Modern C++ Programming

6. C++ Templates and Meta-programming I

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0

Agenda

- **Function Templates**
  - Template parameters
  - Default parameters
  - Template specialization
  - Template overloading
- **Type Deduction**
  - Pass-by-Reference
  - Pass-by-Pointer
  - Pass-by-Value
  - Array type deduction
- **Compile-Time Utilities**
  - `static_assert`
  - `decltype`
  - `declval`
  - `using`
- **Type Traits**
  - Type trait library
  - Type manipulation
  - Type Relation and Transformation
- **Template Parameters**

C++ Function Templates

The problem: We want to define a function to handle different types

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}

char   add(char a, char b)     { ... }
ClassX add(ClassX a, ClassX b) { ... }
```

- Redundant code!!
- How many functions we have to write!?
- If the user introduces a new type we have to write another function!!

Definition (Function Templates)

Function templates are special functions that can operate with *generic* types (independent of any particular type)

Allow to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type

```
template<typename T>
T addX(T a, T b) {
    return a + b;
}

int main() {
    int   c1 = addX(3, 4);              // c1 = 7
    float c2 = addX(3.0f, 4.0f);        // c2 = 7.0f
    int   c3 = addX<int>(3.0f, 4.0f); // c3 = 7 (int forced)
}
```

Function Templates (Benefits and Drawbacks)

Benefits

- **Generic Programming**. Code less redundant and better maintainability
- **Performance**. Computation can be done at compile-time

Drawbacks

- **Readability**. With respect to C++, the syntax and idioms of templates are *esoteric* compared to conventional C++ programming, and templates can be very difficult to understand [wikipedia]
- **Compile Time**. Templates are implicitly instantiated for every different parameters

Function Templates (Parameters)

```
template<typename T>
```

`typename T` is a **template parameter**

In common cases, `T` can be:

- *generic type* (`typename`)
- *non-type template parameters*
  - *integral type* (`int`, `char`, `etc`) (not floating point)
  - *enumerator, enumerator class*

```
template<int A, int B>
int addInt() {
    return A + B; // sum is computed at compile-time
}                 // e.g. addInt<3, 4>();

enum class EnumT { X, Y };

template<EnumT Z>
int addEnum(int a, int b) {
    return (Z == EnumT::X) ? a + b : a;
}                 // e.g. addEnum<EnumT::X>(3, 4);
```

Function Templates (Code Generation)

Note: The compiler generates (at compile-time) a specific function implementation for every template parameter instance

```cpp
template<typename T>
T addX(T a, T b) {
    return a + b;
}
template<int A, int B>
int addInt() {
    return A + B;
}

int main() {
    addX(3, 4);       // generates: int   add(int, int)
    addX(3.0f, 4.0f); // generates: float add(float, float)
    addX(2, 6);       // already generated
    addInt<2, 3>();   // generates: addInt<2, 3>()
    // other instances are not generated
}   // for example: char add(char, char)
```

Function Templates (Parameter Default Value)

Template parameters can have default values

(only at the end of the parameter list)

```cpp
// template<int A = 3, int B>   // compile error
template<int A = 3>
int print1() {
    std::cout << A;
}

print1<2>();  // print 2
print1<>();   // print 3 (default)
print1();     // print 3 (default)
```

```cpp
template<typename T = int>
int print2() {
    std::cout << sizeof(T);
}
print2<char>();  // print 1
print2();        // print 4 (sizeof(int))
```

Function Templates (Parameter Default Value)

Template parameters may have no name

```
void f() {
    std::cout << "hello f()";
}

template<typename = void>
void g() {
    std::cout << "hello g()";
}

int main() {
   g();
}
```

`f()` is always generated in the final code

`g()` is generated in the final code only if it is called

Function Templates (Parameter Default Value)

Unlike function parameters, template parameters can be initialized by previous values

```cpp
template<int A, int B = A + 3>
void f() {
    std::cout << B;
}

template<typename T, int S = sizeof(T)>
void g(T) {
    std::cout << S;
}

int main() {
    f<3>();  // B is 6
    g(3);    // S is 4
}
```

Compiler can be forced to generate user-defined template function specialization

```
template<int A, int B>
int add() {
    return A + B;
}

template int add<2, 3>();

int main() {
    add<1,2>();  // the compiler generates also add<2,3>()
}
```

It is not useful in simple cases but, it has specific purpose if used with multiple cpp files (see "Code Organization" lectures)

Function Templates (Two Examples)

Ceiling division:

```
template<int DIV, typename T>
T ceil_div(T value) {
    return (value + DIV - 1) / DIV;
}
// e.g. ceil_div<5>(11); // returns 3
```

Rounded division:

```
template<int DIV, typename T>
T round_div(T value) {
    return (value + DIV / 2) / DIV;
}
// e.g. round_div<5>(11); // returns 2 (2.2)
```

Since `DIV` is known at compile-time, the compiler can heavily optimized the division (almost for every numbers, not just only for power of two)

Note: the code does not work for all cases...(see next slides)

Function Templates (Specialization)

Another example:

```
template<typename T>
T max_value(T a, T b) {
    return a > b ? a : b;
}
```

`max_value()` does not make sense for floating-point computation because of rounding errors

Solution: **Template (full) specialization**

```
template<>
T max_value<float>(float a, float b) {
    return ...   // floating point relative error implementation
}                // see "Basic I" lecture
```

Full Specialization: Function templates can be specialized only if **ALL** template arguments are specialized

Function Templates (Overloading)

Functions with templates can be *overloaded*

```cpp
template<typename T>
T add(T a, T b) {
    return a + b;
}             // e.g add(3, 4);

template<typename T>
T add(T a, T b, T c) {
    return a + b + c;
}             // e.g add(3, 4, 5);
```

Also function templates themselves can be *overloaded*

```cpp
template<int C, typename T>
T add(T a, T b) {     // it is not in conflict with
    return a + b + C; // T add(T a, T b) thanks to int C
}
```

Function Templates (Overloading)

```
template<typename T>
int f() {}

template<int C>
int f() {}

// template<short C>  // compile error for f<3>()
// int f()            // int, short are integral types

enum EnumT { A, B };

template<EnumT E>
int f() { return sizeof(E); }

int main() {
    f<3>();   // calls second definition
    f<int>(); // calls first definition
//  f<A>();   // conflicts with int
}             // it works if EnumT is an enum class
```

Type Deduction

Type Deduction

When you call a template function, you may omit any template argument that the compiler can determine or deduce (inferred) by the usage and context of that template function call [IBM]

- The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call
- Similar to function default parameters, (any) template parameters can be deduced only if they are at end of the parameter list

Full Story: IBM Knowledge Center

Type Deduction

```cpp
template<typename T>
int add1(T a, T b) {  return a + b; }

template<typename T, typename R>
int add2(T a, R b) { return a + b;  }

template<typename T, int B>
int add3(T a) { return a + B; }

template<int B, typename T>
int add4(T a) { return a + B; }

int main() {
   add1(1, 2);  // ok
// add1(1, 2u); // the compiler expects the same type
   add2(1, 2u); // ok (add2 is more generic)
   add3<int, 2>(1); // int cannot be deduced
   add4<2>(1);      // ok
}
```

Type deduction with references

```
template<typename T>
void f(T& a) {}

template<typename T>
void g(const T& a) {}  // may be also "volatile T&" or
                       // "const volatile T&"
int main() {
    int  x;
    int& y = x;
    const int& z = x;
    f(x);   // T: int
    f(y);   // T: int
    f(z);   // T: const int // <-- !! it works...but note that
    g(x);   // T: int       //     it does not for f(int& a)!!
    g(y);   // T: int       //     (only non-const references)
    g(z);   // T: int       // <-- see the difference
}
```

Type deduction with pointers

```cpp
template<typename T>
void f(T* a) {}

template<typename T>
void g(const T* a) {}  // may be also "volatile T*" or
                       // "const volatile T*"
int main() {
    int* x = nullptr;
    const int* y = nullptr;
    auto z = nullptr;
    f(x);    // T: int
    f(y);    // T: const int
    // f(z); // compile error z: nullptr_t != T*
    g(x);    // T: int
    g(y);    // T: int
}
```

```cpp
template<typename T>
void f(const T* a) {}

template<typename T>
void g(T* const a) {}

int main() {
    int* x;
    const int* y;
    int* const z = nullptr;
    const int* const w = nullptr;
    f(x);   // T: int
    f(y);   // T: int
    f(z);   // T: int
//  g(x);   // compile error, objects pointed are not constant
//  g(y);   // the same (the pointer itself is constant)
    g(z);   // T: int
    g(w);   // T: int
}
```

Type deduction with values

```
template<typename T>
void f(T a) {}

template<typename T>
void g(const T a) {}

int main() {
    int  x;
    const int  y = 3;
    const int& z = y;
    f(x);   // T: int
    f(y);   // T: int!!  (drop const)
    f(z);   // T: int!!  (drop const&)
    g(x);   // T: int
    g(y);   // T: int
    g(z);   // T: int!!  (drop reference)
}
```

Type Deduction (Pass-by-Value) 2/2

