

**A PERSONAL PROJECT REPORT**

*on*

**FINANCE MANAGER**

*by*

**Chris Jordan J**

**USN: 1NH21CS062, Sem-Sec: 5-A**

**BACHELOR OF ENGINEERING UNDERGRADUATE**

*in*

**COMPUTER SCIENCE AND ENGINEERING**

*at*

**NEW HORIZON COLLEGE OF ENGINEERING,  
Marathahalli, Bangalore- 560 013**

---

## ABSTRACT

The "Finance Manager" project is a comprehensive C++ application designed to aid users in effectively managing their finances. With a suite of functionalities, it offers a robust platform for expense tracking, budgeting, income monitoring, and financial reporting. Leveraging various features of the C++ Standard Library, including arrays, vectors, lists, and STL algorithms, the project ensures efficient data storage, manipulation, and file I/O operations.

The expense tracking feature allows users to seamlessly record, view, and save their expenses, providing a convenient means to monitor spending habits. Through budgeting capabilities, users can establish financial goals and analyze their expenditure patterns to ensure prudent money management. Moreover, the income tracking functionality enables users to track diverse income streams from different sources, factoring in tax deductions, and save income reports for future reference.

Central to the project is the generation of comprehensive financial reports, facilitating the entry and storage of monthly financial data for thorough analysis and planning. By empowering users with tools to monitor and control their financial aspects, the "Finance Manager" project aims to contribute positively to individuals' financial well-being. With its user-friendly interface and robust functionalities, it serves as a valuable asset in navigating the complexities of personal finance management, ultimately promoting financial stability and prosperity.

---

# CONTENTS

<b>ABSTRACT</b>	<b>I</b>
<b>LIST OF FIGURES</b>	<b>IV</b>
<b>1. INTRODUCTION</b>	<b>1</b>
1.1. PROBLEM DEFINITION	2
1.2. OBJECTIVES	2
1.3. METHODOLOGY	3
1.4. EXPECTED OUTCOMES	4
1.5. HARDWARE AND SOFTWARE REQUIREMENTS	4
<b>2. FUNDAMENTALS OF C++</b>	
2.1. INTRO TO C++	5
2.2. AVANTAGES OF C++	6
2.3. DATA TYPES	8
2.4. CONTROL FLOW	9
2.5. FUNCTIONS	11
2.6. OBJECT ORIENTED PROGRAMMING	11
2.7. FILE HANDLING	13
2.8. IMPORT	15
<b>3. C++ STL GUIDE</b>	
3.1. INTRO TO STL	16
3.2. STL CONTAINERS	16
3.3. STL ALGORITHMS	18
3.4. STL ITERATORS	20
3.5. STL FUNCTIONS (FUNCTORS)	21

---

<b>4. DESIGN AND ARCHITECTURE</b>	<b>22</b>
<b>5. IMPLEMENTATION</b>	
5.1. PREPROCESSOR DIRECTIVES AND INITIALIZATIONS	23
5.2. FUNCTION TO DISPLAY TOTAL EXPENSES	24
5.3. FUNCTION TO TRACK EXPENSES	24
5.4. FUNCTION TO SAVE EXPENSES TO TXT FILE	26
5.5. FUNCTION FOR BUDGETING	26
5.6. FUNCTION TO TRACK INCOME	28
5.7. FUNCTION TO SAVE INCOME TO TXT FILE	29
5.8. FUNCTION TO GENERATE FINANCIAL REPORT	30
5.9. FUNCTION TO SAVE REPORT TO TXT FILE	32
5.10. MAIN FUNCTION	33
<b>6. RESULTS</b>	
6.1. EXPENSE TRACKER EXECUTION	36
6.2. EXPENSES SAVED TO TXT FILE	36
6.3. BUDGETING	37
6.4. INCOME TRACKER EXECUTION	37
6.5. INCOME SAVED TO TXT FILE	37
6.6. FINANCIAL REPORT EXECUTION	38
6.7. FINANCIAL REPORT SAVED TO TXT FILE	38
<b>7. CONCLUSION</b>	<b>39</b>
7.1. REFERENCES	40

---

---

## LIST OF FIGURES

FIG NO.	FIG DESCRIPTION	PAGE NO.
<b>1</b>	Data Types in C++	<b>8</b>
<b>2</b>	Flowchart of if/ if-else	<b>9</b>
<b>3</b>	Looping Constructs (for and while)	<b>10</b>
<b>4</b>	Break and Continue in C++	<b>10</b>
<b>5</b>	Class and Objects	<b>12</b>
<b>6</b>	Encapsulation in C++	<b>12</b>
<b>7</b>	Types of Inheritances in C++	<b>12</b>
<b>8</b>	Polymorphism (many forms)	<b>13</b>
<b>9</b>	C++ Abstraction	<b>13</b>
<b>10</b>	Code snippet of File streams	<b>14</b>
<b>11</b>	Code snippet of File Reader	<b>14</b>
<b>12</b>	Code snippet of File Writer	<b>15</b>
<b>13</b>	Code snippet for Error Handling	<b>15</b>
<b>14</b>	Code snippet of Binary File I/O	<b>15</b>
<b>15</b>	STL Containers Comprehensive Flow Chart	<b>18</b>
<b>16</b>	STL Pre- def Algorithms implementation	<b>19</b>
<b>17</b>	Output of STL Algorithms implementation	<b>19</b>
<b>18</b>	STL Iterators layered view	<b>20</b>
<b>19</b>	STL Iterators Operation	<b>21</b>

## CHAPTER 1

### 1. INTRODUCTION

In today's fast-paced world, managing personal finances has become increasingly challenging, with individuals juggling multiple income sources, expenses, and financial goals. Recognizing the importance of financial stability and the need for effective money management tools, the "Finance Manager" project emerges as a versatile solution crafted in C++ to address these challenges comprehensively. The "Finance Manager" project embodies a user-centric approach, aiming to simplify the complexities of financial management for individuals from all walks of life. With its intuitive interface and robust functionalities, the project offers a holistic suite of features designed to streamline expense tracking, budgeting, income monitoring, and financial reporting. At the heart of the project lies the essential functionality of expense tracking. Users can effortlessly record their daily expenditures, categorize expenses, and analyze spending patterns to gain insights into their financial habits. By providing a clear overview of where money is being spent, this feature empowers users to make informed decisions and exercise better control over their finances.

Moreover, the project offers powerful budgeting capabilities, enabling users to set financial goals and allocate resources effectively. Whether it's managing monthly expenses, saving for a vacation, or planning for retirement, users can create personalized budgets for various categories and track their progress towards financial objectives. In addition to expense tracking and budgeting, the project facilitates income monitoring, allowing users to track earnings from different sources and assess their overall financial inflow. By factoring in tax deductions and other expenses. Furthermore, the project empowers users to generate comprehensive financial reports, providing a detailed overview of their financial status.

In essence, the "Finance Manager" project stands as a testament to the power of technology in empowering individuals to take control of their financial futures. With its user-friendly interface and robust features, it aims to democratize financial management and pave the way for a more secure and prosperous tomorrow.

## **1.1. PROBLEM DEFINITION**

Many individuals struggle with managing their personal finances due to a lack of efficient tools and processes. Manual methods of tracking expenses, budgeting, and monitoring income prove time-consuming and prone to errors, leading to disorganized finances and missed opportunities for financial optimization. Existing financial management software often lacks user-friendliness and fails to cater to the diverse needs of users.

To address these challenges, there is a pressing need for a user-friendly and comprehensive financial management solution. This solution should leverage modern technology, such as C++ programming language and standard libraries, to streamline processes like expense tracking, budgeting, income monitoring, and financial reporting. By providing intuitive functionalities and robust data management capabilities, the solution aims to empower individuals to take control of their finances, make informed decisions, and achieve long-term financial stability and prosperity. Thus, the problem statement revolves around developing an efficient C++ application that simplifies financial management and enhances user experience to promote better financial outcomes for individuals.

