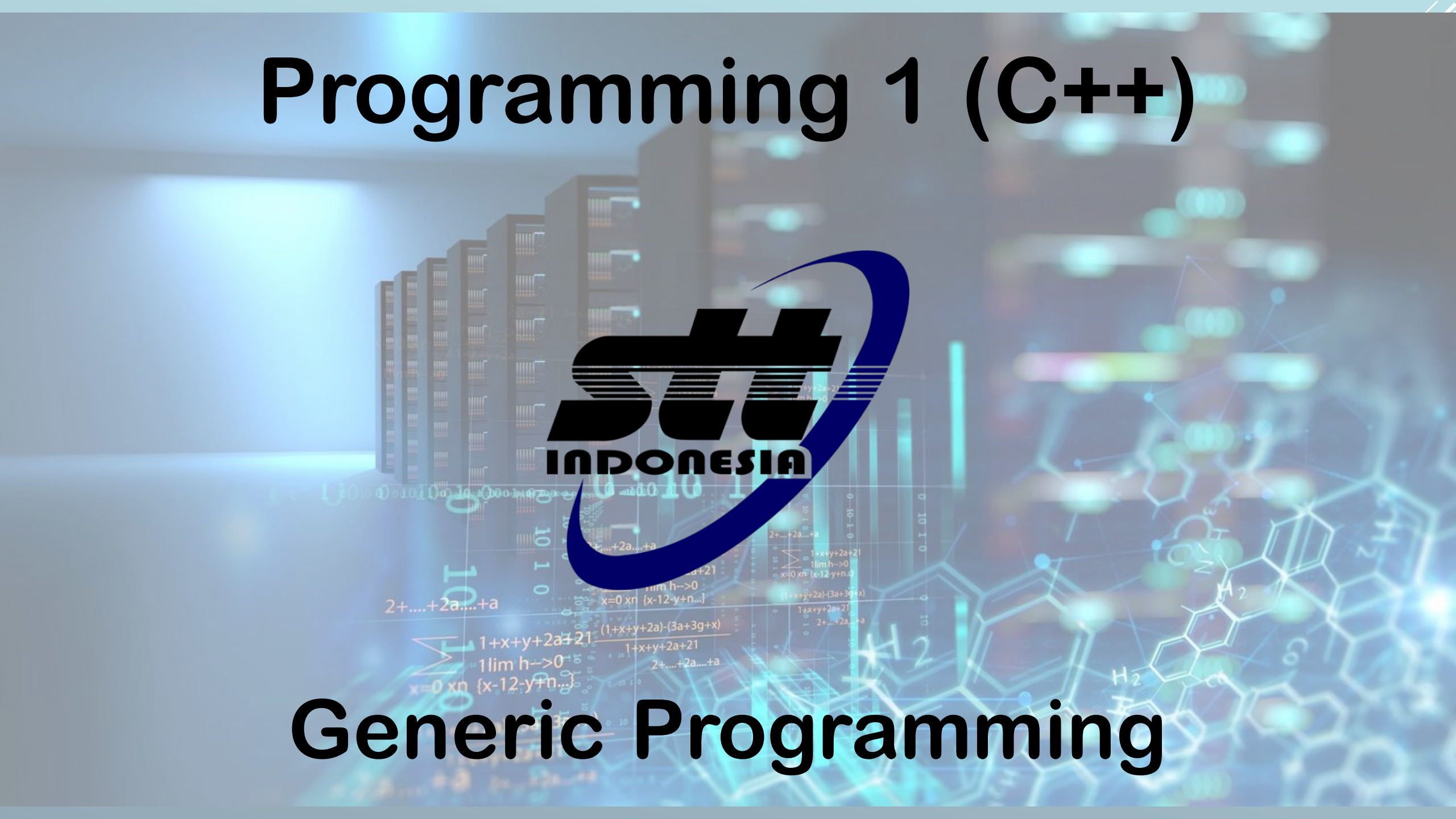
Programming 1 (C++)



Generic Programming

INTRODUCTION

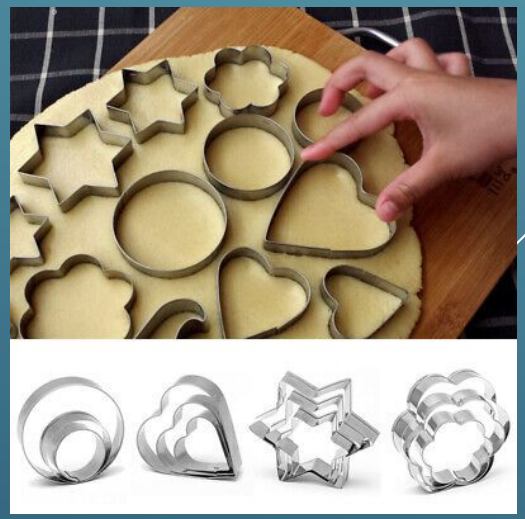
- Generic programming in C++ is a programming paradigm that emphasizes writing code in a way that is independent of data types.
- It allows you to write functions and classes in a way that they can operate with any data type, rather than being tied to a specific type.
- This is achieved through the use of ***templates***, which allow you to define functions and classes with placeholder types that are specified when the code is used.

