

# Modern C++ Programming

## 8. C++ TEMPLATES AND META-PROGRAMMING I

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2019, v2.0



# 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`
- `decltype`
- `using`

- **Type Traits**

- Type trait library
- Type manipulation
- Type Relation and Transformation

- **Template Parameters**

# C++ Function Template

---

# Template Overview

A **template** is a mechanism for generic programming to provide a "schema" to represent the structure of an entity

In C++, *templates* are a compile-time functionality to represent:

- A family of **functions**
- A family of **classes**
- A family of **variables** C++14

Templates are a way to make code *more reusable* and *faster*

*negative sides*: hard to read, cryptic error messages, larger binary size, and higher compile time

**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!!

## 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)
}
```

# 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

# Template 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*



## int parameter

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

## enum parameter

```
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);
```

- **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 optimize the division (almost for every numbers, not just for power of two)

# Code Generation

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

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

int main() {
    add(3, 4);           // generates: int add(int, int)
    add(3.0f, 4.0f);    // generates: float add(float, float)
    add(2, 6);          // already generated
    // other instances are not generated
} // e.g. char add(char, char)
```

## C++11 Template parameters can have default values (only at the end of the parameter list)

```
// 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)
```

## Template parameters may have no name

```
void f() {}

template<typename = void>
void g() {}

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

f() is always generated in the final code

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

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

```
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
}
```

# Function Template - Specialization

```
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 Template - Overloading

## Template Functions can be *overloaded*

```
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 templates themselves can be *overloaded*

```
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
}
```



# auto Deduction

C++17 introduces automatic deduction of *non-type* template parameters with the `auto` keyword

```
template<int X, int Y>
void f() {}

template<auto X, auto Y>
void g() {}

int main() {
    f<2, 3>();           // X: int,      Y: int
    f<2u, 2u>();        // X: unsigned, Y: unsigned

    g<2, 3>();           // X: int,      Y: int
    g<2u, 2u>();        // X: unsigned, Y: unsigned
    g<2, 3u>();         // X: int,      Y: unsigned
}
```

# Compile-Time Utilities

---

## static\_assert

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

If the static assertion fails, the program does not 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 (value)

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

- `decltype` never executes, it only evaluates at compile-time

```
int main() {  
    int      x = 3;  
    int&     y = x;  
    const int z = 4;  
    int array[2];  
  
    decltype(x);      // int  
    decltype(2 + 3.0); // double  
    decltype(y);      // int&  
    decltype(z);      // const int  
    decltype(array);  // int[2]  
}
```

## decltype Keyword ((expression))

```
bool f(int) { return true; }

struct A {
    int x;
};

int main() {
    int x = 3;
    const A a;

    decltype(x);      // int
    decltype((x));    // int&

    decltype(f);      // bool
    decltype((f));    // bool (*)(int)

    decltype(a.x);    // int
    decltype((a.x));  // const int
}
```

## 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;
    IntAlias  b; // the same
}
```

# Type Traits

---

## Introspection

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

## 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

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

The problem:

```
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

If we want 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!!**

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](http://en.cppreference.com/w/cpp/header/type_traits)

- `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])
- `is_function` checks for a function type
- `is_const` checks if a type is const

- `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

## Example (const Deduction)

```
#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** allows also to manipulate types by using the **type field** (can be used also in the return type of a function)

Example: convert `int` to `unsigned`

```
#include <type_traits>

int main() {
    using T = int;
    T x = -3;    // int

    using R = typename std::make_unsigned<int>::type;
    R y = 5;    // unsigned
}
```



In general, type traits (or other *templated* structures) depends on a template (*dependent name*) (int in the previous example). In these cases, the compiler needs to know 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;`

## Signed and Unsigned types:

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

## Pointers and References:

- `remove_pointer` remove pointer ( $T^* \rightarrow T$ )
- `remove_lvalue_reference` remove reference ( $T\& \rightarrow T$ )
- `add_pointer` add pointer ( $T \rightarrow T^*$ )
- `add_lvalue_reference` add reference ( $T \rightarrow T\&$ )

## Const-Volatile Specifiers:

- `remove_const` remove const ( $\text{const } T \rightarrow T$ )
- `remove_volatile` remove volatile ( $\text{volatile } T \rightarrow T$ )
- `remove_cv` remove const and volatile
- `add_const` add const

```
#include <type_traits>

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

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

int main() {
    const char* a = "abc";
    const int b = 3;
    f(a); // T: const char*
    g(b); // T&: const int&
}
```

# Type Relation and Transformation

## Type relation:

- `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

# Example

```
#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 same")
    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
}
```

## 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);
}
```

## std::conditional example

```
#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 in C++14/17

C++14 and C++17 provide utilities to improve the readability of type traits

```
#include <type_traits>

int main() {
    std::is_signed_v<int>;          // std::is_signed<int>::value
    std::is_same_v<int, float>;    // std::same<int, float>::value

    std::make_unsigned_t<int>;
    // typename std::make_unsigned<int>::type

    std::conditional_t<true, int, float>;
    // typename std::conditional<true, int, float>::type;
}
```



# 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`

# Generic Type Example

## Pass multiple values and floating-point types

```
#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
}
```

## Array and pointer

```
#include <iostream>

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

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

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

int main() {
    f<array>(); // print 2
    g<array>(); // print 2
}
```

## Class member

```
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

int main() {
    h1<&A::x>(); // print 5
    h2<&A::y>(); // print 4
}
```

## Function

```
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
}
```