Modern C++ Programming

9. C++ Templates and Meta-programming II

Federico Busato

University of Verona, Dept. of Computer Science

2021, v3.06

Table of Context

Table of Context

Class Template

Class Template

In a similar way to function templates, **class templates** are used to build a family of classes

```
template<typename T>
struct A { // templated class (typename template)
    T x = 0;
};

template<int N1>
struct B { // templated class (numeric template)
    int N = N1;
};
A<int>    a1; // a1.x is int    = 0
A<float>  a2; // a2.x is float  = 0.0f
A<double> a4; // a3.x is double = 0.0
B<1>      b1; // b1.N is 1
B<2>      b2; // b2.N is 2
```

Template Class Constructor

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
```

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
    T x;
};

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

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

```
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
// a1.y; // compile error

A<float, int> a2;
a2.y;    // ok
// 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
cout << is_same<float, float>::value; // print true
```

Example 2: Check if a Pointer is const

```
#include <type_traits>

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

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

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

cout << is_const_pointer<int*>::value;       // print false
cout << is_const_pointer<const int*>::value; // print true
cout << is_const_pointer<int* const>::value; // print false
```

Example 3: Compare Class Templates

```cpp
#include <type_traits>

template<typename T>
struct A {};

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

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

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

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**

```
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
```

```
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 declared 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 unspecialized class cannot
//      be specialized
```

`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`

Class Template Hierarchy

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 (may be also a specialization)
    using A<T>::f;  // needed

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

`friend` Keyword

```
template<typename T>              struct A {};
template<typename T, typename R> struct B {};
template<typename T>              void f() {}
//-------------------------------------------------------------

class C {
    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

    friend void f<int>();                 // match only f<int>

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

Template Dependent Names - `template` Keyword)

The `template` keyword tells the compiler that what follows is a *function template*, and not a member data

This is important when there are two (or more) *dependent names*

```
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
}
```

Template Template Arguments

Template template parameters match *templates* instead of concrete types

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

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 and type

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

Template Variable

C++14 allows the creation of variables that are templated

Template variable can be considered a special case of template class

```
template<typename T>
constexpr T pi{ 3.1415926535897932385 }; // variable template

template<typename T>
T circular_area(T r) {
    return pi<T> * r * r; // pi<T> is a variable template
}                         // instantiation

circular_area(3.3f); // float
circular_area(3.3);  // double
// circular_area(3); // compile error
                     // narrowing conversion on "pi"
```

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 {      // specialization: recursive step
    static constexpr int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {   // 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

```
template<int N>
struct Log2 {
    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

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

template<int N, int BASE>
struct Log {      // specialization: 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) ★

```cpp
template<int NUM_UNROLL, int STEP = 0>
struct Unroll {                          // 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> { // base case
    template<typename Op>                // (specialization)
    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);

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

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>

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"
```

```cpp
#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"
```

```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);
```

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

template<typename T, typename R>
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);    // return 3u
add(A{}, A{}); // add() not supported
```

Function SFINAE Example

Array vs. Pointer:

```cpp
#include <type_traits>

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

template<typename T>
std::enable_if_t<std::is_pointer_v<T>>
f(T array) {} // (2)

int* ptr;
int  array[3];
f(ptr);   // call (2)
f(array); // call (1)
          // without std::is_pointer_v always calls (2)
```

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 {};
```

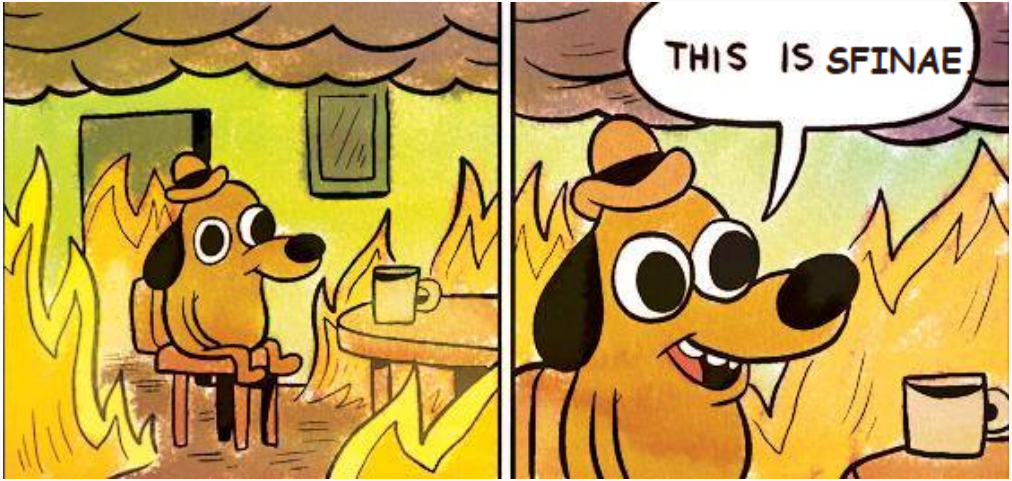
```cpp
#include <type_traits>
template<typename T, typename = void>
struct has_x : std::false_type {};

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::R> > : std::true_type {};

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

Support Trait for Stream Operator ★

```cpp
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 parameter 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...);             // parameters 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 two parameters
```

Variadic Template - Parameter Types

```
template<typename... TArgs>
void f(TArgs... args) {}          // generic

template<typename... TArgs>
void g(const TArgs&... args) {}  // force "const references"

template<typename... TArgs>
void h(TArgs*... args) {}         // force "pointers"

// list of "pointers" followed by a list of "const references"
template<typename... TArgs1, typename... TArgs2>
void f2(const TArgs1*... args, const TArgs2& ...va) {}

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

Variadic Template - Function Application

```cpp
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
}                                // the variadic arguments

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

Variadic Template - Arguments to Array

```
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

```cpp
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)
```

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
}                                // the variadic arguments

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