C++ Note Summary

C++ Notes Directory

Lecture 1 – Introduction

1. What is C++
2. Compilation and Running
3. C++ vs C

Lecture 2 – Basic Syntax

1. Structure of a C++ Program
2. `#include` and `<iostream>`
3. Namespace
4. std and using namespace
5. endl vs \n
6. main() Function
7. Comments

Lecture 3 – Variables and Data Types

1. Declaring Variables
2. Primitive Types
   - int
   - double
   - char
   - bool
   - string

Lecture 4 – Input and Output

1. cout
   - printf vs cout
2. cin
   - scanf vs cin

Lecture 5 – Control Structures

1. if-else Statements
2. for Loops
3. while Loops

Lecture 6 – Functions

1. Defining Functions
2. Parameters and Return Types
3. Multiple File Compilation

Lecture 7 – Arrays and Strings

1. Arrays
   - Declaration
   - Accessing Elements
2. Strings
   - Using string Class
   - Common Methods

Lecture 8 – Classes and Objects

1. Defining Classes
2. Constructors
   - Member Initializer List
3. this Pointer
4. Methods
5. Creating Objects

Lecture 9 – Pointers and References

1. Pointers
   - Declaration
   - Dereferencing
   - Address-of Operator
2. References
   - Reference Parameters
   - swap Example

Lecture 10 – Dynamic Memory

1. new and delete Operators

Lecture 11 – STL (Standard Template Library)

1. vector
2. map
3. set

Lecture 12 – Templates

1. Function Templates
2. Class Templates
3. Designing Generic Functions with Templates

Lecture 13 – File I/O

1. Reading Files (ifstream)
2. Writing Files (ofstream)

Lecture 14 – Exception Handling

1. try-catch Blocks

Lecture 15 – Advanced Topics

1. Operator Overloading
2. Namespaces (Advanced)
3. Multiple Paradigm Support

> 1.Introduction
>
> # 1.1. What is C++
>
> ## Definition

C++ is a general-purpose programming language that supports both procedural and object-oriented programming.

Feature

- Combines C features with object-oriented features
- Fast execution

Syntax

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

int main() {
    cout << "Hello, C++!";
    return 0;
}
```

Example

- Printing "Hello, C++!" to the console.

Explanation

- `#include <iostream>` allows input and output.
- `using namespace std;` lets you use standard names without prefixing `std::`.
- `main()` is the entry point.

Remark

- C++ was developed by Bjarne Stroustrup as an extension of C.

1.2. Compilation and Running

Definition

Compilation converts C++ code into an executable program.

Feature

- Uses a compiler like g++
- Produces machine code

Syntax

```
g++ program.cpp -o program
./program
```

Example

- Compile with `g++ program.cpp -o program` then run with `./program`.

Explanation

- `g++` is the compiler.
- `-o program` sets output file name.

Remark

- Compilation catches syntax errors before running.

1.3. C++ vs C

Definition

C++ extends C by adding object-oriented features.

Feature

- Supports classes and objects
- Better type safety

Syntax

```cpp
class Example {
public:
    int x;
};
```

Example

- Defining a class in C++ (not possible in C).

Explanation

- `class` keyword is for creating user-defined types.

Remark

- C++ can run most C code but not vice versa.

2.Basic Syntax

2.1. Structure of a C++ Program

Definition

A basic C++ program has headers, a `main()` function, and statements.

Feature

- Starts with `#include`
- Has `main()` as entry

Syntax

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

int main() {
    return 0;
}
```

Example

- A minimal C++ program.

Explanation

- Program execution begins from `main()`.

Remark

- Without `main()`, the program won't run.

2.2. `#include` and `<iostream>`

Definition

`#include` **tells the compiler to add code from another file.**

`<iostream>` **is a standard library header file that lets you use input and output commands like** `cin` **and** `cout`**.**

Feature

- `#include` is used to **import libraries**.
- `<iostream>` is needed for **printing and reading** in C++, providing tools like `cin`, `cout`, `cerr`.

Syntax

```cpp
#include <iostream>
```

Example

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

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

Explanation

- `#include <iostream>` adds the input-output library to your program.
- `< >` is for **standard libraries**.
- `" "` is for **your own files** (e.g. `#include "myfile.h"`).

Remark

- Always write `#include <iostream>` at the **top** when you use `cin` or `cout`.

2.3. Namespace

Definition

A namespace is a way to organize names (variables, functions, classes) in C++ to avoid name conflicts by grouping related code together.

Feature

- Prevents naming conflicts
- Groups code logically
- The standard namespace is called `std`

Syntax

```
namespace name {
    // code
}
```

Example

```
namespace LibraryA {
    void print() {
        cout << "From Library A";
    }
}

namespace LibraryB {
    void print() {
        cout << "From Library B";
    }
}

int main() {
    LibraryA::print(); // Calls print from LibraryA
    LibraryB::print(); // Calls print from LibraryB
    return 0;
}
```

Explanation

- `namespace MathTools { ... }` creates a box called `MathTools` that contains the `add` function.
- To use `add`, write `MathTools::add(3, 4)`.
- `::` is the **scope resolution operator**, telling the compiler to find `add` inside `MathTools`.

Remark

- You can have multiple namespaces in a program.
- Avoids conflicts when two functions or classes have the same name.

2.4. `std` and Using Namespace

Definition

`std` is short for **“standard”**. It is a *namespace* that stores all the **standard C++ library features** like `cout`, `cin`, and `string`.

`using namespace std;` allows you to use these names without writing `std::` every time.

Feature

- Groups standard tools to keep code organized
- Saves typing
- Makes code look cleaner

Syntax

```
using namespace std;
```

Example

- **Without using namespace std;**

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    std::string s = "C++";
    std::cout << s << std::endl;
    return 0;
}
```

- **With using namespace std;**

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

int main() {
    cout << "Hello, world!" << endl;
    string s = "C++";
    cout << s << endl;
    return 0;
}
```

Explanation

- `std` is like a **box called “standard”** that stores tools (`cout` , `cin` , `string`).
- Writing `std::cout` tells the computer **“use `cout` from the `std` box.”**
- Adding `using namespace std;` tells the computer **“I will use the `std` box a lot, so please open it for me.”**

Then you can write `cout` directly without `std::` .

