## Templates

**1. Generic Programming with Templates**

Templates in C++ allow writing functions and classes without specifying the exact data type. This enables generic programming.

**2. Create a Function Max() for Integers**

First, let's create a simple function Max() that accepts two integers and returns the larger one.

A computer screen shot of a computer screen

Description automatically generated with medium confidence

**3. Invoke Max() from main() and Print the Result**

In the main() function, we call Max() and print the result.

**4. Overloading Max() for Different Types**

To find the maximum of floats or doubles, we would need to overload Max() for each type.

A white background with blue text

Description automatically generated

**5. Convert Max() into a Function Template**

To avoid overloading for each type, we'll convert Max() into a function template.

**6. Templates with template Keyword**

Templates start with the keyword template, followed by typename (or class) and a placeholder name (e.g., T).

**7. T as a Placeholder**

T serves as a placeholder for the type, substituted by the compiler during compilation.

**8. Update Max() to Use T**

The return type and arguments of Max() are now of type T.

A screenshot of a computer program

Description automatically generated**9. Compiler Automatically Generates Functions**

We don't need the earlier overloaded functions as the compiler generates them for different types when invoked.

**10. Instantiation During Compilation**

The process by which the function is generated is called instantiation, happening during compilation.

**11. Generalized Software Components**

Templates allow creating generalized software components, leading to high-performance algorithms and classes.

**12. Compile-Time Code Generation**

Templates are utilized by the compiler to **generate code at compile time**, eliminating runtime costs.

**13. Libraries Using Templates**

Many libraries like ATL, WTL, Boost, etc., are implemented using templates for high performance.

**14. Summary**

In summary, templates provide flexibility and efficiency in C++ programming by enabling code reuse and compile-time optimization.

By using templates, we can write more flexible and reusable code, which is one of the major strengths of C++ in systems and application programming.

A white background with black text

Description automatically generated

A white paper with black text

Description automatically generated

A white paper with black text

Description automatically generated

**1. Template Argument Deduction**

Template argument deduction is the process by which the compiler determines the types for template type arguments. This is crucial for the instantiation of template functions and classes.

**2. Examining Function Arguments**

The compiler examines each function argument and deduces the corresponding type argument from it.

**3. Example of Type Deduction**

For instance, if the argument type is an integer, the corresponding type argument is deduced as an integer.

A computer screen shot of a computer code

Description automatically generated

**4. Consistency in Deduction**

Subsequent deductions in other function arguments should lead to the same type. No conversions are performed during template argument deduction.

A computer error message

Description automatically generated

**5. Successful Deduction and Instantiation**

Once the type argument is successfully deduced, the template is instantiated.

**6. Overriding Deduction**

In some cases, we may need to override the deduction process by specifying types in the template argument list.

A close-up of a computer code

Description automatically generated

**7. Blueprints for Code Generation**

Template functions or classes serve as blueprints for code generation by the compiler after template argument deduction.

**8. Implicit Instantiation**

Template instantiation occurs implicitly when a function template is invoked, its address is taken, or it is explicitly instantiated.

A close-up of a math problem

Description automatically generated

**9. Explicit Specialization**

Explicit specialization of a function template also triggers instantiation.

A white background with colorful text

Description automatically generated

**10. Visibility of Full Definition**

The full definition of the template must be visible to the compiler for instantiation, so templates are typically defined in header files.

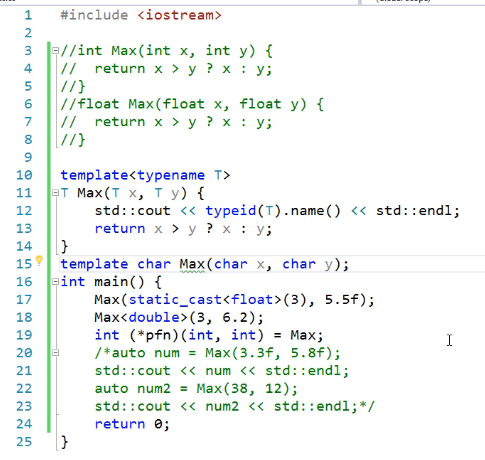
**11. Implementation in Header Files**

Unlike conventional declaration and definition, templates often have their implementation in the header file to ensure visibility for instantiation by the compiler.

A computer screen shot of a person

Description automatically generated

By placing the template definition in the header file, we ensure that any translation unit including this header file has access to the complete definition of the template, allowing the compiler to instantiate it as needed.





**1. Template Argument Deduction**

Template argument deduction is the process of determining the types for template type arguments based on function arguments.

**2. Printing the Type of T Using typeid()**

We can use the typeid() operator to print the type of T and invoke the Max() function with different data types.

A computer code with many text

Description automatically generated with medium confidence

**3. Deduction with int and double**

Invoking Max() with int and double deduces T as int and double respectively.

**4. Consistency in Deductions**

Subsequent function arguments should deduce T as the same type to avoid compilation errors.

**5. Type Mismatch Compilation Error**

If deduction leads to different types, it results in a compilation error as the compiler doesn't perform any type conversions.

A computer error message

Description automatically generated

**6. Solving Type Mismatch**

Two ways to solve this problem: typecasting one argument or explicitly specifying the type of T in the function A close-up of a computer code

Description automatically generated**7. Overriding Deduction**

Explicitly specifying the type of T overrides the compiler's deduction process.

**8. Instantiation by Taking Function Address**

Taking the address of a function template causes the compiler to instantiate it for the specified types.

A close-up of a computer code

Description automatically generated

**9. Explicit Instantiation**

Explicit instantiation can also be done by specifying the types in the template argument list.

A computer code with text

Description automatically generated with medium confidence

**10. Explicit Specialization**

Explicit specialization of a function template triggers instantiation for the specialized type.

A computer code with text

Description automatically generated with medium confidence

**11. Compile-Time Instantiation**

**Template instantiation occurs at compile time**, and the full template definition must be visible to the compiler.

**12. Templates in Header Files**

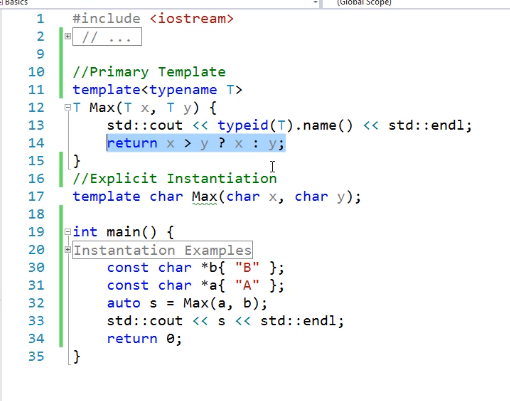
Templates are typically defined in header files, and instantiation happens implicitly in various scenarios like function invocation, address-taking, and explicit instantiation.

A computer code with text

Description automatically generated with medium confidence

By defining templates in header files, we ensure that the full definition is visible to the compiler, enabling it to instantiate the template for any types used in different translation units.

# Explicit Specialization



* This template will not good solution to deal with strings because it will compare the address of arrays, not the actual strings

A white background with black text

Description automatically generated

The provided text elaborates further on the concept of explicit specialization in C++. Here's a breakdown of the content:

1. **Definition:**

* Explicit specialization involves specializing a template for a particular type.
* This specialization provides correct semantics for a specific data type or optimally implements an algorithm for that type.

1. **Implementation:**

* When a template's default implementation may not be suitable for a specific type, explicit specialization allows writing a specialized definition for that type.
* These specialized definitions are explicitly written for specific types to ensure proper behavior and optimization.

1. **Location of Definitions:**

* **Unlike** general template definitions, explicitly specialized functions **must be defined in a `.cpp`** file **rather than a header file.**
* It's emphasized that the primary template definition should occur before any explicit specializations to adhere to the One Definition Rule (ODR).

.



1. **Definition of Explicit Specialization:**

* An explicit specialization is defined using the `template` **keyword followed by an empty template argument list.**
* The return type and name of the function are specified.
* Arguments for the function can be specified as needed.

1. **Comparison Algorithm:**

* In this example, the function compares strings rather than addresses using the `strcmp` function.
* If one string is greater than the other, the function returns the greater string.

1. **Printing a Message:**

* The code includes a message to indicate that the explicit specialized function is being invoked.

1. **Instantiation and Definition:**

* It's mentioned that explicit specialization is already instantiated, so it should not be defined and declared again.
* It's recommended to specify the type for which the specialization is done, though it's optional.
* Differentiating between explicit specialization and explicit instantiation is important due to their similar syntax.

# Nontype Template Arguments

A white paper with black text

Description automatically generated

Non-type templates in C++ add an additional layer of flexibility to template programming by allowing constant values, rather than just types, to be specified as template parameters. Let's break down each point with examples to better understand non-type templates.

