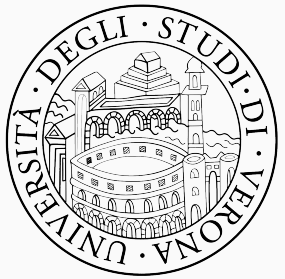
Modern C++ Programming

9. C++ Templates and Meta-programming II

Federico Busato

University of Verona, Dept. of Computer Science

2019, v2.0



Agenda

- **Class Templates**
  - Full/Partial specialization
  - Class + function template specialization
  - `virtual`
  - `friend`
  - Template dependent names
  - Template template arguments
  - Template variable
- **Template Meta-Programming**
  - Factorial
  - Log
  - Unroll
- **SFINAE**
  - Function
  - Class
- **Variadic Template**
  - Parameter recursion
  - Folding expression
  - `sizeof...`
  - Meta-programming
  - Specialization
- **STD Template Utility**
  - `std::pair`
  - `std::tuple`
  - `std::variant`
  - `std::optional`
  - `std::any`

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)
    const int N = N1;
};

int main() {
    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) {}
};

int main() {
    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
    T z;
};
```

```
template<typename T, typename R>
struct A {                  // generic template class
    T x;
};

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

int main() {
   A<float, float> a1;
   a1.x;  // ok
// a1.y;  // compile error!!

   A<float, int> a2;
   a2.y;  // ok
// a2.x;  // compile error!!
}
```

```cpp
#include <iostream>
template<typename T, typename R>
struct A {                  // generic template class
    void f() { std::cout << "A<T, R>"; }
};

template<typename T>
struct A<T, int> {          // partial specialization
    void f() { std::cout << "A<T, int>"; }
};

int main() {
    A<float, float> a1;
    a1.f();  // print "A<T, R>"

    A<float, int> a2;
    a2.f();  // print "A<T, int>"
}
```

Example 1: Class + Function Templates

```cpp
template<typename T>
struct A {
    T x = 2;
};

template<typename T, int N>
struct B : A<T> {

    template<typename R>
    int f(T y, R z) {
        return x * N;
    }
};

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

Example 2: Implement a Simple Type Trait

```cpp
#include <iostream>
#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 {};

int main() {
    using namespace std;
    cout << std::is_const<int*>::value;           // print false
    cout << std::is_const<const int*>::value;     // print false
    cout << std::is_const<int* const>::value;     // print true

    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 <iostream>
#include <type_traits>

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() {
    using namespace std;
    cout << Compare<int, float>::value;       // false
    cout << Compare<int, int>::value;         // false
    cout << Compare<B<int>, B<int>>::value;   // false
    cout << Compare<A<int>, B<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 three ways to specialize the class/function:

- *Generic* class, *generic* function
- *Full class specialization*, generic function or full function specialization
- *Partial class specialization + declaration*, generic function or full function specialization

```
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 specialize
```

Errors

```
template<typename T>
template<typename X, typename Y>
void A<T, int>::f() {}
//!   error: A<T, int> is partially specialized
//!  (A<T, int> class is not declared)

template<typename T, typename R>
template<typename X>
void A<T, R>::f<int, X>() {}
//!   function members 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
```

`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`

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;
        f();
    }
};
```

friend keyword

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

class C {
    friend class A<int>;                    // match only A<int>

    template<typename> friend class A; // match all A templates

//  template<typename T> friend class 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) {        // (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<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

int main() {
    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 = T{3.14159265358979323385}; // variable template

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

int main() {
    circular_area(3.3f); // float
    circular_area(3.3);  // double
//  circular_area(3);    // 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, analyse 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**
  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 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 (easy to debug)
- Faster compile time (no recursion)
- Works with different types (`typename T`)

Example 2: Log2

```cpp
template <int N>
struct Log2 {
    static_assert(N > 0, "N must be greater than zero");

    static const int value = 1 + Log2<N / 2>::value;
};

template <>
struct Log2<1> {   // partial specialization: base case
    static const int value = 0;
};

int main() {
    int x = Log2<20>::value; // 4
}
```