Remark

- In **small programs**, using `using namespace std;` is fine.
- In **large projects**, it is better to avoid it in header files to prevent naming conflicts.
- `std` keeps your code organized and avoids confusion with your own variables or functions.

2.5. `endl` vs. `\n`

Definition

Both `endl` and `\n` are used to insert a new line when printing output in C++.

Feature

- `\n` is a **newline character**.
- `endl` is a **stream manipulator** that adds a newline **and flushes the output buffer**.

Syntax

```
cout << "Hello\nWorld";
cout << "Hello" << endl << "World";
```

Example

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

int main() {
    cout << "Line 1\n";
    cout << "Line 2" << endl;
    cout << "Line 3\n";
    cout << "Line 4" << endl;
    return 0;
}
```

Explanation

- `\n` is faster because it **only adds a newline**.
- `endl` adds a newline **and flushes the output**, which forces the program to display everything in the buffer immediately.

Remark

- Use `\n` for **performance-critical code** (e.g. competitive programming).
- Use `endl` when you want to **flush the output immediately** (e.g. for debugging).

2.6. main() Function

Definition

`main()` is the starting point of a C++ program.

Feature

- Returns an integer
- Has no or two arguments

Syntax

```
int main() {
    // code
    return 0;
}
```

Example

- A `main()` that returns 0.

Explanation

- `return 0;` signals successful execution.

Remark

- You can also write `int main(int argc, char* argv[])`.

2.7. Comments

Definition

Comments are notes ignored by the compiler.

Feature

- Single-line uses `//`
- Multi-line uses `/* */`

Syntax

```
// This is a single-line comment

/*
This is a
multi-line comment
*/
```

Example

- Using both comment types in code.

Explanation

- Comments help explain code for humans.

Remark

- Good comments improve code readability.

3.Variables and Data Types

3.1. Declaring Variables

Definition

Declaring a variable means creating a name to store a value of a specific type in memory.

Feature

- Tells the compiler to reserve space.
- Must specify the data type.

Syntax

```
type variableName;
```

Example

```
int age;
double height;
```

Explanation

- `int age;` creates an integer variable called `age`.
- `double height;` creates a double variable called `height`.

Remark

- Uninitialized variables may contain garbage values.

3.2. Primitive Types

Definition

Primitive types are basic data types provided by C++.

Type Name

Type	Full English Name	Description / Example
`int`	Integer	Stores whole numbers. Example: `int age = 20;`
`double`	Double Precision Floating Point	Stores decimal numbers. Example: `double pi = 3.14159;`
`char`	Character	Stores a single character as its ASCII value. Example: `char letter = 'A';`
`bool`	Boolean	Stores `true` or `false`. Example: `bool isReady = true;`
`string`	String Class	Represents a sequence of characters (text). Example: `string name = "Tom";`

3.2.1. `int`

Definition

Represents integer numbers without decimal points.

Feature

- Typically 4 bytes.

Syntax

```
int x = 10;
```

Example

```
int score = 95;
```

Explanation

- Stores whole numbers like 1, -5, 100.

Remark

- Range depends on the system.

3.2.2. `double`

Definition

Represents numbers with decimal points (floating-point).

Feature

- Typically 8 bytes.

Syntax

```
double y = 5.5;
```

Example

```
double weight = 65.75;
```

Explanation

- Stores values like 3.14, -0.001.

Remark

- Use for precision with decimals.

3.2.3. `char`

Definition

Represents a single character.

Feature

- Stored as a small integer (ASCII value).

Syntax

```
char c = 'A';
```

Example

```
char grade = 'B';
```

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

int main() {
    // Example 1: Print ASCII value of a character
    char ch = 'A';
    cout << "The ASCII value of " << ch << " is " << (int)ch << endl;

    // Example 2: Print character from an ASCII value
    int num = 66;
    cout << "The character for ASCII value " << num << " is " << (char)num << endl;

    // Example 3: Print all uppercase letters with their ASCII values
    cout << "Uppercase letters and their ASCII values:" << endl;
    for (char c = 'A'; c <= 'Z'; c++) {
        cout << c << " : " << (int)c << endl;
    }

    return 0;
}
```

- output:

```
The ASCII value of A is 65
The character for ASCII value 66 is B
Uppercase letters and their ASCII values:
A : 65
B : 66
C : 67
...
```

```
Z : 90
```

Explanation

- Stores characters like 'a', '1', '$'.

Remark

- Use single quotes for characters.

What is ASCII

- ASCII (American Standard Code for Information Interchange) is a **character encoding standard** that assigns a unique numeric code to each character, digit, or symbol used in computers.

ASCII Table

Dec	Hex	Char	Name	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	null	32	20	Space	64	40	@	96	60	`
1	1	SOH	start of heading	33	21	!	65	41	A	97	61	a
2	2	STX	start of text	34	22	"	66	42	B	98	62	b
3	3	ETX	end of text	35	23	#	67	43	C	99	63	c
4	4	EOT	end of transmission	36	24	$	68	44	D	100	64	d
5	5	ENQ	enquiry	37	25	%	69	45	E	101	65	e
6	6	ACK	acknowledge	38	26	&	70	46	F	102	66	f
7	7	BEL	bell	39	27	'	71	47	G	103	67	g
8	8	BS	backspace	40	28	(	72	48	H	104	68	h
9	9	HT	horizontal tab	41	29	)	73	49	I	105	69	i
10	A	LF	line feed	42	2A	*	74	4A	J	106	6A	j
11	B	VT	vertical tab	43	2B	+	75	4B	K	107	6B	k
12	C	FF	form feed	44	2C	,	76	4C	L	108	6C	l
13	D	CR	carriage return	45	2D	-	77	4D	M	109	6D	m
14	E	SO	shift out	46	2E	.	78	4E	N	110	6E	n
15	F	SI	shift in	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	data link escape	48	30	0	80	50	P	112	70	p
17	11	DC1	device control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	device control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	device control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	device control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	synchronous idle	54	36	6	86	56	V	118	76	v
23	17	ETB	end of transmission block	55	37	7	87	57	W	119	77	w
24	18	CAN	cancel	56	38	8	88	58	X	120	78	x
25	19	EM	end of medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	file separator	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	unit separator	63	3F	?	95	5F	_	127	7F	DEL

From https://ajsmith.org/tools/ascii-table/

- Each character is represented by an integer code defined in the ASCII table. For example: `'A'` → 65, `'a'` → 97 , `'0'` → 48, `'$'` → 36

3.2.4. `bool`

Definition

Represents true or false values.