```cpp
template<typename T>
void f(T a) {}

int main() {
    int*  x;
    const int* y = nullptr;
    int* const z = x;
    f(x);  // T = int*
    f(y);  // T = int* !! (const drop)
    f(z);  // T = int* const
}
```

Type deduction with arrays

```cpp
template<typename T, int N>
void f(T (&array)[N]) {}   // type and size deduced

template<typename T, int N>
void g(T array[N]) {}

int main() {
    int  x[3];
    const int y[3] = {};
    f(x);   // T: int, N: 3
    f(y);   // T: int (const drop) (pass-by-value)
//  g(x);   // compile error, not able to deduce
}
```

```cpp
template<typename T>
void add(T a, T b) {}

template<typename T, typename R>
void add(T a, R b) {}

template<typename T>
void add(T a, char b) {}

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

template<typename T>
void f(T* array) {}  // <---

int main() {
    add(2, 3.0f); // ok, call add<T, R>(T, R)
//  add(2, 3);    // compile error (not able to decide)
    add(2, 'b');  // ok, call add(T, char)  // nearest match
    int x[3];
    f(x);         // !! call f<int>() not f<int, 3>()
}
```

Compile-time Utilities

`static_assert` (C++11) is used to tests a software assertion at compile-time

If the static assertion fails, the program doesn't compile

```
int main() {
    static_assert(2 + 2 == 4, "test1"); // ok, it compiles
    static_assert(2 + 2 == 5, "test2"); // compile error
    static_assert(sizeof(void*) * 8 == 64, "test3");
    // depends on the OS (32/64-bit)
}
```

```
template<typename T, typename R>
void f(T, R) {
    static_assert(sizeof(T) == sizeof(R), "test4");
}
int main() {
    f<int, unsigned>(); // ok, it compiles
    f<int, char>();     // compile error
}
```

`decltype` Keyword

`decltype` is a keyword used to get the type of an *entity* or an *expression*

- `decltype` never executes, it only evaluate at compile-type

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

struct A {
    int x;
};
int main() {
    int y = 3;
    decltype(y) z = 4;  // decltype(y) : int
    decltype(f);        // decltype(f) : void(*)(int, int)
    decltype(2 + 3.0);  // decltype(2 + 3.0) : double

    const A a = { 3 };
    decltype(a.x);    // entity! decltype(a.x) : int
    decltype((a.x));  // expression! decltype((a.x)) : const int
}
```

`declval` Keyword

`decltype` can be used only in *"evaluated"* contexts.

`std::declval<T>` allows to use `decltype` in expressions without go through constructors

```
#include <utlities>  // <-- needed
struct A {   // constructor implicitly declared
    int x;
};

class B {    // constructor implicitly declared
public:      // but private
    int x;
};

int main() {
    decltype(A().x);  // ok, A() build an obj A
//  decltype(B().x);  // error, B() is private
    decltype(std::declval<B>().x); // ok
}   // it is like operate on a reference
```

Definition (using keyword)

A typedef-name can also be introduced by an alias-declaration

- using keyword allows also for templated aliases
- using keyword is useful to simplify complex template expression

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

template<typename T>
using Alias = A<T>;      // called "Alias Template"

using IntAlias = A<int>;

int main() {
    Alias<int> a;
}
```

Type Traits

Type Traits

Definition (Introspection)

Introspection is the ability to inspect a type and retrieve its various qualities

Definition (Reflection)

Reflection is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime

C++ provides compile-time reflection and introspection capabilities through type traits

Type Traits

Definition (Type traits)

Type traits (C++11) defines a compile-time interface to query or modify the properties of types

The problem:

```cpp
template<typename T>
T floor_div(T a, T b) {
    return a / b;
}

int main() {
    floor_div(7, 2);    // returns 3 (int)
    floor_div(7ull, 2); // returns 3 (unsigned long long)
    floor_div(7.0, 3.0); // ??? it compiles, but the result is
}                        // not what we expect
```

Possibilities:
(1) Specialize, or (2) Type Traits

Type Traits

If we what to prevent floating-point division at compile-time a first solution consists in specialize for all “integral” types

```
template<typename T>
T floor_div(T a, T b); // declaration (error for other types)

template<>
char floor_div<char>(char a, char b) {   // specialization
    return a / b;
}
template<>
int  floor_div<int>(int a, int b) {      // specialization
    return a / b;
}
...unsigned char
...short
...
```

Very redundant!!

Type Traits

The best solution is to use **type traits**

