Modern C++ Programming

10. Templates and Meta-programming II

Class Templates , SFINAE, and Concepts

Federico Busato

2023-11-14

Class Template

Class Template

Similarly to function templates, **class templates** are used to build a family of classes

```
template<typename T>
struct A { // template class (typename template)
    T x = 0;
};
template<int N1>
struct B { // template class (numeric template)
    int N = N1;
};

A<int>    a1; // a1.x is int    x = 0
A<float>  a2; // a2.x is float  x = 0.0f
B<1>      b1; // b1.N is 1
B<2>      b2; // b2.N is 2
```

The *main difference* with template functions is that classes can be **partially** specialized

Note: Every class specialization (both partial and full) is a completely **new class** and it does not share anything with the generic class

```
template<typename T, typename R>
struct A {};            // generic template class

template<typename T>
struct A<T, int> {};    // partial specialization

template<>
struct A<float, int> {}; // full specialization
```

Template Class Specialization


```cpp
template<typename T, typename R>
struct A {              // GENERIC template class
    T x;
};

template<typename T>
struct A<T, int> {      // PARTIAL specialization
    T y;
};

A<float, float> a1;
a1.x;     // ok, generic template
// a1.y; // compile error

A<float, int> a2;
a2.y;     // ok, partial specialization
// a2.x; // compile error
```

Example 1: Implement a Simple Type Trait

```
template<typename T, typename R> // GENERIC template declaration
struct is_same {
    static constexpr bool value = false;
};

template<typename T>
struct is_same<T, T> {           // PARTIAL template specialization
    static constexpr bool value = true;
};

cout << is_same<  int,  char>::value; // print false, generic template
cout << is_same<float, float>::value; // print true, partial template
```

Example 2: Check if a Pointer is `const`

```
#include <type_traits>

// std::true_type and std::false_type contain a field "value"
//    set to true or false respectively

template<typename T>
struct is_const_pointer : std::false_type {}; // GENERIC template declaration

template<typename R>                          // PARTIAL specialization
struct is_const_pointer<const R*> : std::true_type {};

cout << is_const_pointer<int*>::value;        // print false, generic template
cout << is_const_pointer<const int*>::value;  // print true, partial template
cout << is_const_pointer<int* const>::value;  // print false, generic template
```

Example 3: Compare Class Templates

```cpp
#include <type_traits>

template<typename T>
struct A {};

template<typename T, typename R>
struct Compare : std::false_type {};          // GENERIC template declaration

template<typename T, typename R>
struct Compare<A<T>, A<R>> : std::true_type {}; // PARTIAL specialization

cout << Compare<int, float>::value;           // false, generic template
cout << Compare<A<int>, A<int>>::value;       // true, partial template
cout << Compare<A<int>, A<float>>::value;     // true, partial template
```

Template Class Constructor

Class template arguments don't need to be repeated if they are the default ones

```
template<typename T>
struct A {
    A(const A& x); // A(const A<T>& x);
    A f();         // A<T> f();
};
```

C++17 introduces *automatic* deduction of class template arguments for object constructor

```
template<typename T, typename R>
struct A {
    A(T x, R y) {}
};
A<int, float> a1(3, 4.0f); // < C++17
A             a2(3, 4.0f); // C++17
```

Class Template - Advanced Concepts

Given a template class and a template member function

```
template<typename T, typename R>
struct A {
    template<typename X, typename Y>
    void f();
};
```

There are two ways to specialize the class/function:

- **Generic class + generic function**
- **Full class specialization + generic/full specialization function**

```cpp
template<typename T, typename R>
template<typename X, typename Y>
void A<T, R>::f() {}
// ok, A<T, R> and f<X, Y> are not specialized

template<>
template<typename X, typename Y>
void A<int, int>::f() {}
//  ok, A<int, int> is full specialized
//  ok, f<X, Y> is not specialized

template<>
template<>
void A<int, int>::f<int, int>() {}
//  ok, A<int, int> and f<int, int> are full specialized
```

```cpp
template<typename T>
template<typename X, typename Y>
void A<T, int>::f() {}
// error A<T, int> is partially specialized
//      (A<T, int> class must be defined before)

template<typename T, typename R>
template<typename X>
void A<T, R>::f<int, X>() {}
// error function members cannot be partially specialized

template<typename T, typename R>
template<>
void A<T, R>::f<int, int>() {}
// error function members of a non-specialized class cannot be specialized
//      (requires a binding to a specific template instantiation at compile-time)
```

Structure templates can have different data members for each specialization.

