SWEG2102 Fall 2026

Fundamentals of Programming II

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

Lecture 11

Generic Programming and Templates

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

Learning Objectives

By the end of this lecture, you will be able to:

- **Explain** the concept of generic programming and its advantages.
- **Define and use function** templates and class templates
- **Differentiate** between regular functions/classes and templates.
- **Apply** template specialization for customized behavior.
- **Design** reusable and flexible programs using templates in C++ programs.

Introduction to Generic Programming

Generic Programming

Core Concepts

Generic Programming is a programming style where you **write code** that works with **different data types**.

- *Writing algorithms (logics) with* ***type abstraction (placeholder types)*** *instead of* instead of rewriting the same logic for each type or specific ones.
- *In simple term,* ***write the logic/algorithm once****, reuse it for* ***any type****.*

Core Idea

Instead of writing like this:

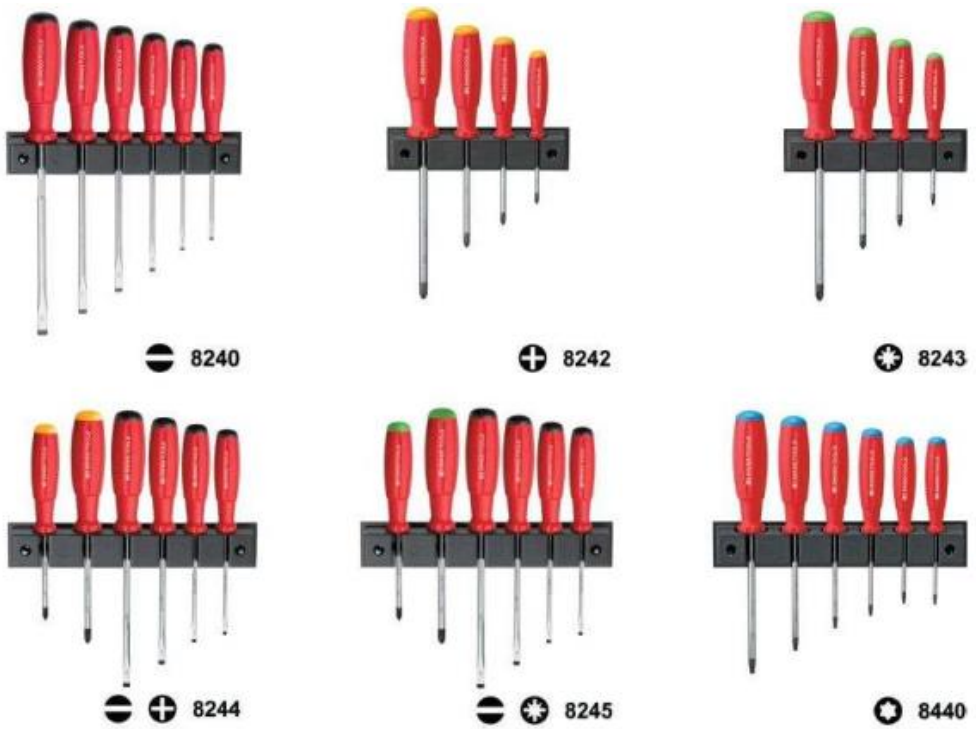
```
int add( int a, int b){
    return a+b;
}
double add( double a, double b){
    return a+b;
}
char add( char a, char b){
    return a+b;
}
```