Feature

- Used for logical conditions.

Syntax

```
bool flag = true;
```

Example

```
bool isPassed = false;
```

Explanation

- `true` or `false` only.

Remark

- Useful in condition checks.

3.2.5. `string`

Definition

Represents a sequence of characters (text).

Feature

- Requires including the `<string>` library.

Syntax

```
string name = "Alice";
```

Example

```
string city = "Tokyo";
```

Explanation

- Stores words or sentences.

Remark

- Different from `char` which stores only one character.

4.Input and Output

4.1. `cout`

Definition

`cout` is used to display output on the screen.

Feature

- Comes from the `iostream` library.
- Uses insertion operator `<<`.

Syntax

```
cout << value;
```

Example

```
cout << "Hello, World!";
```

Explanation

- Prints the text inside quotes to the screen.

Remark

- Use `endl` or `\n` for new lines.

4.1.1. `printf` vs `cout`

Definition

`printf` and `cout` are both used to display output in C++, but they come from different libraries and styles.

Table: `printf` vs `cout`

Feature	printf	cout
Library	C (`<cstdio>`)	C++ (`<iostream>`)
Syntax	Uses format specifiers (`%d`)	Uses `<<` insertion operator
Type Safety	Less type-safe	More type-safe
Readability	Less readable in C++	Easier to read in C++
Buffering	C buffering mechanism	C++ stream buffering
Use Case	Precise formatted output	General C++ output

Syntax

```
// printf
printf("x is %d\n", x);

// cout
cout << "x is " << x << endl;
```

Example

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

int main() {
    int x = 10;
    printf("Using printf: x = %d\n", x);
    cout << "Using cout: x = " << x << endl;
    return 0;
}
```

Explanation

- `printf` uses format strings to specify how to print variables.

- `cout` directly inserts variables into the output stream.

Remark

- Mixing `printf` and `cout` can cause unexpected output order due to different buffer mechanisms.
- In C++ programs, prefer using `cout` for type safety and readability.

4.2. `cin`

Definition

`cin` is used to take input from the user.

Feature

- Comes from the `iostream` library.
- Uses extraction operator `>>`.

Syntax

```
cin >> variable;
```

Example

```
int age;
cin >> age;
```

Explanation

- Reads value entered by the user and stores it in `age`.

Remark

- Input is separated by spaces or newline.

4.2.1. `scanf` vs `cin`

Definition

`scanf` and `cin` are both used to take user input, but they come from different libraries and programming styles.

Table: `scanf` vs `cin`

Feature	scanf	cin
Library	C (`<cstdio>`)	C++ (`<iostream>`)
Syntax	Uses format specifiers (`%d`) with `&`	Uses `>>` extraction operator
Address-of	Requires `&` for variables	No need for `&`
Type Safety	Less type-safe	More type-safe
Readability	Less readable in C++	Easier to read in C++
Use Case	Traditional C input	Recommended in C++

Syntax

```
// scanf
int x;
scanf("%d", &x);

// cin
int x;
cin >> x;
```

Example

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

int main() {
    int age;
    double height;

    // Using scanf
    printf("Enter your age and height: ");
    scanf("%d %lf", &age, &height);

    printf("Using scanf: Age = %d, Height = %.2f\n", age, height);

    // Using cin
    cout << "Enter your age and height again: ";
    cin >> age >> height;

    cout << "Using cin: Age = " << age << ", Height = " << height << endl;

    return 0;
}
```

Explanation

- **scanf**

    - Uses format strings to specify data type (`%d` for int, `%lf` for double).
    - Requires `&` to provide the variable's memory address.
- **cin**
    - Uses `>>` to extract input directly into variables.
    - No need for `&`.

Remark

- In C++ programs, prefer `cin` for readability and type safety.
- Mixing `scanf` and `cin` in the same program may cause input issues due to different input buffer handling.

5.Control Structures

5.1. if-else Statements

Definition

A control structure that executes code blocks based on conditions.

Feature

- Checks a condition.
- Runs code in `if` block if true.
- Runs code in `else` block if false.

Syntax

```
if (condition) {
    // code if true
} else {
    // code if false
}
```

Example

```
int x = 10;
if (x > 5) {
    cout << "x is greater than 5";
} else {
    cout << "x is 5 or less";
}
```

Explanation

- Checks if `x` is greater than 5 and prints a message accordingly.

Remark

- You can use `else if` to check multiple conditions.

5.2. for Loops

Definition

A loop that repeats code a specific number of times.

Feature

- Has initialization, condition, and update.
- Runs until the condition is false.

Syntax

```
for (int i = 0; i < n; i++) {
    // code to repeat
}
```

Example

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

Explanation

- Prints numbers from 1 to 5 with spaces.

Remark

- The loop variable (`i`) increases after each iteration.

5.3. while Loops

Definition

A loop that repeats code while a condition is true.

Feature

- Checks the condition before running the code.
- Runs zero or more times.

Syntax

```
while (condition) {
    // code to repeat
}
```

Example

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

Explanation

- Prints numbers from 1 to 5 with spaces.

Remark

- Make sure to update the loop variable to avoid infinite loops.

6.Functions

6.1. Defining Functions

Definition

A block of code that performs a task and can be called when needed.

Feature

- Has a name, return type, and parameters.
- Can return a value.

Syntax

```
return_type function_name(parameters) {
    // code
    return value;
}
```

Example

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

Explanation

- Defines a function `add` that returns the sum of two integers.

Remark

- `void` is used if the function does not return a value.

6.2. Parameters and Return Types

Definition

Parameters are inputs to a function; the return type is the output type.

Feature

- Parameters are listed in parentheses.
- Return type is written before function name.

Syntax

```
int multiply(int x, int y) {
    return x * y;
}
```

Example

```
cout << multiply(2, 3);
```

Explanation

- Calls `multiply` with 2 and 3, prints 6.

Remark

- You can have multiple parameters but only one return type.

6.3.Multiple File Compilation

Definition

In C++, a program can be split into multiple source files to organize code better. These files are compiled separately and linked together to create one executable.

Feature

- Each `.cpp` file is compiled into an **object file (.o)**.
- A **linker** combines all object files into a single program.
- Code in different files can be shared using **function declarations (prototypes)** and **header files**.

