

# Modern C++ Programming

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

### FUNCTION TEMPLATES AND COMPILE-TIME UTILITIES

---

*Federico Busato*

2022, v3.10

## 1 Function Template

- Overview
- Template Parameters
- Template Parameters - Default Value
- Template Parameters - `auto`
- Overloading
- Specialization

## 2 Template Variable

## 3 Compile-Time Utilities

- `static_assert`
- `using` Keyword
- `decltype` Keyword

## **4** Type Traits

- Overview
- Type Traits Library
- Type Manipulation

## **5** Non-Trivial Template Parameters★

# Function Template

---

## Template

A **template** is a mechanism for generic programming to provide a “*schema*” (or *placeholders*) 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

**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) { // overloading  
    return a + b;  
}  
  
char    add(char a, char b)      { ... } // overloading  
ClassX  add(ClassX a, ClassX b) { ... } // overloading
```

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

## Function Template

A **function template** is a function schema that operates with *generic* types (independent of any particular type) or concrete values

A function template works with multiple types without repeating the entire code for each of them

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

int    c1 = add(3, 4);           // c1 = 7
float  c2 = add(3.0f, 4.0f);    // c2 = 7.0f
```



# Templates: Benefits and Drawbacks

## Benefits

- **Generic Programming:** Less code and reusable. Reduce *redundancy*, better *maintainability* and *flexibility*
- **Performance.** Computation can be done/optimized at compile-time → *faster*

## 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] → hard to read, cryptic error messages
- **Compile Time/Binary Size.** Templates are implicitly instantiated for every distinct parameters

# Template Instantiation

## Template Instantiation

The **template instantiation** is the substitution of template parameters with concrete values or types

The compiler *automatically* generates a **function implementation** for each template instantiation

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

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)
```

# Implicit and Explicit Template Instantiation

## Implicit Template Instantiation

**Implicit template instantiation** occurs when the compiler generates code depending on the deduced argument types or the explicit template arguments, and such entity is used in the code

## Explicit Template Instantiation

**Explicit template instantiation** occurs when the compiler generates code depending only on the explicit template arguments specified in the declaration. Useful when dealing with multiple translation units to reduce the binary size

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

```
f(3);                // generates: void f(int) → implicit  
f<short>(3.0);       // generates: void f(short) → explicit  
template f<int>(int); // generates: void f(int) → explicit
```

# Template Parameters

## Template Parameters

**Template Parameters** are the names following the `template` keyword

```
template<typename T>
```

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

In common cases, a **template parameter** can be:

- *generic type*: `typename`
- *non-type template parameters*
  - *integral type*: `int`, `char`, etc. (but not floating point until C++20)
  - *enumerator*: `enum`, `enum class`

**int parameter**

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

**enum parameter**

```
enum class Enum { Left, Right };

template<Enum Z>
int add_enum(int a, int b) {
    return (Z == Enum::Left) ? a + b : a;
}    // e.g. add_enum<Enum::Left>(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)

## 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() {
    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



**C++11** Unlike function parameters, template parameters can be initialized by previous values

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

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

f<3>();    // B is 6
g(3);     // S is 4
```

## Template Parameter - auto

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

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

template<typename T1, T1 X, typename T2, T2 Y>
void g1() {} // before C++17

template<auto X, auto Y>
void g2() {}

f<2u, 2u>();           // X: int, Y: int
g1<int, 2, char, 'a'>(); // X: int, Y: char
g2<2, 'a'>();          // X: int, Y: char
```

# 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) { // different number of parameters
    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)
} // "C" is part of the signature
```

## Template Specialization

**Template specialization** refers to the concrete implementation for a specific combination of template parameters

The problem:

```
template<typename T>
bool compare(T a, T b) {
    return a < b;
}
```

The direct comparison between two floating-point values is dangerous due to rounding errors

**Solution:** Template specialization

```
template<>
bool compare<float>(float a, float b) {
    return ...    // a better floating point implementation
}
```

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

# Template Variable

---

# Template Variable

C++14 allows variables with templates

A template variable can be considered a special case of a 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 with "pi"
```

# Compile-Time Utilities

---



## static\_assert

C++11 `static_assert` is used to test a software assertion at compile-time

If the *static assertion* fails, the program does not compile

```
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() {
    static_assert(sizeof(T) == sizeof(R)); // message not needed in C++17
}
```

```
f<int, unsigned>(); // ok, it compiles
// f<int, char>();  // compile error
```

