain components of an object and exposing only the necessary parts. This is achieved through access specifiers (private, protected, public) in a class, which helps protect the internal state of the object from unintended interference.

✓ **Data Abstraction**

Data abstraction refers to the concept of hiding complex implementation details and showing only the essential features of an object. It allows programmers to focus on interactions at a higher level without needing to understand the intricate details of how the underlying data is managed.

✓ **Constructors and Destructors**

- **Constructors** are special member functions that are automatically called when an object is created. They are used to initialize the object's attributes.
- **Destructors** are also special member functions called when an object is destroyed. They are used to release resources or perform cleanup tasks before the object goes out of scope.

✓ **Inheritance**

Inheritance is a mechanism that allows a new class (derived class) to inherit properties and behaviors from an existing class (base class). It promotes code reusability and establishes a hierarchical relationship between classes.

✓ **Polymorphism**

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables one interface to be used for different underlying data types, often implemented through function overriding and function overloading.

✓ **Function Overloading**

Function overloading allows multiple functions to have the same name but differ in the number or type of their parameters. This enables different implementations based on the provided arguments, enhancing code readability and usability.

✓ **Function Overriding**

Function overriding occurs when a derived class provides a specific implementation of a function that is already defined in its base class. This allows the derived class to customize or extend the behavior of the base class method while maintaining the same function signature.

Look at the following illustration to see the difference between class and objects:

2. Advantages of using OOP

Using object-oriented programming (OOP) in C++ offers several specific advantages that enhance software design and development. Here are some key benefits:

✓ **Code Reusability**

C++ allows developers to create reusable classes through inheritance. Existing classes can serve as bases for new classes, reducing redundancy and speeding up development.

✓ **Modularity**

C++ supports modular programming by allowing developers to define classes and functions in separate files. This organization helps in managing large projects, making code easier to understand and maintain.

✓ **Encapsulation**

C++ provides strong encapsulation through access specifiers (private, protected, public). This helps protect the internal state of objects, ensuring that data can only be modified through well-defined interfaces, improving data integrity.

✓ **Abstraction**

C++ enables developers to abstract complex realities using classes and interfaces. This allows for high-level design while hiding the underlying complexity, making code easier to work with and understand.

✓ **Polymorphism**

C++ supports polymorphism through virtual functions, allowing objects of different classes to be treated as objects of a common base class. This capability enables dynamic method resolution, providing flexibility in code implementation.

✓ **Inheritance**

C++ allows for single and multiple inheritances, promoting the creation of a hierarchy of classes. This structure enhances code organization and facilitates the sharing of common functionality among related classes.

✓ **Improved Maintenance**

OOP principles in C++ make it easier to maintain and extend code. Changes to a class can be made without affecting other parts of the program, and modifications can be localized, reducing the risk of introducing bugs.

✓ **Enhanced Collaboration**

C++ OOP encourages collaborative development by allowing different team members to work on separate classes independently. This can streamline the development process and reduce integration issues.

✓ **Rich Standard Library**

C++ has a powerful standard library (STL) that makes extensive use of OOP concepts. This provides developers with pre-built classes and functions for various tasks, enhancing productivity.

✓ **Real-World Modeling**

C++ OOP allows for modeling real-world entities and relationships naturally, making it easier to design systems that reflect user requirements and business processes. This intuitive design can enhance understanding and communication.

3. Class

A class in C++ is a user-defined data type that allows you to encapsulate data and functions that operate on that data. It is a fundamental concept in object-oriented programming (OOP), providing a way to model real-world entities and their behaviors.

a. Class Definition

A class is defined using the class keyword, followed by the class name. The class contains data members (attributes) and member functions (methods), grouped by the access specifies.

✓ **Access Specifies**

Control the visibility of class members. The most common are:

✓ **Data Members**

These are variables that hold the state of an object. They can be of any data type, including built-in types, other classes, or even pointers.

✓ **Member Functions**

These are functions defined within the class that operate on the data members. They can manipulate the object's data or perform specific tasks.

✓ **Syntax**

Here's the basic syntax for defining a class:

```
class ClassName {
public:  // Access specifier
 // Data members (attributes)
 DataType member1;
DataType member2;
```

```
 // Member functions (methods)
      ReturnType function1(parameters);
 ReturnType function2(parameters);
private: // Access specifier
 // Private data members or functions
protected: // Access specifier
// Protected data members or functions
};
```

✓ **Example of a Class**

Here's an example to illustrate these concepts:

```
class Car {
private:
   // Data members
   std::string brand;
   std::string model;
   int year;
public:
   // Constructor
   Car(std::string b, std::string m, int y) : brand(b), model(m), year(y) {}
   // Method to display car details
   void displayInfo() {
      std::cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year
<< std::endl;
   }
   // Method to get the car's age
   int carAge(int currentYear) {
      return currentYear - year;
   }
};
```

b. Using the Class

Using a class in C++ involves creating an instance (object) of the class and accessing its members (both data members and member functions).

✓ **Creating an object**

Creating an object in C++ involves instantiating a class, which means allocating memory for it and invoking its constructor.

✓ **Syntax**

You can create an object using the following syntax:

ClassName objectName;

Or, if you want to initialize it with specific values using a constructor:

ClassName objectName(parameters);

✓ **Example: Creating Objects**

```
int main() {
   // Create an object of the Car class
   Car car1("Toyota", "Corolla", 2020);
   // Display car details
   car1.displayInfo();
   // Calculate and display car age
   int currentYear = 2024;
   std::cout << "Car Age: " << car1.carAge(currentYear) << " years" << std::endl;
   return 0;
}
```

c. Way of Creating Objects

In C++, there are several ways to create objects of a class. Each method has its own use cases depending on the context.

Here are the main ways to create objects:

- **Stack Allocation**

Objects can be created on the stack. This is the most common way to create objects and is automatically managed by the compiler.

✓ **Syntax:**

ClassName objectName;

✓ Example:

```
class MyClass {
public:
   MyClass() { std::cout << "Constructor called!" << std::endl; }
};
int main() {
   MyClass obj; // obj is created on the stack
   return 0; // obj is destroyed when it goes out of scope
}
```

- **Heap Allocation**

Objects can also be created on the heap using the new keyword. This gives you more control over the object's lifetime, but you must manually manage memory (using delete to free it).

✓ **Syntax:**

ClassName* objectName = new ClassName();

✓ **Example:**

```
int main() {
   MyClass* obj = new MyClass(); // obj is created on the heap
   // Use the object
   delete obj; // Free the memory
   return 0;
}
```

- **Using Constructors with Parameters**

You can create objects using constructors that take parameters, allowing you to initialize data members.

✓ **Example:**

```
class MyClass {
private:
   int value;
public:
   MyClass(int v) : value(v) { std::cout << "Value set to " << value << std::endl; }
};
int main() {
   MyClass obj(10); // Stack allocation with parameter
   MyClass* objPtr = new MyClass(20); // Heap allocation with parameter
   delete objPtr; // Free heap memory
   return 0;
}
```

- **Default and Copy Constructors**

Objects can be created using default constructors or copy constructors.

✓ **Example:**

```
class MyClass {
public:
   MyClass() { std::cout << "Default Constructor called!" << std::endl; }
   MyClass(const MyClass &obj) { std::cout << "Copy Constructor called!" << std::endl; }
};
int main() {
   MyClass obj1; // Default constructor
   MyClass obj2 = obj1; // Copy constructor
   return 0;
```

```
}
```

- **Using an Array of Objects**

You can create an array of objects. Each object in the array can be individually initialized.

✓ **Example:**

```
class MyClass {
public:
    MyClass() { std::cout << "Constructor called!" << std::endl; }
};
int main() {
    MyClass objArray[3]; // Array of objects (stack allocation)
    MyClass* objPtrArray = new MyClass[3]; // Array of objects (heap allocation)
    delete[] objPtrArray; // Free memory for the heap array
    return 0;
}
```

4. Access specifies

In C++, access specifiers control the visibility and accessibility of class members (data members and member functions). They play a crucial role in implementing encapsulation, one of the fundamental principles of object-oriented programming. There are three main access specifiers in C++:

a. Public

Definition: Members declared as public are accessible from outside the class.

Use Case: Use public access for methods and data members that need to be accessible to users of the class.

✓ **Example:**

```
class MyClass {
public:
    int value;
    void display() {
```

```
        std::cout << "Value: " << value << std::endl;
    }
};
int main() {
    MyClass obj;
    obj.value = 10; // Accessible
    obj.display();  // Accessible
    return 0;
}
```

b. Private

Definition: Members declared as private are accessible only within the class itself. They cannot be accessed from outside the class or by derived classes.

Use Case: Use private access for data members and helper functions that should not be exposed to the user, maintaining encapsulation.

✓ **Example**:

```
class MyClass {
private:
    int value;
public:
    void setValue(int v) {
        value = v; // Accessible within the class
    }
    void display() {
        std::cout << "Value: " << value << std::endl; // Accessible within the class
    }
};
int main() {
    MyClass obj;
```

```
    obj.setValue(10); // Allowed
    obj.display();    // Allowed
    // obj.value = 10; // Not allowed (private access)
    return 0;
}
```

c. Protected

Definition: Members declared as protected are accessible within the class and by derived classes, but not from outside the class hierarchy.

Use Case: Use protected access when you want to allow derived classes to access certain members while keeping them hidden from other classes.

✓ **Example:**

```
class Base {
protected:
    int value;
public:
    void setValue(int v) {
        value = v; // Accessible within Base
    }
};
class Derived : public Base {
public:
    void display() {
        std::cout << "Value: " << value << std::endl; // Accessible in Derived
    }
};
int main() {
    Derived obj;
    obj.setValue(10); // Allowed
```

```
    obj.display();    // Allowed
    // obj.value = 10; // Not allowed (protected access)
    return 0;
}
```

d. Summary of Access Specifiers

Specifier	Accessibility	Use Case
public	Accessible from anywhere (outside and inside)	For members that need to be accessible to all.
Private	Accessible only within the class	For members that should be hidden from users.
protected	Accessible within the class and derived classes	For members that should be accessible in derived classes but hidden from others.

5. Constructor and destructors

a. Overview

Constructors and destructors are special member functions in C++ that manage the initialization and cleanup of objects, respectively. They play a crucial role in resource management, object lifecycle, and ensuring that objects are properly constructed and destructed.

b. Importance

Using constructors and destructors properly helps:

✓ Ensure that objects are in a valid state when created.

✓ Manage dynamic resources effectively, preventing memory leaks and resource wastage.

✓ Implement RAII (Resource Acquisition Is Initialization) principles, which are crucial for effective resource management in C++.

c. Constructors

✓ **Definition**

A constructor is a special member function that is automatically called when an object of a class is created. Its primary role is to initialize the object's data members.

✓ **Characteristics**

Name: A constructor has the same name as the class.

No Return Type: Constructors do not have a return type, not even void .

Can be Overloaded: You can have multiple constructors in a class with different parameter lists (overloading).

✓ **Default Constructor:** A constructor that takes no parameters.

✓ **Parameterized Constructor:** A constructor that takes one or more parameters.

Example

```
#include <iostream>

#include <string>

class Person {

private:

   std::string name;

   int age;

public:

   // Default constructor

   Person() : name("Unknown"), age(0) {

      std::cout << "Default constructor called!" << std::endl;

   }

   // Parameterized constructor

   Person(std::string n, int a) : name(n), age(a) {

      std::cout << "Parameterized constructor called!" << std::endl;

   }

   // Method to display information
```

```
    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
int main() {
    Person person1;                 // Calls default constructor
    Person person2("Alice", 30);    // Calls parameterized constructor
    person1.display();
    person2.display();
    return 0;
}
```

d. Destructors

✓ **Definition**

A destructor is a special member function that is automatically called when an object is destroyed (goes out of scope or is explicitly deleted). Its main role is to free resources that were acquired by the object during its lifetime.

✓ **Characteristics**

Name: A destructor has the same name as the class, preceded by a tilde (~).

No Parameters: Destructors do not take any parameters and cannot be overloaded.

One Per Class: There can only be one destructor for each class.

Example

```
class Person {
private:
    std::string name;
    int age;
public:
    Person(std::string n, int a) : name(n), age(a) {
```

```
        std::cout << "Constructor called for " << name << std::endl;
    }
    // Destructor
    ~Person() {
        std::cout << "Destructor called for " << name << std::endl;
    }
};
int main() {
    Person person1("Alice", 30); // Constructor called
    {
        Person person2("Bob", 25); // Constructor called
    } // Destructor called for person2
    return 0; // Destructor called for person1  }
```

- **Copy Constructor**

A copy constructor is a special constructor in C++ used to create a new object as a copy of an existing object. It is called when an object is passed by value, returned from a function, or explicitly copied.

```
class MyClass {
public:
    int value;
    MyClass(int v) : value(v) {} // Regular constructor
    MyClass(const MyClass &obj) { // Copy constructor
        value = obj.value;
   }};
```

- **Deep Copy**

A deep copy creates a new object and copies all fields, and also allocates separate memory for any dynamically allocated memory in the original object. This ensures that the new object is completely independent of the original.

```
#include <iostream>
#include <cstring>
class MyString {
private:
   char* data;
public:
   // Constructor
   MyString(const char* source) {
      if (source) {
         data = new char[strlen(source) + 1];
         strcpy(data, source);
      } else {
         data = nullptr;
      } }
   // Copy Constructor for Deep Copy
   MyString(const MyString& source) {
      if (source.data) {
         data = new char[strlen(source.data) + 1];
         strcpy(data, source.data);
      } else {
         data = nullptr;
      }
   }
   // Destructor
   ~MyString() {
      delete[] data;  }
   // Display function
```

```
    void display() const {
        if (data) {
            std::cout << data << std::endl;
        } else {
            std::cout << "Empty string" << std::endl;
        }
    }
};
int main() {
    MyString str1("Hello, World!");
    MyString str2 = str1; // Deep copy
    str1.display();
    str2.display();
    return 0;
}
```

In the example above :

The **MyString** class has a constructor that allocates memory for the **string**.

The copy constructor performs a deep copy by allocating new memory and copying the content of the source string.

The destructor ensures that the allocated memory is properly deleted to avoid memory leaks.This way, **str1** and **str2** are independent copies, and changes to one will not affect the other.

- **Shallow Copy**

A shallow copy copies all fields of the original object to the new object. However, it does not allocate new memory for dynamically allocated fields; instead, it copies the references. This can lead to issues if one object modifies the shared data.

a. Key Points of Shallow Copy

- Copies Immediate Data Members: Only the values of the object's data members are copied.

- Shared Pointers: If the object contains pointers, the copied object will point to the same memory locations as the original object.
- Potential Issues: This can lead to problems such as double deletion (when both objects try to delete the same memory) or unintended modifications (changes in one object affect the other).

b. Example

Here's a simple example to illustrate a shallow copy in C++:

```
#include <iostream>

class Shallow {

private:

   int* data;

public:

   // Constructor

   Shallow(int d) {

      data = new int;

      *data = d;

   }

   // Copy Constructor (Shallow Copy)

   Shallow(const Shallow& source) : data(source.data) {

      std::cout << "Shallow copy constructor - shallow copy" << std::endl;

   }

   // Destructor

   ~Shallow() {

      delete data;

      std::cout << "Destructor freeing data" << std::endl;

   }

   // Display function

   void display() const {

      std::cout << "Data: " << *data << std::endl;
```

```
    }
};
int main() {
    Shallow obj1(100);
    Shallow obj2 = obj1; // Shallow copy
    obj1.display();
    obj2.display();
    return 0;
}
```

In this example:

The **Shallow** class has a constructor that allocates memory for an integer.

The copy constructor performs a shallow copy by copying the pointer value, not the actual data.

Both **obj1** and **obj2** point to the same memory location. Therefore, changes to the data through one object will reflect in the other.

e. Static members

- A **static member variable**

In C++ is a variable that is shared among all instances of a class. Unlike regular member variables, which are unique to each instance of a class, static member variables are common to all instances. This means that if one instance changes the value of a static member variable, the change is reflected across all instances of that class.

✓ **Key Characteristics of Static Member Variables**

- Shared Across Instances**:** All objects of the class share the same static member variable.
- Class-Level Scope**:** Static member variables are associated with the class itself, not with any particular object.
- Lifetime**: They** exist for the duration of the program, from the start to the end.

✓ **Example**

Here's an example to illustrate the concept:

```
#include <iostream>
```

```
class Example {
public:
   static int count; // Declaration of static member variable
   Example() {
      count++;
   }
   static void displayCount() {
      std::cout << "Count: " << count << std::endl;
   }
};
// Definition and initialization of static member variable
int Example::count = 0;
int main() {
   Example obj1;
   Example obj2;
   Example::displayCount(); // Accessing static member function
   return 0;
}
```

In this example:

Count is a static member variable of the Example class.

It is incremented each time an Example object is created.

The displayCount static member function can access the static member variable count.

✓ **Important Points**

Initialization: Static member variables must be defined and initialized outside the class definition.

Access: They can be accessed using the class name and the scope resolution operator (ClassName::variableName).

- A **static member function**

In C++ is a function that belongs to the class itself rather than to any particular object instance. This means it can be called without creating an instance of the class. Static member functions have several unique properties:

✓ **Key Characteristics of Static Member Functions:**

- **Class-Level Scope**: They are associated with the class, not with any specific object.
- **No** this **Pointer**: Since they are not tied to any object, they do not have access to the this pointer.
- **Access to Static Members**: They can access static member variables and other static member functions directly.
- **Cannot Access Non-Static Members**: They cannot access non-static member variables or functions directly because they do not have an instance to operate on.

✓ **Example**

```
#include <iostream>

class Example {

private:

 static int count; // Static member variable

public:

   Example() {

      count++;

   }

   // Static member function

   static void displayCount() {

      std::cout << "Count: " << count << std::endl;

   }

};

// Definition and initialization of static member variable

int Example::count = 0;

int main() {
```

```
    Example obj1;
    Example obj2;
    // Calling static member function using class name
    Example::displayCount();
    return 0;
}
```

I**n this example:**

count is a static member variable that keeps track of the number of Example objects created.

displayCount is a static member function that prints the value of count.

The static member function is called using the class name Example::displayCount().

✓ **Important Points:**

Initialization: Static member function must be defined and initialized outside the class definition.

Access: Static member functions can be called using the class name and the scope resolution operator (ClassName::functionName()).

f. Constants members

- **A constant member variable**

In C++ is a member variable whose value cannot be changed once it is initialized. This is achieved by using the const keyword.

Constant member variables must be initialized at the time of their declaration, typically using an initializer list in the constructor.

✓ **Key Characteristics of Constant Member Variables**

- Immutability: Once initialized, their value cannot be modified.
- Initialization: Must be initialized in the constructor's initializer list.
- Scope: They are specific to each instance of the class, unlike static member variables.

- **A constant member function**

In C++ is a member function that does not modify any member variables of the class. This is enforced by adding the const keyword at the end of the function declaration.

Constant member functions can be called on both const and non-const objects, ensuring that they do not alter the state of the object.

✓ **Key Characteristics of Constant Member Functions**

- **Immutability**: They guarantee not to modify any member variables of the class.
- **Const Correctness:** They can be called on const instances of the class.

✓ **Syntax:**

Declared with the const keyword at the end of the function declaration.

Here's an example to illustrate the concept:

```
#include <iostream>

class Example {

private:

   int value;

public:

   Example(int v) : value(v) {}

   // Constant member function

   int getValue() const {

      return value;

   }

   // Non-constant member function

   void setValue(int v) {

      value = v;

   }

};

int main() {
```

```
    const Example obj(10);
    // Calling constant member function on a const object
    std::cout << "Value: " << obj.getValue() << std::endl;
    // The following line would cause a compilation error
    // obj.setValue(20);
return 0;
}
```

In this example:

getValue is a constant member function, which means it can be called on a const object and guarantees not to modify any member variables.

setValue is a non-constant member function and cannot be called on a const object.

✓ **Important Points:**

- **Const Correctness**: Ensures that functions that should not modify the object are correctly marked as const.
- **Usage**: Helps in maintaining the integrity of objects, especially when dealing with constant instances.

Practical Activity 2.1.2: APPLYING CLASS AND OBJECT

Task:

1 Read and perform the task below:

Create a BankAccount class to manage basic banking operations. The class should include private data members for account number, account holder's name, and balance. Implement public member functions: deposit() to add funds, withdraw() to deduct funds if sufficient(not less than 1000), and display() to show account details. Define the class and its functions, then create and test objects to ensure all functionalities work correctly.

2: Read the Key readings **2.1.2**

3: Explore steps demonstrated by trainer to create and execute a C++ program that simulates simple banking system

4: Compile and run the program and ask for assistance if needed.

6: Check the output of the program

Key readings 2.1.2

Applying class and object

1. Steps to implement class and object

Here are the steps to implement a class and create an object in C++:

Step 1: Define the Class

✓ Choose a Class Name: Decide on a meaningful name for your class.

✓ Declare Data Members: Identify the attributes (variables) that the class will contain.

✓ Declare Member Functions: Define functions that will operate on the class data.

✓ Access Specifiers: Use public, private, or protected to control access to class members.

Example:

```
class BankAccount {
private:
   std::string accountNumber;
   std::string accountHolderName;
   double balance;
public:
   // Constructor
   BankAccount(std::string accNumber, std::string holderName, double initialBalance)
```

```
        : accountNumber(accNumber), accountHolderName(holderName),
balance(initialBalance) {
        if (initialBalance < 1000) {
            std::cout << "Initial balance must be at least 1000!" << std::endl;
            balance = 1000; // Set to minimum if less
        }
    }
    // Deposit function
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: $" << amount << std::endl;
        } else {
            std::cout << "Deposit amount must be positive." << std::endl;
        }
    }
    // Withdraw function
    void withdraw(double amount) {
        if (amount > 0 && (balance - amount) >= 1000) {
            balance -= amount;
            std::cout << "Withdrew: $" << amount << std::endl;
        } else {
            std::cout << "Withdrawal denied. Minimum balance must be $1000." <<
std::endl;
        }
    }
    // Display account details
    void display() const {
```

```
        std::cout << "Account Number: " << accountNumber << std::endl;
        std::cout << "Account Holder: " << accountHolderName << std::endl;
        std::cout << "Balance: $" << balance << std::endl;
    }
};
```

Step 2: Implement Member Functions

✓ Define the Constructor: Initialize data members in the constructor.

✓ Define Member Functions: Implement the functions declared in the class.

Example:

```
    BankAccount(std::string accNumber, std::string holderName, double initialBalance)
        : accountNumber(accNumber), accountHolderName(holderName), balance(initialBalance) {
        if (initialBalance < 1000) {
            std::cout << "Initial balance must be at least 1000!" << std::endl;
            balance = 1000; // Set to minimum if less
        }
    }
```

Step 3: Create Objects

✓ In the main() Function: Create instances (objects) of the class using the constructor.

Example:

```
int main() {
    Car myCar("Toyota", 2020);  // Creating an object
    myCar.displayInfo();        // Calling a member function
    return 0;
```

}

Step 4: Compile and Run the Program

✓ Use a C++ Compiler: Compile your code using an IDE.

✓ Check for Errors: Ensure there are no syntax or logical errors.

✓ Run the Executable: Execute the compiled program to see the output.

Complete Example

```
#include <iostream>

#include <string>

class BankAccount {

private:

   std::string accountNumber;

   std::string accountHolderName;

   double balance;

public:

   // Constructor

   BankAccount(std::string accNumber, std::string holderName, double
initialBalance)

      : accountNumber(accNumber), accountHolderName(holderName),
balance(initialBalance) {

      if (initialBalance < 1000) {

         std::cout << "Initial balance must be at least 1000!" << std::endl;

         balance = 1000; // Set to minimum if less

      }

   }

   // Deposit function

   void deposit(double amount) {
```

```
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: $" << amount << std::endl;
        } else {
            std::cout << "Deposit amount must be positive." << std::endl;
        }
    }
    // Withdraw function
    void withdraw(double amount) {
        if (amount > 0 && (balance - amount) >= 1000) {
            balance -= amount;
            std::cout << "Withdrew: $" << amount << std::endl;
        } else {
            std::cout << "Withdrawal denied. Minimum balance must be $1000." <<
std::endl;
        }
    }
    // Display account details
    void display() const {
        std::cout << "Account Number: " << accountNumber << std::endl;
        std::cout << "Account Holder: " << accountHolderName << std::endl;
        std::cout << "Balance: $" << balance << std::endl;
    }
};
int main() {
    // Create a BankAccount object
    BankAccount account("123456", "John Doe", 1500);
```

```
    // Display initial account details
    account.display();
    // Test deposit
    account.deposit(500);
    account.display();
    // Test withdrawal
    account.withdraw(700);
    account.display();
    // Try to withdraw more than allowed
    account.withdraw(1000); // This should fail
    // Test insufficient initial balance scenario
    BankAccount account2("654321", "Jane Smith", 500); // Should set to
minimum balance
    // Display account details for the second account
    account2.display();
    return 0;
}
```

Points to Remember

Description of key terms of OOP

Definition of Key Terms

- ✓ **OOP:** Object-Oriented Programming, a programming paradigm based on the concept of "objects."
- ✓ **Class:** A blueprint for creating objects, bundling data (attributes) and methods (functions) that define behaviors.
- ✓ **Object:** An instance of a class, representing a specific entity with its own data and state.
- ✓ **Encapsulation:** Restricting access to certain components of an object, exposing only necessary parts through access specifiers (private, protected, public).
- ✓ **Data Abstraction:** Hiding complex implementation details while exposing essential features, allowing higher-level interactions.
- ✓ **Constructors/Destructors:**
- ✓ **Constructors:** Automatically called when an object is created to initialize attributes.
- ✓ **Destructors:** Called when an object is destroyed to release resources.
- ✓ **Inheritance:** A mechanism for a new class (derived class) to inherit properties and behaviors from an existing class (base class), promoting code reusability.
- ✓ **Polymorphism:** Treating objects of different classes as objects of a common base class, enabling flexible interfaces through function overriding and overloading.
- ✓ **Function Overloading:** Multiple functions with the same name but different parameters, enhancing usability.
- ✓ **Function Overriding:** A derived class providing a specific implementation of a function defined in its base class.
- ✓ **Advantages of OOP**
- ✓ **Code Reusability:** Promotes creating reusable classes through inheritance.
- ✓ **Modularity:** Facilitates modular programming, making code easier to manage.
- ✓ **Encapsulation:** Protects the internal state of objects, ensuring data integrity**.**
- ✓ **Abstraction:** Hides complexity, simplifying interactions.

-Polymorphism: Allows dynamic method resolution.

-Inheritance: Supports hierarchical class structures for better organization.

Improved Maintenance: Easier to maintain and extend code without widespread changes.
Enhanced Collaboration: Different team members can work on separate classes independently.
Rich Standard Library: C++ STL provides pre-built classes for various tasks.
Real-World Modeling: Models real-world entities, enhancing design intuitiveness.

- ✓ **Class:**

It is a user-defined data type in C++ that encapsulates data and functions.

Access Specifiers: Control visibility (public, private, protected).
Data Members: Variables holding an object's state.
Member Functions: Functions operating on data members.

- ✓ **Access Specifiers:**

Public: Accessible from anywhere.
Private: Accessible only within the class.
Protected: they are accessible within the class and by derived classes.

- ✓ **Constructors and Destructors:**

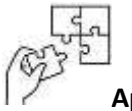
Constructors: Initialize objects; can be default or parameterized.
Destructors: Free resources when objects are destroyed; cannot be overloaded and only one per class

- ✓ A copy constructor is a special constructor in C++ used to create a new object as a copy of an existing object.
- ✓ A deep copy creates a new object and copies all fields, and also allocates separate memory for any dynamically allocated memory in the original object
- ✓ A shallow copy copies all fields of the original object to the new object. However, it does not allocate new memory for dynamically allocated fields; instead, it copies the references
- ✓ **A** static member variable in C++ is a variable that is shared among all instances of a class
- ✓ A constant member function in C++ is a member function that does not modify any member variables of the class.

Applying class and object

When one needs to apply class and object, has to:

- ✓ Define the class with data members and member functions.
- ✓ Implement the class by defining its functions.
- ✓ Create objects of the class in the main() function.
- ✓ Compile and run your program to see results.



Applicationo of learning 2.1:

Task:

Create a class called Person with private member variables for name, age, and country. Implement member functions to set and get the values of these variables.

Indicative content 2.2: Apply Inheritance and Polymorphism

Duration: 8 hours

Theoretical Activity 2.2.1: Description of Inheritance and polymorphism

Task

1: Read the tasks below and provide the answer

1. Describe the following terms

a) Inheritance
b) Polymorphism
c) Discuss the following:
d) Base class
e) Access specifier
f) Derived class

2. Differentiate Function Overriding from function Overloading

3: write your findings on the paper/flipchart

4: Presents your findings

5: Ask for more clarification if any

6: Read the Key readings **2.2.1** in trainee manual.

Key readings 2.2.1.:

Description of inheritance and polymorphism

1. Inheritance:

Inheritance allows a class (derived class) to inherit properties and methods from another class (base class).

a. Example:

```
class Base {
public:
   int baseVar;
};
class Derived : public Base {
public:
   int derivedVar;
};
```

b. Syntax:

The syntax for defining a class and inheritance in C++:

```
class Base {
   // Base class members
};
class Derived : public Base {
   // Derived class members
};
```

c. Types of Inheritance:

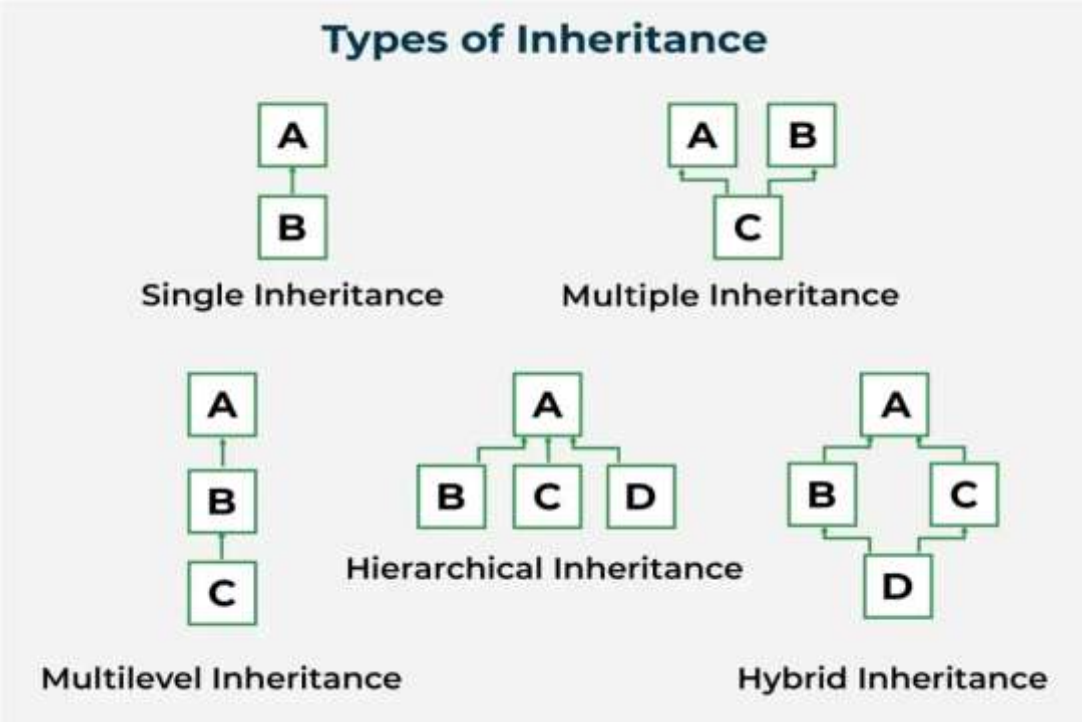
Single Inheritance: A derived class inherits from a single base class.

Multiple Inheritance : A derived class inherits from more than one base class.

Multilevel Inheritance: A derived class is derived from another derived class.

Hierarchical Inheritance: Multiple derived classes inherit from a single base class.

Hybrid Inheritance: A combination of two or more types of inheritance



- **Access Modifiers or Access Specifiers**

In a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

- **Public Inheritance**:

Public and protected members of the base class become public and protected members of the derived class, respectively.

- **Protected Inheritance**:

Public and protected members of the base class become protected members of the derived class.

- **Private Inheritance**:

Public and protected members of the base class become private members of the derived class.

- **Relationship Between Base and Derived Classes**:

The derived class inherits members from the base class, allowing reuse and extension of functionality.

- **Create a Derived Class from a Base Class**:

Example:

```
class Animal {
public:
   void eat() {
      cout << "Eating" << endl;
   }
};
class Dog : public Animal {
public:
   void bark() {
      cout << "Barking" << endl;
   }
};
```

Relationsh between a base class and a derived class in C++:

Aspect	**Base Class (**Animal**)**	**Derived Class (**Dog**)**
Definition	A class from which other classes are derived.	A class that inherits properties and behaviors from the base class.
Purpose	Provides common attributes and methods.	Extends or modifies the functionality of the base class.

Example	class Animal { ... };	class Dog : public Animal { ... };
Data Members	Can have data members (e.g., int age;).	Inherits data members from the base class and can have additional ones.
Member Functions	Can have member functions (e.g., void eat();).	Inherits member functions from the base class and can have additional ones.
Access Specifiers	Controls visibility of members (public, protected, private).	Inherits access specifiers and can define its own.
Inheritance	Not applicable.	Inherits members from the base class.
Overriding	Not applicable.	Can override base class methods.
Constructor	Initializes the base class.	Calls the base class constructor first, then its own.

Destructor	Cleans up resources for the base class.	Calls its own destructor first, then the base class destructor.
Example Code	```cpp class Animal { public: void eat() { cout << “Eating…” << endl; } };	```cpp class Dog : public Animal { public: void bark() { cout << “Barking…” << endl; } };

- **Adding a Property to an Object**:

  Properties (or attributes) are added as member variables.

  Example:

  class Car {

  public:

  string color;

  };

  To add a property to an object in C++, you typically define the property within a class and then create an object of that class.

  Steps:

Step	Description
1. Define a Class	Create a class with data members (properties) and member functions (methods).

2. Create an Object	Instantiate an object of the class.
3. Add Properties	Assign values to the object's properties using the dot operator (.).
4. Access Properties	Access and use the object's properties.

- **Complete Example:**

```
#include <iostream>
using namespace std;
// Step 1: Define a Class
class Car {
public:
   string brand;
   int year;
};
int main() {
   // Step 2: Create an Object
   Car myCar;
   // Step 3: Add Properties
   myCar.brand = "Toyota";
   myCar.year = 2020;
   // Step 4: Access Properties
   cout << myCar.brand << " " << myCar.year << endl;
   return 0;
}
```

- **Explanation:**

Define a Class: The Car class is defined with two properties: brand and year.

Create an Object: An object myCar of the Car class is created.

Add Properties: The properties brand and year are assigned values.

Access Properties: The values of the properties are accessed and printed.

- **Adding a Method to an Object**:

  Methods (or functions) are added as member functions.

  Example:

```
class Car {
public:
   void drive() {
      cout << "Driving" << endl;
   }
};
```

  To add a method to an object in C++, you define the method within a class and then create an object of that class.

  Steps:

- **Complete Example:**

Step	Description
1. Define a Class	Create a class with data members (properties) and member functions (methods).
2. Define the Method	Define the method inside or outside the class definition.
3. Create an Object	Instantiate an object of the class.
4. Add Properties	Assign values to the object's properties using the dot operator (.).
5. Call the Method	Use the dot operator to call the method on the object.

Here's a complete example that follows these steps:

```
#include <iostream>
using namespace std;
// Step 1: Define a Class
class Car {
public:
   string brand;
   // Step 2: Define the Method
   void displayBrand() {
      cout << "Brand: " << brand << endl;
   }
};
int main() {
```

```
    // Step 3: Create an Object
    Car myCar;
    // Step 4: Add Properties
    myCar.brand = "Toyota";
    // Step 5: Call the Method
    myCar.displayBrand();
    return 0;
}
```

Explanation:

Define a Class: The Car class is defined with a property brand and a method displayBrand.

Define the Method: The displayBrand method is defined to print the brand of the car.

Create an Object: An object myCar of the Car class is created.

Add Properties: The brand property is assigned a value.

Call the Method: The displayBrand method is called on the myCar object to print the brand.

3. **Friend Classes and Methods**:

Friend classes and methods can access private and protected members of another class.

Example:

```
class B;
class A {
private:
    int a;
public:
    A() : a(0) {}
    friend class B;
```

```
};

class B {

public:

  void showA(A& x) {

    cout << "A::a = " << x.a << endl;

  }

};
```

In C++, a friend class is a class that is given access to the private and protected members of another class. This is useful when you want to allow one class to access the internal details of another class without exposing those details to the rest of the program

- **How to Create a Friend Class:**

  Steps:

Step	Description
1. Define the Base Class	Create a class with private and protected members.
2. Define the Friend Class	Define a class that will access the private and protected members of the base class.
3. Use the Friend Class	Create objects of both classes and use the friend class to access the base class members.

4. **Polymorphism**:

   Polymorphism: The ability of a function, object, or method to take on many forms

   a. Types of Polymorphism

   - **Compile-Time Polymorphism (Static Polymorphism)**:

   Achieved through function overloading and operator overloading.

The decision about which function to call is made at compile time.

- **Function Overloading Example**:

```
class Print {
public:
   void show(int i) {
      cout << "Integer: " << i << endl;
   }
   void show(double d) {
      cout << "Double: " << d << endl;
   }
   void show(string s) {
      cout << "String: " << s << endl;
   }
};
int main() {
   Print obj;
   obj.show(5);       // Calls show(int)
   obj.show(3.14);    // Calls show(double)
   obj.show("Hello"); // Calls show(string)
   return 0;
}
```

- **Run-Time Polymorphism (Dynamic Polymorphism)**:

Achieved through inheritance and virtual functions.

The decision about which function to call is made at runtime.