CHAPTER 1 : FUNCTION TEMPLATES

- A template is an abstract recipe for producing concrete code.
- Templates can be used to produce functions and classes.
- The compiler uses the template to generate the code for various functions or classes, the way you would use a cookie cutter to generate cookies from various types of dough.
- The actual functions or classes generated by the template are called instances of that template.
- = Generic Function.
- is a blueprint to generate a potentially infinite number of FUNCTION OVERLOADS.

```
template<typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
```

Cookie Stamps



FUNCTION TEMPLATES

```
int max (int a, int b)
{
if (a > b)
return a;
else
return b;
}
double max (double a, double b)
{
if (a > b)
return a;
else
return b;
}
```

TEMPLATES MECHANISM

```
template <typename T>
T max (T a, T b)
{
if (a > b)
return a;
else
return b;
}
```

FUNCTION TEMPLATES EXAMPLES

- Sorting Algorithm ⇒ need to interchange a pair of elements, this can be done by function swaps integers or sorting string objects

Integers Sorting

```
void swap(int& m,int& n)
{ int temp = m;
m=n;
n = temp;
}
```

String Object Sorting

```
void swap(string& s1,string& s2)
{ string temp = s1;
s1 = s2;
s2 = temp;
}
```

- We can avoid this redundancy by replacing both functions with a ***function template***

FUNCTION TEMPLATES EXAMPLES

- The Swap Function Templates.

```
template <class T>
void swap(T&x, T&y)
{ T temp = x;
x=y;
y = temp;
}
```

The symbol T is called a *type parameter* = a place holder that is replaced by an actual type or class when the function is invoked.

INSTANTIATION

- For a non-generic function, the compiler reads its definition, checks for errors, and generates executable code.
- When the compiler processes a generic function’s definition, it can only detect errors that are independent of the template parameters like parsing errors.
- Purpose: Instantiation refers to the process of creating concrete functions or classes from templates by replacing template parameters with actual types or values.
- This process transforms a template into a concrete function or class by substituting template parameters with actual types or values. It generates specific code based on the template and its arguments.
- Example: Instantiating a function template max with int would create a function : max(int a, int b).

PARAMETER TYPE DEDUCTION

- It allows the compiler to automatically deduce the types of function template parameters based on the arguments passed during function calls.
- It enables the compiler to infer the types of template parameters based on the arguments provided during function calls. This allows for flexibility in function templates, making them applicable to a wider range of argument types.
- Examples:
  - Value Parameters: **template<typename T> void func(T value)** deduces **T** from the type of the argument passed **(int, double, etc.).**
  - Lvalue-Reference Parameters: **template<typename T> void func(T& value)** deduces **T** as the reference type of the argument passed **(int&, double&, etc.).**

FORWARD REFERENCES

- Purpose: To handle forwarding of arguments, maintaining their value category (lvalue or rvalue) during function calls.
- Forwarding references (also known as universal references) preserve the value category (lvalue or rvalue) of arguments when passed to other functions. They are typically used in templates to maintain the original characteristics of forwarded arguments.
- Example: Using **std::forward** within a function template to preserve the value category of an argument when forwarding it to another function.

PERFECT FORWARDING

- Purpose: To forward both the argument and its value category to another function.
- This technique allows a function template to forward its arguments, along with their value categories, to another function. It's particularly useful for passing arguments through multiple layers of function calls while preserving their original properties.
- Example: Implementing a function template ***wrapper*** that perfectly forwards its arguments to another function ***foo.***

GENERIC Rvalue FUNCTION

- Purpose: To allow function templates to accept rvalue references as parameters.
- By accepting rvalue references as parameters, function templates can efficiently handle temporary objects or movable types. This improves performance and allows for more flexible use of functions in scenarios involving move semantics.
- Example: Defining a function template ***func*** that accepts rvalue references as arguments.

ERRORS IN TEMPLATES

- Purpose: To handle errors that may arise during the use of function or class templates.
- Templates can introduce complexities, including errors that may arise during compilation or instantiation. Techniques like static assertions and SFINAE help manage these errors by controlling template behavior based on specific conditions.
- Example: Utilizing static assertions or SFINAE (Substitution Failure Is Not An Error) techniques to control template instantiation based on certain conditions.

MIXING TYPES

- Purpose: To enable function templates to work with different types of arguments seamlessly.
- Function templates support mixing different types of arguments seamlessly, enabling generic programming. This flexibility allows templates to operate on diverse data types without needing separate implementations for each type.
- Example: Creating a function template ***add*** that can add values of various types (***int, double,*** etc.).

UNIFORM INITIALIZATION

- Purpose: To provide a consistent syntax for initializing objects regardless of their type.
- This feature provides a consistent syntax for initializing objects across various types. It enhances readability and simplifies code maintenance by using curly braces {} for initialization, irrespective of the type being initialized.
- Example: Initializing objects using curly braces ***{ }*** syntax, e.g., ***int x{5};, std::vector<int> vec{1, 2, 3};.***

AUTOMATIC RETURN TYPE

- Purpose: To allow function templates to deduce their return type based on the type of expressions used within the function.
- Function templates can deduce their return types based on the expressions used within them. This eliminates the need to explicitly specify return types, making the code more concise and adaptable to changes in the return type.
- Example: Defining a function template ***sum*** that returns the sum of two values, allowing the return type to be automatically deduced.

TERSE TEMPLATE PARAMETERS

- Purpose: To reduce verbosity when specifying template parameters by using the ***auto*** keyword.
- Terse template parameters reduce verbosity by allowing the compiler to automatically deduce template arguments using the auto keyword. This simplifies template usage, especially in cases where template parameters can be inferred from function arguments.
- Example: Using ***auto*** as a template parameter for function templates to allow the compiler to deduce the type automatically.

THANK YOU