Modern C++ Programming

6. C++ Templates and Meta-programming II

Federico Busato

University of Verona, Dept. of Computer Science

2018, v1.0

Agenda

- **Class Templates**
  - Full/Partial specialization
  - Declaration and definition
  - `virtual`, `members`, `friend`
  - `template` keyword
  - Template template arguments
  - Template variable
- **Template Meta-Programming**
  - Factorial
  - Log
  - Unroll
- **SFINAE**
  - Function implementation
  - Class implementation
- **Variadic Template**
  - Parameter recursion
  - `sizeof...`
  - Meta-Programming
  - Specialization
- **STD Template Utility**
  - `std::pair`
  - `std::tuple`

Class Templates

Class Templates

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

- The main difference with template functions is that classes can be **partially** specialized
- 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>
struct A {
    T x;
    void f(T y) {}
};

int main() {
    A<int>   a;
    A<float> b;
}
```

Class Templates (example)

```
template<typename T>
class A {
public:
    void g() {}
};

template<typename T, int N>
struct B : public A<T> {
    T x = N;

    template<typename R>
    void f(T y, R z) {}
};

int main() {
    B<int, 3> b;
    b.f(3, 3.3);
}
```

Class Template Full/Partial Specialization

```
template<typename T, typename R = float>
struct A {
    int x;
    void A() { std::cout << "<T, R>"; }
};

template<>
struct A<int, float> {    // template FULL specialization
    void A() { std::cout << "<int, float>"; }
};    // "x" is not inherited

template<typename R>
struct A<int, R> {    // template PARTIAL specialization
    void A() { std::cout << "<int, R>"; }
};   // "x" is not inherited

int main() {
    A<float> a;          // print "<T, R>"
    A<float, float> a;   // print "<T, R>"
    A<int, float>   b;   // print "<int, float>"
    A<int, char>    c;   // print "<int, R>"
}
```

Example 1: Implement a Simple Type Trait

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
struct is_const_pointer : std::false_type {}
// equivalent to
// template<typename T>
// struct is_const_pointer {
//     static constexpr bool value = false;
// };

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

int main() {
    std::cout << std::is_const<int*>::value;          // print false
    std::cout << std::is_const<const int*>::value;    // print false
    std::cout << std::is_const<int* const>::value;    // print true
    std::cout << is_const_pointer<int*>::value;       // print false
    std::cout << is_const_pointer<const int*>::value; // print true
    std::cout << is_const_pointer<int* const>::value; // print false
}
```

Example 2: Compare Class Templates

```cpp
#include <iostream>

template<typename T>
struct A {};

template<typename T>
struct B {};

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

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

int main() {
    std::cout << Compare<int, float>::value;       // false
    std::cout << Compare<int, int>::value;         // false
    std::cout << Compare<B<int>, B<int>>::value;   // false
    std::cout << Compare<A<int>, B<int>>::value;   // false
    std::cout << Compare<A<int>, A<float>>::value; // true
}
```

Separate Declaration and Definition

```
template<typename T>
struct A {
    using MyType = T;

    static const T a;

    MyType f();

    template<typename R>
    void g();
};

template<typename T>
const T A<T>::a = 3;

template<typename T>
typename A::MyType A<T>::f() {}

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

Separate Declaration and Definition + Partial Specialization

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

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

template<>
template<typename X, typename Y>
void A<int, int>::f() {}
//  ok, A<int, int> is full specialized
```

```cpp
// template<typename T, typename R>
// template<typename X>
// void A<T, R>::f<int, X>() {}
//!   functions cannot be partially
//!   specialized

// template<typename T, typename R>
// template<>
// void A<T, R>::f<int, int>() {}
//!   function members of a
//!   unspecialized class cannot
//!   be specialized

int main() {
    A<int, int> a;
    a.f<int, int>();
}
```

Notes about Classes and Templates (`virtual` Functions)

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`

Notes about Classes and Templates (Members)

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
    using A<T>::f;  // needed

    void g() {
        x;
        f();
    }
};
```

Notes about Classes and Templates (`friend` keyword)

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

template<typename T>
struct B {};

template<typename T, typename R>
struct C {};

class D {
    friend class A<int>;                // match only A<int>
    template<typename> friend class B; // match all B templates
//  template<typename T> friend class C<int, T>;
//    partial specialization cannot be declared as a friend
};
```

Notes about Classes and Templates (`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) {       // (g<int> is a dependent name (from int)
//  a.g<int>();        // compile error
//    interpreted as: "(a.g < int) > ()"
    a.template g<int>(); // ok
}
```

Template Template Arguments

Template template parameters match *templates* instead of concrete types

```
template<template <typename> class R>
struct B {
    R<int>   x;
    R<float> y;
};

template<typename T>
struct A {};

template<template <typename> class R, typename S>
void f(R<S> x) {}  // works with every class with
                   // one template paramenter
int main() {
    f( A<int>() );
    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 = T(3.1415926535897932385L); // variable template

// equivalent to
// template<typename T>
// struct PI {
//     static const T value = T(3.1415926535897932385L);
// };

