

3D Game Engine Programming

Helping you build your dream game engine

C++ Template Programming

Posted on [June 1, 2021](#) by [Jeremiah](#)



In this post, I will introduce you to C++ template programming. Some of the topics I will cover are function templates, class templates, variable templates, variadic templates, type traits, SFINAE, and C++ 20 concepts.

— C++ Templates

Contents [\[hide\]](#)

- [1 Introduction](#)
- [2 Terminology](#)
 - [2.1 Expressions and Value Categories](#)
 - [2.1.1 lvalue](#)

2.1.2 prvalue

2.1.3 xvalue

2.1.4 glvalue

2.1.5 rvalue

2.2 Template Arguments versus Template Parameters

2.2.1 Type Template Parameter

2.2.2 Non-type Template Parameter

2.2.3 Template Template Parameter

2.2.4 Template Parameter Pack

3 Basics

3.1 Function Templates

3.2 Class Templates

3.3 Variable Templates

3.4 Alias Template

3.5 typename Keyword

3.6 Template Specialization

3.7 Partial Specialization

3.8 Variadic Templates

3.9 Recursive Variadic Templates

4 Template Type Deduction

4.1 Case 1: ParamType is a Reference or Pointer

4.2 Case 2: ParamType is a Forwarding Reference

4.3 Case 3: ParamType is Neither a Pointer nor a Reference

5 decltype Specifier

6 std::declval()

7 Type Traits

7.1 integral_constant

7.2 bool_constant

7.3 true_type

7.4 false_type

7.5 type_identity

7.6 void_t

7.7 enable_if

- 7.8 remove_const
- 7.9 remove_volatile
- 7.10 remove_cv
- 7.11 remove_reference
- 7.12 remove_cvref
- 7.13 remove_extent
- 7.14 remove_all_extents
- 7.15 remove_pointer
- 7.16 add_const
- 7.17 add_volatile
- 7.18 add_cv
- 7.19 add_lvalue_reference
- 7.20 add_rvalue_reference
- 7.21 add_pointer
- 7.22 conditional
- 7.23 conjunction
- 7.24 disjunction
- 7.25 negation
- 7.26 is_same
- 7.27 is_void
- 7.28 is_null_pointer
- 7.29 is_any_of
- 7.30 is_integral
- 7.31 is_floating_point
- 7.32 is_arithmetic
- 7.33 is_const
- 7.34 is_reference
- 7.35 is_bounded_array
- 7.36 is_unbounded_array
- 7.37 is_array
- 7.38 is_function
- 7.39 decay

8 SFINAE

8.1	SFINAE-Out Function Overloads
8.2	SFINAE-Out Partial Specializations
8.3	SFINAE with enable_if
8.3.1	As a Template Parameter
8.3.2	As a Function Argument
8.3.3	As a Return Type
8.3.4	Summary
9	Concepts
9.1	Concept Definition
9.2	Concept Expression
9.3	Requires Expression
9.3.1	Simple Requirement
9.3.2	Type Requirement
9.3.3	Compound Requirement
9.3.4	Nested Requirement
9.4	Requires Clause
9.4.1	Shorthand Notation
9.5	Constrained Class Templates
9.5.1	Constrained Class Members
9.6	Concept-Based SFINAE
9.7	Constraining Auto & Abbreviated Function Templates
10	Conclusion
11	Bibliography

Introduction

C++ Template Programming has been around for a long time and there are plenty of books and articles on the internet that describe C++ template programming. Although the information is in abundance, there doesn't seem to be a comprehensive source for learning how to apply template programming and demystify some of the more complex features of C++ template programming such as using and writing type traits, template meta programming, and *Substitution Failure Is Not An Error* (SFINAE) which are a few of the topics that I would like to cover in this article.

Terminology

Before delving into the details of C++ template programming, it is important to establish some common terminology when describing template programming.

Value categories are a way to categorize how a value can be used. There are five value categories (*lvalue*, *prvalue*, *xvalue*, *glvalue*, and *rvalue*). You may already be familiar with *lvalue*, and *rvalue*, but might not have encountered *prvalue*, *xvalue*, or *glvalue* yet.

Understanding the difference between *template parameters* and *template arguments* is also important when talking about template programming. In short, template parameters are used to declare the template and template arguments are used to define the template. Template parameters can be either typed, or non typed.

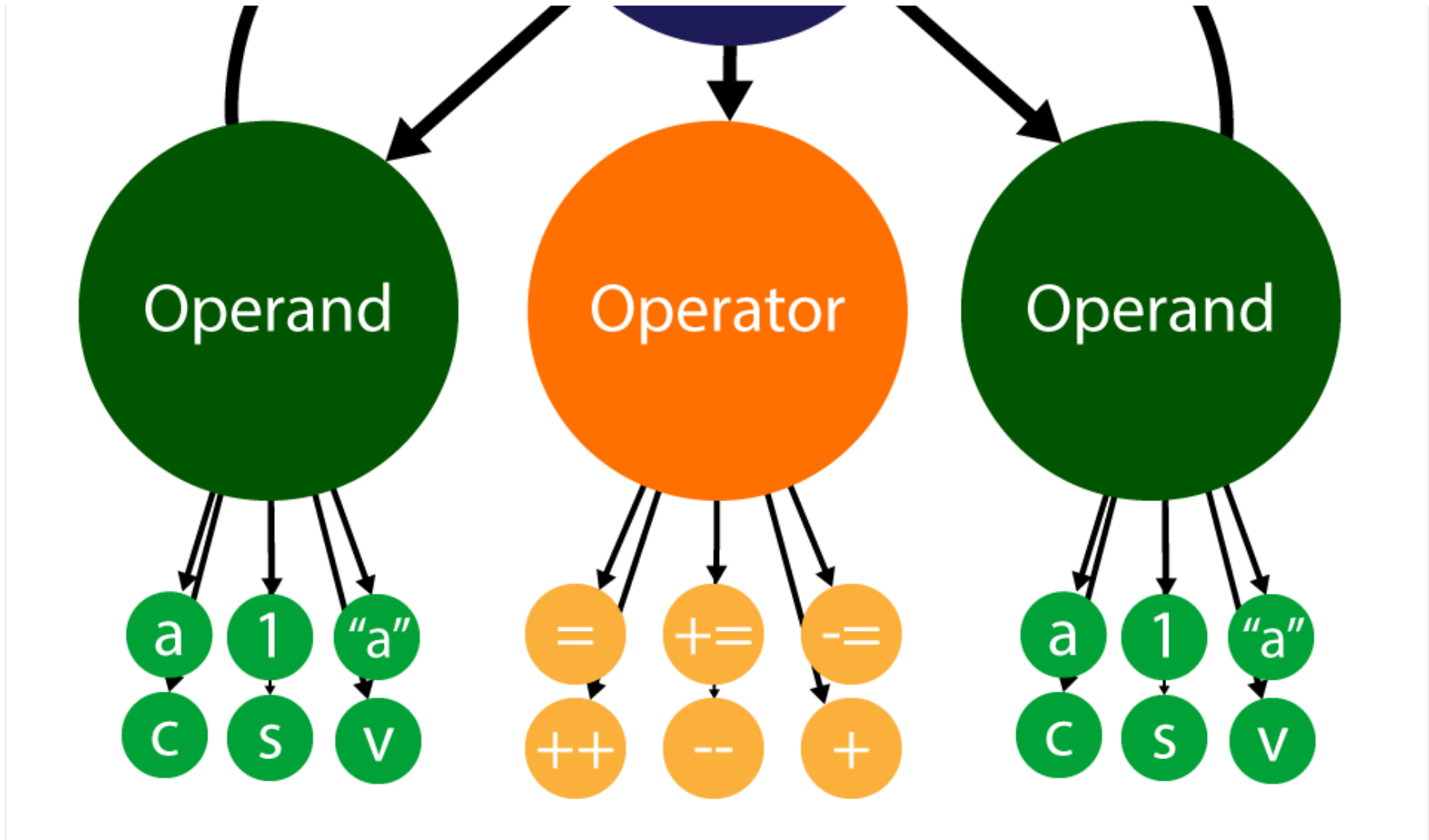
Let's first look at value categories.

Expressions and Value Categories

All C++ programmers work with value categories but may not be able to describe which category a value belongs.

Value categories are often defined in terms of the result of an *expression*. An example of an expression is [assignment](#), [increment and decrement](#), [arithmetic](#), [logical](#), [function calls](#), [etc](#). An expression consists of one or more operands and one operator (or two operators in the case of the ternary operator). Expressions are usually terminated with a semicolon (`;`) but can also be nested inside other expressions or used inside conditional constructs (like `if`, `while`, and `for` loops).

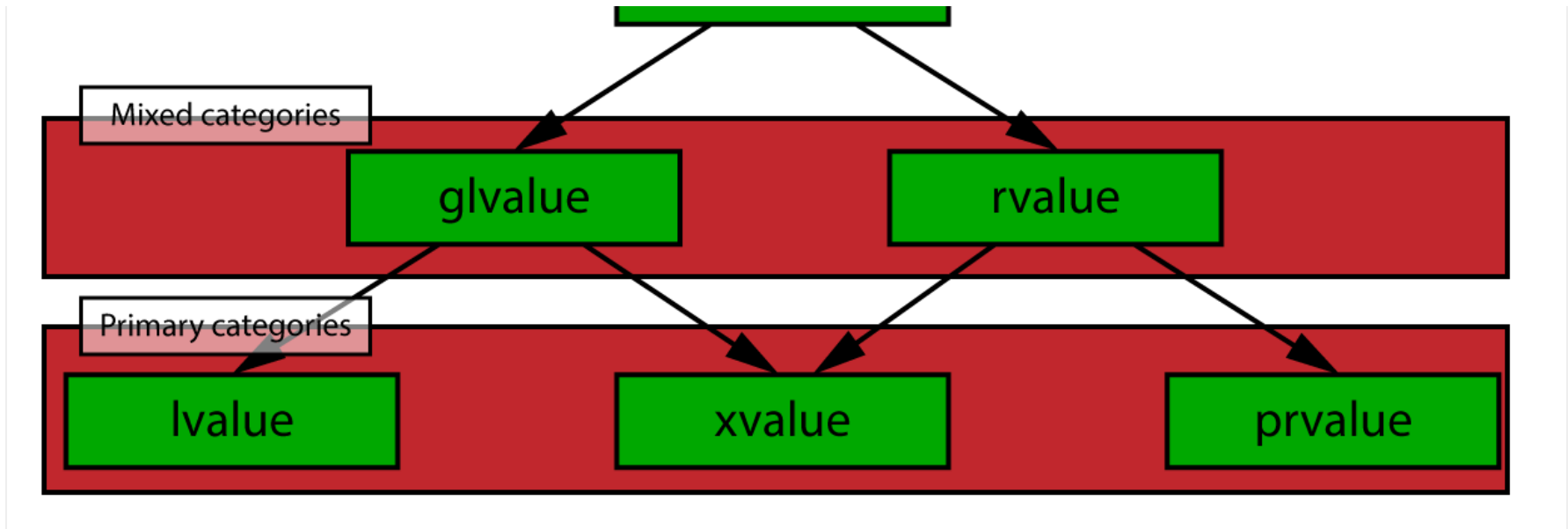




- A C++ Expression is a sequence of operators and operands that produce a result.

Expressions are categorized according to the following taxonomy:

Expressions



- Expressions are categorized according to the following taxonomy.^[6]

According to Stroustrup^[2], value categories can be identified by two independent properties:

1. **“has identity” (i)**: A value is *identified* by a name. Pointers and references also represent identities.
2. **“can be moved from” (m)**: A value can be moved. Expressions that use move semantics like the [move constructor](#) and the [move assignment operator](#) are examples of move expressions.

When a value category has the identity property, it can be denoted with a lower-case **i**. When a value type does not have the identity property, it will be denoted with an upper-case **I**.

Similarly for the move property: if the value category can be moved, it will be denoted with a lower-case **m**, otherwise it will be denoted with an upper-case **M**.

There are five value categories (three primary, and two mixed categories) that are discussed in this section:

- Primary categories
 - lvalue

- prvalue
- xvalue
- Mixed categories
 - glvalue
 - rvalue

lvalue, prvalue, and xvalue are primary value categories because they are mutually exclusive. glvalue and rvalue categories are mixed categories because they are a generalization of the primary value categories.

LVALUE

An *lvalue* is a value that is named by an identifier (**i**) but cannot be moved (**M**). An lvalue is something that has a location in memory. For this reason, lvalues are sometimes referred to as *locator values*.

The following code snippet shows some examples of lvalues.

lvalues	C++
1	
2	
3	<code>int i = 3;</code> // i is an lvalue.
4	
	<code>std::string s = "Hello, world!";</code> // s is an lvalue.
	<code>int* p = nullptr;</code> // p is an lvalue.
	<code>int& r = *p;</code> // r is an lvalue.

It might be tempting to think of lvalues as something that only appears on the left-hand side of an assignment operator, but this is not a good way to think of them. For example, a `const` value is an lvalue, but it cannot be assigned to after initialization.

immutable lvalues
1


```

2
const int i = 3; // i is an lvalue.

i = 4;           // error: assignment of read-only variable 'i'

```

Of course, lvalues can also appear on the right side of the assignment operator.

assign from lvalue	C++
<pre> 1 2 const int i = 3; // i is an lvalue. int j = i; // i is an lvalue on the right side of the assignment operator. </pre>	

However, calling `i` an lvalue in this context (when it appears on the right side of the assignment operator) is not entirely correct. In this case, `i` is implicitly cast to an rvalue (the contents of `i`) which is what gets assigned to `j`^[4].

i An easy way to remember if something is an lvalue is to ask the question “is it possible to take the address of this value?”. If the answer is yes, then the value is an lvalue.

PRVALUE

A *prvalue* is a pure rvalue. Pure rvalues do not have an identifier (I) but can be moved (m). Literal values are examples of prvalues.

prvalues	C++
<pre> 1 2 3 int i = 3; // 3 is a prvalue. 4 std::string s = "Hello, world!"; // "Hello, world!" is a prvalue. </pre>	

```
int* p = nullptr;           // nullptr is a prvalue.

bool b = true;              // true is a prvalue.
```

It is possible to assign a prvalue to an lvalue (as demonstrated in the previous code example) but it is not possible to assign an lvalue to a prvalue.

can't assign a value to a prvalue

C++

```
1
2
3 int i = 3; // prvalue 3 is assigned to lvalue i.

int j = 4; // prvalue 4 is assigned to lvalue j.

3 = i + j; // error: lvalue required as left operand of assignment
```

It is also an error to try to get the address of a prvalue.

Address of prvalues

C++

```
1
2
3 bool* b = &true;           // error: lvalue required as unary '&' operand

int* i = &3;                 // error: lvalue required as unary '&' operand

int* j = &( 3 + 4 ); // error: lvalue required as unary '&' operand
```

The result of an expression using built-in operators are also prvalues.

built-in operators return prvalues

```
1
```

```
2
3 (1 + 3);           // prvalue

(true || false); // prvalue

(true && false); // prvalue
```

i It is possible to override the built-in operators to return non-prvalues, but that is not important in order to describe value categories.

One special thing to note is that prvalues can be candidates for move operations. The following example is valid:

prvalues can be moved

C++

```
1
2
3 int MoveMe(int&& i) // MoveMe takes an rvalue reference.
4
5 {
6
7     int j = i;
8
9     return j;
10
11 }
12

int main()
{

    int i = 0;
```

```

    i = MoveMe(3); // prvalue (3) is used where an rvalue reference is expected.

    return 0;
}

```

This code compiles fine since the prvalue (`3`) is moved into the `i` parameter in the `MoveMe` function.

XVALUE

An *xvalue* are values that are named using an identifier (`i`) and are movable (`m`). Any function that returns an rvalue reference (such as `std::move`) is an xvalue expression. xvalues are the result of casting an lvalue to an rvalue reference as shown in the following example:

xvalues	C++
1	
2	
<pre> int i = 0; // i is an lvalue. int&& j = std::move(i); // The result of std::move is an xvalue. </pre>	

The result of `std::move` is an xvalue and can be assigned to an rvalue reference.

The xvalue is referred to as an “expiring value” since it is used to refer to an object that is expiring since its resources are likely only going to be moved to another object soon^[5].

The xvalue value category is somewhat esoteric and likely only important for compiler writers and people working on the C++ standard documentation. For this article, it’s only important to know of its existence.

GLVALUE

A *glvalue* is a “generalized lvalue”^[2]. glvalues can be either an lvalue or an xvalue. You can think of it as a movable lvalue. glvalues can also be named by an identifier (`i`). Some examples of glvalues are:

glvalues	C++
<pre> 1 2 3 int i; // i is a glvalue (lvalue) 4 int* p = &i; // p is a glvalue (lvalue) int& f(); // the result of f() is a glvalue (lvalue) int&& g(); // the result of g() is an glvalue (xvalue) </pre>	

A glvalue is anything that is not a prvalue.

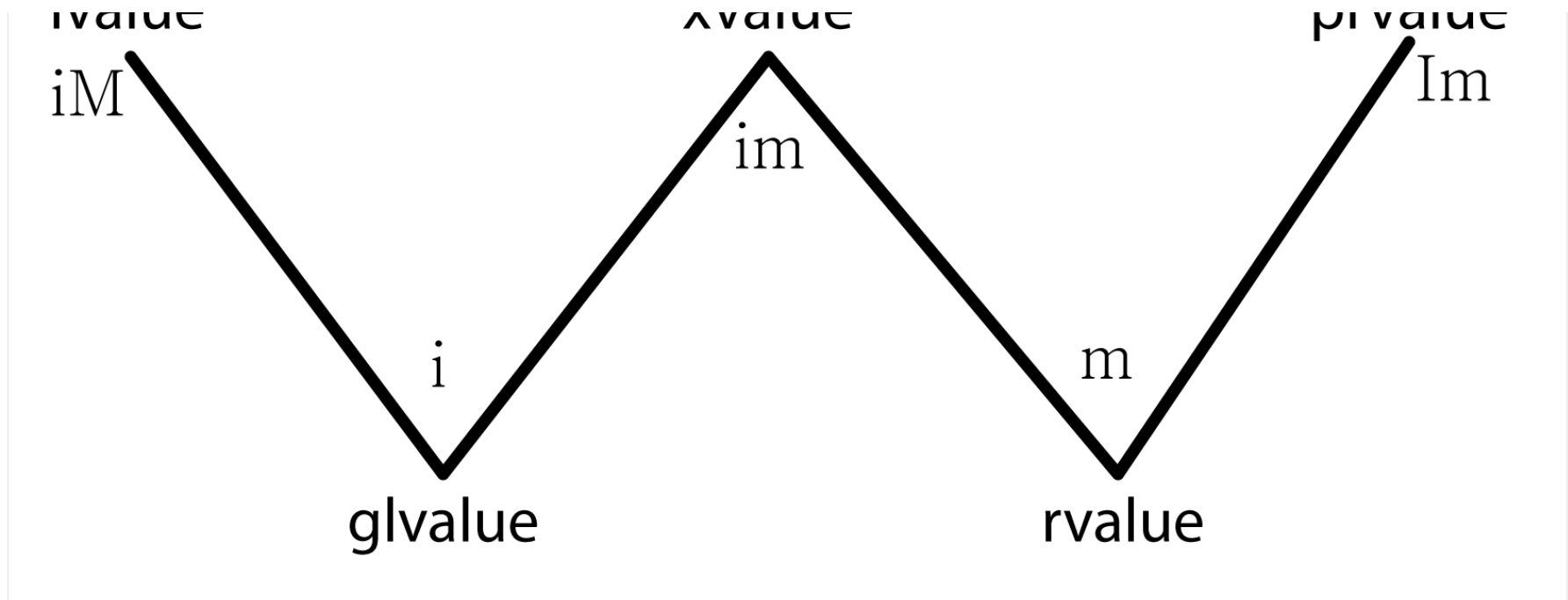
RVALUE

An *rvalue* is a generalization of prvalues and xvalues. An rvalue is not named with an identifier (**I**) but can be moved (**m**). To avoid ambiguity with prvalues, rvalues are only denoted with the lower-case (**m**) to indicate the value “can be moved from”.

rvalue	C++
<pre> 1 2 3 int i = 3; // 3 is an rvalue (prvalue). std::string s = "Hello, world!" // "Hello, world!" is an rvalue (prvalue) int&& g(); // The result of g() is an rvalue (xvalue) </pre>	

An rvalue is anything that is not an lvalue.





