UNIT - 4

TEMPLATE

📌 What is a Function Template?

A function template allows you to write a single function that works with different data types (int, float, char, etc.), without rewriting the same code multiple times.

👉 It is part of generic programming in C++.

📌 Syntax of Function Template

template <typename T>

T functionName(T a, T b) {

// function body

}

template <typename T> → tells the compiler this is a template

T → a placeholder for a data type (can be int, float, char, etc.)

You can also use class instead of typename.

📌 Simple Example: Function Template

#include <iostream>

using namespace std;

// Function Template

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << "Sum of integers: " << add(10, 20) << endl; // int

cout << "Sum of floats: " << add(5.5, 2.3) << endl; // float

cout << "Sum of chars: " << add('A', 'B') << endl; // char (works with ASCII values)

return 0;

}

📌 Output

Sum of integers: 30

Sum of floats: 7.8

Sum of chars: 131

📌 Key Points

1. Function templates help in code reusability.

2. The compiler generates the function code for the type used at compile time (called template instantiation).

3. You can also use multiple parameters like:

template <typename T1, typename T2>

auto add(T1 a, T2 b) {

return a + b;

}

📌 Function Template Overloading

Just like normal functions, a function template can also be overloaded by:

1. Defining multiple function templates with different parameters

2. Defining a normal function along with a template function.

👉 The compiler decides which version to call based on the arguments.

📌 Example 1: Overloading with Different Parameters

#include <iostream>

using namespace std;

// Function template with one parameter

template <typename T>

void display(T x) {

cout << "Template with one parameter: " << x << endl;

}

// Overloaded function template with two parameters

template <typename T>

void display(T x, T y) {

cout << "Template with two parameters: " << x << ", " << y << endl;

}

int main() {

display(10); // Calls single parameter version

display(10, 20); // Calls two parameter version

display(3.5, 7.2); // Works for floats too

return 0;

}

✅ Output:

Template with one parameter: 10

Template with two parameters: 10, 20

Template with two parameters: 3.5, 7.2

📌 Example 2: Normal Function + Template Function

#include <iostream>

using namespace std;

// Normal function (non-template)

void show(int x) {

cout << "Normal function for int: " << x << endl;

}

// Function template

template <typename T>

void show(T x) {

cout << "Template function: " << x << endl;

}

int main() {

show(100); // Normal function gets priority for int

show(12.5); // Template function for double

show('A'); // Template function for char

return 0;

}

✅ Output:

Normal function for int: 100

Template function: 12.5

Template function: A

📌 What is a Class Template?

A class template allows you to create a blueprint for a class that can work with any data type.

👉 It’s like a function template, but for classes.

Instead of writing separate classes for int, float, double, etc., you can write a single generic class.

📌 Syntax

template <typename T>

class ClassName {

T data;

public:

ClassName(T d) {

data = d;

}

void display() {

cout << "Data: " << data << endl;

}

};

template <typename T> → T is a placeholder for a data type.

When creating objects, you specify the type (int, float, etc.).

📌 Example: Class Template

#include <iostream>

using namespace std;

// Class Template

template <typename T>

class Box {

T value;

public:

Box(T v) {

value = v;

}

void display() {

cout << "Value: " << value << endl;

}

};

int main() {

Box<int> b1(10); // int type

Box<double> b2(5.5); // double type

Box<char> b3('A'); // char type

b1.display();

b2.display();

b3.display();

return 0;

}

📌 Output

Value: 10

Value: 5.5

Value: A

📌 Class Template with Multiple Parameters

You can also create a class template with two or more types:

#include <iostream>

using namespace std;

template <typename T1, typename T2>

class Pair {

T1 first;

T2 second;

public:

Pair(T1 a, T2 b) {

first = a;

second = b;

}

void show() {

cout << "First: " << first << ", Second: " << second << endl;

}

};

int main() {

Pair<int, double> p1(10, 3.14);

Pair<string, char> p2("Hello", 'X');

p1.show();

p2.show();

return 0;

}

📌 Output

First: 10, Second: 3.14

First: Hello, Second: X