Advanced Programming for Scientific Computing (PACS)
Lecture title: Template metaprogramming, type_traits and concepts

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Working with types

Generic programming, which in C++ translates in template programming allows to treat types as sort-of "variables".

This fact on the one hand requires tools to interrogate or manipulating types and, on the other hand, opens up the possibility of implementing more efficient code by "letting the compiler do the work".

Indeed, all operations that involve templates must be resolved at the moment of the instance of the template during the compilation process, this fact can be exploited by letting the compiler perform tasks that would be otherwise done at run-time.

Some elements of Metaprogramming

Metaprogramming is the design of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. This may allow programmers to minimize the number of lines of code to express a solution by the user (hence reducing development time), or to generate more efficient code.

In C++ we indicate with template metaprogamming techniques by which the "intermediate code" (that you never see!) is in fact generated by the compiler thanks to the use of C++ templates.

Template metaprogramming

Metaprogramming uses only constructs that can be resolved at compile time (static constructs). It is a small subset of C++ constructs, but sufficient to do quite a lot of stuff.
Our main aims are:

- To allow a more generic programming, by implementing a better control on types (type_traits and concepts).
- Allow to increase the efficiency of some algebraic manipulation by letting the compiler do some operations at compile time (numerical metaprogramming).

Some metaprogramming techniques relies to the SFINAE paradigm. This rule applies during overload resolution of function templates: *when substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error.*

SFINAE

The rules for SFINAE are complex (if you wish look here), but in fact the mechanism is not so complicated, we give here an example

```
template <class T> int f(typename T::value_type);
template <class T> int f(T);
...
int i = f<int>(10);// uses 2nd overload
int j = f<vector<int>> f(10);// uses 1st overload
```

Here, when the compiler tries in the first overload to resolve **int**::value_type has a failure, since **int** does not define any type called value_type (indeed it does not define any type at all). But this failure is not an error since there is an overload that works fine: indeed std::vector stores a type alias called value_type!

SFINAE is essential for generic programming to work! remember: SFINAE works only during template parameter substitution.

Unevaluated context and static evaluation

Other two important concepts are those of unevaluated context and **static** evaluation. Normally an expression is evaluated, i.e. it expresses a value:

```
c= a +b;
```

the expression "a+b" is evaluated and the resulting value is assigned to c.

But there are cases where expressions are not used to provide a value, but only information about a type. We are in an unevaluated context

Unevaluated context

For instance, the operands of **sizeof**, decltype (as well as alignof() and **typeid**()) are never evaluated:

- No code is generated;
- The operand may not be a expression, but just a type.
- We need only declaration of functions or classes.

```
struct A {
double M_x,
int M_n
double fun(double);
};// only declaration

int n= sizeof(A); //# of bytes of an object of type A
int m = sizeof a; //# of bytes of an object of type A
```

In the last use of **sizeof**, a is not evaluated.

Static evaluation

The compiler may perform arithmetic operations if the operands are constant expressions.

```
constexpr double f(const double x){return x*x;}
...
constexpr double z=9.0;
double c = 5.0 +3.0 // it's 8.0
double d = z + f(z);// its 90!
```

The right-hand-side of last two last statements may be computed at compile-time (static evaluation), while here

```
std::cin >>c;
d = fun(c);
```

the computation may only be performed run-time.

Tools for template metaprogramming

We will give description of some techniques for metaprogramming. But let's start with a brief explanation of `decltype`, `declval<T>` and `invoke_result`. `declval` is defined the header `<utility>`, `invoke_result` in `<type_traits>`.

decltype and declval<T>

decltype(expr) returns the type of expression `expr`. declval<T> (in <utility>) makes it possible to deduce the return type of member functions of T using decltype, without the need to go through constructors!.

```
class A{
...
public:
double foo();
}
...
decltype(std::declval<A>().foo()) n;// n is an double
```

No object of type A is constructed and foo() is not called. We just get the type of the return value!. declval can only be used in an unevaluated context.

An example of use of Declval

in MetaProgramming/DecltypeDeclVal/main_decltype.cpp you find an example of the use of `decltype` and `declval<T>`.