- This image shows the various value categories and their corresponding properties as depicted by Bjarne Stroustrup^[2].

Template Arguments versus Template Parameters

Suppose we have the following template class:

Array.h

C++

```
1
2
3  template<typename T, size_t N>
4
5
6  class Array
7
8  {
9
10
11  public:
12
```

```
13
14     Array()
15
16         : m_Data{}
17
18
19
20     {}
21
22
23
24
25     size_t size() const
26
27
28     {
29
30
31         return N;
32
33     }
34
35
36
37
38     T& operator[](size_t i)
39
40     {
41
42         assert(i < N);
43
44
45         return m_Data[i];
46
47     }
```

```
    const T& operator[](size_t i) const  
  
    {  
  
        assert(i < N);  
  
        return m_Data[i];  
  
    }  
  
private:  
  
    T m_Data[N];  
  
};
```

And we also have the following instantiation of the template class:

main.cpp

C++

```
1  
2  
3 Array<float, 3> Position;  
  
    Array<float, 3> Normal;
```



```
Array<float, 2> TexCoord;
```

In this example, the template parameters are `T` and `N` and the type template argument is `float` and the *non-type* template arguments are `3`, and `2`. You can say that “*parameters* are initialized by *arguments*”.

Unlike function arguments, value of non-type template arguments must be determined at compile-time and the definition of a template with its arguments is called the *template-id*.

Each parameter in a template parameter list can be one of following types:

- A type template parameter
- A non-type template parameter
- A template template parameter

The `Array` class template demonstrates the use of both type (`T`), and non-type (`N`) template parameters. In the next section, all three parameter types are explored.

TYPE TEMPLATE PARAMETER

The most common template parameter is a *type template parameter*. A type template parameter starts with either `typename` or `class` and (optionally) followed by the parameter name. The name of the parameter is used as an alias of the type within the template and has the same naming rules as an identifier used in a [typedef](#) or a [type alias](#).

A type template parameter may also define a default argument. If a type template parameter defines a default value, it must appear at the end of the parameter list.

A type template parameter can also be a *parameter pack*. A parameter pack starts with `typename...` (or `class...`) and is used to list an unbounded number of template parameters. Since parameter packs apply to all template parameter categories, parameter packs are discussed in the section about [template parameter packs](#).

The following example demonstrates the use of type template parameters.

```

1
2 // typename introduces a type template parameter.
3
4
5 template<typename T> class Array { ... };
6
7
8
9 // class also introduces a type template parameter.
10
11
12 template<class T> struct MyStruct { ... };
13
14
15
16
17 // A type template parameter with a default argument.
18
19
20 template<typename T = void> struct RType { ... };
21
22
23
24 // The parameter name is optional.
25
26
27 template<typename> struct Tag { ... };

```

NON-TYPE TEMPLATE PARAMETER

Non-type template parameters can be used to specify a *value* rather than a *type* in the template parameter list.

Non-type template parameters can be:

- An integral type (`bool`, `char`, `int`, `size_t`, and unsigned variants of those types)
- An enumeration type
- An lvalue reference type (a references to an existing object or function)
- A pointer type (a pointer to an existing object or function)
- A pointer to a member type (a pointer to a member object or a member function of a class)

- `std::nullptr_t`

C++20 also adds floating-point types and literal class types (with some limitations explained [here](#)) to the list of allowed non-type template parameters.

Similar to type template parameters, the name of the template parameter is optional and non-type template parameters may also define default values.

The following shows examples of non-type template parameters:

Non-Type Template Parameters	C++
<pre>1 2 // Integral non-type template parameter. 3 4 5 template<int C> struct Integral {}; 6 7 8 9 // Also an Integral non-type template parameter. 10 11 12 template<bool B> struct Boolean {}; 13 14 15 16 // Enum non-type template parameter 17 18 19 template<MyEnum E> struct Enumeration {}; 20 21 22 23 // lvalue reference can also be used as a non-type template parameter. 24 25 26 template<const int& C> struct NumRef {}; 27 28</pre>	

```
// lvalue reference to an object is also allowed.

template<const std::string& S> struct StrRef {};


// Pointer to function

template<void(*Func)()> struct Functor {};


// Pointer to member object or member function.

template<typename T, void(T::*Func)()> struct MemberFunc {};


// std::nullptr_t is also allowed as a non-type template parameter.

template<std::nullptr_t = nullptr> struct NullPointer {};


// Floating-point types are allowed in C++20

template<double N> struct FloatingPoint {};


// Literal class types are allowed in C++20.

template<MyClass C> struct Class {};
```

Non-type template parameters must be constant values that are evaluated at compile-time.

TEMPLATE TEMPLATE PARAMETER

Templates can also be used as template parameters:

Template Template Parameter	C++
<pre>1 2 // C is a template class that takes a single type template parameter. 3 4 5 template<template<typename T> class C> struct TemplateClass {}; 6 7 8 9 // C is a template class that takes a type and non-type template parameter. 10 11 12 template<template<typename T, size_t N> class C> struct ArrayClass {}; 13 14 15 // keyword typename is allowed in C++17. 16 17 18 template<template<typename T> typename C> struct TemplateTemplateClass {};</pre>	

Note that until C++17, unlike type template parameters, template template parameters can only use the keyword `class` and not `typename`.

TEMPLATE PARAMETER PACK

A *template parameter pack* is a placeholder for zero or more template parameters. A template parameter pack can be applied to type, non-type, and template template parameters but the parameter types cannot be mixed in a single parameter pack.

A few examples of template parameter packs:

Template Parameter Pack	C++
-------------------------	-----

```

1
2 // Function template with a parmeter pack.
3
4
5 template<typename... Args> void func(Args... args);
6
7
8
9 // Type template parameter pack.
10
11 template<typename... Args> struct TypeList {};

// Non-type template parameter pack.

template<size_t... Ns> struct Integrallist {};

// Template template parameter pack.

template<template<typename T> class... Ts> struct TemplateList {};

```

A pack that is followed by an ellipsis expands the pack. A pack is expanded by replacing the pack with a comma-separated list of the template arguments in the pack.

For example, if a function template is defined with a parameter pack:

Template Parameter Pack

C++

```

1
2
3 template<typename... Args>
4
5 void func(Args... args)

```

```
{  
  
    std::tuple<Args...> values(args...);  
  
}
```

And invoking the function:

Template Parameter Pack
1 func(4, 3.0, 5.0f);

C++

Will result in the following expansion:

Template Parameter Pack
1 2 3 void func(int arg1, double arg2, float arg3) 4 { std::tuple<int, double, float> values(arg1, arg2, arg3); }

C++

More examples of using template parameter packs are shown later in the section about [variadic templates](#).

Basics

The following sections introduce the basics of templates. If you are already familiar with templates, then you may want to skip to the next section.

Function Templates

Function templates provide a mechanism to define a function that operates on different types. Function templates look like ordinary functions but start with the `template` keyword followed by a list of (one or more) template parameters surrounded by angle brackets.

Function Template	C++
<pre>1 2 3 template<typename T> 4 5 T max(T a, T b) 6 7 { 8 9 return a > b ? a : b; 10 11 }</pre>	

The `max` function accepts a single template parameter `T`. The `max` function template defines a *family* of functions that can take different types. The type is defined when the function is invoked either by explicitly specifying the type or by allowing the compiler to deduce the type:

Function Template	C++
<pre>1 2 3 int m = max<int>(3, 5); // Explicit type. 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99</pre>	

On the first line, `int` is explicitly provided as the template argument. On the second line, the template argument is not specified but the compiler automatically deduces it as `int` because the `3` and the `5` are deduced as `int`. In both cases, the same function is instantiated.

Implicit template type deduction does not work if you want to mix types as in the following case:

Function Template

C++

```
1 double x = max(3, 5.0);
```

In this case, the `max` function template is being instantiated with `3` (`int`) and `5.0` (`double`) and the compiler does not know how to implicitly determine the template argument. In this case, the compiler will generate an error. There are a few ways this can be fixed:

1. Use an explicit template argument
2. Cast all function arguments to the same type
3. Multiple template parameters

Explicitly specifying the template arguments will ensure that all of the parameters are cast to the correct type through implicit casting:

Function Template

C++

```
1 double x = max<double>(3, 5.0);
```

`3` is implicitly cast to a `double`. In this case, the compiler may not even issue a warning since this type of cast does not cause any truncation of the original type. However, if a narrowing conversion occurs, the compiler will very likely generate a warning. To avoid this warning, an explicit cast can be used:

Function Template

C++

```
1 int x = max(3, static_cast<int>(5.0));
```

In this example, an explicit cast is used to convert `5.0` from a `double` to an `int`. The compiler no longer generates a warning and `T` is implicitly deduced to `int`.

The other solution to this problem is to allow `a` and `b` to be different types:

Function Template

C++

```
1  
2  
3 template<typename T, typename U>
```

```

4
5
T max(T a, U b)

{

    return a > b ? a : b;

}

```

Now `a` and `b` can be different types and we can call the function template with mixed types without the compiler issuing any warnings... right? What about the return type? If `T` is “narrower” than `U` then the compiler will have to perform a narrowing conversion again and likely issue a warning when this happens. So what should the return type be? Is it possible to let the compiler decide?

Since the compiler will do anything to prevent data loss, it will try to convert all arguments to the largest (widest) type before performing the comparison and the return type will be the widest type. We can use the **auto return type deduction**, **trailing return type**, and the **decltype** specifier to automatically determine the safest type to use for the return value of the function template:

Function Template

C++

```

1
2
3 template<typename T, typename U>
4
5
auto max(T a, U b) -> decltype(a > b ? a : b)

{

    return a > b ? a : b;

}

```

i The trailing return type can be omitted in C++14 and higher.

This version of the `max` function template may still generate a warning about returning a reference to a temporary, **!** but we can use type traits to avoid this. Type traits are discussed later so I won't complicate this example more than necessary for now.

Using this version of the function template, any type can be used for `a` and `b` and the return value is the widest type of `a` or `b`. For example:

Function Template

C++

```
1 auto x = max(3.0, 5);
```

`x` is automatically deduced to `double` because comparing `3.0` (`double`) and `5` (`int`) results in a conversion to `double` and the `max` function template returns `double`.

The `decltype` specifier is explained in more detail later.

There is also a solution to determine the return type using `std::common_type` but requires knowledge of type traits which is discussed later.

Class Templates

Similar to **Function Templates**, classes can also be parameterized with one or more template parameters. The most common use case for class templates are containers for other types. If you have used any of the container types in the **Standard Template Library** (STL) (such as `std::vector`, `std::map`, or `std::array`), then you have already used class templates. In this section, I describe how to create class templates.

Consider the `Array` class template from the previous section. Here it is again for clarity:

Array.h

```
1  
2
```

```
3  template<typename T, size_t N>
4
5
6  class Array
7
8  {
9
10
11  public:
12
13
14      Array()
15
16          : m_Data{}
17
18
19
20      {}
21
22
23
24
25      size_t size() const
26
27      {
28
29
30          return N;
31
32      }
33
34
35      T& operator[](size_t i)
36
37      {
```

```
    assert(i < N);
```

```
    return m_Data[i];
```

```
}
```

```
const T& operator[](size_t i) const
```

```
{
```

```
    assert(i < N);
```

```
    return m_Data[i];
```

```
}
```

```
private:
```

```
    T m_Data[N];
```

```
};
```

i A class template that doesn't specialize any template parameters is called the **primary template**.

The **Array** class template demonstrates two kinds of template parameters:

1. Type template parameters (denoted with **typename** or **class**)
2. Non-type template parameters (denoted with an integral type such as **size_t**)

The **Array** class template defines a simple container for a static (fixed-size) array similar to the **std::array** implementation from the STL.

Inside the **Array** class template, **T** can be used wherever a type is expected (such as the declaration of the **m_Data** member variable or the return value of a member function) and **N** can be used wherever the number of elements is required (such as in the **assert**'s in the index operator member functions).

A class template is instantiated when a variable that uses the class is defined:

Class Template	C++
<pre>1 2 Array<float, 3> Position; Array<float, 2> TexCoord;</pre>	

Here, **Array<float, 3>** represents the *type* of the **Position** variable and **Array<float, 2>** is the *type* of the **TexCoord** variable. Although both types are instantiated from the same class template, they are in no way related. You cannot use **Array<float, 3>** where an **Array<float, 2>** is expected. For example, the following code will not compile:

Class Template	C++
<pre>1 2 3 Array<float, 2> add(const Array<float, 2>& a, const Array<float, 2>& b) 4 5 {</pre>	

```
6
7
8     Array<float, 2> c;
9
10
11
12
13     c[0] = a[0] + b[0];
14
15
16     c[1] = a[1] + b[1];


    return c;
}


...


Array<float, 3> Position1;

Array<float, 3> Position2;


auto Position3 = add(Position1, Position2); // Error: Array<float, 3> is not compatible with Array<float, 2>
```

Although this is a pretty contrived example, it demonstrates that different combinations of template arguments create different (unrelated) types.

Variable Templates

Variable templates were added to the C++ standard with C++14. Variable templates allow you to define a family of variables or static data members of a class using template syntax.

Variable Templates	C++
<pre>1 2 template<typename T> constexpr T pi = T(3.1415926535897932384626433832795L);</pre>	

The variable template `pi` can now be used with varying levels of precision:

Variable Templates	C++
<pre>1 2 3 std::cout << std::setprecision(30); 4 std::cout << PI<int> << std::endl; std::cout << PI<float> << std::endl; std::cout << PI<double> << std::endl;</pre>	

Will print:

Variable Templates	C++
<pre>1 2 3 3 3.1415927410125732421875 3.14159265358979311599796346854</pre>	

Variable templates can also have both type and non-type template parameters:

Variable Templates	C++
<pre>1 2 template<typename T, T N> constexpr T integral_constant = N;</pre>	

T is a type template parameter and **N** is a non-type template parameter of type **T**.

It's important to understand that a variable template does not define a *type*, but rather a *value* that is evaluated at compile-time.

Variable Templates	C++
<pre>1 2 integral_constant<int, 3> i; // ERROR integral_constant<int, 3> is not a type. auto i = integral_constant<int, 3>; // OK: i is an int with the value 3.</pre>	

Variable templates can also be specialized:

Variable Templates	C++
<pre>1 2 3 template<size_t N> 4 5 6 constexpr size_t Fib = Fib<N-1> + Fib<N-2>; 7</pre>	

8

```
template<>
```

```
constexpr size_t Fib<0> = 0;
```

```
template<>
```

```
constexpr size_t Fib<1> = 1;
```

The `Fib` variable template computes the N^{th} Fibonacci number.

Variable Templates

C++

1

```
std::cout << Fib<10> << std::endl;
```

This will print `55` to the console.

Variable templates can also be used as a limited form of type traits:

Variable Templates

C++

1

2

```
3 template<typename T>
```

4

5

```
6 constexpr bool is_integral = false;
```

7

8

9

10

```
11 template<>
```

12

```
13
constexpr bool is_integral<short> = true;

template<>

constexpr bool is_integral<int> = true;

template<>

constexpr bool is_integral<long> = true;

// ... specialized for all other integral types.
```

If `T` is an integral type then `is_integral<T>` is `true`. For all other types, `is_integral<T>` is `false`.

 Type-traits are discussed in more detail later in the article.

Alias Template

Templates can be aliased using the `using` keyword:

Alias Template

C++

```
1
2
3 template<typename T, T v>
4
5     constexpr T integral_constant = v;
```

```
template<bool v>

using bool_constant = integral_constant<bool, v>;
```

On line 5, `bool_constant` is defined as an alias template of the `integral_constant` variable template where `T` is `bool`. The value `v` remains open as a non-type template parameter.

typename Keyword

Besides being used as the keyword used to introduce a [type template parameter](#), the `typename` keyword is also used in a class or function template definition to declare that a type is dependent on a template parameter.

For example, suppose we have the following class template:

typename Keyword	C++
1	
2	
3	<code>template<typename T></code>
4	
5	<code>struct MyClassTemplate</code>
	<code>{</code>
	<code>using type = T;</code>
	<code>};</code>

The `MyClassTemplate` class template has a single template parameter (`T`) and `type` is a [type alias](#) of `T`.

Now suppose we have a function template that uses `MyClassTemplate`:

```
typename Keyword C++
1
2
3 template<typename U>
4
5
6 U MyFuncTemplate(U a)
7
8 {
9
10     MyClassTemplate<U>::type b; // ERROR: Use of dependent type must be prefixed with 'typename'
11
12     b = a;
13
14     return b;
15 }
```

The `MyFuncTemplate` function template has a single template parameter (`U`) and declares a local variable (`b`) whose type is `MyClassTemplate<U>::type`. Since `MyClassTemplate<U>::type` names a type and that type is *dependent* on the template parameter (`U`), then `MyClassTemplate<U>::type` must be preceded by `typename`:

```
typename Keyword C++
1
2
3 template<typename U>
4
5
6 U MyFuncTemplate(U a)
7
8 {
9
10     typename MyClassTemplate<U>::type b; // OK: Use of dependent type is prefixed with 'typename'
11 }
```

```
    b = a;

    return b;
}
```

The need for the `typename` keyword in this case, can be avoided by using an [alias template](#):

typename Keyword

C++

```
1
2
3  template<typename T>
4
5
6  using MyClassTemplate_t = typename MyClassTemplate<T>::type;
7
8
9
10
    template<typename U>


    U MyFuncTemplate(U a)
    {

        MyClassTemplate_t<U> b; // OK: MyClassTemplate_t names a type.

        b = a;

        return b;

    }
```

 The `typename` keyword is used to name a type that is dependent on a template parameter unless it was already established as a type (by using a `typedef` or a (template) type alias).