Example 3: Log

```cpp
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 4: `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) {  // Function call operator
        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/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 B, class T = void>
struct enable_if {
//  "type" is not defined of "B = false"
};
template<class 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 <iostream>
#include <type_traits>

template<typename T>
std::enable_if_t<std::is_signed_v<T>>
f(T) {
    std::cout << "signed";
}

template<typename T>
std::enable_if_t<!std::is_signed_v<T>>
f(T) {
    std::cout << "unsigned";
}

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

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

template<typename T>
void g(std::enable_if_t<std::is_signed_v<T>, T>) {
    std::cout << "signed";
}

template<typename T>
void g(std::enable_if_t<!std::is_signed_v<T>, T>) {
    std::cout << "unsigned";
}

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

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

template<typename T>
void h(T,
       std::enable_if_t<std::is_signed_v<T>, T> = 0) {
    std::cout << "signed";
}

template<typename T>
void h(T,
       std::enable_if_t<!std::is_signed_v<T>, T> = 0) {
    std::cout << "unsigned";
}

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

Function SFINAE (decltype + return type) 4/5

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

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

template<typename T>
std::enable_if_t<std::is_class_v<T>, T>
add(T a, T b) {
    return a;
}

struct A {};

int main() {
    add(1, 2);      // return 3
    add(A(), A()); // add() not supported
}
```

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

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) {}

int main() {
    f(4);
    f(4u);
}
```

Class SFINAE

```cpp
#include <type_traits>

template <typename T, typename Enable = void>
class 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>>>
{};

int main() {
    A<int>;
    A<unsigned>;
}
```

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,e
             decltype((void) std::declval<T>().y)> : std::true_type {};
//-----------------------------------------------------------------

int main() {
    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 {};

int main() {
    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 {};

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

Variadic Templates

Variadic templates

Variadic templates (C++11), 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 Templates

```cpp
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

```
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... TArgs>
auto add(TArgs... args) {
    return (... + 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
}
```

C++17 **Folding expressions** perform a *fold* of a template parameter pack over a *binary* operator

Unary folding

```cpp
template<typename... Args>
auto add(Args... args) {
    return (... + args); // unfold: 1 + 2.0f + 3ull
}

int main() {
    add(1, 2.0f, 3ll); // returns 6.0f (float)
}
```

Binary folding

```cpp
template<typename... Args>
auto add(Args... args) {
    return (1 + ... + args); // unfold: 1 + 1 + 2.0f + 3ull
}

int main() {
    add(1, 2.0f, 3ll); // returns 7.0f (float)
}
```

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 (int + double + char)
}
```

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

```
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

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

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

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

    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 any 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 `==`, $<$, $>$, $\geq$, $\leq$
- swap `std::swap`

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();
}
```

C++17 `std::variant` represents a **type-safe union** as the corresponding objects know which type is currently being held

`std::variant` can be indexed by:

- `std::get<index>(variant)` an integer
- `std::get<type>(variant)` a type

```
#include <variant>

int main() {
    std::variant<int, float, bool> v(3.3f);

    int  x = std::get<0>(v);    // return integer value
    bool y = std::get<bool>(v); // return bool value
//  std::get<0>(v) = 2.0f;      // run-time exception!!
}
```

Another useful method is `index()` which returns position of the type currently held by the variant

```
#include <variant>
using namespace std;

int main() {
    std::variant<int, float, bool> v(3.3f);

    cout << v.index(); // return 1

    std::get<bool>(v) = true
    cout << v.index(); // return 2
}
```

It is also possible to query the index at run-time depending on the type currently being held by providing a **visitor**

```cpp
#include <iostream>
#include <variant>

struct Visitor {
    void operator()(int& value)   { value *= 2; }

    void operator()(float& value) { value += 3.0f; } // <--

    void operator()(bool& value)  { value = true; }
};

int main() {
    std::variant<int, float, bool> v(3.3f);

    std::visit(v, Visitor{});

    std::cout << std::get<float>(v); // 6.3f
}
```

C++17 provides `std::optional` facilities to represent potential "no value" states

As an example, it can be used for representing the state when an element is not found in a set

```
#include <iostream>
#include <optional>

std::optional<std::string> find(const char* set, char value) {
    for (int i = 0; i < 10; i++) {
        if (set[i] == value)
            return i;
    }
    return {}; // std::nullopt;
}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    char set[] = "sdfslgfsdg";
    auto x     = find(set, 'a');

    if (!x)
        cout << "not found";
    if (x.has_value())
        cout << "not found";

    auto y = find(set, 'd');
    cout << *y << " " << y.value(); // print '1' '1'

    x.value_or(-1); // print '-1'
    x.value_or(-1); // print '1'
}
```

std::any

C++17 introduces `std::any` to hold arbitrary values and provides **type-safety**

```cpp
#include <any>
#include <iostream>
using namespace std;

int main() {
    std::any var = 1; // int
    cout << var.type().name(); // print 'i'

    cout << std::any_cast<int>(var);
//  cout << std::any_cast<float>(var); // exception!!

    var = 3.14;         // double
    cout << std::any_cast<double>(var);

    var.reset();
    cout << var.has_value(); // print 'false'
}
```