The compiler needs to known in advance if a symbol within a structure is a type or a static member when the structure template *depends on* another template parameter

The keyword `typename` placed before a *structure template* solves this ambiguous

```
template<typename T>
struct A {
    using type = int;
};

template<typename R>
void g() {
    using X = typename A<R>::type; // "type" is a typename or
}                                  // a data member depending on R
```

Accessing a Dependent Type - `typename` Keyword 2/2

The `using` keyword can be used to simply the expression to get the structure type

```
template<typename T>
struct A {
    using type = int;
};

template<typename T>
using AType = typename A<T>::type;

template<typename R>
void g() {
    using X = AType<R>;
}
```

Template Dependent Names - `template` Keyword

The `template` keyword tells the compiler that what follows is a *template name* (*function* or *class*)

note: some recent compilers don't strictly require this keyword in simple cases

```
template<typename T>
struct A {
    template<typename R>
    void g() {}
};

template<typename T>   // A<T> is a dependent name (from T)
void f(A<T> a) {
//  a.g<int>(); // compile error g<int> is a dependent name (from int)
                // interpreted as: "(a.g < int) > ()"
    a.template g<int>(); // ok
}
```

Class Template Hierarchy and `using`

Member of class templates can be used *internally* in derived class templates by specifying the particular type of the base class with the keyword `using`

```
template<typename T>
struct A {
    T    x;
    void f() {}
};

template<typename T>
struct B : A<T> {
    using A<T>::x;  // needed (otherwise it could be another specialization)
    using A<T>::f;  // needed

    void g() {
        x;   // without 'using': this->x
        f();
    }
};
```

`virtual` Function and Template

Virtual functions cannot have template arguments

- **Templates** are a compile-time feature
- **Virtual functions** are a run-time feature

Full story:
The reason for the language disallowing the particular construct is that there are potentially infinite different types that could be instantiating your template member function, and that in turn means that the compiler would have to generate code to dynamically dispatch those many types, which is infeasible

`stackoverflow.com/a/79682130`

friend Keyword

```
template<typename T>              struct A {};
template<typename T, typename R> struct B {};
template<typename T>              void f() {}
//----------------------------------------------------------------------------------
class C {
    friend void f<int>();                   // match only f<int>

    template<typename T> friend void f(); // match all templates

    friend struct A<int>;                   // match only A<int>

    template<typename> friend struct A;   // match all A templates

//  template<typename T> friend struct B<int, T>;
//      partial specialization cannot be declared as a friend
};
```

Template Template Arguments

Template template parameters match *templates* instead of concrete types

```
template<typename T> struct A {};

template< template<typename> class R >
struct B {
    R<int>   x;
    R<float> y;
};
template< template<typename> class R, typename S >
void f(R<S> x) {}  // works with every class with exactly one template parameter

B<A> y;
f( A<int>() );
```

`class` and `typename` keyword are interchangeably in C++17

Template Meta-Programming

Template Meta-Programming

*"Metaprogramming is the writing of computer programs with the ability to **treat programs as their data**. It means that a program could be designed to read, generate, analyze or transform other programs, and even modify itself while running"*

*"Template meta-programming refers to uses of the C++ template system to **perform computation at compile-time** within the code. Templates meta-programming include compile-time constants, data structures, and complete functions"*

Template Meta-Programming

- **Template Meta-Programming is fast** (runtime)
  Template Metaprogramming is computed at compile-time (nothing is computed at run-time)
- **Template Meta-Programming is Turing Complete**
  Template Metaprogramming is capable of expressing all tasks that standard programming language can accomplish
- **Template Meta-Programming requires longer compile time**
  Template recursion heavily slows down the compile time, and requires much more memory than compiling standard code
- **Template Meta-Programming is complex**
  Everything is expressed recursively. Hard to read, hard to write, and also very hard to debug