```
#include <type_traits>      // <-- std type traits library

template<typename T>
T floor_div(T a, T b) {
    static_assert(std::is_integral<T>::value,
                  "floor_div accepts only integral types");
    return a / b;
}
```

`std::is_integral<T>` is a struct with a boolean field `value`

It is `true` if `T` is a `bool`, `char`, `short`, `int`, `long`, `long long`, `false` otherwise

Full list: `en.cppreference.com/w/cpp/header/type_traits`

Type Traits Library (Basic Types) 1/2

- `is_integral` checks for an integral type (`bool, char, unsigned char, short, unsigned short, int, long, etc.`)
- `is_floating point` checks for a floating-point type (`float, double`)
- `is_arithmetic` checks for a integral or floating-point type
- `is_signed` checks for a signed type (`float, int, etc.`)
- `is_unsigned` checks for an unsigned type (`unsigned T, bool, etc.`)
- `is_enum` checks for an enumerator type (`enum, enum class`)

- `is_void` checks for (`void`)
- `is_pointer` checks for a pointer (`T*`)
- `is_nullptr` checks for a (`nullptr`) C++14
- `is_reference` checks for a reference (`T&`)
- `is_array` checks for an array (`T (&)[N]`)

Type Traits Library (Array vs. Pointer Example)

```cpp
#include <type_traits>

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

template<typename T>
void f(T* array) {}

template<typename T, int N>
void g(T (&array)[N]) {}

template<typename T>
void h(T array) {
  if (std::is_array<T>::value)
      g(array);
  else if (std::is_pointer<T>::value)
      ; // do something
}
```

```cpp
int main() {
    int* a;
    int b[10];
    f(a); // calls f(T*)
    f(b); // !! calls f(T*)
    h(b); // partial solution
          // we can do better
}
```

- `is_const` checks if a type is `const`
- `is_volatile` checks if a type is `volatile`

C++ Special Objects:

- `is_trivial` checks for a trivial type
- `is_standard_layout` checks for a standard-layout type
- `is_pod` checks for POD types

C++ Objects:

- `is_class` checks for a class type (`struct`, `class`, not `enum class`)
- `is_empty` checks for empty class types (`struct A {}`)
- `is_abstract` checks for a class with at least one pure virtual function
- `is_polymorphic` checks for a class with at least one virtual function
- `is_final` checks for a class that cannot be extended
- `is_function` checks for a function type

Type Traits

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

template<typename T>
void f(T x) {  std::cout << std::is_const<T>::value;  }

template<typename T>
void g(T& x) {  std::cout << std::is_const<T>::value;  }

template<typename T>
void h(T& x) {
   std::cout << std::is_const<T>::value;
   x = nullptr; // ok, it compiles for T: (const int)*
}

int main() {
    const int a = 3;
    f(a);  // print false
    g(a);  // print true
    const int* b = nullptr;
    h(b);  // print false!! T: (const int)*
}
```

Type Traits (Type Manipulation)

Type traits allows also to manipulate types by using the `type` field (can be used also in the return type)

e.g. `std::make_unsigned<int>::type` returns the type `unsigned`

In general, type traits (or other *templated* structures) depends on a function template (*dependent name*). In these cases, the compile need to known if `::type` is a type or a static member in advance.

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

e.g. `typename std::make_unsigned<T>::type` is a type

The expression can be combined with `using` or `typedef` to improve the readability

e.g. `using R = typename std::make_unsigned<int>::type;`

Type Traits (Type Manipulation)

Signed and Unsigned types:

- `make_signed` makes a type signed
- `make_unsigned` makes a type unsigned

Pointers and References:

- `remove_pointer` remove pointer (`T*` → `T`)
- `remove_lvalue_reference` remove reference (`T&` → `T`)
- `add_pointer` add pointer (`T` → `T*`)
- `add_lvalue_reference` add reference (`T` → `T&`)

Const-Volatile Specifiers:

- `remove_const` remove const (`const T` → `T`)
- `remove_volatile` remove volatile (`volatile T` → `T`)
- `remove_cv` remove `const` and `volatile`
- `add_const` add `const`

Type Traits (Type Manipulation)

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

template<typename T>
void f(T x) {
  using R = typename std::make_unsigned<T>::type;
  std::cout << static_cast<R>(x);
}

template<typename T>
void g(T ptr) {
  using R = typename std::remove_pointer<T>::type;
  R x = ptr[0];
}

template<typename T>
  void h(T& x) {
  using R = typename std::remove_const<T>::type;
  const_cast<R>(x) = 0; // ok
}
```