Write generic version like this:

```
template <typename T>

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

Cont’d

Real world Scenario



- Is **programming** any different?

Cont'd

Can we manage in a better way?

- **Yes, like this.**

How we do this in **Programming,**

writing an **algorithm** or **business logic** independent of data-types?

Using Generic Programming

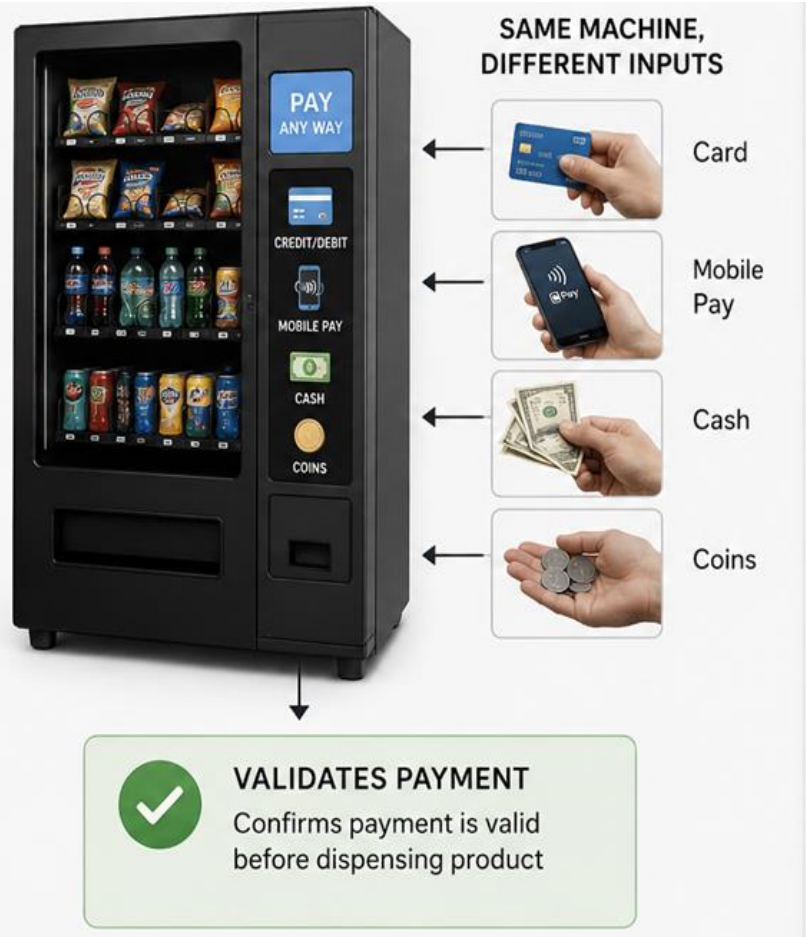
✓ Make implementations as **general as possible.**

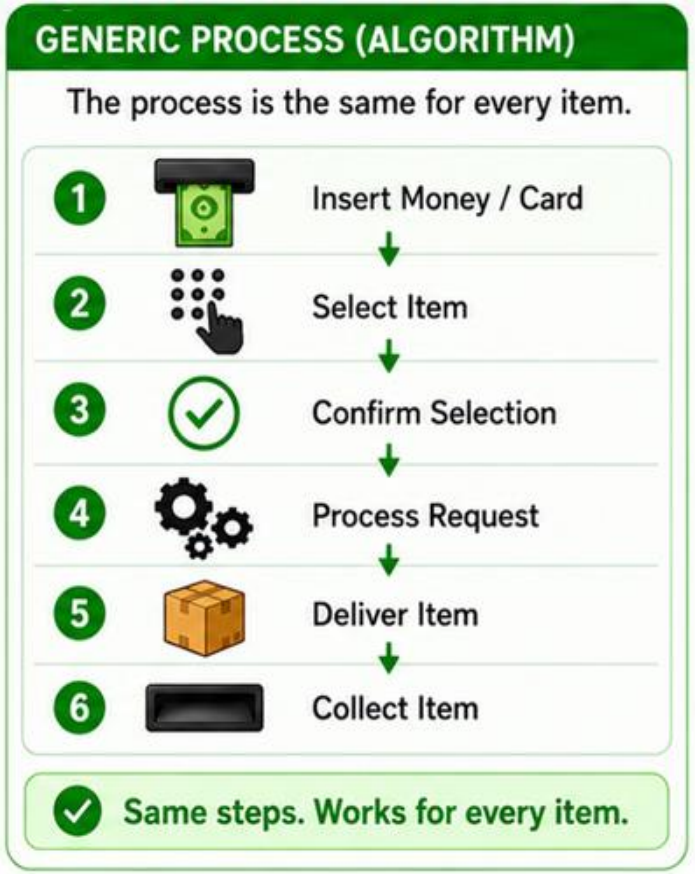
Cont’d

Real-World Analogy: A Vending Machine

Think of a vending machine that accepts coins, bills, or cards

- *it doesn't care what payment method you use; it just needs some valid payment.*





Cont'd . . .

Key Benefits

Type Independence:

- *Code logic works with multiple data types*

Reusability:

- *One implementation for many types*

Type Safety:

- *Compiler checks types at compile time*
- *No runtime overhead*

Flexibility:

- *Can operate on user-defined types*

Maintainability:

- *Less duplicate code, easily maintainable program*

Without templates:

- ❌ Separate functions for every type
- ❌ Code duplication