Example 1: `Factorial`

```cpp
template<int N>
struct Factorial {       // GENERIC template: Recursive step
    static constexpr int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {   // FULL SPECIALIZATION: Base case
    static constexpr int value = 1;
};

constexpr int x = Factorial<5>::value; // 120
// int y = Factorial<-1>::value;       // Infinite recursion :)
```

Example 1: `Factorial` (Notes)

The previous example can be easily written as a `constexpr` in C++14

```
template<typename T>
constexpr int factorial(T value) {
    T tmp = 1;
    for (int i = 2; i <= value; i++)
        tmp *= i;
    return tmp;
};
```

Advantages:

- Easy to read and write (easy to debug)
- Faster compile time (no recursion)
- Works with different types (`typename T`)
- Works at run-time *and* compile-time

Example 2: Log2

```cpp
template<int N>
struct Log2 {     // GENERIC template: Recursive step
    static_assert(N > 0, "N must be greater than zero");

    static constexpr int value = 1 + Log2<N / 2>::value;
};

template<>
struct Log2<1> { // FULL SPECIALIZATION: Base case
    static constexpr int value = 0;
};

constexpr int x = Log2<20>::value; // 4
```

Example 3: Log

```cpp
template<int A, int B>
struct Max { // utility
    static constexpr int value = A > B ? A : B;
};

template<int N, int BASE>
struct Log {      // GENERIC template: Recursive step
    static_assert(N > 0,    "N must be greater than zero");
    static_assert(BASE > 0, "BASE must be greater than zero");
                                // Max is used to avoid Log<0, BASE>
    static constexpr int TMP   = Max<1, N / BASE>::value;
    static constexpr int value = 1 + Log<TMP, BASE>::value;
};
template<int BASE>
struct Log<1, BASE> {   // PARTIAL SPECIALIZATION: Base case
    static constexpr int value = 0;
};

constexpr int x = Log<20, 2>::value; // 4
```

Example 4: `Unroll` (Compile-time/Run-time Mix) ★

```
template<int NUM_UNROLL, int STEP = 0>
struct Unroll {             // GENERIC template: Recursive step
    template<typename Op>
    static void run(Op op) {
        op(STEP);
        Unroll<NUM_UNROLL, STEP + 1>::run(op);
    }
};

template<int NUM_UNROLL>
struct Unroll<NUM_UNROLL, NUM_UNROLL> { // PARTIAL SPECIALIZATION: Base case
    template<typename Op>
    static void run(Op) {}
};

auto lambda = [](int step) { cout << step << ", "; };
Unroll<5>::run(lambda);  // print "0, 1, 2, 3, 4"
```

SFINAE:
Substitution Failure
Is Not An Error

SFINAE

Substitution Failure Is Not An Error (SFINAE) applies during overload resolution of function templates. When substituting the deduced type for the template parameter fails, the specialization is discarded from the overload set *instead* of causing a compile error

The Problem

```
template<typename T>
T ceil_div(T value, T div);

template<>
unsigned ceil_div<unsigned>(unsigned value, unsigned div) {
    return (value + div - 1) / div;
}

template<>
int ceil_div<int>(int value, int div) { // handle negative values
    return (value > 0) ^ (div > 0) ?
           (value / div) : (value + div - 1) / div;
}
```

What about `long long int`, `long long unsigned`, `short`, `unsigned short`, etc.?

`std::enable_if` Type Trait

The common way to adopt `SFINAE` is using the `std::enable_if/std::enable_if_t` type traits

`std::enable_if` allows a function template or a class template specialization to include or exclude itself from a set of matching functions/classes

```
template<bool Condition, typename T = void>
struct enable_if {
//  "type" is not defined if "Condition == false"
};
template<typename T>
struct enable_if<true, T> {
    using type = T;
};
```

helper alias: `std::enable_if_t<T>` instead of `typename std::enable_if<T>::type`

```cpp
#include <type_traits> // std::is_signed_v, std::enable_if_t

template<typename T>
std::enable_if_t<std::is_signed_v<T>>
f(T) {
    cout << "signed";
}

template<typename T>
std::enable_if_t<!std::is_signed_v<T>>
f(T) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

```cpp
#include <type_traits>

template<typename T>
void f(std::enable_if_t<std::is_signed_v<T>, T>) {
    cout << "signed";
}

template<typename T>
void f(std::enable_if_t<!std::is_signed_v<T>, T>) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

