

# Advanced Programming for Scientific Computing (PACS)

## Lecture title: Variant and Ranges

Luca Formaggia

MOX  
Dipartimento di Matematica  
Politecnico di Milano

A.A. 2022/2023

# Some recent additions to c++

This lecture aims at introducing a few recent additions to the language:

- `std::variant` and `std::visit`, with which you can implement the visitor design pattern.
- `std::any`: any type you want, but in a type-safe way
- `ranges` and `views`: a new way of operating on containers! And new algorithms.
- `std::invoke`: to invoke callable objects in a uniform way (since c++17)

# The `variant<...>` utility



You may want a variable able to represent values of different type. In C (and C++) we have **union**, but it is quite complex and unsafe. For this reason, `std::variant` has been introduced to deal with elements of different data types.

```
std::variant<double, std::string> var;  
var = "Hello"; // It's holding a string  
var = 10.5; // now holds a double  
double c = std::get<double>(var); // c is now = 10.5  
// Run-time error: not currently holding a string!!  
// std::string s = std::get<std::string>(var);  
// But I can check!  
if (var.holds_alternative<std::string>())  
{ ... // it's a string
```

## std::visit

Variants becomes very powerful when used together with the 'std::visit' facility, which indeed allow us to implement the [visitor pattern](#). Better start with an example.

Let's have a set of (non necessarily polymorphic) classes representing different geometric objects

```
class Triangle{
    public:
        double area()const;
    ...
}
class Square{
    public:
        double area()const;
    ...
}
class Point{
    public:
        // No area() method!
    ...
}
// etc. many others...
```

Let's define a **visitor** as a set of overloaded function operating on our set of polygons (note the use of an automatic function to identify a fallback action).

```
struct TotalArea {
    double totalArea=0;
    void operator()(Triangle const & x){totalArea+=x.area();}
    void operator()(Square const & x){totalArea+=x.area();}
    void operator()(Pentagon const & x){totalArea+=x.area();}
    //... for all other object with area()
    void operator()(auto const & x){}//fallback
}
```

I define a variant able to hold polygons

```
using GeoObj=std::variant<Point, Triangle, Square, Pentagon>;
```

## Visiting a set of geometric objects

```
std::vector<GeoObj> listOfGeo; // a vector of variants  
//.. I fill the vectos with various objects  
// Now I compute the total area of all objects  
TotalArea area;  
for (auto const & x:listOfGeo)  
    std::visit(area,x); // I visit the variant  
std::cout<<"Total_Area="<<area.totalArea;
```

The function `visit(visitor,variant)` automatically selects the correct overload depending on the actual type stored in the variant.

This way, we can implement a new type of dynamic polymorphism that does not need virtual methods.

# There is much more...

In [STL/Variant](#) folder you find other examples, including using `std::visit` with two variants, and how to make a visitor on the fly using the overloaded utility given in [Utilities/overloaded.hpp](#)

A more elaborated example is found in the [DesignPatterns/VisitorAndVariant/](#) folder.

std::any

C++17 has also introduced a type-safe container for single values of any copy constructible type.

You have to use `std::any_cast<T>` to cast a `std::any` variable to a variable of type `T` (exception if not possible).

```
#include <any>
...
std::any a = 1;
if (a.has_value())
    int k = std::any_cast<int>(a);
...
a=3.14; // now it holds a double
try
{
    auto z=std::any_cast<int>(a); //fails!
}
catch (const std::bad_any_cast& e)
{
    std::err << e.what() << '\n';
}
```



# Ranges

C++20 has revolutionized the concept of ranges and modified the way standard algorithms are used. The **new ranges** and related algorithms are available in the namespace `std::ranges` after the inclusion of the `<ranges>` header.

Before C++20 a range was defined as a pair of iterators  $[b, e[$ . In C++20 ranges a range can be

- A pair of iterators  $[b, e[$  where  $e$  is a "sentinel" that indicates the end of the range;
- A  $\{b, n\}$  pair where  $n$  is the number of elements;
- A  $\{b, p\}$  pair, where  $p$  is a **predicate**: if  $p(i)$  is **false** for iterator  $i$  we are at the end of the range.

Moreover, **every class whose public interface provides the methods `begin()` and `end()` returning valid iterators defines a range**. It applies also to class for which the free functions `std::begin()` and `std::end()` do the same.

## A first consequence

The first consequence (but there is much more!) is that

```
std::vector<double> v={3,7,9.2,44.3};  
std::sort(v.begin(),v.end()); /// sorts in ascending order
```

can now be written as

```
#include<ranges>  
std::vector<double> v={3,7,9.2,44.3};  
std::ranges::sort(v); /// sorts in ascending order
```

since `std::vector` satisfies the conditions highlighted in the previous slide!

So, a first notable consequence is that the call of standard algorithms is simplified when you have to operate on all elements of a container.

**Note:** you can still use the old style.

# Ranges concepts

We have a set of concepts to distinguish ranges depending on their capabilities. They refer to the analogous iterators capabilities, and are

- ▶ **input\_range**. The most basic. The iterators are just `input_iterator`;
- ▶ **forward\_range**. It is only possible to move from one element to the next;
- ▶ **bidirectional\_range**. It is only possible to move from one element to the next or to the previous;
- ▶ **random\_access\_range**. It is possible to address any element of the range;
- ▶ **contiguous\_range**. A `random_access_range` where elements are contiguous in memory;

The inclusions are evident.

# Projectors

Before moving forward, I want to present another feature of the new ranges algorithm. Beside supporting parallel execution, as the old version, many of them support **projections**. With **projector** here we mean an operation on the element of the range to be performed before the actual algorithm. But better look at an example<sup>1</sup>. Here we do a regular sort:

```
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>
int main(){
    std::vector vec { -1, 2, -3, 4, -5, 6 };
    auto print = [](int i) { std::cout << i << ", "; };
    std::ranges::sort(vec);
    std::cout << "regular_sort:\n";
    std::ranges::for_each(vec, print);
}
```

and we get -5, -3, -1, 2, 4, 6

---

<sup>1</sup>These examples have been taken from [cppstories.com](http://cppstories.com) 

# Projectors

Suppose that we want to sort by the absolute value. With the standard `std::sort` algorithm we had to do something complicated. With the range algorithm we can use a projector:

```
int main(){
    std::vector vec { -1, 2, -3, 4, -5, 6 };
    auto print = [](int i) { std::cout << i << ", " ; };
    std::cout << "with abs() projection: \n";
    std::ranges::sort(vec, {}, [](int i) { return std::abs(i); });
    std::ranges::for_each(vec, print);
}
```

We get -1, 2, -3, 4, -5, 6.

Explanation in the next slide.

# Explanation

From [cppreference.com](http://cppreference.com) (ll overload)

```
template< ranges::random_access_range R, class Comp = ranges::less ,  
         class Proj = std::identity >  
requires std::sortable<ranges::iterator_t<R>, Comp, Proj>  
constexpr ranges::borrowed_iterator_t<R>  
sort( R&& r, Comp comp = {}, Proj proj = {} );
```

we can see that `ranges::sort` may take in input a range `r` a comparison operator `comp`, defaulted to `ranges::less` and a **projector** `proj`, which defaults to the `std::identity` (i.e. it does nothing).

With the call

```
std::ranges::sort(vec, {}, [](int i) { return std::abs(i); });
```

we say that we want the default comparison (the `{}` in the second argument) and we give the wanted projector on the fly using a lambda.

## Another example

Projectors also allows to extract members.

```
struct Task {  
    std::string desc;  
    unsigned int priority { 0 };  
};  
  
int main(){  
    std::vector<Task> tasks {  
        { "clean_up_my_room", 10 }, { "finish_homework", 5 },  
        { "test_a_car", 8 }, { "buy_new_monitor", 12 }  
    };  
    std::ranges::sort(tasks, std::ranges::greater{}, &Task::priority);  
    std::cout << "my_next_priorities:\n";  
    ...  
}
```

Here I am sorting tasks according the member variable priority.  
With the greatest priority first!

**Projectors** increase the power of standard range-based algorithm by simplifying operations that before required more convoluted coding.

# Ranges Views

The new definition of ranges allows to create **views that accept in input a range and return another range**.

$$\{b_1, e_1\} \rightarrow \text{view}(\{b_1, e_1\}) = \{b_2, e_2\}$$

Therefore, a view on a range **is itself a range**, typically a modification of the original range.

This allows, as we will see, to concatenate views.




## Predefined operators on ranges (views)

The standard library has a long list of predefined tools that operate on ranges, a list is [here](#). As usual let's start from an example<sup>2</sup>. A function that prints only odd numbers of a range of int

```
#include <ranges>
using namespace std::ranges; // normally you don't do this!
void printOdd(forward_range auto& r) // forward range is enough{
    filter_view v {r, [](int x) { return x%2 == 1; } }; // I create a filter
    cout << "odd_numbers:_"
    for (int x : v)
        cout << x << '_';
}
```

`std::ranges::filter_view(r, pred)` is a **view** that takes a range `r` and a predicate `pred`, returning a range that filters elements satisfying the predicate.

---

<sup>2</sup>Taken from Stroustrup book "A tour of C++ (11 Ed)" 

# Composing Views

Views can be composed:

```
void printFirstHundredOdds(forward_range auto& r)
{
    using namespace std::ranges;
    filter_view v{r, [](int x) { return x%2==1; } }; // view (only) odd
    take_view tv {v, 100 }; // view at most 100 element from v
    cout << "first_100_odd_numbers:_"
    for (int x : tv)
        cout << x << '_';
}
```

or using pipelines, **simpler and nicer!**:

```
void printFirstHundredOdds(forward_range auto& r)
{
    using namespace std::views; //note this!
    auto odd = [](int x) { return x % 2; };
    for (int x : r | views::filter(odd) | views::take(100))
        cout << x << '_';
}
```

Here `r | views::filter(odd) | views::take(100)` is a range comprised by piping a range with two **RangeAdaptorObjects**. See next slide with details.

# Composing views with pipelines

Most views named `std::ranges::name_view` have a corresponding `std::views::name` which is what is called a **range adaptor object**. The difference is that `std::ranges::name_view` is a type (normally a **struct**), while `std::views::name` is a **functor** returning a `std::ranges::name_view` object. Only the latter can be used in a pipe.

From the C++ standard:

*if C is a range adaptor object and R is a viewable\_range, these two expressions are equivalent: C(R) and R | C.*

Being functors they can be piped (thanks to the overload of the | operator ) and used in contexts where you need functor.

# Composing views with pipelines

Eliminate trailing spaces from a string and convert it to upper case

```
#include <string>
#include <cctype>
std::string trimUp(std::string const & text)
{
    using namespace std::views;
    // create the view
    auto conv = text | drop_while(std::isspace) | transform (std::toupper);
    // apply the view
    return std::string temp{conv.begin(), conv.end()};
}
```

Note that a view has the methods `begin()` and `end()` (it's a range!);

```
int main(){
    const std::string text { "   Hello World" };
    auto trimmedText=trimUp(text);
    std::cout << std::quoted(trimmedText) << '\n';
}
```

The code outputs

"HELLO WORLD".

## A last example

You can use non range based standard algorithm with the new views and range adaptors. Compute the L2norm of a vector, eliminating elements too big (outliers):

```
#include <ranges>
#include <vector>
#include <numeric>
#include <cmath>

double norm2Cutted(std::vector<double> const & v, double cut){
    using namespace std::ranges::views;
    auto large = [cut](double x){return std::abs(x)<cut;};
    auto square = [] (double x){return x*x;};
    auto conv = v | filter(large) | transform(square);
    return std::sqrt{std::reduce(conv.begin(), conv.end())};
}

#include <iostream>
int main() {
    std::vector<double> v{1., -4.8, 500.};
    auto vc = norm2Cutted(v, 50.);
    std::cout << vc << '\n';}
```

The code outputs 4.90306.

# Conclusions

The new ranges and views, and related algorithms are a great improvement. The design is more flexible and extendable compared to the pre-c++20 iterators and algorithms. The old stuff is however still valid and indeed it can be integrated with the new goodies.

The porting of the standard algorithms to the ranges version is still ongoing, more will be ported in c++23!

## std::invoke

Let's now talk of another utility of c++ since c++17: `std::invoke()`. It provides a unique interface for calling ANY callable object. Let's understand why with an example

```
template <typename T, typename F>
auto sumup(T& container, F f) {
    decltype(container[0]) res=0;
    for (auto const & elem : container) res+=f(elem);
    return res;
}
```

This function calls `f` on all elements of a container. It works on any callable objects, but it **fails on member functions**: this code gives a compiler error

```
std::vector<Triangle> tv;
auto x=sumup(tv,&Triangle::area);
```

since the way to call of a member function is different: `(elem.*f)()` and not `f(elem)`.

## Solution with `std::invoke`

```
template <typename T, typename F>
auto sumup(T& container, F f) {
    decltype(container[0]) res=0;
    for (auto const & elem : container) res+=std::invoke(f,elem);
    return res;
}
```

In general `std::invoke(f,args...)` returns the call `f(arg1, arg2,...)` if `f` is a function object, while if `f` is a member function is equivalent to `(arg1.*f)(arg2,...)`