## **1.2. OBJECTIVES**

1. Develop a user-friendly C++ application that simplifies financial management processes, including expense tracking, budgeting, income monitoring, and financial reporting.
2. Implement intuitive functionalities for recording and categorizing expenses, enabling users to easily track their spending habits and identify areas for optimization.
3. Design robust budgeting capabilities that allow users to set financial goals, allocate resources effectively, and monitor progress towards achieving budget targets.
4. Integrate features for tracking various income sources, accounting for tax deductions, and generating reports to provide users with a comprehensive overview of their financial inflow.

5. Utilize C++ standard libraries and modern programming techniques to ensure efficient data storage, manipulation, and file I/O operations, enhancing the application's performance and scalability.
6. Conduct thorough testing to identify and address any bugs or issues, ensuring the reliability and stability of the application for users.
7. Ultimately, aim to empower individuals to take control of their finances, improve their financial well-being, and achieve long-term financial stability and prosperity through effective money management.

### 1.3. METHODOLOGY

**Requirement Analysis:** Conduct a comprehensive analysis to identify the specific needs and preferences of users regarding financial management. Define the functional and non-functional requirements for the application based on user feedback and industry best practices.

**Design Phase:** Utilize software engineering principles to design the architecture and user interface of the application. Define the data structures, algorithms, and modules required for implementing the desired functionalities

**Implementation:** Develop the application using the C++ programming language and standard libraries. Implement the functionalities for expense tracking, budgeting, income monitoring, and financial reporting according to the defined requirements. Utilize appropriate data structures (e.g., arrays, vectors) and algorithms for efficient data manipulation and processing.

**Testing:** Conduct rigorous testing to ensure the reliability, stability, and usability of the application.

**Evaluation:** Evaluate the effectiveness of the application in meeting the predefined objectives and addressing user needs. Gather feedback from users through surveys, reviews, and usage analytics. Use this feedback to inform future iterations and improvements of the application.



## **1.4. EXPECTED OUTCOMES**

The Finance Manager project anticipates several positive outcomes for users. Firstly, users can expect improved financial awareness and control through the comprehensive tracking of expenses, enabling better decision-making regarding expenditure. With the budgeting feature, users can set clear financial goals and monitor their progress, fostering disciplined spending habits. Additionally, the income tracking functionality provides a holistic view of earnings from various sources, facilitating better understanding and management of overall income streams. By saving income reports locally, users can maintain a historical record of their earnings for future reference and analysis. Lastly, the financial reporting feature offers users a concise summary of monthly financial expenditure, promoting transparency and accountability in financial management practices. Overall, the Finance Manager project aims to empower users with the tools and insights necessary to achieve greater financial stability and well-being.

## **1.5. HARDWARE AND SOFTWARE REQUIREMENTS**

### **1.5.1. HARDWARE REQUIREMENTS**

- 1) Personal computer
- 2) Minimum of 2GB RAM
- 3) 64-bit operating system
- 4) Processor - x86 Compatible processor, 1.7 GHz Clock speed

### **1.5.2. SOFTWARE REQUIREMENTS**

- 1) Operating system - Windows 7 or above
- 2) Integrated Development Environment (IDE)- VS Code
- 3) C++ Compiler: GCC (GNU Compiler Collection)
- 4) File Management Software

## CHAPTER 2

# 2. FUNDAMENTALS OF C++

## 2.1 INTRO TO C++

C++ 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 a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.

One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high-level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications. C++ has a large, active community of developers and users, and a wealth of resources and tools available for learning and using the language. Some of the key features of C++ include:

**Object-Oriented Programming:** C++ supports object-oriented programming, allowing developers to create classes and objects and to define methods and properties for these objects.

**Templates:** C++ templates allow developers to write generic code that can work with any data type, making it easier to write reusable and flexible code.

**Standard Template Library (STL):** The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.

**Exception Handling:** C++ provides robust exception handling capabilities, making it easier to write code that can handle errors and unexpected situations.

**Pointer and direct Memory-Access:** C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).

**Mid-level language:** It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)

**Machine Independent but Platform Dependent:** A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.

**Simple:** It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.

Overall, C++ is a powerful and versatile programming language that is widely used for a range of applications and is well-suited for both low-level system programming and high-level application development

## 2.2 ADVANTAGES OF C++

**Performance:** C++ is known for its high performance, making it ideal for applications that require efficient memory management and execution speed. It offers low-level control over system resources, allowing developers to optimize code for maximum efficiency. This makes C++ particularly suitable for developing performance-critical applications such as real-time systems, games, and scientific simulations.

**Portability:** C++ is a portable language, meaning that code written in C++ can be compiled and executed on different platforms with minimal changes. This portability is facilitated by the existence of standardized libraries and compilers that support the language across various operating systems and hardware architectures.

**Object-oriented paradigm:** C++ supports object-oriented programming (OOP), which enables developers to organize code into reusable and modular components called objects. This paradigm promotes code reusability, maintainability, and scalability, making it easier to manage and extend large software projects.

**Rich standard library:** C++ comes with a rich standard library that provides a wide range of pre-built functions and data structures, including containers (such as vectors, arrays, and maps), algorithms, input/output operations, and utilities. This library simplifies common programming tasks and reduces the need for developers to write low-level code from scratch.

**Compatibility with C:** C++ is largely compatible with the C programming language, allowing developers to seamlessly integrate existing C code into C++ projects and vice versa. This compatibility enables easy interoperability between C and C++ codebases, providing flexibility and allowing developers to leverage existing libraries and code assets.

**Machine Independent:** C++ is not tied to any hardware or processor. If the compiler compiles the program in the system, it will be able to run no matter the hardware.

**Speed of execution:** C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.

**Community support and ecosystem:** C++ has a large and active community of developers, which provides access to a wealth of resources, including documentation, tutorials, forums, and open-source libraries. This vibrant ecosystem fosters collaboration, innovation, and knowledge sharing, making it easier for developers to learn and master the language.

## 2.3 DATA TYPES

All variables use data type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data they can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory. C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application. Data types specify the size and types of values to be stored. Storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines. C++ supports the following data types:

. **Primary or Built-in or Fundamental data type** - Integer, Character, Boolean, Floating, Point, Double Floating Point, Valueless or Void, Wide Character.

. **Derived data types** – Function, Array, Pointer, Reference

. **User-defined data types** – Class, Structure, Union, Enumeration, Typedef Datatype.

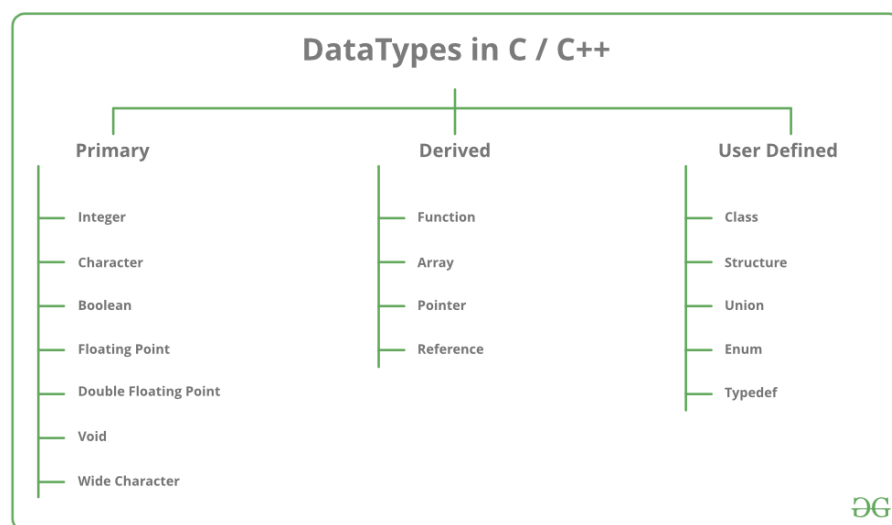


Fig:1. Data Types in C++

## 2.4 CONTROL FLOW

The sequence in which statements are carried out within a program is known as the C++ control flow. It consists of looping constructions (for, while, do-while) for repeating activities, conditional expressions (if, otherwise if, else) for decision-making, and branching statements (break, continue, return) to manage program flow. Unless these control flow structures change it, the main method is executed first and then proceeds in a sequential manner. Loops repeat code blocks recursively, branching statements manage the flow within loops or procedures, and conditional statements let code to run based on predefined criteria.