Pointers and References in Polymorphism

In C++, pointers and references play a crucial role in achieving polymorphism, particularly run-time polymorphism. Here's a detailed look at how they are used:

Pointers:

Definition: A pointer is a variable that holds the memory address of another variable.

Usage in Polymorphism: Pointers to base class objects can be used to refer to derived class objects. This allows the base class pointer to call the overridden methods in the derived class, enabling polymorphism.

```
class Animal {
public:
   virtual void sound() {
      cout << "Some generic animal sound" << endl;
   }
};
class Dog : public Animal {
public:
   void sound() override {
      cout << "Bark" << endl;
   }
};
int main() {
   Animal* animalPtr;
   Dog dog;
   animalPtr = &dog;
   animalPtr->sound();  // Outputs: Bark
   return 0;
}
```

In this example, animalPtr is a pointer to the base class Animal, but it points to an object of the derived class Dog. The sound method of Dog is called, demonstrating polymorphism.

References:

Definition: A reference is an alias for an already existing variable. It is similar to a constant pointer with automatic dereferencing.

Usage in Polymorphism: References to base class objects can be used to refer to derived class objects. This allows the base class reference to call the overridden methods in the derived class, enabling polymorphism.

```
class Animal {
public:
   virtual void sound() {
      cout << "Some generic animal sound" << endl;
   }
};
class Cat : public Animal {
public:
   void sound() override {
      cout << "Meow" << endl;
   }
};
int main() {
   Cat cat;
   Animal& animalRef = cat;
   animalRef.sound();  // Outputs: Meow
   return 0;
}
```

In this example**,**

animalRef is a reference to the base **class Animal**, but it refers to an object of the derived class **Cat**. The sound method of **Cat** is called, demonstrating polymorphism.

Key Differences Between Pointers and References

Table

Aspect	Pointers	References
Initialization	Can be initialized at any time.	Must be initialized when declared.
Reassignment	Can be reassigned to point to different objects.	Cannot be reassigned once initialized.
Nullability	Can be assigned NULL or nullptr.	Cannot be null; must always refer to a valid object.
Memory Address	Has its own memory address and size on the stack.	Shares the same memory address with the original variable.
Indirection	Requires explicit dereferencing using the * operator.	Automatically dereferenced by the compiler.

Summary

Pointers and **references** are essential for implementing polymorphism in C++.

Pointers allow dynamic binding and can be reassigned, while **references** provide a more straightforward syntax and cannot be null or reassigned.

Both enable base class pointers or references to call derived class methods, achieving polymorphism.

Virtual Function Example:

```
class Animal {
public:
    virtual void sound() {
        cout << "Some generic animal sound" << endl;
```

```
    }
};
class Dog : public Animal {
public:
    void sound() override {
        cout << "Bark" << endl;
    }
};
class Cat : public Animal {
public:
    void sound() override {
        cout << "Meow" << endl;
    }
};
int main() {
    Animal* animal;
    Dog dog;
    Cat cat;
    animal = &dog;
    animal->sound();  // Outputs: Bark

    animal = &cat;
    animal->sound();  // Outputs: Meow
    return 0;
}
```

Key Points

Function Overloading: Multiple functions with the same name but different parameters.

Operator Overloading: Giving special meaning to an operator for user-defined data types.

Virtual Functions: Functions in a base class that can be overridden in derived classes to achieve run-time polymorphism.

Inheritance: Allows derived classes to inherit properties and methods from a base class, enabling polymorphism.

Pointers and References:

Pointers store memory addresses, references are aliases for variables.

Example:

```
int a = 10;

int* ptr = &a;

int& ref = a;
```

Function Overloading:

Multiple functions with the same name but different parameters.

Example:

```
void print(int i) {
   cout << "Integer: " << i << endl;
}

void print(double f) {
   cout << "Float: " << f << endl;
}
```

Function Overriding:

Redefining a base class method in a derived class.

Example:

```
class Base {
```

```
public:
    virtual void show() {
        cout << "Base" << endl;
    }
};
class Derived : public Base {
public:
    void show() override {
        cout << "Derived" << endl;
    }
};
```

Virtual Functions:

Functions in a base class that can be overridden in derived classes.

Example:

```
class Base {
public:
    virtual void display() {
        cout << "Base display" << endl;
    }
};
class Derived : public Base {
public:
    void display() override {
        cout << "Derived display" << endl;
    }
};
```

5. **Abstract class**

- **Definition**

Abstract classes contain at least one pure virtual function. An abstract class in C++ is a class that cannot be instantiated on its own and is designed to be a base class for other classes.

It contains at least one pure virtual function, which is a function declared with the = 0 syntax. This means the function has no implementation in the abstract class and must be overridden in any derived class.

Example:

```
class Abstract {
public:
   virtual void pureVirtualFunction() = 0; // Pure virtual function
};
class Concrete : public Abstract {
public:
   void pureVirtualFunction() override {
      cout << "Implementation of pure virtual function" << endl;
   }
};
```

- **Implementing an Abstract Class**

Steps:

Step	Description
1. Define an Abstract Class	Create a class with at least one pure virtual function.
2. Define Derived Classes	Create classes that inherit from the abstract class and override the pure virtual function.

3. Use the Derived Classes	Create objects of the derived classes and call the overridden methods.

- **Complete Example**

Here's a complete example that follows these steps:

```
#include <iostream>
using namespace std;
// Step 1: Define an Abstract Class
class Shape {
public:
   virtual void draw() = 0; // Pure virtual function
};
// Step 2: Define Derived Classes
class Circle : public Shape {
public:
   void draw() override {
      cout << "Drawing Circle" << endl;
   }
};

class Square : public Shape {
public:
   void draw() override {
      cout << "Drawing Square" << endl;
   }
};

int main() {
```

```
    // Step 3: Use the Derived Classes
    Circle circle;
    Square square;
    circle.draw();  // Outputs: Drawing Circle
    square.draw();  // Outputs: Drawing Square


    return 0;
        }
```

Explanation:

Define an Abstract Class: The Shape class is defined with a pure virtual function draw(), making it an abstract class.

Define Derived Classes: The Circle and Square classes inherit from Shape and override the draw() method.

Use the Derived Classes: In the main function, objects of Circle and Square are created, and their draw() methods are called.

- **Implement Interface**:

In C++, interfaces are implemented using abstract classes with only pure virtual functions.

Example:

```
class Interface {
public:
    virtual void method() = 0;
};
class Implementation : public Interface {
public:
    void method() override {
        cout << "Method implementation" << endl;
```

```
    }
};
```

In C++, interfaces are implemented using abstract classes with pure virtual functions.

An abstract class serves as a blueprint that specifies what methods a class must implement, without providing the implementation details.

- **Steps to implementing an interface in C++:**

Step	Description
1. Define an Abstract Class	Create a class with at least one pure virtual function.
2. Define Derived Classes	Create classes that inherit from the abstract class and override the pure virtual function.
3. Use the Derived Classes	Create objects of the derived classes and call the overridden methods.

- **Complete Example**

Here's a complete example that follows these steps:

```
#include <iostream>
using namespace std;
// Step 1: Define an Abstract Class
class Interface {
public:
    virtual void display() = 0; // Pure virtual function
};
// Step 2: Define Derived Classes
class ImplementationA : public Interface {
public:
```

```
    void display() override {
        cout << "Implementation A" << endl;
    }
};

class ImplementationB : public Interface {
public:
    void display() override {
        cout << "Implementation B" << endl;
    }
};

int main() {
    // Step 3: Use the Derived Classes
    ImplementationA objA;
    ImplementationB objB;
    objA.display();  // Outputs: Implementation A
    objB.display();  // Outputs: Implementation B

    return 0;
}
```

Explanation:

Define an Abstract Class: The Interface class is defined with a pure virtual function display(), making it an abstract class.

Define Derived Classes: The ImplementationA and ImplementationB classes inherit from Interface and override the display() method.

Use the Derived Classes: In the main function, objects of ImplementationA and ImplementationB are created, and their display() methods are called.

Key Points:

Abstract Class: Contains at least one pure virtual function and cannot be instantiated.

Pure Virtual Function: Declared with = 0 and must be overridden in derived classes.

Derived Classes: Must provide implementations for all pure virtual functions of the abstract class

6. Composition in C++

Composition is a fundamental concept in Object-Oriented Programming (OOP) that allows you to build complex objects from simpler ones. It represents a "has-a" relationship between objects, where one class contains objects of another class as its members. This is different from inheritance, which represents an "is-a" relationship.

- **Key Points of Composition**

Has-a Relationship: Composition models a "has-a" relationship. For example, a car "has-a" engine, a computer "has-a" CPU, etc.

Ownership: In composition, the contained objects (parts) are typically created and destroyed along with the containing object (whole). The lifetime of the parts is managed by the whole.

Encapsulation: Composition helps in encapsulating the functionality of the parts within the whole, promoting modularity and reusability.

- **Example of Composition**

Here's an example to illustrate composition in C++:

```
#include <iostream>

using namespace std;

// Class representing an Engine

class Engine {

public:
```

```
    void start() {
        cout << "Engine started" << endl;
    }
};

// Class representing a Car
class Car {
private:
    Engine engine; // Car has an Engine
public:
    void startCar() {
        engine.start(); // Using the Engine's functionality
        cout << "Car started" << endl;
    }
};

int main() {
    Car myCar;
    myCar.startCar(); // Outputs: Engine started
                      //          Car started
    return 0;
}
```

Explanation:

Engine Class: Represents a simple class with a method start().

Car Class: Contains an object of the Engine class as a member, demonstrating the "has-a" relationship.

Using Composition: The Car class uses the Engine class's functionality within its own method startCar().

- **Types of Composition**

Composition: The part (member) is an integral part of the whole (class) and cannot exist independently. The part is created and destroyed with the whole.

Aggregation: The part can exist independently of the whole. The whole does not manage the lifetime of the part.

- **Example of Aggregation**

```
#include <iostream>

using namespace std;

// Class representing a Wheel

class Wheel {

public:

   void rotate() {

      cout << "Wheel rotating" << endl;

   }

};

// Class representing a Car

class Car {

private:

   Wheel* wheel; // Car has a Wheel (aggregation)

public:

   Car(Wheel* w) : wheel(w) {}

   void move() {

      wheel->rotate(); // Using the Wheel's functionality

      cout << "Car moving" << endl;

   }

};
```

```
int main() {
   Wheel myWheel;
   Car myCar(&myWheel);
   myCar.move(); // Outputs: Wheel rotating
                 //         Car moving
   return 0;
                    }
```

Explanation:

Wheel Class: Represents a simple class with a method rotate().

Car Class: Contains a pointer to a Wheel object, demonstrating aggregation.

Using Aggregation: The Car class uses the Wheel class's functionality within its own method move().

- **Nested Classes**

A nested class is a class declared within another class. The nested class is a member of the enclosing class and has the same access rights as any other member. However, the members of the enclosing class do not have special access to the members of the nested class; the usual access rules apply.

Example:

```
#include <iostream>
using namespace std;
class Outer {
private:
   int outerVar;
   // Nested class
   class Inner {
   public:
      void display(Outer& o) {
         cout << "Outer class variable: " << o.outerVar << endl;
```

```
        }
    };
public:
    Outer(int val) : outerVar(val) {}
    void show() {
        Inner innerObj;
        innerObj.display(*this);
    }
};

int main() {
    Outer outerObj(10);
    outerObj.show();  // Outputs: Outer class variable: 10
    return 0;
}
```

Explanation:

Outer Class: The Outer class contains a private member outerVar and a nested class Inner.

Inner Class: The Inner class has a method display that can access the private members of the Outer class.

Accessing Nested Class: The Outer class creates an object of the Inner class and calls its method.

- **Nesting of Member Functions**

Nesting of member functions refers to calling one member function from within another member function of the same class. This allows for better organization and modularity of code.

Example:

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

```
class Example {
public:
   void func1() {
      cout << "Inside func1" << endl;
      func2();  // Calling another member function
   }
   void func2() {
      cout << "Inside func2" << endl;
   }
};
int main() {
   Example obj;
   obj.func1();  // Outputs: Inside func1
                 //          Inside func2
   return 0;
}
```

Explanation:

Class Definition: The Example class has two member functions, func1 and func2.

Nesting of Member Functions: The func1 method calls the func2 method within its body.

Calling Nested Functions: When func1 is called, it in turn calls func2.

7. Serialization in C++

Serialization is the process of converting an object's state into a format that can be stored or transmitted and then reconstructed later. This is useful for saving the state of an object to a file, sending it over a network, or storing it in a database. The reverse process, deserialization, reconstructs the object from the stored format.

- **Key Concepts**

Serialization: Converting an object into a sequence of bytes.

Deserialization: Reconstructing the object from the sequence of bytes.

Use Cases: Saving object state to a file, transmitting objects over a network, storing objects in a database.

- **How to Implement Serialization in C++**

  You can implement serialization in C++ using the fstream library for file operations or using external libraries like Boost.Serialization for more robust solutions.

  Example Using fstream

  Here's a simple example to illustrate serialization and deserialization using the fstream library:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
class Serializable {
private:
    string name;
    int age;
public:
    Serializable() {}
    Serializable(const string& name, int age) : name(name), age(age) {}
    // Function for Serialization
    void serialize(const string& filename) {
        ofstream file(filename, ios::binary);
        if (!file.is_open()) {
            cerr << "Error: Failed to open file for writing." << endl;
            return;
        }
```

```
        size_t nameLength = name.size();
        file.write(reinterpret_cast<char*>(&nameLength), sizeof(nameLength));
        file.write(name.c_str(), nameLength);
        file.write(reinterpret_cast<char*>(&age), sizeof(age));
        file.close();
    }
    // Function for Deserialization
    void deserialize(const string& filename) {
        ifstream file(filename, ios::binary);
        if (!file.is_open()) {
            cerr << "Error: Failed to open file for reading." << endl;
            return;
        }
        size_t nameLength;
        file.read(reinterpret_cast<char*>(&nameLength), sizeof(nameLength));
        name.resize(nameLength);
        file.read(&name[0], nameLength);
        file.read(reinterpret_cast<char*>(&age), sizeof(age));
        file.close();
    }
    void display() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    Serializable person("Alice", 30);
    person.serialize("data.bin");
```

```
    Serializable newPerson;
    newPerson.deserialize("data.bin");
    newPerson.display();  // Outputs: Name: Alice, Age: 30
    return 0;
     }
```

Explanation:

Class Definition: The Serializable class has private members name and age, and methods for serialization and deserialization.

Serialization: The serialize method writes the object's state to a binary file.

Deserialization: The deserialize method reads the object's state from the binary file.

Usage: In the main function, an object is serialized to a file and then deserialized from the file, demonstrating the process.

8. **Deserialization in C++**

Deserialization is the process of reconstructing an object from a sequence of bytes that was previously serialized. This is useful for reading the state of an object from a file, receiving it over a network, or retrieving it from a database.

Key Concepts

Deserialization: Converting a byte stream back into an object.

Serialization: The reverse process, converting an object into a byte stream.

Use Cases: Loading object state from a file, receiving objects over a network, retrieving objects from a database.

- **How to Implement Deserialization in C++**

You can implement deserialization in C++ using the fstream library for file operations or using external libraries like Boost.Serialization for more robust solutions.

Example Using fstream

Here's a simple example to illustrate deserialization using the fstream library:

```
#include <iostream>
#include <fstream>
```

```
#include <string>
using namespace std;
class Serializable {
private:
   string name;
   int age;
public:
   Serializable() {}
   Serializable(const string& name, int age) : name(name), age(age) {}
   // Function for Serialization
   void serialize(const string& filename) {
      ofstream file(filename, ios::binary);
      if (!file.is_open()) {
         cerr << "Error: Failed to open file for writing." << endl;
         return;
      }
      size_t nameLength = name.size();
      file.write(reinterpret_cast<char*>(&nameLength), sizeof(nameLength));
      file.write(name.c_str(), nameLength);
      file.write(reinterpret_cast<char*>(&age), sizeof(age));
      file.close();
   }
   // Function for Deserialization
   void deserialize(const string& filename) {
      ifstream file(filename, ios::binary);
      if (!file.is_open()) {
         cerr << "Error: Failed to open file for reading." << endl;
```

```
            return;
        }
        size_t nameLength;
        file.read(reinterpret_cast<char*>(&nameLength), sizeof(nameLength));
        name.resize(nameLength);
        file.read(&name[0], nameLength);
        file.read(reinterpret_cast<char*>(&age), sizeof(age));
        file.close();
    }
    void display() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    Serializable person("Alice", 30);
    person.serialize("data.bin");
    Serializable newPerson;
    newPerson.deserialize("data.bin");
    newPerson.display();  // Outputs: Name: Alice, Age: 30
    return 0;
}
```

Explanation:

Class Definition: The Serializable class has private members name and age, and methods for serialization and deserialization.

Serialization: The serialize method writes the object's state to a binary file.

Deserialization: The deserialize method reads the object's state from the binary file.

Usage: In the main function, an object is serialized to a file and then deserialized from the file, demonstrating the process.

Practical Activity 2.2.2: Applying Inheritance and Polymorphism

Task:

1: Read and perform the task below to Implement Inheritance with a Vehicle:

> **As a computer technician,** Develop a vehicle management system for a car rental company to manage cars, trucks, and motorcycles. Create a base class Vehicle with common attributes and methods, and implement derived classes Car, Truck, and Motorcycle with specific attributes and overridden displayInfo() methods. Demonstrate polymorphism by creating a list of Vehicle pointers and invoking their methods to show their specific behaviors.

2: Read the keyreadings **Key readings 2.2.2** in trainee manual

3: Create and run the program as described in task1

4: Ask for assistance if any.

5: check the output if they match the ones expected.

Key readings 2.2.2

Applying Inheritance and Polymorphism

Task Description

As a computer technician, you are tasked with developing a vehicle management system for a car rental company. This system will effectively manage different types of vehicles, including cars, trucks, and motorcycles. Below are the detailed steps to complete this task:

To implement inheritance and polymorphism follow these steps:

Step 1: Base Class Creation

a. Define the Base Class:

Create a class named Vehicle.

In the Vehicle class, define the following common attributes:

make: A string representing the manufacturer of the vehicle.

model: A string representing the model of the vehicle.

year: An integer representing the year of manufacture.

b. Define Common Methods:

Implement the following methods in the Vehicle class:

start(): A method that prints a message indicating that the vehicle has started.

stop(): A method that prints a message indicating that the vehicle has stopped.

displayInfo(): A virtual method that prints the vehicle's make, model, and year. This method will be overridden in derived classes.

Step 2: Derived Classes Implementation(creation)

a. Create the Car Class:

Define a class named Car that inherits from Vehicle.

Add an additional attribute to the Car class:

numberOfDoors: An integer representing the number of doors on the car.

Override the displayInfo() method to include the number of doors in the output.

b. Create the Truck Class:

Define a class named Truck that inherits from Vehicle.

Add an additional attribute to the Truck class:

payloadCapacity: An integer representing the maximum weight the truck can carry.

Override the displayInfo() method to include the payload capacity in the output.

c. Create the Motorcycle Class:

Define a class named Motorcycle that inherits from Vehicle.

Add an additional attribute to the Motorcycle class:

type: A string representing the type of motorcycle (e.g., cruiser, sport).

Override the displayInfo() method to include the type of motorcycle in the output.

Step 3: Demonstrating Polymorphism

a. Implement a Collection of Vehicle Pointers:

Create a vector (or list) of pointers to the Vehicle class. This will allow you to store objects of different derived classes in a single collection.

b. Populate the Collection:

Create instances of Car, Truck, and Motorcycle.

Add these instances to the collection of Vehicle pointers.

c. Iterate Through the Collection:

Use a loop to iterate through the collection of Vehicle pointers.

For each vehicle in the collection, invoke the following methods:

start(): Call this method to simulate starting the vehicle.

stop(): Call this method to simulate stopping the vehicle.

displayInfo(): Call this method to print out specific details about each vehicle. This will demonstrate polymorphism, as each derived class will provide its specific implementation of displayInfo().

Step 4: Testing and Validation

a. Test the System:

Ensure that each class is functioning as expected by checking the output of the methods when called.

Validate that the overridden displayInfo() method correctly displays the attributes of each specific vehicle type.

b. Document the Code:

Add comments to your code to explain the purpose of each class and method for better readability and maintainability.

Points to Remember

Description of inheritance and polymorphism

Here are key points to remember for describing inheritance and polymorphism

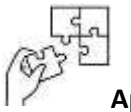
- **Inheritance** refers to a mechanism in object-oriented programming where a new class (derived class) acquires properties and behaviours (methods) from an existing class (base class). It promotes code reusability and establishes a relationship between classes.It can take different forms such as: Single, multiple, hierarchical, multilevel, and hybrid inheritance.
- **Polymorphism** is the ability of different objects to respond to the same method call in different ways. **It can be Compile-time (Static) Polymorphism**; Achieved through method overloading or operator overloading, orr**Run-time (Dynamic) Polymorphism**; achieved through method overriding, allowing a derived class to provide a specific implementation of a method that is already defined in its base class.
- **Base Class is t**he class from which other classes (derived classes) inherit properties and methods. It contains common attributes and behaviours that can be shared with derived classes.
- **Derived Class is a** class that inherits from one or more base classes. **It c**an access public and protected members of the base class; can also override methods to provide specific functionality.
- **Access Specifier** is keyword that sets the accessibility of classes, methods, and other members. **This can be:**
    - **Public: Accessible from any other class.**
    - **Private: Accessible only within the class itself.**
    - **Protected: Accessible within the class and by derived classes.**
- **Function Overriding** is a process of redefining a method in a derived class that already exists in the base class with the same name and signature and it is used to achieve dynamic polymorphism; allowing the derived class to provide a specific implementation.
- **Function Overloading** is creating multiple methods with the same name but different signatures (parameter types or number) within the same class. This allows to achieve compile-time polymorphism; allowing methods to handle different types or numbers of inputs.

Implementing inheritance and polymorphism

Here is a list of key points to remember for implementing inheritance and polymorphism by solving the vehicle management system task as example:

1. Define the Base Class

2. Create Derived Classes
3. Implement Polymorphism:
4. Testing:



Application of learning .2.2:

Define a base class Animal with attributes like name and age, along with methods such as speak() and move(). Then, create derived classes Dog and Cat that inherit from Animal, adding specific attributes like breed and color. Override the speak() method in each class to return "bark" for Dog and "meow" for Cat. Finally, instantiate Dog and Cat objects to demonstrate their inherited properties and methods.

Indicative content 2.3: Apply namespace

Duration: 5 hours

Theoretical Activity 2.3.1: Description of namespace

Tasks:

1: Read and answer the following questions:

1. What is a namespace in C++
2. What is the correct syntax for defining a namespace in C++?
3. What are the rules for using namespaces in C++?
4. Discuss how the using directive affects the visibility of names within a namespace

2: write the findings on paper/flipchart

3: Present your findings

4: Asks clarifications if any

5: Read the key readings **2.3.1.** in trainee manual

Key readings 2.3.1.

Description of Namespace

1. Definition

A namespace is a declarative region that provides a scope to the identifiers (names of types, functions, variables, etc.) inside it.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

A namespace is a container for identifiers, allowing the same identifier to be used in different contexts without conflict.

2. **Syntax**:

The syntax for defining a namespace is:

```
namespace namespace_name {
   // code declarations
    }
```

3. **Mechanism for Namespace** :

Namespaces allow you to group related **functions, classes**, and **variables** under **a single name**.

This helps in avoiding name conflicts and makes the code more modular and readable.

Namespaces in C++ are used to organize code into logical groups and to prevent name collisions, especially when your code base includes multiple libraries.

Here's a brief overview of how namespaces work and how to use them**:**

a. Defining a Namespace

A namespace is defined using the namespace keyword followed by the namespace name and a block of code:

b. Using a Namespace

To access members of a namespace, you can use the scope resolution operator :::

- **The using Directive**

You can also use the using directive to avoid having to prepend the namespace name:

- **Nested Namespaces**

Namespaces can be nested within each other

Nested Namespaces:

Namespaces can be nested within other namespaces:

```
namespace Outer {
   namespace Inner {
      int value;
   }
```

}

Outer::Inner::value = 10;

4. Benefits of Using Namespaces

Avoiding Name Collisions: Namespaces help prevent naming conflicts by grouping logically related identifiers.

Organizing Code: They make the code more modular and easier to manage.

Improving Readability: By clearly indicating the context of identifiers, namespaces improve code readability

5. Rules for namespace

Namespaces can be nested.

There is no limit to the number of levels of nesting.

The using directive can be used to bring all the names in a namespace into the current scope.

- **Uses of Namespaces:**

Accessing Elements:

You can access elements in a namespace using the scope resolution operator :::

MyNamespace::myVar = 5;

MyNamespace::myFunction();

- **Importing**:

Importing a namespace means bringing its members into the current scope using the using directive:

using namespace std;

cout << "Hello, World!" << endl;

- **Alias**

You can create an alias for a namespace to simplify its usage:

Example:

namespace MyNamespace {

```
    int myVar;
}

namespace MN = MyNamespace;

MN::myVar = 5;
```

- **Standard Library**:

The C++ Standard Library is defined within the std namespace.

For example:

```
#include <iostream>

using namespace std;

cout << "Hello, World!" << endl;
```

- **Handling Naming Conflicts**:

Handling naming conflicts in C++ is crucial to ensure that your code compiles and runs correctly, especially in large projects or when integrating multiple libraries.

Here are some common strategies to resolve naming conflicts:

1. Using Namespaces

Namespaces are a primary mechanism to avoid naming conflicts. By encapsulating your code within a namespace, you can prevent clashes with other code that might use the same names.

```
namespace MyNamespace {
    int myVariable;
    void myFunction() {
        // Function implementation
    }
}

// Accessing members of the namespace

MyNamespace::myVariable = 10;

MyNamespace::myFunction();
```

2. Using the Scope Resolution Operator

If you have global variables or functions with the same name, you can use the scope resolution operator :: to specify which one you mean.

```
int myVariable = 10;

void myFunction() {
   // Global function implementation
}

namespace MyNamespace {
   int myVariable = 20;
   void myFunction() {
      // Namespace function implementation
   }
}

int main() {
   ::myVariable = 30;  // Refers to the global myVariable
   MyNamespace::myVariable = 40;  // Refers to the namespace
myVariable::myFunction();  // Calls the global myFunction
   MyNamespace::myFunction();  // Calls the namespace myFunction
   return 0;
}
```

3. Renaming Identifiers

Sometimes the simplest solution is to rename one of the conflicting identifiers to something unique.

```
int globalVariable = 10;

namespace MyNamespace {
   int namespaceVariable = 20;
}
```

4. Using Aliases

You can create aliases for namespaces to make the code cleaner and avoid conflicts.

```
namespace LongNamespaceName {
   void myFunction() {
      // Function implementation
   }
}
namespace LNN = LongNamespaceName;
int main() {
   LNN::myFunction();  // Using the alias to call the function
   return 0;
}
```

5. Local Scope

Variables declared within a function have local scope and do not conflict with variables outside the function.

```
int myVariable = 10;
void myFunction() {
   int myVariable = 20;  // Local variable
   std::cout << myVariable << std::endl;  // Outputs 20
}
int main() {
   myFunction();
   std::cout << myVariable << std::endl;  // Outputs 10
   return 0;
}
```

Practical Activity 2.3.2: Applying namespace

Task:

1:Read and perform the task bellow:

Create a namespace called MathOperations that contains two functions:

add and subtract.

Implement these functions to perform addition and subtraction of two integers

Write a main function to test these operations.

2: Read the key readings **2.3.2** in the trainee manual

3: Apply namespace as asked above ask for assistance if any

4: Present the work to the whole class

Key readings 2.3.2:

Applying namespace

Here are the basic steps about implementing namespace by creating and using the MathOperations namespace

with add and subtract functions:

1. **Namespace Definition:**

   The namespace MathOperations is defined using the namespace keyword followed by the namespace name.

   It is to be noted that, there is no semicolon (;) after the closing brace.

   To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

   namespace_name: :code; // code could be variable , function or class.

2. **Function Definitions:**

   add(int a, int b): This function takes two integers as parameters and returns their sum.

subtract(int a, int b): This function takes two integers as parameters and returns the result of subtracting the second integer from the first.

Encapsulation:

The namespace encapsulates the **add** and subtract functions, grouping them logically under MathOperations.

3. **Using Namespace Functions:**

To call the functions within the namespace, you need to use the scope resolution operator**:** like **MathOperations::add** and **MathOperations::subtract.**

✓ **Main Function:**

The main function demonstrates how to use the add and subtract functions from the MathOperations namespace.

It includes examples of adding and subtracting two integers and prints the results.

✓ **Code Structure:**

The code is structured to include the namespace definition, function implementations, and a main function to test the operations.

Points to Remember

Description of namespace std

The following are points to remember while working with namespace in C++:

A namespace is a declarative region for organizing code and grouping related identifiers (variables, functions, classes).

It prevents naming conflicts, allowing the same name to be used in different namespaces without collision.

- **Syntax**:

Defined using the namespace keyword followed by the name and enclosed in curly braces, e.g.,

namespace NamespaceName { /* Declarations */ }

- **Rules for Use**:

- ✓ Access identifiers with the scope resolution operator (::)
- ✓ Support for nested namespaces for further organization.
- ✓ Use of using keyword to bring names into the current scope, though it may lead to conflicts.

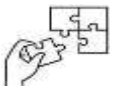
- **Using Directive**

The using directive (e.g., using namespace NamespaceName;) makes all names in that namespace available without qualification.

Applying namespace std

Here are the basic steps about applying namespace:

- **Namespace Definition**
- **Function Definitions:**
- **Encapsulation:**
- **Using Namespace Functions**
- **Main Function:**
- **Code Structure:**



Application of learning 2.3.

Task: About Avoiding Name Conflicts

Create two namespaces NamespaceA and NamespaceB, each containing a function printMessage that prints different messages.

Write a main function to call both functions without causing name conflicts.

Indicative content 2.4: Apply Error and Exception handling

Duration: 7hourrs

Theoretical Activity 2.4.1.: Description of Error and Exception handling

Tasks:

1: In group, discuss about the given questions and answer them

1. Define the term bellow:

   a) Errors
   b) Exception

2. Describe the types the following:

   a) Types of errors
   b) Exception handling mechanisms

2: Write your findings on papers/ flipchart

3: Present your findings to the whole class

4: Ask for clarifications if any

5: Read the key readings **2.4.1** in trainee manual

Key readings 2.4.1:

Description of Error and Exception Handling

1. Errors

Errors are serious issues that occur during the execution of a program and are typically outside the control of the program itself. They often indicate problems with the environment in which the application is running, such as running out of memory or a system crash. Errors are usually not meant to be caught or handled by the application.

- **Types of Errors**

✓ **Syntax Errors**: Mistakes in the code's syntax, such as missing semicolons or incorrect use of keywords.

✓ **Type Errors**: Mismatches between expected and actual data types.

✓ **Linker Errors**: Issues that arise during the linking process, such as missing function definitions.

✓ **Runtime Errors**: Errors that occur during program execution, like division by zero or accessing invalid memory.

✓ **Logical Errors**: Flaws in the program's logic that lead to incorrect results

2. Exceptions

Exceptions are conditions that a program might want to catch and handle. They represent issues that can occur during the execution of a program, such as invalid user input or a failed network connection. Exceptions can be caught and handled to allow the program to continue running or to provide a meaningful error message to the user.

- **Examples of exceptions**

✓ **NullPointerException**: Thrown when an application attempts to use null in a case where an object is required.

✓ **IOException**: Thrown when an I/O operation fails or is interrupted**.**

✓ **StackOverflowError**: Thrown when the call stack overflows due to too many method invocations

- **Exception Handling Mechanism:**

Exception handling is a way to manage errors gracefully, ensuring the program can recover or terminate cleanly.

The following are the components of Errors and Exception Handling Mechanism:

✓ **The try Block**:

Contains code that might throw an exception. If an exception occurs, control is transferred to the catch block.