- ❌ Hard to maintain
- ❌ Bug must be fixed in every version

With Templates:

- ✅ One function
- ✅ Works for all types
- ✅ Fix once, fixed everywhere

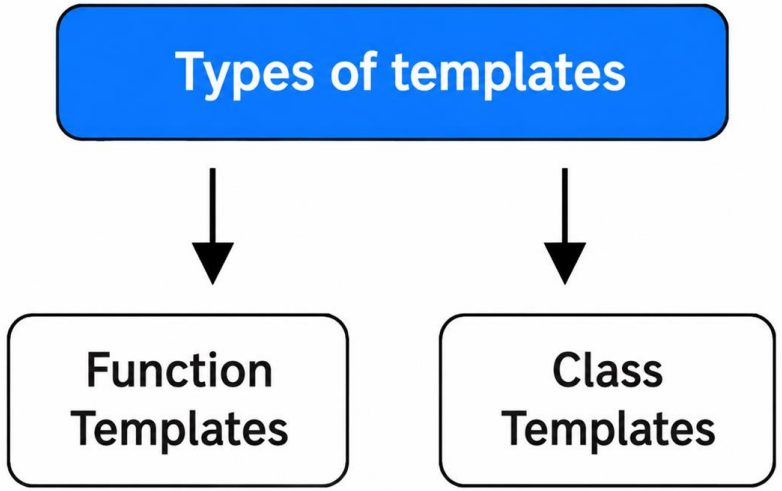
Cont'd . . .

Template

Template is a **tool** provided by the C++ language to write a **common function/class** that is independent of a data type.

- It defined as a **blueprint or formula** for creating a **generic functions** or **classes**.
- It serves as a **placeholder** for a data type and is not a data type itself.
- It embodies a **common algorithm** or **implementation** that can work with different data types.

In C++ templates come in two flavors



Function and Class Templates

2.1 Function Template

What is a Function Template?

- A **special functions** that can operate with **generic data-types.**
- It is a **blueprint**, not an actual function.
- Can be adapted to **multiple type/class** through the **template parameters** without repeating the entire code for each type.
- The **compiler** automatically creates the **correct function version** when the it used or called with a specific data-type.

General Syntax