(i) **IF/ IF- ELSE/ IF- ELSE IF:** The "if" statement is used to execute code conditionally. This is further extended by the "if-else" statement, which offers an additional block to run in the event that the condition is false. "If else ladder" is used in further applications that requires multiple condition processing.

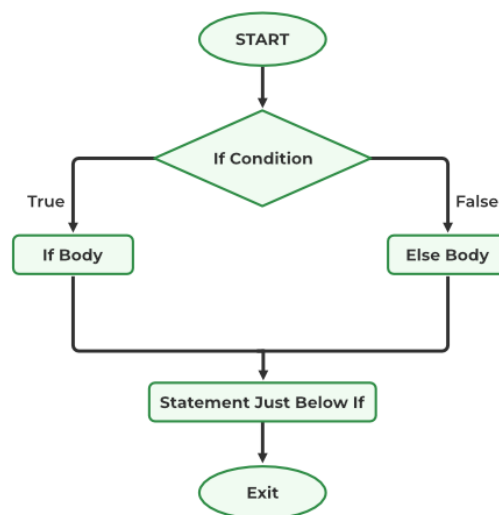


Fig:2. Flowchart of if/ if-else

(ii) **LOOPING CONSTRUCTS:** Loops are control flow structures that let a programming block run repeatedly. When the number of iterations is known ahead of time, the "for" loop is utilised, providing the initialization, condition, and iteration expressions in a condensed format. The "while" loop, on the other hand, checks the condition before each iteration and keeps running as long as it is true. By automating

repeated activities and iterating over data structures or sequences, both loops enable the efficient repetition of code.

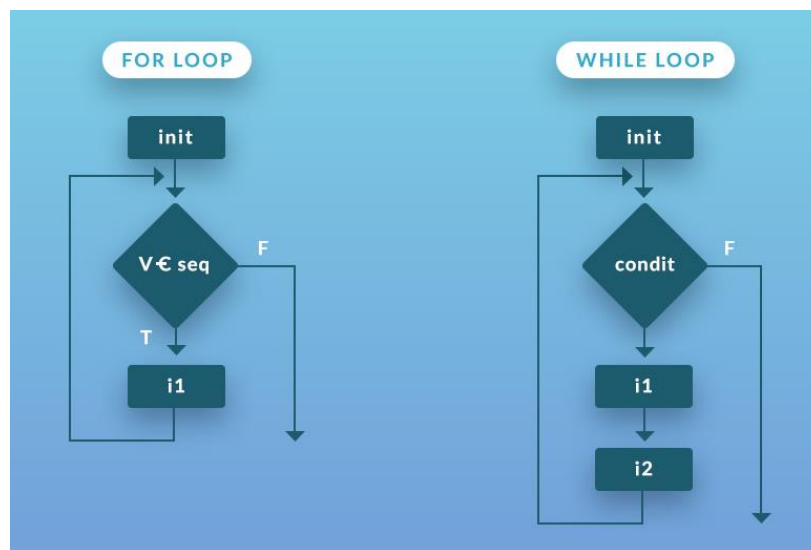


Fig:3. Looping Constructs (for and while)

(iii) **BREAK & CONTINUE:** When a loop in C++ is about to end prematurely, the "break" statement is used to stop it immediately and go on to the following line outside the loop. Conversely, the "continue" command is used in loops to exclude the remaining code from the current iteration and move straight on to the next one, so avoiding the lines that follow in the loop body.

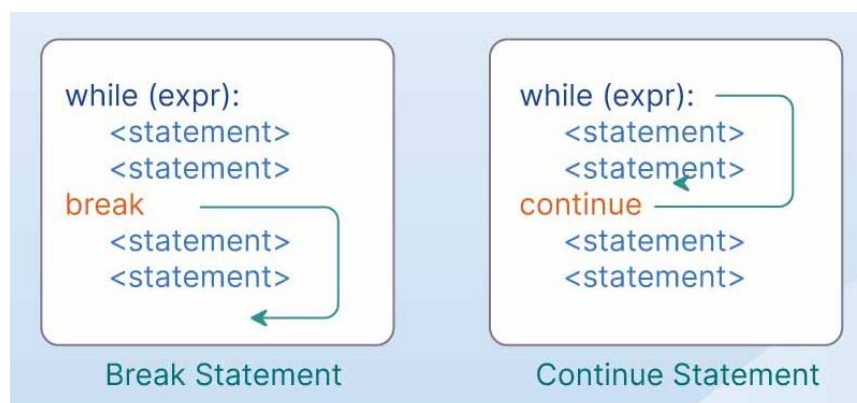


Fig:4. Break and Continue in C++

## 2.5 FUNCTIONS

In C++, functions are fundamental building blocks used to encapsulate a specific task or computation within a program. They serve to organize code, promote reusability, and enhance readability. A function in C++ typically consists of a function signature, which includes the return type, function name, and parameters enclosed within parentheses. The body of the function contains the instructions or statements that define its behavior. Functions can have a return type, indicating the type of value they return upon completion, or they can be void if they do not return any value. Parameters allow functions to accept input data, which can be used within the function body to perform operations.

C++ supports both user-defined functions, which are created by the programmer, and built-in functions provided by the standard library or other libraries. Functions can be declared before they are defined to inform the compiler about their existence, allowing them to be called from other parts of the program before their actual implementation. Additionally, C++ supports function overloading, enabling multiple functions with the same name but different parameter lists to coexist within the same scope, enhancing code flexibility and expressiveness.

## 2.6 OBJECT ORIENTED PROGRAMMING

The programming paradigm known as object-oriented programming, or OOP, organizes code around objects, which are just instances of classes. It encourages polymorphism, which allows objects to take on different forms, which permits classes to inherit traits and behaviors, and encapsulation, which groups data and methods together. OOP makes software development more efficient and maintainable by facilitating code organization, reusability, and a more natural representation of real-world things.

**(i) OBJECTS:** In C++, objects are instances of classes that symbolize actual physical objects. They enable communication within a program by encapsulating behaviors and data. Objects are the building blocks of object-oriented programming; they are made from class blueprints.



(ii) **CLASS:** in C++, a class is a blueprint that specifies an object's composition and actions. It acts as a template directing the production of object instances.

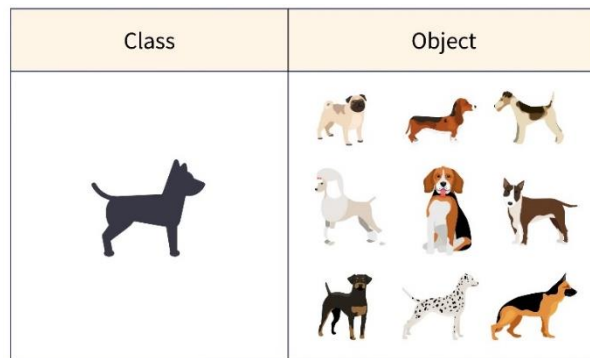


Fig:5. Class and Objects

(iii) **ENCAPSULATION:** C++ encapsulation includes combining methods and data into a class and limiting access to internal features. It encourages modularity, improves security, and makes efficient code organization easier.

### Encapsulation in C++



### Class

Fig:6. Encapsulation in C++

(iv) **INHERITANCE-** A subclass can inherit characteristics and behaviors from a superclass through inheritance, which creates a hierarchical connection between classes.