**1. Definition**

* Non-type template parameters are values (rather than types) that can be integral, pointer, reference, or enumeration types. These parameters are specified within angle brackets < > when declaring a template.
* Non-type template arguments appear in the template argument list and **must be constant expressions.**
* These arguments are commonly used in functions and classes to provide constant values that are **known at compile time.**

A computer code with text

Description automatically generated with medium confidence

**2. Usage**

Non-type template parameters are used to provide constant **expressions known at compile time**. This allows defining functions or classes that operate on these values.

A close-up of a text

Description automatically generated

**3. Example: Fixed-Size Array**

A function template can be created to compute the sum of elements in a fixed-size array. The size of the array can be specified as a non-type template parameter.

A computer code with text

Description automatically generated with medium confidence

**4. Constant Expressions**

Non-type template parameters must be constant expressions, meaning their values must be computable at compile time. This allows the compiler to generate specialized code based on these parameters.

A computer screen shot of a code

Description automatically generated

**5. Flexibility**

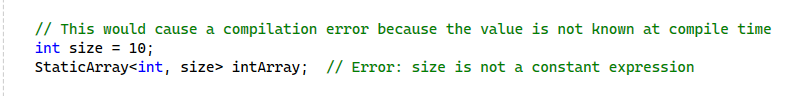
Non-type templates enhance the flexibility of template-based programming by allowing the creation of generic algorithms or data structures that operate on values determined at compile time.

A screenshot of a computer code

Description automatically generated

**6. Limitations**

Non-type templates have limitations. They cannot be used with types determined at runtime or with expressions that cannot be evaluated at compile time.



**Summary**

Non-type template parameters extend the versatility of templates by allowing values to be part of the template's definition. They must be constant expressions known at compile time, which makes them highly efficient for certain types of compile-time computations and optimizations. However, they are limited to values that are known during compilation and cannot be dynamically determined at runtime. This capability allows for more specialized and optimized template instantiation, further enhancing the power and flexibility of C++ template programming.

**Example 1: Print Function:**

A screenshot of a computer program

Description automatically generated

**Example 2: Sum Function with Arrays:**

* Create a function `Sum()` that computes the sum of elements in an array.
* Use a non-type template argument for the size of the array.
* Pass the array as a reference to the function.
* The size should be a constant expression or computed at compile time.

A screenshot of a computer program

Description automatically generated

# Perfect Forwarding

A computer screen shot of a code

Description automatically generated

A computer code with text

Description automatically generated

**Implementing Move Semantics in the Employee Class**

To efficiently handle temporary values for the name and id attributes of the Employee class, we must leverage move semantics. Move constructors enable the transfer of resources from temporary objects without unnecessary copying, thus optimizing performance.

**Constructor Types**

1. **Constructor with const References:**

* **Description**: Takes const references for name and id.
* **Behavior**: When passed temporary objects, they are bound to const references, enforcing copy semantics.
* **Drawback**: Even if temporary objects are used, the constructor will perform copying instead of moving, which can be inefficient.

1. **Constructor with Rvalue References:**

* **Description**: Takes rvalue references (denoted by &&) for name and id.
* **Behavior**: Rvalue references allow the constructor to bind to temporary objects, facilitating move operations.
* **Advantage**: By using std::move, this constructor can transfer the resources of temporary objects efficiently into member variables, avoiding unnecessary copying and enhancing performance.

A computer code with text

Description automatically generated

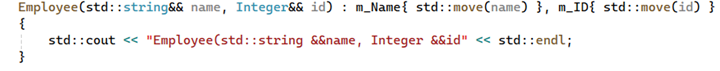
**Practical Usage Scenarios**

1. **Using const Reference Constructor**:

* **Example**: Employee emp1(name, id);
* **Outcome**: This invokes the copy constructor, even if name and id are temporary objects, leading to potential inefficiencies.

1. **Using Rvalue Reference Constructor**:

* **Example**: Employee emp2(std::move(name), std::move(id));
* **Outcome**: This uses move semantics, transferring ownership of name and id resources efficiently, resulting in better performance with temporary objects.



**Challenges in Constructor Design**

1. **Variety of Initialization Scenarios**:

* **Issue**: Users may initialize Employee objects with different sets of arguments. Some may provide pre-initialized values, while others might pass temporary values directly.
* **Implication**: A single constructor type may not suffice to handle all these cases optimally.

A computer code with text

Description automatically generated

1. **Efficiency Concerns**:

* **Issue**: Even with temporary arguments, using the const reference constructor results in copying, which is inefficient.
* **Implication**: This necessitates careful consideration of when to invoke move semantics to enhance performance.

1. **Complexity of Multiple Constructors**:

* **Issue**: Implementing separate constructors for every possible combination of argument types can be tedious and error-prone.
* **Implication**: This complexity can make the class less maintainable and harder to extend.

**Solution: Function Templates for Constructors**

To address these challenges, function templates can be employed. These allow for greater flexibility in handling different types and combinations of arguments without the need for multiple explicit constructor definitions.

* **Template Constructor**:
* **Description**: A template constructor can accept arguments of any type, forwarding them appropriately.
* **Behavior**: By using std::forward, the constructor can preserve the value category (lvalue or rvalue) of each argument, ensuring that move semantics are used when applicable.
* **Advantage**: This approach reduces the need for numerous overloaded constructors, simplifying the codebase and making it more flexible and maintainable.

**Example**

A close-up of a computer code

Description automatically generated

# Std::forward

A computer code with text

Description automatically generated

**Using Function Templates for Constructors in C++**

To handle various argument types efficiently, including both lvalues and rvalues, we can use function templates with perfect forwarding. Here's a detailed explanation:

**Constructor with Function Templates**

* **Template Parameters T1 and T2**:
* The constructor accepts arguments of any type using T1&& name and T2&& id.
* These template parameters allow the constructor to handle both lvalues and rvalues.
* **Perfect Forwarding with std::forward**:
* std::forward is used to preserve the type information of the arguments.
* This ensures that when the arguments are forwarded to initialize the attributes m\_Name and m\_ID, their type information (lvalue or rvalue) is maintained.

**Behavior of std::forward**

* **Lvalues and Rvalues**:
* When lvalues are passed to the function template, the types T1 and T2 become lvalue references (T1& and T2&).
* When rvalues are passed, T1 and T2 become rvalue references (T1&&).
* This distinction ensures that the arguments are handled correctly based on their value category.
* **Type Preservation**:
* By using std::forward<T1>(name) and std::forward<T2>(id), we ensure that the arguments are forwarded with their correct type information.
* This is crucial for maintaining the efficiency of move semantics when rvalues are used.

**Advantages**

* **Flexibility**:
* The template constructor allows for more flexible instantiation of the Employee class.
* It can accommodate various types of arguments for name and id, whether they are lvalues or rvalues.
* **Efficiency**:
* Perfect forwarding ensures that move semantics are used whenever possible, optimizing resource management.
* This leads to better performance by avoiding unnecessary copying.

**Summary**

1. **Function Template Constructor**:

* The constructor accepts any type of arguments using template parameters.
* Perfect forwarding with std::forward preserves the type information of the arguments, ensuring efficient resource management.

1. **Handling Lvalues and Rvalues**:

* Lvalues passed to the constructor become lvalue references.
* Rvalues passed to the constructor become rvalue references.
* std::forward maintains the correct value category during initialization.

1. **Efficiency and Flexibility**:

* The constructor can handle various argument types flexibly.
* Perfect forwarding ensures that move semantics are utilized, leading to efficient resource management.

By employing function templates and perfect forwarding, we ensure that the Employee class can be instantiated flexibly and efficiently, handling both lvalues and rvalues appropriately. This approach simplifies the code and enhances performance, making the class more robust and versatile.

A close-up of a computer code

Description automatically generated

In C++, there are several ways to initialize member variables in a class constructor, including using initializer lists. The two lines you provided demonstrate two different initialization methods:

1. **m\_First\_Name{ std::forward<T>(FirstName) }**:

Here, **m\_First\_Name** is initialized using uniform initialization syntax **{}**, and **std::forward** is used to forward the **FirstName** argument as either an lvalue or an rvalue, depending on its original value category. This is known as perfect forwarding and is typically used when the type of **FirstName** is a template parameter (**T**) and you want to preserve its value category (lvalue or rvalue) when initializing **m\_First\_Name**.

1. **m\_Last\_Name(FirstName)**:

In this case, **m\_Last\_Name** is initialized using the traditional constructor syntax **()**. This will directly copy or move the value of **FirstName** into **m\_Last\_Name**, depending on whether **FirstName** is an lvalue or an rvalue. If **FirstName** is an lvalue, it will be copied; if it's an rvalue, it will be moved.