Function SFINAE - Hidden Parameter


```
#include <type_traits>

template<typename T>
void f(T,
       std::enable_if_t<std::is_signed_v<T>, int> = 0) {
    cout << "signed";
}

template<typename T>
void f(T,
       std::enable_if_t<!std::is_signed_v<T>, int> = 0) {
    cout << "unsigned";
}

f(1);  // print "signed"
f(1u); // print "unsigned"
```

Function SFINAE - Hidden Template Parameter


```cpp
#include <type_traits>

template<typename T,
         std::enable_if_t<std::is_signed_v<T>, int> = 0>
void f(T) {}

template<typename T,
         std::enable_if_t<!std::is_signed_v<T>, int> = 0>
void f(T) {}

f(4);
f(4u);
```

Function SFINAE - decltype + return type


```
#include <type_traits>
template<typename T, typename R>          // (1)
decltype(T{} + R{}) add(T a, R b) {      // T{} + R{} is not possible with 'A'
    return a + b;
}

template<typename T, typename R>          // (2)
std::enable_if_t<std::is_class_v<T>, T> // 'int' is not a class
add(T a, R b) {
    return a;
}

struct A {};

add(1, 2u);     // call (1)
add(A{}, A{}); // call (2)
// if 'A' supports operator+, then we have a conflict
```

Function SFINAE Example - Array vs. Pointer

```cpp
#include <type_traits>

template<typename T, int Size>
void f(T (&array)[Size]) {} // (1)

//template<typename T, int Size>
//void f(T* array) {}         // (2)

template<typename T>
std::enable_if_t<std::is_pointer_v<T>>
f(T ptr) {}                   // (3)

int array[3];
f(array); // It is not possible to call (1) if (2) is present
          // The reason is that 'array' decays to a pointer
          // Now with (3), the code calls (1)
```

Class SFINAE

```cpp
#include <type_traits>

template<typename T, typename Enable = void>
struct A;

template<typename T>
struct A<T, std::enable_if_t<std::is_signed_v<T>>> {
};

template<typename T>
struct A<T, std::enable_if_t<!std::is_signed_v<T>>> {
};

A<int>;
A<unsigned>;
```

Class + Function SFINAE ★

```cpp
#include <type_traits>

template<typename T>
class A {
//  this does not work because T depends on A, not on h
//  void h(T,
//         std::enable_if_t<std::is_signed_v<T>, int> = 0) {
//     cout << "signed";
//  }
    template<typename R = T> // now R dependes on h
    void h(R,
           std::enable_if_t<std::is_signed_v<R>, int> = 0) {
       cout << "signed";
    }
};

A<int>;
```

SFINAE can be also used to check if a structure has a specific data member or type

Let consider the following structures:

```
struct A {
    static int x;
    int        y;
    using type = int;
};

struct B {};
```

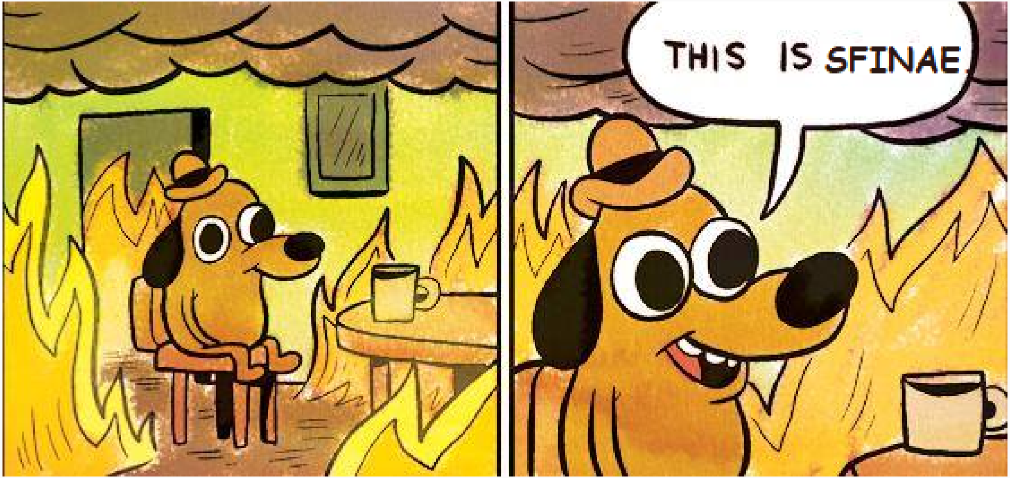
```cpp
#include <type_traits>
template<typename T, typename = void>
struct has_x : std::false_type {};

template<typename T>
struct has_x<T, decltype((void) T::x)> : std::true_type {};

template<typename T, typename = void>
struct has_y : std::false_type {};

template<typename T>
struct has_y<T, decltype((void) std::declval<T>().y)> : std::true_type {};

has_x< A >::value; // returns true
has_x< B >::value; // returns false
has_y< A >::value; // returns true
has_y< B >::value; // returns false
```

```cpp
template<typename...>
using void_t = void;  // included in C++17 <utility>

template<typename T, typename = void>
struct has_type : std::false_type {};

template<typename T>
struct has_type<T,
                std::void_t<typename T::type> > : std::true_type {};

has_type< A >::value; // returns true
has_type< B >::value; // returns false
```

Support Trait for Stream Operator ★

```
template<typename T>
using EnableP = decltype( std::declval<std::ostream&>() <<
                          std::declval<T>() );

template<typename T, typename = void>
struct is_stream_supported : std::false_type {};

template<typename T>
struct is_stream_supported<T, EnableP<T>> : std::true_type {};

struct A {};

is_stream_supported<int>::value; // returns true
is_stream_supported<A>::value;   // returns false
```

SFINAE



https://twitter.com/IAmErikN/status/1252316405724336128

Variadic Templates

Variadic Template

Variadic template (C++11)

Variadic templates, also called *template parameter pack*, are templates that take a *variable number* of arguments of any type

```
template<typename... TArgs> // Variadic typename
void f(TArgs... args) {      // Typename expansion
    args...;                 // Arguments expansion
}
```

Note: variadic arguments must be the last one in the declaration

The number of variadic arguments can be retrieved with the `sizeof...` operator

```
sizeof...(args)
```

Variadic Template - Example

```cpp
// BASE CASE
template<typename T, typename R>
auto add(T a, R b) {
    return a + b;
}

// RECURSIVE CASE
template<typename T, typename... TArgs>  // Variadic typename
auto add(T a, TArgs... args) {           // Typename expansion
    return a + add(args...);             // Arguments expansion
}

add(2, 3.0);        // 5
add(2, 3.0, 4);     // 9
add(2, 3.0, 4, 5);  // 14
// add(2);          // compile error the base case accepts only two arguments
```

Variadic Template - Parameter Types

```
template<typename... TArgs>
void f(TArgs... args) {}          // pass by-value

template<typename... TArgs>
void g(const TArgs&... args) {}  // pass by-const reference

template<typename... TArgs>
void h(TArgs*... args) {}         // pass by-pointer

int* a, *b;
f(1, 2.0);
h(a, b);
```

Variadic Template - Function Application

```
template<typename T>
T square(T value) { return value * value; }

template<typename T, typename R>
auto add(T a, R b) { return a + b; }    // BASE case

template<typename T, typename... TArgs> // RECURSIVE case
auto add(T a, TArgs... args) {
    return a + add(args...);
}
//------------------------------------------------------------
template<typename... TArgs>
auto add_square(TArgs... args) {
    return add(square(args)...); // square() is applied to each
}                                // variadic argument

add_square(2, 2, 3.0f); // returns 17.0f
```

Variadic Template - Arguments to Array

```cpp
template<typename... TArgs>
void f(TArgs... args) {
    constexpr int Size    = sizeof...(args);
    int           array[] = {args...};
    for (auto x : array)
        cout << x << " ";
}

f(1, 2, 3);    // print "1 2 3"
f(1, 2, 3, 4); // print "1 2 3 4"
```

C++17 **Folding expressions** perform a *fold* of a template parameter pack over a *binary* operator

Unary/Binary folding

```
template<typename... Args>
auto add_unary(Args... args) {  // Unary folding
    return (... + args);         // unfold: 1 + 2.0f + 3ull
}

template<typename... Args>
auto add_binary(Args... args) { // Binary folding
    return (1 + ... + args);     // unfold: 1 + 1 + 2.0f + 3ull
}

add_unary(1, 2.0f, 3ll);  // returns 6.0f (float)
add_binary(1, 2.0f, 3ll); // returns 7.0f (float)
```