### Inheritance in C++

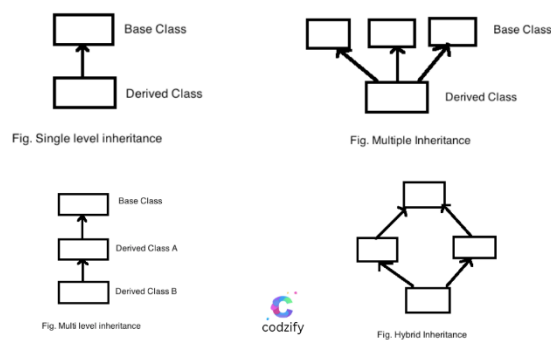


Fig:7. Types of Inheritances in C++

(v) **POLYMORPHISM**- Objects may assume many forms thanks to polymorphism, which is accomplished by overloading and overriding methods. It improves flexibility in C++ by allowing several implementations to be represented by a single interface.

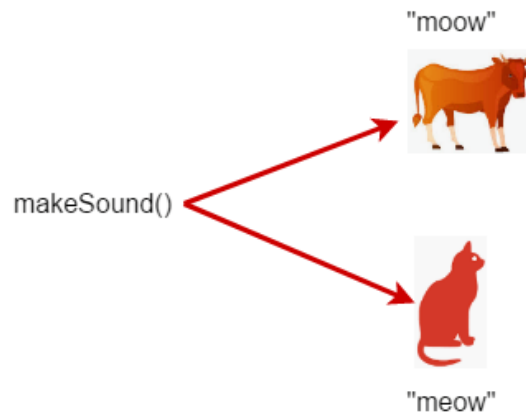


Fig:8. Polymorphism (many forms)

(vi) **ABSTRACTION**- Real-world elements are modelled in C++ as abstract classes or interfaces through the process of abstraction. It makes important features stand out while concealing superfluous details, making code understanding and upkeep easier.

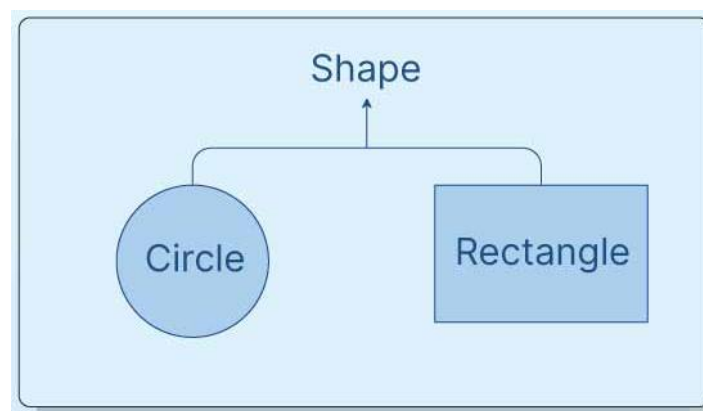


Fig:9. C++ Abstraction

## 2.7 FILE HANDLING

File handling in C++ involves operations for reading from and writing to files on disk. It provides mechanisms for managing file input and output streams, allowing programs to interact with external files for data storage, retrieval, and manipulation. Here's an overview of file handling in C++:

**1. File Streams:** C++ provides file stream classes, namely `ifstream` for input operations and `ofstream` for output operations. These classes are derived from the base class `fstream`, which can perform both input and output operations.

```
#include <fstream>
using namespace std;
int main() {
    ifstream inputFile("input.txt");
    ofstream outputFile("output.txt");

    // perform file operations

    inputFile.close();
    outputFile.close();
    return 0;
}
```

Fig:10. Code snippet of File streams

**2. Opening and Closing Files:** Files are opened using the `open()` method, which takes the file path as its argument. After performing file operations, files should be closed using the `close()` method to release system resources

**3. Reading from Files:** Input file streams (`ifstream`) allow reading data from files using various input operations such as `getline()`, `>>` (extraction operator), and `read()`.

```
ifstream inputFile("input.txt");
string line;
while (getline(inputFile, line)) {
    // process each line
}
inputFile.close();
```

Fig:11. Code snippet of File Reader

**4. Writing to Files:** Output file streams (ofstream) allow writing data to files using various output operations such as << (insertion operator) and write().

```
ofstream outputFile("output.txt");
outputFile << "Hello, world!" << endl;
outputFile.close();
```

Fig:12. Code snippet of File Writer

**5. Error Handling:** File operations may fail due to various reasons such as non-existent files, insufficient permissions, or disk full errors. It's essential to check for errors using methods like fail() and handle them appropriately.

```
ifstream inputFile("non_existent_file.txt");
if (!inputFile.is_open()) {
    cerr << "Error: Failed to open file." << endl;
    return 1;
}
```

Fig:13. Code snippet for Error Handling

**6. Binary File I/O:** C++ supports reading and writing binary data to files using the read() and write() methods. This is useful for dealing with non-textual data or maintaining the exact format of the data.

```
ofstream binaryFile("data.bin", ios::binary);
int num = 10;
binaryFile.write(reinterpret_cast<char*>(&num), sizeof(num));
```

Fig:14. Code snippet of Binary File I/O

## 2.8. IMPORT

In C++, importing external code and libraries is facilitated primarily through the #include directive. This directive allows developers to include header files containing declarations of classes, functions, variables, and constants into their source code. By including these headers, developers can access and utilize the functionality provided by external components, such as standard library modules or user-defined libraries. Additionally, the use of namespaces helps prevent naming conflicts between different

components. While C++ lacks a distinct import statement found in other languages, the `#include` directive serves a similar purpose, enabling code reuse, modularity, and the seamless integration of external functionality into C++ projects. Whether it's leveraging the rich features of the standard library or incorporating custom libraries and dependencies, the inclusion mechanism in C++ plays a vital role in building robust and scalable software solutions.

## **CHAPTER 3**

### **3. C++ STL GUIDE**

#### **3.1 INTRO TO STL**

The Standard Template Library (STL) stands as one of the cornerstones of modern C++ programming, offering a comprehensive collection of reusable algorithms and data structures. Introduced as a part of the C++ Standard Library, the STL provides a rich set of classes and functions that streamline the development process, promoting efficiency, readability, and maintainability in software projects. At its core, the STL embodies the principles of generic programming, allowing developers to write code that operates seamlessly across different data types while maintaining high performance. With its versatile containers, powerful algorithms, and iterators, the STL empowers C++ programmers to tackle a diverse array of tasks, ranging from basic data manipulation to complex algorithmic challenges. As an integral part of the C++ ecosystem, the STL continues to evolve, adapting to the changing needs of software development while remaining a cornerstone of modern C++ programming practices.

#### **3.2 STL CONTAINERS**

Containers in C++ are data structures provided by the Standard Template Library (STL) to store and manipulate collections of objects. They include vectors, lists, sets, maps, queues, and stacks. Containers offer various operations for insertion, removal, traversal, and access, catering to diverse programming needs with efficiency and versatility.

**SEQUENCE CONTAINERS-** Sequence containers implement data structures that can be accessed sequentially.

- array: Static contiguous array (class template)
- vector: Dynamic contiguous array (class template)
- deque: Double-ended queue (class template)
- forward\_list: Singly-linked list (class template)
- list: Doubly-linked list (class template)

**ASSOCIATIVE CONTAINERS-** Associative containers implement sorted data structures that can be quickly searched ( $O(\log n)$  complexity).

- Set: Collection of unique keys, sorted by keys (class template)
- Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).
- multiset: Collection of keys, sorted by keys (class template)
- multimap: Collection of key-value pairs, sorted by keys (class template)

**UNORDERED ASSOCIATIVE CONTAINERS-** Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ( $O(1)$  amortized,  $O(n)$  worst-case complexity).

- unordered\_set: Collection of unique keys, hashed by keys. (class template)
- unordered\_map: Collection of key-value pairs, hashed by keys, keys are unique. (class template)
- unordered\_multiset: Collection of keys, hashed by keys (class template)
- unordered\_multimap: Collection of key-value pairs, hashed by keys (class template)

**CONTAINER ADAPTERS-** Container adapters provide a different interface for sequential containers.

- stack: Adapts a container to provide stack (LIFO data structure) (class template).
- queue: Adapts a container to provide queue (FIFO data structure) (class template).
- priority\_queue: Adapts a container to provide priority queue (class template).