Now, let's delve deeper into the differences between these two initialization methods:

* **Handling of rvalue references**:
  + **m\_First\_Name{ std::forward<T>(FirstName) }**: This uses perfect forwarding to preserve the original value category of **FirstName**, ensuring that if **FirstName** is an rvalue, it will be moved into **m\_First\_Name**.
  + **m\_Last\_Name(FirstName)**: This will always copy or move **FirstName** into **m\_Last\_Name**, depending on its original value category. If **FirstName** is an rvalue, it will be moved; if it's an lvalue, it will be copied.
* **Consistency**:
  + Using perfect forwarding with **std::forward** (**m\_First\_Name{ std::forward<T>(FirstName) }**) ensures that the initialization of **m\_First\_Name** follows the same rules as the forwarding of the constructor argument **FirstName**, maintaining consistency in value category handling.
  + Directly initializing with **m\_Last\_Name(FirstName)** may lead to inconsistency if **FirstName** is an rvalue, as it might be copied instead of moved, depending on the implementation of the class's copy and move constructors.

In summary, **m\_First\_Name{ std::forward<T>(FirstName) }** is a more advanced technique that ensures perfect forwarding of the constructor argument, preserving its original value category and enabling optimal handling of rvalue references. On the other hand, **m\_Last\_Name(FirstName)** provides a simpler and more straightforward initialization, but it may not always result in the most efficient handling of rvalue references.

In C++, both **m\_Last\_Name(FirstName)** and **m\_Last\_Name{FirstName}** are examples of member variable initialization in the constructor's initializer list, but they use different syntax for initialization:

1. **m\_Last\_Name(FirstName)**:
   * This syntax uses parentheses **()** for initialization, indicating direct initialization.
   * It will invoke the appropriate constructor of the member variable **m\_Last\_Name** to initialize it with the value of **FirstName**.
   * If **FirstName** is an rvalue, it will be moved into **m\_Last\_Name**; if it's an lvalue, it will be copied.
2. **m\_Last\_Name{FirstName}**:
   * This syntax uses curly braces **{}** for initialization, indicating uniform initialization.
   * It performs list initialization, which can be more restrictive than direct initialization.
   * It will attempt to initialize **m\_Last\_Name** by selecting the appropriate constructor that matches the provided initializer list **{FirstName}**.
   * If the constructor is marked as **explicit**, this syntax will fail to compile unless there's a constructor that accepts **FirstName** as its single argument.
   * If **FirstName** is an rvalue, it will preferentially call move constructors; if it's an lvalue, it will call copy constructors.

In summary, the main difference between these two syntaxes lies in their initialization semantics:

* **Direct initialization (m\_Last\_Name(FirstName))**: Uses parentheses for initialization and directly invokes the appropriate constructor.
* **Uniform initialization (m\_Last\_Name{FirstName})**: Uses curly braces for initialization and performs list initialization, potentially invoking constructors that match the provided initializer list.

Both syntaxes are valid and commonly used in C++ codebases. The choice between them often depends on personal preference, coding style guidelines, or specific requirements of the project.

**std::forward** is primarily used in the context of perfect forwarding, which is a technique used to forward arguments exactly as they were passed, preserving their value category (lvalue or rvalue) and cv-qualifiers. It's particularly useful when you're writing functions or class templates that need to pass arguments through to other functions, maintaining the same reference type and qualifiers.

Here's a simple example to illustrate its usage:

A screenshot of a computer code

Description automatically generated

The benefits of using **std::forward** and perfect forwarding include:

1. **Avoiding unnecessary copies/moves**: By forwarding arguments as they were passed, unnecessary copies or moves can be avoided, leading to potentially better performance.
2. **Preserving reference type and cv-qualifiers**: It allows you to preserve the exact reference type and const/volatile qualifiers of the original argument, enabling more flexible and precise function forwarding.
3. **Enabling universal interfaces**: Perfect forwarding enables the creation of more generic and flexible interfaces, allowing functions and class templates to work with a wider range of argument types without sacrificing efficiency or safety.

Overall, **std::forward** is an essential tool for writing efficient, flexible, and generic code in C++.

# Variadic Templates

A screen shot of a computer code

Description automatically generated

* Variadic templates in C++ allow functions and classes to accept an arbitrary number of arguments of any type. This feature provides flexibility and versatility in programming, similar to how the `printf` function in C can handle various argument types and counts.
* To implement variadic behavior without using macros, we can utilize C++11's initializer list feature in a function template. By iterating over the initializer list, we can access and print each individual element, allowing for generic argument handling.
* However, a **limitation** of initializer lists is that all **arguments must be of the same type.** This restricts the flexibility of the function. Variadic templates address this limitation by enabling functions to accept arguments of different types and counts.
* **With variadic templates**, we can define functions that **accept any number of arguments of any type**. This powerful feature enhances type safety and allows for more flexible and generic programming practices.

# Here's the implementation of the `Print()` function using variadic templates and recursion:

A computer code with text

Description automatically generated with medium confidence

**Explanation:**

* We define a base case function `Print()` with no arguments to end the recursion.
* The variadic template function `Print()` takes the first argument (`First`) and a parameter pack (`Params...`) representing the remaining arguments.
* Inside the function, we print the first argument and then recursively call `Print()` with the remaining arguments using the parameter pack expansion `Print(params...)`.
* In the `main()` function, we invoke `Print()` with various types of arguments to demonstrate its usage.

**This implementation allows `Print()` to accept any number and any type of arguments, utilizing the power of variadic templates and recursion:**

* To implement a `Print()` function using variadic templates, we start by defining a function template. Variadic templates enable functions to accept a variable number of arguments of any type. The syntax for a variadic template involves using ellipses (`...`) to represent an arbitrary number of type names. These type names are then packed into a parameter pack, which acts as an alias to the list of types.
* Once we've defined the function template with the parameter pack, we can use recursion to access and process each individual argument. Recursion allows us to handle an arbitrary number of arguments by reducing the number of arguments in each recursive call until reaching a base case that terminates the recursion.
* The base case function serves as the stopping condition for the recursion. It's a function without any arguments, effectively ending the recursion when reached.
* During the recursive calls, we utilize parameter pack expansion to pass the arguments to subsequent function calls. This expansion automatically unpacks the parameter pack, allowing us to pass each argument individually.
* However, it's essential to handle the case where no arguments are provided to prevent infinite recursion. This requires defining an overload of the `Print()` function without using variadic templates. This overload acts as the base case and serves to terminate the recursion when no arguments are left to process.
* **By combining variadic templates with recursion and base case handling,** we create a flexible and powerful `Print()` function capable of **accepting and processing any number and any type of arguments.**

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **Initial Invocation of `Print()`:**

* `Print()` is invoked with multiple arguments, including integers, floats, and strings.
* This initiates the recursive process where the call proceeds into the `Print()` function.

1. **Recursive Calls with Reduced Arguments:**

* Initially, `Print()` receives all arguments forming a function parameter pack.
* Each recursive call captures the first argument and passes the remaining arguments to the next invocation.
* Arguments in subsequent calls reduce by one until the base case is reached.

1. **Reduction of Arguments:**

* As recursion progresses, the number of arguments passed to `Print()` decreases by one in each call.
* For instance, subsequent calls receive fewer arguments, such as 2.5, 3, and "4", then 3 and "4", and finally "4".

1. **Base Case: Empty Arguments:**

* Eventually, `Print()` is invoked with no arguments, forming the base case.
* The compiler searches for a function `Print()` without any arguments, indicating the termination of recursion.

1. **Base Case Function:**

* To handle the base case, a separate overload of `Print()` is defined that accepts no arguments.
* This function serves as the stopping point for the recursive process, preventing infinite recursion.

1. **Template Instantiation:**

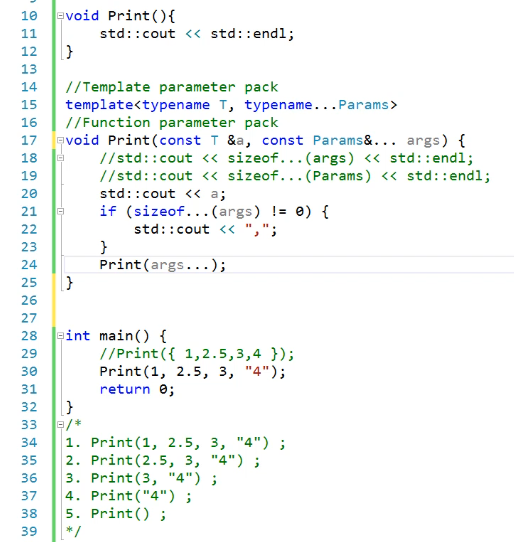
* During template instantiation, the compiler identifies the base case function `Print()` that does not accept any arguments.
* Although no such function exists initially, the base case function is implemented to satisfy template requirements.

1. **Debugging and Call Stack:**

* Debugging reveals the call stack, illustrating how arguments are processed during recursion.
* Each recursive call consumes one argument until reaching the base case with zero arguments.
* The call stack demonstrates the gradual reduction of arguments until the recursion stops.