In that example you find also an use of the type traits (see next slides) `is_constructible` and `is_default_constructible`, and of the `typeinfo` utility!

invoke_result, formely result_of, now declared deprecated, can be used to get the return type of functions, particularly useful in the case of overloaded functions. It can also be used as an alternative to the decltype/declval pair for member functions (with a more complex syntax).
The class

```
template< class F, class... ArgTypes> class invoke_result;
```

contains a type alias, called type with the type returned by the callable objects of class type F when used with arguments Args... (the list may also be empty). The type alias is not present if the callable object cannot be called with those argument types!
You can use invoke_result_t<> in place of invoke_result<>::type (as explained later).
Let's see an example in the next slide.

Use of invoke_result

```
struct S { // a callable object of class type
// two overloads of the call operator
  double operator()(char, int&) const;
  int operator()(int)const;
};
double fun(int x);
...
std::invoke_result_t<S,char,int&> b;// b is a double
std::invoke_result<S,int>::type x; //x is an int
std::invoke_result_t<decltype(fun),int> z;// z is a double
```

Note the use of decltype since fun is not a type but the name of a function. In fact, decltype(fun) returns **double** (∗)(**int**), but maybe you prefer not to know it :).

Use of invoke_result

invoke_result may also be used to interrogate the return type of member function different from the call operator. The syntax is a bit weird:

```
struct C {
  double Func(char, int&);
...};
...
std::invoke_result_t<decltype(&C::Func),C, char, int&> y;
// y is a double.
```

Explanation of the arguments: &C::Func is a pointer to the (non static) member function Func of C, called by an object of type C (the second argument) and taking **char** and **int** & as arguments.

Note: Why the second argument? Well, it accounts for inheritance. That argument may be a class derived from that indicated in the first argument!.

A note on static_assert vs assert

The differences:

assert available in <cassert> is a cpp macro that allows assertion checks at run time, if the predicate is false the program terminates. static_assert operates at compile time and, if the predicate is false you have a compiler error.

assert is disabled if the preprocessor variable NDEBUG is set. static_assert is (obviously) always enabled.

You cannot add a message in assert, but see Utilities/ExtendedAssert.hpp for some home-made extensions of assert

Another note: enriching messages

The preprocessor adds two macro variables: `__FILE__` and `__LINE__`, that contain the name of the file and the current line number! So you can write

```
static_assert(condition,"Error at line __LINE__ of file __FILE__");
```

or

```
std::cerr<<"Got stuck at line __LINE__\n";
```

The Utilities/ExtendedAssert.hpp utilities take advantage of that.

Type Traits

Type traits are tools that operate on types. They are very useful in template metaprogramming and in C++ we have an extensive set of type traits into the language. Look here for the full list. You should include <type_traits>.

C++ Standard Type traits

C++ type-traits are template classes (actually structs), It is not important to know the details, just that they either provide a value (in fact a constant expression) or a type (or both).

If it provides a value, the trait has a static variable called value, containing the value.

If it provides a type, it has a type alias called type which gives the type.

Moreover, if the trait returns a value, it also provides value_type (the type of the value), and is implicitly convertible to that type (providing value).

Two goodies

To simplify life, we have type_trait_t<T> as an alias to **typename** type_trait<T>::type:

```
template<class T>
using type_trait_t=typename type_trait_t<T>::type;
```

It spares us the hideous **typename** for template dependent types

We also have type_trait_v<T>, which is equivalent to type_trait<T>::value:

```
template<class T>
constexpr type_trait<T>::value_type type_trait_v=
                                  type_trait<T>::value;
```

An example

is_pointer<T> may be used to interrogate if a type is a pointer.

```
#include <type_traits>
template<class T>
auto fun(T x)
{
    if constexpr(is_pointer_v<T>) // or is_pointer<T>::value
     // do something
    else
     // do something else
}
```

This function behaves differently whenever the argument is a pointer. Note the use of **if constexpr**, so the compiler will actually compile only the true block.

This use of type-traits spare us the need of writing an overload to specialize the function for pointers.