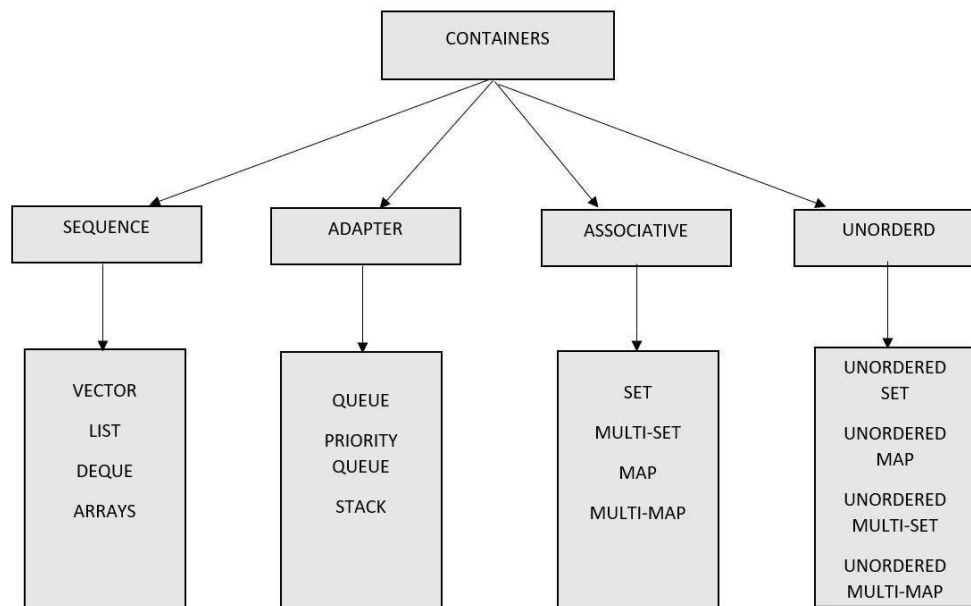


Fig:15. STL Containers Comprehensive Flow Chart

### 3.3 STL ALGORITHMS

STL (Standard Template Library) algorithms are a set of pre-defined generic algorithms provided by the C++ Standard Library. These algorithms operate on various data structures like vectors, arrays, lists, and more. They are designed to work with iterators, which provide a generalized interface for accessing elements in a sequence, allowing these algorithms to be used with different container types.

. **sort(first\_iterator, last\_iterator)**: This algorithm is used to sort elements in the range [first\_iterator, last\_iterator) in ascending order. It uses the < operator to compare elements by default.

. **sort(first\_iterator, last\_iterator, greater<int>())**: This variation of the sort algorithm sorts elements in descending order. It achieves this by providing a comparison function object (greater<int>()) that orders elements in reverse.

. **reverse(first\_iterator, last\_iterator)**: This algorithm reverses the order of elements in the range [first\_iterator, last\_iterator). If the sequence is initially sorted in ascending order, after applying reverse, it will be sorted in descending order, and vice versa.

. **max\_element(first\_iterator, last\_iterator)**: This algorithm returns an iterator pointing to the maximum element in the range [first\_iterator, last\_iterator). It compares elements using the < operator.

. **min\_element(first\_iterator, last\_iterator)**: This algorithm returns an iterator pointing to the minimum element in the range [first\_iterator, last\_iterator). It also compares elements using the < operator.

. **accumulate(first\_iterator, last\_iterator, initial\_value\_of\_sum)**: This algorithm calculates the sum of elements in the range [first\_iterator, last\_iterator) and returns the result. It starts the summation with the initial value specified.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> // For accumulate
using namespace std;
int main() {
    vector<int> vec = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // Sorting in ascending order
    sort(vec.begin(), vec.end());
    cout << "Sorted in ascending order: ";
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
    // Sorting in descending order
    sort(vec.begin(), vec.end(), greater<int>());
    cout << "Sorted in descending order: ";
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
    // Reversing the vector
    reverse(vec.begin(), vec.end());
    cout << "Reversed: ";
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
    // Finding maximum and minimum elements
    cout << "Max element: " << *max_element(vec.begin(), vec.end()) << endl;
    cout << "Min element: " << *min_element(vec.begin(), vec.end()) << endl;
    // Accumulating the sum of elements
    int sum = accumulate(vec.begin(), vec.end(), 0);
    cout << "Sum of elements: " << sum << endl;
    return 0;
}
```

Fig:16. STL Pre- def Algorithms implementation

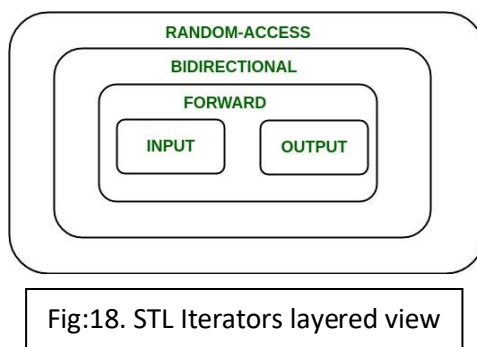
```
Sorted in ascending order: 1 1 2 3 4 5 5 6 9
Sorted in descending order: 9 6 5 5 4 3 2 1 1
Reversed: 1 1 2 3 4 5 5 6 9
Max element: 9
Min element: 1
Sum of elements: 36
```

Fig:17. Output of STL Algorithms implementation



### 3.4 STL ITERATORS

Iterators in C++ are objects that provide a way to access and traverse the elements of a container (like arrays, vectors, lists, etc.) in a sequential manner. They act as pointers to elements within the container and provide a uniform interface for iterating over the elements, regardless of the specific container type. Iterators abstract away the underlying implementation details of the container, allowing algorithms to operate on containers in a generic and efficient way. Iterators typically support operations such as dereferencing, incrementing, and comparing. Some common iterator operations and concepts:



**Dereferencing (\*):** Dereferencing an iterator retrieves the value of the element it points to. For example, `*it` would give you the value of the element pointed to by the iterator `it`.

**Incrementing (++):** Incrementing an iterator moves it to the next element in the container. For example, `++it` advances the iterator `it` to the next element.

**Decrementing (--):** For bidirectional iterators (e.g., those of lists), decrementing moves the iterator to the previous element.

**Equality and Inequality Comparisons:** Iterators can be compared for equality (`==`) and inequality (`!=`). Two iterators are considered equal if they point to the same element or if they both point to the end of the container.

**Relational Comparisons:** Some iterators support relational comparisons such as `<`, `>`, `<=`, and `>=`, allowing you to compare the positions of two iterators within the container.

**Iterator Categories:** Iterators are classified into different categories based on the operations they support and their capabilities. The major iterator categories in C++ are:

**Input iterators:** Support reading values from the container, but only allow forward traversal (incrementing).

**Output iterators:** Support writing values to the container, but only allow forward traversal.

**Forward iterators:** Combine the features of input and output iterators, allowing both reading and writing, and support forward traversal.

**Bidirectional iterators:** Support bidirectional traversal (both forward and backward).

**Random-access iterators:** Provide the most functionality, including random access (efficiently jumping to any element), arithmetic operations like addition and subtraction, and comparisons.

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->, []	= *i	*i=	++, --, +=, -=, +, -	==, !=, <,>, <=,>=

Fig:19. STL Iterators Operation

### 3.5 STL FUNCTIONS (FUNCTORS)

In the context of the C++ Standard Library (STL), a "functor" is a function object or a class that overloads the function call operator operator(). Functors are used as callable objects, providing a way to encapsulate a function or a set of functions within an object. Functors offer more flexibility and customization options compared to regular functions, especially when used with algorithms from the <algorithm> header. Some key points about functors in the STL:

**(i)Function Objects:** Functors are often referred to as function objects because they behave like objects but can be invoked as if they were functions.

**(ii)Customizable Behavior:** Functors can encapsulate more complex behavior than simple functions. They can have internal state and maintain context between function calls, allowing for greater flexibility and customization.

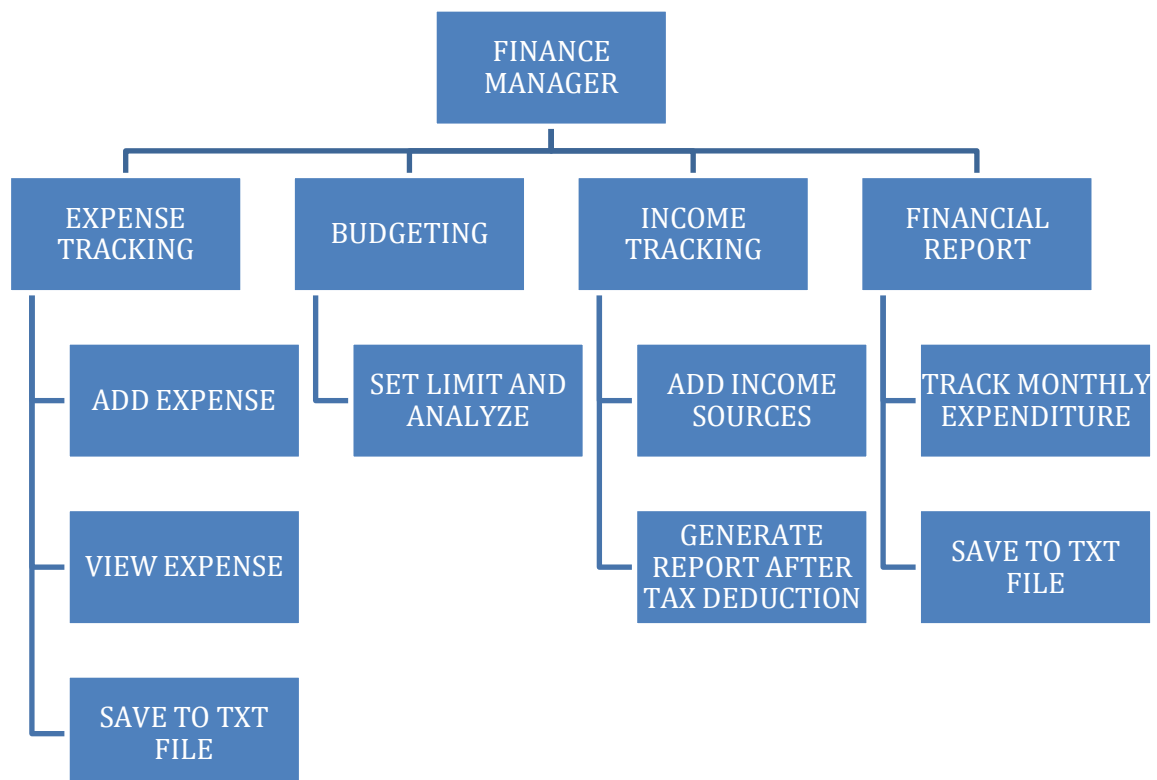
**(iii) Usage with Algorithms:** Many algorithms in the STL, such as `std::sort()`, `std::transform()`, `std::for_each()`, etc., accept functors as arguments. This allows algorithms to perform operations defined by the functor on elements of containers.

**(iv) Adaptable Functionality:** Functors can be used to adapt existing functions or function-like behaviors into the STL algorithms. For example, by creating a functor that wraps a comparison function, you can use it with sorting algorithms to customize the sorting criteria.

**(v) Performance:** Functors can potentially be more efficient than passing function pointers or using function objects because they can be inlined by the compiler and may avoid the overhead of dynamic dispatch.

## CHAPTER 4

### DESIGN AND ARCHITECTURE



## CHAPTER 5

### IMPLEMENTATION

#### 5.1 PREPROCESSOR DIRECTIVES AND INITIALIZATIONS