1. **Execution and Outcome:**

* After the base case is reached, the recursion stops, and the program completes its execution.
* Individual arguments can be accessed and manipulated within the recursive calls, allowing for various operations.



1. **Recursion with Variadic Templates:**

* Recursion is employed with variadic templates to process an arbitrary number of arguments passed to a function.
* Each recursive call captures a subset of arguments until reaching a base case that stops the recursion.

1. **Base Case Function:**

* A base case function is implemented to terminate the recursion.
* It serves as the stopping point when no more arguments remain to be processed.

1. **Reducing Arguments:**

* During recursion, the number of arguments passed to the function is reduced by one in each recursive call.
* This reduction continues until the base case is reached.

1. **Printing Arguments:**

* All arguments passed to the `Print()` function are printed.
* Initially, they are printed together without any separator.

1. **Adding Separator:**

* A comma (`,`) is introduced as a separator between printed arguments.
* However, an extra comma appears after printing the last argument.

1. **Handling Extra Comma:**

* The `sizeof...()` operator is used to determine the number of arguments in the function parameter pack.
* If the size is not zero, indicating that there are more arguments, a comma is printed.

1. **Ensuring Efficient Copying:**

* To avoid unnecessary copying of user-defined types, arguments are passed by constant reference.
* This prevents multiple copies of the arguments from being created during recursion.

A screenshot of a computer code

Description automatically generated

1. **Introduction of Integer Class:**

* The Integer class is added to the project, and its header file is included in the code.

1. **Initial Invocation of Print():**

* The Print() function is invoked with integer types, and the first call is commented out to ensure the correctness of subsequent calls.

1. **Issue with Passing R-Value Arguments:**

* Despite passing an R-value as an argument to the Print() function, it's observed that during recursive calls, the argument is treated as an L-value, leading to a loss of efficiency.

1. **Utilizing Perfect Forwarding:**

* To address the issue of losing R-value references during recursion, perfect forwarding is employed.
* The function parameters are specified as R-value references, ensuring that arguments retain their original value categories (L-value or R-value).
* Std::forward is used within the function to preserve the value category of the arguments during forwarding.
* The sizeof...() operator is utilized to determine the number of arguments in the function parameter pack, enabling proper handling of the comma separator.

1. **Implementation of Print() Function:**

* The Print() function is defined with a base case and a template parameter pack to handle an arbitrary number of arguments.
* Within the function, each argument is printed, and if there are more arguments to follow, a comma separator is added.
* Recursion is used to process each argument in the parameter pack until the base case function is reached.

# Class Templates

A screenshot of a computer program

Description automatically generated

This code snippet defines a templated Stack class that allows users to create a stack with a specified data type and size:

1. **Stack Class Template:**

* The Stack class template is defined with two template parameters: `T` for the data type and `size` for the size of the stack.
* It contains a dynamically allocated array `m\_Stack` of type `T` to hold the stack elements and an integer `m\_stackpointer` to track the top of the stack.
* The default constructor initializes the stack.
* The copy constructor `Stack(Stack &Stackobj)` copies the contents of one stack object to another.
* The `Push()` function adds an element to the stack.
* The `Pop()` function removes an element from the stack.
* The `isempty()` function checks whether the stack is empty.
* The `Top()` function returns a reference to the top element of the stack.
* The `Create()` function is a static member function that creates a new stack object.

1. **Function Definitions Outside Class Body:**

* The `Pop()` and `Create()` member functions are defined outside the class body.
* The definitions are prefixed with the template parameters `<typename T, int size>` to indicate that they belong to the Stack class template.
* The syntax `Stack<T, size>::` specifies that these functions are members of the Stack class template.

1. **Additional Includes:**

* The code includes `<iostream>` for standard input-output operations.
* The `Stack.h` header file is included to ensure that the Stack class template is available.

Overall, this code provides a basic implementation of a templated stack data structure with functions for push, pop, top, and checking whether the stack is empty. It demonstrates how to define member functions of a class template outside the class body and how to create and manipulate objects of the templated class.

A screenshot of a computer program

Description automatically generated

In **the `Push()`** function of the Stack class template, **`const T &Data`** is passed as a parameter to indicate that the function accepts a constant reference to the data that needs to be pushed onto the stack. Here's why this approach is commonly used:

1. **Avoiding Unnecessary Copies:**

* Passing by reference (`const T &Data`) avoids making unnecessary copies of the data being pushed onto the stack.
* If `Data` were passed by value (`T Data`), a copy of the data would be made when the function is called, which can be inefficient for large or complex data types.

1. **Maintaining Data Integrity:**

* Using `const T &Data` ensures that the original data passed to the function remains unchanged.
* It provides a guarantee that the function will not modify the original data unintentionally.

1. **Support for Different Data Types:**

* By using a templated approach, the `Push()` function can accept data of any type (`T`), making the stack class more versatile.
* The `const T &Data` parameter allows the function to handle both built-in and user-defined data types without requiring separate implementations.

1. **Ensuring Read-Only Access:**

* The `const` qualifier indicates that the function does not intend to modify the data referenced by `Data`.
* It serves as a reminder to programmers that the function should not attempt to modify the data, enhancing code clarity and maintainability.

1. **Polymorphic Behavior:**

* User-defined types can have custom constructors, destructors, and overloaded assignment operators.
* Despite this, the Push() function treats all types uniformly, pushing them onto the stack without modification due to the const reference parameter.
* Whether the type T is a built-in type or a user-defined type, the behavior of the function remains consistent.

1. **Compile-Time Polymorphism:**

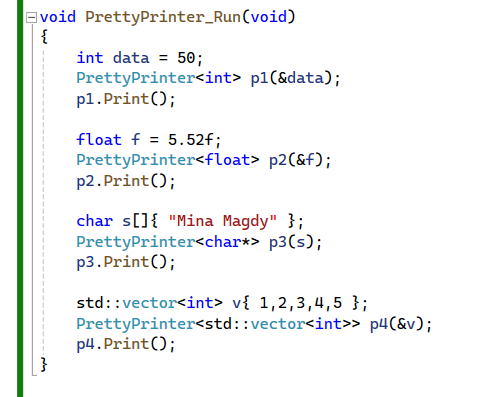
* Templated functions like Push() exhibit compile-time polymorphism, meaning that the correct version of the function is determined by the data type (T) specified during instantiation.
* This enables seamless handling of user-defined types without the need for runtime type checks or special handling.

Overall, passing `const T &Data` in the `Push()` function strikes a balance between efficiency, data integrity, flexibility, and readability, making it a common practice in stack implementations and other data structures.

# Class Template Explicit Specialization

A screenshot of a computer program

Description automatically generated



1. **Implementation of prettyprinter Class Template:**

* The `prettyprinter` class template is defined with a template parameter `T`.
* It has a constructor to initialize data and a `Print()` function to print the data, adding braces for decoration.
* A `getdata()` function is implemented to return a pointer to the stored data.

1. **Usage of prettyprinter with Different Types:**

* Instances of `prettyprinter` are created for `int` and `float` types, and their respective data is printed using the `Print()` function.
* When trying to use `prettyprinter` with a `char\*` (string), an error occurs because the template parameter `T` becomes `char\*`, resulting in a pointer-to-pointer (`char`).
* Passing a `char\*` directly as an argument leads to a compilation error since it doesn't match the expected pointer-to-pointer type.

1. **Explicit Specialization for `char\*` Type:**

* To address the issue with `char\*` type, an explicit specialization for `char\*` is created.
* In the specialization, the template parameter list is left empty, and `T` is explicitly defined as `char\*`.
* The `getdata()` function is modified to return `char\*` instead of `char`.
* Now, when using `prettyprinter` with a `char\*`, it correctly prints the entire string without dereferencing issues.

A screenshot of a computer program

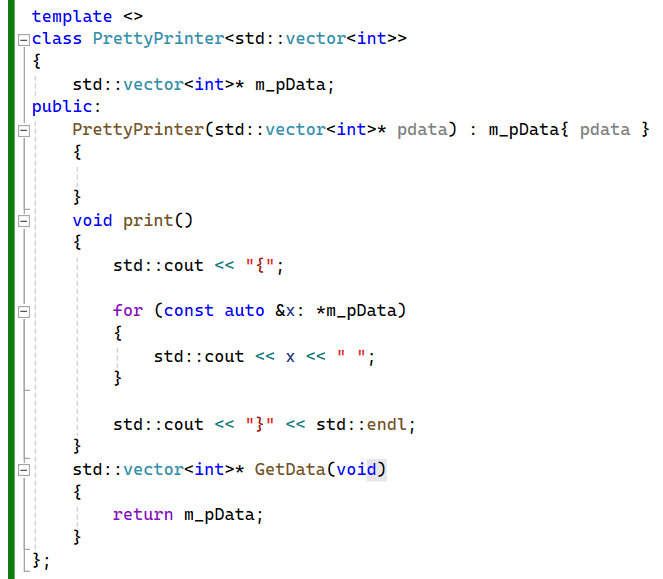
Description automatically generated

1. **Functionality Verification:**

* After applying the explicit specialization, the code is tested again with `char\*` type, and it successfully prints the string data.