```
try {
  // Code that may throw an exception
}
```

✓ **The throw Statement**:

Used to signal the occurrence of an exception. It transfers control to the nearest catch block.

throw exception;

// Example: throw std::runtime_error("Error occurred");

✓ **The catch Block**:

Catches and handles the exception thrown by the try block.

Each catch block is associated with a specific type of exception.

catch (const std::exception& e)

{

// Code to handle the exception

std::cerr << "Exception: " << e.what() << std::endl;

}

- **Exception Safety and Resource Management techniques**

✓ **Basic Guarantee** The function guarantees that if an exception is thrown, the program remains in a valid state, and no resources are leaked. However, the state of the program might be partially modified.

Use Case:

General-purpose functions where maintaining a valid state is crucial, but some side effects are acceptable.

✓ **No Guarantee (Weakest)** The function provides no guarantees about the program's state if an exception is thrown. The program might be left in an inconsistent state.

Use Case:

Functions where exception safety is not a priority, or where performance considerations outweigh the need for exception safety.

✓ **No-throw guarantee**: This means that a function is guaranteed not to throw any exceptions. It's a strong guarantee that ensures the function will complete without causing any exception-related issues

Practical Activity 2.4.2: Applying Error and Exception handling

Task:

1: Read and perform the task below:

Create a C++ program that safely performs division using a function safeDivide, which throws an exception for division by zero. Prompt the user for input in the main function and use a try block to call safeDivide. Handle any exceptions by displaying an error message, or print the division result if successful

2: Read key reading 2.4.2 in trainee manual

3: Perform the task described in task

4: Ask for assistance if needed

5: check the outputs if they match the expected results.

Key readings 2.4.2.

Applying Error and Exception handling

To apply errors and exception handlindling by performing the task of creating a base class Employee and derived classes Manager and Develop an overridden calculateBonus() methods,you have to emphasis on the following

1. Base Class Definition:

- The Employee class includes attributes like name and salary.
- It has a virtual method calculateBonus() which can be overridden by derived classes

2. Virtual Method:

- The calculateBonus() method in the Employee class is declared as virtual, allowing derived classes to provide their own implementation.

3. Inheritance:

- The Manager and Developer classes inherit from the Employee class using public inheritance.
  - This means they inherit the public and protected members of Employee.

4. Overriding Methods:

- Both Manager and Developer classes override the calculateBonus () method to provide specific implementations for calculating bonuses.

5. Polymorphism:

- By using a virtual method, you enable polymorphism, allowing you to call calculateBonus () on an Employee pointer or reference and execute the correct derived class method.

6. Encapsulation:

- Each class encapsulates its own data and behavior, promoting modularity and code reuse.

Points to Remember

Errors and exceptions handling

While working on errors and handling exceptions in C++ note the following:

- Errors refer to issues that occur during program execution, causing it to behave unexpectedly or crash. They can arise from syntax mistakes, logical flaws, or resource unavailability. Errors can be broadly classified into compile-time and runtime errors.
- An exception is an unexpected event that disrupts the normal flow of a program during execution. Exceptions are objects that encapsulate error conditions and can be thrown and caught to handle these issues gracefully, allowing the program to recover or terminate cleanly.
- **Types of Errors**:
  - ✓ **Syntax Errors**: Mistakes in the code structure that prevent the program from compiling (e.g., missing semicolons).
  - ✓ **Runtime Errors**: Errors that occur during program execution, such as division by zero or accessing invalid memory.

- ✓ **Logical Errors**: Flaws in the program's logic that lead to incorrect results, despite the program compiling and running without crashing.

- **Exception Handling Mechanisms**:

  - ✓ **Try-Catch Blocks**: Code that may throw exceptions is placed within a try block, and potential exceptions are caught and handled in corresponding catch blocks.
  - ✓ **Throwing Exceptions**: The throw keyword is used to signal an exception when an error condition occurs, transferring control to the nearest catch block.
  - ✓ **Finally Blocks (not native to C++)**: While not present in C++, similar functionality can be achieved through destructors or cleanup code to ensure resources are released regardless of whether an exception occurs.

Applying errors and exception handling

In order to do the aply error handling and exception handling you have to:

- **Define the Function**:

  - ✓ Create a function named safeDivide(double numerator, double denominator) that performs division.
  - ✓ Check if the denominator is zero. If it is, throw an exception (e.g., std::runtime_error).

- **Implement Exception Handling**:

  - ✓ Use a try block in the main function to call safeDivide().
  - ✓ Surround the division call with exception handling to catch errors.

- **User Input**:

  - ✓ Prompt the user to enter two numbers: the numerator and the denominator.
  - ✓ Use std::cin to read the input values and ensure they are of type double.

- **Error Handling**:

  - ✓ Catch exceptions using a catch block.
  - ✓ Display a user-friendly error message if an exception is caught, indicating that division by zero is not allowed.

- **Output the Result**:

    - ✓ If no exceptions occur, print the result of the division.
    - ✓ Ensure that the output is clear and easy to understand.

- **Testing**:

    - ✓ Test the program with various inputs, including valid divisions, division by zero, and non-numeric inputs to ensure robustness.
    - ✓ Consider edge cases, such as very large or very small numbers.

- **Code Clarity**:

    - ✓ Keep the code well-organized and commented to explain the logic, especially in error handling sections.
    - ✓ Use meaningful variable names for clarity.

- **Compilation and Execution**:

    - ✓ Ensure the program compiles without errors.
    - ✓ Run the program in an environment where user input can be tested interactively.

Application of learning 2.4:

Create a program for an employee management system with a base class Employee that has attributes like name and salary, and a virtual method calculateBonus(). Implement derived classes Manager and Developer, each overriding calculateBonus() for their specific bonus calculations. Include error handling for scenarios such as invalid salary values and incorrect employee types, using exceptions to manage errors effectively. This task highlights practical applications of error and exception handling within an inheritance structure in a real-world context.

Learning outcome 2 end assessment

Theoretical assessment

I. Referring from the contents about Apply OOP concepts in C++ focusing on class, object, inheritance, polymorphism, namespaces, error handling and exception handling, answer the following questions by **True** or **False**:

1. In C++, an object is an instance of a class.

2. Inheritance allows one class to acquire the properties and behavior of another class.

3. In C++, runtime polymorphism can be achieved using function overloading.

4. A class constructor in C++ can have a return type.

5. Namespaces in C++ help avoid name conflicts by encapsulating identifiers.

6. A virtual function must be redefined in all derived classes to enable polymorphism.

7. When a derived class inherits privately from a base class, all public members of the base class become private in the derived class.

8. In C++, exceptions are handled using try, catch, and finally blocks.

9. The standard C++ exception handling mechanism guarantees that every exception will be caught.

10. If a function is declared inside a namespace, you must use the namespace qualifier to access it unless using a using directive.

11. Can derived class access private members of its base class directly.

II. Choose the best answer by circling the letter that fits:

1. Which of the following correctly defines an object in C++?
   a) A method of a class
   b) An instance of a class
   c) A static variable
   d) A friend function

2. What is the main purpose of a constructor in C++?
   a) To destroy an object when it goes out of scope
   b) To initialize an object when it is created
   c) To allocate memory for an object
   d) To return values from an object

3. In C++, which type of inheritance allows a derived class to inherit from more than one base class?
   a) Single inheritance
   b) Multiple inheritance
   c) Hierarchical inheritance
   d) Multilevel inheritance

4. How run-time polymorphism is typically achieved in C++?
   a) Function overloading
   b) Operator overloading
   c) Virtual functions
   d) Friend functions

5. What is the primary purpose of using namespaces in C++?
   a) To group constants together
   b) To avoid name conflicts in large programs
   c) To increase runtime performance
   d) To create abstract classes

6. In C++, what happens to the public members of a base class when inherited privately?
   a) They remain public in the derived class
   b) They become protected in the derived class
   c) They become private in the derived class
   d) They are not inherited at all

7. Which of the following is used to throw an exception in C++?
   a) catch
   b) throw
   c) try
   d) error

8. Which of the following statements about virtual functions in C++ is correct?
   a) A virtual function cannot be overridden
   b) A virtual function can only be used in abstract classes
   c) A virtual function is defined in a base class and can be overridden in a derived class
   d) A virtual function must return an integer

9. If you have a function inside a namespace, how can you access it without using the namespace prefix every time?
   a) By declaring the function as static
   b) By using the using directive
   c) By re-declaring the function globally
   d) By using a friend function

10. In C++, what happens if an exception is thrown but not caught by any catch block?
    a) The program terminates with an error message
    b) The program continues to run without interruption
    c) The exception is ignored
    d) The exception is automatically handled by the operating system

III. Complete sentences that follow using one of the words: **catch,throw, virtual, class,inherit, multiple, initialize,conflicts,private, compile**

1. An object is an instance of a _____.
2. A constructor is used to _____ an object.
3. Inheritance allows a class to _____ properties from another class.
4. Runtime polymorphism is achieved using _____ functions.
5. Namespaces are used to avoid name _____.
6. The keyword used to handle exceptions in C++ is _____.
7. A class that inherits from more than one base class exhibits _____ inheritance.
8. When a class is inherited privately, public members of the base class become _____ in the derived class.
9. Function overloading is a form of _____-time polymorphism.
10. Exceptions are thrown using the _____ keyword.

IV. Answer the following questions:

1. What is the relationship between a class and an object in C++?
2. What is the role of a constructor in C++?.
3. How does inheritance promote code reusability?
4. How is runtime polymorphism achieved in C++?

5. What is the purpose of using namespaces in C++?
6. Which keywords are used for exception handling in C++?
7. What is multiple inheritance in C++?
8. What happens to the public members of a base class when it is inherited privately?
9. What is function overloading in C++?
10. What happens if an exception is thrown but not caught in C++?

I. Match the following OOP Terms with their corresponding meanings

Answer	OOP Terms	Meaning
……	1. Class	A. An instance of a class.
……	2. Object	B. A special member function of a class that initializes objects of the class.
…….	3. Encapsulation	C. A mechanism where a new class (derived class) inherits properties and behavior (methods) from an existing class (base class).
……	4. Data Abstraction:	D. A blueprint for creating objects.
……	5. Constructor:	E. A special member of a class that cleans up when an object is destroyed.

……..	6. Destructor	F. The concept of wrapping data and methods that operate on the data within a single unit
…….	7. Inheritance:	G. The process of hiding the complex implementation details and showing only the necessary features of an object
…….	8. Polymorphism	H. The ability to create multiple functions with the same name but different parameters
…….	9. Function Overloading	I. When a derived class has a definition for one of the member functions of the base class.
……	10. Function Overriding	J. Is a member function that does not modify any member variables of the class.
…….	11. A constant function	K. The ability of a function, object, or method to take on many forms
		L. It define the access level of class members.

Practical assessment

Scenario 1: Library Management System

As a computer technician design a simple Library Management System that will hehlp librarian to manager the following manage books, members, and transactions by using OOP principles like classes, inheritance, polymorphism, and encapsulation:

END

References

Bjarne, S. (2014). The C++ programming language (4th ed.). Addison-Wesley.

Bjarne, S. (2018). A tour of C++ (2nd ed.). Addison-Wesley.

C++ Classes and Objects - W3Schools
C++ Exception Handling - W3Schools
C++ Namespaces - W3Schools

C++ Standards Committee. (2014). ISO/IEC 14882:2014 - Programming languages — C++. International Organization for Standardization.

Classes and Objects in C++ - GeeksforGeeks
Classes and Objects in C++ - Tutorialspoint

Deitel, P. J., & Deitel, H. M. (2015). C++: How to program (10th ed.). Pearson.

Eckstein, D., & Heller, R. (2007). C++ programming for the absolute beginner. Course Technology.

Effective C++ (2019). Polymorphism in C++: A Tutorial. Retrieved from https://www.effectivecpp.com/polymorphism-in-c/

Ellis, P., & Stroustrup, B. (2014). The Annotated C++ Reference Manual. Addison-Wesley.

Error Handling in C++ - Tutorialspoint
Exception Handling in C++ - GeeksforGeeks

Hsu, W. (2018). Object-oriented programming with C++. Oxford University Press.

Inheritance and Polymorphism – Programming Fundamentals
Inheritance and Polymorphism in Java - CodingDrills
Inheritance and Polymorphism in Java - Syntax Savvy
Inheritance, Polymorphism, and Abstract Classes - Javanotes

Khosravi, R., & Huang, D. (2017). Error and exception handling in C++: Best practices. IEEE Transactions on Software Engineering, 43(5), 1110-1122.

Koenig, A., & Moo, B. E. (2005). C++ Templates: The Complete Guide. Addison-Wesley.

Lippman, S. B., Lajoie, J., & Moo, B. E. (2012). C++ primer (5th ed.). Addison-Wesley.

Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.

McKinsey, M. (2020). Namespaces in C++: A Comprehensive Guide. Journal of Software Engineering, 12(4), 45-59.

Meyers, S. (2014). Effective C++: 55 specific ways to improve your programs and designs (3rd ed.). Addison-Wesley.

Miller, S. (2019). Understanding Exception Handling in C++. Retrieved from https://www.geeksforgeeks.org/exception-handling-c/

Namespaces in C++ - GeeksforGeeks

Roberts, D. (2016). Understanding C++: An Introduction to Object-Oriented Programming. Cengage Learning.

Sedgewick, R., & Wayne, K. (2011). Algorithms in C++ (3rd ed.). Addison-Wesley.

Sia, J. (2021). C++ Inheritance: A Beginner's Guide. O'Reilly Media

Stroustrup, B. (2013). Programming: Principles and practice using C++ (2nd ed.). Addison-Wesley.

Sutherland, I. E. (2009). C++: An Introduction to Object-Oriented Programming. Cengage Learning.

Understanding Encapsulation, Inheritance, Polymorphism, Abstraction in OOPs - GeeksforGeeks
Understanding Namespaces in C++ - Tutorialspoint


C++

Indicative contents

3.1.Apply pointers
3.2.Apply file handling
3.3. Apply multithreading and concurrency
3.4. Apply inline assembly

Key Competencies for Learning Outcome 3: Perform CPU Optimization

Knowledge	Skills	Attitudes
• Description of pointer • Declaration of pointer • Initialization of pointer • Description of file • Description of header file • Description of fstream class • Definition of assembly language • Description of inline assembly	• Declaring and Initializing the pointer • Dereferencing pointer • Calling function with pointer • Creating threads • Managing threads • Applying thread synchronization • Implementing thread pool • Applying parallel algorithm • Implementing Parallel Data Processing • Implementing Task-Based Parallelism • Implementing Load Balancing • Applying inline assembly • Using register constraints	• Being self-motivated • Being critical thinker • Being detailed oriented • Being skilful • Being creative

Duration: 20 hrs

Learning outcome 3 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe properly pointer in C++ programming language
2. Differentiate correctly the pointer from standard variables
3. Declare properly the pointer inline with C++ language
4. Us correctly a pointer as function argument during its call
5. Implement properly the dynamic memory allocation in C++
6. Describe properly the deadlock and race conditions in multithreading context
7. Implement correctly thread pool during concurrency process
8. Apply correctly parallel algorithm in multithreading context
9. Apply correctly inline assembly using C++ programming language

Resources

Equipment	**Tools**	**Materials**
• Computer • Storage device	• Integrated Development Environment (IDE)	• Online Tutorials • Internet Connection

Indicative content 3.1: Apply pointers

Duration: 4hrs

Theoretical Activity 3.1.1: description of pointers

Tasks:

1: Answer the questions reflecting to C++ pointers:

1) What do you mean by pointer in C++?
2) Describe the following :
   a) Memory address
   b) Dereferencing a pointer
   c) Memory leak
   d) Smart pointers
3) How do you declare pointer in C++?
4) Discuss the relationship between variables and pointers in C++.

2: In your respective groups, write the key findings on papers/ flipcharts

3: Presentation of the findings to the whole class

4: Ask questions clarifications if any

5: Read through Key readings 3.1.1 in trainee manual

Key readings 3.1.1.:

Description of an pointer

1. Definition

In C++, a pointer is a variable that stores the memory address of another variable. It "points" to the location in memory where the value of another variable is stored, allowing indirect access and manipulation of the variable's value.

a. Memory address

In C++, a memory address for a pointer refers to the location in memory where the pointer itself is stored. This is distinct from the memory address that the pointer holds, which is the address of the variable it points to

b. Relationship between variables and pointers

The relationship between variables and pointers in C++ revolves around the concept of memory management and access. A pointer is a special type of variable that stores the memory address of another variable. Understanding this relationship is crucial for dynamic memory management, passing data efficiently to functions, and handling arrays and structures.

c. Pointer declaration

In C++, a pointer declaration is the process of defining a pointer variable. The pointer is declared with a specific data type, indicating the type of data it can point to.

Syntax:

data_type * Pointer_Name;

Where

data_type: The type of the variable the pointer will point to (e.g., int, float, char).

*****: Indicates that the variable is a pointer.

pointer_name: The name of the pointer variable.

Examples

int* ptr; //This declares ptr as a pointer to an integer.

char* charPtr;//This declares char Ptr as a pointer to a char (character).

float* floatPtr;//This declares floatPtr as a pointer to a float.

double* doublePtr;//This declares doublePtr as a pointer to a double.

d. Initialization of pointer

To initialize a pointer, you specify the pointer type, followed by the pointer variable, and assign it an address or nullptr (in C++11) and null(in older version of C++)

Syntax: With different ways to initialize a pointer use the following syntax:

- ✓ **Pointer to a specific type**

type *pointer_name=address;

Example for an integer pointer:

```
int *ptr = nullptr; // C++11 and later
// or
int value = 10;
int *ptr = &value; // Points to the address of 'value'
```

✓ **Pointer with dynamic memory allocation**

type *pointer_name=new type;

Example:
int *ptr = new int; // Allocates memory for one integer
*ptr = 10; // Assigns value to allocated memory

Note: remember to de-allocate the memory when done

Delete ptr;

✓ **Pointer to a string literal**:

char *ptr = "Hello, World!";

✓ **Null pointer initialization:**

type *pointerName = nullptr; // C++11 and later
// or
type *pointerName = NULL; // in older version of C++

e. **Pointer data type**

The pointer data type determines the type of data the pointer is allowed to point to and how much memory to read when dereferencing the pointer. The data type of a pointer depends on the type of variable it points to.

Examples:

Pointer to an Integer (**int*ptr;):** A pointer that points to an integer.

Pointer to a Character (**char*ptr;):** A pointer that points to a character or string

Pointer to a Float (**float*ptr;):** A pointer that points to a floating-point number

Pointer to a Double (**double*ptr;):** A pointer that points to a double precision floating-point number.

void Pointer (**void*ptr**): A generic pointer that can point to any data type. It cannot be dereferenced directly without typecasting.

f. Dereferencing pointer

Dereferencing a pointer in C++ means accessing or modifying the value located at the memory address the pointer holds. This is achieved using the dereference operator (*)

Syntax:

pointerName;

Where

pointerName: contains a memory address.
When you dereference (*pointerName), you access the value stored at that memory address.

Examples

- ✓ **Dereferencing a Pointer to a Variable**

```
int value = 42;
int *ptr = &value;  // 'ptr' holds the address of 'value'

int dereferencedValue = *ptr;  // Dereferencing the pointer gives the value of 'value'
std::cout << dereferencedValue;  // Output: 42
```

- ✓ **Dereferencing Dynamically Allocated Memory**

```
int *ptr = new int;  // Allocates memory for an integer
*ptr = 100;         // Dereferencing the pointer to store a value

std::cout << *ptr;   // Output: 100

delete ptr;  // Deallocates the memory to avoid a memory leak
```

- ✓ **Dereferencing Array Pointers**

Array names can be used as pointers, and you can dereference them like regular pointers.

```
int arr[3] = {10, 20, 30};
int *ptr = arr;  // 'ptr' points to the first element of the array

std::cout << *ptr;       // Output: 10 (first element)
std::cout << *(ptr + 1);  // Output: 20 (second element)
std::cout << *(ptr + 2);  // Output: 30 (third element)
```

- ✓ **Dereferencing Pointers to Functions**

Pointers can also point to functions, and they can be dereferenced when you call the function.

```
void myFunction() {
   std::cout << "Function called!" << std::endl;
}
void (*funcPtr)() = &myFunction;  // Pointer to function
```

(*funcPtr)(); // Dereferencing and calling the function (Output: Function called!)

Notes:

Null Pointers: Dereferencing a nullptr (or NULL) results in undefined behavior. Always check if a pointer is not null before dereferencing.

```
int *ptr = nullptr;
if (ptr != nullptr) {
   std::cout << *ptr;
}
```

Uninitialized Pointers: Dereferencing an uninitialized pointer leads to undefined behavior. Always initialize pointers before using them.

```
int *ptr;  // Uninitialized, don't dereference!
```

Dangling Pointers: refers to dereferencing a pointer that no longer points to a valid memory location (after using delete or when it goes out of scope) can lead to crashes or undefined behavior.

```
int *ptr = new int(10);
delete ptr;
// ptr is now dangling, dereferencing would cause an issue
```

So, Dereferencing is used to access the value stored at the memory location the pointer is pointing to. Always ensure that a pointer is properly initialized and valid before dereferencing to.

g. Pointer arithmetic

Pointer arithmetic is a programming concept that allows you to perform mathematical operations on pointers. In C++, allows you to perform operations on pointers to navigate through memory. This is useful for iterating over arrays and handling dynamic memory

- ✓ **Basic operations**

Incrementing a Pointer (ptr++)

Moves the pointer to the next element of the type it points to.

Example

```
int arr[3] = {1, 2, 3};
int *ptr = arr;  // Points to arr[0]
ptr++;         // Now points to arr[1]
std::cout << *ptr;  // Output: 2
```

Decrementing a Pointer (ptr--)

It moves the pointer to the previous element.

Example

```
int arr[3] = {1, 2, 3};
int *ptr = arr + 2;  // Points to arr[2]
ptr--;           // Now points to arr[1]
std::cout << *ptr;  // Output: 2
```

Pointer Addition (ptr + n)

Moves the pointer forward by n elements of the type it points to.

Example

```
int arr[3] = {1, 2, 3};
int *ptr = arr;    // Points to arr[0]
ptr = ptr + 2;     // Now points to arr[2]
std::cout << *ptr;  // Output: 3
```

Pointer Subtraction (ptr - n)

It moves the pointer backward by n elements.

Example

```
int arr[3] = {1, 2, 3};
int *ptr = arr + 2; // Points to arr[2]
ptr = ptr - 1;     // Now points to arr[1]
std::cout << *ptr;  // Output: 2
```

Difference Between Pointers (ptr2 - ptr1)

It computes the number of elements between two pointers.

```
int arr[3] = {1, 2, 3};
int *ptr1 = arr;    // Points to arr[0]
int *ptr2 = arr + 2; // Points to arr[2]
std::cout << (ptr2 - ptr1);  // Output: 2 (distance between arr[2] and arr[0])
```

Pointer Arithmetic with Arrays

You can use pointer arithmetic to traverse arrays.

Example

```
int arr[3] = {1, 2, 3};
int *ptr = arr;  // Points to arr[0]

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

```
   std::cout << *(ptr + i) << " ";  // Output: 1 2 3
}
```

Using Pointer Arithmetic with Dynamic Arrays

Example

```
int *arr = new int[3]{1, 2, 3};
int *ptr = arr;

for (int i = 0; i < 3; ++i) {
   std::cout << *(ptr + i) << " ";  // Output: 1 2 3
}
delete[] arr;
```

Pointer Arithmetic with Structures

Pointer arithmetic with structures can be used to access array elements of structures.

```
struct Point {
   int x, y;
};
Point arr[2] = {{1, 2}, {3, 4}};
Point *ptr = arr;
std::cout << (ptr + 1)->x << ", " << (ptr + 1)->y;  // Output: 3, 4
```

Pointer Arithmetic with Function Pointers

Function pointers don't support arithmetic directly. They are treated as a single entity, not an array.

```
void func1() {}
void func2() {}
void (*funcPtrs[2])() = {func1, func2};
void (**ptr)() = funcPtrs;
ptr++;  // Moves to the next function pointer
(*ptr)();  // Calls func2
```

Pointer Arithmetic and Array Bounds

Accessing elements outside of an array's bounds via pointer arithmetic leads to undefined behavior. Always ensure pointers stay within valid memory regions.

```
int arr[3] = {1, 2, 3};
int *ptr = arr;
```

```
std::cout << *(ptr + 3);  // Undefined behavior: accessing out-of-bounds memory
```

program example

```
#include <iostream>
int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int *p = arr;
   // Accessing elements using pointer arithmetic
   std::cout << *(p + 2) << std::endl; // Output: 3
   // Iterating over the array using pointer arithmetic
   for (int i = 0; i < 5; i++) {
      std::cout << *(p + i) << " ";
   }
   std::cout << std::endl;
   // Using references
   int x = 10;
   int &ref = x;
   ref = 20;
   std::cout << x << std::endl; // Output: 20
   // Using nullptr
   int *ptr = nullptr;
   if (ptr != nullptr) {
      // Code that would be executed if ptr was not null
   }
   return 0;
}
```

h. Passing pointers as function argument

Passing pointers to functions as arguments in ++, allow you to:

- ✓ **Modify the Original Data**: Functions can alter the data at the address pointed to by the pointer.
- ✓ **Avoid Data Copying**: This is useful for large data structures, as you only pass the pointer (an address), not the entire data.
- ✓ **Handle Dynamic Memory**: You can manage dynamically allocated memory efficiently

Example: Pointer to pointer

```
#include<iostream.h>

void myFunction(int *ptr) {
   // Access and modify the value pointed to by ptr
```

```
    *ptr = 10;
}
int main() {
    int x = 5;
    myFunction(&x); // Pass the address of x
    std::cout << x << std::endl; // Output: 10
}
```

Example2: Pointer to variable

```
#include <iostream>
void updateValue(int *ptr) {
    *ptr = 20;  // Modify the value pointed to by ptr
}
int main() {
    int value = 10;
    updateValue(&value);  // Pass the address of 'value'
    std::cout << value;  // Output: 20
    return 0;
}
```

Example3: Pointer to array

```
#include <iostream>
void printArray(int *arr, int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";  // Access array elements using pointer
    }
}
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    printArray(arr, 5);  // Pass the array to the function
    return 0;
}
```

Example 4: Pointer to a Pointer

```
#include <iostream>
void allocateMemory(int **ptr, int size) {
    *ptr = new int[size];  // Allocate memory and update the pointer
}
int main() {
    int *arr = nullptr;
```

```
    allocateMemory(&arr, 10);  // Pass address of the pointer
    // Use arr ...
    delete[] arr;  // Remember to deallocate memory
    return 0;
}
```

Example5: Pointer to a Pointer

```
#include <iostream>
void allocateMemory(int **ptr, int size) {
    *ptr = new int[size];  // Allocate memory and update the pointer
}
int main() {
    int *arr = nullptr;
    allocateMemory(&arr, 10);  // Pass address of the pointer
    // Use arr ...
    delete[] arr;  // Remember to deallocate memory
    return 0;
}
```

i. Use of dynamic memory allocation

Dynamic memory allocation is a technique used in C++ to allocate memory during runtime based on the program's needs. This provides flexibility and efficiency, especially when dealing with variable-sized data structures or when the exact memory requirements are unknown beforehand.

- **Common use cases**

✓ **Arrays of unknown size:**

When you don't know the exact size of an array at compile time, dynamic memory allocation allows you to create an array of the required size at runtime.

✓ **Data structures:**

Many data structures, such as linked lists, trees, and graphs, rely on dynamic memory allocation to create and manage nodes and connections.

✓ **Memory management:**

Dynamic memory allocation provides more granular control over memory usage compared to static allocation, allowing for more efficient memory management.

- **Key functions:**

new: Allocates memory for an object of a specified type.
delete: De-allocates memory previously allocated with new.
new[]: Allocates memory for an array of objects.
delete[]: De-allocates memory previously allocated with new[].

Example:

```
#include <iostream>
int main() {
   int *arr;
   int size;
   std::cout << "Enter the size of the array: ";
   std::cin >> size;
   // Allocate memory for the array
   arr = new int[size];
   // Populate the array
   for (int i = 0; i < size; i++) {
      arr[i] = i + 1;
   }
   // Print the array
   std::cout << "Array elements: ";
   for (int i = 0; i < size; i++) {
      std::cout << arr[i] << " ";
   }
   std::cout << std::endl;
   // De-allocate memory
   delete[] arr;
   return 0;
}
```

- ✓ **The new Operator**

It is used in C++ to allocate memory dynamically for objects on the heap.
Syntax:

New type_name;

New type_name(initialize_list);

New type_name[size];
type_name: The type of object to be allocated (int, float,...)

initializer_list: An optional list of initializers for the object's members.
size: The number of elements in the array to be allocated.

Allocation

The new operator allocates memory on the heap for the specified object or array. It returns a pointer to the allocated memory.
If allocation fails due to insufficient memory, the new operator throws a std::bad_alloc exception.

Initialization

If an initializer list is provided, the object's members are initialized with the corresponding values.
If no initializer list is provided, the object's default constructor is called.

Array Allocation

When used with square brackets [], the new operator allocates memory for an array of objects.
The size argument specifies the number of elements in the array.
The returned pointer points to the first element of the array.

Example

```
#include <iostream>
int main() {
   // Allocate memory for an integer
   int* ptr = new int(10);
   // Access the value
   std::cout << *ptr << std::endl; // Output: 10
   // Allocate memory for an array of integers
   int* arr = new int[5];
   // Initialize the array
   for (int i = 0; i < 5; i++) {
      arr[i] = i + 1;
   }
   // Print the array
   for (int i = 0; i < 5; i++) {
      std::cout << arr[i] << " ";
   }
   std::cout << std::endl;
```

```
// Deallocate memory
delete ptr;
delete[] arr;
return 0;}
```

Important Notes

Always use **delete** to de-allocate memory allocated with **new** to avoid memory leaks. For array allocation, use **delete []** to de-allocate the entire array. Consider using **smart pointers** like **unique_ptr** and **shared_ptr** to *automatically* manage memory and prevent memory leaks.
Be aware of potential exceptions like **std::bad_alloc** that might be thrown during allocation.

✓ **The delete operator**

The **delete** operator in C++ is used to de-allocate memory that was previously allocated using the **new** operator. When you allocate memory with new, you're essentially telling the program to reserve a specific amount of space in memory for your data. Once you're done using that memory, you need to **release** it back to the system so it can be used for other purposes. This is where the **delete** operator comes in.

Syntax:

The syntax for the delete operator is:

```
delete pointer_variable;
```

pointer_variable: This is a pointer that points to the memory location you want to de-allocate. It must be the same type of pointer you used with new to allocate the memory.

Examples

Deleting a single variable

```
int* p = new int;
*p = 42;
// ... use p
delete p; // Deallocates the memory pointed to by p
```

Deleting an array

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

```
// ... initialize arr
delete[] arr; // Deallocates the entire array
```

✓ **Important points**

- You should always use delete to deallocate memory that you've allocated with new. Failing to do so can lead to memory leaks, which can cause your program to run out of memory and crash.
- It's important to only delete a pointer once. Deleting a pointer multiple times can lead to undefined behavior.
- If you have a pointer that points to multiple objects, you need to delete each object individually.
- If you have a pointer that points to an array, you need to use the delete[] operator to deallocate the entire array.

✓ **Handling Null in C++**

✓ **Understanding Null**

Null pointer is a pointer that doesn't point to a valid memory location. It's often represented by the macro nullptr (C++11 onwards) or null (in C and C++) or the value Ø.

Null reference is a reference that doesn't refer to an object. This is not possible in C++ and is a compile-time error.

✓ **Common Approaches to Handling Null**

 **Pointer Checks**

Explicitly check for NULL: Before dereferencing a pointer, always verify that it's not NULL. This prevents access violations and crashes.

Example

```
int* ptr = nullptr;
if (ptr != nullptr) {
   // Access the object pointed to by ptr
   int value = *ptr;
} else {
   // Handle the case where ptr is null
   std::cout << "Pointer is null." << std::endl;
}
```

 **Null-Safe Functions:**

Design functions that gracefully handle null arguments and return appropriate values or provide error messages.

Example

```
int safe_strlen(const char* str) {
   if (str == nullptr) {
      return 0; // Or throw an exception
   }
   return strlen(str);
}
```

Defensive Programming

Assume that null values can occur and write code accordingly.
Use defensive programming techniques like assertions and error handling to catch potential null pointer issues.

Example

```
void process_data(const int* data, int size) {
   assert(data != nullptr); // Check for null data pointer
   // ... process data ...
}
```

✓ **Smart Pointers**

Use smart pointers like **std::unique_ptr** and **std::shared_ptr** to automatically manage memory and handle null values safely.

These pointers provide features like automatic de-allocation and null checking, reducing the risk of memory leaks and null pointer exceptions.

Example

```
std::unique_ptr<int> ptr = std::make_unique<int>(10);
// ... use ptr ...
// ptr is automatically deleted when it goes out of scope
```

Error Handling

Implement robust error handling mechanisms to deal with null values gracefully.
This might involve throwing exceptions, returning error codes, or providing informative error messages.

Example

```
void process_data(const int* data) {
   if (data == nullptr) {
```

```
        throw std::invalid_argument("Data cannot be null");
    }
    // ... process data ...
}
```

Global program example

```
#include <iostream>
#include <memory>
int main() {
    // Explicit check
    int* ptr = nullptr;
    if (ptr != nullptr) {
        std::cout << *ptr << std::endl;
    } else {
        std::cout << "Pointer is null" << std::endl;
    }

    // Smart pointer
    std::unique_ptr<int> smartPtr = std::make_unique<int>(42);
    std::cout << *smartPtr << std::endl; // No need to check for null
    // Defensive programming
    int* anotherPtr = new int;
    if (anotherPtr != nullptr) {
        *anotherPtr = 100;
        std::cout << *anotherPtr << std::endl;
        delete anotherPtr;
    } else {
        std::cerr << "Memory allocation failed" << std::endl;
    }

    return 0;
}
```

j. Smart pointers

Smart pointers are a C++ language feature designed to **automatically** manage memory allocation and de-allocation, thereby preventing common memory-related errors like memory leaks and dangling pointers.

They encapsulate a raw pointer and provide additional functionality to ensure memory safety and efficiency.

✓ **How Smart Pointers Work**

Ownership

When a smart pointer is created, it takes ownership of the object it points to. This means it's responsible for de-allocating the object's memory when it's no longer needed.

Reference Counting

Most smart pointers use a reference counting mechanism. This means they keep track of the number of references to the object. When the reference count becomes zero, the object is automatically deleted.

Automatic De-allocation

When a smart pointer goes out of scope or is explicitly deleted, its destructor is called. If the reference count is zero, the destructor will delete the object it points to.

✓ **Types of Smart Pointers in C++**

std::unique_ptr

Provides exclusive ownership of the object it points to.
Only one **unique_ptr** can point to a given object at a time.
Automatically deletes the object when the **unique_ptr** is destroyed.
Can be moved but not copied.

std::shared_ptr

It allows multiple shared_ptr objects to share ownership of the same object.
Uses reference counting to determine when the object should be deleted.
Automatically deletes the object when the reference count becomes zero.
Can be copied and moved.

std::weak_ptr

It is a non-owning pointer that can be used to observe an object owned by a **shared_ptr**.

It does not increment the reference count.

It can be used to prevent circular references.

It requires a **shared_ptr** to access the object it points to.

✓ **When to Use Which Smart Pointer**

unique_ptr: When you need exclusive ownership of an object and want to avoid accidental copying.
shared_ptr: When multiple objects need to share ownership of a resource.

weak_ptr: To prevent circular references and to observe an object without taking ownership.

Example

```
#include <memory>
#include <iostream>
int main() {
   // Create a unique_ptr to a dynamically allocated integer
   std::unique_ptr<int> ptr = std::make_unique<int>(42);
   // Access the value through the pointer
   std::cout << "Value: " << *ptr << std::endl;
   // The unique_ptr automatically deletes the allocated memory when it goes out of
scope
   // No need for manual deletion
   return 0;
}
```

k. **Memory Leaks in C++**

A memory leak occurs when a program allocates memory dynamically but fails to deallocate it properly, leading to the memory being unavailable for reuse. This can cause performance issues and, in severe cases, program crashes.

Common causes of memory leaks

- ✓ **Unmatched new and delete**

Forgetting to use delete to de-allocate memory allocated with new is a common cause of memory leaks.

- ✓ **Incorrect pointer arithmetic**

When going out of bounds while accessing elements in an array or allocated memory can lead to memory leaks.

- ✓ **Circular references**

In data structures like linked lists or trees, circular references can prevent the memory from being de-allocated.

- ✓ **Exceptions**

If an exception is thrown after memory allocation but before deallocation, the memory may be leaked.

Preventing memory leaks

- ✓ **Smart pointers**

Smart pointers like **unique_ptr** and **shared_ptr** automatically manage memory, reducing the risk of memory leaks.

- ✓ **Match new and delete**

Always use delete to deallocate memory allocated with new.

- ✓ **Avoid circular references**

Carefully design data structures to prevent circular references.

- ✓ **Handle exceptions properly**

Ensure that memory is deallocated even if exceptions occur.

- ✓ **Use a memory leak detector**

Tools like **Valgrind** can help identify memory leaks in your programs.

Example:

```
#include <iostream>
int main() {
   int *ptr = new int;
   *ptr = 10;
   // ...
   delete ptr; // Deallocate memory
}
```

Additional tips:

Use unique_ptr for exclusive ownership: unique_ptr takes ownership of the allocated memory and automatically deallocates it when the unique_ptr goes out of scope.
Use shared_ptr for shared ownership: shared_ptr allows multiple objects to share ownership of the allocated memory, and it automatically deallocates the memory when the last shared_ptr goes out of scope.

- **Array of pointers**

In C++, an array of pointers is a data structure that stores a collection of pointers to other data types. Each element in the array is a pointer, pointing to a specific memory location where a value of the corresponding data type is stored.

- ✓ **Declaration**

To declare an array of pointers, you specify the data type of the elements being pointed to, followed by the array size

Data_type*array_name[array_size];

Example

int *ptr_array[5];// to declare an array of pointers to integers

char * myname[30];// array of 30 pointers to characters

- ✓ **Initialization**

You can initialize the array elements with pointers to existing variables or dynamically allocated memory

Examples

```
int a = 10, b = 20, c = 30;

int *ptr_array[5];
ptr_array[0] = &a;
ptr_array[1] = &b;
ptr_array[2] = &c;
```

- ✓ **Accessing Elements**

To access an element in an array of pointers, you use the array index, followed by the dereference operator (*)

Example

```
int value = *ptr_array[2]; // Access the value pointed to by the third element
```

- ✓ **Dynamic Allocation**

You can dynamically allocate memory for the elements pointed to by the array using **new**

Example

```
for (int i = 0; i < 5; i++) {
   ptr_array[i] = new int;
   *ptr_array[i] = i * 10;
}
```

- ✓ **De-allocation**

Remember to de-allocate the dynamically allocated memory using **delete** when you're done with it

Example

```
for (int i = 0; i < 5; i++) {
   delete ptr_array[i];
}
```

✓ **Key Points**

- Each element in an array of pointers is a pointer, pointing to a memory location.
- You can store pointers to variables or dynamically allocated memory.
- Use the dereference operator (*) to access the value pointed to by a pointer.
- Dynamically allocate memory using **new** and deallocate it using **delete**.

Program example

```
#include <iostream>
int main() {
   int *ptr_array[3];
   // Initialize with pointers to variables
   int x = 10, y = 20, z = 30;
   ptr_array[0] = &x;
   ptr_array[1] = &y;
   ptr_array[2] = &z;
   // Access and print the values
   for (int i = 0; i < 3; i++) {
      std::cout << *ptr_array[i] << std::endl;
   }
   // Dynamically allocate memory
   for (int i = 0; i < 3; i++) {
      ptr_array[i] = new int;
      *ptr_array[i] = i * 10;
   }
   // Access and print the values
   for (int i = 0; i < 3; i++) {
      std::cout << *ptr_array[i] << std::endl;
   }
```

```
    // Deallocate memory
    for (int i = 0; i < 3; i++) {
        delete ptr_array[i];
    }
    return 0;
}
```

l. Array of function pointers

A function pointer is a variable that stores the memory address of a function. This allows you to dynamically call functions based on their addresses, providing flexibility and reusability in your code.

✓ **Declare array of function pointers**

To declare an array of function pointers in C++, you need to specify the return type and parameter list of the functions they will point to.

Syntax:

<return_type>(*function_pointers[array_size])(<parameter_list>);

Example

If you have functions **int add(int, int)** and **int subtract(int, int)**, you can create an array of function pointers to them as follows:

```
int (*math_functions[2])(int, int);
math_functions[0] = add;
math_functions[1] = subtract;
```

✓ **Using Function Pointers in an Array**

Once you've created the array, you can access and call the functions through the pointers:

```
int result = math_functions[0](5, 3); // Calls add(5, 3)
std::cout << "Result: " << result << std::endl;
```

✓ **Key Points**

The **return type** and **parameter list** of the functions pointed to must match the declaration of the function pointer array.

You can initialize the array elements with the addresses of functions using the function name without parentheses.

To call a function through a pointer, use the pointer name followed by parentheses with the appropriate arguments.

Example

Generic calculator

```
#include <iostream>
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
int divide(int a, int b) { return a / b; }
int main() {
   int  (*math_operations[4])(int, int) = {add, subtract, multiply, divide};
   int num1, num2, choice;
   std::cout << "Enter two numbers: ";
   std::cin >> num1 >> num2;
   std::cout << "Choose an operation:\n";
   std::cout << "1. Add\n2. Subtract\n3. Multiply\n4. Divide\n";
   std::cin >> choice;
   int result = math_operations[choice - 1](num1, num2);
   std::cout << "Result: " << result << std::endl;
   return 0;
}
```

Practical Activity 3.1.2: Applying pointers

Task:

1: Read and perform the task below:

As a firmware development technician you are tasked to Write and Run a C++ Program for managing student grades using dynamic memory and pointers

2: Read Key readings **3.1.2** in trainee manual

3: Perform the task involving the process of applying pointers in writing C++ program described in task1.

4: Ask for assistance if needed

5: Verify the output whether it is the same as the one expected.