```
template <typename T>
T biggest (T arg1, T arg2)
{
   //function body,
   //generic algorithms
}
```

Note:

- *template defines a **generic function**, and typename T is a placeholder type.*

Cont'd

Description of Syntax

```
template <typename T>
T biggest (T arg1, T arg2) {
    //function body,
    //generic algorithms
}
```

1. template:

- *A keyword indicating a template definition*
- *Tells the compiler that this is a template, and to treat it differently from a normal function*

2. <typename T>:

- ***Generic** type definition and declares the type parameter(s)*
- *keyword **class** can be used instead of typename*
- *It can be two or more*

< typename T, typename U, . . . >

3. T - type parameters :

- *It is **a placeholder** replaced by the **actual type** when the function is called*
- *Identifier **T** is most often used, but any identifier is acceptable.*
- *The parameters **arg1** and **arg2** are type **T***

Cont'd

Example 1:

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

// Function template to swap two values
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

// Function template to return the larger value
template <typename T>
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}
```

```cpp
int main() {
    int x = 10, y = 20;
    swapValues(x, y);                          // T = int
    cout << "x=" << x << " y=" << y << endl;   // x=20 y=10

    double p = 3.5, q = 7.2;
    cout << maxVal(p, q) << endl;        // T = double → Output: 7.2

    string s1 = "apple", s2 = "mango";
    cout << maxVal(s1, s2) << endl;      // T = string → Output: mango
}
```

Compiler auto-generates:

- swapValues<int>(int&, int&)
- maxVal<double>(double, double)
- maxVal<string>(string, string)

Cont'd

Function Templates Vs. Function Overloading

Aspect	Function Overloading	Function Template
Core Concept	Multiple separate functions	One generic definition
Type handling	Every type handled explicitly	Automatic handling for any type
Code size	Grows with data types	Fixed – no code grow with types
Custom behavior	Each function can be different	Same logic for all types
When to use	Different behavior per data type	Same behavior for any data type
Compile-time	Each version written explicitly	Compiler instantiates based on type

Rule Of Thumb:

Same logic, different types → ***Function Template***

Different logic per type → ***Function Overloading***

2.2 Class Template

Core Concepts

A **class template** refers to a **generic class** that can work with different data types without rewriting it for each type.

- *It serves as a **blueprint** for generating classes at-compile time.*
- *The data type is specified **during object creation**.*
- *A **single class template** can instantiated with **different types**.*

General Syntax

```
template < class T>
class ClassName {
    // Class members
};
```

or

```
template < typename T>
class ClassName {
    // Class members
};
```

Example

```
template < typename T>
class Box {
    private:
        T value;
    public:
        Box(T v)  {
            value = v;
        }
        void display()  {
            cout << "\nValue = " << value;
        }
};
```

Cont’d

Why Class Template?

👉 With Class Templates:

```
template < typename T>
class Box {
   private:
      T value;

   public:
      void set(T v) {   value = v;  }
      T get()  {   return value;  }
};
```

✅ *One class for all types*

👉 Code Without Class Templates:

```
class IntBox {
    private:
        int value;
    public:
        void set( int v) {   value = v;  }
};
class DoubleBox {
    private:
        double value;
    public:
        void set( double v)  {  value = v;  }
};
```

Two classes,
the same structure
and **similar behavior**;
only the **data types**
are different.

2. Class Template

Creating Template Objects

- A **template object** is an **instance** of a class template.
- A class template becomes a real class only when a data type is provided.
- The **actual data type** that will provided either *explicitly by programmer* or *automatically deduced by the compiler*.