In summary, by explicitly specializing the `prettyprinter` class template for the `char\*` type, the issues related to pointer-to-pointer and dereferencing are resolved, allowing correct printing of string data. This approach ensures the proper functionality of `prettyprinter` for various data types, including strings.

# Prettyprinter with vector



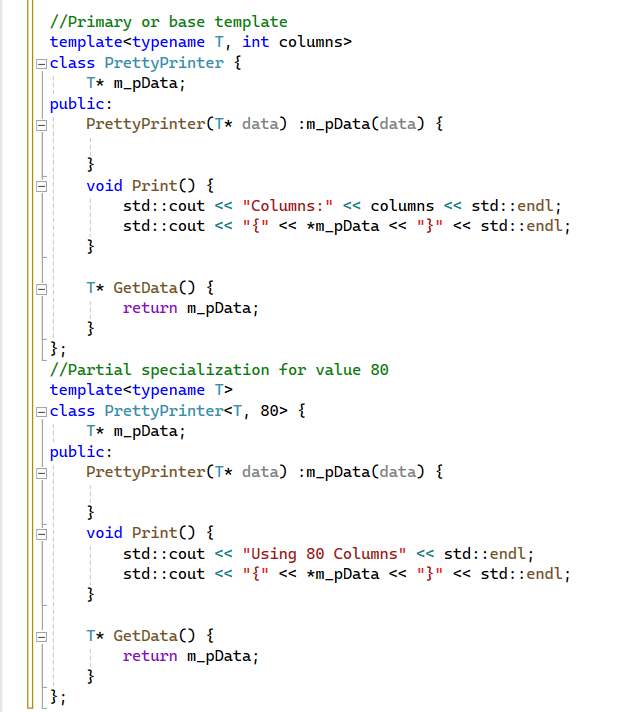
We did not need to **Explicit Specialization** the whole class again like with **(char \*)** only need to **Explicit specialization** the **“Print()”** Function.

A computer code with text

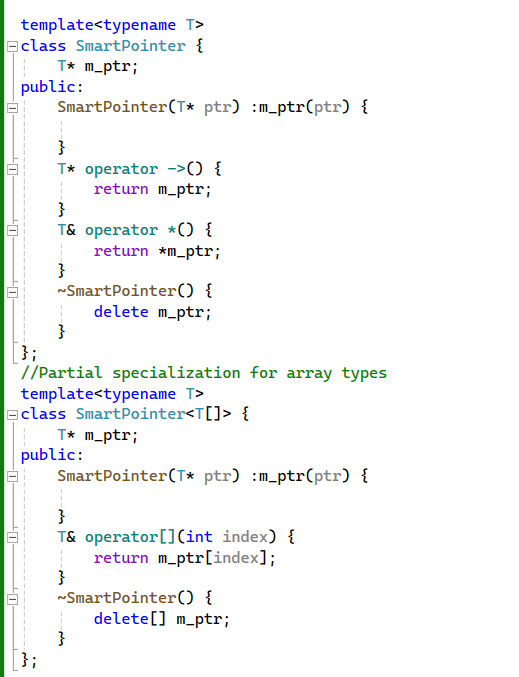
Description automatically generated with medium confidence

* To handle printing for a vector of integers using the `prettyprinter` class, we create a specialized version of **the `Print`** function. Initially, when attempting to print a vector, we encounter issues because the pointer to the vector **causes the asterisk operator (\*)** to dereference it, resulting in printing the entire vector object itself. To address this, we explicitly specialize **the `prettyprinter`** class for the **`std::vector<int>` type**.
* With this specialization, we utilize a range-based for loop inside **the `Print`** function to iterate over the vector elements and print them individually. By doing so, we ensure that the contents of the vector are printed correctly.
* Comparing this specialization to the one for **`char\*`**, we note that for **`char\*`,** we had to specialize the entire class due to issues with other member functions like **`getdata()`.** However, for **`std::vector<int>`,** the problem **is solely related to printing**, so we only specialize the **`Print`** function. This approach allows us to maintain the functionality of other member functions without modification.
* It's important to mention that when explicitly specializing a member function template, its **definition must be outside the class**, within the namespace scope. This ensures proper resolution of the specialization. With this specialized **`Print`** function, we successfully achieve correct printing for vectors of integers using **the `prettyprinter`** class.

# ‎Partial specialization‎



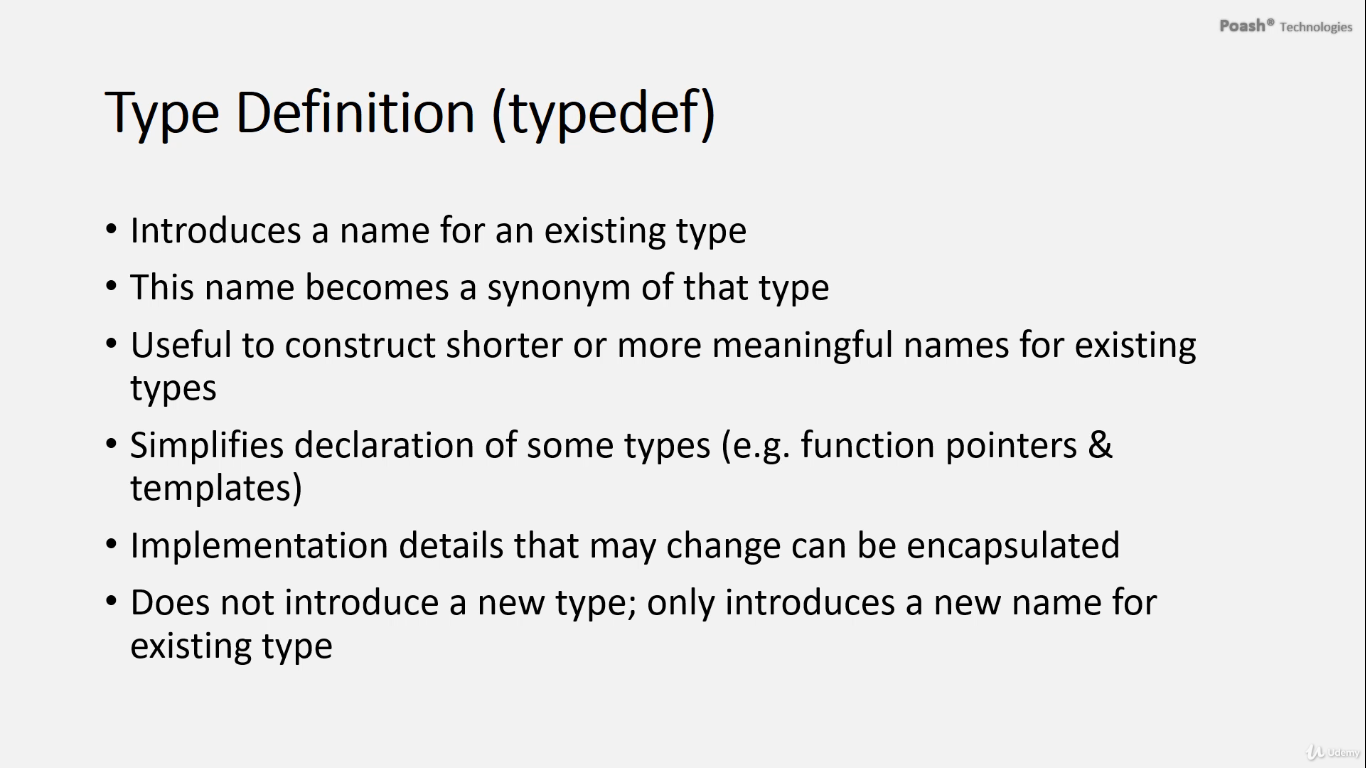
* In partial specialization of class templates, we specialize only some of the template parameters while leaving others unspecified. This differs from explicit specialization, where all template parameters are specialized.
* To illustrate this concept, let's revisit our **`prettyprinter`** class example. We'll introduce a new non-type template parameter called **`columns`,** representing the available width for printing data. This parameter will be constant and immutable.
* After defining the **`prettyprinter`** class with the **`columns`** parameter, we create an instance of **`prettyprinter`** for any data type and invoke **the `Print()`** function. Running the program, we observe that it shows the available columns as 40.
* Now, let's assume that the class will be used on devices where the common number of columns is 80. We want to take advantage of this and format the printing accordingly. While we could achieve this with conditional statements in the **`Print()`** function, there's a more efficient approach**: partial specialization.**
* By partially specializing **the `prettyprinter`** class for the **`columns`** parameter with a value of 80, we can tailor the implementation of the class functions accordingly. In this specialized version, we can adjust the printing format or behavior based on the specific value of **`columns`.**
* Upon running the program again, we observe that the compiler has chosen the partial specialization for **the `columns`** parameter value of 80, demonstrating the effectiveness of partial specialization in customizing class behavior based on specific template parameter values.