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

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, analyse or transform other programs, and even modify itself while running"*

Wikipedia

*"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"*

Wikipedia

Template Meta-Programming

- **Template Meta-Programming is fast**
  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

```
template <int N>
struct Factorial {      // specialization: recursive step
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {   // specialization: base case
    static const int value = 1;
};

int main() {
    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
- Faster compile-time (no recursion)
- Works with different types (`T`)

Example 2: Log

```
template <int A, int B>
struct Max {
    static const 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");

    static const int TMP   = Max<1, N / BASE>::value;
    static const int value = 1 + Log<TMP, BASE>::value;
};

template <int BASE>
struct Log<1, BASE> {   // partial specialization: base case
    static const int value = 0;
};

int main() {
    int x = Log<20, 2>::value; // 4
}
```

Example 3: `Unroll` (Compile-time/Run-time Mix)

```cpp
template<int MAX_VALUE, int STEP = 0>
struct Unroll {                         // recursive step
    template<typename Op>
    static void run(Op op) {
        op(STEP);
        Unroll<MAX_VALUE, STEP + 1>::run(op);
    }
};
template<int MAX_VALUE>
struct Unroll<MAX_VALUE, MAX_VALUE> { // base case (specialization)
    template<typename Op>
    static void run(Op) {}
};

struct MyOp {
    void operator()(int step) {
        std::cout << step << ", ";
    }
};
int main() {
    Unroll<5>::run( MyOp() );  // 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) {
    return (value > 0) ^ (div > 0) ?
        (value / div) : (value + div - 1) / div;
}
// what about "char", "unsigned char", "short", etc.?
```

`std::enable_if` Type Trait

The most common way to adopt SFINAE is using the `std::enable_if` 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 B, class T = void>
struct enable_if {
//  "type" is not defined
};

template<class T>
struct enable_if<true, T> {
    using type = T;
};
```

Function SFINAE (return type)

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_signed<T>::value>::type
f(T) {
    std::cout << "signed";
}

template<typename T>
typename std::enable_if<!std::is_signed<T>::value>::type
f(T) {
    std::cout << "unsigned";
}

int main() {
    f(1);  // print "signed"
    f(1u); // print "unsigned"
}
```

Function SFINAE (parameter)

```
#include <iostream>
#include <type_traits>

template<typename T>
void g(typename std::enable_if<std::is_signed<T>::value, T>::type) {
    std::cout << "signed";
}

template<typename T>
void g(typename std::enable_if<!std::is_signed<T>::value, T>::type) {
    std::cout << "unsigned";
}

int main() {
    h<int>(1);      // print "signed"
    h<unsigned>(1); // print "unsigned"
}
```

Function SFINAE (hidden parameter)

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
void h(T,
       typename std::enable_if<std::is_signed<T>::value, T>::type = 0) {
    std::cout << "signed";
}

template<typename T>
void h(T,
       typename std::enable_if<!std::is_signed<T>::value, T>::type = 0) {
    std::cout << "unsigned";
}

int main() {
    h(1);  // print "signed"
    h(1u); // print "unsigned"
}
```

Function SFINAE (`decltype` + return type)

```cpp
#include <type_traits>
#include <utility>

template<typename T>
decltype(std::declval<T>() + std::declval<T>())
add(T a, T b) {
    return a + b;
}

template<typename T>
typename std::enable_if<std::is_class<T>::value, T>::type
add(T a, T b) {
    return a;
}

struct A {};

int main() {
    add(1, 2);
    add(A(), A());
}
```

Function SFINAE (check type member)

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

template<typename T, typename = T::type>
struct CheckType {}

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

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

int main() {
    CheckType< A<char> > x;     // ok
//  CheckType< A<int> >  y;     // compile error
    has_type< A<char> >::value; // returns true
    has_type< A<int> >::value;  // returns false
}
```

Class SFINAE

```cpp
#include <type_traits>

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

template <typename T>
struct A<T, typename std::enable_if<std::is_signed<T>::value>::type>
{};

template <typename T>
struct A<T, typename std::enable_if<!std::is_signed<T>::value>::type>
{};
```

Stream Support Trait

```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 {};

int main() {
    is_stream_supported<int>::value; // returns true
    is_stream_supported<A>::value;   // returns false
}
```

Variadic Templates

Variadic Templates

C++11

Variadic templates

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

C++11 allows template definitions to take an arbitrary number of arguments of any type

```
template<typename... TArgs> // variadic typename
void f(TArgs... args) {     // typename expansion
    args...;                // arguments expansion
}
```

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 Templates

```
template<typename T, typename R>
auto add(T a, R b) {                          // base case
    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
}

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

int main() {
    add(2, 3.0);       // 5
    add(2, 3.0, 4);    // 9
    add(2, 3.0, 4, 5); // 14
//  add(2);            // compile error
    f(2);              // ok, all parameters are variadic
    f();               // ok, all parameters are variadic
}
```

Variadic Template Parameters

```cpp
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 "pointer"

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

int main() {
    f(1, 2.0);
    g(1, 2.0);
    int* a, *b;
    h(a, b);
    f2(a, b, 3);
}
```

Variadic Template Parameters Recursion

```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; }

template<typename T, typename... TArgs>
auto add(T a, TArgs... args) {
    return a + add(args...);
}

template<typename... TArgs>
int add_square(TArgs... args) {
    return add(square(args)...); // square is applied to
}                                // variadic arguments

int main() {
    add_square(2, 2, 3); // returns 17
}
```

Variadic Template sizeof and sizeof...