General Syntax

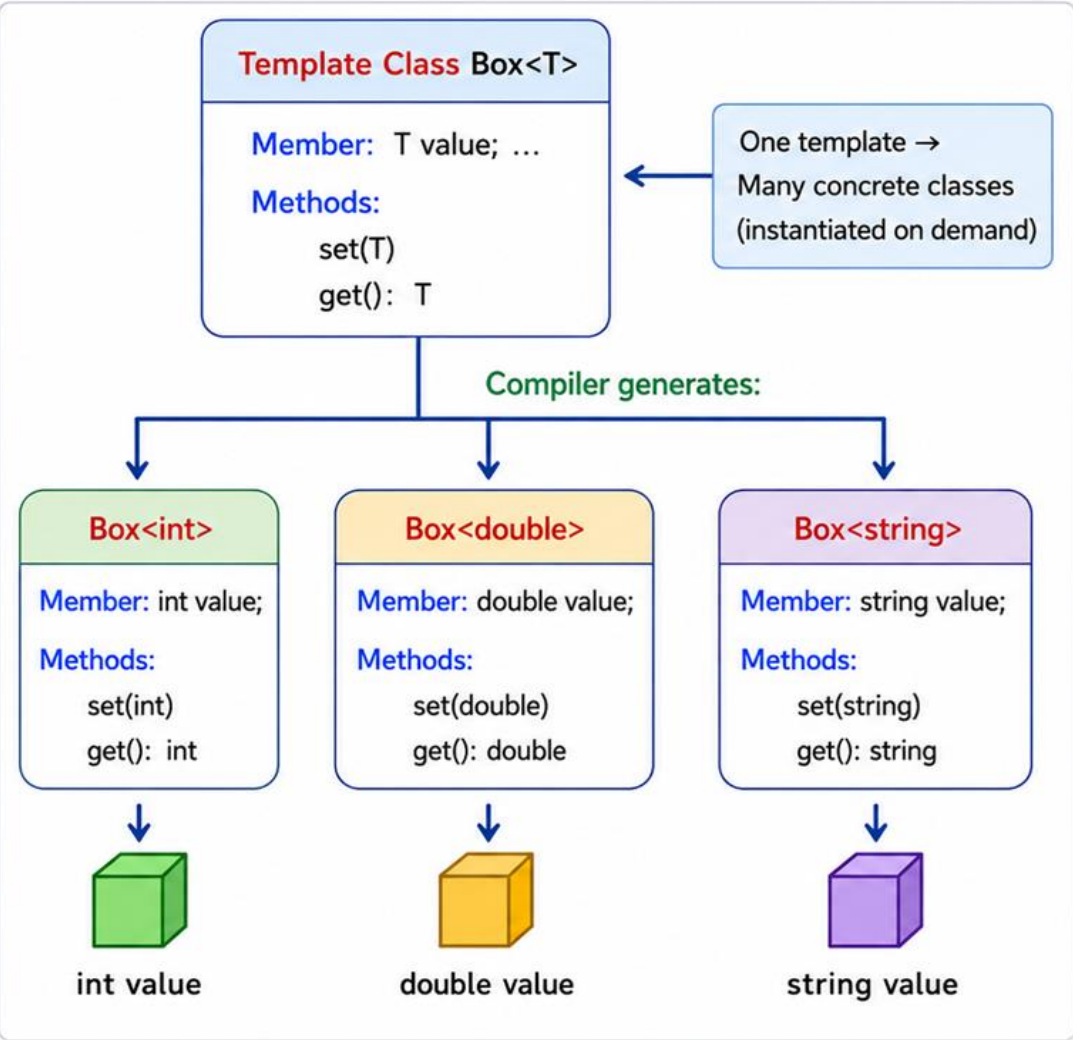
```
ClassName < DataType > objectName;

            or

ClassName < DataType > objectName ( arguments );
```

Example

```
template<class T>
class Adder {

public:
    T num1, num2;

    Adder(T x, T y) {
        num1 = x;
        num2 = y;
    }

    void add() {
        cout << "Sum of the numbers: ";
        cout << num1 + num2 << endl;
    }
};

int main() {

    // Integer Object
    Adder<int> d1(4, 5);
    d1.add();

    // Double Object
    Adder<double> d2(14.5, 65.75);
    d2.add();

    return 0;
}
```

Cont'd

Visual Illustration



Description

- A class template **Box<T>** defines a generic class with a member of type T.
- The compiler generates concrete classes when **Box** is used with a specific type.
- Examples of instantiated classes:

  Box<int> Box<double> Box<string>

- Each instantiated class has its own version of the member and methods based on the type.
- This is how one template → many concrete classes works in C++.