6.3.1. Function Declaration vs Definition

Term	Explanation
Declaration	Tells the compiler the function **exists** and its signature, but no implementation is provided here.
Definition	Provides the full implementation (body) of the function.

Example

File1.cpp

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

void hello() { // definition
    cout << "Hello from File1" << endl;
}
```

File2.cpp

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

```cpp
void hello(); // declaration (prototype)

int main() {
    hello(); // calls File1's hello function
    return 0;
}
```

Explanation

- `File2.cpp` declares `void hello();` to let the compiler know it exists.
- During linking, it finds the actual implementation in `File1.cpp`.

6.3.2.Header Files

Usage

To avoid writing declarations repeatedly, header files (.h) are used.

Example

hello.h

```cpp
void hello(); // function declaration
```

File1.cpp

```cpp
#include "hello.h"
#include <iostream>
using namespace std;

void hello() { // definition
    cout << "Hello from File1" << endl;
}
```

File2.cpp

```cpp
#include "hello.h"
#include <iostream>
using namespace std;

int main() {
    hello(); // calls hello from File1
    return 0;
}
```

Explanation

- Both files include `hello.h` for the declaration.
- Only File1.cpp has the **definition**.
- During compilation:
  - Each .cpp → .o (object file)
  - Then linker combines all .o files into one executable

6.3.3.Compilation Process (Simplified)

Step	What Happens
Compilation	Each .cpp → .o (checks syntax, compiles code)
Linking	All .o files combined into final executable

Remark

- Without a declaration, you cannot use functions defined in other files.
- Without definition, the linker will throw an error: **undefined reference**.

7.Arrays and Strings

7.1. Arrays

Definition

An array is a collection of elements of the same type stored in continuous memory locations.

Feature

- Fixed size
- Same data type for all elements
- Index starts from 0

Syntax

```
type arrayName[size];
```

Example

```
int numbers[5];
```

Explanation

- Creates an array called `numbers` that can hold 5 integers.

Remark

- Access elements with `arrayName[index]`.

7.1.1.Declaration

Definition

Declaring an array reserves space for multiple elements.

Feature

- Must specify type and size

Syntax

```
double grades[10];
```

Example

- `double grades[10];`

Explanation

- Declares an array of 10 double values named `grades`.

Remark

- Size cannot change after declaration.

7.1.2.Accessing Elements

Definition

Accessing array elements uses indices.

Feature

- First element is at index 0

Syntax

```
arrayName[index] = value;
```

Example

```
grades[0] = 95.5;
```

Explanation

- Assigns 95.5 to the first element of `grades`.

Remark

- Using an out-of-range index causes errors.

7.2. Strings

Definition

A string is a sequence of characters.

Feature

- Can use C-style char arrays or string class
- Easier with `string` class from `<string>`

Syntax

```
string strName = "Hello";
```

Example

```
string name = "Alice";
```

Explanation

- Creates a string variable `name` with value "Alice".

Remark

- Requires `#include <string>`.

7.2.1. Using string Class

Definition

The string class allows easy handling of text.

Feature

- Supports many operations like length, append, compare

Syntax

```
string s = "text";
```

Example

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

int main() {
    // 1. Declare and initialize strings
    string s1;                  // Default initialization, empty string
    string s2 = "Hello";        // Initialize with C-style string
    string s3("World");         // Constructor initialization
    string s4 = s2 + " " + s3;  // String concatenation

    // 2. String length and empty check
    cout << "s4: " << s4 << endl;              // Output: Hello World
    cout << "Length of s4: " << s4.length() << endl;  // Output: 11
    cout << "Is s1 empty? " << (s1.empty() ? "Yes" : "No") << endl;  // Output: Yes

    // 3. Access characters in the string
    cout << "First character of s4: " << s4[0] << endl;  // Output: H
    cout << "Last character of s4: " << s4[s4.length()-1] << endl;  // Output: d

    // 4. Modify the string
    s4[6] = 'w';  // Modify the 7th character (index 6)
    cout << "Modified s4: " << s4 << endl;  // Output: Hello world

    // 5. String comparison
    string s5 = "Hello world";
    cout << "s4 == s5? " << (s4 == s5 ? "Yes" : "No") << endl;  // Output: Yes

    // 6. Substring
    string sub = s4.substr(6, 5);  // Start at index 6, length 5
    cout << "Substring: " << sub << endl;  // Output: world
```

```
    // 7. Search and replace
    size_t pos = s4.find("world");  // Find position of substring
    if (pos != string::npos) {
        s4.replace(pos, 5, "WORLD");  // Replace substring
    }
    cout << "After replacement: " << s4 << endl;  // Output: Hello WORLD

    // 8. String to/from number conversion
    string numStr = "12345";
    int num = stoi(numStr);  // String to integer
    cout << "Converted number: " << num + 1 << endl;  // Output: 12346

    string newNumStr = to_string(num * 2);  // Integer to string
    cout << "New string: " << newNumStr << endl;  // Output: 24690

    return 0;
}
```

Explanation

- Defines a string called `greeting` with value "Hi".

Remark

- Part of the C++ Standard Library.

7.2.2.Common Methods

Definition

Methods that can be used with string objects.

Feature

- `.length()`, `.append()`, `.substr()`

Syntax

```
s.length();
s.append(" world");
```

Example

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

```cpp
int main() {
    // Initialize a sample string
    string message = "Hello";
    cout << "Original string: " << message << endl;

    // 1. .length() - Returns the number of characters in the string
    size_t len = message.length();
    cout << ".length():\n";
    cout << "  Length of '" << message << "': " << len << " characters\n";

    // 2. .append() - Adds characters to the end of the string
    cout << "\n.append():\n";

    // Method 1: Append another string
    message.append(" World");
    cout << "  After appending ' World': " << message << endl;

    // Method 2: Append a C-style string
    message.append("! Welcome", 9); // Append first 9 characters of "! Welcome"
    cout << "  After appending '! Welcome' (first 9 chars): " << message << endl;

    // Method 3: Append individual characters
    message.append(3, '.'); // Append 3 dots ('.')
    cout << "  After appending 3 dots: " << message << endl;

    // 3. .substr() - Returns a substring from the original string
    cout << "\n.substr():\n";

    // Method 1: substr(pos, length)
    string substr1 = message.substr(6, 5); // Start at index 6, length 5
    cout << "  message.substr(6, 5): '" << substr1 << "'\n";

    // Method 2: substr(pos) - Extracts from pos to the end
    string substr2 = message.substr(12); // Start at index 12 to the end
    cout << "  message.substr(12): '" << substr2 << "'\n";

    return 0;
}
```

Explanation

- Adds " there" to `s`, making it "Hi there".