Organization of standard type_traits

- Primary type categories: is_int<T>, is_pointer<T>, is_function<T>, is_rvalue_reference<T>, is_enum<T>...
- Composite type categories: is_scalar<T>, is_reference<T>, is_member_pointer<T> ...
- Type properties: is_const<T>, is_trivial <T>, is_abstract<T>, is_polymorphic<T>...
- Supported operations: is_copy_constructible<T>, is_assignable<T>, has_virtual_destructor<T>...
- Property queries: rank<T>, extent<T>,...
- Type relationships: is_base_of<B,D>, is_convertible<From,To>..
- Type modifications: remove_const<T>, remove_reference<T>, decay<T>, add_const<T>, make_unsigned<T>...

and more... The full list in cppreference.com.

It determines the common type among all types `T...`, that is the type all `T...` can be implicitly converted to. If such a type exists, the member `type` names that type.

Otherwise, there is no member `type`. Useful in mixed mode arithmetic. Its use however, is less crucial now that we have automatic function where the return type is automatically deduced.

An example of common_type<T...>

The common type of the arguments and **double**.

```
template<class T, class U>
struct Foo{
    std::common_type_t<double,T,U> v;
     ...
    };
..

Foo<int,int> a;// a.v is a double
Foo<std::complex<double>,double> b;// b.v is complex<double>
Foo<long double,double> c;// c.v is long double
```

The rules for common_type are rather complex, but usually it does what you expect (at least for POD).

A complex type trait: enable_if

```
template< bool B, class T = void > struct enable_if;
```

It is a convenient way to leverage SFINAE to conditionally remove functions from overload resolution based on type traits and to provide separate function overloads and specializations for different type traits.

`std::enable_if` can be used as an additional function argument (not applicable to operator overloads), as a return type (not applicable to constructors and destructors), or as a class template or function template parameter.

How does enable_if work?

```
enable_if<b,MyType>
```

If `b` is **true** it defines a type in enable_if<b,MyType>::type, equal to MyType (or **void** if MyType has not been given). If `b` is **false** then it does not define ::type. So, enable_if<**false**,MyType>::type may be used to generate a substitution failure.

A possible implementation (just to show how it can be done) is

```
template<bool B, class T = void>
struct enable_if {};
// partial specialization for B=true
template<class T>
struct enable_if<true, T>
 { using type=T; };
```

A possible use in MetaProgramming/Enable_if/Norms.hpp.

An example of enable_if

I want that the gcd template function exist only if the arguments are integers. Trying to do gcd(5.0,6.0) should generate the compiler error: gcd(**double**, **double**) is not defined!.

```
template
<class T,
   std::enable_if_t
     <std::is_integral<T>::value,T > =0 >
     // NOTE THE SPACE BETWEEN > AND = !!
>
T gcd(T a, T b){ return (b == 0)? a: gcd(b, a % b);}
```

If T is an integer the second template parameter is an integer defaulted to 0. If not, enable_if does not define a type, and the function is not generated. The default value to enable calling gcd without specifying the second template parameter, which is used only to activate enable_if.

Note the use of enable_if_t alias to avoid **typename**.

An explanation

When a template parameter is not actually used, I can avoid giving it a name

In the above example, then, if I instantiate the function with T=**int** (it means that the argument is an **int**) the template parameters expands to

```
<int, int=0>
```

which is admissible.
I can give a name to the second parameter if I want to, but it is irrelevant since it is not used. The second template parameter is indeed a sort of “dummy” parameter, necessary only to activate the mechanism of enable_if.

Another example of enable_if

I have a class transposeView<Matrix> that provides a view of a Matrix as its transpose. I want to make sure that the non-const version of **operator**(**int**,**int**) used to address elements of the transposed Matrix is not generated if Matrix is const!

```
template<class Matrix>
class MatrixView
{
  // the type ot the matrix element
  using value_type=typename Matrix::value_type;
   ...
  template<typename T=Matrix>
  std::enable_if_t<!std::is_const_v<T>,value_type &>
    operator()(size_type r, size_type c){return matrix_(c,r);}
```

Note the trick of using a defaulted template to enforce template parameter substitution (otherwise it will not work: I need SFINAE!).

The full example in MetaProgramming/transposeView

A note

The use of generic programming in C++ is increasing, since it has several advantages. Consequently, the language is evolving in order to make "template metaprogramming" techniques simpler.