Template Specialization

Both function templates and class templates can be specialized for specific types. When all template parameters are specialized, then it is called fully specialized. Suppose we have the following function template definition:

Template Specialization

C++

```
1
2
3 template<typename T, typename U>
4
5 auto add(T a, U b) -> decltype(a + b)
{
    return a + b;
}
```

The function template can be specialized by declaring the function with an empty template parameter list `template<>` and specifying the specialized template arguments after the function name:

Function Overloading

C++

```
1
2
3 template<>
4
5 double add<double, double>(double a, double b)
{
    return a + b;
}
```

```
    return a + b;
}
```

All occurrences of template parameters (**T** and **U**) in the function must also be replaced with the specialized template arguments (**double**).

It is possible to provide the same functionality by using function overloading:

C++

```
1
2
3 double add(double a, double b)
4
i  {
    return a + b;
}
```

It is perfectly legal to overload function templates in this way.

The compiler will use the specialized (or overloaded) version of the function template if all of the substituted template arguments (either explicitly or implicitly) match the specialized version:

Template Specialization

C++

```
1
2
float a = add(3.0f, 4.0f); // Uses generic version.

double b = add(3.0, 4.0); // Uses specialized version for doubles.
```

Similar to function templates, class templates can also be specialized. If we take the **Array** class template from the

Template section and we want to specialize it for 4-component floating-point values:

Template Specialization

C++

```
1
2
3  template<>
4
5
6  class Array<float, 4>
7  {
8
9
10
11 public:
12
13
14     Array()
15
16
17     : m_Vec(_mm_setzero_ps())
18
19
20     {}
21
22
23
24
25     size_t size() const
26
27
28     {
29
30
31         return 4;
32
33
34     }
```

```
float& operator[](size_t i)
```

```
{
```

```
    assert(i < 4);
```

```
    return m_Data[i];
```

```
}
```

```
const float& operator[](size_t i) const
```

```
{
```

```
    assert(i < 4);
```

```
    return m_Data[i];
```

```
}
```

```
private:
```

```
    union
```

```
{  
  
    __m128 m_Vec;    // Vectorized data.  
  
    float m_Data[4]; // Float data.  
  
};  
  
};
```

The specialized version of the `Array` class template allows you to provide a different implementation of the class depending on its template arguments. In this case, we provide a specialization for `Array<float, 4>` that allows for some SSE optimizations to be made.

It is important to note that if you specialize a class template, you must also specialize all of the member functions of that class. This can be quite cumbersome for large classes, especially if you decide to refactor the generic class template, you must also update all specialized versions of the class.

Keep in mind that the compiler will only generate code for class template member functions *that are used*. That is, if you never call a specific member function of a specialized class template, then no code will be generated for that version of the member function. If a specialized class template defines a member function that just doesn't make any sense for a certain specialized type, and you are sure that the member function is not being used anywhere in the codebase, then you can leave that function undefined in the specialized version of the class template.

Partial Specialization

Although it is not possible to partially specialize function templates, we can achieve something similar by using function template overloading. Let's consider the `max` function template introduced in the previous section on [Function Templates](#). Suppose we want to provide an implementation for pointer types:

Partial Specialization

C++

```

1
2
3 template<typename T, typename U>
4
5 auto max(const T* a, const U* b) -> decltype(*a > *b ? *a : *b)
6
7 {
8
9     return *a > *b ? *a : *b;
10
11 }

```

This version of the function template is used whenever pointers are used as the arguments to the function as in the following example:

Partial Specialization

C++

```

1
2
3 double d = 3.0;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
```

```
1
2
3  template<typename T, size_t N>
4
5
6  class Array<T*, N>
7
8  {
9
10
11  public:
12
13
14      explicit Array(T* data)
15
16          : m_Data(data)
17
18
19
20      {}
21
22
23
24
25      T& operator[](size_t i)
26
27      {
28
29          assert(i < N);
30
31
32          return m_Data[i];
33
34      }
```

```
const T& operator[](size_t i) const

{

    assert(i < N);

    return m_Data[i];

}

private:

    T* m_Data;

};
```

A class template can be partially specialized by specifying the `template` keyword followed by a list of template parameters surrounded by angle brackets, just as with the non-specialized class template. The specialized template parameters are specified after the class name (`T*` and `N` in this case).

This implementation of the `Array` class template will be used when `T` is a pointer type. This may not seem like a very useful thing, but now we have a class template that can provide all the functionality of the original `Array` class template, but instead of allocating a static array, it now works with arbitrary data that is allocated elsewhere.

```
2
3 float p[16] = {};
4
   auto a = Array<float*, 16>(p);
   // a provides all the functionality of Array on arbitrary data.
   a[8] = 1.0f;
```

Similarly to a fully specialized class template, if one of the template parameters is fully defined then it does not need to be listed in the template parameter list, but must be specified in the template argument list (after the class name):

Partial Specialization

C++

```
1
2 // Specialized for arrays of size 4 and arbitrary type.
3
4
5 template<typename T>
6
7
8 class Array<T, 4>
9
10 {
11
12
13     ...
    };
    // Specialized for float arrays of arbitrary size.
    template<size_t N>
```

```
class Array<float, N>
{
    ...

};
```

Partial template specialization is the cornerstone of type traits and SFINAE.

Variadic Templates

Variadic templates are function templates or class templates that can accept an unbounded number of template arguments.

For example, suppose we want a function that creates an `std::unique_ptr` from an arbitrary type:

Variadic Templates

C++

```
1
2
3 template<typename T, typename... Args>
4
5 std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

i For some reason, `std::make_shared` was introduced in C++11 but `std::make_unique` wasn't introduced until C++14. This example provides a possible implementation of `std::make_unique` for C++11 compilers.

There are a few things to note here:

1. The template parameter list contains a *template parameter pack* in the form of `typename... Args`
2. An arbitrary number of arguments are passed to the function in the form of `Args&&...`. Not to be mistaken by an *rvalue reference*, this is called a *forwarding reference* which preserves the value category of the function arguments when used in conjunction with `std::forward`
3. The arguments are unpacked by performing a *pack expansion* which replaces the parameter pack by a comma-separated list of the arguments using the pattern immediately preceding the `...` (ellipsis)

For example, suppose we have the following class:

Object.h

C++

```
1
2
3 class Object
4 {
5
6
7
8 public:
9
10
11     Object(int i, float f, int* ip, double d)
12
13
14     : m_i(i)
15
16
17     , m_f(f)
18
19
20     , mp_i(ip)
21
22
23     , m_d(d)
24
25
26 {}
```

```
private:

    int m_i;

    float m_f;

    int* mp_i;

    double m_d;

};
```

And if the `make_unique` function template was invoked with:

Variadic Templates

C++

```
1
2
3 int i = 3;
4
5
   float f = 4.0f;

   double d = 6.0;

auto o = make_unique<Object>(i, f, &i, d);
```

Then the `make_unique` function would generate something like this:

Variadic Templates

C++

```
1
2
3 std::unique_ptr<Object> make_unique(int args0, float args1, int* args2, double args3)
4
5 {
6
7     return std::unique_ptr<Object>(new Object(std::forward<int>(args0), std::forward<float>(args1), std::forward<int*>(args2), std::forward<double>(args3)));
8
9 }
```

Recursive Variadic Templates

Suppose you want to write a function that prints an arbitrary number of arguments to the standard output stream. Using a C++17 compiler, this can be accomplished using *fold expressions*:

Variadic Templates

C++

```
1
2
3 template<typename... Args>
4
5 void print(Args... args)
6
7 {
8
9     (std::cout << ... << args) << std::endl;
10
11 }
```

But how could this be accomplished without a C++17 compiler? To accomplish this without fold expressions, we need to create a recursive template function. To do this we must:

1. Define the base case (only a single template parameter)
2. Define the recursive case (where multiple template parameters are passed in a parameter pack)

First, let's define the case where only a single argument is passed:

Base case

C++

```
1
2
3 template<typename Arg>
4
5 void print(Arg arg)
6
7 {
8
9     std::cout << arg << std::endl;
10
11 }
```

And the recursive case with a parameter pack:

Recursive case

C++

```
1
2
3 template<typename Arg, typename... Args>
4
5 void print(Arg arg, Args... args)
6
7 {
8
9     std::cout << arg;
10
11     print(args...);
12
13 }
```

The subtle trick here is that the recursive case has two template parameters:

1. `typename Arg`
2. `typename... Args`

This way, the first argument can be extracted from the parameter pack and the rest of the arguments are passed to the recursive `print` function. When there is only a single argument left in the parameter pack, the base case (with a single template argument) is used.

At this point, you should have a pretty good idea of how to use templates in your code. In the following sections, I will show a few more complex uses of templates.

Template Type Deduction

Template type deduction is the process the compiler performs to determine the type that is used to instantiate a function or class template. Many programmers use templates with a reasonable amount of success without really understanding how template type deduction works. This might be sufficient for simple use cases but becomes complicated (and perhaps unintuitive) in more complex applications of templates.

Understanding template type deduction forms the foundation for understanding how the `decltype` specifier works.

Scott Meyers provides a very good explanation of how type deduction works^[7]. I will attempt to summarize Scott Meyers' explanation here.

As we've seen in previous examples, function templates have the following basic form:

Template Type Deduction	C++
<pre>1 2 template <typename T> void f(ParamType param);</pre>	

In the snippet above, `T` and `ParamType` may be different in the case that `ParamType` has modifiers (`const`, pointer (`*`), reference (`&`) qualifiers). For example, if the template is declared like this:

Template Type Deduction	C++
<pre> 1 2 template<typename T> void f(const T& param); // ParamType is const T& </pre>	

Now suppose the template function is invoked like this:

Template Type Deduction	C++
<pre> 1 2 int x = 0; f(x); // Call f with an int (lvalue) </pre>	

In this case, `T` is deduced to `int` and `ParamType` is deduced to `const int&`.

In this case, it seems obvious that `T` is deduced to `int` since `f` was invoked with an `int` argument, but it's not always that obvious. The type deduced for `T` is dependent on not only the argument type, but also the form of `ParamType`. There are three forms of `ParamType` that must be considered:

1. `ParamType` is a pointer or reference type, but not a forwarding reference
2. `ParamType` is a forwarding reference
3. `ParamType` is neither a pointer nor a reference

Scott Meyers uses the term *universal reference* to refer to the double ampersand (`&&`) being applied to template parameters (not to be mistaken with rvalue references). Since the C++ standard uses the term *forwarding reference*, I will use that term in this article.

i A forwarding reference is a special kind of reference that preserve the value category of a (function) argument

making it possible to forward it to another function using `std::forward`.

For each of the three cases, consider the general form of invoking the template function:

Template Type Deduction	C++
<pre>1 2 3 template <typename T> 4 void f(ParamType param); f(expr); // Deduce T and ParamType from expr.</pre>	

Case 1: ParamType is a Reference or Pointer

In the first case, `ParamType` is a reference or pointer type, but not a [forwarding reference](#). In this case, type deduction works like this:

1. If `expr` evaluates to a reference, ignore the reference part.
2. Then match `expr`'s type against `ParamType` to determine `T`.

For example, if the function template is declared like this:

Template Type Deduction	C++
<pre>1 2 template<typename T> void f(T& param); // param is a reference.</pre>	

Then given the following variables:

Template Type Deduction

C++

```
1
2
3 int i          = 3; // i is an int.
4
5
6 const int ci   = i; // ci is a const int.
7

    const int& rci = i; // rci is a reference to a const int.


f(i);    // T is int, param's type is int&

f(ci);   // T is const int, param's type is const int&

f(rci);  // T is const int, param's type is const int&
```

Notice on lines 6, and 7 where `expr` is a `const int` or `const int&`, then `T` is deduced to be `const int`. The constness of `expr` becomes part of the type deduced for `T`.

If, on the other hand, `ParamType` is changed to `const T&` then type deduction works slightly differently:

Template Type Deduction

C++

```
1
2
3 template<typename T>
4
5
6 void f(const T& param); // param is now a reference to const T.
7
8
9
10 int i          = 3; // i is an int.
```



```
const int ci = i; // ci is a const int.

const int& rci = i; // rci is a reference to a const int.

f(i); // T is int, param's type is const int&

f(ci); // T is int, param's type is const int&

f(rci); // T is int, param's type is const int&
```

Since the constness is now part of `param`'s type, there is no need for `const` to be part of `T`'s type deduction.

If `param` were a pointer or a pointer to `const`, then the type deduction for `T` works the same way:

Template Type Deduction

C++

```
1
2
3 template<typename T>
4
5
6 void f(T* param); // param is now a pointer to T.
7
8

int i = 3; // i is an int.

const int* pi = &i; // pi is a pointer to const int.
```

```
f(&i); // T is int, param's type is const int*

f(pi); // T is const int, param's type is const int*
```

This may seem obvious so far, but becomes less obvious if `ParamType` is a [forwarding reference](#).

Case 2: ParamType is a Forwarding Reference

Now let's consider the case where `ParamType` is a [forwarding reference](#):

Template Type Deduction

C++

```
1
2
3  template<typename T>
4
5
6  void f(T&& param); // param is now a forwarding reference.
7
8
9
10
11 int i          = 3; // i is an int (lvalue).

    const int ci   = i; // ci is a const int (lvalue).

    const int& rci = i; // rci is a reference to a const int (lvalue).

    f(i);   // T is int&, param's type is also int&

    f(ci);  // T is const int&, param's type is also const int&
```

```
f(rci); // T is const int&, param's type is also const int&

f(3);   // T is int, param's type is int&&
```

On line 4, an `int` variable (`i`) is defined. This is an `lvalue` (according to the [value category](#) rules defined at the beginning of this article). On line 9, `i` is passed to `f`. In this case, `ParamType` is deduced to `int&` (lvalue reference) and `T` is deduced to `int&` (lvalue reference).

On line 9, `ci` (lvalue) is passed to `f` and `ParamType` is deduced to `const int&` (lvalue reference) and `T` is deduced to `const int&` (lvalue reference). Similar deduction rules are applied to `rci` on line 10.

On line 11, the value `3` (prvalue, which is a primary value category of rvalue) is passed to `f`. In this case `ParamType` is deduced to `int&&` (rvalue reference) and `T` is deduced to `int`. The deduction rules for rvalues are the same as Case 1 above (`expr`'s type is matched against `ParamType` to determine `T`).

The general rules of type deduction when `ParamType` is a [forwarding reference](#), are:

- If `expr` is an lvalue, both `T` and `ParamType` are deduced to be lvalue references.
- If `expr` is a rvalue (prvalue or xvalue), then the rules for [Case 1](#) are applied.

In short, use a [forwarding reference](#) when you need to maintain the value category of the template argument. This is almost *always* the case when the arguments are being forwarded to another function.

Case 3: ParamType is Neither a Pointer nor a Reference

If `ParamType` is neither a pointer nor a reference, then we say that the parameter is *passed-by-value*:

Template Type Deduction

C++

```
1
2
template<typename T>

void f(T param); // param is now passed-by-value.
```

In this case, the rules for type deduction are:

1. If `expr`'s type is a reference, ignore the reference part
2. If `expr`'s type is `const`, ignore that too.

Then we have:

Template Type Deduction	C++
<pre>1 2 3 int i = 3; // i is an int. 4 5 6 const int ci = i; // ci is a const int. 7 const int& rci = i; // rci is a reference to a const int. f(i); // T is int, param's type is also int f(ci); // T is int, param's type is also int f(rci); // T is int, param's type is also int</pre>	

Note that despite `ci` and `rci` being `const` values, `param` doesn't become `const`. Just because `expr` can't be modified doesn't mean that a copy of it can't be.

This pretty much summarizes the rules that are applied during template parameter type deduction. There are a few more cases that can be considered, for example how static arrays and function objects decay to their pointer types (see the [decay](#) trait for more information). I encourage the reader to consult Scott Meyers' books^[7] for more information regarding

edge cases of template parameter type deductions. But at this point, you should have a good foundation for understanding the `decltype` specifier and `std::declval` which is the subject of the next sections.

decltype Specifier

The `decltype` specifier is used to inspect the declared type and value category of an expression.

Earlier, in the section about [Function Templates](#), the `decltype` specifier was used to determine the return type for the `max` function template. If you read the warning that followed the code example, you might know that in certain cases, the compiler will generate a warning about returning a reference to a temporary. But under what circumstances does this happen?

In most cases, the `decltype` produces the expected type:

decltype specifier	C++
<pre>1 2 3 int h = 0; 4 5 6 int* i = &h; 7 8 9 int& j = h; 10 11 12 const int k = 0; 13 const int* l = &k; const int& m = k; decltype(h) n = 0; // n is int</pre>	

```
decltype(i) o = &n; // o is int*

decltype(j) p = n;  // p is int&

decltype(k) q = 0;  // q is const int

decltype(l) r = &q; // r is const int*

decltype(m) s = q;  // s is const int&
```

No surprises here. `decltype` gives the exact type as the provided expression maintaining `const` (and `volatile`), reference (`&`) and pointer (`*`) attributes.

When the expression passed to `decltype` is parenthesized, then the expression is treated as an lvalue and `decltype` adds a reference to the expression:

decltype specifier

C++

```
1
2
3 int h      = 0;
4
5
6 int* i     = &h;
7
8
9 int& j      = h;
10
11
12 const int k = 0;
13

    const int* l = &k;

    const int& m = k;
```

```
decltype((h)) n = h; // n is int&

decltype((i)) o = i; // o is int*&

decltype((j)) p = j; // p is int&

decltype((k)) q = k; // q is const int&

decltype((l)) r = l; // r is const int*&

decltype((m)) s = m; // s is const int&
```

Okay, but this doesn't explain why the `max` function template can sometimes return a reference. To understand when this happens, we need to look at the deduction guide for the ternary (conditional) operator. The ternary operator has the form:

Ternary Operator		C++
1	condition ? <code>expr1</code> : <code>expr2</code>	

First, `condition` is evaluated and (implicitly converted) to `bool`. If the result is `true`, then `expr1` is evaluated. If the result is `false`, then `expr2` is evaluated. The deduction rules for the resulting value of the ternary operator are complex and you can read the full guide [here](#).

One of the rules of the deduction guide for the ternary operator states that if `expr1` and `expr2` are [glvalues](#) ([lvalues](#) or [xvalues](#)) of the the same type and the same value category, then the result has the same type and value category.

Here is the `max` function template again:

```

1
2
3 template<typename T, typename U>
4
5     auto max(T a, U b) -> decltype(a > b ? a : b)
6
7     {
8
9         return a > b ? a : b;
10
11     }

```

So if `T` and `U` are the same type and value category (see [Case 3](#) above when the template parameter is passed-by-value) then `decltype(a > b ? a : b)` will have the same type and value category of both `a` and `b`. If both `a` and `b` are the same type and they are always both lvalues (since they are identified by a name), then in order to avoid creating a temporary, `decltype(a > b ? a : b)` results in an lvalue references that refers to either `a` or `b` (whichever is larger).

Let's take a look at a few examples of this:

declval with the Ternary Operator

C++