```
#include <iostream>

#include <vector>

#include<iomanip>

#include<numeric>

#include<list>

#include<array>

#include<fstream>

using namespace std;

float income, debt, loan,tot2=0;

int choice1, count3=0, count4=0;

char choice2, choice5, choice6;

double Finrepincome;

void ExpenseTracking();

void Budgeting();

void IncomeTracking();

void FinancialReport();

void saveExpenses(const vector<double>& expenses);

void saveIncomes(const list<pair<string, double>>& incomes);

void saveFinancialReport(const array<int, 7>& Financial_rep);
```

## 5.2 FUNCTION TO DISPLAY TOTAL EXPENSES

```
void displayTotalExpenses(const vector<double>& expenses) {  
  
    double total = 0.0;  
  
    for (double expense : expenses) {  
  
        total += expense;  
  
    }  
  
  
    cout << fixed << setprecision(2);  
  
    cout << "Total Expenses: " << total << endl;  
  
}
```

## 5.3 FUNCTION TO TRACK EXPENSES

```
void ExpenseTracking() {  
  
    static vector<double> expenses;  
  
    do{  
  
        cout << "1. Add Expense\n";  
  
        cout << "2. View Total Expenses\n";  
  
        cout << "3. Save contents to file\n";  
  
        cout << "Enter your choice: ";  
  
        cin >> choice1;  
  
        switch (choice1) {  
  
            case 1: {  
  
                cout << "Enter expense amount:\t";  
  
                double expense;  
  
                cin >> expense;
```

```
        expenses.push_back(expense);

        cout << "Expense added successfully!\n";

        break;
    }

    case 2:

        if (expenses.empty()) {

            cout << "No expenses added yet.\n";

        } else {

            displayTotalExpenses(expenses);

        }

        break;

    case 3:

        if (expenses.empty()) {

            cout << "No expenses to save.\n";

        } else {

            saveExpenses(expenses);

        }

        break;

    default:

        cout << "Invalid choice!";

    }

    cout << "\n-----\n";

    cout << " Do you wish to add/ view your expenses? (Y/N)" << endl;

    cin >> choice2;

} while (choice2 == 'Y' || choice2 == 'y');
```

```
}
```

## 5.4 FUNCTION TO SAVE EXPENSES TO TXT FILE

```
void saveExpenses(const vector<double>& expenses) {  
  
    ofstream outputFile("FileManager.txt", ios::app); // Open the file in append mode  
  
    double i=0;  
  
    outputFile<<"EXPENSE TRACKING\n";  
  
    if (outputFile.is_open()) {  
  
        for (double expense : expenses) {  
  
            outputFile << expense << "\n";  
  
            i+=expense;  
  
        }  
  
        outputFile<<"TOTAL:\t"<< i <<endl;  
  
        outputFile<<"-----\n";  
  
        cout << "Expenses saved to file successfully!\n";  
  
        outputFile.close();  
  
    } else {  
  
        cout << "Error opening file for writing!\n";  
  
    }  
  
}
```

## 5.5 FUNCTION FOR BUDGETING

```
void Budgeting() {  
  
    vector<double> incomes;  
  
    vector<double> expenses;  
  
    double income, expense;
```

```
char addmore;

do {

    cout << "Enter income amount:\t";

    cin >> income;

    incomes.push_back(income);

    cout << "Enter expense amount:\t";

    cin >> expense;

    expenses.push_back(expense);

    cout << "Do you want to add more income or expenses? (Y/N): ";

    cin >> addmore;

} while (addmore == 'Y' || addmore == 'y');


double totalIncome = accumulate(incomes.begin(), incomes.end(), 0.0);

double totalExpense = accumulate(expenses.begin(), expenses.end(), 0.0);

double budget = totalIncome - totalExpense;

cout << "\nBudget Summary\n";

cout << "-----\n";

cout << "Total Income:\t" << totalIncome << endl;

cout << "Total Expenses:\t" << totalExpense << endl;

cout << "-----\n";

cout << "Budget: $" << budget << endl;

if (budget > 0) {

    cout << "You have a surplus. Good job!\n";
```



```
} else if (budget < 0) {  
    cout << "Warning: You have a deficit. Adjust your expenses.\n";  
} else {  
    cout << "Your budget is balanced. Keep it up!\n";  
}  
}
```

## 5.6 FUNCTION TO TRACK INCOME