# Templates & Partial specialization‎

**With Smart pointers**

1. We discussed the concept of RAII (Resource Acquisition Is Initialization) for creating smart pointers, and we implemented a basic smart pointer for integer pointers. Now, leveraging templates, we can create a smart pointer that can work with any type.
2. Let's call this new class `smartpointer`. It contains a constructor for initializing its internal pointer, overloads the arrow (`->`) and asterisk (`\*`) operators to access the pointed-to value, and has a destructor to release the memory.
3. Using this `smartpointer`, memory is automatically freed at the end of the scope, and we can access the value using the asterisk operator. Running the code, we observe the value `3` printed.
4. However, what if we want to use `smartpointer` with a dynamic array? In such cases, the existing class behavior doesn't provide correct semantics. We can't access individual elements of the array, and the destructor isn't correctly releasing memory using `delete[]`.
5. To address this, we can partially specialize the `smartpointer` class for array types. This means that whenever `smartpointer` is instantiated with an array type, this specialized version will be used.
6. In the array specialization, we replace the arrow and asterisk operators with the subscript (`[]`) operator for array access. Additionally, we ensure that the destructor uses `delete[]` for array memory deallocation.
7. When creating an instance of `smartpointer` for an array, we specify the array type as the argument. This tells the compiler to choose the partial specialization.
8. Using the subscript operator, we can read and write array elements. Running the code, we see the correct value printed.
9. It's worth noting that smart pointers in the standard library also provide partial specialization for array types, allowing for correct behavior when dealing with dynamic arrays.



A screen shot of a computer code

Description automatically generated

A white background with black text

Description automatically generated

A screen shot of a computer code

Description automatically generated

Type aliases, introduced in C++11, provide a more intuitive and flexible way to define alternative names for existing types. Let's break down the key points from the provided text:

1. **Type Definitions (typedef):** Before C++11, type definitions (typedef) were commonly used to create new names for existing types. They allowed developers to introduce shorter or more meaningful names for types, reducing complexity and enhancing readability. Type definitions are created using the `typedef` keyword.
2. **Benefits of Type Definitions:**

* Simplify declaration of types, especially with function pointers and templates.
* Hide implementation details of types, allowing for changes to the underlying type without affecting client code.
* Reduce complexity and enhance readability of code.

1. **Examples of Type Definitions:**

* Creating aliases for unsigned integers (`UINT` for `unsigned int`).
* Creating aliases for long long integers (`LLONG` for `long long`).
* Simplifying complex type declarations, such as for vectors of lists of employees (`Teams` for `std::vector<std::list<Employee>>`).
* Creating aliases for function pointer types, making function pointer usage more intuitive and readable.

1. **Type Alias (using):** C++11 introduced type aliases as an alternative to `typedef`. Type aliases, declared using the `using` keyword, provide a more natural syntax for defining alternative names for types. They behave similarly to `typedef` but offer some advantages, such as improved readability and consistency with variable initialization syntax.
2. **Syntax of Type Alias:**

* Use the `using` keyword followed by an identifier (the alias name).
* Specify the type on the right-hand side of the assignment.
* The identifier then acts as a synonym for the specified type.

1. **Examples of Type Aliases:**

* Creating aliases for unsigned integers (`UINT`).
* Creating aliases for long long integers (`LLONG`).
* Simplifying complex type declarations, such as for vectors of lists of employees (`Teams`).
* Creating aliases for function pointer types (`errorfn`).

Overall, type aliases provide a more modern and expressive way to define alternative names for existing types, offering improved readability and consistency in code. They are especially useful when dealing with complex type declarations and function pointer types.

# Type Definitions vs. Type Aliases in Templates

* When dealing with lengthy template declarations, like in the case of lists of names, using type definitions (typedefs) can simplify the syntax by assigning a shorter name, such as `Names`, to the complex type. However, typedefs cannot be templatized themselves, limiting their flexibility for reuse in different contexts.
* To overcome this limitation, we can use type aliases to create alias templates. Alias templates allow us to define a template alias that can be used with different types and templatized with other templates. This provides more flexibility and ease of reuse across various scenarios.
* For instance, by defining an alias template `Names`, like `using Names = std::vector<std::list<T>>`, we can represent a list of names stored as strings with `Names<std::string>`. Additionally, alias templates enable us to permanently bind the alias to a specific type, if needed.
* In summary, while type definitions using typedefs can simplify code readability, type aliases provide greater flexibility and versatility, especially in template-heavy scenarios.

A screen shot of a computer program

Description automatically generated



A screenshot of a computer code

Description automatically generated

## Type Traits

A white text on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

* Type traits in C++ offer a powerful mechanism for compile-time introspection of types, enabling developers to examine and manipulate type properties during compilation. They are implemented as class templates and are extensively used in template metaprogramming to generate optimized code based on type characteristics.
* Most type traits return either a boolean value indicating a type property or a type itself. They are typically accessed through a template-based interface and often feature a `value` member that holds either `true` or `false`. While many type traits are implemented as class templates, some may rely on compiler intrinsics for efficient implementation.
* C++11 introduced a set of fundamental type traits, with additional traits added in C++14. These traits are widely utilized with templates but can also be applied in non-template scenarios.
* For example, `std::is\_integral` is a commonly used type trait that determines whether a type is integral. To employ type traits, developers include the `<type\_traits>` header file, specify the type as a template argument, and access the `value` member to ascertain the trait's boolean value.
* Type traits are invaluable when designing templates to tailor operations based on type properties. They ensure that functions operate safely and efficiently on specific types, such as restricting a division function to floating-point types only.
* In summary, type traits provide a robust mechanism for compile-time type analysis, facilitating the creation of more flexible and optimized C++ code through template metaprogramming techniques.

Type traits are a powerful feature in C++ that allow compile-time introspection of types. Here's a breakdown of the discussion:

**1. `is\_reference` Type Trait:** This trait is used to determine whether a given type is a reference type or not. It returns a boolean value indicating whether the type is a reference. For example:

* Std::is\_reference<int>::value; // false
* Std::is\_reference<int&>::value; // true

**2. `remove\_reference` Type Trait:** This trait is used to remove any reference qualifiers from a given type. It transforms a reference type into the underlying non-reference type. It's commonly used in generic programming to ensure consistency in type handling. For example:

* Std::remove\_reference<int>::type; // int
* Std::remove\_reference<int&>::type; // int

A screen shot of a computer

Description automatically generated

* There are additional type traits available for transforming types. One such trait is `is\_reference`. By accepting a forwarding reference, a function can handle both L-values and R-values. We can then demonstrate this by invoking `Check()` with both an R-value and an L-value.
* Upon running the code, we observe that when we pass `5` as an argument, it's not considered a reference type. However, when we pass `'value'`, it is indeed a reference type.
* In certain scenarios, we might need to remove a reference from a type, as required by certain library functions. For this purpose, we utilize the `remove\_reference` type trait. Unlike type traits such as `is\_reference`, those that perform transformations don't have a `value` member. Instead, they provide a `type` member, which represents the resulting type after the transformation.
* To demonstrate, we pass the result of `remove\_reference` as an argument inside `is\_reference`. In some compilers, it may be necessary to use the `typename` keyword to inform the compiler that the type is dependent on the `remove\_reference` class.
* Upon running the code again, we observe that for `5` as an argument, `is\_reference` returns false for both statements. However, when we pass `'value'`, `is\_reference` initially returns true, but after applying the `remove\_reference` type trait, the reference is removed, and `type T` is no longer considered a reference.
* These type traits that perform transformations on types are widely used internally by many library functions, including `std::forward` and `std::move`.

# Static Assertion

A close-up of a message

Description automatically generated

* In C, the `assert` macro is commonly used to test expressions. If the expression evaluates to false, the program is terminated by calling `abort()`. However, C++11 introduced a more powerful assertion mechanism called `static\_assert`.
* Unlike the runtime `assert` macro, `static\_assert` operates at compile time. It evaluates expressions and, if the expression evaluates to false, it triggers a compilation error with a user-defined message.
* `static\_assert` is frequently utilized alongside type traits in C++ programming. It enables compile-time checks based on type properties, providing a robust mechanism for ensuring type safety and correctness in template-based code.

Here's a brief comparison:

1. **`assert` macro in C:**

* Used for runtime assertion testing.
* Terminates the program if the expression evaluates to false.
* Dynamic runtime check.