Key readings 3.1.2

Applying pointers

Write and Run a C++ Program for Managing Student Grades Using Dynamic Memory and Pointers

Steps

- Start the Dev C++
- Write the C++ Program

1. Include the Necessary Header Files: Start by including the iostream header, which is necessary for input and output operations.
   ```
   #include <iostream>
   using namespace std;
   ```
2. Define the main() Function: This will be the entry point of the program.
   ```
   int main() {
      // Your code will go here
   }
   ```
3. Prompt the User to Enter the Number of Students: Declare a variable for storing the number of students.
4. Dynamically Allocate Memory for Storing Grades: Use new to dynamically allocate memory for an array of student grades
   ```
   float* grades = new float[numStudents];  // dynamically allocate an array of floats
   ```

```
int numStudents;
cout << "Enter the number of students: ";
cin >> numStudents;
```

5. Input Grades Using Pointer Arithmetic:
   Use a loop to input grades and store them in the array using pointer arithmetic.

```
for (int i = 0; i < numStudents; ++i) {
   cout << "Enter grade for student " << (i + 1) << ": ";
   cin >> *(grades + i);  // pointer arithmetic
}
```

6. Display All Grades Using Pointer Arithmetic:
   Use a loop to display all grades by accessing them with a pointer.

```
cout << "\nGrades of all students:\n";
for (int i = 0; i < numStudents; ++i) {
   cout << "Student " << (i + 1) << ": " << *(grades + i) << endl;
}
```

7. Modify a Specific Student's Grade:
   Ask the user for the student number whose grade needs modification, then update the grade.

```
int studentIndex;
cout << "\nEnter the student number to modify the grade: ";
cin >> studentIndex;
cout << "Enter the new grade: ";
cin >> *(grades + studentIndex - 1);  // update the grade using pointer arithmetic
```

8. Re-Display All Grades After Modification:
   Show the updated list of grades.

```
cout << "\nUpdated Grades:\n";
for (int i = 0; i < numStudents; ++i) {
   cout << "Student " << (i + 1) << ": " << *(grades + i) << endl;
}
```

9. **Calculate and Display the Average Grade:**
   Use a loop to calculate the sum of all grades, then compute and display the average.

```
float sum = 0;
for (int i = 0; i < numStudents; ++i) {
   sum += *(grades + i);  // accumulate grades using pointer arithmetic
}
cout << "\nAverage Grade: " << (sum / numStudents) << endl;
```

10. Deallocate the Dynamically Allocated Memory:
    Release the memory using delete[].

```
delete[] grades;  // deallocate memory to avoid memory leaks
```

11. Return 0 to Indicate Successful Program Execution:

```
return 0;
```

Full code

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

int main() {
   // Step 1: Input the number of students
   int numStudents;
   cout << "Enter the number of students: ";
   cin >> numStudents;
   // Step 2: Dynamically allocate memory for grades
   float* grades = new float[numStudents];
   // Step 3: Input grades using pointer arithmetic
   for (int i = 0; i < numStudents; ++i) {
      cout << "Enter grade for student " << (i + 1) << ": ";
      cin >> *(grades + i);
   }
   // Step 4: Display all grades
   cout << "\nGrades of all students:\n";
   for (int i = 0; i < numStudents; ++i) {
      cout << "Student " << (i + 1) << ": " << *(grades + i) << endl;
   }
   // Step 5: Modify a student's grade
   int studentIndex;
   cout << "\nEnter the student number to modify the grade: ";
   cin >> studentIndex;
   cout << "Enter new grade for student " << studentIndex << ": ";
   cin >> *(grades + studentIndex - 1);
   // Step 6: Display updated grades
   cout << "\nUpdated Grades:\n";
   for (int i = 0; i < numStudents; ++i) {
      cout << "Student " << (i + 1) << ": " << *(grades + i) << endl;
   }
   // Step 7: Calculate the average grade
   float sum = 0;
   for (int i = 0; i < numStudents; ++i) {
      sum += *(grades + i);
   }
```

```
    cout << "\nAverage Grade: " << (sum / numStudents) << endl;
    // Step 8: Free dynamically allocated memory
    delete[] grades;
    return 0; }
```

- Compile the Program: Build or compile the project by clicking the "Build" button or using the shortcut (often F9 or Ctrl+B depending on the IDE).
- Run the Program: Run the program by clicking the "Run" button or using the appropriate shortcut (often F5 or Ctrl+R).
- Test the Program:

1. Input the Number of Students:

The program will prompt you to enter the number of students. Provide an input (e.g., 3).

2. Input Grades for Each Student:

The program will prompt for the grades of each student.

3. Modify a Grade:

After displaying the grades, the program will ask if you want to modify any specific grade.

4. Display the Average Grade: The program will display the average grade of all students.

- Check the outputs

Verify if the outputs are the similar to something like:

```
Enter the number of students: 5
Enter grade for student 1: 46
Enter grade for student 2: 89
Enter grade for student 3: 34
Enter grade for student 4: 67
Enter grade for student 5: 89

Grades of all students:
Student 1: 46
Student 2: 89
Student 3: 34
Student 4: 67
Student 5: 89

Enter the student number to modify the grade: 2
Enter new grade for student 2: 100

Updated Grades:
Student 1: 46
Student 2: 100
Student 3: 34
Student 4: 67
Student 5: 89

Average Grade: 67.2

--------------------------------
Process exited after 30.94 seconds with return value 0
Press any key to continue . . .
```

Points to Remember

Description of arrays

Here bellow is a summary of key points that should help you understanding effectively the use of pointers and dynamic memory allocation in your C++ programs.

A pointer as a variable that stores the memory address of another variable has to be declared before being used. To declared using the following **syntax:**

data_type *pointer_name; (e.g., int *p;)

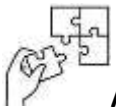
It is common knowledge that to access values stored in the pointer is referred to as dereferencing a pointer. The pointer can point to invalid address, so it is called null pointer. Initialize a pointer to **nullptr** or valid address to avoid undefined behaviours. You can perform arithmetic operations on pointers, but they are interpreted in the terms of size of data type they point to. This action is referred to as pointer arithmetic.

The primary purpose of dynamic memory allocation is allocating a memory at runtime based on the program needs using the new operator to allocate and the delete to de-allocate it. You can also use the **new[]** and **delete**[] operator to allocate and de-allocate array pointers For memory management; avoid memory leaks by de-allocating memory when it is no longer needed. You can also use smart pointers to automatically manage the memory.

Applying pointers

When running a C++ program that should manage students' grades using dynamic memory and pointers follow these steps

- Start the Dev C++ as a C++ IDE
- Write the C++ Program by first include the **iostream** header file, define the main function as the main part of the C++ program. At this step you need to
  - ✓ Prompt the User to Enter the Number of Students
  - ✓ Dynamically Allocate Memory for Storing Grades
  - ✓ Input Grades Using Pointer Arithmetic
  - ✓ Input Grades Using Pointer Arithmetic
  - ✓ Display All Grades Using Pointer Arithmetic
  - ✓ Modify a Specific Student's Grade
  - ✓ Re-Display All Grades After Modification
  - ✓ Calculate and Display the Average Grade
  - ✓ Deallocate the Dynamically Allocated Memory and
  - ✓ Return 0 to Indicate Successful Program Execution
- Compile the Program and run the code
- Test the Program and check the outputs**.**



Application of learning: 3.1.

At ABC TSS they want a student management system that can dynamically store information about students (name, age, and grade). This system should allow for adding new students, removing existing students, and updating student information. As a firmware development technician, you are tasked to create and run a C++ program that simulates the above described system by implementing pointers and dynamic memory allocation.

Indicative content 3.2: Apply file handling

Duration: 5 hrs

Theoretical Activity 3.2.1: Description of file handling

Tasks:

1: Answer the questions reflecting to C++ file handling:

1) What do you mean by file in C++?
2) Define the following :
    a. Header file
    b. File path
    c. File opening mode
    d. File handling
3) How do you open a file in C++?
4) Discuss the role of fstream header file in file handling in C++.

2: Write tyour key findings on papers/flipchart

3: Present your findings to the whole class

4: Ask clarifications if any

5: Read through Key readings 3.2.1 in the trainee manual

Key readings 3.2.1.

Descrcription of file handling

ii.Definition

✓ **File**

A file in C++ is a sequence of bytes stored on a disk or other secondary storage device. It's a fundamental unit for storing and organizing data.

✓ **Header file**

A header file is a file containing C++ declarations (such as class definitions, function declarations, and variable declarations) that are used by other source files. These declarations are typically compiled separately from the main

program code and then included in the main program using the **#include** statement

✓ **File Handling**

In programming, file handling refers to the process of interacting with files on a computer system. This involves operations like:

- Creating new files
- Reading data from existing files
- Writing data to files
- Updating existing file content
- Deleting files

✓ **File Paths**

File path also known as a directory path or pathname, is a string of characters that specifies the location of a file or directory within a file system. It's essentially the address of a file on your computer.

✓ **File Modes**

When working with files in programming, a file mode determines how the file is opened. It specifies the permissions and operations allowed on the file. (e.g., read-only, write-only, read-write).

✓ **File I/O Operations**

File I/O (Input/Output) operations refer to the process of reading data from or writing data to files. These operations are essential for interacting with files in programming.

iii.Importance In C++ Programming

File handling is a crucial aspect of C++ programming, allowing for the interaction and management of data stored on external storage devices. This functionality is essential for various applications, including:

✓ **Data Persistence**

Saving and Retrieving Data

File handling enables programs to store data persistently, ensuring its availability even after the program terminates.

- **Creating Databases**

Files can be used to create and maintain databases, providing a structured way to store and organize large amounts of information.

- **Data Exchange**
- ✓ **Importing and Exporting Data**

Files can be used to import data from external sources or export data for further analysis or processing.

- ✓ **Interoperability**

File formats like CSV or XML can facilitate data exchange between different applications and systems.

- ✓ **Program Configuration**
- **Storing Settings**

Programs can use files to store user preferences, configuration settings, and other parameters.

- **Customization**

Users can customize the behavior of applications by modifying these settings files.

- **Logging and Debugging**
- ▪ **Recording Events**

Programs can log events, errors, and debugging information to files for later analysis.

- ▪ **Troubleshooting**

Logs can help developers identify and fix issues in their applications.

- ✓ **Other Applications**
- **Text Processing**

File handling is fundamental for tasks like reading, writing, and manipulating text files.

- **Image and Audio Processing**

Files are used to store and process multimedia data such as images, audio, and video.

iv. Benefits of File I/O

File I/O (Input/Output) is an essential part of programming that allows applications to interact with files stored on a computer. The benefits of File I/O include:

- ✓ **Data Persistence**

File I/O enables the storage of data beyond the runtime of a program. Unlike data stored in memory (RAM), which is temporary and lost when a program ends, data written to files remains available for future use.

- ✓ **Large Data Handling**

Files can store large volumes of data that would be impractical to hold entirely in memory. This is particularly useful for applications that need to process large datasets or logs.

- ✓ **Data Sharing**

Files can be easily shared and transferred between different systems or users. This makes them an effective way to exchange information or distribute software components.

- ✓ **Logging and Debugging**

File I/O is commonly used for logging program activities, errors, or debugging information. By writing logs to a file, developers can keep track of how their applications behave over time and diagnose issues.

- ✓ **Configuration Management**

Many applications use files (such as text or XML files) to store configuration settings. This allows users to modify the program's behavior without altering its source code.

✓ **Backup and Recovery**

Files can be used to create backups of important data. Applications can periodically save data to files, ensuring that a recoverable copy exists in case of failures or crashes.

✓ **Data Analysis**

File I/O is essential in data analysis tasks, as it allows reading data from various sources (such as CSV, JSON, or XML files) for processing, analysis, and visualization.

✓ **Modularity**

By reading from and writing to files, programs can be designed to work in a modular way. Different components of an application can generate or consume data stored in files, making it easier to develop, test, and maintain.

✓ **Interoperability**

File formats like text, CSV, JSON, and XML are widely recognized and used, allowing programs written in different languages or running on different platforms to exchange data effectively.

v.File types

Here's a list of common file types and their corresponding extensions often used in programming

✓ **Text Files**
- **Plain Text**: .txt
- **HTML**: .html, .htm
- **CSS**: .css
- **JavaScript**: .js
- **Python**: .py
- **Java**: .java
- **C/C++:** .c, .cpp
- **Ruby:** .rb
- **PHP**: .php
- **Markdown**: .md
- **JSON:** .json
- **XML**: .xml

- **YAML**: .yaml
- **Rich Text Format (RTF)**: .rtf
- **Microsoft Word**: .docx (newer versions), .doc (older versions)
- **OpenDocument Text**: .odt
- **PDF (Portable Document Format):** .pdf

✓ Spreadsheet Files

Microsoft Excel: .xlsx (newer versions), .xls (older versions)
OpenDocument Spreadsheet: .ods
CSV (Comma-Separated Values): .csv

✓ Presentation Files

Microsoft PowerPoint: .pptx (newer versions), .ppt (older versions)
OpenDocument Presentation: .odp

✓ **Binary Files**

- **Images**:

JPEG (Joint Photographic Experts Group): .jpg or .jpeg
PNG (Portable Network Graphics): .png
GIF (Graphics Interchange Format): .gif
BMP (Bitmap): .bmp
TIFF (Tagged Image File Format): .tif or .tiff

- Audio Files:

MP3 (MPEG-1 Audio Layer III): .mp3
WAV (Waveform Audio File Format): .wav
AAC (Advanced Audio Coding): .aac
FLAC (Free Lossless Audio Codec): .flac

- Video Files

MP4 (MPEG-4 Part 14): .mp4
MOV (QuickTime Movie): .mov
AVI (Audio Video Interleave): .avi
WMV (Windows Media Video): .wmv

- Databases: .db, .sqlite, .sql
- Executable Files: .exe (Windows), .sh (Linux/macOS)

✓ Other types

- Configuration Files: .ini, .conf, .properties
- Archive Files: .zip, .rar, .tar, .gz
- Library Files: .dll (Windows), .so (Linux/macOS)

Theoretical Activity 3.2.2: Description of header file and Stream Classe

Tasks:

1: Answer the questions reflecting to C++ pointers:

1. What do you mean by pointer in C++?
2. Define the following :
   a) Header file
   b) Class
3. How do you create a file in C++?
4. Describe different file opening mode in C++.

2: write your key findings on paper/flipchart

3: Presentation of the findings

4: Ask questions for more clarifications or provide concerns

5: Read through Key readings 3.2.2 for additional clarifications

Key readings 3.2.2

Description of header file and Stream Classe

1. Introduction to fstream Header file

The header file used for file handling in C++ is <fstream>. The fstream header file in C++ provides classes for file input and output operations. It includes the following classes:

ifstream: For input file stream (reading files).
ofstream: For output file stream (writing to files).
fstream: For both input and output file streams (reading and writing to files).

2. Role

The fstream header file in C++ provides a class template for file stream operations, allowing you to read from and write to files. It combines the functionality of both input and output file streams (ifstream and ofstream).

3. Accessing fstream header file

To include the fstream header file in your C++ code, use the following preprocessor directive:
syntax:

#include<fstream>

4. Functions of fstream header file

✓ **open():**

The **open()** function in C++ is used with file stream objects (std::ifstream, std::ofstream, and std::fstream) to open a file. Here's the general syntax of the open function:

Syntax :

fileStreamObject.open("filename",mode);

Where :

fileStreamObject: This is an instance of either std::ifstream, std::ofstream, or std::fstream.

filename: A string representing the name of the file to open.

mode (optional): Specifies how the file should be opened

Examples:

Opening a File for Writing (Using std::ofstream)

```
#include <fstream>
int main() {
   std::ofstream outFile;
   outFile.open("example.txt", std::ios::out | std::ios::app);  // Open in write and
append mode

   if (outFile.is_open()) {
      outFile << "Hello, World!" << std::endl;
      outFile.close();
   }
```

```
    return 0;
}
```

- Opening a File for Reading (Using std::ifstream)

```
#include <fstream>
int main() {
    std::ifstream inFile;
    inFile.open("example.txt", std::ios::in);  // Open in read mode
 if (inFile.is_open()) {
        std::string line;
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }
        inFile.close();
    }

    return 0;
}
```

- Opening a File for Both Reading and Writing (Using std::fstream)

```
#include <fstream>
int main() {
    std::fstream file;
    file.open("example.txt", std::ios::in | std::ios::out | std::ios::app);  // Open in read, //write, and append mode
    if (file.is_open()) {
        file << "Additional text." << std::endl;  // Write to the file
        file.seekg(0);  // Move cursor to the beginning for reading
        std::string line;
        while (std::getline(file, line)) {
            std::cout << line << std::endl;  // Read from the file
        }

        file.close();
    }
    return 0;
}
```

✓ **close()**

The close function in C++ is used to close an open file stream. Closing a file is important to ensure all data is properly written to the file and to release system resources associated with the file

Syntax:

fileStreamObject.close();
Where

fileStreamObject: This is an instance of either std::ifstream, std::ofstream, or std::fstream that has been used to open a file

Example:


Closing a File Opened for Writing

```
#include <fstream>
int main() {
   std::ofstream outFile;
   outFile.open("example.txt");
   if (outFile.is_open()) {
      outFile << "Hello, World!" << std::endl;
      outFile.close();  // Closing the file
   }
   return 0;
}
```


Closing a File Opened for Reading

```
#include <fstream>
int main() {
   std::ifstream inFile;
   inFile.open("example.txt");
   if (inFile.is_open()) {
      std::string line;
      while (std::getline(inFile, line)) {
         std::cout << line << std::endl;
      }
      inFile.close();  // Closing the file
   }
   return 0;
}
```


Closing a File Opened for Both Reading and Writing

```
#include <fstream>
int main() {
```

```
    std::fstream file;
    file.open("example.txt", std::ios::in | std::ios::out);
    if (file.is_open()) {
        file << "Adding text to the file." << std::endl;
        file.seekg(0);
        std::string line;
        while (std::getline(file, line)) {
            std::cout << line << std::endl;
        }
        file.close();  // Closing the file
    }
    return 0;
}
```

Note:

It's good practice to always call close() once you're done working with a file to ensure all data is properly flushed to disk and resources are released.

If you forget to close a file, the destructor of the file stream object will automatically close it when the object goes out of scope, but explicitly closing it is considered a best practice.

✓ **read()**

In C++, the read function is used to read binary data from a file into a buffer. It is typically used with input file streams (std::ifstream or std::fstream). The read function is designed to handle raw binary data, so it's often used when dealing with files that contain binary formats rather than plain text.

Syntax

fileStream.read(char*buffer,std::streamsize size);

fileStream: An instance of either std::ifstream or std::fstream opened in binary mode (std::ios::binary).
buffer: A pointer to a character array where the data will be stored.
size: The number of bytes to read from the file.

Examples

Writing and Reading Binary Data Using read

```
#include <iostream>
#include <fstream>
int main() {
   // Data to writeconst char data[] = "Hello, Binary File!";
     // Writing binary data to a file
   {
      std::ofstream outFile("binaryfile.bin", std::ios::binary);
      if (outFile.is_open()) {
         outFile.write(data, sizeof(data));  // Write binary data to the file
         outFile.close();
      } else {
         std::cerr << "Error opening file for writing." << std::endl;
      }
   }
   // Buffer to read data into
   char buffer[50];  // Assuming the data size will fit into this buffer
   // Reading binary data from the file
   {
      std::ifstream inFile("binaryfile.bin", std::ios::binary);
      if (inFile.is_open()) {
         inFile.read(buffer, sizeof(data));  // Read binary data into buffer
         inFile.close();

         // Print the data read from the file
         std::cout << "Data read from file: " << buffer << std::endl;
      } else {
         std::cerr << "Error opening file for reading." << std::endl;
      }
   }   return 0;
}
```

Note:

The read function is commonly used when working with files that store data in binary formats, such as images, serialized objects, or any non-textual data that must be read as raw bytes.

✓ **write()**

In C++, the write function is used to write binary data to a file. It is typically used with output file streams (std::ofstream or std::fstream) and allows for the direct writing of raw binary data to a file.

Syntax:

fileStream.write(const char*buffer, std::streamsize, size);

fileStream: An instance of either std::ofstream or std::fstream opened in binary mode (std::ios::binary).

buffer: A pointer to a character array (or raw memory buffer) that contains the data to be written to the file.

size: The number of bytes to write from the buffer to the file.

Example

Writing Binary Data Using write

```
#include <iostream>
#include <fstream>
int main() {
   // Data to write to the file
   const char data[] = "Hello, Binary World!";
     // Create an ofstream object and open a file in binary mode
   std::ofstream outFile("binaryfile.bin", std::ios::binary);
     // Check if the file is open
   if (outFile.is_open()) {
      // Write binary data to the file
      outFile.write(data, sizeof(data));
           // Close the file
      outFile.close();
      std::cout << "Data written to binary file successfully!" << std::endl;

} else {
      std::cerr << "Error opening file for writing." << std::endl;
   }
   return 0;
}
```

Notes:

The write function is commonly used for writing binary data to files, such as images, serialized objects, and other non-textual data formats.

✓ **seekg()**

The **seekg** function in C++ is used with input file streams (std::ifstream or std::fstream) to move the "get" pointer to a specific position within the file. This pointer determines the location in the file from where the next read operation will occur.

Syntax :

fileStream.seekg(position,direction);

Where

fileStream: An instance of std::ifstream or std::fstream.
position: An integer value specifying the number of bytes to move the pointer.
direction (optional): A constant that specifies the reference point for the movement. The possible values are:

- **std::ios::beg (default):** Beginning of the file.
- **std::ios::cur:** Current position of the pointer.
- **std::ios::end:** End of the file.

Example

Moving to a Specific Position in the File

```
#include <iostream>
#include <fstream>
int main() {
   // Open a file for reading
   std::ifstream inFile("example.txt", std::ios::binary);
   if (inFile.is_open()) {
      // Move the "get" pointer to the 5th byte from the beginning
      inFile.seekg(5, std::ios::beg);
      // Read the next character from the new position
      char ch;
      inFile.get(ch);
      std::cout << "Character at position 5: " << ch << std::endl;
      inFile.close();
   } else {
```

```
        std::cerr << "Error opening file." << std::endl;
    }
    return 0;
}
```

Moving to the End of the File

```
#include <iostream>
#include <fstream>
int main() {
    // Open a file for reading
    std::ifstream inFile("example.txt", std::ios::binary);
    if (inFile.is_open()) {
        // Move to the end of the file
        inFile.seekg(0, std::ios::end);
        // Get the position of the pointer (size of the file)
        std::streampos fileSize = inFile.tellg();
        std::cout << "File size: " << fileSize << " bytes." << std::endl;
        inFile.close();
    } else {
        std::cerr << "Error opening file." << std::endl;
    }
    return 0;
}
```

Note:

You can use seekg to jump to specific parts of a file, making it useful when working with large files or binary data.

After using seekg, you can use the tellg() function to get the current position of the "get" pointer.

✓ **seekp()**

The seekp function in C++ is used with output file streams (std::ofstream or std::fstream) to move the "put" pointer to a specific position in the file. This pointer determines the location in the file where the next write operation will occur.

Syntax:

fileStream.seekp(position,direction);

Where

fileStream: An instance of std::ofstream or std::fstream.
position: An integer value specifying the number of bytes to move the pointer.
direction (optional): A constant that specifies the reference point for the movement. Possible values are:

- **std::ios::beg (default):** Beginning of the file.
- **std::ios::cur:** Current position of the pointer.
- **std::ios::end:** End of the file.

Examples

Writing at a Specific Position in a File

```
#include <iostream>
#include <fstream>
int main() {
   // Open a file for writing
   std::ofstream outFile("example.txt", std::ios::binary);
   if (outFile.is_open()) {
      // Move the "put" pointer to the 5th byte from the beginning
      outFile.seekp(5, std::ios::beg);
      // Write data at the new position
      outFile << "XYZ";
      // Close the file
      outFile.close();
   } else {
      std::cerr << "Error opening file." << std::endl;
   }
   return 0;
}
```

Appending Data to the End of a File

```
#include <iostream>
#include <fstream>
int main() {
   // Open a file for both reading and writing
   std::fstream file("example.txt", std::ios::in | std::ios::out | std::ios::binary);
   if (file.is_open()) {
      // Move the "put" pointer to the end of the file
      file.seekp(0, std::ios::end);
```

```
        // Write data at the end of the file
        file << "Appended Text";
        // Close the file
        file.close();
    } else {
        std::cerr << "Error opening file." << std::endl;
    }
    return 0;
}
```

Note:

Binary Mode: When working with binary data, ensure the file is opened in binary mode (std::ios::binary).

tellp(): After using seekp, you can use the tellp() function to get the current position of the "put" pointer.

✓ **tellg()**

The tellg function in C++ is used with input file streams (std::ifstream or std::fstream) to get the current position of the "get" pointer, which indicates where the next read operation will occur in the file.

Syntax:

std::streampos position=fileStream.tellg();

Where

fileStream: An instance of std::ifstream or std::fstream.
position: A variable of type std::streampos that stores the current position of the "get" pointer in the file.

Example

 Getting the Current Read Position

```
#include <iostream>
#include <fstream>
int main() {
    // Open a file for reading
```

```
    std::ifstream inFile("example.txt", std::ios::binary);
    if (inFile.is_open()) {
        // Read a few bytes
        char ch;
        inFile.get(ch);  // Read the first character
        // Get the current position of the "get" pointer
        std::streampos position = inFile.tellg();
        std::cout << "Current position of 'get' pointer: " << position << std::endl;
        // Close the file
        inFile.close();
    } else {
        std::cerr << "Error opening file." << std::endl;
    }
    return 0;
}
```

Note:

Return Value: The tellg function returns -1 if an error occurs, so it is good practice to check the return value before using it.
Binary Files: tellg is often used with binary files to determine offsets when reading specific data chunks.

✓ **tellp()**

The **tellp** is useful for determining the current position within an output stream. It can be used to calculate the number of bytes written to the stream.The tellp returns a streampos object, which can be used for various purposes, such as seeking within the stream.

Syntax:

std::streampos position=fileStream.tellp();

Where

fileStream: An instance of std::ifstream or std::fstream.
position: A variable of type std::streampos that stores the current position of the "get" pointer in the file.

Example

```
#include <iostream>
#include <fstream>

using namespace std;
int main() {
   ofstream outputFile("example.txt");
   if (outputFile.is_open()) {
      outputFile << "Hello, world!";
      // Get the current position
      streampos currentPosition = outputFile.tellp();
      cout << "Current position: " << currentPosition << endl;
      // Write more data
      outputFile << " This is additional text.";
      // Get the new position
      currentPosition = outputFile.tellp();
      cout << "New position: " << currentPosition << endl;
      outputFile.close();
   } else {
      cout << "Error opening file." << endl;
   }
   return 0;
}
```

✓ **eof()**

In C++, the eof() function is used to check if the end of a file has been reached while reading from a stream, such as ifstream for file input. It belongs to the istream class and can be used with various input streams, including cin and file streams (ifstream).

Syntax:

bool eof() const;

Where

- **eof()** returns a boolean (true or false)
- It returns true if the stream has reached the end of the file.
- It returns false otherwise.

Example

```
#include <iostream>
#include <fstream>
int main() {
   std::ifstream file("example.txt");
   if (!file) {
      std::cerr << "Error opening file!" << std::endl;
      return 1;
   }   char ch;
   while (!file.eof()) {
      file.get(ch);
      if (file) {  // Checks if the read was successful
         std::cout << ch;
      }
   }   file.close();
   return 0;
}
```

Notes

It is generally a good practice to check the success of the read operation itself (e.g., file.get() or file >> var) rather than relying solely on eof(). This is because eof() only becomes true after an attempt is made to read past the end of the file. This often leads to processing the last data twice if not handled correctly.

5. Main stream classes

In C++, the mainstream classes used for file handling within the <fstream> header are:

✓ **ofstream (Output File Stream)**

Used for creating files and writing data to files. It provides output operations, which write data to a file.

Example usage

```
#include <fstream>
std::ofstream outFile("example.txt");
outFile << "Writing to a file!";
outFile.close();
```

✓ **ifstream (Input File Stream)**

Used for reading data from files. It provides input operations, which read data from a file.

Example usage:

```
#include <fstream>
#include <iostream>
std::ifstream inFile("example.txt");
std::string line;
while (getline(inFile, line)) {
   std::cout << line << std::endl;
}
inFile.close();
```

✓ **fstream (File Stream)**

It combines the capabilities of both ofstream and ifstream. It can be used for both reading from and writing to files.

Example usage

```
#include <fstream>
#include <iostream>
std::fstream file("example.txt", std::ios::in | std::ios::out | std::ios::app);
if (file.is_open()) {
   // Writing to the file
   file << "Appending text to the file." << std::endl;

   // Moving the file pointer to the beginning
   file.seekg(0);
   // Reading from the file
   std::string line;
   while (getline(file, line)) {
      std::cout << line << std::endl;
   }
   file.close();
}
```

6. File opening modes

In C++, file opening modes define how a file is to be opened (read, write, append, etc.). These modes are used with file stream objects (ifstream, ofstream, and fstream) to specify the file's behavior when it is opened. These modes are typically defined as constants in the **ios** namespace and can be combined using bitwise OR operations.

Common file opening modes

a) The ios::in – Input (Read) Mode

It opens the file for reading and it is commonly used with **ifstream** or **fstream** in which the file must exist; otherwise, the operation will fail.

Example:

Reading from a file using **ios::in**

```
#include <fstream>
#include <iostream>
#include <string>
int main() {
   std::ifstream file("example.txt", std::ios::in);
   if (!file) {
      std::cerr << "Error opening file for reading!" << std::endl;
      return 1;
   }
   std::string line;
   while (std::getline(file, line)) {
      std::cout << line << std::endl;
   }
   file.close();
   return 0;
}
```

b) The ios::out – Output (Write) Mode

It opens the file for writing and is used with **ofstream** or **fstream** class. If the file exists, its content will be truncated (erased). If the file does not exist, it will be created.

Example

Writing to a file using **ios::out**

```
#include<iostream.h>

#include <fstream>
int main() {
   std::ofstream file("example.txt", std::ios::out);
   if (!file) {
      std::cerr << "Error opening file for writing!" << std::endl;
      return 1;
   }
   file << "Hello, World!" << std::endl;
   file.close();
   return 0;
}
```

c) **The ios::app – Append Mode**

It opens the file for writing in append mode. All write operations occur at the end of the file. That means the existing content is preserved, and new data is added to the end.

Example

Appending to a file using **ios::app**

```
#include<iostream.h>
#include <fstream>
int main() {
   std::ofstream file("example.txt", std::ios::app);
   if (!file) {
      std::cerr << "Error opening file for appending!" << std::endl;
      return 1;
   }
   file << "This line is appended to the file." << std::endl;
   file.close();
   return 0;
}
```

d) **The ios::ate – At End**

It opens the file and moves the file pointer to the end, allowing reading or writing at any position, but starts with the pointer at the end. It is used with **ifstream**, **ofstream,** or **fstream**.

Example

Opening a file with **ios::ate**

```
#include <fstream>
#include <iostream>
int main() {
   std::fstream file("example.txt", std::ios::in | std::ios::out | std::ios::ate);
   if (!file) {
      std::cerr << "Error opening file with ios::ate!" << std::endl;
      return 1;
   }
   // Write at the end of the file
   file << "\nAdding text at the end using ios::ate.";
   // Move back to the beginning to read the file
   file.seekg(0);
   std::string line;
   while (std::getline(file, line)) {
      std::cout << line << std::endl;
   }
   file.close();
   return 0;
}
```

e) **The ios::binary – Binary Mode**

It opens the file in binary mode, which means data is read and written in binary format. It is basically used for **non-text files**, such as images, videos, or any file where binary data manipulation is required.

Example

Writing and reading in binary mode using **ios::binary**

```
#include <fstream>
#include <iostream>
int main() {
```

```
    // Writing binary data
    std::ofstream outFile("binary.dat", std::ios::binary);
    if (!outFile) {
        std::cerr << "Error opening file for binary writing!" << std::endl;
        return 1;
    }
    int data = 12345;
    outFile.write(reinterpret_cast<char*>(&data), sizeof(data));
    outFile.close();
    // Reading binary data
    std::ifstream inFile("binary.dat", std::ios::binary);
    if (!inFile) {
        std::cerr << "Error opening file for binary reading!" << std::endl;
        return 1;
    }
    int readData;
    inFile.read(reinterpret_cast<char*>(&readData), sizeof(readData));
    inFile.close();
    std::cout << "Read binary data: " << readData << std::endl;
    return 0;
}
```

f) The ios::trunc – Truncate Mode

It truncates the file to zero length if it exists before opening it and is often used in conjunction with ios::out to delete the existing content before writing new data. If the file exists, it is truncated (erased) to zero length.

Example

Using ios::trunc to clear a file

```
#include <fstream>
#include<iostream.h>
int main() {
    std::ofstream file("example.txt", std::ios::out | std::ios::trunc);
    if (!file) {
        std::cerr << "Error opening file with ios::trunc!" << std::endl;
        return 1;
    }
    file << "This file has been truncated and overwritten." << std::endl;
```

```
    file.close();
    return 0;
}
```

Combining File Modes

You can combine multiple modes using the bitwise OR (|) operator to specify multiple behaviors for a file.

Example

Opening a file for both input and output in binary mode:

```
#include <fstream>
#include<iostream.h>
int main() {
    std::fstream file("example.dat", std::ios::in | std::ios::out | std::ios::binary);
    if (!file) {
        std::cerr << "Error opening file for both reading and writing in binary mode!"
<< std::endl;
        return 1;
    }
    // Perform read and write operations...
    file.close();
    return 0;
}
```

Summary table

Mode	Description
ios::in	Open for reading
ios::out	Open for writing(truncate if exists)
ios::app	Open for appending
ios::ate	Open and move to end of file
ios::binary	Open in binary mode

ios:: trunk	Truncate file if it exists

Practical Activity 3.2.3: Implementing file handling

Task:

1: Read and perform the following task:

> As a firmware development technician you are tasked to Write and Run a C++ Program to Save Text" Why are you willing to become a millionaire when you are still joking with cash? It's a pity!!" to Hard Storage

2: Read steps involved in applying file handling in C++ program from Key readings **3.2.3.**

3: Explore steps explained by trainer to initiate the process of applying file handling in writing C++ program.

4: Compile and run C++ program by following demonstrated steps and ask for assistance if needed.

5: Verify the output whether it is the same as the one expected.

Key readings 3.2.3

Applying file handling

Steps to apply file handling by Solving the the task: write and Run a C++ Program to Save Text" Why are you willing to become a millionaire when you are still joking with cash? It's a pity!!" to Hard Storage

Step 1: **Write the C++ Program**

1. **Open Your Code Editor or IDE**: Use a code editor like Visual Studio Code, Sublime Text, or an IDE like Code::Blocks, Visual Studio, or CLion.
2. **Create a New C++ Source File:** Create a new file and name it save_text.cpp.
3. **Include the Necessary Header Files**: Include iostream for input/output operations and fstream for file handling.

```
#include <iostream>
#include <fstream>
```

4. **Write the Main Function:**Start by creating the main() function, which is the entry point of every C++ program.

```
int main() {
   // Your code will go here
}
```

5. **Create and Open a File for Writing:** Inside the main() function, use std::ofstream to create an output file stream object.

```
std::ofstream outFile("output.txt");
```

6. **Check If the File Opened Successfully:** Check if the file was created or opened correctly.

```
if (!outFile) {
   std::cerr << "Error: Could not create or open the file!" << std::endl;
   return 1;  // Exit the program if the file couldn't be opened
}
```

7. **Write the Specified Text to the File:**Define the text and use the outFile object to write the text to the file.

```
std::string text = "Why are you willing to become a millionaire when you are still joking with cash? It's pitty!!";
outFile << text;
```

8. **Close the File :** Always close the file to ensure data is properly saved.

```
outFile.close();
```

9. **Display a Confirmation Message:** Let the user know that the text has been successfully saved.