Remark

- Methods make string operations simpler than C-style char arrays.

8.Classes and Objects

8.1. Defining Classes

Definition

A class is a blueprint for creating objects with data and functions.

Feature

- Has members: variables and functions

Syntax

```
class ClassName {
public:
  // members
};
```

Example

```
class Car {
public:
  int speed;
};
```

Explanation

- Defines a class `Car` with an integer `speed`.

Remark

- `public` makes members accessible outside the class.

8.2. Constructors

Definition

A constructor initializes objects when they are created.

Feature

- Same name as class
- No return type

Syntax

```
ClassName() {
  // initialization
}
```

Example

```
Car() {
  speed = 0;
}
```

Explanation

- Constructor sets `speed` to 0 when a `Car` is created.

Remark

- Called automatically on object creation.

8.2.1 Member Initializer List

Definition

***Member initializer list* is a way to initialize class member variables directly when calling the constructor, before the constructor body runs.**

Feature

- Uses `:` **followed by member-variable(initializer-value)** syntax after constructor's parameter list.
- Runs **before** the constructor body.
- Improves efficiency, especially for class-type members.
- Required when:
  - Initializing **const** members.
  - Initializing **reference** members.
  - Calling **base class constructors** in inheritance.

Syntax

```
ClassName(parameters) : member1(value1), member2(value2) {
    // constructor body (optional)
}
```

Explanation of Syntax

1. `ClassName(parameters)` – constructor definition with parameters.
2. `: member1(value1), member2(value2)` – initializes `member1` with `value1` and `member2` with `value2` **before entering constructor body**.
3. `{ ... }` – constructor body. May be empty if all initialization is done in the list.

Example

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

class Complex {
public:
    int real, imag;

    Complex(int r=0, int i=0) : real(r), imag(i) {}

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3);
    c1.display(); // 2 + 3i
    return 0;
}
```

Explanation

- `Complex(int r=0, int i=0)` is the constructor with default values.
- `: real(r), imag(i)` is the **member initializer list**. It:
    - Sets `real = r`
    - Sets `imag = i`

  before the constructor body runs.
- `{}` is the empty constructor body since all initialization is done.

Remark

- For **basic types (int, double)**, member initializer list or assignment inside constructor has similar effect.
- For **class members, const members, references**, always use initializer lists.

8.3. `this` Pointer

Definition

`this` **is a pointer to the current object.**

Feature

- Used to refer to object's own members

Syntax

```
this->memberName;
```

Example

```
void setSpeed(int speed) {
  this->speed = speed;
}
```

Explanation

- Differentiates between parameter `speed` and member `speed`.

Remark

- Useful when parameter names are same as member names.

8.4. Methods

Definition

Functions defined inside a class are called methods.

Feature

- Operate on class data

Syntax

```
returnType methodName() {
  // code
}
```

Example

```
void display() {
  cout << speed;
}
```

Explanation

- Method `display` prints the `speed`.

Remark

- Can be called using an object.

8.5. Creating Objects

Definition

Objects are instances of classes.

Feature

- Have their own copies of members

Syntax

```
ClassName objName;
```

Example

```
Car myCar;
```

Explanation

- Creates an object `myCar` of class `Car`.

Remark

- Members can be accessed using dot operator: `myCar.speed`.

9.Pointers and References

9.1. Pointers

Definition

A pointer is a variable that stores the address of another variable.

Feature

- Points to memory location
- Uses `*` for declaration and dereferencing
- Uses `&` to get address

Syntax

```
int *ptr; // declaration
ptr = &x; // assigning address
```

Example

```
int x = 10;
int *p = &x;
cout << *p; // prints 10
```

Explanation

- `*p` accesses the value at the address stored in `p`.

Remark

- Pointers are powerful but can cause errors if used incorrectly.

9.2. References

Definition

A reference is an alias for another variable.

Feature

- Uses `&` in declaration
- Must be initialized when declared
- No need for dereferencing

Syntax

```
int x = 5;
int &ref = x;
```

Example

```
int x = 5;
int &y = x;
y = 10;
cout << x; // prints 10
```

Explanation

- Changing `y` changes `x` because `y` refers to `x`.

Remark

- References are safer and easier than pointers.

9.3. swap Example

Definition

Using reference parameters to swap two variables.

Feature

- No need to return values
- Changes original variables

Syntax

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Example

```
int x = 3, y = 4;
swap(x, y);
```

```
// x is 4, y is 3
```

Explanation

- `a` and `b` are references to `x` and `y`, so swapping affects them directly.

Remark

- Common use of references in functions.

10.Dynamic Memory

**10.1. `new` and `delete` Operators

Definition

Dynamic memory allows you to allocate and free memory during program execution instead of defining all memory at compile time.

Feature

- Memory can be allocated at runtime based on user input or program needs.
- Prevents wastage of unused memory.
- Gives flexibility to create data structures like dynamic arrays, linked lists.

Syntax

```
// Allocate memory for one int
int* ptr = new int;

// Deallocate memory
delete ptr;

// Allocate memory for an array of ints
int* arr = new int[5];

// Deallocate array
delete[] arr;
```

Example

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

int main() {
```

```
    int* num = new int; // allocate
    *num = 10;
    cout << "Value: " << *num << endl;
    delete num; // free

    int size;
    cout << "Enter size: ";
    cin >> size;
    int* arr = new int[size]; // dynamic array

    for(int i = 0; i < size; ++i) {
        arr[i] = i * 2;
    }

    for(int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }

    delete[] arr; // free array
    return 0;
}
```

Explanation

- `new` allocates memory from the heap and returns its address.
- `delete` frees the memory to avoid memory leaks.
- Use `delete[]` to free arrays allocated with `new[]` .

Remark

- Always match `new` with `delete` and `new[]` with `delete[]` .
- Forgetting `delete` causes memory leaks.
- Using `delete` on memory not allocated with `new` causes undefined behavior.

11.STL (Standard Template Library)

11.1. STL (Standard Template Library)

Definition

The Standard Template Library (STL) provides common classes and functions for data structures and algorithms in C++.