1. **`static\_assert` declaration in C++11:**

* Used for compile-time assertion testing.
* Triggers a compilation error with a user-defined message if the expression evaluates to false.
* Static compile-time check.

Using `static\_assert`, we can enforce compilation in a specific mode, such as 32-bit mode, at compile time. This ensures that the code will not compile if it is not being compiled in the desired mode. Here's how we can achieve this:

A computer screen shot of a program code

Description automatically generated```

* In this example, `static\_assert` checks if the size of a pointer **(`sizeof(void\*)`)** is equal to 4 bytes, which indicates 32-bit mode. If the size is not 4 bytes, the compilation will fail with the specified message **"Compile in 32-bit mode only".**
* This `static\_assert` statement ensures that the code is compiled only in 32-bit mode, preventing compilation in other modes. **By using `static\_assert**`, we perform this check at compile time rather than runtime, ensuring that the code meets the required conditions before compilation proceeds.
* **Using `static\_assert`** is indeed a powerful tool for performing compile-time checks and ensuring certain conditions are met during compilation. It allows you to **halt the compilation process with an error message if a specified condition evaluates to false**. This can be particularly useful for enforcing constraints on the compilation environment, such as ensuring that the code is being compiled in a specific mode or with certain features enabled.

1. **Runtime Check:** Initially, you attempted to perform a runtime check by examining the size of a pointer to determine if the code is being compiled in 32-bit mode. However, this approach only provides information during runtime and does not prevent compilation in other modes.
2. **`static\_assert`:** To enforce the constraint at compile time, you utilized **`static\_assert`.** This allows you to evaluate an expression at compile time and halt compilation if the expression evaluates to false. You provided a meaningful error message to indicate the reason for compilation failure.
3. **Compile-Time Checking with Type Traits:** Additionally, you mentioned the possibility of using type traits, which evaluate to boolean values at compile time. Type traits provide a convenient way to perform compile-time checks based on type properties.

By leveraging `static\_assert` and possibly type traits, you can enforce compile-time constraints effectively, ensuring that your code meets certain requirements before it is compiled. This can help catch potential issues early in the development process and improve code reliability and maintainability.

A screenshot of a computer code

Description automatically generated

* The condition **`if (std::is\_floating<T>::value == false)`** in our code currently checks whether the **type `T`** is not a floating-point type. However, **this check occurs at runtime**, despite the fact that the type trait **`std::is\_floating<T>`** can deduce the **type `T`** at compile time. This means that the entire condition is evaluated **dynamically**, which may not be the most desirable approach.
* To enhance the code and ensure that the check is performed at compile time, preventing compilation if a non-floating-point type is passed to **the “Divide**” function, we can employ static assertions. By utilizing **`static\_assert`,** we can **trigger a compilation error if the condition is not met at compile time.** This approach provides a **compile-time check**, which is preferred over **a runtime check**.

**Here's how we can implement this using `static\_assert`:**

A close-up of a white background

Description automatically generated

In this updated version, if **`T`** is not a floating-point type, the **`static\_assert`** statement **triggers a compilation error with the specified error message.** This ensures that any attempt to call **the `Divide`** function **with a non-floating-point type will result in a compile-time failure**, providing better safety and catching errors earlier in the development process.

A computer code with colorful text

Description automatically generated

* Use static\_assert to check if a given type is a floating-point type. This check is performed at compile time, allowing the compiler to verify the type's properties before generating the executable code. If the condition specified in the static\_assert expression evaluates to false, compilation will halt, and the provided diagnostic message will be displayed, indicating the reason for the failure.
* By combining static\_assert with type traits, you can add compile-time validation to your code, enhancing its robustness and preventing potential runtime errors. This approach is particularly useful for enforcing type-related constraints and ensuring that your code operates correctly with specific types.

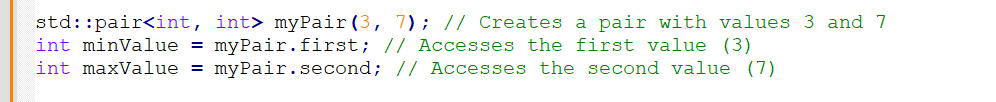
# Std::pair<T, T>

* `std::pair<T, T>` is a type provided by the C++ Standard Library that represents a pair of values of type `T`. It's a templated class where `T` can be any type. This class is commonly used to store two related values together.
* For example, if you want to store the minimum and maximum values of a certain type `T`, you can use `std::pair<T, T>`. In the context of the `minmax` function provided earlier, `std::pair<T, T>` is used to return both the minimum and maximum values found in an array of type `T`.

**Here's a brief explanation of how it works:**

* **“std::pair”** has two template parameters: the first one represents the type of the first value, and the second one represents the type of the second value.
* You can create a **“std::pair”** object by **specifying the types of the values it will hold** and providing those values as arguments to its constructor or by using **“std::make\_pair”.**
* You can access the first and second values of the pair using the **“first” and “second”** member variables, respectively.

**For example:**



* In the `minmax` function, `std::pair<T, T>` is used to return both the minimum and maximum values found in an array of type `T`.

To receive a `std::pair<T, T>` as a function return value, you can declare your function with the return type `std::pair<T, T>`. Here's how you can define a function that returns a `std::pair<int, int>`:

A screenshot of a computer code

Description automatically generated

In this example, the `minmax` function returns a `std::pair<int, int>`, where the first value of the pair represents the minimum value found in the array and the second value represents the maximum value found in the array.

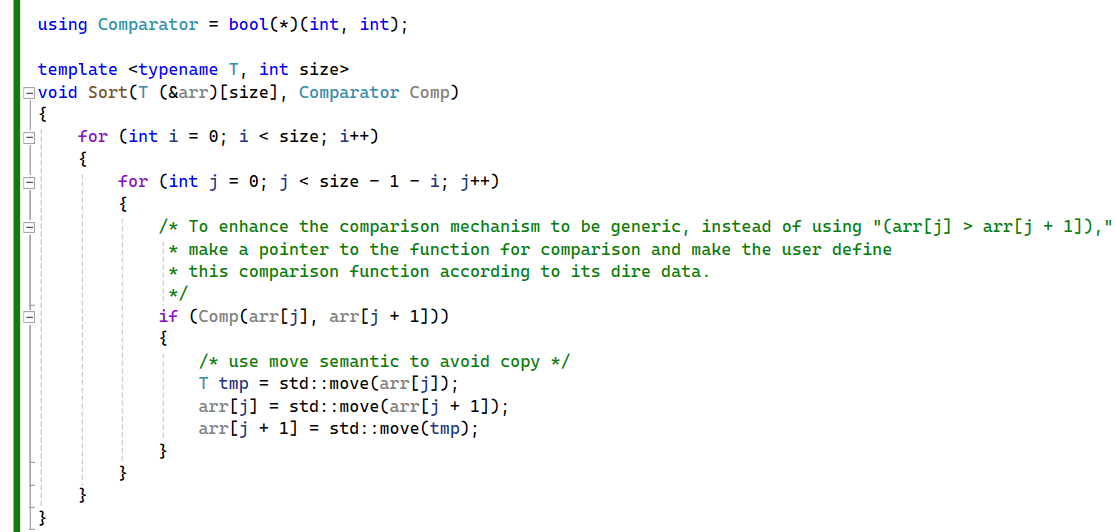
You can call this function like this:

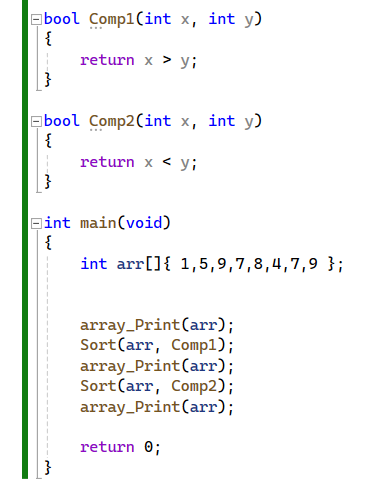
A close-up of text

Description automatically generated

This way, you can receive the minimum and maximum values from the `minmax` function as a `std::pair<int, int>`.