```
std::cout << "Text has been saved to 'output.txt'." << std::endl;
```

10. **Complete the Program:** The final code should look like this:

```
#include <iostream>
#include <fstream>
int main() {
   // Create and open a file
   std::ofstream outFile("output.txt");
```

```
    // Check if the file opened successfully
    if (!outFile) {
        std::cerr << "Error: Could not create or open the file!" << std::endl;
        return 1;
    }
    // Text to write to the file
    std::string text = "Why are you willing to become a millionaire when you are
still joking with cash? It's pitty!!";
    // Write the text to the file
    outFile << text;
    // Close the file
    outFile.close();
    // Confirmation message
    std::cout << "Text has been saved to 'output.txt'." << std::endl;
    return 0;
}
```

Step 2: Compile the Program

1. **Compile the Program Using a C++ Compiler:** Use a C++ compiler like g++ to compile the program. This command compiles save_text.cpp and generates an executable named save_text.
2. **Check for Compilation Errors:** If there are any syntax errors or issues in the code, the compiler will display them. Fix the errors and recompile if needed.

Step 3: Run the Compiled Program

1. **Execute the Program**: using C++ compiler click run
2. **View the Output Message:** The program should display: Text has been saved to 'output.txt'.
3. **Check the Created File:** Go to the directory where the program was executed.You should find a file named output.txt. Open this file using a text editor to verify that it contains:

   Why are you willing to become a millionaire when you are still joking with cash? It's pitty!!

Step 4: Verify and troubleshoot (if necessary)

File Not Found: If the file output.txt is not created, ensure that the program has the correct permissions to write to the directory.

File Content Incorrect: If the file content is not what you expected, double-check the text you wrote to the file in the program code.

Summary Steps

1. Write the Program: Include necessary headers, create a file, write the text, and close the file.
2. Compile the Program: Use a C++ compiler like g++ to compile the source code.
3. Run the Program: Execute the compiled program to create the file and write the text to it.
4. Verify the Output: Check the directory to ensure the file output.txt contains the specified text.

Points to Remember

Description of File Handling

File handling is an essential aspect of programming that allows applications to create, read, modify, and delete files for data storage and management. It provides a way to store data permanently, enabling it to be accessed and manipulated beyond the runtime of a program. In programming languages like C++, file handling is made possible using stream classes from the <**fstream**> header file. These classes include:

- **ifstream**: Used for reading data from files.
- **ofstream**: Used for writing data to files.
- **fstream**: Used for both reading and writing.

To perform file operations, a program must open a file in a specific mode. Common file opening modes include:

- **Read (ios::in):** Opens a file for reading.
- **Write (ios::out):** Opens a file for writing, overwriting existing content if the file already exists.
- **Append (ios::app):** Opens a file for appending data to the end without modifying existing content.
- **Binary (ios::binary):** Opens a file in binary mode for processing non-text data.

Proper file handling practices, such as selecting the right file mode and managing file streams, are crucial for maintaining data integrity and ensuring smooth file operations in various applications.

Applying file handling

Applying file handling

Steps involved in applying file handling and solve the task of writing and runing a C++ Program to Save Text" Why are you willing to become a millionaire when you are still joking with cash? It's a pity!!" to Hard Storage are as follows:

- **Write the Program**: Include necessary headers, create a file, write the text, and close the file.
- **Compile the Program:** Use a C++ compiler like g++ to compile the source code.
- **Run the Program:** Execute the compiled program to create the file and write the text to it.
- **Verify the Output:** Check the directory to ensure the file output.txt contains the specified text.

Application of learning 3.2.:

Imagine you're a computer programmer working on a team that makes smart home devices. These devices are like fancy gadgets that can do things like control lights, adjust temperatures, and play music.

Your job is to make sure these devices can save and remember important information. This includes stuff like settings you chose, data from the users, and updates to the device's software. To do this, you'll be using a programming language called C++.

Think of it like building a digital filing system for your devices. You need to make sure it's strong, reliable, and easy to use

Indicative content 3.3: Apply multithreading and concurrency

Duration: 7hrs

Theoretical Activity 3.3.1: description of multithreading and concurrency.

Tasks:

1: In your groups, answer the questions reflecting to multithreading and concurrency:

1. What do you understand by:
    a. Thread
    b. Race condition
    c. Thread pool
    d. Load balancing
2. Discuss the main differences between multithreading and multiprocessing, and when would you choose one over the other?
3. How do you ensure thread safety when multiple threads access shared resources? What techniques or constructs can you use?
4. Can you explain the role of mutexes, condition variables, and semaphores in managing concurrency? How do they differ?
5. What challenges might arise from using multithreading in an application, such as race conditions or deadlocks? How can you mitigate these issues?
6. How does the operating system schedule threads, and what factors influence the performance of a multithreaded application?

2: Write findings on paper/ flipchart

3: Make presentation of the findings

4: Ask questions for more clarifications or provide concerns

5: Read through Key **readings 3.3.1** for additional clarifications

Key readings 3.3.1.

Description of multithreading and concurrency

1. **Definition**

✓ **Thread**

A thread is the smallest unit of execution in a program. A single process can have multiple threads running independently, but they share the same resources such as memory and file handles.

✓ **Thread Life Cycle**

A thread typically goes through several states during its lifetime:

- **New**: A thread is created.
- **Runnable**: The thread is ready to run.
- **Running**: The thread is actively executing.
- **Blocked/Waiting**: The thread is paused, waiting for a condition or resource to become available.
- **Terminated**: The thread has finished its execution.

✓ **Multithreading**

Multithreading refers to the ability of a CPU or a single process to manage the execution of multiple threads at the same time. Each thread can perform a different task, enabling a program to run tasks in parallel or concurrently.

✓ **Concurrency**

Concarency refers to the ability of a system to handle multiple tasks or operations at the same time. It is a core concept computing systems particularly in system programming, multi-threading, and distributed systems.

✓ **Parallelism**

Parallelism is a concept in computer science and systems design where multiple tasks or operations are executed simultaneously, typically across multiple processors or cores. It's a subset of concurrency but with a focus on true simultaneous execution, leading to faster processing, especially in multi-core or distributed systems.

✓ **Context Switching**

Context switching occurs when the operating system switches the CPU's focus from one thread to another. This process saves the state of the current thread and

restores the state of the next thread. Context switching can add overhead and slow down performance.

- ✓ **Race Condition**

A race condition occurs when multiple threads try to access and modify shared resources (like variables or memory) simultaneously, leading to unpredictable results. Without proper synchronization, threads may overwrite each other's changes or read inconsistent data.

- ✓ **Thread Synchronization**

To avoid race conditions and ensure correct results, synchronization is used to control how threads interact with shared resources.

- ✓ **Common synchronization mechanisms include:**

i. **Mutex (Mutual Exclusion)**

Prevents multiple threads from accessing the same resource simultaneously.

ii. Locks

Locks are a key synchronization primitive in multithreading. They ensure that only one thread can access a shared resource at a time. There are different types of locks:

Used to manage access to shared resources, ensuring only one thread can use them at a time.

iii. Semaphores

Similar to mutexes but can allow a specific number of threads to access a resource simultaneously.

iv. Condition Variables

Used to block a thread until a certain condition is met.

v. **Deadlock**

A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource. For example, Thread A locks resource 1 and waits for resource 2, while Thread B locks resource 2 and waits for resource 1. Deadlocks can halt the execution of programs.

vi. Livelock

In a livelock, threads continuously change states in response to each other without making any progress. Unlike a deadlock, where threads are blocked, in a livelock, threads are active but stuck in an infinite loop, unable to complete their tasks.

vii. **Starvation**

Starvation occurs when a thread is perpetually denied access to resources because other threads monopolize them. As a result, the starved thread cannot make progress.

✓ **Thread pooling**

A thread pool is a collection of pre-initialized threads that can be reused to perform tasks. Thread pools improve performance by reducing the overhead of creating and destroying threads repeatedly.

✓ **Atomic Operations**

An atomic operation is an operation that is completed in a single step without interference from other threads. Atomic operations prevent race conditions by ensuring that no other thread can interrupt them.

✓ **Mutex Lock**

A simple binary lock (either locked or unlocked) that ensures mutual exclusion.

Read-Write Lock: Allows multiple readers or one writer to access a resource at a time, improving performance in read-heavy applications.

Spinlock: A lightweight lock where a thread continuously checks whether the lock is available.

✓ **Critical Section**

A critical section is a part of a program where shared resources are accessed or modified. Proper synchronization (using locks, mutexes, etc.) is required to ensure that only one thread enters the critical section at a time to prevent data corruption or inconsistent results.

✓ **Load balancing**

Load balancing is a technique used to distribute incoming network traffic across multiple servers to ensure no single server is overwhelmed, thus optimizing resource use, improving performance, and ensuring high availability. It's a critical concept in large-scale web services and infrastructure.

✓ **Types of Load Balancing**

Hardware Load Balancers: These are physical devices that balance traffic. They offer high performance but can be expensive.

Software Load Balancers: These are applications that distribute traffic across multiple servers. Examples include Nginx, HAProxy, and Apache Traffic Server.

Cloud-Based Load Balancers: Many cloud providers, like AWS (Elastic Load Balancing), Google Cloud (Cloud Load Balancing), and Microsoft Azure (Load Balancer), offer load balancing as a managed service.

2. The Tread class

In C++, the **std::thread class**, introduced in C++11, provides a way to create and manage threads. It is part of the <thread> header and allows concurrent execution of code by running different parts of the program in parallel on separate threads.

a. **Key Features of std::thread Class**

✓ **Creating a Thread**

A thread can be created by passing a callable object (like a function, lambda, or functor) to the std::thread constructor.

✓ **Joining a Thread**

After a thread has been started, you can call the **join()** method to block the current thread until the thread execution finishes.

✓ **Detaching a Thread**

Instead of joining, you can call detach, to allow the thread to run independently. The detached thread will continue execution even if the main thread finishes.

✓ **Thread ID**

The get_id() function returns the unique identifier for a thread.

b. Thread Local Storage

C++ allows you to declare thread-local variables using the thread_local keyword, which makes sure each thread has its own instance of the variable.

✓ **Important Member Functions**

Constructor

std::thread can be initialized with a callable object.

syntax:

std::thread t(function_name);

join(): Waits for the thread to finish its execution.

syntax:

t.join();

detach(): Separates the thread from the thread object, allowing it to run independently.

Syntax:

t.detach();

- **joinable**(): Checks if the thread can be joined (i.e., if it is still running).

```
if (t.joinable()) {
   t.join();
}
```

- **get_id():** Returns the thread's unique ID.

syntax:

std::thread::id id = t.get_id();

Notes:

If a thread is not joined or detached, it results in undefined behavior (typically, a program crash).
Threads should be properly synchronized when accessing shared data, using mechanisms like mutexes (std::mutex) to avoid race conditions.

c. Creating and managing a trhead

The std::thread class in C++ offers a convenient way to create and manage threads of execution within a program.

Here's a breakdown of the steps involved:

✓ **Include the Necessary Header**

Include the thread header to access the **std::thread class**

#include <thread>

✓ **Define a Callable Object**

A callable object is a function, lambda, or functor that will be executed by the thread.
Example: Function

```
void myFunction(int x) {
   std::cout << "Thread function: x = " << x << std::endl;
}
```

✓ **Create a Thread Object**

Construct a std::thread object, passing the callable object and any necessary arguments

Example

```
std::thread myThread(myFunction, 42);
```

This creates a new thread and starts executing the myFunction with the argument 42.

✓ **Join or Detach**

Decide whether to wait for the thread to finish or let it run independently:

Joining

Use **join()** to wait for the thread to complete before continuing with the main thread:

```
myThread.join();
```

Detaching:

Use **detach()** to let the thread run independently without the main thread waiting for it:

```
myThread.detach();
```

Example:

```
#include <iostream>
#include <thread>
void myFunction(int x) {
   std::cout << "Thread function: x = " << x << std::endl;
}
int main() {
   std::thread t1(myFunction, 42);
   std::thread t2(myFunction, 100);
   t1.join();
   t2.join();
   std::cout << "Main thread finished." << std::endl;
   return 0;
}
```

Explanation:

Include Necessary Header:

The **iostream** header is included for input/output operations, and the **thread header** is included for thread-related functionality.

Define a Callable Object

The myFunction function is defined, which will be executed by the threads. It takes an integer x as input and prints a message to the console.

Create Threads

Two std::thread objects, t1 and t2, are created. Each thread is passed the myFunction and an integer argument (42 and 100, respectively). This starts the threads executing the myFunction concurrently.

Join or Detach

Join: If you want the main thread to wait for the child threads to finish before proceeding, you can use the join() method on each thread. This ensures that the main thread doesn't continue until both child threads have completed their tasks.

Detach: If you want the child threads to run independently and continue executing even after the main thread has finished, you can use the detach() method. This means the child threads will run in the background and may not be joined later.

3. Thread Synchronization primitives

Thread synchronization primitives in C++ are essential for ensuring that multiple threads can safely access shared resources or coordinate their activities without introducing race conditions, deadlocks, or other issues.

The following are the key synchronization primitives in C++:

✓ **Std::mutex**

The std::mutex synchronization primitive in C++ is a basic tool used to ensure mutual exclusion in a multi-threaded environment. It allows threads to lock critical sections of code so that only one thread at a time can access a shared resource, ensuring thread safety and preventing race conditions.

a. **Key Features of std::mutex**

Mutual Exclusion: Ensures that only one thread can hold the lock at a time.

Non-Recursive: A thread that already holds the lock on a std::mutex cannot lock it again without first unlocking it. To do so, causes undefined behavior.

Block on Lock: If a thread tries to lock a std::mutex that is already locked by another thread, the calling thread will block (i.e., wait) until the mutex becomes available.

b. **Basic Operations on std::mutex**

lock()

It acquires the mutex, blocking the calling thread if the mutex is already locked by another thread.

unlock()

It releases the mutex, allowing other threads to lock it.

try_lock()

It attempts to lock the mutex without blocking. Returns true if the mutex was successfully locked; otherwise, returns false.

c. Usage Example of std::mutex

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;  // Create a mutex to protect shared resource
void print_message(const std::string &message, int id) {
   mtx.lock();  // Lock the mutex to ensure only one thread can execute the
critical section at a time
   std::cout << "Thread " << id << ": " << message << std::endl;
   mtx.unlock();  // Unlock the mutex after the critical section is done
}
int main() {
   std::thread t1(print_message, "Hello from thread 1", 1);
   std::thread t2(print_message, "Hello from thread 2", 2);
   t1.join();
   t2.join();
   return 0;
}
```

Explanation of the Example

Critical Section: The code inside the **mtx.lock()** and mtx.unlock() block is called a critical section. In this case, it's the std::cout operation which accesses the shared resource (standard output).
Thread Safety: Only one thread can execute the critical section at a time. The other thread will wait until the mutex is unlocked.
Mutex Locking: mtx.lock() locks the mutex, and mtx.unlock() releases it.

✓ **The Std::lock_guard synchronization primitive**

The std::lock_guard synchronization primitive in C++ is a convenient RAII (Resource Acquisition Is Initialization) wrapper for managing the locking and unlocking of mutexes. It ensures that a mutex is automatically locked when the **std::lock_guard** object is created and automatically unlocked when the **std::lock_guard** object goes

out of scope, making it easier to avoid mistakes such as **forgetting to unlock a mutex**, especially in the presence of exceptions or early returns.

a. Key Features of std::lock_guard

RAII-based Mutex Management: It locks a mutex when the std::lock_guard object is created and automatically unlocks it when the object is destroyed (when it goes out of scope).

No Manual Unlocking Needed: This eliminates the need for manual unlocking of the mutex, ensuring that the mutex is properly released even if an exception occurs or a function returns early.

Simple and efficient: std::lock_guard is a lightweight, non-copyable object that offers efficient and straightforward locking functionality.

Usage Example of std::lock_guard

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;  // Mutex to protect shared resource
void print_message(const std::string& message, int id) {
   std::lock_guard<std::mutex> lock(mtx);  // Automatically locks the mutex
   std::cout << "Thread " << id << ": " << message << std::endl;
   // Mutex is automatically unlocked when lock goes out of scope
}
int main() {
   std::thread t1(print_message, "Hello from thread 1", 1);
   std::thread t2(print_message, "Hello from thread 2", 2);
   t1.join();
   t2.join();
   return 0;
}
```

Explanation

Locking: When std::lock_guard<std::mutex> lock(mtx) is called, the mutex mtx is automatically locked by the lock_guard object.

Scope-Based Unlocking: When the lock_guard goes out of scope (in this case, when the function print_message ends), the mutex is automatically unlocked. You don't need to manually call mtx.unlock(), which helps prevent errors such as forgetting to unlock the mutex.

b. **Key Advantages of std::lock_guard**

Exception Safety: If an exception is thrown, std::lock_guard ensures the mutex is unlocked when the object is destroyed. This prevents deadlocks that could occur if the mutex isn't manually unlocked due to an unexpected exception.

Example

```
void safe_function() {
   std::lock_guard<std::mutex> lock(mtx);  // Mutex is locked
   // Some code that might throw an exception
   if (some_condition) {
      throw std::runtime_error("Error!");
   }
   // Mutex is automatically unlocked even if an exception occurs
}
```

Simplified Code: By automatically managing the lock and unlock process, std::lock_guard simplifies the code, reducing the risk of mistakes and making the code easier to read.

Non-copyable: std::lock_guard objects cannot be copied, which ensures that only one instance is responsible for unlocking the mutex. This prevents accidental unlocking by multiple threads.

Comparison with Manual Locking

Manual Locking:

```
std::mutex mtx;
void manual_locking() {
   mtx.lock();  // Locking the mutex
   // Critical section
   mtx.unlock();  // Manually unlocking the mutex
}
```

In the above case, if an exception or early return occurs between lock() and unlock(), the mutex may not be unlocked properly, leading to deadlocks.

c. **Using std::lock_guard:**

```
std::mutex mtx;
void safe_locking() {
   std::lock_guard<std::mutex> lock(mtx);  // Mutex automatically locked and
unlocked
   // Critical section
```

```
    // Mutex automatically unlocked when going out of scope
}
```

This is safer because the mutex is automatically unlocked when the lock_guard object goes out of scope, avoiding the potential for errors like forgetting to unlock the mutex.

✓ **The std::unique_lock synchronization primitive**

The std::unique_lock synchronization primitive in C++ is a more flexible and powerful mutex wrapper than std::lock_guard. Like std::lock_guard, it is an RAII (Resource Acquisition Is Initialization) wrapper that automatically manages the locking and unlocking of a mutex. However, std::unique_lock provides additional control over locking, unlocking, and ownership of the mutex, making it suitable for more complex synchronization needs.

a. Key functions and features of std::unique_lock

Deferred Locking: std::unique_lock provides an option to defer locking until you explicitly call lock().

```
std::unique_lock<std::mutex> lock(mtx, std::defer_lock);  // Mutex is not locked yet
// Perform other operations
lock.lock();  // Lock the mutex later
```

This feature allows you to initialize the unique_lock object without locking the mutex and lock it later when needed.

Manual Unlocking: You can explicitly unlock the mutex during the lifetime of the unique_lock object by calling unlock(). This is useful when the lock is only needed for part of the function.

```
void manual_unlock_example() {
    std::unique_lock<std::mutex> lock(mtx);  // Lock the mutex
    // Critical section
    lock.unlock();  // Unlock the mutex early
    // Perform non-critical work without holding the lock
}
```

Try Locking: std::unique_lock supports try_lock() to attempt to acquire the lock without blocking. If the lock is already held by another thread, try_lock() returns false.

```
if (lock.try_lock()) {
    // Successfully locked
} else {
```

```
    // Failed to lock, another thread holds the lock
}
```

Timed Locking: std::unique_lock supports timed locking, allowing you to specify a time duration to wait for the lock.

try_lock_for(): Attempts to acquire the lock, waiting for a specified duration.

try_lock_until(): Attempts to acquire the lock until a specific time point.

```
if (lock.try_lock_for(std::chrono::milliseconds(100))) {
    // Successfully locked within 100 milliseconds
} else {
    // Timed out, failed to acquire the lock
}
```

Ownership Transfer: std::unique_lock is movable, meaning ownership of the mutex lock can be transferred between different unique_lock objects. This is useful when you need to pass ownership between functions or threads.

```
std::unique_lock<std::mutex> lock1(mtx);
```

```
std::unique_lock<std::mutex> lock2 = std::move(lock1);  // Ownership transferred to lock2
```

Once ownership is transferred, the original unique_lock (lock1) no longer holds the lock, and the new unique_lock (lock2) manages the mutex.

b. Syntax and Usage

Basic Usage of std::unique_lock

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;
void print_message(const std::string& message, int id) {
    std::unique_lock<std::mutex> lock(mtx);  // Mutex is locked here
    std::cout << "Thread " << id << ": " << message << std::endl;
    // Mutex is unlocked automatically when 'lock' goes out of scope
}
int main() {
    std::thread t1(print_message, "Hello from thread 1", 1);
    std::thread t2(print_message, "Hello from thread 2", 2);
    t1.join();
    t2.join();
    return 0;
}
```

In this example, std::unique_lock works similarly to std::lock_guard by locking the mutex when the unique_lock object is created and automatically unlocking it when the object goes out of scope.

c. Example of std::unique_lock with Deferred and Manual Locking

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;
void critical_section(int id) {
   std::unique_lock<std::mutex> lock(mtx, std::defer_lock);  // Create the lock
without locking the mutex
   std::cout << "Thread " << id << " performing non-critical work...\n";
   lock.lock();  // Lock the mutex when entering the critical section
   std::cout << "Thread " << id << " in critical section\n";
   lock.unlock();  // Unlock the mutex after leaving the critical section
   std::cout << "Thread " << id << " performing other work...\n";
}
int main() {
   std::thread t1(critical_section, 1);
   std::thread t2(critical_section, 2);
   t1.join();
   t2.join();
   return 0;
}
```

d. Comparison: std::unique_lock vs std::lock_guard

Feature	Std:: lock_guard	Std::unique_lock
Locking	Always locks up construction	Can defer locking, lock later as needed

Unlocking	Automatic when going out of the scope	Manual unlock possible
Try-locking	Not supported	Supports try_lcok()
Timed locking	Not suppored	Supports try_lock_for(), try_lock_until()
Ownership transfer	Not movable	movable
Complexity	Simple	More flexible but slightly more complex

4. Apply Deadlocks and Race Conditions

Deadlocks and race conditions are two common problems that arise in multi-threaded programming when threads compete for shared resources. Both issues can cause serious bugs and unpredictable behavior in a program.

a. **Deadlock**

A deadlock occurs when two or more threads are unable to proceed because each thread is waiting for a resource that the other thread holds, resulting in a circular dependency. This situation causes all involved threads to be permanently blocked, unable to proceed with their execution.

✓ **Key Conditions for Deadlock**

For a deadlock to occur, four conditions (also known as the Coffman conditions) must be present:

Mutual Exclusion: At least one resource must be held in a non-sharable mode (i.e., only one thread can use the resource at a time).

Hold and Wait: A thread holding a resource is waiting for additional resources held by other threads.

No Preemption: A resource cannot be forcibly taken away from a thread; it can only be released voluntarily.

Circular Wait: A circular chain of two or more threads exists, where each thread is waiting for a resource held by the next thread in the chain.

✓ **Example of Deadlock**

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx1, mtx2;
void thread1() {
   std::lock_guard<std::mutex> lock1(mtx1);  // Thread 1 locks mtx1
   std::this_thread::sleep_for(std::chrono::milliseconds(50));  // Simulate work
   std::lock_guard<std::mutex> lock2(mtx2);  // Thread 1 tries to lock mtx2 (but
it's held by thread2)
}
void thread2() {
   std::lock_guard<std::mutex> lock1(mtx2);  // Thread 2 locks mtx2
   std::this_thread::sleep_for(std::chrono::milliseconds(50));  // Simulate work
   std::lock_guard<std::mutex> lock2(mtx1);  // Thread 2 tries to lock mtx1 (but
it's held by thread1)
}
int main() {
   std::thread t1(thread1);
   std::thread t2(thread2);
   t1.join();
   t2.join();
   return 0;
}
```

In this example:

Thread 1 locks mtx1 and then tries to lock mtx2, which is held by Thread 2. Thread 2 locks mtx2 and then tries to lock mtx1, which is held by Thread 1. Both threads are now waiting on each other to release their locks, causing a deadlock.

✓ **Ways to Avoid Deadlock**

Locking Order: Ensure that all threads acquire locks in the same order. This prevents circular wait conditions.

```
std::lock(mtx1, mtx2);  // Acquire both mutexes at the same time in a deadlock-
free manner
```

Try Locking: Use try_lock() to avoid blocking if a mutex is already locked, allowing a thread to back off if the lock cannot be acquired.

```
if (mtx1.try_lock()) {
   // Do work
}
```

Timeout Mechanism: Use timed locking methods like std::unique_lock::try_lock_for() to give up waiting after a certain time.

Avoid Nested Locks: Limit the scope of mutexes and avoid holding multiple locks simultaneously whenever possible.

- ✓ **Race Conditions**

A race condition occurs when two or more threads access shared data concurrently, and at least one thread modifies the data. The final result of the program depends on the timing or order in which the threads execute, leading to non-deterministic behavior. This is problematic because the outcome of the program becomes unpredictable and inconsistent.

Example of a Race Condition

```
#include <iostream>
#include <thread>
int counter = 0;  // Shared resource
void increment_counter() {
   for (int i = 0; i < 100000; ++i) {
      ++counter;  // Race condition: multiple threads can modify `counter`
concurrently
   }
}

int main() {
   std::thread t1(increment_counter);
   std::thread t2(increment_counter);
   t1.join();
   t2.join();
   std::cout << "Final counter value: " << counter << std::endl;  // The value will be
unpredictable
   return 0;
}
```

In this example:

Both threads increment the shared variable counter concurrently. Since no synchronization is used, the threads can read, increment, and write back to the variable in an unpredictable order, leading to a race condition. The final

value of counter will likely be incorrect because increments from one thread may overwrite increments from the other.

- ✓ **Ways to Avoid Race Conditions**

Mutexes: Use a mutex to protect shared resources.

```
std::mutex mtx;
void increment_counter() {
   for (int i = 0; i < 100000; ++i) {
      std::lock_guard<std::mutex> lock(mtx);  // Protect the counter
      ++counter;
   }
}
```

Atomic Variables: Use atomic variables for simple operations like incrementing counters. Atomic variables guarantee that operations on them are performed atomically (i.e., without interruption).

```
std::atomic<int> counter(0);
void increment_counter() {
   for (int i = 0; i < 100000; ++i) {
      ++counter;  // Automatically safe from race conditions
   }
}
```

Avoid Shared Data: Where possible, avoid sharing data between threads. Instead, pass data to threads by value or use thread-local storage.

Summary of Deadlock vs. Race Condition

Features	Deadlock	Race condition
Definition	occurs when two or more threads are unable to proceed because each thread is waiting for a resource that the other thread holds, resulting in a circular dependency	Occurs when multiple threads access shared data concurrently and at least one modifies the data leading to unpredictable results.
Symptoms	Threads are blocked indefinitely	Program behavior is non-deterministics;

	waiting for each other	results vary with each other
Causes	Circular dependency between threads waiting for lock	Lack of proper synchronization when accessing shared data
Solution	Acquire locks in the consistent order. Use try_lock() and timed lock	Use mutexes, atomic variables or avoid shared data
Complexity	Harder to detect because the program can stall completely	Easy to detect because of incorrect and inconsistent results

Both deadlocks and race conditions are critical issues in concurrent programming, and understanding how to prevent them is essential for writing robust, thread-safe code.

Theoretical Activity 3.3.2: Description of thread pool and parallel algorithm

Tasks:

1: In your groups, answer the questions reflecting to multithreading and concurrency:

1. What do you understand by:
    a) Thread pool
    b) Load balancing
2. Discuss the application areas of thread pool.
3. Identify parallel algorithms

2: Write findings on paper/ flipchart

3: Make presentation of the findings

4: Ask questions for more clarifications or provide concerns

5: Read through Key **readings 3.3.2** for additional clarifications

Key readings 3.3.2: Description of thread pool and parallel algorithm

. Description of thread pool

1. **Introduction**

   A thread pool is a collection of pre-instantiated, reusable threads that can be used to execute multiple tasks concurrently.

2. **Working principle**

   Instead of creating a new thread for each task, tasks are submitted to the pool, which assigns them to available threads. Once a thread completes a task, it can be reused to handle another task, improving performance by reducing the overhead of thread creation and destruction.

3. **Implementation of thread pool**

   In C++, a thread pool can be implemented using the Standard Library with **std::thread** and synchronization primitives like **std::condition_variable**, **std::mutex**, and **std::queue**. Here's a basic implementation of a thread pool in C++ using the C++11 threading features:

4. **Usage of thread pool**

   The common use cases of thread pool are:

   Web Servers: Handling multiple concurrent HTTP requests.

   Database Connections: Managing database connections efficiently.

   Background Tasks: Executing tasks in the background without blocking the main application thread.

   I/O-Bound Operations: Handling tasks that involve input/output operations (e.g., file reading/writing, network communication).

   CPU-Bound Operations: Executing computationally intensive tasks concurrent

Example of a Simple Thread Pool in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <future>
class ThreadPool {
public:
```

```
    ThreadPool(size_t threads);
    ~ThreadPool();
    // Add new work to the pool
    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args) -> std::future<typename
std::result_of<F(Args...)>::type>;
private:
    // Worker threads
    std::vector<std::thread> workers;
      // Task queue
    std::queue<std::function<void()>> tasks;

    // Synchronization
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
    // Worker function
    void workerThread();
};


// Constructor: create worker threads
ThreadPool::ThreadPool(size_t threads) : stop(false) {
    for(size_t i = 0; i < threads; ++i) {
        workers.emplace_back([this] { this->workerThread(); });
    }
}
// Destructor: Join all worker threads
ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }
    condition.notify_all();  // Notify all threads to finish
    for(std::thread &worker: workers) {
        worker.join();  // Join the threads
    }
}
// Add new work to the pool
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args) -> std::future<typename
```

```
std::result_of<F(Args...)>::type> {
    using return_type = typename std::result_of<F(Args...)>::type;
    // Create a packaged task to hold the function
    auto task = std::make_shared<std::packaged_task<return_type()>>(
        std::bind(std::forward<F>(f), std::forward<Args>(args)...)
    );
    // Create a future to get the result of the task
    std::future<return_type> res = task->get_future();
    // Add the task to the queue
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        if(stop) throw std::runtime_error("enqueue on stopped ThreadPool");
        tasks.emplace([task]() { (*task)(); });
    }
    condition.notify_one();  // Notify one worker thread
    return res;
}
// Worker function that processes tasks
void ThreadPool::workerThread() {
    while(true) {
        std::function<void()> task;
        {
            std::unique_lock<std::mutex> lock(this->queue_mutex);
            this->condition.wait(lock, [this]{ return this->stop || !this->tasks.empty();
});
            if(this->stop && this->tasks.empty())
                return;
            task = std::move(this->tasks.front());
            this->tasks.pop();
        }
        task();
    }
}
// Test the ThreadPool
int main() {
    ThreadPool pool(4);
    // Enqueue some work
    auto result1 = pool.enqueue([](int x, int y) { return x + y; }, 3, 4);
    auto result2 = pool.enqueue([](int x) { return x * 2; }, 10);
    std::cout << "Result of addition: " << result1.get() << std::endl;  // Outputs 7
```

```
    std::cout << "Result of multiplication: " << result2.get() << std::endl;  /*
Outputs 20*/
    return 0;
}
```

Explanations

Thread Pool Constructor: Initializes a fixed number of worker threads that continuously wait for tasks to be assigned.

Enqueue Method: Adds a task to the pool and returns a std::future so that the caller can retrieve the result of the task once it's complete.

Worker Thread: Each worker thread waits for tasks to be added to the queue. When a task is available, the thread executes it.

Synchronization: The **std::condition_variable** ensures that the worker threads wake up only when there is work to do, and std::mutex protects the task queue from concurrent access.

Task Execution: Tasks are enqueued in the form of std::function**<void()>** and executed by worker threads.

I. Description of parallel algorithm

1. Definition

A parallel algorithm in C++ refers to an algorithm that is designed to execute multiple operations or tasks concurrently, leveraging multiple processing cores or threads to improve performance.

2. Working principle

Unlike sequential algorithms, where operations are executed one after another, parallel algorithms break down a task into smaller, independent sub-tasks that can be processed simultaneously on different CPU cores.

3. Characteristics of parallel algorithms

- ✓ **Concurrency**: Multiple tasks or operations are performed at the same time.
- ✓ **Independence**: Tasks should ideally be independent of one another to avoid conflicts such as race conditions or deadlocks.
- ✓ **Task distribution:** The workload is distributed across multiple threads or cores.
- ✓ **Synchronization:** Some algorithms may require mechanisms to synchronize shared data between threads to ensure correctness.

4. Application of parallel algorithm

Common use cases for parallel algorithms:

- ✓ **Sorting Large Datasets:**

  Parallel sorting algorithms like std::sort with std::execution::par can drastically reduce the time needed to sort large datasets, which is particularly useful in data analytics and database systems

- ✓ **Reduction**:

  Summing, multiplying, or finding the maximum/minimum in a large array (e.g., std::reduce).

- ✓ **Numerical Simulations:**

  Parallel algorithms are used to simulate physical phenomena such as fluid dynamics, weather modeling, and seismic analysis. By distributing the computations across multiple cores, large-scale simulations can be run more efficiently.

- ✓ **Data Preprocessing:**

  When handling large datasets for training machine learning models, operations like normalization, transformation, and feature extraction can be parallelized to improve performance.

- ✓ **Rendering**:

  In real-time graphics rendering, parallel algorithms can be used to render different parts of a scene concurrently. Techniques like ray tracing and rasterization benefit from parallelization, improving frame rates and graphical performance.

- ✓ **Physics Simulation:**

  Games often require simulations of real-world physics for object interaction, collisions, and motion. These physics computations can be distributed across multiple threads to enhance game performance.

- ✓ **Transformations**:

  Applying functions to collections of data (e.g., std::transform).

- ✓ **Parallel Image Filters:**

Image processing tasks such as applying convolution filters, edge detection, or color transformations can be parallelized to enhance performance when processing high-resolution images.

- ✓ **Feature Detection:**

  Algorithms for detecting objects, features (such as corners or edges), and performing pattern recognition in computer vision can utilize parallel algorithms to speed up the detection process.

- ✓ **Parallel Query Execution:**

  In large-scale database systems, parallel algorithms are used to execute SQL queries in parallel, especially for complex queries involving large datasets. Operations like joins, aggregations, and filtering can be distributed across multiple threads to improve query response times.

5. **Available Parallel Algorithms in C++**

   Some of the commonly used algorithms that can be parallelized with **std::execution::par** are

   std::for_each: Apply a function to each element of a range.

   std::transform: Transform a range by applying a function to its elements.

   std::sort: Sort a range of elements.

   std::copy: Copy elements from one range to another.

   std::reduce: Perform a reduction operation (e.g., sum, product) over a range.

   Examples

- ✓ **Parallel Sort Using Parallel STL (C++17**)

  Here's an example of sorting a large dataset in parallel using C++17's Parallel STL

```
#include <iostream>
#include <vector>
#include <algorithm>   // For std::sort
#include <execution>   // For parallel execution policies
int main() {
   std::vector<int> data(10'000'000);
   std::generate(data.begin(), data.end(), std::rand);  // Fill vector with random
values
   // Parallel sort using std::execution::par
   std::sort(std::execution::par, data.begin(), data.end());
```

```
    std::cout << "Data sorted in parallel." << std::endl;

    return 0;
}
```

This simple example uses the std::execution::par execution policy to sort a large vector of integers in parallel, reducing the time required for sorting compared to sequential execution.

✓ **Parallel Algorithm for Summing a Large Array**

Here's a simple example of a parallel algorithm using C++17's Parallel STL:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>   // For parallel execution policies
int main() {
    // Generate a large vector of integers
    std::vector<int> data(1'000'000, 1);  // 1 million elements, each initialized to 1
    // Sum the array using a parallel execution policy
    long long sum = std::reduce(std::execution::par, data.begin(), data.end(), 0LL);
    std::cout << "Parallel sum: " << sum << std::endl;  // Output: 1 million
    return 0;
}
```

6. The std::for_each algorithm

The std::for_each algorithm in C++ is a part of the Standard Library, defined in the <algorithm> header that applies a given function to each element in a range, making it easy to perform operations on containers like vectors or arrays.

1. Features of std::for_each algorithm

Here are the key features of the std::for_each algorithm in C++:

Function Application: Applies a specified callable (function, lambda, functor) to each element in a specified range.

Iterator-Based: it takes two iterators: one pointing to the beginning of the range and the other pointing to one past the last element. This allows it to work with any standard container that supports iterators.

Flexibility: it can accept various types of callable objects, including:

Functions

Lambda expressions

Functors (objects with operator())

Modification Capability: If a non-const reference is used in the callable, std::for_each can modify the elements in place.

Return Value: it returns the first iterator passed as the argument, allowing for potential chaining of operations.

Complexity: The time complexity is O(n), where n is the number of elements in the range, as it processes each element once.

Standard Library Compatibility: Works with any container that provides compatible iterators, such as vectors, lists, and arrays.

Const Correctness: Can operate on both const and non-const ranges, making it versatile for different contexts.

Enhanced Readability: It promotes a functional programming style, for making code easier to read and maintain compared to traditional loops.

Syntax:

2. Application of std::for_each algorithm

The std::for_each algorithm in C++ is versatile and can be applied in various scenarios. Here are some common applications:

✓ **Transforming Data**

You can use std::for_each to modify each element in a container.

Example: Doubling Elements

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

std::for_each(numbers.begin(), numbers.end(), [](int& n) {

 n *= 2;  // Double each element

});
```

✓ **Performing Side Effects**

std::for_each is often used to execute functions that have side effects, such as printing or logging.

Example: Printing Elements

```
std::vector<std::string> names = {"Alice", "Bob", "Charlie"};

std::for_each(names.begin(), names.end(), [](const std::string& name) {

   std::cout << name << std::endl;  // Print each name

});
```

- ✓ **Accumulating Results**

While std::for_each is not specifically designed for accumulation (unlike std::accumulate), it can still be used to update a shared state.

Example: Summing Values

```
int sum = 0;

std::vector<int> values = {1, 2, 3, 4, 5};

std::for_each(values.begin(), values.end(), [&sum](int n) {

   sum += n;  // Accumulate sum

});

std::cout << "Total: " << sum << std::endl;
```

- ✓ **Filtering with Side Effects**

You can apply a function to filter elements based on a condition while performing some operation on those elements.

Example: Counting Even Numbers

```
int even_count = 0;

std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

std::for_each(numbers.begin(), numbers.end(), [&even_count](int n) {

   if (n % 2 == 0) {

      even_count++;  // Count even numbers
```

```
    }
});
std::cout << "Even Count: " << even_count << std::endl;
```

- ✓ **Complex Operations**

  You can perform complex operations involving multiple data structures by iterating through them simultaneously.

  Example: Pairing Elements

```
std::vector<int> a = {1, 2, 3};
std::vector<int> b = {4, 5, 6};
std::vector<int> results(3);
std::for_each(a.begin(), a.end(), [&](int n) {
    size_t index = &n - &a[0];  // Get index
    results[index] = n + b[index];  // Pair and sum elements
});
```

- ✓ **Applying Custom Functions**

  You can use std::for_each to apply any custom-defined function to elements.

  Example: Using a Function

```
struct Square {
    void operator()(int& n) const {
        n *= n;  // Square the number
    }
};

std::vector<int> numbers = {1, 2, 3, 4, 5};
std::for_each(numbers.begin(), numbers.end(), Square())
```

3. **The std::transform algorithm**

The std::transform is a powerful algorithm in the C++ Standard Library that applies a function to a range of elements and stores the result in a new range. It is often used to perform element-wise transformations on containers such as arrays, vectors, or other iterables.

- **Features of std::transform**

Here are its key features:

✓ **Function Signature**

The basic signature of std::transform is as follows:

template< class InputIt, class OutputIt, class UnaryOperation > OutputIt transform(InputIt first, InputIt last, OutputIt d_first, UnaryOperation op);

There's also an overloaded version that takes two input ranges:

template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation > OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2, OutputIt d_first, BinaryOperation op);

Where parameters are:

InputIt: Iterators defining the range of input elements (first, last).

OutputIt: An iterator where the output will be stored (d_first).

UnaryOperation: A function or functor that takes one argument (for the unary version) and produces a result.

BinaryOperation: A function or functor that takes two arguments (for the binary version) and produces a result.

✓ **Usage with STL Iterators**

std::transform works seamlessly with standard library iterators, including pointers, iterators from std::vector, std::list, etc.

In-place Transformation

When the output range overlaps with the input range, std::transform can be used for in-place transformations. Care should be taken to avoid undefined behavior.

Performance

Optimized for performance, as it can be implemented to work with iterators efficiently, reducing the overhead of copying elements.

Custom Functions

You can easily define your own operations using lambda expressions or function objects to customize the behavior of the transformation.

Example Usage

Here's a simple example of using std::transform with a unary operation:

```
#include <algorithm>
#include <vector>
#include <iostream>
int main() {
   std::vector<int> vec = {1, 2, 3, 4};
   std::vector<int> result(vec.size());
   std::transform(vec.begin(), vec.end(), result.begin(), [](int x) { return x * 2; });
   for (int val : result) {
      std::cout << val << " "; // Output: 2 4 6 8
   }
}
```

The use of **std::transform** algorithm is to square each element in the numbers vector and store the results in the squares vector.

```
#include <algorithm>
#include <vector>
int main() {
   std::vector<int> numbers = {1, 2, 3, 4, 5};
   std::vector<int> squares(numbers.size());
   // Square each number and store the result in squares
   std::transform(numbers.begin(), numbers.end(), squares.begin(), [](int x) {
return x * x; });
   // Print the squared numbers
   for (int square : squares) {
      std::cout << square << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

- **Application of the std::transform algorithm**

  The std::transform algorithm in C++ is widely applicable in various scenarios where data transformation or processing is needed. Here are several common applications:

✓ **Data Transformation**

Scaling Values: Adjust values in a dataset by applying a scaling factor or any mathematical operation.

Example

```
std::vector<double> original = {10.0, 20.0, 30.0};
std::vector<double> scaled(original.size());
std::transform(original.begin(), original.end(), scaled.begin(), [](double x) { return x * 1.5; });
```

✓ **Element-wise Operations**

It may be used for combining two containers: use the binary version to combine elements from two containers, such as adding corresponding elements of two vectors.

Example

```
std::vector<int> a = {1, 2, 3};
std::vector<int> b = {4, 5, 6};
std::vector<int> result(a.size());
std::transform(a.begin(), a.end(), b.begin(), result.begin(), std::plus<int>());
```

✓ **String Manipulation**

It can be used for transforming characters: Convert a string to uppercase or lowercase using a character transformation.

Example

```
std::string str = "hello";
std::transform(str.begin(), str.end(), str.begin(), ::toupper);
```

✓ **Filtering and Modifying Data**

Conditionally Transforming Elements: Modify elements based on certain conditions, like squaring only even numbers.

Example

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> squares(numbers.size());
std::transform(numbers.begin(), numbers.end(), squares.begin(), [](int x) { return
(x % 2 == 0) ? x * x : x; });
```

- ✓ **Data Preparation for Algorithms**

  Preprocessing Data: Prepare data before feeding it into algorithms by transforming raw inputs (e.g., normalizing values).

  Example

```
std::vector<double> rawData = {1.0, 2.0, 3.0};
std::vector<double> normalizedData(rawData.size());
double maxVal = *std::max_element(rawData.begin(), rawData.end());
std::transform(rawData.begin(), rawData.end(), normalizedData.begin(),
[maxVal](double x) { return x / maxVal; });
```

- ✓ **Complex Data Structures**

  Transforming Structures: Use std::transform to manipulate complex data structures like std::pair or user-defined types.

  Example

```
struct Point {
   int x, y;
};
std::vector<Point> points = {{1, 2}, {3, 4}, {5, 6}};
std::vector<double> distances(points.size());
std::transform(points.begin(), points.end(), distances.begin(), [](Point p) { return
std::sqrt(p.x * p.x + p.y * p.y); });
```

- ✓ **Creating Derived Data Structures**

  Generating New Data from Existing Data: Create a new collection based on a transformation of an existing one, such as generating a list of lengths from a list of strings.

  Example

```
std::vector<std::string> words = {"hello", "world", "C++"};
std::vector<size_t> lengths(words.size());
std::transform(words.begin(), words.end(), lengths.begin(), [](const std::string&
word) { return word.size(); });
```

- ✓ **In-place Transformation**

Very useful in modifying a Container Directly: Use std::transform for in-place transformations when the output is intended to overwrite the input.

Example

```
std::vector<int> nums = {1, 2, 3, 4};
std::transform(nums.begin(), nums.end(), nums.begin(), [](int x) { return x * 2; });
```

4. The std::reduce algorithm

The **std::reduce** in C++ is used to perform a reduction operation across a range of elements, simplifying tasks like summing, multiplying, or finding the maximum value. It takes iterators that define the range and an optional binary operation, which is applied to combine the elements.

- **Features of std::reduce algorithm**

The **std::reduce** in C++ offers several key features that make it a powerful tool for aggregating data:

✓ **Range-Based Operation**

It takes a pair of iterators defining the range of elements to be reduced, allowing flexibility in choosing the input data.

✓ **Custom binary operation**

It is possible to specify a binary operation (e.g., addition, multiplication) to combine elements, enabling a wide variety of aggregation tasks.

✓ **Initial value**

It allows an optional initial value for the reduction. If not provided, the first element of the range is used as the initial value, and reduction starts from the second element.

✓ **Parallel execution**

Supports parallel execution via the <execution> header (using std::execution::par), which can significantly improve performance for large datasets.

✓ **Support for associative operations**

Works best with associative operations, ensuring correctness especially when executed in parallel.

✓ **Type Deduction**

The return type is automatically deduced based on the type of the initial value and the operation, making it convenient to use without explicit type specifications.

- ✓ **Efficiency**

  It is optimized for performance, making it suitable for high-performance computing tasks, such as numerical computations and data processing.

- ✓ **No Side Effects**

  The algorithm assumes that the provided operation is free of side effects, ensuring predictable results.

- **Basic Usage of std::reduce algorithm**

  Include Required Headers

  To use std::reduce, include the <numeric> header. If you're using parallel execution, also include <execution>.

  Define the Range

  Specify the range of elements you want to reduce using iterators.

  Specify the Initial Value

  You can provide an initial value for the reduction.

  Custom Binary Operation

  Optionally, define a custom binary operation.

  Example 1: Summing Elements

  This is a simple example that sums the elements of a vector:

```
#include <iostream>
#include <vector>
#include <numeric>
int main() {
   std::vector<int> numbers = {1, 2, 3, 4, 5};
      // Using std::reduce to sum the elements
   int sum = std::reduce(numbers.begin(), numbers.end(), 0);
      std::cout << "Sum: " << sum << std::endl; // Output: Sum: 15
```

```
    return 0;
}
```

Example 2: Custom Binary Operation

You can define your own binary operation. For example, to find the product of elements:

```
#include <iostream>
#include <vector>
#include <numeric>
int multiply(int a, int b) {
    return a * b;
}
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Using std::reduce with a custom multiplication function
    int product = std::reduce(numbers.begin(), numbers.end(), 1, multiply);
    std::cout << "Product: " << product << std::endl; // Output: Product: 120
    return 0;
}
```

Example 3: Parallel Execution

Using parallel execution can speed up the reduction process for large datasets. Here's how to do it:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
```

```
    // Using std::reduce with parallel execution
    int sum = std::reduce(std::execution::par, numbers.begin(), numbers.end(), 0);
        std::cout << "Sum (parallel): " << sum << std::endl; // Output: Sum (parallel): 15
    return 0;
}
```

Example 4: Using Complex Types

You can also use std::reduce with more complex types, such as structures. For instance, if you want to aggregate a vector of structs:

```
#include <iostream>
#include <vector>
#include <numeric>
struct Point {
    int x, y;
};

Point operator+(const Point& a, const Point& b) {
    return {a.x + b.x, a.y + b.y};
}
int main() {
    std::vector<Point> points = {{1, 2}, {3, 4}, {5, 6}};
    Point total = std::reduce(points.begin(), points.end(), Point{0, 0});
        std::cout << "Total Point: (" << total.x << ", " << total.y << ")" << std::endl; // Output: Total Point: (9, 12)
    return 0;
}
```

- **Applications of std::reduce algorithm**

The **std::reduce** algorithm in C++ has a wide range of applications across different domains. Here are some key applications along with related examples:

✓ **Numerical Computations**

In this case it can be used for summing or multiplying numerical values.

Example:

```
#include <iostream>

#include <vector>

#include <numeric>

int main() {

   std::vector<int> numbers = {1, 2, 3, 4, 5};

   int sum = std::reduce(numbers.begin(), numbers.end(), 0);

   int product = std::reduce(numbers.begin(), numbers.end(), 1, std::multiplies<int>());

      std::cout << "Sum: " << sum << ", Product: " << product << std::endl; // Output: Sum: 15, Product: 120

   return 0;

}
```

✓ **Data Processing**

It is used for aggregating data from large datasets, such as total sales or average scores.

Example:

```
#include <iostream>

#include <vector>

#include <numeric>

int main() {

   std::vector<double> sales = {100.5, 200.75, 150.0, 300.25};

   double totalSales = std::reduce(sales.begin(), sales.end(), 0.0);
```

```
    double averageSales = totalSales / sales.size();

        std::cout << "Total Sales: " << totalSales << ", Average Sales: " << averageSales
<< std::endl; // Output: Total: 751.5, Average: 187.875

    return 0;

}
```

- ✓ **Parallel Processing**

It can be used for speeding up computations by utilizing multiple threads, particularly in scientific computing.

Example:

```
#include <iostream>

#include <vector>

#include <numeric>

#include <execution>

int main() {

    std::vector<int> numbers(1'000'000, 1); // Vector with 1 million elements, all 1

    int sum = std::reduce(std::execution::par, numbers.begin(), numbers.end(), 0);

        std::cout << "Parallel Sum: " << sum << std::endl; // Output: Parallel Sum:
1000000

    return 0;

}
```

- ✓ **Signal and image processing**

Used in combining pixel values or signal amplitudes for processing.

Example:

```
#include <iostream>

#include <vector>

#include <numeric>

int main() {

    std::vector<int> pixelValues = {255, 128, 64, 32};
```

```
    int totalBrightness = std::reduce(pixelValues.begin(), pixelValues.end(), 0);
    double averageBrightness = totalBrightness /
static_cast<double>(pixelValues.size());
      std::cout << "Total Brightness: " << totalBrightness << ", Average Brightness: "
<< averageBrightness << std::endl; // Output: Total: 479, Average: 119.75
    return 0;
}
```

✓ **Game Development**

It serves for calculating scores or points across various game entities.

Example:

```
#include <iostream>
#include <vector>
#include <numeric>
int main() {
    std::vector<int> scores = {10, 20, 15, 30};
    int totalScore = std::reduce(scores.begin(), scores.end(), 0);
    std::cout << "Total Score: " << totalScore << std::endl; // Output: Total Score: 75
    return 0;
}
```

✓ **Machine Learning**

At this stage it is used for aggregating features from datasets or combining predictions from models.

Example:

```
#include <iostream>
#include <vector>
#include <numeric>
int main() {
    std::vector<double> predictions = {0.1, 0.4, 0.5, 0.2};
```

```
    double ensemblePrediction = std::reduce(predictions.begin(),
predictions.end(), 0.0) / predictions.size();

        std::cout << "Ensemble Prediction: " << ensemblePrediction << std::endl; //
Output: Ensemble Prediction: 0.25

    return 0;

}
```

II. Implement Parallel Data Processing

Parallel data processing refers to the simultaneous execution of computations across multiple processors or cores to speed up processing times and efficiently handle large datasets. In C++, several features and libraries enable parallel data processing, particularly from C++17 onwards.

1. **Key concepts**

- **Execution Policies**:

  C++17 introduced execution policies that allow algorithms to run in parallel or sequentially. The most common are:

  std::execution::par: Enables parallel execution.

  std::execution::seq: Forces sequential execution.

- **Parallel algorithms**

  Standard algorithms in the C++ Standard Library (like **std::for_each**, **std::transform**, and **std::reduce**) can be executed in parallel using these execution policies.

- **Thread Management**

  You can also manage threads directly using the <thread> library for finer control over parallel execution.

2. Applications of Parallel Data Processing

- ✓ **Data Transformation**: Applying transformations to elements of large datasets.
- ✓ **Aggregation Operations**: Summing, averaging, or otherwise combining data elements.
- ✓ **Image and Signal Processing**: Processing pixels or signal data concurrently for performance gains.
- ✓ **Machine Learning**: Training models on large datasets using parallel algorithms.

Example of: Parallel data processing in C++

1. Parallel Transformation Using std::transform

```
#include <iostream>
#include <vector>
#include <execution> // For execution policies
// Function to square a number
int square(int x) {
   return x * x;
}
int main() {
   std::vector<int> numbers(1'000'000, 2); // Vector with 1 million elements initialized to 2
   std::vector<int> results(numbers.size());
   // Perform the transformation using std::transform with parallel execution
   std::transform(std::execution::par, numbers.begin(), numbers.end(), results.begin(), square);
   // Output the first few results to verify
   std::cout << "First 10 squared results: ";
   for (size_t i = 0; i < 10; ++i) {
      std::cout << results[i] << " "; // Should print 4 for all
   }
   std::cout << std::endl;
   return 0;
}
```

2. Parallel Summation Using std::reduce

```
#include <iostream>
#include <vector>
#include <numeric>
```

```
#include <execution>

int main() {

  std::vector<int> numbers(1'000'000, 1); // Vector with 1 million elements initialized to 1

  // Compute the sum using std::reduce with parallel execution

  int sum = std::reduce(std::execution::par, numbers.begin(), numbers.end(), 0);

  std::cout << "Parallel Sum: " << sum << std::endl; // Output: Parallel Sum: 1000000

  return 0;

}
```

3. Parallel Processing Using std::for_each

Here's a complete example that demonstrates how to use std::for_each to increment each element of a large vector in parallel.

```
#include <iostream>

#include <vector>

#include <execution> // For execution policies

int main() {

  // Create a large vector with one million elements initialized to 1

  std::vector<int> numbers(1'000'000, 1);

  // Parallel processing using std::for_each

  std::for_each(std::execution::par, numbers.begin(), numbers.end(), [](int &n) {

    n += 1; // Increment each element

  });

  // Output the first few results to verify

  std::cout << "First 10 results: ";

  for (size_t i = 0; i < 10; ++i) {

    std::cout << numbers[i] << " "; // Should print 2 for all

  }
```

```
    std::cout << std::endl;

    return 0;

}
```

V. Implement Task-Based parallelism

Task-based parallelism in C++ is a programming paradigm that allows developers to express computations as a set of independent tasks that can be executed concurrently.

This model abstracts the complexities of thread management, enabling easier and more efficient parallel programming.

1. Key Characteristics

- **Independence of Tasks**

  Tasks should be able to run independently without requiring communication or synchronization, which simplifies concurrency and reduces potential bottlenecks.

- **Dynamic scheduling**

  A task scheduler manages the execution of tasks, distributing them among available threads. This allows for dynamic load balancing, where tasks can be assigned based on the current load on each processor.

- **Granularity**

  Tasks can vary in size. Fine-grained tasks (smaller units of work) can lead to better resource utilization but may incur overhead from frequent context switching. Coarse-grained tasks (larger units) can reduce overhead but might lead to underutilization of resources.

2. Advantages of task-baesd parallelism

- **Simplified Programming Model**

  Developers can focus on defining tasks rather than managing threads, which reduces the complexity of parallel programming**.**

- **Improved performance**

  By effectively utilizing available CPU cores, task-based parallelism can significantly speed up processing times for computationally intensive applications.

- **Scalability**

Task-based models can easily scale with the addition of more cores or processors, allowing applications to take full advantage of hardware improvements.

3. Implementation in C++

In C++, several features and libraries support task-based parallelism:

✓ **C++ Standard Library**

The std::async and **std::future**

These allow you to run tasks asynchronously and retrieve their results later, promoting a task-oriented approach to concurrency.

✓ **Parallel Algorithms (C++17)**

Algorithms like **std::for_each, std::transform, and std::reduce** can be executed in parallel using execution policies.

✓ **Third-Party Libraries**

Intel TBB (Threading Building Blocks)

It provides a rich set of components for task scheduling, parallel algorithms, and concurrent data structures.

OpenMP

It offers directives for parallelism that can be added to existing code with minimal changes.

Example: Using std::async

Here's a brief example that illustrates the concept:

```
#include <iostream>
#include <vector>
#include <future>
#include <numeric>
int sum_chunk(const std::vector<int>& numbers, size_t start, size_t end) {
   return std::accumulate(numbers.begin() + start, numbers.begin() + end, 0);
}
```

```
int main() {
   std::vector<int> numbers(1'000'000, 1); // 1 million elements initialized to 1
   size_t num_tasks = 4;
   std::vector<std::future<int>> futures;
   // Split the work into tasks
   for (size_t i = 0; i < num_tasks; ++i) {
      size_t start = i * (numbers.size() / num_tasks);
      size_t end = (i + 1) * (numbers.size() / num_tasks);
      if (i == num_tasks - 1) end = numbers.size(); // Handle last chunk
      futures.push_back(std::async(std::launch::async, sum_chunk,
std::ref(numbers), start, end));
   }
   // Aggregate results
   int total_sum = 0;
   for (auto& fut : futures) {
      total_sum += fut.get();
   }
   std::cout << "Total Sum: " << total_sum << std::endl; // Output: Total Sum:
4000000
   return 0;
}
```

Explanation of the Example

Task Definition: The compute_sum function calculates the sum of a given range of numbers.

Task Creation: In the main function, we divide the large vector into chunks and create asynchronous tasks using std::async.

Result Collection: After launching the tasks, we retrieve the results using get() and combine them to produce the final sum

Example: Using std::future for Parallel Summation

Here's a complete example that demonstrates how to use std::future for summing elements in a large vector in parallel.

```
#include <iostream>

#include <vector>

#include <future>

#include <numeric>

// Function to sum a portion of a vector

int sum_chunk(const std::vector<int>& numbers, size_t start, size_t end) {

   return std::accumulate(numbers.begin() + start, numbers.begin() + end, 0);

}

int main() {

   std::vector<int> numbers(1'000'000, 1); // Vector with 1 million elements initialized to 1

   size_t num_tasks = 4;

   std::vector<std::future<int>> futures;

   // Divide the work into tasks

   for (size_t i = 0; i < num_tasks; ++i) {

      size_t start = i * (numbers.size() / num_tasks);

      size_t end = (i + 1) * (numbers.size() / num_tasks);

      if (i == num_tasks - 1) end = numbers.size(); // Handle last chunk

      // Launch the task asynchronously

      futures.push_back(std::async(std::launch::async, sum_chunk, std::ref(numbers), start, end));

   }

   // Aggregate results from each future

   int total_sum = 0;

   for (auto& fut : futures) {

      total_sum += fut.get(); // Blocks until the result is ready
```

```
    }

    std::cout << "Total Sum: " << total_sum << std::endl; // Output: Total Sum: 4000000

    return 0;

}
```

Explanation of the Example

Function Definition: The sum_chunk function takes a vector and a range (start and end indices) and computes the sum of the elements in that range.

Data Initialization: A vector numbers is created with one million elements, all initialized to 1.

Task Creation: The work is divided into chunks. For each chunk, an asynchronous task is launched using std::async, which runs the sum_chunk function.

Result Aggregation: The results from each task are collected using fut.get(), which blocks until the result is ready. These results are summed to get the final total.

Output: The total sum is printed, confirming the correctness of the parallel summation.

4. Applications of Task-Based parallelism

Here are some applications of task-based parallelism along with examples to illustrate how it is used in various domains:

✓ **Scientific Computing**

Example: **Monte Carlo Simulations**

In finance, Monte Carlo methods can be used to assess risk. Each simulation run can be an independent task, allowing multiple simulations to run concurrently.

```
std::vector<std::future<double>> futures;

for (int i = 0; i < num_simulations; ++i) {

    futures.push_back(std::async(std::launch::async, run_simulation));

}
```

✓ **Data Processing**

Example: **ETL Operations**

Extracting, transforming, and loading large datasets can be parallelized. Each stage (e.g., transforming different chunks of data) can run as a separate task.

```
std::vector<std::future<void>> futures;

for (auto& chunk : data_chunks) {

   futures.push_back(std::async(std::launch::async, process_chunk, chunk));

}
```

- ✓ **Image and Video Processing**

  Example: **Image Filtering**
  Applying a filter to different sections of an image can be done in parallel. Each section can be processed as a separate task.

```
std::vector<std::future<Image>> futures;

for (auto& section : image_sections) {

   futures.push_back(std::async(std::launch::async, apply_filter, section));

}
```

- ✓ **Machine Learning**

  Example: **Training Models**
  Training on different batches of data or tuning hyperparameters can be run concurrently. Each training run can be a separate task.

```
std::vector<std::future<Model>> futures;

for (auto& batch : data_batches) {

   futures.push_back(std::async(std::launch::async, train_model, batch));

}
```

- ✓ **Financial Modeling**

  Example: **Risk Assessment**

Running different financial scenarios (e.g., stress tests) can be parallelized. Each scenario can be a separate task.

```
std::vector<std::future<double>> futures;

for (auto& scenario : scenarios) {

   futures.push_back(std::async(std::launch::async, evaluate_scenario, scenario));

}
```

- ✓ **Game Development**

  Example: **AI Processing**

  In a game, different AI characters can make decisions independently. Each AI decision-making process can run as a separate task.

```
std::vector<std::future<Decision>> futures;

for (auto& character : characters) {

   futures.push_back(std::async(std::launch::async, character_decision,
character));

}
```

- ✓ **Web and Network Applications**

  Example: **Concurrent API Calls**
  Fetching data from multiple APIs can be done in parallel. Each API call can be an independent task.

```
std::vector<std::future<Response>> futures;

for (auto& endpoint : api_endpoints) {

   futures.push_back(std::async(std::launch::async, fetch_data, endpoint));

}
```

- ✓ **Robotics and Control Systems**

  Example: **Sensor Fusion**
  Processing data from multiple sensors concurrently allows for faster decision-making in robots. Each sensor's data processing can be a separate task.

```
std::vector<std::future<SensorData>> futures;

for (auto& sensor : sensors) {
```

futures.push_back(std::async(std::launch::async, process_sensor_data, sensor));

}

V. Implement Load Balancing

1. Overview

Load balancing is a technique used to distribute workloads evenly across multiple resources, such as servers, processors, or threads. This ensures that no single resource becomes a bottleneck, optimizing performance, enhancing reliability, and improving overall system responsiveness.

2. Purpose

To maximize resource utilization, minimize response time, and prevent overload on any single resource.

3. Types of Load Balancing

✓ **Static Load Balancing**

Workloads are distributed based on predefined rules, often determined at system design time.

✓ **Dynamic Load Balancing**

Workloads are distributed in real-time based on current system conditions and resource availability.

4. **Algorithms**

Round Robin: Distributes tasks sequentially among resources.

Least Connections: Directs traffic to the resource with the fewest active connections.

Least Response Time: Sends requests to the resource that responds the fastest.

Weighted Load Balancing: Allocates tasks based on resource capacity or performance metrics.

Example in C++

In a C++ context, implementing load balancing can be done through task scheduling and thread management. Here's a simple example using a thread pool to balance tasks across available threads:

```
#include <iostream>
#include <vector>
#include <thread>
#include <future>
#include <queue>
#include <functional>
#include <mutex>
class ThreadPool {
public:
   ThreadPool(size_t num_threads);
   ~ThreadPool();
   template<class F>
   auto enqueue(F&& f) -> std::future<typename std::result_of<F()>::type>;
private:
   std::vector<std::thread> workers;
   std::queue<std::function<void()>> tasks;
   std::mutex queue_mutex;
   std::condition_variable condition;
   bool stop;
};
ThreadPool::ThreadPool(size_t num_threads) : stop(false) {
   for (size_t i = 0; i < num_threads; ++i) {
      workers.emplace_back([this] {
         while (true) {
            std::function<void()> task;
```

```
            {
                std::unique_lock<std::mutex> lock(this->queue_mutex);
                this->condition.wait(lock, [this] { return this->stop || !this->tasks.empty(); });
                if (this->stop && this->tasks.empty())
                    return;
                task = std::move(this->tasks.front());
                this->tasks.pop();
            }
            task();
        }
    });
  }
}
ThreadPool::~ThreadPool() {
  {
    std::unique_lock<std::mutex> lock(queue_mutex);
    stop = true;
  }
  condition.notify_all();
  for (std::thread& worker : workers)
    worker.join();
}
template<class F>
auto ThreadPool::enqueue(F&& f) -> std::future<typename std::result_of<F()>::type> {
  using return_type = typename std::result_of<F()>::type;
```

```
auto task=
std::make_shared<std::packaged_task<return_type()>>(std::forward<F>(f));
    std::future<return_type> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        tasks.emplace([task]() { (*task)(); });
    }
    condition.notify_one();
    return res;
}
int main() {
    ThreadPool pool(4); // Create a thread pool with 4 threads
    std::vector<std::future<int>> results;
    for (int i = 0; i < 10; ++i) {
        results.push_back(pool.enqueue([i] {
            std::this_thread::sleep_for(std::chrono::milliseconds(100));    //    Simulate
work
            return i * i; // Example task
        }));
    }
    for (auto& result : results) {
        std::cout << result.get() << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Explanation of the Example

Thread Pool Class: Manages a fixed number of threads and a queue of tasks. Tasks are distributed among available threads.

Task Enqueuing: Tasks can be submitted to the thread pool, which will execute them based on availability.

Main Function: Creates a thread pool and enqueues several tasks. Each task simulates work and returns a result.

5. Applications of load balancing and examples

Load balancing in C++ can be applied in various domains to optimize resource utilization and enhance performance.

Here are several applications along with examples:

✓ **Web Server Load Balancing**

This is the case of distributing HTTP requests among multiple server instances.

Example: A simple C++ HTTP server can use a round-robin approach to direct requests to different server instances.

```
std::vector<std::string> servers = {"server1", "server2", "server3"};

int current = 0;

std::string get_next_server() {
    std::string server = servers[current];
    current = (current + 1) % servers.size(); // Round-robin
    return server;
}
```

✓ **Thread Pool for Task Distribution**

This is present when distributing computational tasks across multiple threads to enhance parallel processing.

Example: A thread pool that balances tasks among available threads.

```
class ThreadPool {
    // Implementation as discussed earlier
};

ThreadPool pool(4); // 4 threads
```

```
for (int i = 0; i < 10; ++i) {
    pool.enqueue([i] { return i * i; }); // Example task
}
```

Database Query Load Balancing

You can apply load balancing when distributing database queries across multiple replicas.

Example: A simple function to route read queries to replicas.

```
std::vector<std::string> replicas = {"replica1", "replica2", "replica3"};
int read_query_count = 0;
std::string get_next_replica() {
    return replicas[read_query_count++ % replicas.size()]; // Round-robin for reads
}
```

- ✓ **Distributed Computing**

When balancing computation tasks across nodes in a cluster you can use the load balancing approach.

Example: Using a message queue to distribute tasks.

```
std::queue<Task> task_queue; // A queue of tasks
std::mutex queue_mutex;
void worker_thread() {
    while (true) {
        Task task;
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
            if (task_queue.empty()) break;
            task = task_queue.front();
            task_queue.pop();
        }
        process_task(task); // Process the task
```

```
    }
}
```

- ✓ **Microservices Communication**

  Apply load balancing requests between multiple microservice instances.

  Example: A simple load balancer that routes requests.

```
std::map<std::string, std::vector<std::string>> microservices;
std::string route_request(const std::string& service_name) {
    auto& instances = microservices[service_name];
    return instances[current_instance++ % instances.size()]; // Round-robin
}
```

- ✓ **Game Server Load Balancing**

  Lets' consider the case of distributing player connections among multiple game servers.

  Example: Allocating players to servers based on current load.

```
std::map<std::string, int> server_load; // Server name to load
std::string allocate_player() {
    // Find the server with the least load
    return std::min_element(server_load.begin(), server_load.end(),
                            [](const auto& a, const auto& b) { return a.second < b.second; })-
>first;
}
```

Practical Activity 3.3.3: Applying Multithreading and concurrency

Task: C++ Program: Bakery Order Processing

1: Read and perform the following task:

You are a firmware developer and you need to write and compile a C++ program that simulates a real-life scenario of a bakery that processes customer orders using multithreading and concurrency. Each thread represents a baker who prepares different types of baked goods. The program demonstrates how multiple bakers can work concurrently to fulfill customer orders.

2: Read steps involved in applying multithreading and concurrency in C++ program from Key readings **3.3.3.** in the trainee manual

3: Perform the task.

4: Ask assistance if needed

5: Verify the output whether it is the same as the one expected.

Key readings 3.3.3

Applying multithreading and concurrency

This is a C++ program that simulates a real-life scenario of a bakery that processes customer orders using multithreading and concurrency. Each thread represents a baker who prepares different types of baked goods. The program demonstrates how multiple bakers can work concurrently to fulfill customer orders.

C++ Program: Bakery Order Processing Steps

1. **Start your text editor for C++** :

   You can use any text editor (e.g., Visual Studio Code, Sublime Text, or even Notepad++, Dev-C++).

2. **Create a New C++ Source File**

   Create a new file Name the file something like BakerySimulation.cpp.

3. **Write the C++ Code**:

   Using your chosen IDE type in the following codes

```
#include <iostream>
#include <vector>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <chrono>
#include <random>
class Bakery {
public:
   void addOrder(const std::string& order);
   void processOrders();
private:
   std::queue<std::string> orders;
   std::mutex queue_mutex;
   std::condition_variable condition;
};
void Bakery::addOrder(const std::string& order) {
   {
      std::lock_guard<std::mutex> lock(queue_mutex);
      orders.push(order);
   }
   condition.notify_one(); // Notify a baker that an order is available
}
void Bakery::processOrders() {
   while (true) {
      std::string order;
```

```
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
            condition.wait(lock, [this] { return !orders.empty(); }); // Wait until an order is available
            order = orders.front();
            orders.pop();
        }
        // Simulate processing the order
        std::cout << "Processing order: " << order << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(200)); // Simulate time taken to process the order
        std::cout << "Finished order: " << order << std::endl;
    }
}
void customerOrder(Bakery& bakery, const std::string& order) {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 1000)); // Random delay for order placement
    std::cout << "Customer placed order: " << order << std::endl;
    bakery.addOrder(order);
}
int main() {
    Bakery bakery;
    std::vector<std::thread> bakers;
    // Create bakers (threads)
    for (int i = 0; i < 3; ++i) {
        bakers.emplace_back(&Bakery::processOrders, &bakery);
    }
    // Simulate customer orders
```

```
    std::vector<std::thread> customers;
    std::vector<std::string> orders = {"Bread", "Cake", "Cookies", "Muffins", "Brownies"};
    for (const auto& order : orders) {
        customers.emplace_back(customerOrder, std::ref(bakery), order);
    }
    // Wait for customer orders to finish
    for (auto& customer : customers) {
        customer.join();
    }
    // Allow some time for bakers to finish processing remaining orders
    std::this_thread::sleep_for(std::chrono::seconds(5));
    // Stop the bakers (in a real program, you would use a better mechanism)
    for (auto& baker : bakers) {
        baker.detach(); // Detach for demonstration purposes; normally you'd join or signal to stop.
    }
    return 0;
}
```

Explanation of the Code

Bakery Class:

Manages a queue of orders.

Uses a mutex and condition variable for thread-safe access to the orders.

addOrder Method:

Adds a new order to the queue and notifies a waiting baker.

processOrders Method:

Continuously waits for orders to process.

Once an order is available, it simulates processing time and prints messages.

customerOrder Function:

Simulates a customer placing an order with a random delay.

Calls the addOrder method to submit the order to the bakery.

Main Function:

Creates several baker threads to process orders concurrently.

Simulates customer order placements by creating customer threads.

Joins customer threads and allows bakers some time to finish remaining orders.

4. **Compile the Program:**

   By using the execute tab compile the source code to check and eliminate errord

5. **Run the Program:**

   Execute the program by choosing the run from execute tab

6. **Observe the Output**

   You should see output indicating that customers are placing orders and bakers are processing them concurrently.

   The output will show the order being processed and when it is finished.

Points to Remember

Description of multithreading and concurrency

Multithreading in C++ allows developers to create applications that can perform multiple operations concurrently, enhancing efficiency and responsiveness. C++ provides robust support for multithreading through its Standard Library, particularly since C++11, which introduced the **<thread>** library. This allows for the creation, management, and synchronization of threads in a straightforward manner.

In C++, a thread can be spawned by instantiating a std::thread object and passing a function or callable as its argument. This thread runs independently, allowing the main program to continue executing. The use of multiple threads enables better CPU utilization, particularly on multi-core processors, where different threads can run on different cores simultaneously.

Concurrency in C++ is about structuring the program to effectively handle multiple tasks. This can include not only multithreading but also asynchronous programming techniques using **std::async**, which allows functions to run asynchronously and return futures for results. C++ also supports other concurrency constructs, such as condition variables and mutexes from the <**mutex**> library, which help manage shared resources and prevent race conditions.

Challenges associated with multithreading and concurrency in C++ include ensuring thread safety and avoiding deadlocks. Developers must carefully design their applications to manage shared data access, using locks or atomic operations to maintain data integrity. C++ provides atomic types and lock-free programming techniques to help mitigate these issues.

Overall, C++ offers powerful tools for implementing multithreading and concurrency, enabling developers to build high-performance applications that can efficiently handle complex tasks in real-time environments. By leveraging these capabilities, C++ programmers can create responsive applications that maximize resource utilization and improve user experience.

Applying multithreading and concurrency

Applying multithreading and concurrency by Writing and running a C++ Program Simulating a Bakery Processing Customer Orders with Multithreading and Concurrency you must follow these steps:

1. **Set Up the Environment:** Install a C++ compiler (e.g., GCC or MSVC) and an IDE (like Visual Studio Dev-C++).
2. **Define the Program Structure:** Outline the components: classes for Customer, Order, and Baker, each responsible for handling orders and processing.
3. **Create the Classes:** Implement the Customer class to generate orders, the Order class to define order details, and the Baker class to process these orders.
4. **Implement Multithreading** Use the <thread> library to create separate threads for bakers and customers, allowing simultaneous order processing.
5. **Manage Order Queue:** Develop a thread-safe queue to handle incoming orders, ensuring that bakers can access and process them without conflicts.
6. **Synchronize Threads:** Utilize mutexes and condition variables to manage access to the order queue and notify bakers when new orders are available.
7. **Write the Main Function:** Launch customer and baker threads within the main() function, ensuring proper thread management by joining them at the end.
8. **Compile the Program:**Use your IDE or command line to compile the code into an executable.
9. **Run the Simulation:** Execute the program to observe the bakery simulation, showcasing real-time order generation and processing.

10. **Test and Debug:** Conduct tests to check for efficiency and correctness, ensuring that all threads function properly without race conditions or deadlocks.

Application of learning 3.3:

You're tasked with creating a multithreaded C++ application to simulate an embedded temp erature monitoring system. This system includes three temperature sensors generating rand om readings, a dedicated thread processing data and triggering alerts when temperatures ex ceed 75°C, and a communication thread sending alerts to a server. The console provides real -time feedback, displaying current readings, processed notifications, and triggered alerts.

This ensures efficient monitoring and management of temperature data in smart home devi ces.

Indicative content 3.4: Apply inline assembly

Duration: 4hrs

Theoretical Activity 3.4.1: Description of Inline assembly

Tasks:

1: 1: In your groups, answer the questions reflecting to multithreading and concurrency:

1. What do you understand by:
   a) Assembly language
   b) Machine code
2. Describe Low level optimization
3. Discuss fine grained assembly.
4. What is the syntax to include assembly codes into C++?
5. Explain the role of hardware interfacing

2: In your respective groupswrite your key findings on papers/flipcharts

3: Make presentation of the findings to the whole class

4: Ask questions for more clarifications if any

5: Read through Key **readings 3.4.1** in the trainee manual

Key readings 3.4.1.:

Description of inline assembly

1. **Introduction**

   Inline assembly is a feature in C++ that allows programmers to embed assembly language instructions directly within their C or C++ code. This capability provides a means to execute low-level operations while still leveraging the structure and syntax of high-level languages. Inline assembly can be particularly useful for performance optimization, hardware interaction, and when specific CPU instructions are required that are not available in the high-level language

2. **Characteristics of inline assembly**

   Direct Integration:

   Inline assembly can be written alongside regular C++ code, making it easier to mix high-level logic with low-level operations. This integration allows developers to

optimize critical sections of their code without having to create separate assembly files.

- ✓ **Performance Enhancement:**

  Programmers can use inline assembly to write highly optimized code for performance-critical tasks. For example, low-level mathematical operations, graphics rendering, or specialized algorithms can benefit from direct access to CPU instructions.

- ✓ **Access to Hardware**

  Inline assembly allows for direct manipulation of hardware registers and control of system resources. This is particularly useful in embedded systems, device drivers, and operating systems where hardware interaction is necessary.

- ✓ **Architecture-Specific**

  The syntax and capabilities of inline assembly vary between different compilers and processor architectures. For instance, inline assembly in GCC differs from that in Microsoft Visual C++, making it essential for developers to understand the specific environment they are working in.

- ✓ **Complexity and Maintainability**

  While inline assembly can enhance performance, it adds complexity to the code-base. It requires a deeper understanding of the underlying architecture and assembly language, which can make code harder to read and maintain.

3. Assembly Language

a. Definition

Assembly language is a low-level programming language that serves as a symbolic representation of a computer's machine code. It provides a way to write instructions that can be directly executed by the CPU, offering a more human-readable format compared to binary or hexadecimal machine language.

b. Key Features

- ✓ **Symbolic Representation:**

  Assembly language uses mnemonic codes (like MOV, ADD, SUB) instead of numeric opcodes, making it easier for programmers to understand and write instructions.

- ✓ **Architecture-Specific:**

  Each type of CPU architecture has its own assembly language. This means that code written for one architecture (e.g., x86) cannot be directly executed on another (e.g., ARM) without modification.

✓ **Direct Hardware Control**

Assembly language allows programmers to directly manipulate hardware components, including registers, memory addresses, and I/O ports. This level of control is essential for system programming and embedded systems.

✓ **Efficiency**

Programs written in assembly language can be highly optimized for performance and resource usage, making it suitable for critical applications like operating systems, device drivers, and real-time systems.

c. **Low-Level Operations**

Assembly language enables operations that are not possible in high-level languages, such as precise control over CPU instructions and manipulation of bits and bytes.

✓ **Basic Syntax**

Assembly language syntax generally consists of several key components: labels, mnemonics (instructions), operands, and comments. The exact syntax can vary depending on the assembler and the CPU architecture.

✓ **Basic Components**

- **Labels**

  A symbolic name that represents a memory address. They are used for jumps and branches.

  Example: loop_start:

- **Mnemonics:**

  The actual instruction to be executed, such as MOV, ADD, SUB.

  Example: MOV EAX, 1 (moves the value 1 into the EAX register).

- **Operands:**

  Values or addresses that the instructions operate on. These can be registers, memory locations, or immediate values.

  Example: EBX, 5, var_name.

- **Comments:**

  Text that is ignored by the assembler, usually providing explanations. Comments typically start with a semicolon (;).

Example: ; This is a comment

Example of Assembly Code

Here's a simple example using x86 assembly language that adds two numbers and prints the result:

```
section .data
   num1 db 5            ; Define first number
   num2 db 3            ; Define second number
   result db 0          ; Define space for result
section .text
   global _start        ; Entry point for the program

_start:
   ; Load numbers into registers
   mov al, [num1]       ; Load num1 into AL
   add al, [num2]       ; Add num2 to AL
   mov [result], al     ; Store result

   ; Exit the program
   mov eax, 1           ; syscall: sys_exit
   xor ebx, ebx         ; exit code 0
   int 0x80             ; call kernel
```

Breakdown of the Example:

- **Data Section:**

  The .data section is where initialized data is declared. Here, num1, num2, and result are defined.

- **Text Section:**

  The .text section contains the actual code. The global _start directive specifies the entry point.

- **Instructions:**

mov al, [num1]: Moves the value of num1 into the AL register.

add al, [num2]: Adds the value of num2 to the AL register.

mov [result], al: Stores the sum back into the result variable.

- **Program Exit:**

The program uses system calls to exit. mov eax, 1 indicates a syscall to exit, and xor ebx, ebx sets the exit code to 0.

d. Use Cases

✓ **Operating Systems:**

Writing kernels and low-level system utilities that require direct hardware access.

✓ **Embedded Systems:**

Programming microcontrollers and other devices with limited resources where efficiency is critical.

✓ **Performance-Critical Applications:**

Optimizing specific routines in high-performance applications like graphics engines and scientific computing.

✓ **Device Drivers:**

Interfacing with hardware components to control peripherals.

✓ **Reverse Engineering:**

Analyzing and understanding existing binary executables through disassembly.

4. The asm

a. Definifition

In C++, **asm** refers to inline assembly, which allows you to embed assembly language instructions directly within your C++ code. This can be useful for performance optimization or accessing specific hardware features not easily available through C++.

b. Features of asm

✓ **Syntax**

The syntax for inline assembly varies between compilers. For example, GCC uses asm or __asm__, while MSVC uses __asm.

In GCC, the syntax for inline assembly is as follows:

asm (assembler_template : output_operands : input_operands : clobbered_registers);

where:

- **assembler_template:** The assembly code itself, which can include placeholders for operands.
- **output_operands**: Specifies variables that will receive values from the assembly code, using constraints (e.g., =r for a register).
- **input_operands:** Specifies variables that the assembly code will read, also using constraints.
- **clobbered_registers:** Lists any registers that the assembly code modifies but does not output, indicating to the compiler that those registers may have changed.

Example

```
#include <iostream>

int main() {
   int result;
   asm (
      "movl $5, %0;"   // Move 5 into the output operand
      "addl $10, %0;"  // Add 10 to the value
      : "=r" (result)  // Output operand
   );
   std::cout << "Result: " << result << std::endl; // Should print 15
   return 0;
}
```

Notes:

Placeholders (e.g., %0): refer to the order of the operands specified after the assembly code.

Use volatile to prevent the compiler from optimizing away the inline assembly when necessary.

In MSVC (Microsoft Visual C++), the syntax for inline assembly is as follows:

```
__asm {
   // Assembly instructions here
}
```

Example

```
#include <iostream>

int main() {
   int result;
   __asm {
      mov eax, 5     // Move 5 into the EAX register
      add eax, 10    // Add 10 to EAX
      mov result, eax // Move the value from EAX to result
   }
   std::cout << "Result: " << result << std::endl; // Should print 15
   return 0;
}
```

Key Points:

Direct Block: You write the assembly code directly within the curly braces.

Registers: You can use x86 registers like eax, ebx, etc.

No Constraints: Unlike GCC, MSVC does not use output or input constraints in the same way, making it simpler but less flexible for passing variables.

Portability: Inline assembly is specific to the x86 architecture and may not work in other environments.

Integration: Inline assembly is often used to optimize specific parts of code where high performance is required.

Control: It offers precise control over CPU registers and instructions, enabling operations that may not be possible in standard C++.

Portability: Code using inline assembly is less portable, as it is architecture-specific.

5. Fine- grained assembly

a. Definition

In the context of inline assembly in C++, **fine-grained** refers to the ability to control specific low-level operations with high precision. This allows developers to optimize performance and utilize processor features in a very detailed manner

b. Features of Fine-Grained Assembly

Here are some key features associated with fine-grained assembly:

- ✓ **Precision Control**

  It allows developers to control individual CPU instructions, enabling optimization for specific operations and better performance.

- ✓ **Optimized Resource Usage**

  Enables efficient management of CPU registers and memory, allowing for tailored use of resources based on the application's needs.

- ✓ **Enhanced Performance Tuning**

  It provides the ability to fine-tune specific parts of code for speed and efficiency, often resulting in faster execution times.

- ✓ **Direct Hardware Interaction**

  It facilitates direct manipulation of hardware features, such as processor flags and special registers, allowing developers to leverage specific hardware capabilities.

- ✓ **Reduced Overhead**

  Fine-grained assembly can minimize overhead by eliminating unnecessary function calls or high-level abstractions, leading to leaner, faster code.

- ✓ **Targeted Optimization**

  It allows for micro-optimizations in critical sections of code, such as loop unrolling or branch prediction adjustments.

- ✓ **Low-Level Access**

  It provides low-level access to system resources and peripherals, essential for system programming, embedded systems, and performance-critical applications.

- ✓ **Control Flow Management**

  Enables detailed control over program flow, such as precise branching and looping mechanisms, which can be optimized for specific use cases.

  Examples

  Here's a simple example that demonstrates fine-grained assembly in a context where low-level manipulation of registers and memory addresses can yield performance improvements. This example calculates the sum of an array of integers using inline assembly in both GCC and MSVC.

  In GCC:

```
#include <iostream>

int main() {
```

```
    int arr[] = {1, 2, 3, 4, 5};
    int sum = 0;
    int length = sizeof(arr) / sizeof(arr[0]);
    // Fine-grained assembly to calculate the sum of the array
    asm (
        "movl $0, %%eax;"         // Clear EAX (accumulator for sum)
        "movl %1, %%ecx;"         // Load the length of the array into ECX
        "movl %0, %%edi;"         // Load the base address of the array into EDI
    "loop_start:"
        "addl (%%edi), %%eax;"    // Add the value at address in EDI to EAX
        "addl $4, %%edi;"         // Move to the next integer (4 bytes)
        "loop %%ecx, loop_start;" // Loop until ECX is 0
        : "=a" (sum)              // Output: EAX to sum
        : "r" (length), "r" (arr) // Inputs: length and base address of arr
        : "%ecx", "%edi"          // Clobbered registers
    );
    std::cout << "Sum: " << sum << std::endl; // Should print 15
    return 0;
}
```

In MSVC:

```
#include <iostream>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int sum = 0;
    int length = sizeof(arr) / sizeof(arr[0]);
    __asm {
        mov eax, 0        // Clear EAX (accumulator for sum)
        mov ecx, length   // Load the length of the array into ECX
```

```
        lea edi, arr       // Load the address of the array into EDI
    loop_start:
        add eax, [edi]     // Add the value at address in EDI to EAX
        add edi, 4         // Move to the next integer (4 bytes)
        loop loop_start    // Loop until ECX is 0
    }
    sum = eax; // Move the result from EAX to sum
    std::cout << "Sum: " << sum << std::endl; // Should print 15
    return 0;
}
```

Explanations:

Registers: The examples use registers (EAX, ECX, EDI) to perform arithmetic and manage control flow.

Memory Manipulation: The assembly code accesses the array directly through memory addresses.

Looping: The loop structure uses the loop instruction, which decrements ECX and continues looping until it reaches zero.

Efficiency: This approach can be more efficient than a high-level loop due to reduced overhead in managing control flow and memory accesses.

c. Applications of fine-grained

Fine-grained control using inline assembly in C++ allows developers to access hardware features, optimize performance-critical sections of code, or manipulate system-level registers. Here are some common applications of using fine-grained inline assembly with examples:

✓ **Optimizing Performance-Critical Code**

In some cases, specific hardware instructions can improve the performance of certain operations like mathematical computations or data handling. Inline assembly can be used to access these specialized instructions for optimization.

Example:

Using **ADD** instruction for faster integer addition

```
#include <iostream>
int add_numbers(int a, int b) {
   int result;
   asm("addl %%ebx, %%eax;"
      : "=a" (result)
      : "a" (a), "b" (b));
   return result;
}
int main() {
   int x = 5, y = 10;
   std::cout << "Result: " << add_numbers(x, y) << std::endl;
   return 0;
}
```

This example shows how to use the ADD instruction to add two integers directly using assembly languge

- ✓ **Accessing CPU-Specific Instructions**

Certain instructions may be CPU-specific and not available in high-level C++. For example, Intel provides specialized instructions such as rdtsc to read the processor's timestamp counter, which can be useful for benchmarking and timing.

Example:

Using **RDTSC** for high-resolution timing

```
#include <iostream>
#include <cstdint>
uint64_t read_rdtsc() {
   unsigned int lo, hi;
   asm volatile ("rdtsc" : "=a"(lo), "=d"(hi));
   return ((uint64_t)hi << 32) | lo;
}
```

```
int main() {
    uint64_t start = read_rdtsc();
    // Do something time-consuming here
    uint64_t end = read_rdtsc();
    std::cout << "CPU cycles: " << (end - start) << std::endl;
    return 0;
}
```

This uses RDTSC to measure the number of CPU cycles elapsed between two points in time, allowing fine-grained benchmarking.

- ✓ **Interacting with Hardware**

Inline assembly is often used in embedded systems to interact with hardware by setting or clearing bits in registers or memory-mapped I/O.

Example:

Controlling an I/O Port

```
#include <iostream>
void outb(unsigned short port, unsigned char val) {
    asm volatile ("outb %0, %1" : : "a"(val), "Nd"(port));
}
int main() {
    outb(0x80, 0xFF);  // Write 0xFF to I/O port 0x80
    return 0;
}
```

This example shows writing a value to an I/O port, which can be useful in embedded programming.

- ✓ **System Programming: Managing CPU Privileges**

Inline assembly can also be used for tasks such as modifying CPU flags or switching between different CPU modes.

Example:

Enabling and Disabling Interrupts

```
#include <iostream>
void disable_interrupts() {
   asm volatile ("cli");  // Clear interrupt flag
}
void enable_interrupts() {
   asm volatile ("sti");  // Set interrupt flag
}
int main() {
   disable_interrupts();
   // Critical section where interrupts must be disabled
   enable_interrupts();
   return 0;
}
```

This code demonstrates how to disable and enable CPU interrupts using the CLI and STI instructions.

- ✓ **Atomic Operations**

Inline assembly allows for the use of atomic instructions, ensuring that operations like incrementing a shared counter are done without interference from other threads or CPUs.

Example:

Atomic Increment Using **LOCK** Prefix

```
#include <iostream>
void atomic_increment(int* ptr) {
   asm volatile("lock incl %0" : "+m" (*ptr));
}
int main() {
   int counter = 0;
   atomic_increment(&counter);
```

```
    std::cout << "Counter: " << counter << std::endl;
    return 0;
}
```

This example shows how to perform an atomic increment of a shared variable using the LOCK prefix.

- ✓ **Accessing and Manipulating CPU Registers**

Inline assembly allows you to directly read or write CPU registers, which can be useful in debugging, context switching, or low-level tasks in operating systems.

Example

Reading CPU's EFLAGS Register

```
#include <iostream>
unsigned int get_eflags() {
    unsigned int eflags;
    asm volatile("pushf\n\t"
                 "pop %0"
                 : "=r"(eflags));
    return eflags;
}
int main() {
    unsigned int flags = get_eflags();
    std::cout << "EFLAGS Register: " << std::hex << flags << std::endl;
    return 0;
}
```

This example shows how to read the EFLAGS register using inline assembly.

- ✓ **Security Features: Preventing Buffer Overflows**

Inline assembly can be used in security-sensitive applications to enforce certain constraints, like stack integrity checks, or zeroing out sensitive data in memory.

Example:

Securely Clearing a Sensitive Variable

```
#include <cstring>
#include <iostream>
void secure_clear(void* ptr, size_t size) {
   asm volatile("rep stosb"
                : "+D"(ptr), "+c"(size)
                : "a"(0)
                : "memory");
}
int main() {
   char password[] = "secret_password";
   std::cout << "Before clear: " << password << std::endl;
   secure_clear(password, strlen(password));
   std::cout << "After clear: " << password << std::endl;
   return 0;
}
```

This shows clearing sensitive data like passwords from memory to prevent it from being retained after usage.

6. **Machine code**

a. Definition

Is the lowest-level programming language, consisting of binary instructions that a computer's central processing unit (CPU) can directly execute. Each instruction corresponds to specific operations that the hardware can perform, such as arithmetic calculations, data movement, and control flow.

b. Characteristics of machine code

✓ **Binary Format**

Machine code is represented in binary (0s and 1s), making it difficult for humans to read or write. Each instruction is a sequence of bits that encodes specific operations and operands.

✓ **Architecture-Specific:**

Machine code is closely tied to the architecture of the CPU. Different CPU families (like x86, ARM, or MIPS) have their own machine code formats, meaning code compiled for one architecture will not run on another without modification.

- ✓ **Direct Execution:**

  Unlike higher-level languages, machine code instructions are executed directly by the CPU without the need for translation or interpretation.

- ✓ **Instruction Set Architecture (ISA):**

  Machine code is based on a specific set of instructions that a CPU can execute, known as the **Instruction Set Architecture (ISA)**. This ISA defines the set of operations that the processor can perform, such as arithmetic, logical operations, data movement, etc.

- ✓ **Instruction Format**

  Machine code is usually composed of:

  Opcode (Operation Code): Specifies the operation to be performed (e.g., addition, subtraction, load, store).

  Operands: The data on which the operation acts, which could be registers, memory addresses, or immediate values.

- ✓ **Memory Addressing**

  Machine code often interacts with memory, and instructions include mechanisms to address or refer to specific memory locations. This can be done through direct addressing (specifying the memory address directly), indirect addressing (using pointers or registers), or other complex addressing modes

  Example

  Let's consider an example in **x86 architecture**. A simple operation like adding two numbers could look like this in machine code:

  Assembly (human-readable form):

  mov eax, 5 ; Move the value 5 into register EAX

  add eax, 3 ; Add 3 to the value in EAX

  Corresponding Machine Code (in hexadecimal):

  B8 05 00 00 00 ; Move 5 into EAX (B8 is the opcode for mov)

  83 C0 03 ; Add 3 to EAX (83 is the opcode for add)

  Here, B8 is the opcode for the mov instruction, followed by 05 00 00 00, which is the immediate value 5 stored in little-endian format. The next instruction 83 C0 03 is an add operation, where C0 refers to the register EAX, and 03 is the value being added.

  Corresponding Machine Code (in binary):

10111000 00000101 00000000 00000000 00000000 // mov eax, 5

10000011 11000000 00000011 // add eax, 3

c. Use Cases

✓ **Operating Systems**

Machine code is essential for operating systems, as it directly controls hardware resources.

✓ **Embedded Systems**

Embedded devices often run on machine code optimized for specific tasks with limited resources.

✓ **Performance-Critical Applications**

In high-performance computing, machine code can be used to squeeze out maximum performance by avoiding high-level abstractions.

✓ **Reverse Engineering**

Understanding machine code is crucial for reverse engineering applications, malware analysis, and debugging.

✓ **Advantages of Machine Code**

Performance: Since machine code is directly executed by the CPU, it is highly efficient, as there is no need for interpretation or translation during runtime.

Low-Level Access: Machine code provides direct control over the hardware, which is beneficial in systems programming (like operating system kernels or embedded systems).

d. Disadvantages of Machine Code

✓ **Difficult to Write and Debug**

Because it consists of binary numbers and lacks human-readable syntax, machine code is difficult to write, understand, and debug compared to higher-level languages.

✓ **Architecture-Specific**

Machine code written for one type of CPU architecture cannot be directly executed on another without recompilation or emulation.

e. Translation from Higher-Level Languages

When you write code in a high-level language like C++ or Python, it is eventually translated into machine code through a process known as **compilation**. The

compiler translates human-readable code into assembly language, which is then converted into machine code. This final machine code is what the CPU executes.

For example, consider this C++ code:

```
int add(int a, int b) {
   return a + b;
}
```

When compiled, it may eventually result in machine code similar to the following (in hexadecimal for x86):

```
55          ; push ebp
89 e5       ; mov ebp, esp
8b 45 08    ; mov eax, [ebp+8] (load argument a)
03 45 0c    ; add eax, [ebp+12] (add argument b)
5d          ; pop ebp
c3          ; return
```

And it may result in machine code similar to the following (in binary for x86):

```
01010101                    // push ebp
10001001 11100101           // mov ebp, esp
10001011 01000101 00001000  // mov eax, [ebp + 8]
00000011 01000101 00001100  // add eax, [ebp + 12]
01011101                    // pop ebp
11000011                    // ret
```

ii. I/O Operands

1. Definition

In inline assembly, I/O operands refer to the variables in your C/C++ code that interact with the assembly code, either by supplying input values to the assembly block or by receiving output from it. These operands form the bridge between the assembly instructions and the higher-level C/C++ variables, allowing you to pass values in and out of the assembly block.

2. **Types of I/O Operands**

✓ **Input Operands**

These operands supply data from the C/C++ code to the assembly block. They are read-only from the perspective of the assembly code.

Example

If you have a variable in C++ (int a = 10;) and you want to use its value in an assembly operation, you would declare a as an input operand.

✓ **Output Operands**

These operands are used to return data from the assembly block back into the C/C++ program. They are write-only from the perspective of the assembly code.

Examples

If the result of an assembly computation should be stored in a C++ variable (int result;), you would declare result as an output operand.

Example of I/O Operands in Inline Assembly

```
int a = 5, b = 3, result;

asm ("add %1, %2\n\t"  // Assembly instruction: add a and b
    "mov %2, %0"      // Move the result into the output operand
    : "=r" (result)   // Output operand: result stored in a register ("r")
    : "r" (a), "r" (b)  // Input operands: a and b stored in registers ("r")
    :                 // No clobbered registers in this case
);
```

Explanation

- **Input operands ("r" (a), "r" (b)):** The values of **a** and **b** are passed into the assembly block and stored in registers.
- **Output operand ("=r" (result)):** The result of the computation is stored in a register and assigned to result in the C code.
- **Operand Constraints**

  To specify how the operands are used (input/output) and where they should be stored (registers, memory, immediate values), we use constraints:

✓ **Input Operand Constraints**

These constraints tell the compiler how the assembly code will access the input variables.

For example

"r": Use a general-purpose register.

"m": Use a memory address.

✓ **Output Operand Constraints**

These constraints indicate how the output variable will be stored.

For example

"=r": Write the output value to a general-purpose register.

✓ **Importance of Operand Constraints**

The constraints are essential because they inform the compiler how to allocate resources (like registers or memory) for passing values between your C/C++ code and the assembly block. This ensures the correct values are used

iii. **Use register constraints**

1. Definition

Register constraints are typically used in assembly programming and compiler optimizations to specify how and where certain variables or values should be stored or operated on within the CPU. Registers are the fastest form of memory available to a CPU, so optimizing register usage can significantly improve the efficiency and speed of code execution.

2. Examples

Here's a more detailed explanation of how register constraints are used in different contexts:

✓ **Assembly Language Programming**

Direct Register Access: In low-level assembly language, registers are accessed directly to store variables, intermediate results, and addresses. When writing assembly, the programmer can explicitly choose which register to use for different operations. Each register often has a specific purpose or is optimized for certain operations (e.g., the eax register in x86 architecture for arithmetic operations).

Register Constraints: Some instructions or architectures may constrain certain operations to specific registers. For example, division operations in x86 architecture typically require the use of the eax register for the dividend.

✓ **Inline Assembly in High-Level Languages (e.g., C/C++ with GCC)**

When embedding assembly code within a higher-level language like C or C++, you can use register constraints to instruct the compiler to store specific variables in

certain registers. This is typically done using inline assembly with the asm keyword in GCC.

Syntax: The basic syntax in GCC inline assembly for specifying register constraints looks like this:

```
asm("assembly-code"
   : output-constraints
   : input-constraints
   : clobbers);
```

Example

Here's an example using register constraints:

```
int a = 10, b;
asm ("addl %%ebx, %%eax" // Assembly code: adds `ebx` to `eax`
    : "=a"(b)          // Output constraint: `b` will be stored in `eax`
    : "a"(a), "b"(5));   // Input constraints: `a` will be stored in `eax`, and 5 in `ebx`
```

In this example:

"=a"(b) means the result will be stored in the eax register and then assigned to b.

"a"(a) means the variable a is assigned to the eax register.

"b"(5) means the constant 5 is stored in the ebx register.

✓ **Compiler Optimizations**

Compilers, particularly those that allow inline assembly (like GCC or Clang), enable developers to constrain certain variables to specific registers during the execution of inline assembly or critical sections of code. These constraints help the compiler understand how to allocate registers and optimize register usage, balancing between the need for performance and the available CPU resources.

Register constraints can also be used to ensure that certain registers are not overwritten or "clobbered" by the compiler during function execution.

3. **Types of Register Constraints**

✓ **General Register Constraints**

These allow the compiler to select any available general-purpose register. For instance, "r"(a) tells the compiler that **a** can be placed in any general-purpose register.

✓ **Specific Register Constraints**

These force a variable to be placed in a particular register. For example, "a"(a) forces **a** to be placed in the eax register on x86 architectures.

- ✓ **Output Constraints**

These specify how the output of an inline assembly block is mapped to C++ variables. For example, **"=r"(b)** tells the compiler to place the result of the assembly operation in a general-purpose register and then store it in **b**.

- ✓ **Input Constraints**

These define how input C variables or constants are passed into the assembly code.

- ✓ **Clobber Constraints**

These are used to tell the compiler which registers or memory locations will be modified ("clobbered") by the assembly code so it can avoid using them for other variables. For instance, "cc" indicates that the assembly code will modify the condition code register, and "memory" tells the compiler that memory will be changed, requiring it to reload values from memory afterward.

4. Use Cases for Register Constraints

- ✓ **Performance Optimization**

When writing performance-critical code, such as in embedded systems or real-time applications, register constraints allow a developer to reduce memory access and directly operate on data stored in registers, which is much faster.

- ✓ **Hardware-Specific Code**

In systems programming, where software closely interacts with hardware, specific registers may be required for certain operations, especially when interacting with device drivers, peripherals, or interrupt handlers.

- ✓ **Cross-Platform Code**

When targeting different CPU architectures, specifying register constraints can help ensure that the code runs optimally on different processors by utilizing the appropriate registers for each architecture.

Examples in GCC Inline Assembly

Example1:

Below is an example that shows how to use inline assembly (**asm**) in GCC to specify register constraints. The code demonstrates how you can use certain registers directly using register constraints.

```
int a = 5, b = 10, result;
asm("addl %%ebx, %%eax;"
```

```
    : "=a"(result)     // Output: store the result in `eax`
    : "a"(a), "b"(b)); // Input: `a` in `eax`, `b` in `ebx`
printf("Result: %d\n", result);
```

The **asm** block adds **b** to **a** using the **eax** and **ebx** registers.

The result is stored in the **eax** register and then moved to the result variable.

Register Constraints:

"a" means the value is stored in the EAX register.

"b" means the value is stored in the EBX register.

"=a" means that the result should be written to the EAX register.

Example 2: Simple Addition

This example demonstrates adding two integers using a register constraint for the result.

```
#include <iostream>

int add(int a, int b) {
    int result;
    asm volatile (
        "add %[a], %[b];"
        : [result] "=r" (result) // Output operand
        : [a] "r" (a), [b] "r" (b) // Input operands
    );
    return result;
}


int main() {
    std::cout << "5 + 3 = " << add(5, 3) << std::endl;
    return 0;
}
```

Example 3: Using Multiple Registers

This example uses multiple registers to perform a calculation involving multiplication.

```
#include <iostream>
int multiply(int a, int b) {
   int result;
   asm volatile (
      "imul %[b];"
      : "=r" (result)
      : "r" (a), [b] "r" (b) // Input operand b
      : "cc" // Clobber list (condition codes)
   );
   return result;
}
int main() {
   std::cout << "4 * 5 = " << multiply(4, 5) << std::endl;
   return 0;
}
```

Example 4: Using Specific Registers

In this example, we will attempt to use specific registers (like eax for the result).

```
#include <iostream>
int subtract(int a, int b) {
   int result;
   asm volatile (
      "sub %[b], %[a];"
      : "=a" (result) // Use eax for the output
      : [a] "r" (a), [b] "r" (b)
   );
```

```
    return result;
}
int main() {
    std::cout << "10 - 3 = " << subtract(10, 3) << std::endl;
    return 0;
}
```

Example 5: Division with Remainder

This example demonstrates integer division, returning both quotient and remainder using output operands.

```
#include <iostream>
void divide(int a, int b, int &quotient, int &remainder) {
    asm volatile (
        "xor edx, edx;" // Clear remainder
        "div %[b];"     // Divide eax by b
        : "=a" (quotient), "=d" (remainder) // Output operands
        : "r" (a), [b] "r" (b) // Input operands
    );
}
int main() {
    int q, r;
    divide(20, 3, q, r);
    std::cout << "20 / 3 = " << q << ", remainder = " << r << std::endl;
    return 0;
}
```

Example 6: Swapping Two Variables

This example demonstrates how to swap two integers using registers.

```
#include <iostream>
```

```
void swap(int &a, int &b) {
    asm volatile (
        "xchg %[a], %[b];"
        : [a] "+r" (a), [b] "+r" (b) // Use '+' to indicate read/write
    );
}
int main() {
    int x = 10, y = 20;
    swap(x, y);
    std::cout << "After swap: x = " << x << ", y = " << y << std::endl;
    return 0;
}
```

Example 7: Square Using Register

This example demonstrates how to compute the square of an integer.

```
#include <iostream>
int square(int x) {
    int result;
    asm volatile (
        "imul %[x];"
        : "=r" (result)
        : "0" (x) // Input operand
    );
    return result;
}
int main() {
    std::cout << "Square of 6 = " << square(6) << std::endl;
    return 0;
}
```

Example 8: Modulus Operation

This example demonstrates how to compute the modulus of two integers.

```
#include <iostream>
int modulus(int a, int b) {
   int result;
   asm volatile (
      "xor edx, edx;" // Clear edx for remainder
      "div %[b];"
      : "=d" (result) // Remainder in edx
      : "a" (a), [b] "r" (b) // a in eax, b in register
   );
   return result;
}
int main() {
   std::cout << "20 % 3 = " << modulus(20, 3) << std::endl;
   return 0;
}
```

iv. Low-level optimization

1. Definition

Low-level optimization refers to techniques that improve the performance of software at a close-to-hardware level, typically focusing on memory management, CPU instruction usage, and other hardware-specific features.

2. Strategies for low-level optimization

Low-level optimization in programming, particularly in C++, focuses on improving the performance of applications by leveraging specific features of the hardware and compiler. Here are some effective strategies for low-level optimization:

✓ **Use of inline assembly**

Inline assembly can be used to directly control CPU instructions, allowing for more efficient code. It enables you to take advantage of specific CPU features or instruction sets.

```
#include <iostream>
void optimizedFunction(int a, int b) {
   int result;
   asm volatile (
      "addl %1, %0"      // Assembly instruction to add a and b
      : "=r" (result)    // Output operand
      : "r" (a), "0" (b) // Input operands
   );
   std::cout << "Result: " << result << std::endl;
}
```

- ✓ **Profile-Guided optimization (PGO)**

  Using profiling tools to analyze your program and understand which parts consume the most resources can guide optimization efforts.

- ✓ **Memory Access Optimization**

  Data Locality: Ensure that frequently accessed data is stored together in memory to take advantage of CPU caching.

  Cache-Friendly Structures: Use structures like arrays of structures (AoS) versus structures of arrays (SoA) to improve data access patterns.

- ✓ **Loop Unrolling**

  Increase the performance of loops by reducing the overhead of loop control and allowing for better instruction pipelining.

```
for (int i = 0; i < n; i += 4) {
   result[i] = a[i] + b[i];
   result[i + 1] = a[i + 1] + b[i + 1];
   result[i + 2] = a[i + 2] + b[i + 2];
   result[i + 3] = a[i + 3] + b[i + 3];
}
```

- ✓ **Use of SIMD Instructions:**

Single Instruction, Multiple Data (SIMD) allows a single instruction to process multiple data points simultaneously. This is often done using compiler intrinsics or assembly.

```
#include <immintrin.h>

void addSIMD(float* a, float* b, float* result, int n) {
   for (int i = 0; i < n; i += 4) {
      __m128 a_vals = _mm_load_ps(&a[i]);
      __m128 b_vals = _mm_load_ps(&b[i]);
      __m128 result_vals = _mm_add_ps(a_vals, b_vals);
      _mm_store_ps(&result[i], result_vals);
   }
}
```

- ✓ **Compiler Optimization Flags**

  Using appropriate compiler flags can significantly enhance performance. Common flags include:

  -O2 or -O3 for general optimizations.

  -march=native to enable optimizations specific to the architecture of the machine.

  -funroll-loops to enable loop unrolling.

- ✓ **Avoiding Function Calls**

  Minimize the use of function calls in performance-critical sections. Inline functions or macros can sometimes help reduce the overhead.

- ✓ **Algorithm Optimization**

  While not strictly low-level, ensuring that algorithms are optimal (e.g., using quicksort instead of bubble sort) can have a massive impact on performance.

- ✓ **Threading and Concurrency**

  Utilize multithreading or asynchronous programming to leverage multiple CPU cores effectively.

- ✓ **Memory Allocation Optimization**

Using custom memory allocators or pooling techniques to reduce the overhead associated with dynamic memory allocation can enhance performance.

Example Program with Multiple Optimizations:

Here's a simple C++ example incorporating several of these techniques:

```
#include <iostream>
#include <immintrin.h>
void addArrays(float* a, float* b, float* result, int n) {
   // Loop unrolling with SIMD
   for (int i = 0; i < n; i += 4) {
      __m128 a_vals = _mm_load_ps(&a[i]);
      __m128 b_vals = _mm_load_ps(&b[i]);
      __m128 result_vals = _mm_add_ps(a_vals, b_vals);
      _mm_store_ps(&result[i], result_vals);
   }
}
int main() {
   const int N = 1024;
   float a[N], b[N], result[N];
   // Initialize arrays
   for (int i = 0; i < N; ++i) {
      a[i] = static_cast<float>(i);
      b[i] = static_cast<float>(N - i);
   }
   addArrays(a, b, result, N);
   std::cout << "Result: " << result[0] << ", " << result[1] << ", " << result[2] << ", "
<< result[3] << std::endl;
   return 0;
}
```

3. **Importance of low-level optimization**

Low-level optimization is a fundamental practice in software development that can lead to substantial improvements in performance, resource utilization, and overall efficiency

a. Performance Improvement

✓ **Execution Speed**

Low-level optimization can significantly reduce the execution time of programs. This is especially important in performance-critical applications such as gaming, scientific computing, and real-time systems where every millisecond counts.

✓ **Resource Utilization**

Optimized code can make better use of system resources (CPU, memory, etc.), leading to more efficient applications that can handle larger workloads or more complex computations.

b. Memory Efficiency

✓ **Reduced Memory Footprint**

Low-level optimizations can minimize memory usage by reducing the size of data structures, using memory more efficiently, and employing techniques such as memory pooling and custom allocators.

✓ **Improved Cache Performance**

By optimizing data access patterns, developers can increase cache hits and decrease cache misses, which significantly improves performance because accessing data from cache is much faster than accessing it from main memory.

c. Control over Hardware Features

✓ **Direct Hardware Access**

Low-level optimization allows developers to take advantage of specific hardware features such as SIMD (Single Instruction, Multiple Data) instructions, multi-core processing, and hardware-specific instructions that are not always accessible through high-level abstractions.

✓ **Efficient Use of CPU Registers**

Register allocation can be optimized to ensure that frequently used variables are stored in CPU registers, reducing the time spent on memory accesses.

d. Impact on Battery Life

✓ **Energy Efficiency**

In mobile and embedded systems, low-level optimizations can reduce power consumption. Efficient code often leads to lower CPU usage, which can extend battery life as a critical factor in portable devices.

- ✓ **Thermal Management**

  Reduced CPU load can help manage thermal output, improving the overall stability and lifespan of hardware.

e. **Scalability**

- ✓ **Handling Increased Load**

  As applications scale and handle larger datasets or more simultaneous users, low-level optimizations can help maintain performance levels. This is particularly important for server-side applications, where poor performance can lead to increased costs or decreased user satisfaction.

f. **Optimized Algorithms and Data Structures**

- ✓ **Algorithmic Efficiency**

  Low-level optimization encourages developers to choose and implement the most efficient algorithms and data structures for their specific use cases. This often involves understanding the underlying hardware to optimize algorithm performance.

- ✓ **Fine-Tuning Performance**

  Low-level techniques allow developers to fine-tune performance by implementing custom solutions that meet specific application needs, leading to better overall system performance.

g. **Critical Applications**

- ✓ **Real-Time Systems**

  In systems that require strict timing guarantees (e.g., automotive, aerospace, medical devices), low-level optimizations are vital for meeting deadlines and ensuring predictable behavior.

- ✓ **High-Performance Computing (HPC)**

  Applications in scientific computing, simulations, and large-scale data analysis often rely on low-leve

viii. Hardware Interfacing in C++

1. Overview

In C++, hardware interfacing refers to the process of communicating with physical hardware devices, such as sensors, actuators, and peripherals, through your C++ program. C++ is widely used for hardware interfacing because of its low-level access capabilities, portability, and performance. Interfacing with hardware typically involves sending and receiving data using various communication protocols, managing memory-mapped I/O, interacting with device drivers, and utilizing system calls or libraries designed for the target platform.

2. Components of Hardware Interfacing in C++:

a. Communication Protocols

Different devices use different protocols for communication. Some common protocols include:

- ✓ **GPIO (General Purpose Input/Output)**: Used to control or sense binary signals (high/low).
- ✓ **UART (Universal Asynchronous Receiver-Transmitter)**: A serial communication protocol used for data transmission between two devices.
- ✓ **I2C (Inter-Integrated Circuit)**: A two-wire protocol for communication between multiple devices over short distances.
- ✓ **SPI (Serial Peripheral Interface)**: A faster, four-wire communication protocol for short-distance data exchange between microcontrollers and peripherals.
- ✓ **USB (Universal Serial Bus)**: A widely used protocol for connecting peripherals like keyboards, printers, etc.

b. **Device Drivers**

C++ can interact with device drivers to access and control hardware. These drivers provide the necessary abstraction for the operating system to communicate with the hardware device.

c. **Memory-Mapped I/O**

In embedded systems, hardware registers are often mapped to specific memory addresses. C++ can directly access these addresses to control hardware. Using the volatile keyword is essential to ensure that the compiler doesn't optimize away hardware register accesses.

d. **Interrupt Handling**

Hardware devices often raise interrupts to signal events (e.g., a keypress or data availability). C++ programs can register interrupt handlers to respond to these events.

3. Methods for Hardware Interfacing in C++

a. Accessing Hardware Directly via Memory-Mapped I/O

In many embedded systems, hardware components (like registers) are mapped to specific memory addresses. C++ can access these addresses using pointers. You'll need to mark such addresses as volatile to prevent the compiler from optimizing away these accesses, which are crucial for hardware control.