Feature

- Ready-to-use containers like `vector` , `map` , and `set` .
- Generic and type-safe with templates.
- Reduces coding time and errors.

11.2. vector

Definition

`vector` is a dynamic array that can change size automatically.

Feature

- Stores elements in contiguous memory.
- Automatically resizes when elements are added or removed.
- Provides random access with index.

Syntax

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

vector<int> v;
v.push_back(1);
v.push_back(2);
int x = v[0];
```

Example

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

int main() {
    vector<int> numbers;
    numbers.push_back(5);
    numbers.push_back(10);

    for(int i = 0; i < numbers.size(); ++i) {
        cout << numbers[i] << " ";
    }
    return 0;
}
```

Explanation

- `push_back` adds an element at the end.
- `size` returns the number of elements.
- Access elements with `[]` or `at()`.

Remark

- Prefer `vector` over raw arrays for dynamic lists.
- `vector` manages memory automatically.

11.3. map

Definition

`map` stores key-value pairs with unique keys.

Feature

- Each key is unique.
- Automatically sorted by key.
- Fast lookup, insertion, and deletion.

Syntax

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

map<string, int> ages;
ages["Alice"] = 25;
ages["Bob"] = 30;
int age = ages["Alice"];
```

Example

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

int main() {
    map<string, int> phoneBook;
    phoneBook["Tom"] = 1234;
    phoneBook["Jerry"] = 5678;

    cout << "Tom's number: " << phoneBook["Tom"] << endl;

    for(auto it = phoneBook.begin(); it != phoneBook.end(); ++it) {
        cout << it->first << ": " << it->second << endl;
    }
    return 0;
}
```

Explanation

- Access or insert with `[]`.

- `begin()` and `end()` allow iteration.
- Keys are unique; adding a key replaces the old value.

Remark

- Use `map` when you need fast lookups by key.
- For multiple values with the same key, use `multimap`.

11.4 set

Definition

`set` stores unique elements in sorted order.

Feature

- Each element appears only once.
- Elements are automatically sorted.
- Supports fast search and insertion.

Syntax

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

set<int> s;
s.insert(5);
s.insert(10);
bool exists = s.find(5) != s.end();
```

Example

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

int main() {
    set<int> numbers;
    numbers.insert(3);
    numbers.insert(1);
    numbers.insert(3); // duplicate ignored

    for(int num : numbers) {
        cout << num << " ";
    }

    if(numbers.find(1) != numbers.end()) {
```

```
        cout << "\n1 exists in the set.";
    }
    return 0;
}
```

Explanation

- `insert` adds an element if not present.
- `find` returns an iterator to the element or `end()` if not found.
- Elements are always sorted.

Remark

- `set` is useful for storing unique elements.
- For multiple identical elements, use `multiset`.

12.Templates

12.1. Function Templates

Definition

Function templates allow writing functions that work with any data type.

Feature

- Generic programming
- One function for multiple types

Syntax

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Example

- `add<int>(3,4);`

Explanation

- Replace `T` with actual type when calling.

Remark

- Use `template` keyword to define.

12.2. Class Templates

Definition

Class templates allow classes to work with any data type.

Feature

- Generic classes
- Type decided at object creation

Syntax

```
template <class T>
class Box {
  int num;
  T value;
};
```

Example

- `Box<int> b;`

Explanation

- Define type inside `<>` when creating objects.

Remark

- Class templates improve code reuse.

12.3. Designing Generic Functions with Templates in C++

Definition

Designing generic functions with templates in C++ involves creating functions that operate on different data types using template parameters, generating type-specific code at compile time.

Feature

- Templates enable type-agnostic function definitions.
- Compile-time code generation ensures performance without runtime overhead.
- Concepts (C++20) can constrain template types for additional safety.

Syntax

```
template <typename T>
int index_of(T* array, int size, T element) {
    for (int i = 0; i < size; i++) {
        if (array[i] == element) {
            return i;
        }
    }
    return -1;
}
```

Example

```
#include <iostream>
#include <string>

template <typename T>
int index_of(T* array, int size, T element) {
    for (int i = 0; i < size; i++) {
        if (array[i] == element) {
            return i;
        }
    }
    return -1;
}

int main() {
    int numbers[] = {10, 20, 30, 40};
    std::string words[] = {"apple", "banana", "cherry"};
    std::cout << index_of(numbers, 4, 30) << std::endl; // Outputs 2
    std::cout << index_of(words, 3, std::string("banana")) << std::endl; // Outputs 1
    std::cout << index_of(numbers, 4, 50) << std::endl; // Outputs -1
    return 0;
}
```

Explanation

- template `<typename T>` declares a type parameter T for the function.
- The function index_of takes a C-style array, its size, and an element, using == for comparison.
- The compiler generates type-specific code for each type used (e.g., int, std::string).
- The function returns the first index of the element or -1 if not found.

Remark

- Templates offer high performance due to compile-time code generation.
- Unlike Java generics, C++ templates support both primitives and custom types.
- Concepts (e.g., requiring T to support `==`) can improve error messages and type safety.

13.File I,O

13.1 Reading Files (ifstream)

Definition

`ifstream` stands for input file stream, and it is a data type in C++ used to read data from files on your computer into your program.

Feature

- Used to **read** (input) data from files.
- You must include the header file `<fstream>` to use `ifstream`.
- Before reading, you need to **open** the file. This can be done:
  - Directly when creating `ifstream` object (e.g. `ifstream inFile("file.txt");`), or
  - By calling `.open("file.txt")` on an existing `ifstream` object.
- After reading, you should **close** the file using `.close()` to free resources.

Syntax

```cpp
#include <fstream>

ifstream inFile("input.txt"); // open file when creating inFile
inFile >> data; // read data from file
inFile.close(); // close the file
```

Explanation of Syntax

1. `#include <fstream>` allows use of file stream classes (`ifstream` and `ofstream`).
2. `ifstream inFile("input.txt");` creates an input file stream named `inFile` and opens the file `input.txt` for reading.
3. `inFile >> data;` reads data from the file into the variable `data`.
4. `inFile.close();` closes the file after reading is finished to avoid memory/resource leaks.