Source:

AI-generated image (ChatGPT 5.3, 2026)

Cont'd

Rules and Features of Class Templates

Complete Definition Must Be Available

- Definition of **class templates** and **their member function** typically placed in the same header file.
- A member function defined outside the class declaration, must also be defined as a template.
- Use **ClassName<T>::** as the scope resolution operator for external member functions.
- **Example**:

```
template <typename T>
void Box<T>::set(T value) {
    data = value;
}
```

Template Arguments Requirement

- Before **C++17**, a class template must be instantiated with explicit template arguments.
- Starting from C++17, **Class Template Argument Deduction** (CTAD) allows the compiler to deduce template arguments automatically.

Before C++17:	Since C++17:
`Box<int> b(10);`	`Box b(10); // Deduces Box<int>`

No Template Parameter Re-declaration

Template parameter names must be unique within the same template parameter list.

```
template <class T, class T>   // Error
```

Cont'd

Complete Code Example

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

// Class Template
template <typename T>
class Student {
   private:
       string name;
       T score;

   public:
      Student(string n, T s) {
          name = n;
          score = s;
      }
      void displayInfo() {
          cout<<"\nName : "<<name;
          cout<<"\nScore: "<<score;
      }

      // Function prototypes
      void setScore(T s);
      T getScore();
};

// Member functions outside the class
template <typename T>
void Student<T>::setScore(T s) {
      score = s;
}
template <typename T>
T Student<T>::getScore() {
      return score;
}
int main() {
      // Explicit template arguments
      Student<int> s1("John", 85);
      Student<double> s2("Sara", 92.5);

      s1.displayInfo();
      s2.displayInfo();
      cout << endl;

      // Update score
      s1.setScore(90);
      cout << "Updated Score: "
           << s1.getScore()
      cout<<"\n\n";
      // C++17 CTAD
      Student s3("David", 78);
      Student s4("Helen", 88.7);

      s3.displayInfo();
      s4.displayInfo();

      return 0;
}
```

2.3 Template Type Deduction

Core Concepts

- It is the **process** by which the C++ compiler automatically determines the **template argument(s)** from the **arguments provided** to a function call or object initialization.
- In other words, it allows the compiler to **determine template types** automatically and you do not always need to explicitly specify the template type.

How it works?

When a **template** is called, the **compiler**:

1. ***Examines*** *the arguments provided.*
2. ***Matches*** *them with the template parameter(s).*
3. ***Deduces*** *the appropriate type(s).*
4. ***Generates/creates*** *the corresponding function or class instance.*

General Form:

```
Template Definition
  template <typename T>
  void func(T value);

Function Call: func ( 100 );

Compiler Deduces: T = int
```

Cont'd . . .

Examples

(i) Function Template:

```
template <typename T>
T maximum(T a, T b){
    return (a > b) ? a : b;
}

int main(){
    cout << maximum(10, 20) << endl;
    cout << maximum(5.5, 8.2) << endl;
    return 0;
}
```

Function Call	Deduced Type
maximum (10, 20)	T = int
maximum (5.5, 8.2)	T = double

(ii) Class Template:

```
template < typename T>
class Box {
  public:
        T value;
        Box(T v) {
              value = v;
        }
};
```

```
int main() {
        Box b(100);
        Box b2(3.14);
        Box b3('A');
}
```

Object Instantiation	Deduced Type
Box b1(100);	Box<int> b1(100); T = int
Box b2(3.14);	Box<double> b2(3.4); T = double
Box b3('A');	Box<char> b3('A'); T = char

Cont'd . . .

Explicit Template Argument

- Sometimes, instead of allowing the **compiler to deduce** the template type automatically, the programmer explicitly specifies the template argument.
- This is often referred to as **Explicit Template Argument Specification**.
- It gives the programmer full control.

When it is needed?

- ✓ When you want to **force a specific type**.
- ✓ When the **compiler cannot deduce** the type because of unambiguity.

Example