```

1
2
3 int a = 3;
4
5
6 int b = 5;
7
8
9 double c = 3.0;

double d = 5.0;

decltype(a > b ? a : b) g; // g is int&

```



```

decltype(a > c ? a : c) h; // h is double

decltype(c > d ? c : d) i; // i is double&

decltype(3 > 5 ? 3 : 5) k; // k is int

```

On line 6, both `a` and `b` are `int`s and they are both lvalues. The result of the ternary operator is an lvalue reference.

On line 7, `a` is an `int` and `c` is a `double`. In this case, they are not the same type and the result of the ternary operator is the type of the widest operand (in this case, `double`)

On line 8, both `c` and `d` are `double`s and they are both lvalues. The result of the ternary operator is an lvalue reference.

On line 9, both operands are prvalues of type `int`. In this case, the result of the ternary operator is also a prvalue.

Consequently, the result of the ternary operator has an interesting side effect that you should be aware of. You can *sometimes* assign a value to the result of the ternary operator:

Result of Ternary Operator

C++

```

1
2
3 int a = 3;
4
5
6 int b = 5;

```

```
( a > b ? a : b ) = 10; // OKAY, b now has the value 10.
```

```
( 3 > b ? 3 : b ) = 10; // ERROR: expression must be a modifiable lvalue.
```

```
( 3 > 5 ? 3 : 5 ) = 10; // ERROR: expression must be a modifiable lvalue.
```

In the case where the the result of the ternary operator is an lvalue reference, you can actually assign a value to that result. In all other cases, the result of the ternary operator is a temporary prvalue that can't be modified directly (unless it is stored in a lvalue first).

Okay, you've probably heard enough about the ternary operator. This should be enough knowledge to know under which circumstance the `max` function template returns a reference, but how can we fix this? In later sections, I'll talk about using type traits to coerce `declval` to give us what we want. But before we get to type traits, there is one more tool we need in our template toolbox, and that's the `std::declval`.

`std::declval()`

`std::declval` is a utility function that converts any type to a reference type without the need to create an instance of an object.

Okay, maybe that doesn't help to understand why we need `std::declval`, so let's take a look at an example. Suppose we have an abstract base class:

```
std::declvalC++
1
2
3 struct Abstract
4 {
5
6     virtual ~Abstract() = default;
7
8     virtual int value() const = 0;
9
10 };

```

We know that `Abstract` is an abstract type because it declares at least one pure virtual function. Pure virtual functions are not required to (but may) have a definition. Classes with pure virtual functions are called *abstract* classes and cannot be instantiated.

Now suppose we wanted to determine the type that is returned from the `Abstract::value` method. As explained in the previous section, we can use the `decltype` keyword for this:

```
declvalC++
1
2
3 decltype(Abstract().value()) a; // ERROR: cannot instantiate abstract class.

    decltype(Abstract::value()) b; // ERROR: illegal call of non-static member function.

    decltype(std::declval<Abstract>().value()) c; // OK: c is type int.
```

On line 10, we try to determine the return type by constructing an instance of `Abstract` and inspect the return value of the `value` method. In this case, the compiler complains since, as was previously established, `Abstract` is an abstract class and can't be instantiated (even in *unevaluated expression* like the `decltype` operator).

On line 11, we try to determine the return value by using a scope resolution operator (`::`). This only works if `Abstract::value` is a static function.

On line 12, the `std::declval` function template allows for the use of non-static member functions of abstract base classes, (or with types with deleted or private constructors, which is common when dealing with *singletons*) without requiring an instance of the type.

The `std::declval` utility function can be implemented like this:

```
std::declvalC++
1
2
    template<typename T>
```

```
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

We haven't looked at type traits yet, but I think you can guess that `std::add_rvalue_reference<T>` makes `T` an rvalue reference.

You may have noticed that the `declval` function template only provides a declaration but not a definition. This is no mistake. This function does not have a definition! It simply converts `T` to an rvalue reference so that it can be used in an `unevaluated context` such as `decltype`.

`std::declval` converts any type (`T`) to a reference type to enable the use of member functions without the need to construct an instance of `T`.
Since `std::declval` is not defined and therefore, it can only be used in an unevaluated context such as `decltype`.

Type Traits

C++11 introduces the `type_traits` library.

Type traits defines a compile-time template-based interface to query or modify the properties of types.

Type traits allow you to discover certain things about a type at compile-time. Some things you may want to know about types are:

1. Is it an integral type?
2. Is it a floating-point type?
3. Is it a class type?
4. Is it a function type?
5. Is it a pointer type?
6. Are two types the same?
7. Is one type derived from the other?
8. Is one type convertible to another?

And the list goes on... There are many things we might want to know about one or more types that can be determined at compile-time.

Don't confuse type traits with Run-Time Type Information (RTTI) which is used to query type information at run-time. Type traits are resolved at compile-time and impose no run-time overhead.

Type traits are the cornerstone for "Substitution Failure Is Not An Error" (SFINAE). But before we look at SFINAE, let's investigate a few type traits that we can use as the basis for SFINAE later.

Keep in mind that a lot of the type traits described in following sections comes from the Standard Template Library (STL). You don't need to define these types yourself in your own code. You can find the original source code for the `type_traits` library here:

- Microsoft STL: <https://github.com/microsoft/STL>
- GCC: <https://github.com/gcc-mirror/gcc>
- Clang/LLVM: <https://github.com/llvm/llvm-project>

My motivation for describing the type traits in this article is to give the reader a better understanding of how they work. Once you know how they work, you will have a better understanding of how to use them correctly.

`integral_constant`

The `integral_constant` structure is the base class for the type traits library. It is a wrapper for a static constant of a specified type. It wraps both the type and a value in a struct so it can be used as a type. You'll see why this is useful later.

`integral_constant`

C++

```
1
2
3  template<typename T, T v>
4
5
6  struct integral_constant
7
8  {
```

```
9
10
11 // Member types
12
13
14 using value_type = T;
15
16
17 using type = integral_constant;
18
19
20
21
22 // Member constants
23
24 static constexpr T value = v;
25
26
27 // Member functions
28
29 constexpr operator value_type() const noexcept
30 {
31     return value;
32 }
33
34 constexpr value_type operator()() const noexcept
35 {
```

```
        return value;

    }

};
```

The `integral_constant` class (struct) template is composed of a type template parameter (`T`) and a *non-type* template parameter `v` (of type `T`).

The value type that was used to instantiate the `integral_constant` can be queried through the `value_type` type alias and the type of the `integral_constant` itself can be queried through the `type` type alias.

The `operator value_type()` member function defined on line 12 is an *implicit conversion operator* which allows an instance of the `integral_constant` template to be converted to `value_type` at compile-time. This allows an instance of `integral_constant` to be used in place where `value_type` is expected (in mathematical expressions for example).

The `value_type operator()` member function defined on line 17 is a *function call operator* that takes no parameters and returns `value`. This allows `integral_constant` to be used as a function object that takes no parameters and returns the stored value.

integral_constant example

C++

```
1
2
3 using three_t = integral_constant<int, 3>;
4
5
6 using five_t = integral_constant<int, 5>;
7
8
9
10 three_t three;
```

```
five_t five;

auto fifteen = three_t() * five_t();

fifteen = three * five;

std::cout << "3 * 5 = " << fifteen << std::endl;
```

On lines 1 and 2, two type aliases of the `integral_constant` template are defined: `three_t` which is a type that represents `3`, and `five_t` which is a type that represents `5`.

On lines 4 and 5, two instances are instantiated using the type aliases that were just defined. `three` is an instance of `type_constant<int, 3>` and `five` is an instance of `type_constant<int, 5>`.

On line 7 and 8, two different methods to get the internal value are demonstrated. The first method on line 7 uses the function call operator to retrieve the internal value. On line 8, the implicit conversion operator is used to convert `three` and `five` to their integer equivalents to be multiplied together. Both expressions result in 15. If you run this program, the following is printed to the console:

```
3 * 5 = 15
```

That might be the most contrived method of computing the value 15 I've ever seen, but in the next section it will become clear why this is useful.

`bool_constant`

With the definition of `integral_constant` from the previous section, other types can be derived from `integral_constant`. A useful type that can be derived from `integral_constant` is `bool_constant`. It is not necessary to define a full class...

this as a [template alias](#) will do:

bool_constant	C++
<pre>1 2 template<bool v> using bool_constant = integral_constant<bool, v>;</pre>	

The `bool_constant` defines a “boolean” `integral_constant`. And as you may have guessed, we can define two new types based on `bool_constant`.

true_type

`true_type` is a type alias of `bool_constant` with a `value` of `true`:

true_type	C++
<pre>1 using true_type = bool_constant<true>;</pre>	

false_type

`false_type` is a type alias of `bool_constant` with a `value` of `false`:

false_type	C++
<pre>1 using false_type = bool_constant<false>;</pre>	

Both `true_type` and `false_type` are aliases of `integral_type` which means that they can be used wherever a class or struct type can be used. For example, we can create a partial specialization of a class that is derived from either `true_type` or `false_type`. But before I get into that, I need to introduce another useful tool for our type traits library: [enable_if](#).

type_identity

At first glance, the `type_identity` class template may not seem very useful. It simply mimics the type `T` that was specified in the template argument.

type_identity

C++

```
1
2
3 template<class T>
4
5
6 struct type_identity
7
8 {
    using type = T;
};

template<class T>

using type_identity_t = typename type_identity<T>::type;
```

The `type_identity` class template becomes useful in a [non-deduced context](#) (for example when used with the `decltype` specifier). We'll use it later to help form SFINAE enabled types (see [add_lvalue_reference](#) and [add_rvalue_reference](#)).

void_t

`void_t` is an alias template that maps any number of type template parameters to `void`. This is useful in the context of SFINAE where you only want to check if a certain set of operations is valid on a type but you don't care about the return type of that check. `void_t` allows you to form these expressions without concern for the return type.

void_t

C++

```
1
2
template<class... T>
```

```
using void_t = void;
```

The `void_t` alias template is commonly used to check if a certain operation is valid on a specific type. For example, if we want to check if a type supports the pre-increment operator, but we don't care about the the actual result type, then we could do something like this:

void_t	C++
<pre>1 2 3 template<class, class = void> 4 5 6 struct has_pre_increment_operator : false_type 7 8 9 {};</pre>	
<pre>template<class T> 10 11 12 struct has_pre_increment_operator<T, void_t<decltype(++std::declval<T>())>> : true_type 13 14 15 {};</pre>	

In this example, the primary template for `has_pre_increment_operator` is derived from `false_type`. The primary template is *only* chosen if there isn't a partial specialization that is a better match. In order to get the compiler to choose the specialized version of the template definition, the operation `++std::declval<T>()` must succeed. But since we only want to see if the operation succeeded, but we don't care what the return value is, we can wrap the result of performing the pre-increment operator in the `void_t` alias template.

Since `void_t` takes a variadic number of template parameters, `void_t` can be used to check if any number of operations are valid on one or more types.

When the compiler fails to instantiate the second template argument in the specialized version of the `has_pre_increment_operator`, this is called *substitution failure* and is the basis of SFINAE ([Substitution Failure Is Not An Error](#)). SFINAE is used to express type trait operations and the `void_t` template alias is used to help formulate those expressions.

`enable_if`

The `enable_if` struct template provides a convenient mechanism to leverage SFINAE in our template classes. “Substitution Failure Is Not An Error” (or [SFINAE](#)) is a C++ technique to conditionally remove specific functions from [overload resolution](#) based on a type’s traits. We’ll get into more details of SFINAE later, for now let’s take a look at how we can define the `enable_if` template:

<pre>enable_if 1 2 3 template<bool B, class T = void> 4 5 6 struct enable_if 7 8 9 {};</pre> <pre>template<class T> struct enable_if<true, T> { using type = T; };</pre>	C++
--	-----

The *primary template* for `enable_if` is a class template that has two template parameters:

1. `bool B`: A non-type template parameter that can be either `true` or `false`.
2. `class T`: A type template parameter that can be any type. If no type is provided, `void` is used by default.

The magic trick comes from the partial specialization that is defined when `B` is `true`. The `type` type alias is **not** defined in the primary template and is *only* defined when `B` is `true`. Any attempt to access the `type` type alias when `B` is `false` will fail (since it's just not defined in this case).

We'll see how we can use this to leverage SFINAE later. Before we can do that, we'll need some type trait templates to work with.

To simplify coding, we'll also define a helper template alias for the `enable_if` template:

enable_if	C++
<pre>1 2 template<bool B, class T = void> using enable_if_t = typename enable_if<B, T>::type;</pre>	

Now, instead of typing `typename enable_if<B, T>::type`, we only need to type `enable_if_t<B, T>`. As you can see, this saves us a lot of typing.

i The `enable_if_t` template alias was introduced in C++14 but there is nothing stopping you from defining these kinds of template aliases in your C++11 code.

remove_const

Sometimes is it convenient to express a type without the `const` qualifier associated with it. The `remove_const` class template allows us to do just that:

```

1
2
3  template<class T>
4
5
6  struct remove_const
7
8  {
9
10
11     using type = T;

};

template<class T>

struct remove_const<const T>
{

    using type = T;

};

```

The primary template for `remove_const` is only used when `T` is a non-const type. If `T` is const, then the partial specialization kicks-in to remove the const modifier from `T`.

And again, we'll also define a helper template alias:

```
remove_const
```

```

1
2

```

C++

```
template<class T>

using remove_const_t = typename remove_const<T>::type;
```

! The `remove_const` class template only removes the *topmost* `const` qualifier from `T`.

remove_volatile

Similar to `remove_const`, the `remove_volatile` class template can be used to remove the `volatile` qualifier from a type:

remove_volatile

C++

```
1
2
3  template<class T>
4
5
6  struct remove_volatile
7
8  {
9
10
11      using type = T;

};

template<class T>

struct remove_volatile<volatile T>

{

    using type = T;
```

```
};
```

And the helper template alias:

```
remove_volatile
```

```
C++
```

```
1
2
   template<class T>

   using remove_volatile_t = typename remove_volatile<T>::type;
```

```
remove_cv
```

The `remove_cv` class template is used to remove the topmost `const`, `volatile`, or both qualifiers if present:

```
remove_cv
```

```
C++
```

```
1
2
3  template<class T>
4
5
6  struct remove_cv
7
8  {
9
   using type = remove_const_t<remove_volatile_t<T>>;

};

// Helper template alias.
```



```
template<class T>

using remove_cv_t = typename remove_cv<T>::type;
```

In this example, the `remove_const` and `remove_volatile` are combined on line 4 to remove both `const` and `volatile` qualifiers from `T` (if present).

remove_reference

The `remove_reference` class template is used to remove any reference (or rvalue references) from a type.

remove_reference	C++
<pre>1 2 3 template<class T> 4 5 6 struct remove_reference 7 8 { 9 10 11 using type = T; 12 13 14 }; 15 16 17 18 19 template<class T> 20 21 struct remove_reference<T&> 22 23 { 24 25 using type = T;</pre>	

```

};

template<class T>

struct remove_reference<T&&>
{

    using type = T;

};

// Helper template alias.

template<class T>

using remove_reference_t = typename remove_reference<T>::type;

```

If `T` is a reference (or rvalue reference) type, then the `remove_reference` class template will strip off the reference from the type.

`remove_cvref`

Now we can combine `remove_cv` and `remove_reference` to remove `const`, `volatile`, and references from a type:

`remove_cvref`

1
2

```

3  template<class T>
4
5
6  struct remove_cvref
7
8  {

    using type = remove_cv_t<remove_reference_t<T>>;

};

template<class T>

using remove_cvref_t = typename remove_cvref<T>::type;

```

remove_extent

It might be useful to extract the type of an array. The `remove_extent` class template can be used to extract the type of an array. For example, if `T` is an (bounded or unbounded) array of type `X`, then `remove_extent_t<T>` evaluates to `X`.

remove_extent

C++

```

1
2
3  template<class T>
4
5
6  struct remove_extent
7
8  {
9
10
11     using type = T;
12
13

```

```

14 };
15
16
17
    template<class T>

    struct remove_extent<T[]>
    {

        using type = T;

    };

    template<class T, std::size_t N>

    struct remove_extent<T[N]>
    {

        using type = T;

    };

```

The primary template (lines 1-5) is used if `T` is not an array type. If `T` is an unbounded array (lines 7-11) or a bounded array (lines 13-17), then `remove_extent<T>::type` is defined to be the type of the array with the extents removed.

! If `T` is a multidimensional array, `remove_extent<T>` only removes the first dimension.

A short-hand for the `remove_extent` class template:

remove_extent	C++
<pre>1 2 template<typename T> using remove_extent_t = typename remove_extent<T>::type;</pre>	

`remove_all_extents`

Since the `remove_extent` class template only removes the first dimension of multidimensional arrays, the `remove_all_extents` class template can be used to remove all of the extents of multidimensional arrays.

remove_all_extents	C++
<pre>1 2 3 template<class T> 4 5 6 struct remove_all_extents 7 { 8 { 9 10 11 using type = T; 12 13 14 }; 15 16 17 template<class T> struct remove_all_extents<T[]></pre>	

```

{

    using type = typename remove_all_extents<T>::type;

};

template<class T, std::size_t N>

struct remove_all_extents<T[N]>
{

    using type = typename remove_all_extents<T>::type;

};

```

The primary template (lines 1-5) for the `remove_all_extents` class template looks similar to the `remove_extent` class template. The primary template is only used when `T` is not an array type or all of the extents have already been removed from the type.

If `T` is an unbounded (line 7-11) or bounded (line 13-17) array type, then the appropriate partial specialization is used. In this case, the first dimension is removed from `T` and the `remove_all_extents` class template is invoked recursively to remove the next extent. This process continues until the primary template is reached.

And the shorthand form:

```

remove_all_extents
1
2
template<class T>

```

C++

```
using remove_all_extents_t = typename remove_all_extents<T>::type;
```

remove_pointer

The `remove_pointer` class template can be used to remove the pointer from a type. Any `const` or `volatile` qualifiers added to the *pointer* are also removed.

Any `const` or `volatile` qualifiers added to *the pointed-to type* are not removed. For example, `const int*` becomes `const int`, but `int* const` becomes `int` and `const int* const` becomes `const int`. If you *also* want to remove the `const` or `volatile` qualifiers from the pointed-to type, then you must use the `remove_cv` class template as well.

remove_pointer

C++

```
1
2
3  template<class T>
4
5
6  struct remove_pointer
7  {
8
9
10
11      using type = T;
12
13
14  };
15
16
17
18
19  template<class T>
20
21
22  struct remove_pointer<T*>
23
24  {
25
```

```
26
27     using type = T;
28
29 };

template<class T>

struct remove_pointer<T* const>
{

    using type = T;

};

template<class T>

struct remove_pointer<T* volatile>
{

    using type = T;

};

template<class T>
```



```

struct remove_pointer<T* const volatile>
{

    using type = T;

};

```

The primary template (lines 1-5) is used if `T` is not a pointer type. Partial specializations (lines 7-29) are used if `T` is a (`const` or `volatile` qualified) pointer type.