Indeed, the introduction of the "compile time if" **if constexpr** can sometimes (but not always) avoid the need of enable_if. Also the tag dispatching technique that we will present later can now be replaced by **if constexpr**.

And the introduction of concepts may simplify life further.

Testing memory layout

Some types of traits may be used to check the memory layout of a type. We need to clarify two important concepts: trivially-copyable and standard layout.

I give just a general explanation of the terms, for the details consult any good manual or the web, like cppreference.com.

Trivially copyable and standard layout

If a type is trivially copyable it is possible to copy data via memcpy, rather than having to use the standard copy operations. In general, a trivially copyable type is a type for which the underlying bytes can be copied directly to a buffer and then back into another object of the same type. So you can define more efficient copy operations and serialize the object directly. This is also useful in an Message Passing context.

A type has a standard-layout if it orders and packs its members in a way that is compatible with the C language. A pointer to a standard-layout class may be converted (with `reinterpret_cast`) to a pointer to its first non-static data member and vice versa. But the most important thing is: you can communicate with C code.

See the example in MetaProgramming/Trivial.

Some notes

The type traits is_trivially_copyable and has_standard_layout do what they say but they give sufficient, not necessary conditions. The compiler is not psychic and cannot examine the semantic of your code:

```
struct S{
    S& operator=(const S& r){i=r.i; a=r.a; return *this;}
    int i;
    double a;
};
```

This class is "morally" trivially copyable, since the copy-assignment I have defined does exactly what the synthetic (trivial) one would do!. But the compiler cannot know it, so is_trivially_copyable<S>::value is false.

Some advice

- Don't define your own constructor/move-copy assignment operators if the synthetic ones are provided and do what you want (and it happens in most of the cases).
- Containers and algorithms of modern implementation of the standard library make use of the properties of contained types to optimize copy of objects. Indeed, they may use `memcopy` (or `memmove`), which are more efficient than the usual copy operations. Another reason to use standard containers and algorithms!

Defining your type-traits

So far we have seen some used of type-traits provided by the C++ standard library. But maybe you may want to define yours!.
We will present here only a few basic examples, also because with the introduction of concepts some things can (sometimes) be made simpler.

First, we introduce `std::integral_constant<T,V>` and its most important specialization, `std::false_type<T>` and `std::true_type_t`

integral_constant<T,V>

This little trait expresses and integral constant expression of value V and type T (clearly a integral type!). It containd value=V and is implicitly convertible to T and the conversion provides the value V. You can thing it as a map from an (integral) value to a type. An example of type tagging

```
enum ORDERING{ ROWMAJOR, COLUMNMAJOR };
template <ORDERING O> // a template alias for simplicity
using OT=std::integral_constant<ORDERING,O>;
template<ORDERING O>
Class Matrix{
public:
...
  auto getIndex(size_t i, size_t j){ return getIndex(i, j, OT<O>{});}
private:
 auto getIndex(size_t i, size_t j, OT<ROWMAJOR>);// version for ROWMAJOR
 auto getIndex(size_t i, size_t j, OT<COLUMNMAJOR>);// version for COLUMNMAJOR
 };
```

the correct version is obtained automatically by choosing the correct overload (tag dispatching)

dispatching with **if constexpr**

Clearly you can obtain the same result with an **if constexpr** (more similar to ordinary programming)

```
enum ORDERING{ ROWMAJOR, COLUMNMAJOR };
template<ORDERING O>
Class Matrix{
public:
...
  auto getIndex(size_t i, size_t j){
  if constexpr (O == ROWMAJOR)
   // row major version
   else
    // columnmajor version
  }

 };
```

Simpler, but I like also the solution with tag dispatch wich is also working if you compile with a non c++17 compliant compiler.

true_type and false_type

These specializations of integral_constant. They represent bool and they convert implicitly to the corresponding value

```
bool a= true_type{}; // a is true
auto c=false_type::value; // c is a bool with value false
```

We will see a possible use in the following handcrafted examples.

A simple example

We want to assess if a type is an std::complex<T>

```
#include <type_traits>
#include <complex>

template<typename T>
struct is_complex : std::false_type {};// primary template
// partial specialization for complex
template<typename T>
struct is_complex<std::complex<T> > : std::true_type {};

// Example of usage
template<typename T> auto modulo(T x){
  if constexpr (iscomplex<T>{}){
   return std::sqrt(x.real()*x.real()...
  else
   return std::abs(x)
...
}
```

This example of usage is useless since *std* :: *abs* has already an overload for complex that computes the modulo.

Testing for the presence of a member function

It would be nice if I could make the compiler check if a type contains the method `clone()`. It is possible, but it is not so easy, there are different ways of doing it. I present one in Utilities/CloningUtilities.hpp. It is rather complex. I leave it to the nerds. In the folder MetaProgramming/IsClonable you have a simpler version, which however does not check the return type.

We omit the detail since now with the introduction of concepts and requires expressions the test can be made much more easily.

Concepts

Concepts have been introduces in C++20 to make template programming safer and simplify the creation of type traits, Concepts, in fact, introduce semantic to types. The main objectives are

- Have a safer and more understandable template code;
- Simplify some template metaprogramming contructs;
- Provide better error messages;

Let start with an example

I want to create a template function that however accepts only integral types. Before C++20 I could do

```
#include <type_traits>
template <typename T>
T absdiff (T const & a, T const & b)
{
  std::static_assert(std::is_integral_v<T>,"Must be integral");
  return a>b? a-b: b-a;
}
```

This is fine and still valid, but it may become tedious and not scalable if the condition on the type is more complex.

The concept solution

Using a concept with a requires clause

```
#include <concepts>
template<class T>
requires std::integral<T>
T absdiff (T const & a, T const & b)
  {return a>b? a-b: b-a;}
```

or, if you prefer a trailing requires,

```
#include <concepts>
template<class T>
T absdiff (T const & a, T const & b) requires std::integral<T>
  {return a>b? a-b: b-a;}
```

or, even nicer, just

```
#include <concepts>
template <std::integral T>
T absdiff (T const & a, T const & b)
  {return a>b? a-b: b-a;}
```

Simpler and clearer, isn't it! std::integral is a predefined concept provided by the standard library that represent the semantic of *integral type*, the (almost) full list is here.

Other examples

If the concept takes just one template parameter you may use a constrained template parameter

```
template<std::floating_point T>
T foo(T const &);
```

The requires clause allows more flexibility

```
template<typename B, typename D>
requires std::derived_from<B,D>// D must derive from B
void foo(B const &, D const &);
```

You can combine concepts

```
template<typename T>
requires std::integral<T> || std::floating_point<T>
void foo(B const &)
```

Constrained **auto**

In many places where you can use **auto** you can now use constrained **auto**:

```
// Error if the function does not return and integral type
template<class T>
std::integral auto foo(T val);
// This is equivalent to
//  template<std::movable T> void foo(T&& a)
void foo(std::movable auto&& a);// ok only if arg can be moved
// this lambda accepts only floaing points
auto f=[](std::floating_point auto x){return 3*x;}
```

Use with class templates

```
template<std::floating_point T, typename U>
class C{
public:
 void push_back(const U & e) requires std::copyable<U>;
 ...
}
```

Also for class you have also the requires clause syntax

```
template<typename U>
requires std::swappable<U,double>
class C{
...
}
```

Overloading with concepts

Concepts participate to overloading/specialization mechanism

```
#include <iterator>
template <std::forward_iterator I>
auto foo(I start, I end);// version 1

template <std::random_access_iterator I>
auto foo(I start, I end);// version 2
...
...
std::vector<double> v;
std::set<double> s;
foo(std::begin(v),std::end(v));// calls version 2
foo(std::begin(s),std::end(s));// calls version 1
```

Specialization with concepts

Another example with classes

```
#include <concepts>
template <typename T>
class MyClass{....}

// Specialization for floating points
template <std::floating_point T>
class MyClass{....}
....

MyClass<float> a; // uses specialization
MyClass<int> b; // uses primary template
```

Writing your concepts

The general syntax is

```
template<typename T> // you can have more the 1 param.
concept Name = constraint_expression;
```

A `constraint_expression` can be a **constexpr** boolean expression, logical combination of other concepts or a requires expression.

```
// using a user defined trait (defined in is_complex.hpp)
template <class T>
concept complex = apsc::TypeTraits::is_complex_v<T>;
// combining two concepts
template <class T>
concept numeric = std::floating_point<T> || complex<T>
// using requires expression
template <typename T, typename Q>
concept multipliable = requires( T v, Q q)
{ v*q;}// multiplication btw T and Q must be valid
```

Let's look in mode details the requires expression (not to be confused with the requires clause seen in the previous slides).