## using keyword (C++11)

The `using` keyword introduces an *alias-declaration* or *alias-template*

- `using` has the same semantic of `typedef` specifier with a better syntax
- `using` is an enhanced version of `typedef`
- `using` is useful to simplify complex template expression
- `using` allows to define partial and full specializations

```
typedef int distance_t; // equal to:
```

```
using distance_t = int;
```

```
typedef void (*function)(int, float);
```

```
using function = void (*)(int, float);
```

Full/Partial specialization alias:

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

template<typename T>
using Alias = A<T, int>;           // partial specialization alias

using IntAlias = A<int, int>; // full specialization alias
```

Accessing a type within a structure:

```
struct A {
    using type = int;
};

using Alias = A::type;
```

C++11 `decltype` keyword captures the type of an *entity* or an *expression*

- `decltype` never executes, it is always evaluated at compile-time

```
int      x = 3;
int&     y = x;
const int z = 4;
int      array[2];
void     f(int, float);

decltype(x);           // int
decltype(2 + 3.0);     // double
decltype(y);           // int&
decltype(z);           // const int
decltype(array);       // int[2]
decltype(f(1, 2.0f));  // void

using function = decltype(f);
```

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

struct A {
    int x;
};

int x = 3;
const A a;

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

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

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

## C++11

```
template<typename T, typename R>
decltype(T{} + R{}) add(T x, R y) {
    return x + y;
}
```

```
unsigned v1 = add(1, 2u);
double   v2 = add(1.5, 2u);
```

## C++14

```
template<typename T, typename R>
auto add(T x, R y) {
    return x + y;
}
```

# Type Traits

---

## Introspection

**Introspection** is the ability to inspect a type and query its properties

## Reflection

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

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



## Type traits (C++11)

**Type traits** define a compile-time interface to query or modify the properties of types

The problem:

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

integral_div(7, 2);      // returns 3 (int)
integral_div(71, 21);   // returns 3 (long int)
integral_div(7.0, 3.0); // !!! a floating-point value is not an integral type
```

Two alternatives: (1) Specialize (2) Type Traits + static\_assert

If we want to **prevent floating-point/other objects division at compile-time**, a first solution consists in specialize for all integral types

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

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

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

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

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

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

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

- `is_integral` checks for an integral type ( `bool` , `char` , `unsigned char` , `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` , `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

### Entity type queries:

- `is_reference` checks for a reference ( `T&` )
- `is_array` checks for an array ( `T ( & ) [N]` )
- `is_function` checks for a function type

### Class queries:

- `is_class` checks for a class type ( `struct` , `class` )
- `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

## Type property queries:

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

## Type relation:

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

## Example - const Deduction

```
#include <type_traits>

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

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

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

const int a = 3;
f(a); // print false, "const" drop in pass by-value
g(a); // print true
const int* b = new int;
h(b); // print false!! T: (const int)*
```

## Example - Type Relation

```
#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 have the same type")
    return a + b;
}

add(1, 2);           // ok
// add(1, 2.0); // compile error, "T and R must have the same type"
```

```
#include <type_traits>

struct A {}
struct B : A {}

std::is_base<A, B>::value;           // true
std::is_convertible<int, float>::value; // true
```



# Type Manipulation

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

Example: produce `unsigned` from `int`

```
#include <type_traits>

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

## Signed and Unsigned types:

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

## 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` specifiers:

- `remove_const` remove `const` ( `const T`  $\rightarrow$  `T` )
- `add_const` add `const`

Other 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 a function parameter passed by-value

## Type Manipulation Example

```
#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::add_const<T>::type;
    R y = 3;
    // y = 4;    // compile error
}

char a[] = "abc";
f(a); // T: char*
g(3); // T: int
```

## 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;
}

// we can also use decltype to derive the result type
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>
auto f(T a, R b) {
    constexpr bool pred = sizeof(T) > sizeof(R);
    using S = typename std::conditional<pred, T, R>::type;
    return static_cast<S>(a) + static_cast<S>(b);
}

f( 2,  'a'); // return 'int'
f( 2, 2ull); // return 'unsigned long long'
f(2.0f, 2ull); // return '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>

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>;
// instead of: typename std::make_unsigned<int>::type

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

# Non-Trivial Template Parameters★

---



# Non-Trivial Template Parameters

Template parameters can be:

- *integral type*
- *enumerator, enumerator class*
- *generic type (can be anything)*
- *floating-point type* since C++20

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

```
// template<float V>    // compiler error
// void print() {      // not valid before C++20

template<typename T>
void print() {
    cout << T::x << ", " << T::y;
}

struct Multi {
    static const    int    x = 1;
    static constexpr float y = 2.0f; // preferred
};

print<Multi>(); // print 2.0, 3.0
```

## Array and pointer

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

template<int (&array)[3]> // reference
void f() {
    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
}
```