```
void IncomeTracking() {  
    static list<pair<string, double>> Incomes;  
  
    long double tax1;  
  
    do{  
  
        count3++;  
  
        string a; double b;  
  
        cout<< " enter your source of income number: "<< count3 <<endl;  
  
        cin>> a;  
  
        cout<< " enter your income/ revenue from source: " << count3<<endl;  
  
        cin>> b;  
  
        Incomes.push_back({a,b});  
  
        cout<< "do you want to add more? (Y/N)"<<endl;  
  
        cin>>choice5;  
  
        } while(choice5=='Y' || choice5 == 'y');  
  
        cout <<"                               INCOME CHART                               " <<  
endl;  
  
        cout << setw(5)<<"\tS.no.\t " <<setw(30)<< "INCOME SOURCE\t\t\t"<<setw(20) <<  
"INCOME AFTER TAX DEDUCTION\n";
```

```
//cout << setw(10) << "S.no." << setw(35) << "Income source" << setw(35) <<
"Income after tax deduction\n";

for (const auto& income : Incomes) {

    count4++;

    if (income.second > 1000000 || income.second == 1000000) {

        tax1 = income.second - (12500 + ((income.second - 500000) * 0.10));

    }

    else if ((income.second > 500000) && (income.second < 1000000)){

        tax1 = income.second - ((income.second - 250000) * 0.05);

    }

    else{

        tax1= income.second;

    }

    cout << setw(10) << count4 << setw(35) << income.first << setw(35) << fixed <<
    setprecision(2)<< tax1 << endl;

}

saveIncomes(Incomes);

}
```

## 5.7 FUNCTION TO SAVE INCOME TO TXT FILE

```
void saveIncomes(const list<pair<string, double>>& incomes) {

    ofstream outputFile("FileManager.txt", ios::app); // Open the file in append mode

    if (outputFile.is_open()) {

        outputFile<< "INCOME TRACKING\n";

        int sno=1;
```

```
double tot1=0.0;

for (const auto& income : incomes) {

    outputFile<<sno<<" " << income.first << " :" << income.second << "\n";

    tot1+=income.second;

    sno++;

}

outputFile<< "TOTAL:\t" << tot1<<endl;

outputFile<<"-----\n";

cout << "Income data saved to file successfully!\n";

outputFile.close();

} else {

    cout << "Error opening file for writing!\n";

}

}
```

## 5.8 FUNCTION TO GENERATE FINANCIAL REPORT

```
void FinancialReport() {

    const int s = 7;

    cout<< " Enter your total income\n";

    cin>>Finrepincome;

    array <int, s> Financial_rep;

    cout<< " Enter your Rent/ mortgage amount, if any\n";

    cin>> Financial_rep[0];

    cout<< " Enter your Utilities expense\n";

    cin>> Financial_rep[1];

    cout<< " Enter your Interest/ EMI amount, if any\n";
```

```
cin>> Financial_rep[2];

cout<< " Enter your Electricity expenditure\n";

cin>> Financial_rep[3];

cout<< " Enter your Gasoline expense\n";

cin>> Financial_rep[4];

cout<< " Enter Miscallaneous, if any\n";

cin>> Financial_rep[5];

// Calculate total expenses

int totalExpenses = accumulate(Financial_rep.begin(), Financial_rep.end()-1, 0);

// Calculate savings

double savings = Finrepincome - totalExpenses;

Financial_rep[6]= savings;

cout<< "-----"<<endl;

cout<< "          FINANCIAL REPORT          "<<endl;

cout<< "-----"<<endl;

cout<<"Rent:\t"<<Financial_rep[0]<<endl;

cout<<"Utilities:\t"<<Financial_rep[1]<<endl;

cout<<"Loan/ EMIs:\t"<<Financial_rep[2]<<endl;

cout<<"Electricity:\t"<<Financial_rep[3]<<endl;

cout<<"Gasoline:\t"<<Financial_rep[4]<<endl;

cout<<"Miscallenous:\t"<<Financial_rep[5]<<endl;

cout<<"Savings:\t"<<Financial_rep[6]<<endl;

if(Financial_rep[6] >0 ){
```

```
    cout<< "You have surplus for the month!!, and "<<(Finrepincome -
totalExpenses)<< " rupees is your total expenditure"<<endl;

}

else if (Financial_rep[6]==0){

    cout<< "You have the perfect amount for the month!!, but you have 0 rupees in your
Financial savings"<<endl;

}

else{

    cout<<"Oops!! you fell short by "<< abs((Finrepincome - totalExpenses)) << "
amount"<<endl;

}

saveFinancialReport(Financial_rep);

}
```

## 5.9 FUNCTION TO SAVE REPORT TO TXT FILE

```
void saveFinancialReport(const array<int, 7>& Financial_rep) {

    ofstream outputFile("FileManager.txt", ios::app); // Open the file in append mode

    if (outputFile.is_open()) {

        for (const auto& finrep : Financial_rep) {

            tot2+=finrep;

        }

        outputFile<< "FINANCIAL REPORT\n "<<endl;

        outputFile<< "BALANCE BEFORE EXPENDITURE:\t"<<tot2<<endl;

        outputFile << "Rent: " << Financial_rep[0] << "\n";

        outputFile << "Utilities: " << Financial_rep[1] << "\n";

    }
```

```
    outputFile << "Loan/EMIs: " << Financial_rep[2] << "\n";
    outputFile << "Electricity: " << Financial_rep[3] << "\n";
    outputFile << "Gasoline: " << Financial_rep[4] << "\n";
    outputFile << "Miscellaneous: " << Financial_rep[5] << "\n";
    outputFile << "Savings: " << Financial_rep[6] << "\n\n";
    outputFile << "-----\n";
    cout << "Financial report saved to file successfully!\n";
    outputFile.close();
} else {
    cout << "Error opening file for writing!\n";
}
}
```

## 5.10 MAIN FUNCTION

```
int main() {
    int choice;

    cout << "===== " << endl;
    cout << "    FINANCE MANAGER    " << endl;
    cout << "===== " << endl;
    cout << endl;

    do {
        const string a[4] = {"1. Expense Tracking", "2. Budgeting", "3. Income Tracking",
"4. Financial Report"};

        for (auto i : a) {
            cout << i << endl;
        }
    }
```

```
cout << "Enter which operation to perform: ";

cin >> choice;

switch (choice) {

    case 1: {

        ExpenseTracking();

        break;

    }

    case 2: {

        Budgeting();

        break;

    }

    case 3: {

        IncomeTracking();

        break;

    }

    case 4: {

        FinancialReport();

        break;

    }

    default: {

        cout << "Invalid entry\n";

    }

}


cout << "Do you want to continue with Finanace Manager (Y/N)? ";
```

```
    cin >> choice6;  
    } while (choice6 == 'Y' || choice6 == 'y');  
    return 0;  
}
```



## CHAPTER 6

## RESULTS

```
FINANCE MANAGER
=====

1. Expense Tracking
2. Budgeting
3. Income Tracking
4. Financial Report
Enter which operation to perform: 1
1. Add Expense
2. View Total Expenses
3. Save contents to file
Enter your choice: 1
Enter expense amount: 2300
Expense added successfully!

-----
Do you wish to add/ view your expenses? (Y/N)
Y
1. Add Expense
2. View Total Expenses
3. Save contents to file
Enter your choice: 2
Total Expenses: 2300.00

-----
Do you wish to add/ view your expenses? (Y/N)
Y
1. Add Expense
2. View Total Expenses
3. Save contents to file
Enter your choice: 3
Expenses saved to file successfully!