Folding Expression

Same example of “Variadic Template - Function Application” ... but shorter

```
template<typename T>
T square(T value) { return value * value; }

template<typename... TArgs>
auto add_square(TArgs... args) {
    return (square(args) + ...); // square() is applied to each
}                                 // variadic argument

add_square(2, 2, 3.0f); // returns 17.0f
```

Variadic Template and Classes

```
template<int... NArgs>
struct Add;                   // data structure declaration

template<int N1, int N2>
struct Add<N1, N2> {          // BASE case
    static constexpr int value = N1 + N2;
};

template<int N1, int... NArgs>
struct Add<N1, NArgs...> {  // RECURSIVE case
    static constexpr int value = N1 + Add<NArgs...>::value;
};

Add<2, 3, 4>::value; // returns 9
// Add<>;             // compile error no match
// Add<2>::value;     // compile error
                      // call Add<N1, NArgs...>, then Add<>
```

Variadic Class Template ★

Variadic Template can be used to build recursive data structures

```
template<typename... TArgs>
struct Tuple;       // data structure declaration

template<typename T>
struct Tuple<T> {  // base case
    T value;        // specialization with one parameter
};

template<typename T, typename... TArgs>
struct Tuple<T, TArgs...> {  // recursive case
    T                value;   // specialization with more
    Tuple<TArgs...> tail;     // than one parameter
};

Tuple<int, float, char> t1 { 2, 2.0, 'a' };
t1.value;           // 2
t1.tail.value;      // 2.0
t1.tail.tail.value; // 'a'
```

Get function arity at compile-time:

```
template <typename T>
struct GetArity;

// generic function pointer
template<typename R, typename... Args>
struct GetArity<R(*)(Args...)> {
    static constexpr int value = sizeof...(Args);
};
// generic function reference
template<typename R, typename... Args>
struct GetArity<R(&)(Args...)> {
    static constexpr int value = sizeof...(Args);
};
// generic function object
template<typename R, typename... Args>
struct GetArity<R(Args...)> {
    static constexpr int value = sizeof...(Args);
};
```

```cpp
void f(int, char, double) {}

int main() {
    // function object
    GetArity<decltype(f)>::value;

    auto& g = f;
    // function reference
    GetArity<decltype(g)>::value;

    // function reference
    GetArity<decltype((f))>::value;

    auto* h = f;
    // function pointer
    GetArity<decltype(h)>::value;
}
```

stackoverflow.com/a/27867127

Get `operator()` (and `lambda`) arity at compile-time:

```
template <typename T>
struct GetArity;

template<typename R, typename C, typename... Args>
struct GetArity<R(C::*)(Args...)> {          // class member
    static constexpr int value = sizeof...(Args);
};
template<typename R, typename C, typename... Args>
struct GetArity<R(C::*)(Args...) const> {  // "const" class member
    static constexpr int value = sizeof...(Args);
};

struct A {
    void operator()(char, char) {}
    void operator()(char, char) const {}
};

GetArity<A>::value;       // call GetArity<R(C::*)(Args...)>
GetArity<const A>::value; // call GetArity<R(C::*)(Args...) const>
```

C++20 Concepts

C++20 Concepts

C++20 introduces **concepts** as an extension for *templates* to enforce *constraints*, which specifies the *requirements* on template arguments

Concepts allows to perform compile-time validation of template arguments

Advantages compared to SFINAE (`std::enable_if`):

- Concepts are easier to read and write
- Clear compile-time messages for debugging
- Faster compile time

Keyword:

`concept` Constrain

`requires` Constrain list/Requirements, *clause* and *expression*

- The concept behind C++ concepts
- Constraints and concepts
- What are C++20 concepts and constraints? How to use them?