And a short-hand version:

remove_pointer	C++
<pre> 1 2 template<class T> using remove_pointer_t = typename remove_pointer<T>::type; </pre>	

add_const

In some cases, you may want to add the `const` qualifier to a type. The `add_const` class template can be used to add the `const` qualifier to any type `T` (except for function and reference types, since these can't be `const` qualified).

add_const	C++
<pre> 1 2 #pragma warning(push) 3 4 #pragma warning(disable: 4180) // Disable C4180: qualifier applied to function type has no meaning; ignored 5 6 7 template<class T> </pre>	

```

8
9
10 struct add_const
11 {
    using type = const T;
};

#pragma warning(pop)

template<class T>

using add_const_t = typename add_const<T>::type;

```

Due to `const` collapsing rules, if `T` is already `const`, then adding another `const` to `T` does not change the constness of `T`.

If we try to apply the `add_const` class template to a function type, then the compiler will generate a [C4180](#) warning that applying a `const` to a function type does not make sense and has no meaning. `#pragma warning(disable: 4180)` causes this warning to be suppressed in Visual Studio.

add_volatile

Similar to `add_const` class template, the `add_volatile` class template can be used to add the `volatile` qualifier to a type (except for function and reference types, since these can't be `volatile` qualified).

add_volatile	C++
<pre> 1 2 #pragma warning(push) 3 </pre>	

```

4  #pragma warning(disable: 4180) // Disable C4180: qualifier applied to function type has no meaning; ignored
5
6
7  template<class T>
8
9
10 struct add_volatile
11 {
    using type = volatile T;
};

#pragma warning(pop)

template<class T>

using add_volatile_t = typename add_volatile<T>::type;

```

Similar to the `add_const` class template, the `add_volatile` class template, adds the `volatile` qualifier to a type (`T`). Adding the `volatile` qualifier if `T` is already `volatile` does not change `T`.

Similar to `add_const`, if `T` is a function type, then the compiler will generate a C4180 warning. `#pragma warning(disable: 4180)` suppresses this warning in Visual Studio.

add_cv

The `add_cv` template combines `add_const` and `add_volatile`.

```

add_cv
1
2  #pragma warning(push)

```

```

3
4 #pragma warning(disable: 4180) // Disable C4180: qualifier applied to function type has no meaning; ignored
5
6
7 template<class T>
8
9
10 struct add_cv
11 {
12
13     using type = const volatile T;
14
15 };
16
17 #pragma warning(pop)
18
19
20 template<class T>
21
22 using add_cv_t = typename add_cv<T>::type;

```

Similar to `add_const` and `add_volatile`, we also need to suppress the `C4180` compiler warning if we try to use `add_cv` with a function type.

`add_lvalue_reference`

The `add_lvalue_reference` class template is used to create an lvalue reference from a type. We have to be careful since trying to add a reference to non-referenceable type (for example, `void` is a non-referenceable type) will generate a compilation error. To account for this, we'll use a helper template that is specialized for referenceable types. Trying to use `add_lvalue_reference` with a non-referenceable type should produce the original type.

`add_lvalue_reference`

1

```
2
3 template<class T, class = void>
4
5
6 struct add_lvalue_reference_helper
7 {
8
9
10
11     using type = T;
12
13
14 };
15
16
17
18 template<class T>
19
20 struct add_lvalue_reference_helper<T, void_t<T&>>
21 {
22
23     using type = T&;
24
25 };
26
27
28 template<class T>
29
30 struct add_lvalue_reference : add_lvalue_reference_helper<T>
31
32 {};
```

```
template<class T>

using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
```

The `add_lvalue_reference_helper` class template uses SFINAE to safely convert `T` to `T&`. If `T` is a referenceable type, then the partial specialization (lines 7-11) succeeds and `add_lvalue_reference_helper<T>::type` evaluates to `T&`. If the partial specialization fails, then `T` is a non-referenceable type, and the primary template is chosen. In this case, `add_lvalue_reference_helper<T>::type` evaluates to `T`.

On lines 13-15, the `add_lvalue_reference` class template is defined which is derived from `add_lvalue_reference_helper` allowing the `add_lvalue_reference` to be defined with a single template parameter. Theoretically, the `add_lvalue_reference` class template could be implemented without `add_lvalue_reference_helper`, but then we'd need to define `add_lvalue_reference` using two template parameters. Since the second template parameter is *only* used for SFINAE, using `add_lvalue_reference_helper` allows us to define the `add_lvalue_reference` class template using a single template parameter.

add_rvalue_reference

Similar to `add_lvalue_reference`, the `add_rvalue_reference` class template is used to create an rvalue reference from a type. Once again, we'll use a helper class template to account for non-referenceable types (such as `void`).

```
add_rvalue_referenceC++
1
2
3 template<class T, class = void>
4
5
6 struct add_rvalue_reference_helper
7
8 {
9
10
11     using type = T;
12
13
```

```

14 };
15
16
17
18
    template<class T>

    struct add_rvalue_reference_helper<T, void_t<T&&>>
    {

        using type = T&&;

    };

    template<class T>

    struct add_rvalue_reference : add_rvalue_reference_helper<T>

    {};

    template<class T>

    using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

```

The derivation for the `add_rvalue_reference` class template is similar to that of `add_lvalue_reference`, so I won't repeat it here.

Due to reference collapsing rules, `add_rvalue_reference<T&>` will result in `T&` not `T&&`. There are two reference collapsing rules:

1. An rvalue reference to an rvalue reference becomes an rvalue reference.
- ! 2. All other references to references (all combinations involving an lvalue reference) becomes an lvalue reference.

In the case where `T` is an lvalue reference, then `add_rvalue_reference<T&>` will collapse into a lvalue reference^[8].

add_pointer

Similar to `add_lvalue_reference` and `add_rvalue_reference` class templates, the `add_pointer` class template adds a pointer to a give type `T`. If `T` is a reference type, then `add_pointer<T>::type` becomes `remove_reference_t<T>*`, that is, a pointer is added to the type `T`, after removing the reference.

- ! Although it is possible to have a reference to a pointer (`T*&`), it is not possible to add a pointer to a reference (`T&*`). Attempting to create a pointer to a reference type will result in a compiler error.

add_pointer

C++

```
1
2
3  template<class T>
4
5
6  auto add_pointer_helper(int) -> type_identity<remove_reference_t<T>*>;
7
8
9
10
11  template<class T>
12
    auto add_pointer_helper(...) -> type_identity<T>;
```



```

template<class T>

struct add_pointer : decltype(add_pointer_helper<T>(0))

{};

template<class T>

using add_pointer_t = typename add_pointer<T>::type;

```

Although it is possible to implement the `add_pointer` class template using a similar technique that is used to implement the `add_lvalue_reference` and `add_rvalue_reference` class templates, I want to demonstrate a different technique that utilizes SFINAE to achieve a similar result. Instead of using a struct with partial specialization, we declare two functions.

The first function declared on lines 1-2 takes an `int` parameter and returns the template argument `T` with the reference removed and a pointer added to the type wrapped in the `type_identity` template (which provides the `type` member).

If `T` a `const`, `volatile`, or reference-qualified function type, then trying to add a pointer to `T` (`T*`) would normally generate a compiler error. Instead of generating an error, the compiler will consider the second overload of `add_pointer_helper` instead.

The second function declared on lines 4-5 is a fallback function that takes any other parameter type using the ellipsis (`...`) and since the compiler considers this the worst possible match, it *only* chooses this overload if the compiler fails to instantiate the first version of the function. In this case, the template argument `T` is wrapped in the `identity_type` unmodified.

This technique that forces the compiler to choose worse match during function overload resolution is discussed in more detail later in the article (see SFINAE Out Function Overloads)

Both functions can't take the same parameters (in this case `int`) because then the function signatures would become ambiguous and would fail to compile. Using the ellipsis (`...`) tells the compiler to only match this version of the function if it fails to instantiate the first version of the function.

You'll notice that we provide a declaration of the `add_pointer_helper` functions but do not provide a definition. This is perfectly fine, as long as these functions are only used in a `non-deduced context` such as an expression that is only evaluated using the `decltype` specifier.

On lines 7-9, the `add_pointer` class template is defined and is derived from the return value of the `add_pointer_helper` function.

And, as usual, the short-hand alias of the `add_pointer` class template is defined on lines 11-12.

conditional

In some cases, it is useful to choose a specific type based on some condition. The `conditional` class template can be used to return one type or another type based on some condition (usually based on another type trait). The `conditional` class template is equivalent to an `if` condition in regular C++.

conditional

C++

```
1
2
3  template<bool B, class T, class F>
4
5
6  struct conditional
7
8  {
9
10
11      using type = T;
12
13
14  };
15
```

```

template<class T, class F>

struct conditional<false, T, F>
{

    using type = F;

};

// Helper template alias.

template<bool B, class T, class F>

using conditional_t = typename conditional<B, T, F>::type;

```

On lines 1-5 the primary template is defined. When `B` is `true`, then the `type` is an alias of `T` (let's call this the *true type*). However, when the partial specialization where `B` is `false` is matched, then `type` is `F` (the *false type*).

Pretty simple right? Let's see how we can use the `conditional` class template to implement more complex logical template types.

conjunction

A **logical conjunction** is a *truth-functional* operator that is *true* if and only if all of its operands are *true*. This is equivalent to a logical **AND** (`&&`) operation.

The `conjunction` class template works by passing any number of type traits that are derived from either `true_type` or `false_type` (or has a static member variable `value` that is convertible to `bool`) as template arguments to `conjunction`. ...

`conjunction` class template is derived from the first template argument whose `value` member variable is (or is convertible to) `false`. If all type traits passed to `conjunction` are `true`, then conjunction is derived from the last type trait in the template argument list.

We haven't looked at many type trait templates that are derived from `true_type` or `false_type` yet, but these are introduced later in the article. For now, we just have to keep in mind that a class template that derives from either `true_type` or `false_type` has a `static const` member variable `value` that is `true` if it is derived from `true_type` or `false` if it is derived from `false_type`.

We'll use a `recursive variadic template` to implement the `conjunction` class template. First, let's take a look at the primary template.

```
conjunctionC++
1
2 // Primary template
3
4
   template<class...>
   struct conjunction : true_type
   {};
```

The *primary template* is a class template that doesn't specialize any of the template parameters. The compiler will choose this version of the `conjunction` class template only if there isn't a specialization of the class template that is a better match. As we'll see in a second, the primary template will only be chosen when `conjunction` is used without any template arguments (which is probably a mistake by the programmer). Although the primary template should never be chosen by the compiler, we need it before we can specialize it.

Now let's take a look at the base case of the `recursive variadic template`, that is, when `conjunction` has only a single template argument.

```
conjunctionC++
```

```

1
2 // Specialized for a single template argument.
3
4
template<class T>

struct conjunction<T> : T

{};

```

In the base case, when `conjunction` has only a single template argument (`T`), then `conjunction` is derived from `T`. Since all of the template arguments passed to `conjunction` must have a static member variable called `value` that is convertible to `bool` (like `true_type` and `false_type`), then `conjunction::value` is `true` when `T::value` is `true` and `false` when `T::value` is `false`.

Now let's look at the recursive case:

conjunction	C++
<pre> 1 2 // Specialized for 2 or more template arguments. 3 4 template<class T, class... Tn> struct conjunction<T, Tn...> : conditional_t<bool(T::value), conjunction<Tn...>, T> {}; </pre>	

The recursive case is used when `conjunction` has two or more template arguments. In this case, the `conditional` class template is used to conditionally select either `conjunction<Tn...>` (recursively calling itself) if `T::value` is `true` or `T` when `T::value` is `false`.

Consequently, if `T::value` is `false` then the expansion of `Tn...` stops and no further types need to be instantiated to

determine the type of `conjunction`. This is in contrast to using fold expressions `(... && Tn::value)` where every `T` in `Tn` is instantiated before the expression is evaluated.

In order to reduce some typing later, we'll also define an inline `variable template` for `conjunction::value`:

```
conjunctionC++
1
2 // Requires C++17
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658

```

Then prepend inline variable templates with the `INLINE_VAR` macro instead of `inline constexpr`.

disjunction

Similar to `conjunction`, the `disjunction` class template is equivalent to a logical OR operator.

The `disjunction` class template works by passing any number of type traits that are derived from either `true_type` or `false_type` (or has a static member variable `value` that is convertible to `bool`) as template arguments. The `disjunction` class template is derived from the first template argument whose `value` member variable is (or is convertible) to `true`. If all template arguments passed to `disjunction` are `false`, then `disjunction` is derived from the last type trait in the template argument list.

Similar to the implementation of the `conjunction` class template, we'll use a `recursive variadic template` to implement the `disjunction` class template. First, let's take a look at the primary template.

```
disjunctionC++
1
2 // Primary template
3
4
   template<class...>

   struct disjunction : false_type

{};
```

The *primary template* doesn't specialize any of the template parameters. The compiler will use the primary template only if there isn't a specialization of the `disjunction` class template that is a better match. Since there is a specialization for a single template argument and a specialization for two or more template arguments (see below), the primary template is only chosen when `disjunction` is used without any template arguments (which is probably a mistake). Although the primary template should never be chosen by the compiler, we need to define it before we can specialize it.

The base case for the `recursive variadic template` has only a single template parameter:

disjunction

C++

```
1
2 // Base case
3
4
   template<class T>

   struct disjunction<T> : T

   {};
```

In the base case, when the `disjunction` class template has only a single template argument (`T`), then `disjunction` is derived from `T`. Since all of the template arguments passed to `disjunction` must have a static member variable called `value` that is convertible to `bool`, then `disjunction::value` is `true` when `T::value` is `true` and `false` if `T::value` is `false`.

Now, let's look at the recursive case.

disjunction

C++

```
1
2
3 template<class T, class... Tn>

   struct disjunction<T, Tn...> : conditional_t<bool(T::value), T, disjunction<Tn...>>

   {};
```

The recursive case is instantiated when `disjunction` is used with two or more template arguments. In this case, the `conditional` class template is used to conditionally select either `T` if `T::value` is `true` or `disjunction<Tn...>` (recursively calling itself) if `T::value` is `false`.

Consequently, if `T::value` is `true`, then the expansion of `Tn...` stops and no further types need to be instantiated to determine the type of `disjunction`. This is in contrast to using the fold expression `(... || Tn::value)` which must

instantiate every `T` in `Tn` before the expression is evaluated.

And similar to `conjunction`, we'll define an inline `variable template` to create a short-hand for `disjunction::value`

```
disjunctionC++
1
2 // Requires C++17
3
   template<class... T>

   inline constexpr bool disjunction_v = disjunction<T...>::value;
```

Now, instead of typing `disjunction<...>::value`, we only need to type `disjunction_v<...>`.

negation

The `negation` class template forms a logical negation of the type trait `T`.

```
negationC++
1
2
3 template<class T>

   struct negation : bool_constant<!bool(T::value)>

   {};
```

The `negation` class template is derived from the `bool_constant` class template. If `T::value` is `true`, then `negation::value` is `false` (which is equivalent to being derived from `false_type`) and if `T::value` is `false`, then `negation::value` is `true` (which is equivalent to being derived from `true_type`).

And a short-hand version of `negation::value`:

negation

C++

```
1
2 // Requires C++17
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

With the definition of `conditional`, `conjunction`, `disjunction`, and `negation`, we have the logical operators that are needed to make any logical combination of type traits that we need. In the following sections, we'll use these logical class templates as the basis for other type traits.

is_same

The `is_same` class template is the first class in our type traits library that can actually be considered a type trait. Most type traits result in a boolean `value` that is either `true` or `false`. This is accomplished by inheriting from either `true_type` if the type trait is `true` or `false_type` if the type trait is `false`.

In most cases, the primary template for the type trait is derived from `false_type` and one or more `partial specializations` exist that are derived from `true_type`.

First, let's look at the primary template for the `is_same` type trait.

is_same

C++

```
1
2 // Primary template.
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

The primary template is chosen by the compiler when `T` and `U` are different types. Next, we'll create a partial specialization of `is_same` when `T` and `U` are the same types.

```
is_sameC++
1
2 // Specialization for matching types.
3
4
5 template<class T>
6
7     struct is_same<T, T> : true_type
8
9
10 {};
```

The partial specialization is only chosen when `T` and `U` are *exactly* the same types. That means that their `const`, `volatile`, reference or pointer attributes must also be the same. If you want to ignore any `const`, `volatile`, or references that might adorn the type, then wrap the type in either the `remove_cv` or `remove_cvref` class template.

A short-hand for `is_same::value` can be defined using a `variable template` (requires C++14):

```
is_sameC++
1
2 // Requires C++17
3
4
5 template<class T, class U>
6
7
8     inline constexpr bool is_same_v = is_same<T, U>::value;
```

is_void

With the `is_same` and `remove_cv` type traits defined, we can use these to define other type traits which can be used

identify all of the primary types. The `is_void` type trait evaluates to `true_type` if the template argument is `void` (ignoring any `const` and `volatile` qualifiers) and `false_type` otherwise.

```
is_voidC++
1
2
3 template<class T>

    struct is_void : is_same<void, remove_cv_t<T>>

    {};
```

As was shown in the previous section, the `is_same` type trait is derived from `true_type` when the first and second template arguments are exactly the same type, and `false_type` otherwise. We can use this to define a type trait that checks for a primary type (like `int`s and `float`s which we'll look at in the following sections).

We'll also define an inline variable template called `is_void_v` that can be used as a short-hand for `is_void::value`:

```
is_voidC++
1
2 // Requires C++17
3
    template<class T>

    inline constexpr bool is_void_v = is_void<T>::value;
```

is_null_pointer

Similar to the `is_void` type trait, the `is_null_pointer` type trait is derived from `true_type` if the template argument is `std::nullptr_t`.

```
is_null_pointer...
```

```

1
2 #include <cstddef> // for std::nullptr_t
3
4
5
6
7 template<class T>
8
9     struct is_null_pointer : is_same<std::nullptr_t, remove_cv_t<T>>
10
11     {};
12
13 // Requires C++17
14
15 template<class T>
16
17     inline constexpr bool is_null_pointer_v = is_null_pointer<T>::value;

```

The implementation of `is_null_pointer` is similar to `is_void`.

is_any_of

The `is_any_of` type trait can be used to check if a certain type template argument matches *any one of* the other type template arguments. We'll use the `disjunction` and the `is_same` type traits defined earlier to implement the `is_any_of` type trait.

The `is_any_of` type trait does not currently exist in the C++ standard, but we'll use this type trait to simplify the definition of other type traits later in this article.

is_any_of

C++

```

1
2

```

```

3 template<class T, class... Tn>
4
5
6 struct is_any_of : disjunction<is_same<T, Tn>...>
7
8     {};
9
10
11 // Requires C++17
12
13 template<class T, class... Tn>
14
15 inline constexpr bool is_any_of_v = is_any_of<T, Tn...>::value;

```

The implementation of the `is_any_of` type trait is super simple since we can rely on the `disjunction` and the `is_same` type traits that were previously defined. Variadic template arguments are used to check if the type `T` matches any one of types in the pack represented by `Tn`.