# Callbacks Revisited



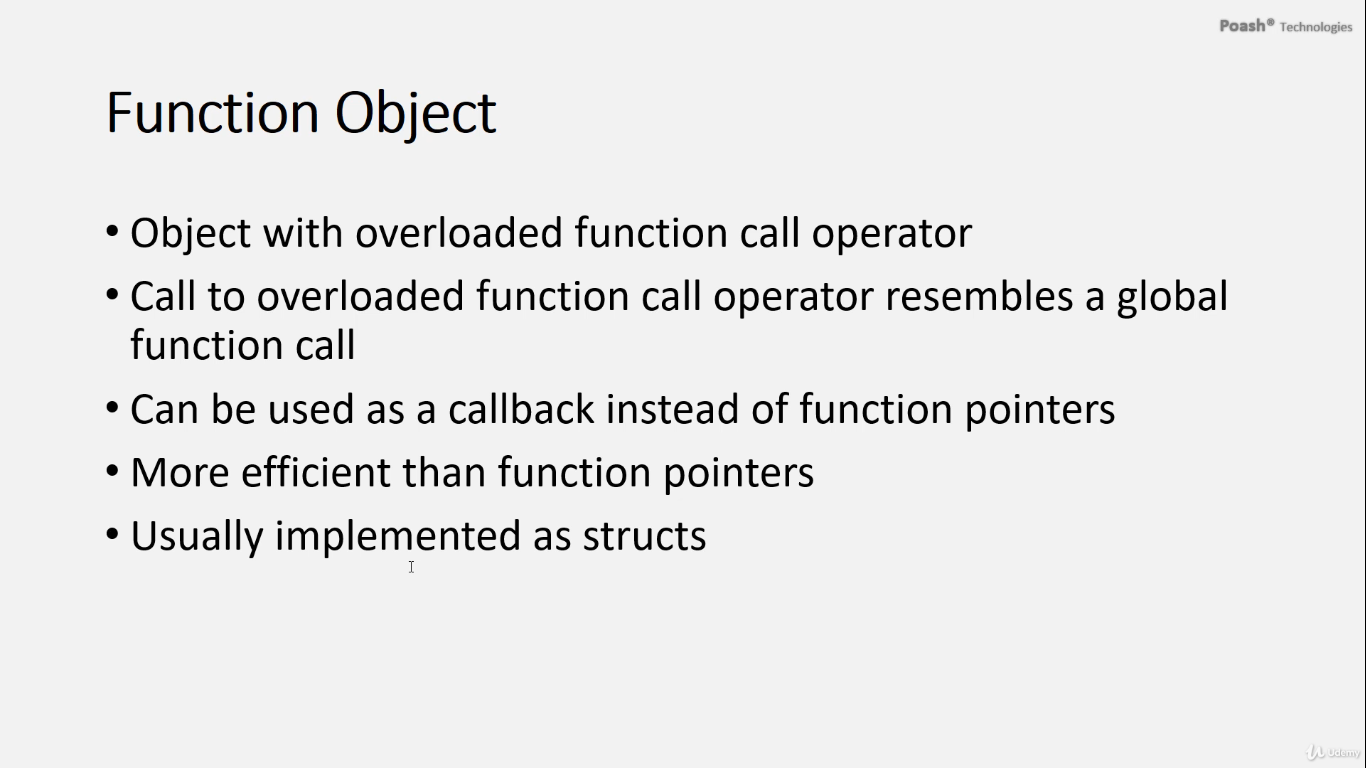


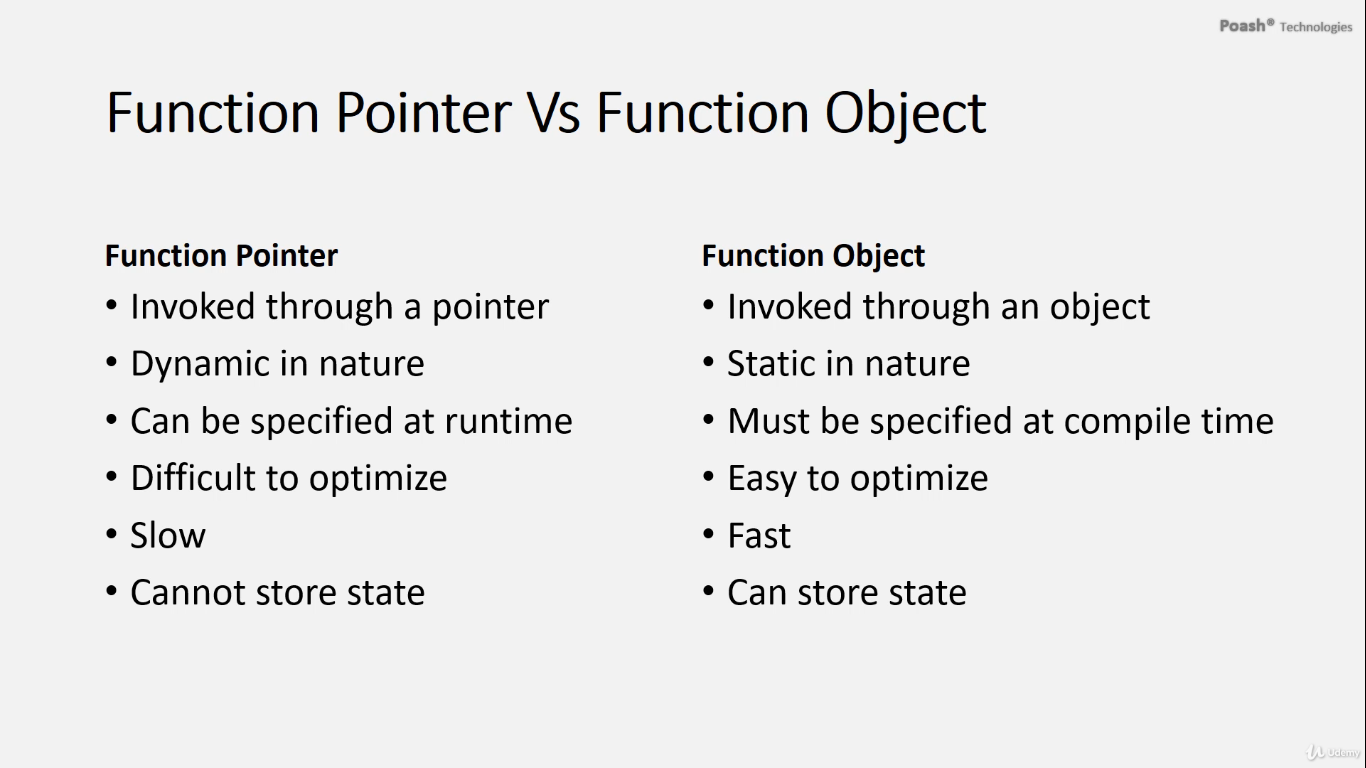
* In this example, we're using function pointers to implement a sorting algorithm with flexible comparison capabilities. By passing a pointer to a comparison function, users can define custom sorting criteria, overcoming limitations such as always sorting in ascending order or inability to compare C-style strings.
* Function pointers allow for dynamic control over sorting behavior, enabling sorting in descending order or other custom arrangements by implementing different comparison functions.
* However, using function pointers comes with drawbacks. Compiler optimization is hindered as function calls via pointers cannot be inlined. Additionally, function pointers point to global functions, prohibiting stateful operations within the comparison function, limiting their use in scenarios requiring state management.

# Callbacks - Function Objects



* In this discussion, we compare the usage of a normal function with that of a function object, which is a class or structure with the function call operator overloaded. While invoking a normal function is straightforward, invoking the function call operator requires creating an instance of the structure and then invoking the operator.
* By converting **the “Comparator**” type into a template argument in the **“Sort()”** function, we can **generalize its usage**. When using a function pointer as the argument, **“Comparator”** becomes a function pointer type, and the code works as before. However, by passing a function object as the argument, “Comparator” becomes the type of the object, **allowing for potential performance optimizations like inlining by the compiler.**
* Function objects also offer the advantage of **storing state between invocations**, unlike global functions. Consequently, **STL algorithms** can accept either function pointers or function objects as callbacks, but **function objects** are generally **preferred** for their **efficiency and state-handling capabilities.**





A function object, also known as a functor, is an object that has the function call operator overloaded. When you invoke the function call operator through the object, the call resembles a global function call. That is why a function object can be used as a callback instead of function pointers with templates, and using a function object as a callback is more efficient than function pointers. Function objects are usually implemented as structures rather than classes. Here, i'll explain the differences between a function pointer and a function object:

1. **Invocation:**

* A function pointer is invoked through a pointer, whereas a function object is invoked through an object of a class or a struct.

1. **Dynamic vs. Static Nature:**

* A function pointer is dynamic in nature and can be specified at runtime, but the function object is static in nature and must be specified at compile time. It is not possible to specify a function object at runtime.

1. **Optimization:**

* A function pointer is difficult to optimize because the call to the function is made through an address, and the compiler cannot know at compile time which function is invoked through that address. In contrast, the compiler precisely knows which function has to be invoked with a function object, allowing it to easily inline the function calls.

1. **Performance:**

* A function pointer is slower due to its dynamic nature, but the function object is faster.

1. **State:**

* A function pointer is stateless, meaning it cannot store state. However, multiple instances of function objects can have unique states, which is not possible with function pointers.

Function objects are a powerful mechanism for creating callbacks in C++. In C++11, there is a better way of implementing callbacks, and that is through lambda expressions.