Example

```cpp
#include <fstream>
#include <iostream>
```

```cpp
using namespace std;

int main() {
    ifstream inFile("data.txt"); // open data.txt for reading
    int num;
    if (inFile.is_open()) { // check if file opened successfully
        while (inFile >> num) { // read integers one by one from file into num
            cout << num << endl; // print each number to console
        }
        inFile.close(); // close the file after reading
    } else {
        cout << "Failed to open the file." << endl;
    }
    return 0;
}
```

Detailed Explanation

- `ifstream inFile("data.txt");` tries to open `data.txt` immediately.
- `inFile.is_open()` checks if the file is successfully opened.
- `while (inFile >> num)` keeps reading integers from the file until it reaches end of file.
- Inside the loop, `cout << num << endl;` prints each read number on a new line.
- `inFile.close();` closes the file properly.

Remark

- Always check `is_open()` before reading to avoid reading from an unopened or non-existing file, which may cause program errors.

13.2 Writing Files (ofstream)

Definition

`ofstream` stands for output file stream, and it is a data type in C++ used to write data from your program to files on your computer.

Feature

- Used to **write** (output) data to files.
- You must include the header file `<fstream>` to use `ofstream`.
- Before writing, you need to **open** the file. This can be done:
    - Directly when creating `ofstream` object (e.g. `ofstream outFile("file.txt");`), or
    - By calling `.open("file.txt")` on an existing `ofstream` object.
- After writing, you should **close** the file using `.close()` to ensure all data is saved properly.
- If the file does not exist, it will be **created automatically**. If it exists, its contents will be **overwritten** (erased) unless you open it in append mode (using an additional parameter).

Syntax

```
#include <fstream>

ofstream outFile("output.txt"); // open file when creating outFile
outFile << data; // write data to file
outFile.close(); // close the file
```

Explanation of Syntax

1. `#include <fstream>` allows use of file stream classes (`ifstream` and `ofstream`).
2. `ofstream outFile("output.txt");` creates an output file stream named `outFile` and opens `output.txt` for writing.
3. `outFile << data;` writes the value of `data` into the file.
4. `outFile.close();` closes the file after writing to ensure data is properly saved.

Example

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

int main() {
    ofstream outFile("result.txt"); // open result.txt for writing
    if (outFile.is_open()) { // check if file opened successfully
        outFile << "Hello file" << endl; // write string into the file
        outFile << 123 << endl; // write number into the file
        outFile.close(); // close the file after writing
    } else {
        cout << "Failed to open the file." << endl;
    }
    return 0;
}
```

Detailed Explanation

- `ofstream outFile("result.txt");` tries to open `result.txt` immediately. If it does not exist, it will be created.
- `outFile.is_open()` checks if the file is successfully opened.
- `outFile << "Hello file" << endl;` writes the string `"Hello file"` followed by a newline to the file.
- `outFile << 123 << endl;` writes the number `123` followed by a newline to the file.
- `outFile.close();` closes the file properly after writing.

Remark

- **Closing the file is important**. Without `.close()`, some data may not be saved because it is kept in the buffer (temporary storage).

14.Exception Handling

14.1. try-catch Blocks

Definition

A `try-catch` block is a structure in C++ used to **handle errors (exceptions) that happen during program execution (runtime)**. It allows you to write special code to deal with errors instead of letting the program crash immediately.

Feature

- The `try` **block** contains code that **might cause an error** (throw an exception).
- The `catch` **block** is used to **catch and handle the error** thrown by the `try` block.
- This structure **prevents the entire program from crashing** when an error occurs, because the error is handled properly.

Syntax

```
try {
    // code that may throw an exception
} catch (exceptionType variableName) {
    // code to handle the exception
}
```

Explanation of Syntax

1. `try { ... }` encloses code that **might throw an exception**.
2. `catch (exceptionType variableName) { ... }` catches the thrown exception and **executes code to handle it**.

- `exceptionType` is the **data type of the exception thrown**, such as `int`, `double`, `const char*`, `std::exception`, etc.
- `variableName` is a **name you choose** to refer to the caught exception **inside the catch block**. It is just a **parameter name** like a function parameter.

Example

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

```cpp
int main() {
    try {
        int a = 10, b = 0;
        if (b == 0)
            throw "Division by zero!";
        cout << a / b << endl;
    } catch (const char* msg) {
        cout << "Error: " << msg << endl;
    }
    return 0;
}
```

Detailed Explanation

1. `int a = 10, b = 0;`
   Defines two integers `a` (value 10) and `b` (value 0).
2. `if (b == 0) throw "Division by zero!";`
   Checks if `b` is zero. Since dividing by zero is an error, it **throws an exception** here.
   - `throw "Division by zero!";`
     Throws a **string literal** (of type `const char*`) as an exception. This string `"Division by zero!"` describes the error.
3. `catch (const char* msg)`
   Catches exceptions of type `const char*`.
   - `msg` is a **variable name** (parameter) that stores the exception caught.
   - Here, `msg` will have the value `"Division by zero!"`.
4. `cout << "Error: " << msg << endl;`
   Prints the error message to the console.

What is `msg`?

`msg` is just a name you choose to refer to the caught exception value inside the catch block.

In this example:

- `throw "Division by zero!";` throws a value of type `const char*` (a pointer to a string literal).
- `catch (const char* msg)` catches this exception and stores it in the variable `msg`.
- So, `msg` is now equal to `"Division by zero!"`.

Remark

- Using `try-catch` **makes your program safer**, because instead of crashing when an error happens, you can handle it gracefully (e.g. show an error message to the user and continue running or exit cleanly).

15.1. Operator Overloading

Definition

Operator overloading allows you to redefine how an operator works for user-defined types (classes/structs) to make code intuitive and natural.

Feature

- Improves readability and usability of your classes
- Most operators can be overloaded (e.g. `+`, `-`, `*`, `/`, `=`, `==`, `<`, `>`, `[]`, `()`, `->`, etc.)
- Some operators **cannot** be overloaded: `::`, `.`, `.*`, `sizeof`, `typeid`

Syntax

```
return_type operator symbol (parameters) {
    // implementation
}
```

As member function

- First operand is the calling object (`this`).

As non-member function (friend function)

- Both operands are passed as parameters.

Example 1 – Overloading '+' for Complex Numbers (Member Function)

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

class Complex {
public:
    int real, imag;

    Complex(int r=0, int i=0) : real(r), imag(i) {}

    Complex operator + (const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2,3), c2(4,5);
```

```
    Complex c3 = c1 + c2;
    c3.display(); // 6 + 8i
    return 0;
}
```

Explanation

- The `+` operator is overloaded to add two Complex objects.
- `c1 + c2` is translated to `c1.operator+(c2)`.