`is_integral`

With the `is_any_of` type trait defined previously, it is now super simple to implement other type traits. The `is_integral` type trait is `true_type` if the template argument is one of the following types:

- `bool`
- `char`
- `char8_t` (since C++20)
- `char16_t`
- `char32_t`
- `wchar_t`
- `short`
- `int`
- `long`

- `long long`

Or any other equivalent type including signed or unsigned, with const, and volatile qualified variants. Otherwise, it is `false_type`.

is_integral

C++

```
1
2
3  template<class T>
4
5
6  struct is_integral : is_any_of<remove_cv_t<T>,
7
8
9      bool,
10
11
12      char,
13
14
15      signed char,
16
17
18      unsigned char,
19
20
21      wchar_t,
22
23
24      char16_t,
25
26
27      char32_t,
28
29
30      short,
31
32
33      unsigned short,
```

```

    int,

    unsigned int,

    long,

    unsigned long,

    long long,

    unsigned long long
#endif
    , char8_t // Since C++20
#endif
>

{};

// Requires C++17

template<class T>

inline constexpr bool is_integral_v = is_integral<T>::value;

```

Using the `is_any_of` type trait defined previously, the `is_integral` type trait becomes much simpler. If the template

argument `T` is any one of the types listed (after removing `const` and `volatile` qualifiers), then `is_integral` is `true_type`, otherwise it is `false_type`.

`is_floating_point`

Similar to the `is_integral` type trait, the `is_floating_point` type trait is `true_type` if the template argument is one of the following types:

- `float`
- `double`
- `long double`

Including `const` and `volatile` qualified variants. Otherwise, it is `false_type`.

```
is_floating_pointC++
1
2
3  template<class T>
4
5
6  struct is_floating_point : is_any_of<remove_cv_t<T>,
7
8      float,
9
10     double,
11
12     long double
13
14     >
15
16  {};
```

```
// Requires C++17

template<class T>

inline constexpr bool is_floating_point_v = is_floating_point<T>::value;
```

is_arithmetic

The `is_arithmetic` type trait is `true_type` if its template argument is either an integral type or a floating-point type, and `false_type` otherwise. As you may have guessed, we can use the `is_integral` and `is_floating_point` type traits defined earlier in combination with `disjunction` to implement this type trait.

is_arithmetic	C++
<pre>1 2 3 template<class T> 4 5 6 struct is_arithmetic : disjunction<is_integral<T>, is_floating_point<T>> 7 8 9 {};</pre>	
<pre>// Requires C++17 template<class T> inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;</pre>	

An alternative and equivalent implementation of the `is_arithmetic` type trait uses the `bool_constant` class template directly:

```

1
2
3 template<class T>

    struct is_arithmetic : bool_constant<is_integral_v<T> || is_floating_point_v<T>>

    {};

```

Although it doesn't change the meaning of the `is_arithmetic` type trait, it demonstrates that you can use the logical OR operator (`||`) to evaluate template arguments. Note however that the `disjunction` class template expects its template arguments to be types, while the `bool_constant` class template expects a non-type template argument that is convertible to `bool`. In this case, we can use the `_v` short-hand variants of the type traits to get the `value` member variable of the type trait.

is_const

The `is_const` type trait checks to see if a type is const-qualified. The `is_const` type trait is derived from `true_type` if the type is const-qualified, or `false_type` otherwise.

```

is_const
1
2
3 template<class T>
4
5
6 struct is_const : false_type
7
8
9 {};
10
11

    template<class T>

```

```

struct is_const<const T> : true_type

{};

// Required C++17

template<class T>

inline constexpr bool is_const_v = is_const<T>::value;

```

Similar to the `remove_const` class template that was previously shown, the `is_const` type trait uses [partial specialization](#) to detect const-qualified types.

`is_reference`

The `is_reference` type trait can be used to check if a type is either an lvalue reference or rvalue reference type.

is_reference	C++
<pre> 1 2 3 template<class T> 4 5 6 struct is_reference : false_type 7 8 9 {}; 10 11 template<class T> struct is_reference<T&> : true_type </pre>	

```
{};
```

```
template<class T>
```

```
struct is_reference<T&&> : true_type
```

```
{};
```

The `is_reference` type trait is derived from `true_type` if the type `T` is either an lvalue (line 6) or an rvalue (line 10) reference. Otherwise, `is_reference` is derived from `false_type`.

And the short-hand version:

```
is_const
```

C++

```
1
2
template<class T>

inline constexpr bool is_reference_v = is_reference<T>::value;
```

`is_bounded_array`

The `is_bounded_array` type trait checks if `T` is an array type with a known size.

```
is_bounded_array
```

C++

```
1
2
3 template<class T>
4
5
```

```

6 struct is_bounded_array : false_type
7
8 {};
```

```

9
10 template<class T, std::size_t N>
11
12 struct is_bounded_array<T[N]> : true_type
13
14 {};
```

If `T` is a bounded array (an array with a known size in the form of `T[N]`), then `is_bounded_array` is derived from `true_type`, otherwise it is derived from `false_type`.

And the short-hand variant:

is_bounded_array	C++
<pre> 1 2 // Requires C++17 3 4 template<class T> 5 6 inline constexpr bool is_bounded_array_v = is_bounded_array<T>::value;</pre>	

is_unbounded_array

The type trait for an unbounded array is very similar to that of the bounded array. The primary difference is that the size of the array is unspecified.

is_unbounded_array
<pre> 1</pre>

```

2
3  template<class T>
4
5
6  struct is_unbounded_array : false_type
7
8
9  {};
10
11

    template<class T>

    struct is_unbounded_array<T[]> : true_type

    {};

    // Requires C++17

    template<class T>

    inline constexpr bool is_unbounded_array_v = is_unbounded_array<T>::value;

```

is_array

The `is_array` type trait can be used to check if a type is either a bounded or unbounded array.

is_array

C++

```

1
2
3  template<class T>
4
5

```

```

6 struct is_array : bool_constant<is_bounded_array_v<T> || is_unbounded_array_v<T>>
7
    {};

    // Requires C++17

    template<class T>

    inline constexpr bool is_array_v = is_array<T>::value;

```

is_function

The `is_function` type trait can be used to check if a type is a function. The `is_function` type trait only considers free functions and static member functions of a class to be function types. `std::function`, `lambdas`, callable function objects (classes or structs with overloaded function call operator `operator()`) and pointers to functions are not considered function types.

is_function	C++
<pre> 1 2 #pragma warning(push) 3 4 #pragma warning(disable: 4180) // Disable C4180: qualifier applied to function type has no meaning; ignored 5 6 7 template<class T> 8 9 struct is_function : bool_constant<!is_const_v<const T> && !is_reference_v<T>> {}; #pragma warning(pop) </pre>	


```
template<class T>

inline constexpr bool is_function_v = is_function<T>::value;
```

The `is_function` type trait works on the basis that only function types and reference types can't be const-qualified. Adding the `const` qualifier to a function type does not change its constness. If `T` is a function type (or a reference type), then `is_const_v<const T>` will be `false`. If `is_reference_v<T>` is also `false`, then `T` must be a function type.

In fact, trying to add a `const` qualifier to a function type will generate a warning (C4180) in Visual Studio. The warning about applying a const qualifier to a function type can be disabled using the `#pragma warning(disable: 4180)` pragma as shown in the code snippet.

decay

The `decay` class template is used to perform the same conversion operations that are applied to template arguments when passed by value. The `decay` class template will do one of three things depending on the argument type:

- **Array:** If `T` is an array of type `U` (either an unbounded array of the form `U[]` or a bounded array of the form `U[N]`) then `decay<T>::type` will be `U*`. That is, array types decay to pointers to the array element type.
- **Function:** If `T` is a function type or a reference to a function, then `decay<T>::type` will be `add_pointer_t<T>`. That is, function types become pointers to functions.
- **Neither array nor function:** `decay<T>::type` removes any reference, `const`, and `volatile` qualifiers from the type using `remove_cv_t<remove_reference_t<T>>`

We'll use a piecewise technique to implement the `decay` class template that is split into 3 parts:

1. The primary template which is used when `T` is neither an array type nor a function type.
2. A partial specialization when `T` is an array type.
3. A partial specialization when `T` is a function type.

Similar to how we implemented `add_lvalue_reference`, and `add_rvalue_reference`, we'll use a helper class template so that

the `decay` class template only requires a single template parameter.

First, we'll look at the primary template for the `decay_helper` class template.

```
decay                                                                    C++
1
2
3  template<class T,
4
5
6      bool IsArray = is_array_v<T>,
7
8      bool IsFunc = is_function_v<T>
9
10 >
11
12  struct decay_helper
13  {
14
15      using type = remove_cv_t<T>;
16
17  };

```

The primary template for the `decay_helper` class template takes three template arguments:

1. `T`: The type to decay,
2. `IsArray`: A boolean non-type template parameter that is `true` if `T` is an array type or `false` otherwise.
3. `IsFunc`: A boolean non-type template parameter that is `true` if `T` is a function type or `false` otherwise.

The `IsArray` template parameter defaults to `is_array_v<T>` and the `IsFunc` template parameter defaults to `is_function_v<T>`.

Since we will provide a partial specialization when `T` is an array type and another specialization for when `T` is a

function type, the primary template is only used when `T` is neither an array nor a function type (that is, both `isArray` and `IsFunc` are `false`).

When `T` is neither an array nor a function type, we'll use the `remove_cv` class template to remove any `const`, and `volatile` qualifiers from `T`.

Next, we'll provide a partial specialization of `decay_helper` when `T` is an array type.

decay	C++
<pre>1 2 3 template<class T> 4 5 struct decay_helper<T, true, false> 6 7 { 8 9 using type = remove_extent_t<T>*; 10 11 }; </pre>	

This partial specialization will be used when `is_array_v<T>` evaluates to `true` and `is_function_v<T>` evaluates to `false`. In this case, the resulting type is `remove_extent_t<T>*`, which removes the array extent from the type and adds a pointer to the resulting element type.

Next, we'll provide a partial specialization of `decay_helper` when `T` is a function type.

decay	C++
<pre>1 2 3 template<class T> 4 5 struct decay_helper<T, false, true> </pre>	

```
{  
  
    using type = add_pointer_t<T>;  
  
};
```

This partial specialization is used when `is_array_v<T>` evaluates to `false` and `is_function_v<T>` evaluates to `true`. In this case, we just add a pointer to the function type using the `add_pointer` class template.

With all the possible combinations handled, we can now define the `decay` class template.

decay	C++
<pre>1 2 3 template<class T> 4 5 6 struct decay 7 8 { using type = typename decay_helper<remove_reference_t<T>>::type; }; template<class T> using decay_t = typename decay<T>::type;</pre>	

The `decay` class template uses the `decay_helper` class template to determine the type of the `type` member after removing any references from `T`.

And of course, on lines 28-29, the short-hand alias template for `typename decay<T>::type` is defined.

SFINAE

SFINAE (*sfee-nay*) is an acronym for “Substitution Failure Is Not An Error” and it refers to a technique used by the compiler to turn potential errors into “deduction failures” during template argument deduction. It is a technique used to eliminate certain functions from being chosen during overload resolution or to choose a particular class template specialization based on characteristics of the template arguments. If the compiler finds an invalid expression during template argument deduction, instead of generating a compiler error, it removes that function or class specialization from the set of possible candidates.

We’ve already seen a few examples of using SFINAE to generate a few of the type traits in the previous sections. The `add_pointer` type trait uses a set of function overloads to *SFINAE-out* the case where adding a pointer to `T` would result in an error. For example, if `T` was a `const`, `volatile` or reference qualified function type, then attempting to add a pointer to `T` would generate a compile-time error. Instead of generating an error, the compiler eliminates the first function from the set of overloads and considers the next function.

Another SFINAE technique is used to define the `add_lvalue_reference` and `add_rvalue_reference` type traits. Instead of function overloads, partial specialization of a class template is used to SFINAE-out cases where adding a reference to `T` would result in a compiler error (for example, if `T` is `void`).

SFINAE-Out Function Overloads

To demonstrate SFINAE using function template overloads, we’ll create a SFINAE-based type trait to determine if a type `T` is constructible using a particular set of arguments (`Args...`).

The approach to implementing SFINAE-based traits with function overloads is to declare two overloaded function templates named `test` (you can use any name for this function, but `test` is traditionally used for SFINAE-based traits).

```
is_constructible
```

```
1
```

```

2
3 template<...>
4
5
6 true_type test(int);

// fallback

template<...>

false_type test(...);

```

The first overload version of the `test` function template takes an `int` (any parameter type can be used, but `int` seems like a good choice) and returns `true_type`. This version of the function template is used if the template parameters for the provided template arguments are well-formed.

The second overload of the `test` function is called the *fallback* and takes any argument types using the ellipsis operator (`...`). Since the compiler considers functions taking the ellipsis operator the worst possible match during function overload resolution, the second version of the `test` function will only be chosen if the template parameters in the first overload are not well formed. In this case, the fallback for the test function returns `false_type`.

The form of the template parameters for the overload that returns `true_type` should only be valid if (and only if) the condition we want to check is `true`. In this case, we want to check if it is possible to construct a type `T` from a given set of arguments (`Args...`). That is, we want to check if `T(Args...)` is valid.

To avoid name clashes in the current scope, the `test` functions are wrapped in a struct called `is_constructible_helper`.

```
is_constructible_helper
```

C++

```

1
2
3 struct is_constructible_helper
4
5 {

```

```

6
7
8     template<class T, class... Args, class = decltype(::new T(std::declval<Args>())...)>

        static true_type test(int);


        template<class...>

        static false_type test(...);


};

```

The `is_constructible_helper` declares (but does not define) two static member function templates called `test`. The first version of the `test` function template takes a type `T` (the type we are testing) and a variadic set of arguments `Args...` (the arguments we are using to test if it is possible construct an instance of `T`). If the expression `::new T(Args...)` is valid, then the first version of the `test` function is chosen during overload resolution.

i See `decltype` and `std::declval` if you're not sure what they do.

If, for any reason, the expression `::new T(Args...)` is not a valid expression, then the compiler will choose the second overload of the `test` function, which returns `false_type`.

Next, we'll define the `is_constructible` trait that is derived from `true_type` if `T(Args...)` is valid, or `false_type` otherwise.

is_constructible	C++
<pre> 1 2 3 template <class T, class... Args> </pre>	

```
struct is_constructible : decltype(is_constructible_helper::test<T, Args...>(0))  
  
{};
```

The `is_constructible` class template is derived from the return type of one of the `test` functions. As was just derived, this will be either `true_type` if `T` can be constructed from `Args...`, or `false_type` otherwise.

Next, we'll implement the exact same SFINAE-based type trait, but this time using [partial specialization](#) of a class template.

SFINAE-Out Partial Specializations

Using function overloads is one technique to implement SFINAE-based traits. Another technique uses [partial specialization](#) of a class template. Let's see how we can implement the `is_constructible` type trait using [partial specialization](#).

The basic principal for using [partial specialization](#) for implementing SFINAE-based type traits is to first define the primary template (the template that does not specialize any of the template parameters) which is derived from `false_type` and define a partial specialization that is derived from `true_type` that tests the condition we want to check for. Since the compiler consider a partial specialization a better match than the primary template, the partial partial specialization is used when the template arguments are well-formed.

```
is_constructible  
1  
2  
3 template<class, class T, class... Args>  
4  
5  
6 struct is_constructible_helper : false_type  
7  
8  
{};
```

C++


```

template<class T, class... Args>

struct is_constructible_helper<void_t<decltype(::new T(std::declval<Args>())...)>>,

    T, Args...> : true_type

{};

```

The primary template is defined on lines 1-3. The primary template has three template parameters. The first (unnamed) template parameter is a dummy placeholder for the condition we want to check. The 2nd and 3rd template parameters are the `T` and `Args...` that we want to check if `T` is constructible from `Args...`.

In most cases, the dummy placeholder template parameter that is used to check the condition appears as the last i template parameter in the parameter list. However in this case, the type trait uses a template parameter pack, which must appear at the end of the template parameter list.

The specialized version on lines 5-8 uses `void_t` to test if the expression is valid. As was described earlier in the post, `void_t` can be used with SFINAE-based expressions when the resulting type is not used (since `void_t` maps any number of template arguments to `void`). In this case, we don't care about the type that results from the expression, we just want to test if the expression is valid.

With the helper template defined, we can then create the actual type trait that is derived from the helper trait.

is_constructible	C++
<pre> 1 2 3 template<class T, class... Args> struct is_constructible : is_constructible_helper<void, T, Args...> {}; </pre>	

Whether we use function overloads or partial specialization to define the SFINAE-based type traits, we can define the short-hand version of the type trait the same way:

```
is_constructibleC++
1
2 // Requires C++17.
3
4
5     template<class T, class... Args>
6
7
8     inline constexpr bool is_constructible_v = is_constructible<T, Args...>::value;
```

SFINAE with `enable_if`

The `enable_if` trait uses SFINAE to disable certain function overloads from being considered during overload resolution. As is described earlier, the `enable_if` trait defines a member called `type` (which defaults to `void`) only if the first template argument to `enable_if` evaluates to `true`. If the first template argument to `enable_if` is `false`, then `type` is not defined and attempting to use it would result in a substitution failure.

The `enable_if` utility can be used as:

1. An additional (type or non-type) template parameter,
2. An additional function argument,
3. The return type of a function

To demonstrate the various ways that `enable_if` utility can be used to SFINAE-out function overloads, suppose we have a struct called `Scalar` that can hold either an `int`, a `float`, or a `double`, but no other types are allowed. We want to be able to retrieve the stored value, and update the stored value. Trying to retrieve the internal value using a different type than the stored type or trying to set the stored value to a type other than the stored type should produce an error (either by throwing an exception or by triggering an assertion. For this simple demo, we will use asserts if there is a type mismatch).

The `Scalar` class shown here is purely for demonstration purposes only. I do not recommend you implement something like this in your own projects. In practice, `std::variant` (since C++17) would be a much better choice for solving this kind of problem.

For this example, we also define a few type traits to check for `int`, `float`, and `double` types.

Scalar.hpp

C++

```
1
2
3  template<typename T>
4
5
6  struct is_int : is_same<remove_cvref_t<T>, int>
7
8
9  {};
10
11
12
13
14  template<typename T>
15
16
17  inline constexpr bool is_int_v = is_int<T>::value;
18
19
20
21
22  template<typename T>
23
24
25  struct is_float : is_same<remove_cvref_t<T>, float>
26
27
28  {};
```

```

template<typename T>

inline constexpr bool is_float_v = is_float<T>::value;

template<typename T>

struct is_double : is_same<remove_cvref_t<T>, double>

{};

template<typename T>

inline constexpr bool is_double_v = is_double<T>::value;

```

You should be familiar with the construct of the `is_int`, `is_float`, and `is_double` type traits. If you need a refresher, check out `remove_cvref`, `is_same`, `is_integral`, and `is_floating_point`.