The Problem

Goal: define a function to sum only arithmetic types

```
template<typename T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
struct A {};

add(3, 4);          // ok
// add(A{}, A{}); // not supported
```

SFINAE solution (ugly, verbose):

```
template<typename T>
std::enable_if_t<T, std::is_arithmetic_v<T>>
add(T valueA, T valueB) {
    return valueA + valueB;
}
```

`concept` Keyword

```
[template arguments]
concept [name] = [compile-time boolean expression];
```

Example: arithmetic type concept

```
template<typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

- *Template argument constrain*

```
template<Arithmetic T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
```

- `auto` *deduction constrain* (*constrained* `auto`)

```
auto add(Arithmetic auto valueA, Arithmetic auto valueB) {
    return valueA + valueB;
}
```

`requires` Clause

```
requires [compile-time boolean expression or Concept]
```

it acts like `SFINAE`

- After *template parameter list*

```cpp
template<typename T>
requires Arithmetic<T>
T add(T valueA, T valueB) {
    return valueA + valueB;
}
```

- After *function declaration*

```cpp
template<typename T>
T add(T valueA, T valueB) requires (sizeof(T) == 4) {
    return valueA + valueB;
}
```

`requires` Clause and `concept` Notes

Concepts and *requirements* can have *multiple* statements. It must be a *primary expression*, e.g. `constexpr` value (not a `constexpr` function) or a sequence of *primary expressions* joined with the operator `&&` or `||`

```
template<typename T>
concept Arithmetic2 = std::is_arithmetic_v<T> && sizeof(T) >= 4;
```

Concepts and *requirements* can be used together

```
template<Arithmetic T>
requires (sizeof(T) >= 4)
T add(T valueA, T valueB) {
```

`requires` Expression

A `requires` **expression** is a *compile-time* expression of type `bool` that defines the **constraints** on template arguments

```
requires [(arguments)] {
    [SFINAE contrain];    // or
    requires [predicate];
} -> bool
```

```
template<typename T>
concept MyConcept = requires (T a, T b) { // First case: SFINAE constrains
    a + b;           // Req. 1 - support add operator
    a[0];            // Req. 2 - support subscript operator
    a.x;             // Req. 3 - has "x" data member
    a.f();           // Req. 4 - has "f" function member
    typename T::type; // Req. 5 - has "type" field
};
```

Concept library

```
#include <concept>

template<typename T>
concept MyConcept2 = requires (T a, T b) {
    {*a + 1} -> std::convertible_to<float>; // Req. 6 - can be deferred and the sum
                                            // with an integer is convertible
                                            // to float
    {a * a}  -> std::same_as<int>;          // Req. 7 - "a * a" must be valid and
                                            // the result type is "int"
};
```

`requires` Expression + Clause

`requires` **expression** can be combined with `requires` **clause** (see `requires` definition, second case) to compute a boolean value starting from SFINAE expressions

```
template<typename T>
concept Arithmetic = requires {       // expression -> bool (zero args)
    T::value;                         // clause     -> direct SFINAE
    requires std::is_arithmetic_v<T>; // clause     -> SFINAE from boolean
};
```

```
template<typename T>
concept MyConcept = requires (T value) {  // expression -> bool  (one arg)
    requires sizeof(value) >= 4;          // clause     -> SFINAE from boolean
    requires std::is_floating_point_v<T>; // clause     -> SFINAE from boolean
};
```

`requires` Clause + Expression

`requires` **clause** can be combined with `requires` **expression** to apply SFINAE (functions, structures) starting from a compile-time *boolean expressions*

```
template<typename T>
void f(T a) requires requires { T::value; }
//          clause -> SFINAE followed by
//                   expression -> bool (zero args)
{}
```

```
template<typename T>
T increment(T a) requires requires (T x) { x + 1; }
//               clause -> SFINAE followed by
//                        expression -> bool (one arg)
{
   return a + 1;
}
```

requires and constexpr

Some examples:

- ```
  constexpr bool has_member_x = requires(T v){ v.x; };
  ```
- ```
  if constexpr (MyConcept<T>)
  ```
- ```
  static_assert(requires(T v){ ++v; }, "no increment");
  ```
- ```
  template<typename Iter>
  constexpr bool is_iterator() {
      return requires(Iter it) { *it++; };
  }
  ```

Nested `requires`

Nested `requires` example:

```
requires(Iter v) { // expression -> bool (one arg)
    Iter it;
    requires requires(typename Iter::value_type v) {
//  clause -> SFINAE followed by
//               expression -> bool (one arg)
        v   = *it; // read
        *it = v;   // write
    };
}
```