-----
Do you wish to add/ view your expenses? (Y/N)
Y
1. Add Expense
2. View Total Expenses
3. Save contents to file
Enter your choice: 
```

## 6.1. Expense Tracker Execution

```
File Edit View
Loan/EMIs: 100
Electricity: 100
Gasoline: 100
Miscellaneous: 100
Savings: 400

TOTAL: 1000

-----
FINANCIAL REPORT

BALANCE BEFORE EXPENDITURE: 2000
Rent: 1000
Utilities: 100
Loan/EMIs: 100
Electricity: 100
Gasoline: 100
Miscellaneous: 100
Savings: 500

-----
EXPENSE TRACKING
2300
TOTAL: 2300
-----
```

## 6.2. Expenses saved to txt file

```

=====
FINANCE MANAGER
=====
1. Expense Tracking
2. Budgeting
3. Income Tracking
4. Financial Report
Enter which operation to perform: 2
Enter income amount: 1000
Enter expense amount: 300
Do you want to add more income or expenses? (Y/N): Y
Enter income amount: 500
Enter expense amount: 1000
Do you want to add more income or expenses? (Y/N): N

Budget Summary
-----
Total Income: 1500
Total Expenses: 1300
-----
Budget: $200
You have a surplus. Good job!
Do you want to continue with Finance Manager (Y/N)? █
    
```

### 6.3. Budgeting

```

=====
FINANCE MANAGER
=====
1. Expense Tracking
2. Budgeting
3. Income Tracking
4. Financial Report
Enter which operation to perform: 3
Enter your source of income number: 1
FREELANCE
Enter your income/ revenue from source: 1
23000
do you want to add more? (Y/N)
Y
Enter your source of income number: 2
EDITING
Enter your income/ revenue from source: 2
1000
do you want to add more? (Y/N)
Y
Enter your source of income number: 3
INTERNSHIP
Enter your income/ revenue from source: 3
23000
do you want to add more? (Y/N)
N

INCOME CHART
-----
S.no.      INCOME SOURCE      INCOME AFTER TAX DEDUCTION
1          FREELANCE          23000.00
2          EDITING           1000.00
3          INTERNSHIP         23000.00
-----
Income data saved to file successfully!
Do you want to continue with Finance Manager (Y/N)? █
    
```

### 6.4. Income Tracker Execution

```

crysta x
File Edit View
BALANCE BEFORE EXPENDITURE: 2000
Rent: 1000
Utilities: 100
Loan/EMIs: 100
Electricity: 100
Gasoline: 100
Miscellaneous: 100
Savings: 500

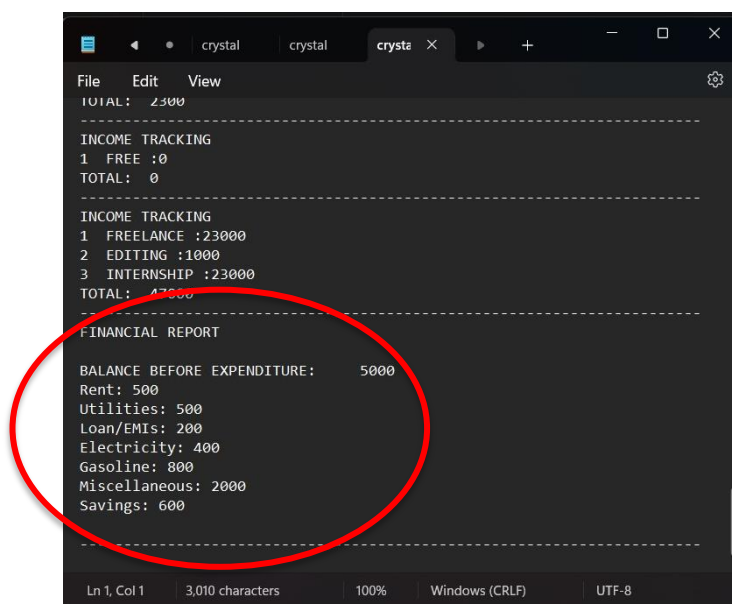
-----
EXPENSE TRACKING
2300
TOTAL: 2300
-----
INCOME TRACKING
1 FREELANCE :23000
TOTAL: 0
-----
INCOME TRACKING
1 FREELANCE :23000
2 EDITING :1000
3 INTERNSHIP :23000
TOTAL: 47000
-----
Ln 1, Col 1 2,781 characters 100% Windows (CRLF) UTF-8
    
```

### 6.5. Income Saved to Txt File

```
=====
FINANCE MANAGER
=====

1. Expense Tracking
2. Budgeting
3. Income Tracking
4. Financial Report
Enter which operation to perform: 4
Enter your total income
5000
Enter your Rent/ mortgage amount, if any
500
Enter your Utilities expense
500
Enter your Interest/ EMI amount, if any
200
Enter your Electricity expenditure
400
Enter your Gasoline expense
800
Enter Miscellaneous, if any
2000
=====
FINANCIAL REPORT
=====
Rent:          500
Utilities:     500
Loan/ EMIs:    200
Electricity:   400
Gasoline:      800
Miscellaneous: 2000
Savings:       600
You have surplus for the month!! and 600 rupees is your total expenditure
Financial report saved to file successfully!
Do you want to continue with Finance Manager (Y/N)?
```

### 6.6. Financial Report Execution



```
File Edit View
TOTAL: 23000
=====
INCOME TRACKING
1 FREE :0
TOTAL: 0
=====
INCOME TRACKING
1 FREELANCE :23000
2 EDITING :1000
3 INTERNSHIP :23000
TOTAL: 47000
=====
FINANCIAL REPORT
=====
BALANCE BEFORE EXPENDITURE: 5000
Rent: 500
Utilities: 500
Loan/EMIs: 200
Electricity: 400
Gasoline: 800
Miscellaneous: 2000
Savings: 600
=====
```

### 6.7. Financial Report Saved to Txt File

## CHAPTER 7

### CONCLUSION

In conclusion, the Finance Manager project represents a significant stride towards empowering individuals with the necessary tools to manage their finances effectively. By integrating various features such as expense tracking, budgeting, income monitoring, and financial reporting, the project offers users a comprehensive platform to gain insight into their financial activities. Through the diligent utilization of C++ Standard Library features and data structures such as arrays, vectors, and lists, the project ensures efficient data storage, manipulation, and retrieval, contributing to a seamless user experience.

The anticipated outcomes of the Finance Manager project are promising. Users can expect to achieve improved financial awareness and control, enabling them to make informed decisions about their expenditures. With the ability to set budgets and monitor spending habits, users can cultivate disciplined financial practices, ultimately leading to better financial stability. The income tracking functionality provides users with a holistic view of their earnings, facilitating a better understanding of their financial inflows and outflows. Additionally, the project's capability to generate financial reports offers users concise summaries of their monthly expenditures, promoting transparency and accountability in financial management.

Furthermore, the project's flexibility and compatibility with different operating systems ensure accessibility to a broader user base. By adhering to software development best practices and ensuring robust documentation, the Finance Manager project is poised to deliver a reliable and user-friendly solution for managing personal finances.

In essence, the Finance Manager project not only serves as a practical tool for individuals to track and manage their finances but also signifies the potential of technology to empower individuals in achieving financial well-being. As financial literacy continues to be a crucial aspect of modern life, projects like Finance Manager play a vital role in equipping individuals with the knowledge and resources to navigate their financial journey with confidence.

## 7.1 REFERENCES

- Stroustrup, B. (2013). The C++ Programming Language (4th Edition). Addison-Wesley Professional.
- Josuttis, N. M. (2012). The C++ Standard Library: A Tutorial and Reference (2nd Edition). Addison-Wesley Professional.
- <https://www.javatpoint.com/cpp-tutorial>
- <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>
- <https://en.wikipedia.org/wiki/C%2B%2B>
- W3Schools
- CPlusPlus.com
- Programiz