First, we will see how `enable_if` can be used as an additional template parameter to choose a function overload based on type traits.

AS A TEMPLATE PARAMETER

For the `Scalar` class, we will use `enable_if` utility to SFINAE-out the constructor based on the type that is used to initialize an instance of `Scalar`.

Scalar.hpp

C++

```

1
2
3 class Scalar

```

```
4
5 {
6
7
8     enum class Type
9
10
11     {
12
13         Integer,
14
15         Float,
16
17         Double
18
19     };
20
21
22
23
24
25
26
27
28     const Type m_Type;
29
30
31
32
33     union
34
35
36     {
37
38         int m_Int;
39
40
41         float m_Float;
42
43
44         double m_Double;
```

```
};
```

```
public:
```

```
template<typename I, enable_if_t<is_int_v<I>>* = nullptr>
```

```
Scalar(I i)
```

```
    : m_Type(Type::Integer)
```

```
    , m_Int(i)
```

```
{}
```

```
template<typename F, enable_if_t<is_float_v<F>>* = nullptr>
```

```
Scalar(F f)
```

```
    : m_Type(Type::Float)
```

```
    , m_Float(f)
```

```
{}
```

```

template<typename D, enable_if_t<is_double_v<D>>* = nullptr>

Scalar(D d)

    : m_Type(Type::Double)

    , m_Double(d)

{}

```

In this example, three constructors for the `Scalar` class are defined on lines 41-57. The constructors are template member functions that have two template parameters.

1. The first template parameter is the type of the passed parameter.
2. The second template parameter uses `enable_if` to SFINAE-out the constructor based on the type of the first template parameter.

The construct of the second template parameter might look strange. Let's take a closer look at the first constructor:

Scalar.hpp

C++

```

1
2
3 struct Scalar
4
5 {

    template<typename I, enable_if_t<is_int_v<I>>* = nullptr>

    Scalar(I i)

```

...

The second template parameter doesn't use `typename` to declare the template parameter. This is because the second template parameter is a [non-type template parameter](#). If `I` is an `int`, then this is what the compiler would produce:

Scalar.hpp	C++
<pre>1 2 3 struct Scalar 4 5 { template<typename I, void* = nullptr> Scalar(I i) ...</pre>	

Since pointer types are perfectly valid as non-type template parameters (see [Non-type Template Parameters](#)), this code compiles.

A common mistake when using `enable_if` as an additional template parameter, is to use it as a type template parameter with a default template argument like this:

Scalar.hpp	C++
<pre>1 2 3 template<typename I, typename = enable_if_t<is_int_v<I>>> Scalar(I i)</pre>	

...

But this only creates an ambiguous function overload since the default template arguments are not taken into consideration during overload resolution and cannot be used to SFINAE-out function overloads!

Only failures in the types and expressions in the *immediate context* of a function type or its template parameter types are SFINAE errors. Default template arguments are not part of the *immediate context* and therefore cannot be used to SFINAE-out function overloads.

Next, we'll see how `enable_if` can be used as an additional function argument to SFINAE-out function overloads.

AS A FUNCTION ARGUMENT

`enable_if` can also be used as an additional function argument. To demonstrate this, we'll implement a set of functions called `set` in the `Scalar` class that are used to update the internal value. The overload that is used is determined by the type of the first function argument to the `set` function.

Scalar.hpp

C++

```
1
2
3     template<typename I>
4
5
6     void set(I i, enable_if_t<is_int_v<I>>* = nullptr)
7
8
9     {
10
11
12         assert(m_Type == Type::Integer);
13
14
15         m_Int = i;
16
17
18     }
19
```

```
template<typename F>

void set(F f, enable_if_t<is_float_v<F>>* = nullptr)

{

    assert(m_Type == Type::Float);

    m_Float = f;

}

template<typename D>

void set(D d, enable_if_t<is_double_v<D>>* = nullptr)

{

    assert(m_Type == Type::Double);

    m_Double = d;

}
```

Similar to the way that `enable_if` is used to SFINAE-out the constructors, it is used here as an additional function

argument to SFINAE-out the overload of the `set` member function.

In the case of the first overload on lines 59-60, if `I` is an `int`, this is what the compiler produces:

Scalar.hpp	C++
<pre>1 2 3 void set(int i, void* = nullptr) 4 5 { assert(m_Type == Type::Integer); m_Int = i; }</pre>	

Optionally, we can name the second parameter, but since it's not being used in the body of the function, doing so **!** may cause a warning about the unused function parameter. Omitting the name of the unused function parameters avoids that warning.

Another way to use `enable_if` to SFINAE-out function overloads is as the return type of the function.

AS A RETURN TYPE

`enable_if` can also be used as the return type of a function. To demonstrate this, we'll create a set of member functions for the `Scalar` class called `get`:

Scalar.hpp	C++
<pre>1 2 3 template<typename I></pre>	

```
4
5
6     enable_if_t<is_int_v<I>, I> get() const
7
8
9     {
10
11
12         assert(m_Type == Type::Integer);
13
14
15         return m_Int;
16
17
18     }
19
20
```

```
template<typename F>
```

```
enable_if_t<is_float_v<F>, F> get() const
```

```
{

    assert(m_Type == Type::Float);

    return m_Float;

}
```

```
template<typename D>
```

```

enable_if_t<is_double_v<D>, D> get() const

{

    assert(m_Type == Type::Double);

    return m_Double;

}

```

If you recall from the derivation of the `enable_if` utility template, the second template argument is the type that is used for the return value of the function if the first template argument evaluates to `true`. In this case, we could have written:

Scalar.hpp

C++

```

1
2
3     template<typename I>

    enable_if_t<is_int_v<I>, int> get() const

    ...

```

Which may have been a better choice since we know `I` must be `int` if this overload is chosen. Regardless, the return value of the function is actually the second template argument of the `enable_if` utility template.

In this case, there are three overloads of the `get` function. The version of the `get` function that is chosen during overload resolution is determined by the type of the template argument. For example, if the template argument is `int`, then the compiler will generate something like this:

Scalar.hpp

C++

```

1
2
3     int get() const
4
5     {

        assert(m_Type == Type::Integer);

        return m_Int;

    }

```

We've now seen three ways of using `enable_if` to SFINAE-out function template overloads. But what is the best technique to use? Let's look at that next.

SUMMARY

To summarize, we've seen three different techniques showing how to use `enable_if` to SFINAE-out function template overloads. The following table attempts to summarize when you should use which technique.

USAGE	USE CASE
Template Parameter	When used with special member functions like constructors and destructors that don't have a return type. Also for operator overloads of a class that have a specific function signature with a fixed number of function arguments.
Function Argument	There is probably no good use case for using this form. Since it adds an additional parameter to the function signature, you add the risk that the end user thinks that they need to pass an argument. The dummy argument also appears in IntelliSense in the Visual Studio IDE (or whatever IDE you are using) which may cause more confusion for the poor end user who doesn't understand the <code>enable_if</code> construct. I would advise against using <code>enable_if</code> as a function argument.

Return Type Whenever possible. Using `enable_if` as the return type of a function does not change the number of template parameters nor the number of function arguments, this is arguably the least intrusive way of using `enable_if`. This form should be preferred whenever possible.

If you think that the use of `enable_if` is clumsy and makes the code harder to read, well then join the club. Luckily the C++ standards committee agrees with you, which is why C++20 introduces *concepts*.

Concepts

In the previous section, we used `enable_if` to constrain the types of template parameters in class and function templates. Although it is possible to use `enable_if` to impose constraints on template parameters, it's not the most elegant syntax, nor does it improve the readability of the diagnostic error messages produced by the compiler when those constraints are violated. C++20 *concepts* are a way to express a set of constraints on template parameters that make it possible to:

- Clearly communicate the constraints on template parameters, providing better “self-documenting code”
- Improve the diagnostic error messages from a compiler when constraints are not met
- Specialize function and class templates based on a set of type constraints

Concept Definition

A *concept* refers to a template for a named set of *constraints* where each constraint is defined by one or more *requirements* for the set of template parameters. A *concept definition* has the following form:

```
template< template-parameter-list >  
concept concept-name = constraint-expression;
```

A concept's parameter list specifies one or more template parameters. Similar to function and class templates, these can be either `type` or `non-type` template parameters. Unlike regular templates, concepts are never instantiated by the compiler and they never produce code that is executed at run-time. They are used to define a set of constraints on the template parameters that the compiler uses to check if the type satisfies those constraints.

The constraints in the *constraint expression* is a logical set of boolean expressions that consists of conjunctions (using `&&`) and disjunctions (using `||`) that must evaluate (at compile time) to either `true` if the constraints on the template parameter are satisfied, or `false` otherwise.

If a type `T` satisfies the *constraint expression*, that is, the *constraint expression* evaluates to `true` for the given type `T`, then it is said that the type `T` *models* the concept.

Small concept	C++
<pre>1 2 template <typename T> concept Small = sizeof(T) <= sizeof(int);</pre>	

In the above example, the type `T` *models* the `Small` concept if the size of `T` is not larger than the size of an `int`.

Besides regular C++ expressions, the *constraint expression* can be defined in terms of type traits (that evaluate to `true` or `false`) and they can also be either a *concept expression*, or a *requires expression*.

In the following sections, we will look at *concept expressions* and *requires expressions*.

Concept Expression

A *concept expression* is used to verify if a give type models a concept. A concept expression consists of the name of a previously defined concept followed by a set of template arguments between angle brackets. For example, `Small<char>` and `Small<short>` are concept expressions that evaluate to `true`, but `Small<double>` and `Small<long double>` generally evaluate to `false`.

Concepts can be defined in terms of previously defined named concepts by using a *concept expression*.

Integral concept	C++
<pre>1 2 3 template <class T></pre>	


```

4
5
6 concept Integral = is_integral_v<T>;
7
8

template <class T>

concept SignedIntegral = Integral<T> && is_signed_v<T>;

template <class T>

concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;

```

In this example, the `SignedIntegral` concept is defined in terms of the previously defined `Integral` concept, and the `UnsignedIntegral` is in-turn defined by the previously defined `Integral` and `SignedIntegral` concepts.

Although you can get pretty far with defining concepts using *concept expressions*, at some point, you may find that there isn't a concept expression that tests the requirements of the type that you need. In this case you can use a `requires` expression.

Requires Expression

A `requires` expression has one of the following forms:

```

requires { requirement-seq }
requires ( parameter-list ) { requirement-seq }

```

The optional *parameter list* is a comma-separated list of typed arguments that can be used in the *requirement sequence*. The *requirement sequence* is a list of one or more expressions that are evaluated by the compiler. The expressions in the

requirement sequence never produces executable code. They are only used by the compiler to check if the expressions form valid C++ code.

Each requirement in the requirement sequence can be one of the following types:

- Simple requirement
- Type requirement
- Compound requirement
- Nested requirement

In the following sections, we'll look at each type of requirement.

SIMPLE REQUIREMENT

A *simple requirement* is an arbitrary C++ expression that is evaluated by the compiler for correctness. A simple requirement has one of the the following forms:

```
requires { simple-requirement; }  
requires ( parameter-list ) { simple-requirement; }
```

An example of a simple requirement that checks if two types can be added together might look like this:

Addable	C++
<pre>1 2 3 template<class T, class U> 4 concept Addable = requires (const T& t, const U& u) { t + u; };</pre>	

The `Addable` concept shown here is simplified for demonstration purposes. For example, you may want to check if both `t + u` and `u + t` are valid expressions. You may also want to account for cases where `T` and `U` are already reference types by adding `remove_reference_t` to the parameter types.

A *simple requirement* is just a C++ expression that is terminated by a semicolon (`;`).

TYPE REQUIREMENT

A *type requirement* is used to check for the existence of a named nested type. A type requirement has the following form:

```
requires { typename name; }  
requires ( parameter-list ) { typename name; }
```

Where `name` refers to a nested member type, an alias type, or a class template type.

The following example checks if the template argument `T` has a nested member type called `value_type`:

HasValueType	C++
<pre>1 2 3 template<class T> 4 concept HasValueType = requires { typename T::value_type; };</pre>	

Another common usage for a type requirement is to check if a given type can be used to instantiate a specific class template. For example, if your code requires some template argument to be used with an `std::vector`, then you can check

that with this concept:

WorksWithVectors	C++
<pre>1 2 #include <vector> 3 4 5 6 template<class T> concept WorksWithVectors = requires { typename std::vector<T>; };</pre>	

The `WorksWithVectors` concept checks if instantiating `std::vector<T>` would not produce a compiler error.

The `WorksWithVectors` concept may erroneously succeed with abstract class types. The `WorksWithVectors` concept only checks if a vector can be instantiated with a specific type, but it does not check any of the operations that can be performed on an `std::vector` of type `T`.

COMPOUND REQUIREMENT

Similar to *simple requirements*, *compound requirements* are used to check the validity of a C++ expression. In addition to simple requirements, compound requirements can also verify that the result of the expression meets some kind of type constraint or that the expression does not throw an exception.

Compound requirements have one of the following forms:

```
requires { { compound-requirement }; }
```

```
// Same as a simple requirement
```

```

requires { { compound-requirement } noexcept; }           // compound-requirement does not throw an exception
requires { { compound-requirement } -> type-constraint; }   // The result of compound-requirement satisfies type-cons
requires { { compound-requirement } noexcept -> type-constraint; } // The result of compound-requirement satisfies type-cons
// And parameter-list variants:
requires ( parameter-list ) { { compound-requirement }; }   // Same as a simple requirement
requires ( parameter-list ) { { compound-requirement } noexcept; } // compound-requirement does not throw
requires ( parameter-list ) { { compound-requirement } -> type-constraint; } // The result of compound-requirement
requires ( parameter-list ) { { compound-requirement } noexcept -> type-constraint; } // The result of compound-requirement

```

For example, the `EqualityComparable` concept could be implemented like this:

EqualityComparable	C++
<pre> 1 2 3 template<class T, class U> 4 5 6 concept Same = is_same_v<T, U>; 7 8 template<class T> concept EqualityComparable = requires (const T & a, const T & b) { { a == b } -> Same<bool>; { a != b } -> Same<bool>; }; </pre>	

The `EqualityComparable` concept uses two compound requirements to check if an object of type `T` defines the `==` and `!=` operators and that those operators return a `bool` result.

! Notice that the semicolon comes at the end of the compound requirement. There is no semicolon inside the body of the compound requirement.

NESTED REQUIREMENT

A *nested requirement* is used to test a *constraint expression* inside of a parent *requires expression*. The *constraint expression* of a *nested requirement* can use any of the local parameters introduced in any one of the the parent *requires expression*.

A *nested requirement* can have one of the following forms:

```
requires { requires constraint-expression; }  
requires ( parameter-list ) { requires constraint-expression; }
```

Similar to the *constraint expression* of the [concept definition](#), The *constraint expression* of a *nested requirement* is a logical set of boolean expression that consist of conjunctions (`&&`) and disjunctions (`||`) that are evaluated at compile-time to `true` if the constraint is satisfied or `false` otherwise.

The *constraint expression* can be:

- A C++ (compile-time) expression that evaluates to `true` or `false`
- A type trait that evaluates to `true` or `false`
- A [concept expression](#)
- A [requires expression](#)

Using nested requirements, it is technically possible to create atrocities such as this (but please don't do this in your own code):

AddressOf

C++

```
1  
2  
3 template<class T, class U>  
4
```

```

5
6 concept Same = is_same_v<T, U>; // Uses is_same type trait.
7
8
9
10
11 template<class T>
12
13 concept AddressOf = requires (T t) {           // Requires expression
    requires requires (T u) {                   // Nested requirement
        requires requires (T v) {               // Nested requirement
            requires requires (T w) {             // Nested requirement
                requires Same<T*, decltype(&w)>; // Concept expression
            };
        };
    };
};

```

This of course can be simplified to just a single nested requirement:

AddressOf

C++

```

1
2

```

```

3 template<class T, class U>
4
5
6 concept Same = is_same_v<T, U>;
7

template<class T>

concept AddressOf = requires (T t) {

    requires Same<T*, decltype(&t)>;

};

```

The `AddressOf` concept uses a nested requirement that, in turn, uses a `concept expression` to verify that the address of (`&`) operator applied to an instance of `T` results in `T*`. This concept is useful to check if `T` does not overload the address of operator to return a different type than expected.

i `std::addressof` was added in C++11 to obtain the actual address of an object, even in the presence of an overloaded address of operator (`&`).

Requires Clause

The *requires clause* can be placed after the template parameter list in a class template or a function template definition. The template parameter list and the requires clause together form the *template head*. A template definition that uses a requires clause has this form:

```

template< parameter-list > requires constraint-expression
template body;

```


 Notice that the `requires` clause is not terminated by a semicolon but any sub expressions of the constraint expression are.

The *constraint expression* of the `requires` clause is analogous to the constraint expression of the [concept definition](#) and the [requires expression](#). That is, the constraint expression is a logical set of boolean expressions consisting of conjunctions (`&&`) or disjunctions (`||`) of one or more constant expressions that evaluate to `true` or `false` at compile time. The boolean expressions can be:

- A C++ (compile-time) [primary expression](#) that evaluates to `true` or `false`
- A type trait that evaluates to `true` or `false`
- A [concept expression](#)
- A [requires expression](#)

Using the same example from before, the following example shows that it is possible to use a nested `requires` expression in the `requires` clause:

Requires clause

C++

```
1
2
3 template<typename T> requires requires (T t) {
4
5
6     requires requires (T u) {
7
8
9         requires requires (T v) {
10
11             requires Same<T*, decltype(&v)>;
12
13         };
14
15     };
16 }
```

```

}

T* addressof(T& t)
{

    return &t;

}

```

! I don't recommend you use this in your own code. I recommend you use `std::addressof` instead. The example shown here demonstrates that you *can* use nested `requires` expressions in a `requires` clause.

The `requires` keyword appears twice in the template head because the first one introduces the *requires clause* and the second one introduces a *requires expression*.

Using a `requires` clause, we can constrain the template arguments that are used in the `max` function template from the [Function Templates](#) section above:

max	C++
<pre> 1 2 3 template<typename T, typename U> requires requires (T a, U b) { { a > b } -> Same<bool>; } 4 5 auto max(T a, U b) -> remove_reference_t<decltype(a > b ? a : b)> { return a > b ? a : b; } </pre>	

Trying to use the `max` function template with a types that do not define the greater than operator (`>`) will result in a compiler error that states something like “the associated constraints are not satisfied”. It should be possible to improve this diagnostic error message by creating a named concept definition:

maxC++

```
1
2
3  template<class T, class U>
4
5
6  concept Same = is_same_v<T, U>;
7
8
9
10
11 template<typename T, typename U>

    concept GreaterThanComparableWith = requires (T a, U b) { { a > b } -> Same<bool>; };

    template<typename T, typename U> requires GreaterThanComparableWith<T, U>

    auto max(T a, U b) -> remove_reference_t<decltype(a > b ? a : b)>

    {

        return a > b ? a : b;

    }
```

Ideally, the compiler should generate better diagnostics when using named concepts. At the time of this writing, only the GCC compiler mentions the constraint that is being violated. Regardless, you should prefer to use named constraints in a `requires` clause to improve the compiler diagnostic error messages (at least, in the future).