```cpp
template<typename... TArgs>
int count(TArgs... args) {  // count number of arguments
    return sizeof...(args);
}

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

template<typename T, typename... TArgs>
auto add(T a, TArgs... args) {
    return a + add(args...);
}

template<typename... TArgs>
int f(TArgs... args) {    // get the sum of argument sizes
    return add(sizeof(args)...);
}

int main() {
    count(2, 2.0, 'a'); // returns 3
    f(2, 2.0, 'a');     // returns 4 + 8 + 1 = 13
}
```

Class Variadic 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
};

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

Variadic Template and Meta-Programming

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

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

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

int main() {
    Add<2, 3, 4>::value; // returns 9
//  Add<2>::value;       // compile error Add<"empty">
}
```

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);
};
```

```
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;
}
```

Full Story:
`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 const int value = sizeof...(Args);
};

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

struct A {
    void operator()(char, char) {}
    void operator()(char, char) const {}
};
int main() {
    GetArity<A>::value;       // call GetArity<R(C::*)(Args...)>
    GetArity<const A>::value; // call GetArity<R(C::*)(Args...) const>
}
```

Variadic Template and Macro Trick

Variadic macro can be used to simplify class definition

```
template<typename type1, typename type2, typename type3>
struct MyClass {
    A<int,type1> f();
    A<int,type2> g();
};

#define MYCLASS(...)                                        \
template<typename type1, typename type2, typename type3>    \
__VA_ARGS__ MyClass<type1, type2, type3>

template<typename type1, typename type2, typename type3>
A<int, type1> MyClass<type1, type2, type3>:f() {
    ...
}

MYCLASS(A<int, type1>)::g() {
    ...
}
```

STD Template Classes

std::pair 1/2

```
#include <utility>
```

std::pair class couples together a pair of values, which may be of different types

Construct a std::pair

- std::pair<T1, T2> pair(value1, value2)
- std::pair<T1, T2> pair = {value1, value2}
- auto pair = std::make_pair(value1, value2)

Data members:

- first access first field
- second access second field

Methods:

- comparison ==, <, >, ≥, ≤
- swap std::swap

```cpp
#include <utility>
#include <iostream>

int main() {
    std::pair<int, std::string> pair1(3, "abc");
    std::pair<int, std::string> pair2 = { 4, "zzz" };
    auto pair3 = std::make_pair(3, "hgt");

    std::cout << pair1.first;  // print 3
    std::cout << pair1.second; // print "abc"

    std::swap(pair1, pair2);
    std::cout << pair2.first;  // print "zzz"
    std::cout << pair2.second; // print 4

    std::cout << (pair1 > pair2); // print 1
}
```

`#include <tuple>`

`std::tuple` is a fixed-size collection of heterogeneous values. It is a generalization of `std::pair`. It allows <u>any</u> number of values

Construct a `std::tuple` (of size 3)

- `std::tuple<T1, T2, T3> tuple(value1, value2, value3)`
- `std::tuple<T1, T2, T3> tuple = {value1, value2, value3}`
- `auto tuple = std::make_tuple(value1, value2, value3)`

Data members:

`std:get<I>(tuple)` returns the *i*-th value of the tuple

Methods:

- comparison `==, <, >, ≥, ≤`
- swap `std::swap`

std::tuple

Utility methods:

- `auto t3 = std::tuple_cat(t1, t2)`
  concatenate two tuples
- `const int size = std::tuple_size<TupleT>::value`
  returns the number of elements in a tuple at compile-time
- `using T = typename std::tuple_element<TupleT>::type`
  obtains the type of the specified element
- `std::tie(value1, value2, value3) = tuple`
  creates a tuple of references to its arguments
- `std::ignore`
  an object of unspecified type such that any value can be assigned to it with no effect

```cpp
#include <tuple>
#include <iostream>
std::tuple<int, float, char> f() { return {7, 0.1f, 'a'}; }

int main() {
    std::tuple<int, char, float> tuple1(3, 'c', 2.2f);
    std::tuple<int, char, float> tuple2 = {2, 'd', 1.5f};
    auto tuple3 = std::make_tuple(3, 'c', 2.2f);

    std::cout << std::get<0>(tuple1); // print 3
    std::cout << std::get<1>(tuple1); // print 'c'
    std::cout << std::get<2>(tuple1); // print 2.2f
    std::cout << (tuple1 > tuple2);   // print 1

    auto concat = std::tuple_cat(tuple1, tuple2);
    std::cout << std::tuple_size<decltype(concat)>::value; // print 6

    using T = std::tuple_element<4, decltype(concat)>::type; // T is int

    int value1; float value2;
    std::tie(value1, value2, std::ignore) = f();
}
```