```cpp
int main() {
  f(-1); // print 4,294,967,295

  int a[3] = {1, 2, 3};
  g(a);

  const int b = 3;
  h(b);
}
```

Type Traits (Type Relation and Transformation)

Type relations:

- `is_same<T, R>` check if `T` and `R` are the same type
- `is_base_of<T, R>` check if `T` is base of `R`
- `is_convertible<T, R>` check if `T` can be converted to `R`

Type Transformation:

- `common_type<T, R>` returns the common type between `T` and `R`
- `conditional<pred, T, R>` returns `T` if `pred` is `true`, `R` otherwise
- `decay<T>` returns the same type as function pass-by-value

Type Traits (examples)

```
#include <type_traits>

template<typename T, typename R>
T add(T a, R b) {
    static_assert(std::is_same<T, R>::value,
                  "T and R must be the samae)
    return a + b;
}

struct A {}
struct B : A {}

int main() {
    add(1, 2);    // ok
//  add(1, 2.0);  // compile error
    std::is_base<A, B>::value; // true
    std::is_base<A, A>::value; // true
    std::is_convertible<int, float>::value; // true
}
```

Type Traits (std::common_type example)

```
#include <type_traits>

template<typename T, typename R>
typename std::common_type<R, T>::type     //<-- return type
add(T a, R b) {
    return a + b;
}

int main() {
    add(3, 4.0f); // .. but we don't know the type of the result

    // we can use decltype to derive the result type of
    // a generic expression
    using result_t = decltype(add(3, 4.0f));
    result_t x = add(3, 4.0f);
}
```

Type Traits (`std::conditional` example)

```cpp
#include <type_traits>

template<typename T, typename R>
void f(T a, R b) {
    const bool pred = sizeof(T) > sizeof(R);
    using S = typename std::conditional<pred, T, R>::type;
    S result = a + b;
}

int main() {
    f(2, 'a');  // S: int
    f(2, 2ull); // S: unsigned long long
}
```

Type Traits (Get Type Name)

```cpp
#include <cxxabi.h>
#include <type_traits>
#include <string>

template <class T>
std::string type_name() {
    using TR = typename std::remove_reference<T>::type;
    auto r = abi::__cxa_demangle(typeid(TR).name(), nullptr,
                                 nullptr, nullptr);
    if (std::is_const<TR>::value)
        r += " const";
    if (std::is_volatile<TR>::value)
        r += " volatile";
    if (std::is_lvalue_reference<T>::value)
        r += "&";
    else if (std::is_rvalue_reference<T>::value)
        r += "&&";
    return r;
}
// e.g. const int a = 3;
//      std::cout << type_name<decltype(a)>(); // print "const int"
```

Template Parameters

Template Parameters

Template parameters can be:

- *integral type* (`int, char, etc`) (not floating point)
- *enumerator, enumerator class*
- *generic type* (**can be anything**)

But also:

- *function*
- *reference* to global static function or object
- `pointer` to global static function or object
- *pointer to member type* cannot be used directly, but the function can be specialized
- `nullptr_t`

Template Parameters (`example`)

Pass multiple values and floating-point types

```cpp
#include <iostream>

template<typename T>  // generic typename
void print() {
    std::cout << T::x << ", " << T::y;
//  std::cout << T::z;  // compiler error!!
}                       // "z" is not a member of Multi

struct Multi {
    float  x = 2.0f;
    double y = 3.0;
};

int main() {
   print<Multi>();   // print 2.0, 3.0
}
```

Template Parameters (example)

```cpp
#include <iostream>

template<int* ptr>    // pointer
void g() {
    std::cout << ptr[0];
}

template<int (&array)[3]>  // reference
void f() {
    std::cout << array[0];
}

struct A {
    int x    = 5;
    int y[3] = {4, 2, 3};
};

template<int A::*z>       // pointer to
void h1() {}              // member type

template<int (A::*z)[3]> // pointer to
void h2() {}              // member type
```

```cpp
int array[] = {2, 3, 4}; // global

int main() {
    f<array>();        // print 2
    g<array>();        // print 2
    h1<&A::x>();       // print 5
    h2<&A::y>();       // print 4
    print<Float>();    // print 2.0, 3.0
}
```

Function Template Parameter

```
template<int (*)(int, int)>  // <-- signature of "f"
int apply1(int a, int b) {
    return g(a, b);
}

int f(int a, int b) {
    return a + b;
}

template<decltype(f)>          // alternative syntax
void apply2(int a, int b) {
    return g(a, b);
}

int main() {
    apply1<f>(2, 3); // return 5
    apply2<f>(2, 3); // return 5
}
```