It is also possible to specify the `requires` clause after the parameter list of a function declaration. For example, the two function template definitions are identical:

Trailing requires clause	C++
<pre>1 2 // Regular requires clause 3 4 5 template<typename T, typename U> requires GreaterThanComparableWith<T, U> 6 7 8 auto max(T a, U b) 9 10 { 11 12 13 return a > b ? a : b; 14 15 } 16 17 // Trailing requires clause 18 19 template<typename T, typename U> 20 21 auto max(T a, U b) requires GreaterThanComparableWith<T, U> 22 23 { 24 25 return a > b ? a : b; 26 27 }</pre>	

The second form of the `max` function template uses a *trailing requires clause*. It is also possible to combine a regular `requires` clause with a trailing `requires` clause in the same function template declaration. This will come in handy w

want to constrain the template parameters of a member function template of a class template.

SHORTHAND NOTATION

Instead of using a *requires clause*, a constraint can be placed directly on a template parameter. A named concept can be used in the template parameter list instead of `typename` (or `class`).

For example, the `max` function template can be written without the `requires` clause:

```
maxC++
1
2
3 template<typename T, GreaterThanComparableWith<T> U>
4
5     auto max(T a, U b) -> remove_reference_t<decltype(a > b ? a : b)>
6
7     {
8
9         return a > b ? a : b;
10
11     }
```

In the above example, the second template parameter becomes `GreaterThanComparableWith<T> U` which moves the type constraint out of the `requires` clause and places the constraint directly on the type of `U`. This shorthand notation also allows us to omit the first template parameter from the named concept. In this case, `U` is implicitly used as the first argument to `GreaterThanComparableWith` and `T` is explicitly used as the second argument. That is, `GreaterThanComparableWith<T> U` (when used in the template parameter list) is equivalent to `GreaterThanComparableWith<U, T>` (when used in the `requires` clause).

The astute reader will realize that the shorthand notation, used in this way, changes the result of the concept. Instead of checking for the validity of `a > b` (as was the case when using the `requires` clause), the concept now checks `b > a` which may not be the intention and may fail. To fix this issue, you could swap the order of the types in the template parameter list:

```
maxC++
```

```

1
2
3 template<typename U, GreaterThanComparableWith<U> T>
4
5
6 auto max(T a, U b) -> remove_reference_t<decltype(a > b ? a : b)>
7
8 {
9
10     return a > b ? a : b;
11
12 }

```

Which would be fine if the end user relies on implicit type deduction. If the end user tries to specify the types explicitly, but gets the order of the types wrong, this could result in unintended behaviour.

The following is not allowed:

max	C++
<pre> 1 2 3 template<GreaterThanComparableWith<U> T, GreaterThanComparableWith<T> U> 4 5 6 auto max(T a, U b) -> remove_reference_t<decltype(a > b ? a : b)> 7 8 { 9 10 return a > b ? a : b; 11 12 } </pre>	

The first occurrence of `GreaterThanComparableWith<U>` on line 7 will fail since `U` was not defined yet.

If the concept expression only applies to a single template parameter, then the template argument on the constraint can be omitted.

omitted:

Increment	C++
1	
2	
3	<code>template<typename T></code>
4	
5	
6	<code>concept Increment = requires (T a) { ++a; a++; };</code>
7	
8	
	<code>template<Increment T></code>
	<code>T increment(T a)</code>
	<code>{</code>
	<code> return ++a;</code>
	<code>}</code>

In this case, the type `T` is constrained with the `Increment` concept. Since `Increment` only applies to a single template parameter, using the shorthand notation, `T` is used as an implicit template argument and can be omitted.

Constrained Class Templates

Constraining the template parameters of a class template is similar to that of a function template.

If you recall from the section about [template arguments versus template parameters](#), we defined a simple `Array` class template. We can now constrain the types that can be used with the `Array` class template.

Array	C++
1	
2	

```
3  template<typename T, size_t N>
4
5
6  requires std::default_initializable<T> && std::destructible<T>
7
8
9  class Array
10 {
11 {
12
13
14 public:
15
16     Array()
17
18
19         : m_Data{}
20
21
22     {}
23
24
25
26
27     size_t size() const
28
29
30     {
31
32         return N;
33
34     }
35
36
37     T& operator[](size_t i)
```



```
{  
  
    assert(i < N);  
  
    return m_Data[i];  
  
}  
  
const T& operator[](size_t i) const  
  
{  
  
    assert(i < N);  
  
    return m_Data[i];  
  
}  
  
private:  
  
    T m_Data[N];  
  
};
```

`std::default_initializable` is a concept which checks if the type can be created using a default constructor. This eliminates types that have an inaccessible default constructor (because it is either deleted, protected, or private to the class). Additionally, the `std::destructible` concept checks if a type can be safely destroyed at the end of its lifetime.

If you want to define the member functions of a class template outside of the class declaration, you must repeat the `requires` clause.

Array

C++

```
1
2 // Array header file.
3
4
5 template<typename T, size_t N>
6
7
8 requires std::default_initializable<T> && std::destructible<T>
9
10
11 class Array
12 {
13 {
14
15
16 public:
17
18
19     Array();
20
21
22
23
24     size_t size() const;
25
26
```

...

```

};

// Array implementation file.

template<typename T, size_t N>

requires std::default_initializable<T> && std::destructible<T>

Array<T, N>::Array()

    : m_Data{}

{}

template<typename T, size_t N>

requires std::default_initializable<T>&& std::destructible<T>

size_t Array<T, N>::size() const
{

    return N;

}

```

This example shows the `Array` constructor and `size` member functions are defined outside of the `Array` declaration. In this

case, the template head (including the requires clause) must be repeated for each of the member functions of the class template.

CONSTRAINED CLASS MEMBERS

It can happen that you need to specify additional constraints on the member functions of a class template. For example, if we wanted to implement copy semantics on the `Array` class template, we would need to specify additional constraints the copy constructor and the assignment operator of the `Array` class:

```
ArrayC++
1
2 // Array header file.
3
4
5 template<typename T, size_t N>
6
7
8 requires std::default_initializable<T> && std::destructible<T>
9
10
11 class Array
12
13 {
14
15
16 public:
17
18
19     Array();
20
21
22
23
24     Array(const Array & copy) requires std::copyable<T>;
25
26
27
28
29     Array& operator=(const Array& rhs) requires std::copyable<T>;
30
```

```
31
32
33
34 ...
35
36
37 };
38
39

// Array implementation file.

template<typename T, size_t N>

requires std::default_initializable<T> && std::destructible<T>

Array<T, N>::Array()

    : m_Data{}

{}

template<typename T, size_t N>

requires std::default_initializable<T>&& std::destructible<T>

Array<T, N>::Array(const Array& copy) requires std::copyable<T>

{

    std::ranges::copy_n(copy.m_Data, N, m_Data);
```

```

}

template<typename T, size_t N>

requires std::default_initializable<T>&& std::destructible<T>

Array<T, N>& Array<T, N>::operator=(const Array& rhs) requires std::copyable<T>
{
    if (&rhs != this)
    {
        std::ranges::copy_n(rhs.m_Data, N, m_Data);
    }

    return *this;
}

```

Although the code is starting to look a bit unwieldy, the example demonstrates that:

- The `requires` clause after the template parameter list constrains the class template parameters.
- The `requires` clause after the member function can further constrain the types. These constraints are only checked if the member function is instantiated.
- The `requires` clause on the class template parameter list and the member function must be repeated in the definition.

the member function (when defined outside the class declaration)

Concept-Based SFINAE

Similar to how the `enable_if` utility template is used to SFINAE-out function overloads, we can now use concepts instead. To demonstrate this, we'll modify the `Scalar` class template from the [SFINAE with `enable_if`](#) example above. First, we'll define a few concepts that mimic the `is_int`, `is_float`, and `is_double` type traits:

```
Scalar.hpp C++
1
2
3 template<typename T>
4
5
6 concept Int = is_int_v<T>;
7
8
9
10
11 template<typename T>
12
13
14 concept Float = is_float_v<T>;
15
16
17
18
19 template<typename T>
20
21
22 concept Double = is_double_v<T>;
```

In the previous example, we used `enable_if` as an additional template parameter to SFINAE-out the constructor of the `Scalar` class based on the template argument type. Using concepts and the [shorthand notation](#), this can be written much more succinctly:

```
Scalar.hpp
1
```

```
2
3 class Scalar
4 {
5
6
7     ...
8
9
10
11 public:
12
13     template<Int I>
14
15     Scalar(I i)
16
17         : m_Type(Type::Integer)
18
19
20         , m_Int(i)
21
22     {}
23
24     template<Float F>
25
26     Scalar(F f)
27
28         : m_Type(Type::Float)
29
30
31         , m_Float(f)
32
33     {}
```



```
template<Double D>

Scalar(D d)

    : m_Type(Type::Double)

    , m_Double(d)

{}

```

Instead of adding an additional template parameter using `enable_if`, we can constrain the template parameter directly using a named concept in the template parameter list. I hope you agree that using concepts to choose the correct function overload is much nicer looking than using `enable_if`.

The other functions of the `Scalar` class can also be improved using the same technique:

Scalar.hpp

C++

```
1
2
3     template<Int I>
4
5
6     void set(I i)
7
8
9     {
10
11
12         assert(m_Type == Type::Integer);
13
14

```

```
15         m_Int = i;
16
17
18     }
19
20
21
22
23     template<Float F>
24
25
26     void set(F f)
27
28
29     {
30
31
32         assert(m_Type == Type::Float);
33
34
35         m_Float = f;
36
37
38     }
39
40
41
42
43     template<Double D>
44
45
46     void set(D d)
47
48
49     {
50
51
52         assert(m_Type == Type::Double);
53
54
55         m_Double = d;
```

```
}
```

```
template<Int I>
```

```
I get() const
```

```
{
```

```
    assert(m_Type == Type::Integer);
```

```
    return m_Int;
```

```
}
```

```
template<Float F>
```

```
F get() const
```

```
{
```

```
    assert(m_Type == Type::Float);
```

```
    return m_Float;
```

```
}
```

```

template<Double D>

D get() const

{

    assert(m_Type == Type::Double);

    return m_Double;

}

```

No more extraneous template parameters, function arguments, or ugly return types using `enable_if` when named concepts will suffice.

I realize that the `Scalar` class is a contrived example that could be solved with just regular function overloading `i` or `if constexpr`. Regardless, I hope you have a better understanding of how concepts can be used to choose function overloads based on type traits.

Constraining Auto & Abbreviated Function Templates

Concepts can be used to constrain `auto`. Wherever `auto` can be used, `Concept auto` can also be used (where `Concept` is a previously defined named concept).

For example, suppose we have the following function template:

```
add
```

```
1
2
3 template<typename T>
4
5
6 concept Arithmetic = is_arithmetic_v<T>;
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

This form of the `add` function uses the [shorthand notation](#) to place constraints on the types `T` and `U`. This function can also be written as an *abbreviated function template*:

```
add
1
2
3 auto add(Arithmetic auto a, Arithmetic auto b)
4
5 {
6
7     return a + b;
8
9 }
```

The *abbreviated function template* notation replaces the template parameter list with *constrained* `auto` function parameters.

i Note: You can also use *abbreviated function templates* without constraints.

Constraints can also be placed on the return value of the function:

add	C++
<pre>1 2 3 Arithmetic auto add(Arithmetic auto a, Arithmetic auto b) 4 5 { 6 7 return a + b; 8 9 }</pre>	

We can further constrain the function parameters using a *trailing requires clause*, but when using *abbreviated function templates*, we need to use the `decltype` specifier to get at the underlying type:

add	C++
<pre>1 2 3 template<class T, class U> 4 5 6 concept Addable = requires (const T& t, const U& u) { 7 8 t + u; 9 10 }; 11 12 13 Arithmetic auto add(Arithmetic auto a, Arithmetic auto b) requires Addable<decltype(a), decltype(b)></pre>	

```
{  
  
    return a + b;  
  
}
```

i You may argue that if `a` and `b` are `Arithmetic` types, then they will also be `Addable`. The example demonstrates how to specify additional requirements when using *abbreviated function templates*.

And we can also constrain the expected return value of a function:

Constrained auto

C++

```
1  
2  
3 Arithmetic auto i = add(3, 5);  
  
Arithmetic auto j = add(3.0, 5);  
  
Arithmetic auto k = add(3, 5.0f);
```

In the above example, `i`, `j`, and `k` are constrained to be valid `Arithmetic` types (see `is_arithmetic` for more information) and the return value from the `add` function template is determined by the types of the arguments (which must also be arithmetic types).

Conclusion

A lot was covered in this article and if you got this far then congratulations! You are now an expert on C++ templates. You learned about [value categories](#), [template arguments and template parameters](#), [function templates](#), [class templates](#), and [variable templates](#). You also learned about the many uses of the [typename keyword](#) and how to [specialize templates](#). You've also seen example of using [variadic templates](#) and how to use a [template parameter pack](#) to implement [recursive function templates](#).

You also learned how to use the `decltype specifier` and `std::declval` to form correct template expressions. You were bombarded with a set of (almost) 40 `type traits` that can be used to query or transform your template types and give you the tools needed to specify a set of constraints on template arguments. This article doesn't even cover half of the type traits that are available in the standard template library (see `type_traits` for more information).

You learned about `SFINAE` and the various ways to utilize `SFINAE` to implement more complex type traits. You also learned about the various ways that `enable_if` can be used to specify constraints on template arguments. And finally, you learned about C++20 `Concepts` that allow you to specify constraints on template arguments without using `enable_if`.

As an added bonus, you also learned about `abbreviated function templates` that lets you write (constrained) function templates using a very succinct syntax.

But despite covering all of these topics, there are still a few that I'd like to cover:

1. Typelists
2. Type erasure

And possibly a few more templates related topics that I can't even think of right now. In any case, I think this article provides a good foundation for getting started with C++ templates and template meta programming. Thanks for reading, and please leave a comment if you notice any omissions or corrections or if you can think of any templates related topics that you'd like me to cover.

Bibliography

- [1] D. Vandevoorde, N. M. Josuttis, and D. Gregor, C++ templates: the complete guide, Second edition. Boston: Addison-Wesley, 2018.
- [2] B. Stroustrup, "'New' Value Terminology." Accessed: Jul. 06, 2020. [Online]. Available: <https://www.stroustrup.com/terminology.pdf>.
- [3] "Value categories - cppreference.com." https://en.cppreference.com/w/cpp/language/value_category (accessed Jul. 06, 2020).
- [4] "Understanding lvalues and rvalues in C and C++ - Eli Bendersky's website." <https://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c> (accessed Jul. 06, 2020).
- [5] "C++11 Tutorial: Explaining the Ever-Elusive Lvalues and Rvalues," SmartBear.com. <https://smartbear.com/blog/dev>

[/c11-tutorial-explaining-the-ever-elusive-lvalues-a/](#) (accessed Jul. 06, 2020).

[6] W. M. Miller, “A Taxonomy of Expression Value Categories,” p. 20.

[7] S. Meyers, Effective modern C++: 42 specific ways to improve your use of C++11 and C++14, First edition. Beijing ; Sebastopol, CA: O’Reilly Media, 2014.

[8] “Universal References in C++11 – Scott Meyers : Standard C++.” <https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers> (accessed May 16, 2021).

[9] I. Horton and P. van Weert, Beginning C++20: from novice to professional. 2020.

[10] “c++ – How is std::is_function implemented?,” Stack Overflow. <https://stackoverflow.com/questions/59654482/how-is-stdis-function-implemented> (accessed Apr. 29, 2021).

[11] “c++ – What is std::decay and when it should be used?,” Stack Overflow. <https://stackoverflow.com/questions/25732386/what-is-stddecay-and-when-it-should-be-used> (accessed May 13, 2021).

[12] C++ Weekly With Jason Turner, C++ Weekly – Ep 189 – C++14’s Variable Templates, (Oct. 14, 2019). Accessed: Apr. 13, 2021. [Online Video]. Available: <https://www.youtube.com/watch?v=2kY-go52rNw>

[13] “Constraints and concepts (since C++20) – cppreference.com.” <https://en.cppreference.com/w/cpp/language/constraints> (accessed May 21, 2021).

[14] “Expressions – cppreference.com.” <https://en.cppreference.com/w/cpp/language/expressions> (accessed Jul. 06, 2020).

[15] “Function Templates Partial Specialization in C++,” Fluent C++, Aug. 15, 2017. <https://www.fluentcpp.com/2017/08/15/function-templates-partial-specialization-cpp/> (accessed Mar. 30, 2021).

[16] gcc-mirror/gcc. gcc-mirror, 2021. Accessed: Apr. 06, 2021. [Online]. Available: <https://github.com/gcc-mirror/gcc>

[17] llvm/llvm-project. LLVM, 2021. Accessed: Apr. 06, 2021. [Online]. Available: <https://github.com/llvm/llvm-project>

[18] microsoft/STL. Microsoft, 2021. Accessed: Apr. 06, 2021. [Online]. Available: <https://github.com/microsoft/STL>

[19] B. Stroustrup, “‘New’ Value Terminology.” Accessed: Jul. 06, 2020. [Online]. Available: <https://www.stroustrup.com/terminology.pdf>

[20] “SFINAE – cppreference.com.” <https://en.cppreference.com/w/cpp/language/sfinae> (accessed May 21, 2021).

[21] “Singleton.” <https://refactoring.guru/design-patterns/singleton> (accessed Apr. 19, 2021).

[22] “Template parameters and template arguments – cppreference.com.” https://en.cppreference.com/w/cpp/language/template_parameters (accessed Mar. 31, 2021).

[23] “Type alias, alias template (since C++11) – cppreference.com.” https://en.cppreference.com/w/cpp/language/type_alias (accessed Apr. 13, 2021).

[24] “Yet another type-trait: decay.” <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2069.html> (accessed May 13, 2021).

This entry was posted in [C++ Tutorials](#), [Programming](#), [templates](#) and tagged [abbreviated function template](#), [alias template](#),

[C++](#), [class template](#), [Concepts](#), [decltype](#), [decltype](#), [function template](#), [glvalue](#), [lvalue](#), [prvalue](#), [requires](#), [rvalue](#), [SFINAE](#), [template parameter pack](#), [template specialization](#), [templates](#), [tutorial](#), [type deduction](#), [type traits](#), [typename](#), [value categories](#), [variable template](#), [xvalue](#) by [Jeremiah](#). Bookmark the [permalink \[https://www.3dgep.com/beginning-cpp-template-programming/\]](https://www.3dgep.com/beginning-cpp-template-programming/) .

1 THOUGHT ON “C++ TEMPLATE PROGRAMMING”



stephane

on [July 23, 2022 at 12:17 pm](#) said:

Best resources I ever found

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)