```
#include <iostream>

// Assume hardware register at this memory location
```

```
volatile uint32_t* register_address = reinterpret_cast<volatile
uint32_t*>(0x40000000);

void writeRegister(uint32_t value) {

   *register_address = value;  // Write value to the hardware register

}

int main() {

   writeRegister(0x01);  // Write to the register

   std::cout << "Wrote to hardware register." << std::endl;

   return 0;

}
```

b. **Using System Calls and APIs**

Many hardware devices are accessed via system APIs or system calls provided by the operating system (such as POSIX APIs on Linux). This is often necessary when interacting with devices like serial ports, USB devices, or network interfaces.

Example: Interfacing a serial port using the termios API on Linux:

```
#include <fcntl.h>

#include <unistd.h>

#include <termios.h>

#include <iostream>

int main() {

   int serial_port = open("/dev/ttyS0", O_RDWR); // Open the serial port


   if (serial_port < 0) {

      std::cerr << "Error opening serial port" << std::endl;

      return -1;

   }

   struct termios tty;

   tcgetattr(serial_port, &tty);  // Get current serial port settings
```

```
    // Set baud rate, 8N1 (8 bits, no parity, 1 stop bit)
    cfsetispeed(&tty, B9600);  // Set input speed to 9600 baud
    cfsetospeed(&tty, B9600);  // Set output speed to 9600 baud
    tty.c_cflag &= ~PARENB;    // No parity
    tty.c_cflag &= ~CSTOPB;    // 1 stop bit
    tty.c_cflag &= ~CSIZE;     // Clear current data size setting
    tty.c_cflag |= CS8;        // 8 data bits
    tcsetattr(serial_port, TCSANOW, &tty);  // Apply the settings
    const char* message = "Hello, UART!";
    write(serial_port, message, sizeof(message));
    close(serial_port);  // Close the serial port
    return 0;
}
```

Example: Interfacing a Serial Port Using Windows API

Here's an example of how you can interface with a serial port on Windows using the Windows API

```
#include <windows.h>
#include <iostream>
int main() {
    // Open the serial port (COM1 in this case)
    HANDLE hSerial = CreateFile(
        "COM1",              // Port name
        GENERIC_READ | GENERIC_WRITE, // Read and write access
        0,                   // No sharing
        NULL,                // Default security attributes
        OPEN_EXISTING,       // Opens existing port only
        0,                   // Non-overlapped I/O
        NULL);               // Null for serial communications devices
```

```
    if (hSerial == INVALID_HANDLE_VALUE) {
        std::cerr << "Error opening serial port." << std::endl;
        return -1;
    }
    // Set up the serial port parameters (baud rate, data bits, stop bits, etc.)
    DCB dcbSerialParams = {0};
    dcbSerialParams.DCBlength = sizeof(dcbSerialParams);
    if (!GetCommState(hSerial, &dcbSerialParams)) {
        std::cerr << "Error getting serial port state." << std::endl;
        CloseHandle(hSerial);
        return -1;
    }

    // Set the baud rate, byte size, parity, and stop bits
    dcbSerialParams.BaudRate = CBR_9600;  // Baud rate = 9600
    dcbSerialParams.ByteSize = 8;         // 8 data bits
    dcbSerialParams.Parity = NOPARITY;    // No parity
    dcbSerialParams.StopBits = ONESTOPBIT; // 1 stop bit
    if (!SetCommState(hSerial, &dcbSerialParams)) {
        std::cerr << "Error setting serial port parameters." << std::endl;
        CloseHandle(hSerial);
        return -1;
    }
    // Set the timeouts for reading and writing
    COMMTIMEOUTS timeouts = {0};
    timeouts.ReadIntervalTimeout = 50;  // Maximum time between two bytes in
ms
    timeouts.ReadTotalTimeoutConstant = 50;  // Total read timeout in ms
```

```
    timeouts.ReadTotalTimeoutMultiplier = 10;  // Multiplier for calculating read timeout
    timeouts.WriteTotalTimeoutConstant = 50;  // Total write timeout in ms
    timeouts.WriteTotalTimeoutMultiplier = 10; // Multiplier for calculating write timeout
    if (!SetCommTimeouts(hSerial, &timeouts)) {
        std::cerr << "Error setting timeouts." << std::endl;
        CloseHandle(hSerial);
        return -1;
    }
    // Write data to the serial port
    const char* dataToSend = "Hello, Serial Port!";
    DWORD bytesWritten;
    if (!WriteFile(hSerial, dataToSend, strlen(dataToSend), &bytesWritten, NULL)) {
        std::cerr << "Error writing to serial port." << std::endl;
        CloseHandle(hSerial);
        return -1;
    }
    std::cout << "Data sent: " << dataToSend << std::endl;
    // Read data from the serial port
    char buffer[256];
    DWORD bytesRead;
    if (!ReadFile(hSerial, buffer, sizeof(buffer), &bytesRead, NULL)) {
        std::cerr << "Error reading from serial port." << std::endl;
        CloseHandle(hSerial);
        return -1;
    }
    if (bytesRead > 0) {
        std::cout << "Received: " << std::string(buffer, bytesRead) << std::endl;
```

```
    } else {
        std::cout << "No data received." << std::endl;
    }

    // Close the serial port
    CloseHandle(hSerial);
    return 0;
}
```

Explanation

Opening the Serial Port:

The CreateFile() function opens the serial port. In this example, "COM1" is the name of the serial port, and GENERIC_READ | GENERIC_WRITE specifies read/write access.

The function returns a handle (HANDLE hSerial) which is used for subsequent operations like reading and writing.

Configuring the Serial Port:

The DCB structure (Device Control Block) holds the serial port configuration parameters such as baud rate, data bits, parity, and stop bits.

GetCommState() retrieves the current configuration, and SetCommState() applies the new configuration.

Setting Timeouts:

The COMMTIMEOUTS structure is used to specify the timeouts for reading and writing. These timeouts help in avoiding the program hanging while waiting for data that might not arrive.

Writing Data:

The WriteFile() function sends data to the serial port. In this example, the string "Hello, Serial Port!" is written.

Reading Data:

The ReadFile() function reads data from the serial port into the buffer. If data is received, it is printed to the console.

Closing the Serial Port:

The CloseHandle() function is used to close the serial port after communication is complete.

Notes:

Port Name: On Windows, serial ports are named as COM1, COM2, etc. For ports numbered above 9, you need to specify the name in this format: \\.\COM10.

Baud Rate and Configuration: You can set different baud rates (e.g., CBR_9600, CBR_115200) and configurations (e.g., number of data bits, parity, stop bits) according to the device you are communicating with.

Error Handling: Always check the return values of system functions (like CreateFile(), SetCommState(), and WriteFile()) to ensure they succeed.

c. Using Libraries for Specific Hardware

Several libraries abstract the low-level details of interfacing with hardware. These libraries handle hardware interactions and provide easy-to-use functions in C++.

- ✓ **wiringPi** (for Raspberry Pi GPIO)
- ✓ **libi2c-dev (for I2C communication in Linux)**
- ✓ **libserialport (for serial communication)**

4. list of specific libraries and APIs for common hardware interfaces:

a. Serial Port (COM Port)

✓ **Library**: Windows API (WinAPI)

Description: Provides low-level access to serial ports for communication with devices like modems, microcontrollers, GPS modules, etc.

Key Functions: CreateFile(), ReadFile(), WriteFile(), SetCommState()

✓ **Alternative Libraries**:

Boost.Asio: Provides higher-level access to serial ports using asynchronous I/O.

CSerial: C++ class wrapper for serial communication.

b. **USB (Universal Serial Bus)**

✓ **Library:** Windows API (WinAPI), libusb

Description: For communicating with USB devices, like printers, cameras, and custom USB hardware.

Key Functions: DeviceIoControl(), SetupDiGetClassDevs()

✓ **Alternative Libraries:**

libusb: A cross-platform USB library for interfacing with USB devices without writing a custom driver.

WinUSB: Windows USB driver API that simplifies access to USB devices.

I2C (Inter-Integrated Circuit)

✓ **Library**: **Windows.Devices.I2c** (for UWP apps)

Description: Used to communicate with I2C devices like sensors, OLED displays, and real-time clocks.

Key Functions: Windows::Devices::I2c::I2cDevice

✓ **Alternative Libraries**:

libi2c: For cross-platform I2C communication.

Manufacturer-specific SDKs (e.g., for Raspberry Pi running Windows IoT).

c. **GPIO (General Purpose Input/Output)**

✓ **Library**: **Windows.Devices.Gpio** (for UWP apps)

Description: Provides access to GPIO pins, useful for controlling LEDs, reading button presses, or driving motors.

Key Functions: Windows::Devices::Gpio::GpioPin

✓ **Alternative Libraries**:

WiringPi for Windows: GPIO library for platforms like Raspberry Pi running Windows IoT.

d. **SPI (Serial Peripheral Interface)**

✓ **Library**: **Windows.Devices.Spi** (for UWP apps)

Description: Used for communication with devices like displays, sensors, and other SPI peripherals.

Key Functions: Windows::Devices::Spi::SpiDevice

✓ **Alternative Libraries**:

wiringPi: For SPI communication in embedded platforms like Raspberry Pi with Windows IoT Core.

libmpsse: A library for interfacing with SPI and I2C devices.

e. **Bluetooth**

✓ **Library**:

Windows.Devices.Bluetooth (for UWP apps)

WinSock (for Bluetooth sockets)

Description: Enables wireless communication with Bluetooth devices such as smartphones, headsets, and BLE (Bluetooth Low Energy) sensors.

Key Functions: Windows::Devices::Bluetooth::BluetoothDevice, Socket(), connect(), send()

- ✓ **Alternative Libraries**:

BlueZ (Linux): Not available on Windows, but helpful for cross-platform development.

InTheHand.Net.Bluetooth: A .NET Bluetooth library supporting Windows Bluetooth operations.

f. PCI (Peripheral Component Interconnect) Devices

- ✓ **Library**: Windows Driver Kit (WDK)

Description: For developing drivers and interfacing with PCI or PCIe hardware like network cards, graphics cards, or custom FPGA boards.

Key Functions: Kernel-mode driver APIs.

- ✓ **Alternative Libraries**:

libpci: Cross-platform PCI access library (Linux, BSD).

g. Network Interfaces (Ethernet, Wi-Fi)

- ✓ **Library**: **Windows Sockets (**WinSock)

Description: Provides access to network interfaces for Ethernet or Wi-Fi communication, supporting TCP/IP, UDP, and other network protocols.

Key Functions: socket(), bind(), connect(), recv(), send()

- ✓ **Alternative Libraries**:

Boost.Asio: Asynchronous I/O library for network communication.

POCO C++ Libraries: High-level networking APIs.

h. HID (Human Interface Devices)

- ✓ **Library**:

Windows API (hid.dll)

Windows Driver Kit (WDK) for writing custom drivers.

Description: Interfacing with HID devices such as keyboards, mice, game controllers, or custom HID peripherals.

Key Functions: HidD_GetHidGuid(), CreateFile(), ReadFile(), WriteFile()

- ✓ **Alternative Libraries**:

  hidapi: A cross-platform library for interfacing with HID devices.

  libhid: HID library for accessing and controlling USB HID devices.

i. Audio Devices (Microphones, Speakers)

- ✓ **Library**:

  MMDevice API for accessing audio devices.

  XAudio2 for high-performance audio.

  Description: Used to capture and render audio from microphones, speakers, or headsets.

  Key Functions: IAudioClient, IAudioCaptureClient, IAudioRenderClient

- ✓ **Alternative Libraries**:

  PortAudio: Cross-platform library for audio I/O.

  FMOD: High-performance audio library for games and applications.

  OpenAL: Cross-platform 3D audio API.

j. Camera Interfaces (Webcams)

- ✓ **Library**:

  Media Foundation API (for video capture and streaming).

  DirectShow (legacy API for audio/video capture).

  Description: Used to capture video streams from webcams and other video sources.

  Key Functions: IMFMediaSource, IMFSourceReader

- ✓ **Alternative Libraries**:

  OpenCV: Cross-platform computer vision and video capture library.

  libv4l: Linux-based, but useful for cross-platform development.

k. **Graphics and Display Interfaces**

- ✓ **Library**:

  DirectX (Direct3D) for high-performance graphics rendering.

  GDI (Graphics Device Interface) for simple 2D rendering and display control.

  Description: Used to render graphics on monitors, control display settings, or interface with display adapters.

Key Functions: IDirect3DDevice, ChangeDisplaySettings()

✓ **Alternative Libraries**:

SDL: Cross-platform library for graphics, input, and audio.

SFML: Simple and Fast Multimedia Library for 2D graphics and input.

l. **Sensors (e.g., Accelerometers, Gyroscopes)**

✓ **Library**: **Windows.Devices.Sensors** (for UWP apps)

Description: Used to access sensors such as accelerometers, gyroscopes, or proximity sensors on laptops, tablets, or embedded platforms.

Key Functions: Windows::Devices::Sensors::Accelerometer

✓ **Alternative Libraries**:

Sensor API (part of Windows API).

SensorKit: Third-party sensor library for accessing sensor data in different devices.

m. **Power Management and Battery Information**

✓ **Library**: Windows Power Management API (Powrprof.dll)

Description: For managing system power states and retrieving battery information.

Key Functions: CallNtPowerInformation(), GetSystemPowerStatus()

✓ **Alternative Libraries**:

BatteryInfoView: Third-party tool for advanced battery monitoring.

Example: GPIO control using wiringPi on Raspberry Pi:

```
#include <wiringPi.h>
#include <iostream>
int main() {
  // Initialize wiringPi
  if (wiringPiSetup() == -1) {
    std::cerr << "Failed to initialize wiringPi" << std::endl;
    return 1;
  }
  int ledPin = 0;  // GPIO Pin 0 (wiringPi numbering)
```

```
    pinMode(ledPin, OUTPUT);  // Set pin mode to output
    // Blink LED 10 times
    for (int i = 0; i < 10; ++i) {
        digitalWrite(ledPin, HIGH); // Turn LED on
        delay(500);              // Wait for 500 ms
        digitalWrite(ledPin, LOW);  // Turn LED off
        delay(500);              // Wait for 500 ms
    }
    return 0;
}
```

7. **Interfacing via SPI and I2C**

   C++ can also interact with SPI or I2C devices using standard Linux device interfaces or specific libraries like libi2c-dev or spidev.

   Example: Communicating with an I2C device (like a temperature sensor):

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
int main() {
    const char *i2c_device = "/dev/i2c-1";  // I2C bus
    int i2c_addr = 0x48;                // I2C address of sensor
    int i2c_fd = open(i2c_device, O_RDWR);  // Open I2C device
    if (i2c_fd < 0) {
        std::cerr << "Failed to open the I2C bus" << std::endl;
        return -1;
    }
    // Set the I2C address of the slave device
    if (ioctl(i2c_fd, I2C_SLAVE, i2c_addr) < 0) {
```

```
        std::cerr << "Failed to acquire bus access or talk to slave" << std::endl;
        return -1;
    }

    // Read 2 bytes of data from the sensor
    char buf[2];
    if (read(i2c_fd, buf, 2) != 2) {
        std::cerr << "Failed to read from the I2C device" << std::endl;
        return -1;
    }
    // Process and convert the data (e.g., temperature)
    int temp = (buf[0] << 8 | buf[1]) >> 4;
    float temperature = temp * 0.0625;  // Convert to Celsius
    std::cout << "Temperature: " << temperature << " °C" << std::endl;
    close(i2c_fd);  // Close the I2C bus
    return 0;
}
```

8. **Writing Device Drivers in C++**

   In some cases, C++ can be used to write custom device drivers for a hardware component. Device drivers are a specialized type of program that handles communication between the operating system and hardware devices.

   Writing drivers involves:

   Interfacing with hardware registers.

   Managing interrupts and direct memory access (DMA).

   Exposing an API for the rest of the system to use.

9. **Key Considerations for Hardware Interfacing in C++**

- ✓ **Real-Time Constraints**: Embedded systems often have strict timing requirements, meaning that your C++ code needs to run with minimal latency.
- ✓ **Memory Management**: Careful memory management is crucial, especially in embedded systems where resources are limited.

- ✓ **Concurrency:** Many hardware systems run concurrently with software, which may require handling interrupts or using multithreading for proper synchronization.
- ✓ **Platform-Specific Code**: Hardware interfacing is often platform-dependent. Code that works on a Linux system may not work on a microcontroller or Windows system without modification.

10. I/O ports

a. Overview

I/O Ports (Input/Output Ports) refer to the interfaces through which a computer communicates with external devices (like keyboards, mice, printers, sensors, etc.). These ports are essential for performing basic hardware interfacing tasks, allowing the processor to send data to or receive data from peripheral devices.

b. categories of I/O ports

In computing systems, I/O ports can be divided into two categories:

- ✓ **Memory-Mapped I/O (MMIO)**

In Memory-Mapped I/O, device registers are mapped into the same address space as the system's main memory. The CPU uses the same instructions (like read and write) to interact with both memory and I/O devices.

Advantage: Simpler, unified address space.

Example: Video memory on a graphics card may be mapped directly into the address space.

- ✓ **Port-Mapped I/O (PMIO)**

In Port-Mapped I/O, a special address space is dedicated exclusively for communication with I/O devices. Separate instructions (like IN and OUT in x86 assembly) are used to communicate with the I/O devices.

Advantage: It separates the address space of the I/O devices from the system memory.

Example: Reading from a parallel or serial port using a specific I/O port address.

Examples of I/O Port

- ✓ **Serial Ports (COM Ports)**

Commonly used for RS-232 communication with devices like modems, GPS receivers, and microcontrollers. Communication involves reading/writing data bytes through specific port addresses using UART (Universal Asynchronous Receiver-Transmitter).

Example I/O Port Address: 0x3F8 (for COM1).

- ✓ **Parallel Ports (LPT Ports)**

  These are used for older printers and data acquisition devices. Communication involves sending data through 8-bit parallel data lines.

  Example I/O Port Address: 0x378 (for LPT1).

- ✓ **USB Ports**

  Universal Serial Bus (USB) is a more modern interface that abstracts much of the I/O port management, using device drivers. USB devices use the USB protocol and handle complex device addressing and communication over the USB bus.

- ✓ **GPIO (General Purpose Input/Output) Ports**

  GPIO pins are used on embedded systems (like Raspberry Pi) to interface with sensors, buttons, LEDs, and other simple devices. Each GPIO pin can be configured as an input (to read data) or an output (to send signals).

- ✓ **Network Interface Ports (Ethernet/Wi-Fi)**

  Network interfaces communicate via well-defined ports such as port 80 for HTTP or port 443 for HTTPS. Data is transmitted through specific communication channels (sockets) associated with these ports.

c. I/O Ports in Hardware Interfacing

To interface with hardware through I/O ports in Windows or other operating systems, there are specific libraries and APIs for accessing and controlling I/O ports.

- ✓ **Libraries/APIs for I/O Ports in Windows**

  Windows API for Serial and Parallel Ports:

  Use **CreateFile(**), **ReadFile(),** and **WriteFile()** for high-level access to serial and parallel ports in Windows. Example: Interfacing with COM ports, LPT ports, or USB virtual COM ports.

  Windows Driver Kit (WDK):

  It is for creating kernel-mode drivers that access I/O ports and other hardware devices. It is also necessary for accessing low-level hardware, such as writing PCI or USB drivers.

  WinSock API:

  For communication over network ports (TCP/UDP sockets). Provides functions like **socket(), connect(), send(), and recv()** for interfacing with networking hardware through software ports.

libusb:

it is for interacting with USB devices, offering cross-platform support for communicating with USB hardware.

Direct I/O (for low-level port access on Windows):

Some third-party libraries like **InpOut32** or **WinIo** offer access to I/O ports on Windows systems without needing a kernel-mode driver. **InpOut32** is used to access parallel ports (LPT) or general I/O ports, particularly for older legacy systems.

d. Accessing I/O Ports in C++

To interface directly with I/O ports in Windows, you generally need to use either **Windows API** functions or kernel-level programming because user-mode applications are not allowed direct access to hardware I/O ports for security and stability reasons.

Example 1: Accessing I/O Ports in Linux (C/C++)

In Linux, you can access I/O ports directly using system calls like inb() and outb(). This example shows how you might read from and write to an I/O port:

```
#include <stdio.h>

#include <sys/io.h>  // for inb() and outb()

#define PORT 0x3F8  // COM1 port address


int main() {
   // Give access to the I/O ports
   if (ioperm(PORT, 1, 1)) {
      perror("ioperm");
      return 1;
   }
   // Write to the port
   outb(0xFF, PORT);  // Send a byte to the port
   // Read from the port
   int byte = inb(PORT);
   printf("Received byte: 0x%x\n", byte);
```

```
    // Remove access to the I/O ports
    if (ioperm(PORT, 1, 0)) {
        perror("ioperm");
        return 1;
    }
    return 0;
}
```

This example is for Linux and won't work on Windows directly because user-mode applications in Windows are restricted from accessing I/O ports directly. Windows requires using **drivers** to access hardware ports.

Example 2: Windows Accessing I/O Ports (Legacy)

Direct access to I/O ports in Windows is restricted for user-mode applications due to security and stability concerns. Instead, interacting with hardware such as serial or parallel ports is typically done through higher-level APIs like the Windows API (CreateFile(), ReadFile(), and WriteFile() functions).

However, if you want to access I/O ports (e.g., serial or parallel ports) directly, you would typically need a kernel-mode driver. For educational or legacy systems, libraries like **InpOut32** or **WinIo** can be used to access ports in Windows without writing a driver.

Here's a C++ program using **InpOut32**, which allows you to read from and write to I/O ports on Windows.

```
#include <iostream>
#include <windows.h>
#include "inpout32.h"  // Make sure InpOut32.dll and inpout32.lib are available
int main() {
    // Load the InpOut32 DLL
    HINSTANCE hLib = LoadLibrary(TEXT("InpOut32.dll"));
    if (!hLib) {
        std::cerr << "Unable to load InpOut32.dll!" << std::endl;
        return 1;
    }
```

```
    // Function pointers to access InpOut32 functions
    typedef void (__stdcall *Out32Func)(short portAddress, short data);
    typedef short (__stdcall *Inp32Func)(short portAddress);
    Out32Func Out32 = (Out32Func)GetProcAddress(hLib, "Out32");
    Inp32Func Inp32 = (Inp32Func)GetProcAddress(hLib, "Inp32");
    if (!Out32 || !Inp32) {
        std::cerr << "Unable to get function addresses!" << std::endl;
        return 1;
    }
    // Define the port address (e.g., 0x378 for LPT1 - parallel port)
    short portAddress = 0x378;
    // Write data to the port
    short dataToWrite = 0xFF;  // Send all high signals to the port
    Out32(portAddress, dataToWrite);
    std::cout << "Written 0xFF to port 0x378." << std::endl;
    // Read data from the port
    short dataRead = Inp32(portAddress);
    std::cout << "Read from port 0x378: " << std::hex << dataRead << std::endl;
    // Free the DLL
    FreeLibrary(hLib);
    return 0;
}
```

Explanation:

Load InpOut32.dll: The program dynamically loads the InpOut32.dll at runtime.

GetProcAddress: Retrieves the addresses of the Out32 and Inp32 functions for writing to and reading from the I/O port.

Write Data: Out32(portAddress, data) sends data to the specified port.

Read Data: Inp32(portAddress) reads data from the specified port.

Port Address: The parallel port (LPT1) typically uses the address 0x378. You can modify this for other ports like 0x3F8 (COM1).

✓ **Access to Processor Registers**

Accessing processor registers directly in C++ is generally not possible in user-mode applications due to operating system restrictions. However, you can access processor registers in kernel-mode programming or by using inline assembly. Here's how you can do this in different contexts.

✓ **Accessing Processor Registers Using Inline Assembly (GCC)**

If you're using a compiler that supports inline assembly (like GCC), you can access registers directly within your C++ code. Note that inline assembly is not supported by MSVC.

Example: Inline Assembly in GCC

```
#include <iostream>

int main() {
    unsigned int eaxValue;
    // Inline assembly to read the EAX register
    __asm__ (
        "mov %%eax, %0"  // Move the value of EAX into eaxValue
        : "=r" (eaxValue) // Output operand
    );
    std::cout << "EAX Value: " << eaxValue << std::endl;
    return 0;
}
```

✓ **Accessing Registers in User Mode (Limited)**

In user-mode applications, you can access register information using system calls or libraries. For instance, in Windows, you can use the **GetThreadContext** and **SetThreadContext** functions to manipulate thread registers.

Example: Using GetThreadContext in Windows

```
#include <windows.h>
#include <iostream>
int main() {
```

```
    HANDLE hThread = GetCurrentThread();
    CONTEXT context;
    // Initialize the CONTEXT structure
    context.ContextFlags = CONTEXT_CONTROL;
    // Get the current thread's context
    if (GetThreadContext(hThread, &context)) {
        std::cout << "EAX: " << context.Eax << std::endl;
        std::cout << "EBX: " << context.Ebx << std::endl;
        std::cout << "ECX: " << context.Ecx << std::endl;
        std::cout << "EDX: " << context.Edx << std::endl;
    } else {
        std::cerr << "Failed to get thread context." << std::endl;
    }

    return 0;
}
```

Practical Activity 3.4.2: Hardware interfacing

Task:

1: read and perform the task below:

You are a firmware developer and you requsted toedevelop and execute a C++ program that shows how you might read from and write to an I/O port in windows using library in Dev-C++.

2: Read steps involved in applying nline assembly in C++ program from Key readings **3.4.2.**

3: Explore steps explained by trainer to initiate the process of applying inline assembly in C++ program.

4: Compile and run C++ program by following demonstrated steps and ask for assistance if needed.

5: Verify the output whether it is the same as the one expected.

Key readings 3.4.2

Hardware interfacing: **Read data from and write data to I/O port**

Executing a C++ program that shows how you might read from and write to an I/O port in windows using InpOut32 in Dev-C++ involves several steps, including setting up your project and linking the necessary libraries.

Here's a detailed guide to help you through the process:

Step 1: Download InpOut32

Download **InpOut32** from the official site: Highrez.co.uk - InpOut32 Download.

Extract the downloaded zip file, which should contain:

InpOut32.dll

inpout32.lib

Step 2: Set Up Dev-C++

Install Dev-C++: Make sure you have Dev-C++ installed on your system. If not, download it from Dev-C++ SourceForge.

Create a New Project:

Open Dev-C++.

Go to File > New > Project.

Select Console Application, choose C++, and give your project a name (e.g., IOLibExample).

Click OK, and choose a location to save your project.

Step 3: Add Source Code

After creating the project, a new source file (main.cpp) will be opened automatically. If not, create a new source file by going to File > New > Source File.

Copy and paste the following C++ code into the source file:

#include <iostream>

```
#include <windows.h>
#include "inpout32.h"  // Make sure InpOut32.dll and inpout32.lib are available
int main() {
   // Load the InpOut32 DLL
   HINSTANCE hLib = LoadLibrary(TEXT("InpOut32.dll"));
   if (!hLib) {
      std::cerr << "Unable to load InpOut32.dll!" << std::endl;
      return 1;
   }
   // Function pointers to access InpOut32 functions
   typedef void (__stdcall *Out32Func)(short portAddress, short data);
   typedef short (__stdcall *Inp32Func)(short portAddress);
   Out32Func Out32 = (Out32Func)GetProcAddress(hLib, "Out32");
   Inp32Func Inp32 = (Inp32Func)GetProcAddress(hLib, "Inp32");
   if (!Out32 || !Inp32) {
      std::cerr << "Unable to get function addresses!" << std::endl;
      return 1;
   }
   // Define the port address (e.g., 0x378 for LPT1 - parallel port)
   short portAddress = 0x378;

   // Write data to the port
   short dataToWrite = 0xFF;  // Send all high signals to the port
   Out32(portAddress, dataToWrite);
   std::cout << "Written 0xFF to port 0x378." << std::endl;
   // Read data from the port
   short dataRead = Inp32(portAddress);
   std::cout << "Read from port 0x378: " << std::hex << dataRead << std::endl;
```

```
    // Free the DLL
    FreeLibrary(hLib);
    return 0;
}
```

Step 4: Link the InpOut32 Library

Go to Tools > Compiler Options.

In the Linker tab, find the Add the following commands when linking section.

Add the line: **inpout32.lib**

Click OK to save the settings.

Step 5: Include the InpOut32 DLL

Copy the InpOut32.dll file to the folder where your Dev-C++ executable is created (typically in the bin or Debug folder within your project directory).

Alternatively, you can place InpOut32.dll in the C:\Windows\System32\ directory if you want it to be accessible from anywhere.

Step 6: Compile and Run

Go to Execute > Compile & Run (or press F9).

If everything is set up correctly, the program should compile without errors and execute.

You should see output indicating the data written to and read from the I/O port.

Points to Remember

Hardware interfacing in C++ means using the language to control and communicate with hardware components like sensors, motors, and ports. This can be done by directly accessing the hardware's memory, using drivers, or through APIs (Application Programming **Port Access:** Interfaces).

This involves:

- **Low-level Memory Access**: Using pointers to directly access hardware registers to control or read data.
- **Device Drivers and APIs:** C++ can work with operating system tools like Windows or Linux to make communication with hardware easier.

- **Communication Protocols:** C++ can handle serial communication (like UART, SPI, I2C) using libraries such as Boost.
- **Interrupt Handling: C**++ manages hardware interrupts efficiently to control hardware in real time.
- **Timing:** For precise timing, C++ uses tools like std::chrono or system functions.
- **RTOS:** In real-time systems, C++ works with real-time operating systems to ensure the hardware runs on time.
- Peripheral Devices: C++ sends signals to devices like sensors and displays or reads their data.
- Embedded Systems: In small systems like Arduino or Raspberry Pi, C++ communicates with hardware through pins and modules like Bluetooth or Wi-Fi.

To interact with hardware ports, C++ uses specific libraries or system commands (e.g., /dev/ttyS0 for Linux serial ports).

An example would be using memory-mapped I/O to directly control hardware registers.

- Steps to Execute a C++ Program for I/O Port Access Using InpOut32

  - ✓ Download InpOut32 Library:
  - ✓ Set Up Your Development Environment:
  - ✓ Add InpOut32 to Your Project:
  - ✓ Include the InpOut32 Header:
  - ✓ Write the C++ Code
  - ✓ Configure Project Settings
  - ✓ Compile the Program
  - ✓ Run the Program:
  - ✓ Observe the Output:

Application of learning 3.4.:

Scenario: **Real-Time Image Processing for Autonomous Drones**

Background:

You work for ABCCo Ltd, a company that makes drones used for farming. These drones fly over fields; taking high-quality pictures to check the health of crops, look for pests, and measure how plants have grown. The drones need to analyze these pictures quickly because any delays could affect their ability to adjust their flight paths and help farmers.

Problem:

The drone's processor is not very powerful, and it struggles to process the pictures fast enough. Your job is to speed up an important part of the image analysis, called "edge detection," which helps the drone identify the outlines of objects in the pictures (like crops or pests). The original code for edge detection is written in C++, but it's too slow. To make it faster, you need to use a programming technique called "inline assembly." This allows you to directly control the hardware of the drone's processor, making the image processing quicker and more efficient

Task:

Improve the part of the code that detects edges in the images. Rewrite a section of the C++ code using inline assembly. This will allow you to give the processor direct instructions to work faster. By doing this, you will help the drone process images more quickly, improving its ability to adjust its flight and assist farmers in monitoring their crops more efficiently

Learning outcome 3 end assessment

Theoretical assessment

I. Referring from the contents about CPU optimization in C++ focusing on pointers, file handling, multithreading, concurrency, and inline assembly, answer the following questions by **True** or **False**:

1. Using pointers in C++ can improve performance because it allows direct memory access.
2. Dereferencing a null pointer in C++ is a common optimization technique.
3. Reading from large files using a single large buffer is generally more CPU-efficient than reading the file in small chunks.
4. Multithreading always results in faster program execution due to parallelism.
5. Mutexes are used to prevent race conditions in multithreaded C++ programs, but they can also negatively impact CPU performance if overused.
6. Inline assembly can be used in C++ to optimize critical performance areas, but it may make the code less portable.
7. Using too many threads in a CPU-bound program can lead to diminishing returns or even performance degradation.
8. Moving frequently accessed data to the CPU's cache (cache locality) is unrelated to pointer optimization.
9. File handling optimizations such as memory-mapped files (mmap) can reduce CPU load in I/O-bound applications.
10. In C++, atomics can be used for lock-free programming, reducing the need for locks and improving CPU performance in some concurrent applications.

II. Choose the best answer by circling the letter that fits:

1. What is a key advantage of using pointers in C++ for CPU optimization?
   a) Easier code readability
   b) Direct memory access
   c) Avoiding memory leaks
   d) Guaranteed portability

2. Which of the following file handling methods is typically more CPU-efficient for reading large files?
   a) Reading the file byte by byte
   b) Reading the file using a small buffer
   c) Using memory-mapped file (mmap)
   d) Opening and closing the file multiple times during read operations

3. What is a potential downside of excessive multithreading in a CPU-bound program?
   a) Improved memory usage

b) Diminishing returns due to context switching overhead
c) Increased disk usage
d) Reduced memory allocation speed

4. Which of the following is a technique to prevent race conditions in C++ multithreaded programs?
   a) Using global variables
   b) Implementing mutex locks
   c) Ignoring shared data access
   d) Avoiding the use of pointers

5. How does cache locality improve CPU performance in C++ programs?
   a) By reducing memory allocation times
   b) By ensuring frequently used data is stored close to the CPU
   c) By increasing the file read speed
   d) By reducing the number of active threads

6. What is one of the trade-offs of using inline assembly in C++ for CPU optimization?
   a) Improved portability
   b) Increased debugging complexity
   c) Easier cross-platform support
   d) Simplified code maintenance

7. Which of the following approaches is commonly used to improve CPU performance when handling large amounts of file data?

   a) Opening multiple file streams
   b) Reading small chunks of data sequentially
   c) Loading the entire file into memory at once (if feasible)
   d) Using low-level assembly instructions to read the file

8. What is the primary benefit of using atomic operations in multithreaded C++ programs?
   a) Eliminating the need for memory allocation
   b) Avoiding context switching
   c) Achieving lock-free synchronization
   d) Increasing memory usage

9. Which of the following optimizations can help reduce the overhead of mutexes in a multithreaded C++ program?
   a) Increasing the number of threads
   b) Reducing the frequency of locking and unlocking
   c) Using pointers instead of variables

d) Avoiding memory allocation within threads

10. Why would using inline functions and inline assembly in C++ result in faster execution in certain scenarios?
    a) It ensures the code is compiled into a separate binary file
    b) It reduces the overhead of function calls and provides finer control over CPU instructions
    c) It reduces memory usage
    d) It increases the number of function calls for better performance

III. Match the term with the correct description regarding CPU optimization by completing the answer column of the table bellow referring to the given example.

Answers	**Terms**	**Descriptions**
…….	1. Pointers	A. A synchronization primitive that ensures only one thread accesses a resource at a time, reducing race conditions.
…….	2. Memory-mapped files (mmap)	B. Embedding low-level CPU instructions in the C++ code to optimize performance in critical sections.
……..	3. Mutex	C. Organizing data so that it is accessed from the CPU's cache, reducing memory access time.
……..	4. Cache locality	D. Operations that ensure safe manipulation of shared data without the need for locks.
…….	5. Inline assembly	E. A lock mechanism that keeps the CPU busy waiting for a resource, typically used in real-time systems.
……..	6. Atomic operations	F. A collection of reusable threads that can execute tasks to reduce the overhead of thread creation.
……	7. Race condition	G. A technique to map file contents into memory for efficient I/O operations.
……..	8. Context switching	H. The CPU's process of switching between threads, which can lead to performance overhead.

……	9. Thread pool	I. Directly allows access to physical memory for faster data manipulation
…….	10. Spinlock	J. A situation where multiple threads access shared data unsafely, causing unpredictable behavior
…L….(Example)	11. I/O ports	K. It is the techniques where special CPU instructions are used to communicate with devices at specific port addresses.
		L. They enable data transfer between CPU and external devices for input or output operations

IV. Complete sentences that follow using one of the words: **data, mmap, race, inline assembly, pointers, context switching, mutexes, cache, thread, corruption.**

1. Accessing memory directly in C++ for optimization can be done using _____.
2. To prevent race conditions in multithreaded programs, C++ developers often use _____.
3. For efficient file handling in C++, large files can be accessed using _____.
4. Switching between threads incurs overhead due to _____.
5. Atomic operations in C++ allow lock-free manipulation of _____.
6. Critical performance sections in C++ can be optimized by using _____.
7. Data stored and accessed sequentially improves _____ locality, reducing CPU cache misses.
8. A _____ pool is often used to manage and reuse threads efficiently.
9. When multiple threads access shared resources without synchronization, _____ conditions can occur.
10. In multithreaded programs, threads must be synchronized to avoid _____ of shared data.

Practical assessment

Integrated situation:

Optimizing Firmware for an Embedded System in a Real-Time Automotive Engine Control Unit (ECU)

You are a firmware developer working on an Engine Control Unit (ECU) for a real-time automotive application.

The ECU collects data from multiple sensors, such as the throttle position, air intake, and crankshaft speed. Based on this data, it processes the information to control actuators like fuel injectors, ignition timing, and exhaust systems in real-time to ensure optimal engine performance. The system operates under strict timing constraints, requiring that sensor data be processed within microseconds, as delays could lead to engine misfires or performance degradation.

The ECU runs in a resource-constrained environment with limited CPU power, memory, and I/O resources. To meet the stringent real-time requirements, you are tasked with optimizing the ECU's firmware using advanced CPU optimization techniques in C++, focusing on pointers, file handling (memory and flash storage operations), multithreading and concurrency, and inline assembly to ensure that the system operates efficiently and reliably.

END

References

Bjorner, N. (2016). Optimizing C++ code for embedded systems. IEEE Transactions on Computers, 65(2), 338-349. https://doi.org/10.1109/TC.2015.2459671

Breshears, C. (2009). The art of concurrency: A thread monkey's guide to writing parallel applications. O'Reilly Media.

Butenhof, D. R. (1997). Programming with POSIX threads. Addison-Wesley.

Downey, A. B. (2019). The performance of pointers in C++: A guide for embedded systems developers. Embedded Systems Journal, 45(3), 123-145. https://doi.org/10.1109/ESJ.2019.024332

Fraser, C. W., & Hanson, D. R. (1995). A retargetable C compiler: Design and implementation. Addison-Wesley.

Grama, A., Gupta, A., & Karypis, G. (2003). Introduction to parallel computing (2nd ed.). Addison-Wesley.

Hill, M. D., & Jouppi, N. P. (2007). Computer architecture: A quantitative approach. Elsevier.

Josuttis, N. (2012). The C++ standard library: A tutorial and reference (2nd ed.). Addison-Wesley.

Klemens, B. (2012). 21st century C: C tips from the new school. O'Reilly Media.

Kleppmann, M. (2017). Designing data-intensive applications. O'Reilly Media.

Meyers, S. (2014). Effective modern C++: 42 specific ways to improve your use of C++11 and C++14. O'Reilly Media.

Molloy, A. (2019). Embedded C++ development for automotive systems. Springer.

O'Neil, T., & Ballman, D. (2019). Inline assembly in C++: Optimizing firmware performance in resource-constrained systems. Embedded Computing Design, 37(1), 47-59. https://doi.org/10.1109/ECD.2019.120058

Pohl, I. (2011). C++ for engineers and scientists (3rd ed.). Pearson Education.

Schmit, H. (2014). Multithreaded programming in real-time embedded systems. Journal of Real-Time Systems Engineering, 22(3), 105-119. https://doi.org/10.1109/JRSE.2014.062810

Shah, A., & Moudgalya, K. M. (2019). Understanding real-time systems with C++. IEEE Embedded Systems, 36(1), 20-30. https://doi.org/10.1109/EMSYS.2019.01230

Starke, M. (2018). High-performance computing and embedded systems. Springer.

Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley.

Sutter, H., & Larus, J. (2005). Software and the concurrency revolution. ACM Queue, 3(7), 54-62. https://doi.org/10.1145/1095408.1095416

Williams, A. (2012). C++ concurrency in action: Practical multithreading. Manning Publications.

RTB | RWANDA TVET BOARD

October, 2024