The requires expression

The requires expression is a sort of "template function" whose content is however not evaluated but just checked for consistency. If we have a failure the requires returns **false**. A simple example

```
template <typename T>
concept addable = requires (T a, T b) {a+b;};

template <typename T>
concept has_value_type = requires {
typename T::value_type; // Type member value_type must exist
};

...
template <has_value_type T> // fails it T han no value_type
class C{...};
void function(addable auto x){...} // failes if x not addable

...
std::set<double> s;
function(1); // OK
function(s); // Fails: operator + not defined for sets
C<std::set<double>> c; //ok
C<int> d; //Fails. int does not contain a type called value_type
}
```

Testing the presence of a member function

Inside the requires expression the {} block (called compound expression) may be used to create a unevaluated context for an expression. It tests its validity and may also check the expression using a concept. Here, a possible concept for clonable classes :

```
template <typename T>
concept clonable = requires( T v)
{
  {v.clone()}->std::convertible_to<std::unique_ptr<T>>;
};
```

Here an alternative way of testing a if a type is std::complex number

```
template<class T>
concept Complex=
requires(T x)
{
   {x.imag()} -> std::same_as<T>;
   {x.real()} -> std::same_as<T>;
  };
```

Note the use of the same_as concept to test the type.

A note

Note the difference between the type-trait is_complex defined at the beginning of this lecture, where the construction relies on class specialization and SFINAE mechanism, and the concept Complex<T> in a previous slide.

The main difference it that the type traits is_complex<T> checks whether T is in fact a complex<C>, for any C; the concept Complex<T> tests whether T satisfies a specific semantic: having two function members of a given name and return type.

The nested requires expression

Look at this concept that wants to test if a class defines objects of size greater than 16 bytes:

```
#include <concepts>
template <typename T>
concept bigsize = requires (T a) {
requires sizeof(a) > 16; // size of object must be >16bytes
};
```

If I had written just

```
sizeof(a) > 16;
```

the concept will always return **true** since that expression is always valid. I need a nested requires clause to test the value of the resulting constant expression.

Overview of the synopsis of requires expressions

```
template<typename T> /*...*/
requires (T x) // optional set of fictional parameter(s)
{
    // simple requirement: expression must be valid
    x++;    // expression must be valid

    // type requirement: `typename T`, T type must be a valid type
    typename T::value_type;
    typename S<T>;

    // compound requirement: {expression}[noexcept][-> Concept];
    // {expression} -> Concept<A1, A2, ...> is equivalent to
    // requires Concept<decltype((expression)), A1, A2, ...>
    {*x}; // dereference must be valid
    {*x} noexcept; // dereference must be noexcept
    // dereference must  return T::value_type
    {*x} noexcept -> std::same_as<typename T::value_type>;

    // nested requirement: requires ConceptName<...>;
    requires integral<T>; // constraint integral<T> must be satisfied
};
```

Concepts and type-traits

Concepts and type traits are very much linked. We have seen that a type-trait can be easily transformed into concepts. For instance here I exploit the `std::is_integral` trait:

```
template<typename T>
concept integral = std::is_integral_v<T>;
```

At the same time, concepts can be used to test condition on types, similarly to type traits. This code snippet

```
if constexpr(integral<T>)
{...
```

is equivalent to

```
if constexpr(is_integral_v<T>)
{...
```

Beware: Sometimes there are subtle differences between type-traits and related concept. Always consult good references.

Conclusions

What we have seen are the major feature of concepts. On references on the web you find more details, but what we have said so far is quite sufficient to enable you to use constraints in your program if you wish.
Some other examples are in the folder C++20/Concepts/.

To know more

If you want to know more, besides cppreference.com you have some nice blogs, here, here, here and here (in order of complexity), plenty of examples.