Example 2 – Overloading '<<' Operator (Friend Function)

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

class Complex {
public:
    int real, imag;
    Complex(int r=0, int i=0) : real(r), imag(i) {}

    friend ostream& operator << (ostream& out, const Complex& c);
};

ostream& operator << (ostream& out, const Complex& c) {
    out << c.real << " + " << c.imag << "i";
    return out;
}

int main() {
    Complex c(3,4);
    cout << c << endl; // 3 + 4i
    return 0;
}
```

Explanation

- `<<` is overloaded as a **friend function** because `cout` is the first operand and not part of the class.
- Returns `ostream&` to support chaining (`cout << c << c2;`).

Remark

- Overloading should preserve operator meaning to avoid confusion.
- You cannot create new operators; only overload existing ones.

15.2. Namespaces (Advanced)

Definition

A namespace is a container for identifiers (like functions, classes, variables) to avoid name conflicts and organize code.

Feature

- Prevents name conflicts when multiple libraries have same function or class names
- Supports nested namespaces
- Can use aliases to simplify long namespace names

Syntax

```
namespace name {
    // declarations
}
```

Example 1 – Creating and Using a Namespace

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

namespace Math {
    int add(int a, int b) {
        return a + b;
    }
    int sub(int a, int b) {
        return a - b;
    }
}

int main() {
    cout << Math::add(5, 3) << endl; // 8
    cout << Math::sub(5, 3) << endl; // 2
    return 0;
}
```

Explanation

- `Math` is a namespace containing two functions `add` and `sub`.
- Use `Math::add()` to access them.

Remark

- You can avoid typing `Math::` repeatedly by adding:

```cpp
using namespace Math;
```

But this may cause name conflicts if other namespaces also define `add` or `sub`.

Example 2 – Nested Namespaces

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

namespace Company {
    namespace Department {
        void show() {
            cout << "Welcome to Department." << endl;
        }
    }
}

int main() {
    Company::Department::show(); // Welcome to Department.
    return 0;
}
```

Explanation

- `Department` is inside `Company`, accessed as `Company::Department::show()`.

Remark

- Since C++17, you can write nested namespaces like:

```cpp
namespace Company::Department {
    void show() {
        // implementation
    }
}
```

Example 3 – Namespace Aliases

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

namespace VeryLongNamespaceName {
    void greet() {
        cout << "Hello!" << endl;
    }
}
```

```cpp
int main() {
    namespace VLN = VeryLongNamespaceName;
    VLN::greet(); // Hello!
    return 0;
}
```

Explanation

- `namespace VLN = VeryLongNamespaceName;` creates an alias to shorten code.

Remark

- Aliases improve readability when using long namespace names frequently.

15.2.1. Anonymous Namespaces

Definition

An anonymous namespace is a namespace without a name, used to restrict functions, variables, or classes to the current file (translation unit).

Feature

- Limits visibility to only the file it is declared in (internal linkage)
- Similar purpose to `static` for global variables or functions in C

Syntax

```cpp
namespace {
    // declarations
}
```

Example

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

namespace {
    int secret = 42;

    void showSecret() {
        cout << "Secret is " << secret << endl;
    }
}
```

```cpp
int main() {
    showSecret(); // Secret is 42
    return 0;
}
```

Use Case Example – Avoiding Name Conflicts Across Files

File1.cpp

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

namespace {
    void greet() {
        cout << "Hello from File1!" << endl;
    }
}

int main() {
    greet(); // Hello from File1!
    return 0;
}
```

File2.cpp

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

namespace {
    void greet() {
        cout << "Hello from File2!" << endl;
    }
}

int main() {
    greet(); // Hello from File2!
    return 0;
}
```

- Both files define a function `greet()` but they do **not** conflict, because each is in its own anonymous namespace limited to that file.

Explanation

- The variable `secret` and function `showSecret` can only be used in this file.
- If another file declares `int secret;` globally, there is no conflict because anonymous namespaces have internal linkage.

Remark

- This is preferred over `static` for global functions or variables in modern C++ because it works for all declarations, including classes and templates.

15.2.2. Extending Namespaces Across Files

Definition

A namespace can be defined in multiple files, and all declarations are treated as belonging to the same namespace.

Feature

- Organizes code modules into a single logical namespace
- Useful for large projects split into many files

Syntax

File1.cpp

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

namespace Project {
    void func1() {
        cout << "This is func1 in File1" << endl;
    }
}
```

File2.cpp

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

namespace Project {
    void func2() {
        cout << "This is func2 in File2" << endl;
    }
}
```

Explanation

- Both `func1` and `func2` are part of namespace `Project`, even though declared in different files.
- In a combined build, you can use them together:

Main.cpp

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

```cpp
namespace Project {
    void func1(); // declaration
    void func2(); // declaration
}

int main() {
    Project::func1(); // This is func1 in File1
    Project::func2(); // This is func2 in File2
    return 0;
}
```

Remark

- Ensure each function has a declaration visible where called (via headers) or include their definitions directly.
- This practice supports modular programming, keeping related code grouped logically.

15.3. Multiple Paradigm Support

Definition

C++ supports multiple programming paradigms: **procedural, object-oriented, and generic programming** in the same language.

Feature

- **Procedural Programming**: Writing functions and processing data sequentially (like C).
- **Object-Oriented Programming (OOP)**: Organizing code using classes, objects, encapsulation, inheritance, and polymorphism.
- **Generic Programming**: Writing code that works for any data type using templates.

Example 1 – Procedural Programming

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

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

int main() {
    cout << add(2,3) << endl; // 5
    return 0;
}
```

Example 2 – Object-Oriented Programming

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

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(4,5) << endl; // 9
    return 0;
}
```

Example 3 – Generic Programming (Templates)

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

template <class T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(1,2) << endl;       // 3 (int)
    cout << add(2.5,3.1) << endl;   // 5.6 (double)
    return 0;
}
```

Explanation

- Templates allow the same code to work for multiple types without rewriting.

Remark

- The ability to mix paradigms is what makes C++ powerful and suitable for both **low-level system programming** and **high-level application development**.