```
template < typename T>
void display(T value) {
        cout << value << endl;
}

int main() {
        display< int>( 100);
        display< double>( 3.14);
        display< char>( 'A');
}
```

Note: *Here, the programmer **explicitly tells** the compiler which type to use.*

Quick Check – Function & Class Templates

1. What keyword is used to declare a function template in C++?

A. generic
B. typedef
C. template
D. type

Answer: B (template)

2. What does typename T represent in a function or class template?

A. A variable name
B. A return type specifier
C. A placeholder for any data type
D. A function name alias

Answer: C *(A placeholder for any data type)*

3. How do you create an object from template <typename T> class Box?

A. Box b;
B. Box(int) b;
C. Box<int> b;
D. template Box b;

Answer: B (Box<int> b;)

4. vector<int> and vector<string> are examples of which template feature?

A. Function
B. Object
C. Overloading
D. Class

Answer: D (class template)

Quick Check – Function & Class Templates

3. Given function template definition, what happens when you call add(2, 3.5)?

```
template <typename T>
T add(T a, T b)
```

A. It adds them as integers
B. It adds them as doubles
C. A compile-time error
D. It automatically converts to string

Answer: C
→ T can't be both **int** and **double** at the same time

5. When defining a member function outside a class template, what prefix must you use?

A. class <T>
B. typename T
C. template <typename T>
D. No prefix needed

Answer: C (template <typename T>)

Multiple Template Parameters and Template Specialization

3.1 Multiple Template Parameters

Core concepts

- A **template** can have **more than one type parameter**, allowing it to work with **multiple data types** simultaneously.
- The parameters can be
  - ✓ ***type parameters (typename/class)***
  - ✓ ***non-type parameters (int, char, bool)***

👉 Generic Syntax:

```
template < typename T1, typename T2, ..., typename Tn >
```

👉 Specific to Function & Class

1- Function Template

```
template < typename T1, typename T2, ..., typename Tn >
return_type function_name (parameter_list) {
    // function body
}
```

2- Class Template

```
template < typename T1, typename T2, ..., typename Tn >
class ClassName {
    // class members
};
```

Cont'd . . .

👉 **Example:** function template with multiple parameters

```
template <typename T1, typename T2>
void display(T1 a, T2 b)
{
    cout << "First Value: " << a << endl;
    cout << "Second Value: " << b << endl;
}

int main()
{
    display(100, 45.6);
    display("Age", 25);

    return 0;
}
```

In this example,

the compiler automatically deduces:

- **T1 = int, T2 = double**
- **T1 = int, T2 = double**

Note:

*Multiple template parameters increase the **flexibility** and **reusability** of generic code.*

3.2 Non-type Template Parameters (NTTP)

What is NTTP?

Non-type Template Parameters are **template parameter** that represents a **constant value** rather than a data type.

Syntax:

```
template < typename T, DataType ValueParameter>
return_type function_name (parameters);
// or
template < typename T, DataType ValueParameter>
class ClassName {
    // class members
};
```

👉 **Example:**

T – is s type parameter

```
template <typename T, int SIZE>
void printArray (T arr[SIZE]) {
        for ( int i = 0; i < SIZE; i++)
                cout << arr[i] << " ";
}

int main() {
        int nums[] = { 1, 2, 3, 4, 5};
        printArray< int, 5>(nums);
}
```

SIZE: *Non-type parameter*

Cont'd . . .

Common NTTPs

```
template < int N> // integer

template < char C> // character

template < bool B> // boolean

template < auto Value> // C++17 and later
```

Common uses of NTTPs

1. Specify Array Sizes.
2. Create containers with fixed-size.
3. Represent mathematical constants
4. Define configuration settings / values

Take away:

👉Unlike **type template parameter** that acts as a placeholder for a **data type**, **NTTP** acts as a **placeholder** for a compile-time constant value.

👉It allow templates to be **customized with constant values** known at compile-time, making programs more **efficient**, **flexible**, and **type-safe**.

3.3 Template Specialization

Core Concepts

- A **template specialization** is a C++ programming feature that allows you to **define a customized implementation** of a generic template for a specific data type.
- Sometimes a specific data type requires behavior that differs from the generic template implementation.
- So, template specialization enables the compiler to use a **specialized version** of a template whenever a specific data type is encountered.

Syntax:

```
template <>
class ClassName<SpecificType> {
    // Specialized implementation
};
```

Why Specialization?

- To customize the behavior of a template for specific data types.
- To handle special cases differently from the generic implementation.
  - ✓ *E.g. bool types may need special output formatting.*
  - ✓ *char* needs string comparison, not pointer comparison*
- Performance optimization for a specific type

Cont'd

Example: Template Specialization

```
// Function Template
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

// Function Template Specialization for strings
template <>
string findMax<string> (string a, string b) {
    cout << "String comparison: ";
    return (a.length() > b.length()) ? a : b;
}
```

```
// Class Template
template <typename T>
class Display {
public:
    void show(T value) {
        cout << "Value: " << value << endl;
    }
};

// Class Template Specialization for string
template <>
class Display<string> {
public:
    void show( string value)    {
        cout << "Student Name: " << value << endl;
    }
};
```

👉 Key Principle:

*The **specialization** is only for the specified type; all other types use the **generic template**.*

Cont'd

Generic vs Specialized Templates

Aspect	Generic Template	Specialized Template
Applies to	All types (unless overridden)	One specific type only
Syntax	template <typename T>	template <>
Purpose	Common behavior	**Custom behavior for edge case**
Priority	Lower (fallback)	Higher (preferred for that type)
Use case	add<int>, add<double>	maxVal<const char*>

How the compiler chooses between Generic & Specialized?

```
Call: maxVal("hello", "world")   →  T = const char*
  1. Is there a specialization for const char*?  YES → Use it
  2. Otherwise → Use generic template
```

Quick Check – Template Specialization

1. Which declaration correctly defines a class template with two type parameters?

A. template <class T, class U> class Pair {};
B. template <T, U> class Pair {};
C. class Pair <typename T, typename U> {};
D. template (class T, class U) Pair {};

Answer: A

2. What is the purpose of template specialization?

A. To increase the number of template parameters
B. To provide a custom implementation for types
C. To replace inheritance in C++
D. To eliminate all function overloading

Answer: B

3. What syntax signals a full template specialization?

A. template <class T>
B. specialize <T>
C. template <>
D. Ttemplate <T>

Answer: C

4. Full template specialization provides a completely separate implementation for a specific type. [True / False]

Answer: True

Practice Exercise

Practical Exercise

Exercise — Generic Data Store

Objective

Implement a **Generic Data Store** in C++ that combines function templates, class templates, and template specialization to demonstrate the practical application of generic programming.

This exercise will demonstrate how to:

- *Create **generic classes** using class templates.*
- *Create **generic function** using function templates.*
- *Use **template specialization** to customize behavior for specific data types.*
- *Store and retrieve data of different types*

Requirements

- Create a class template named **DataStore.**
- Implement both **Default** constructor and **Parameterized** constructor.
- Define public method – ***setData ()***, ***getData ()*** *and* ***display ().***
- Implement a function template named ***printValue()*** that works with any data type.
- Provide a **specialized** version of the ***display()*** function for the **string data type**.

Cont'd

Solution: —

Skeleton Code

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

template < typename T>
class DataStore {
        private:
                T data;
        public:
                DataStore();
                DataStore(T value);

                void setData(T value);
                T getData();

                void display();
};
```

```
// Template Specialization for string
template < >
class DataStore < string > {

        //Member definition goes here
};

template < typename T>
void printValue (T value);
```

```
int main() {

        // Create integer store

        // Create float store

        // Call display()

        // Call printValue()

        return 0;
}
```

The complete Solution Found Here

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia