lecture11

what:functions and lambdas
what more:
functions and lambdas
algorithms
Ranges and views

引入：find()的泛型编程（general way）

How can we make `find` even more general!?

- Our `find` searches for the first occurrence of `value` in a container
- What if we wanted to find the first occurrence of:
  - A vowel in a `string`?
  - A prime number in a `vector<int>`?
  - A number divisible by 5 in a `set<int>`?

functions and lambdas

首先引入一个新概念：predicate（谓词）

> 2025Fall-11-FunctionsAndLambdas, p.18
>
> Definition: A predicate is a boolean-valued function

简单来说这个谓词就是类似于那种匿名函数里面判断的小函数

举个例子：
example：Unary and Binary

Predicate Examples

Unary

```
bool isVowel(char c) {
    c = toupper(c);
    return c == 'A' || c == 'E' ||
           c == 'I' || c == 'O' ||
           c == 'U';
}

bool isPrime(size_t n) {
    if (n < 2) return false;
    for (auto i = 3; i<=sqrt(n); i++)
        if (n % i == 0) return false;
    return true;
}
```

Binary

```
bool isLessThan(int x, int y) {
    return x < y;
}

bool isDivisible(int n, int d) {
    return n % d == 0;
}
```

其实引入谓词，我们很快就引入了匿名函数了

2025Fall-11-FunctionsAndLambdas, p.21

> Key Idea: We need to pass a predicate to a function

准备升级find()函数了
原版的find()函数：

Modifying our `find` function

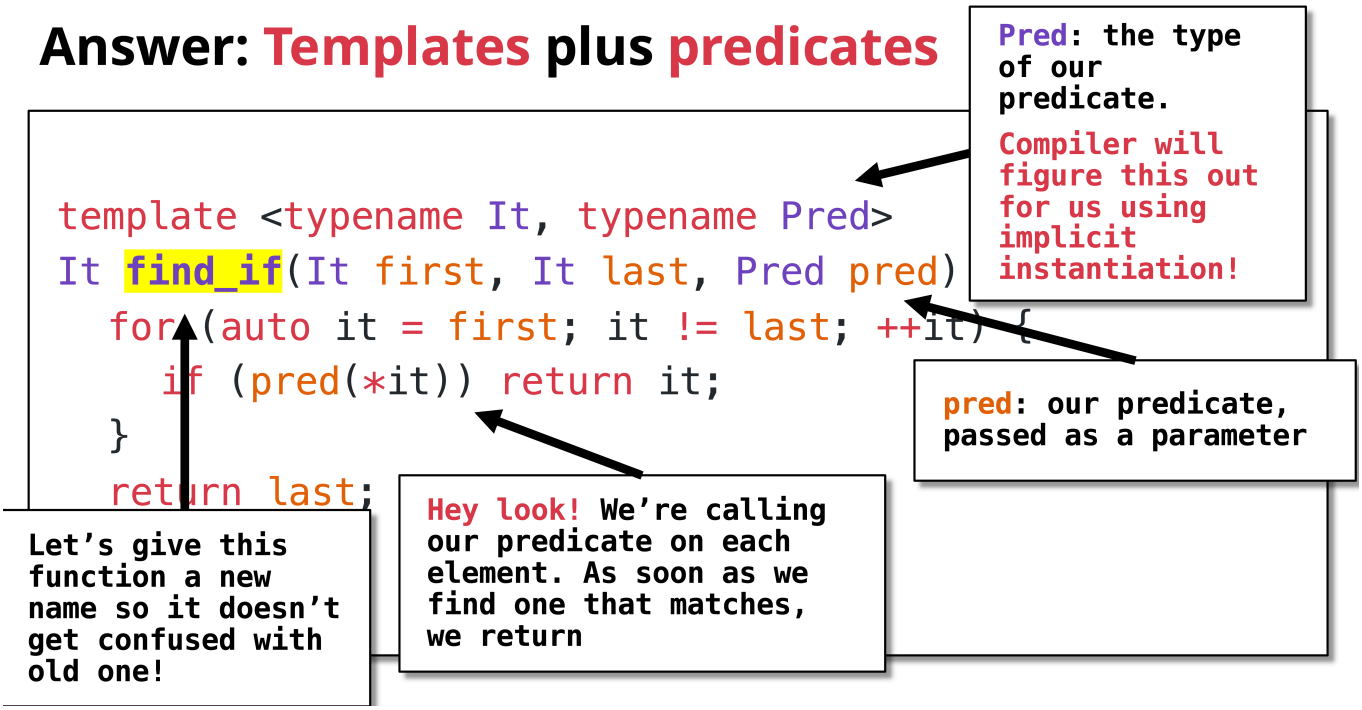
```
template <typename It, typename T>
It find(It first, It last, const T& value) {
  for (auto it = first; it != last; ++it) {
    if (*it == value) return it;
  }
  return last;
}
```

This condition worked for finding a specific value, but it's too specific.

How can we modify it to handle a general condition?

find_if()函数：引入了谓词

Answer: Templates plus predicates



find_if()函数的应用：

Using our find_if function

```
bool isPrime(size_t n) {
   if (n < 2) return false;
   for (size_t i = 3; i <= std::sqrt(n); i+
      if (n % i == 0) return false;
   return true;
}

std::vector<int> ints = {1, 0, 6};
auto it = find_if(ints.begin(), ints.end(), isPrime);
assert(it == ints.end());
```

You: "What type is this!!?"
Compiler: "I gottttchuuu man"

这个其实就是提到了用户自己定义的行为（谓词）的意义

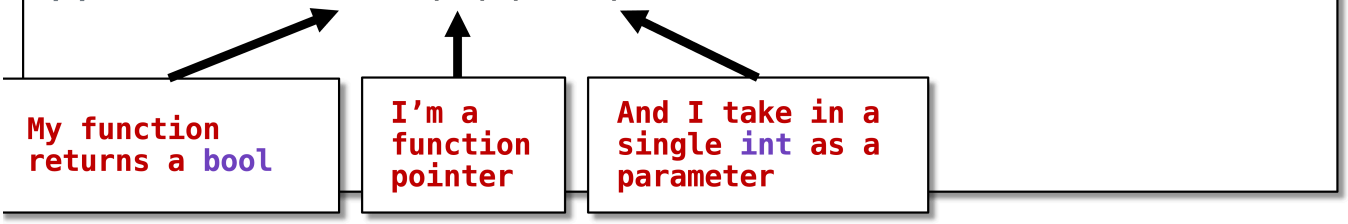
2025Fall-11-FunctionsAndLambdas, p.32

> Passing functions allows us to generalize an algorithm with user-defined behaviour

观察到这个pred,不属于确定性的string int, 这是一种函数指针

Pred is a function pointer

```
find_if(corlys.begin(), corlys.end(), isVowel);
// Pred = bool(*)(char)

find_if(ints.begin(), ints.end(), isPrime);
// Pred = bool(*)(int)
```



As we'll see shortly, a function pointer is ***just one*** of the things we can pass to `find_if`

进一步扩展，functions pointers有点繁琐
function pointer指向固定函数，很难泛化

Function pointers generalize poorly

Consider that we want to find a number less than **N** in a vector

```
bool lessThan5(int x) { return x < 5; }
bool lessThan6(int x) { return x < 6; }
bool lessThan7(int x) { return x < 7; }

find_if(begin, end, lessThan5);
find_if(begin, end, lessThan6);
find_if(begin, end, lessThan7);
```

1.不能改变比较数字
2.function pointer不能传两个参数，只能传一个谓词

引入匿名函数：比谓词更强大，那个n可以自己定义了

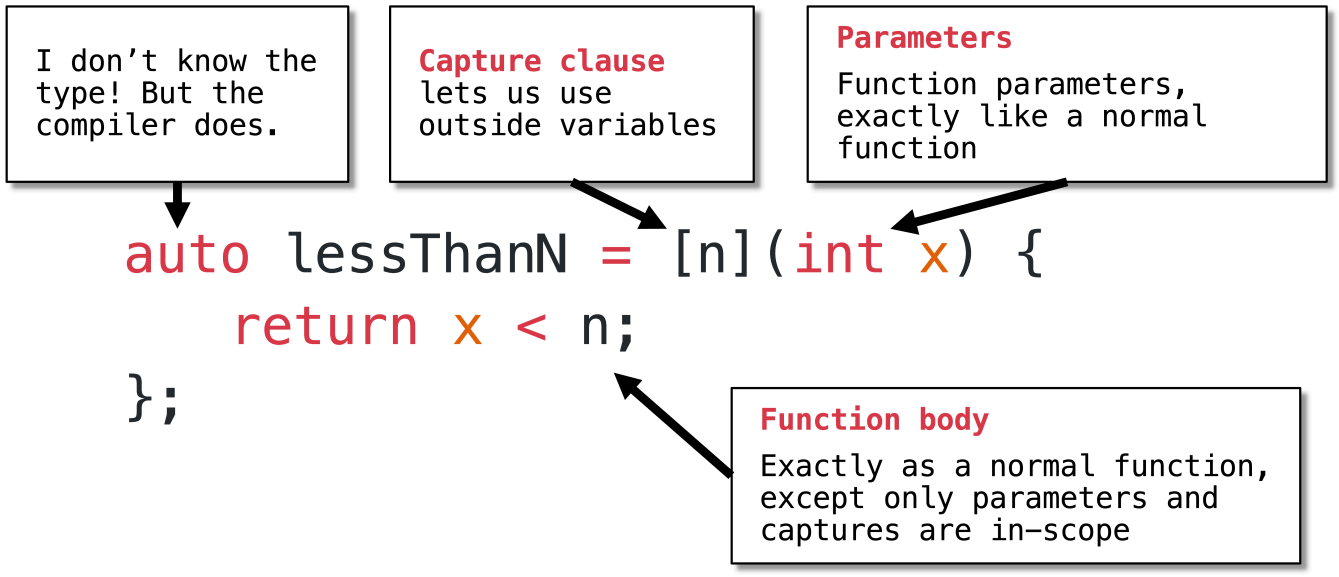
Introducing... lambda functions

Lambda functions are functions that capture state from an enclosing scope

```
int n;
std::cin >> n;

auto lessThanN = [n](int x) { return x < n; };

find_if(begin, end, lessThanN); // 😎 😎
```

具体语法解析：

Lambda Syntax



不同的捕获说明：

A note on captures

```
auto lambda = [capture-values](arguments) {
   return expression;
}

[x](arguments)     // captures x by value (makes a copy)
[x&](arguments)    // captures x by reference
[x, y](arguments)  // captures x, y by value
[&](arguments)     // captures everything by reference
[&, x](arguments)  // captures everything except x by reference
[=](arguments)     // captures everything by value
```

[]是不捕获
[x]值捕获
[&x]引用捕获

`auto` parameters are shorthand for templates

```
auto lessThanN = [n](auto x) {
  return x < n;
};
```

```
template <typename T>
auto lessThanN = [n](T x) {
  return x < n;
};
```

This is true wherever you see an `auto` parameter, not just in lambda functions!

Uses implicit instantiation! Compiler figures out types when function is called

implict instantaition 隐式实例化

functors（仿函数）不是functions

2025Fall-11-FunctionsAndLambdas, p.49

> Definition: A functor is any object that defines an operator() In English: an object that acts like a function

举个例子：greater与hash

An example of a functor: std::greater<T>

```
template <typename T>
struct std::greater {
  bool operator()(const T& a, const T& b) const {
    return a > b;
  }
};

std::greater<int> g;
g(1, 2); // false
```

Hmm.. Seems like a function

Another STL functor: std::hash<T>

```
template <>
struct std::hash<MyType>{
  size_t operator()(const MyType& v) const {
    // Crazy, theoretically rigorous hash function
    // approved by 7 PhDs and Donald Knuth goes here
    return ...;
  }
};

MyType m;
std::hash<MyType> hash_fn;
hash_fn(m); // 125123201 (for example)
```

Aside: This syntax is called a *template specialization* for type MyType

Hint hint: This is also *one* of the ways to create a hash function for a custom type

其实functors很像function，但是区别是functors是用类封装好的函数，有private的部分，就是也

有成员变量
state就是成员的意思

> 2025Fall-11-FunctionsAndLambdas, p.52
>
> > Since a functor is an object, it can have state

举例：

Functors can have state!

```
struct my_functor {
   bool operator()(int a) const {
      return a * value;
   }

   int value;
};

my_functor f;
f.value = 5;
f(10); // 50
```

下面将要讲述lambda构造的底层代码

> 2025Fall-11-FunctionsAndLambdas, p.55
>
> > When you use a lambda, a functor type is generated

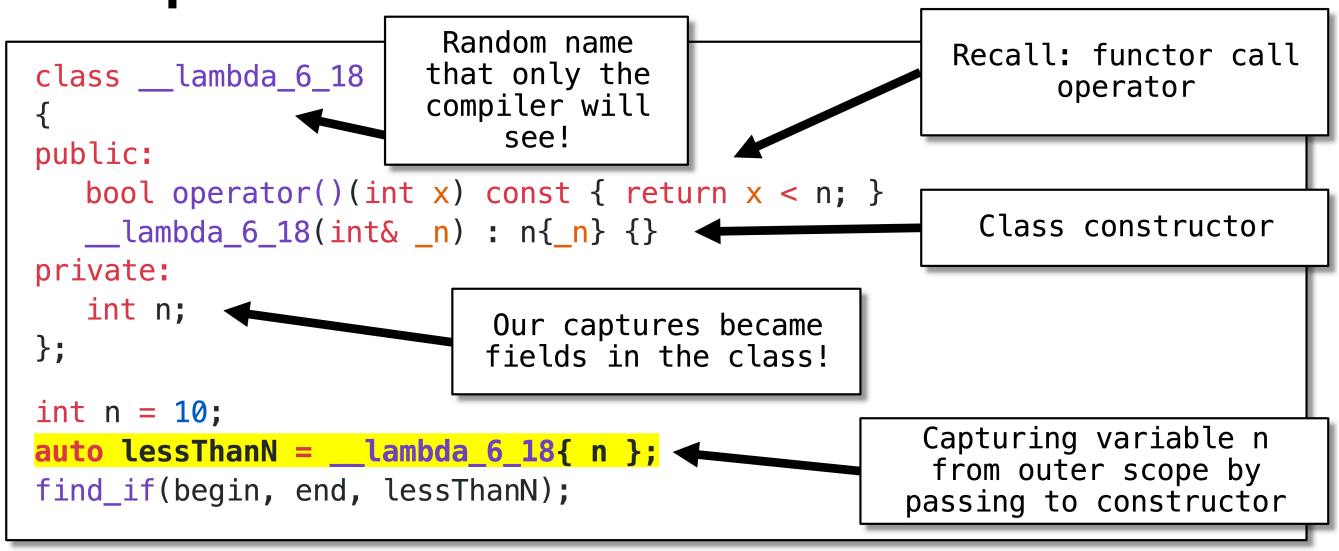
先看个原本的code

This code...

```
int n = 10;
auto lessThanN = [n](int x) { return x < n; };
find_if(begin, end, lessThanN);
```

lambda的内部实现：

...is equivalent to this code!



If you are curious about this stuff, check out https://cppinsights.io/!

如果没有lambda，我们还得写这么多类来封装，现在不用了

总结一下：

Functions & Lambdas Recap

- Use functions/lambdas to pass around **behaviour** as variables
- Aside: `std::function` is an overarching type for functions/lambdas
  - Any functor/lambda/function pointer can be cast to it
  - It is a bit slower
  - I usually use auto/templates and don't worry about the types!

```
std::function<bool(int, int)> less = std::less<int>{};
std::function<bool(char)> vowel = isVowel;
std::function<int(int)> twice = [](int x) { return x * 2; };
```

我们可以用function关键字来定义写函数（不常用）
实际上用auto和templates就行了

Algorithms

algorithms是stl库里非常庞大的存在，这里列举一些常见函数

`<algorithm>` is a collection of template functions

```
std::count_if(InputIt first, InputIt last, UnaryPred p);
  How many elements in [first, last] match predicate p?

std::sort(RandomIt first, RandomIt last, Compare comp);
Sorts the elements in [first, last) according to comparison comp

std::max_element(ForwardIt first, ForwardIt last, Compare comp);
Finds the maximum element in [first, last] according to comparison comp
```

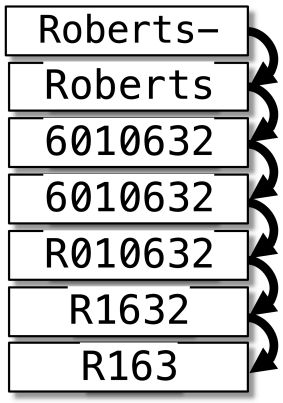
<algorithm> functions operate on iterators

```
std::copy_if(InputIt r1, InputIt r2, OutputIt o, UnaryPred p);
Copy the only elements in [r1, r2) into o which meet predicate p

std::transform(ForwardIt1 r1, ForwardIt1 r2, ForwardIt2 o, UnaryOp op);
Apply op to each element in [r1, r2), writing a new sequence into o

std::unique_copy(InputIt i1, InputIt i2, OutputIt o, BinaryPred p);
Remove consecutive duplicates from [r1, r2), writing new sequence into o
```

我们将用algorithm来做个语音识别的例子
要求：

How do we implement soundex?

1. Given a string s, extract the letters from s
2. Replace each letter with its soundex encoding
3. Coalesce adjacent duplicates (222025 becomes 2025)
4. Replace first digit with the uppercase first letter of s
5. Discard any zeros from the code
6. Make the code exactly length 4 (truncate or zero-pad)



Digit	represents the letters
0	A E I O U H W Y
1	B F P V
2	C G J K Q S X Z
3	D T
4	L
5	M N
6	R

1.字符串只保留字母
2.每个字母转换成soundex编码
3.合并相邻相同数字
4.把第一个数字换回字母，并且大写
5.去0
6.定长为4，不够的补0就行

准备：
stl库

Only need 3 functions from <algorithm>!

```
std::copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPred p);
```
Only copies the elements for which p is true.
d_first cannot overlap with [first, last).

```
std::transform(InputIt first, InputIt last, OutputIt d_first, UnaryOp p);
```
The unary operation p is applied to elements between
[first, last) and is stored in a range starting from d_first.

```
std::unique_copy(InputIt first, InputIt last, OutputIt d_first);
```
Copies elements from [first, last) to the range d_first in such a way
that there are no consecutive equal elements.

具体步骤：

```
std::string letters;
std::copy_if(s.begin(), s.end(), std::back_inserter(letters), ::isalpha);
```

Global namespace function to find if the provided char is in the alphabet.

2. Replace each letter with its soundex encoding

```
std::transform(letters.begin(), letters.end(), letters.begin(), soundexEncode);
```

```
static char soundexEncode(char c)
{
  static const std::map<char, char> encoding = {
    {'A', '0'}, {'E', '0'}, {'I', '0'}, {'O', '0'}, {'U', '0'}, {'H', '0'}, {'W', '0'}, {'Y', '0'},
    {'B', '1'}, {'F', '1'}, {'P', '1'}, {'V', '1'},
    {'C', '2'}, {'G', '2'}, {'J', '2'}, {'K', '2'}, {'Q', '2'}, {'S', '2'}, {'X', '2'}, {'Z', '2'},
    {'D', '3'}, {'T', '3'},
    {'L', '4'},
    {'M', '5'}, {'N', '5'},
    {'R', '6'}
  };
  return encoding.at(std::toupper(c));
}
```

3. Coalesce adjacent duplicates (222025 becomes 2025)

```
std::string unique;
std::unique_copy(letters.begin(), letters.end(), std::back_inserter(unique));
```

4. Replace first digit with the uppercase first letter of s

```
char first_letter = letters[0];
unique[0] = std::toupper(first_letter);
```

5. Discard any zeros from the code

```
std::string no_zeros;
std::copy_if(unique.begin(), unique.end(), std::back_inserter(no_zeros), notZero);
```

```
static bool notZero(char c)
{
    return c != '0';
}
```

6. Make the code exactly length 4 (truncate or zero-pad)

```
no_zeros += "0000";
return no_zeros.substr(0, 4);
```

最终结果：

Soundex has been implemented!

```
std::string soundex(const std::string& s)
{
    std::string letters;
    std::copy_if(s.begin(), s.end(), std::back_inserter(letters), ::isalpha);

    std::transform(letters.begin(), letters.end(), letters.begin(), soundexEncode);

    std::string unique;
    std::unique_copy(letters.begin(), letters.end(), std::back_inserter(unique));

    char first_letter = letters[0];
    unique[0] = std::toupper(first_letter);

    std::string no_zeros;
    std::copy_if(unique.begin(), unique.end(), std::back_inserter(no_zeros), notZero);

    no_zeros += "0000";
    return no_zeros.substr(0, 4);
}
```

引入range（c++20）
range and view（非常重要）

2025Fall-11-FunctionsAndLambdas, p.86

> Ranges are a new version of the STL

range的定义：

> 2025Fall-11-FunctionsAndLambdas, p.87
>
> Definition: A range is anything with a begin and end

所有用begin 和end的都可以用range写

先回到find()

Recall: why did we pass iterators to `find`?

It allows us to find in a subrange! But most of the time, we don't need to.

```
int main() {
  std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};
  auto it = std::find(v.begin(), v.end(), 'c');
}
```

Do we really care about iterators here? I just wanted to search the entire container!

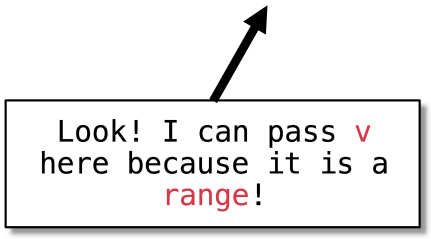
range实际上是个std库里的小库

像这样：

Range algorithms operate on ranges

STD ranges provides new versions of <algorithm> for ranges

```
int main() {
  std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};
  auto it = std::ranges::find(v, 'c');
}
```



再举个例子：

Range algorithms operate on ranges

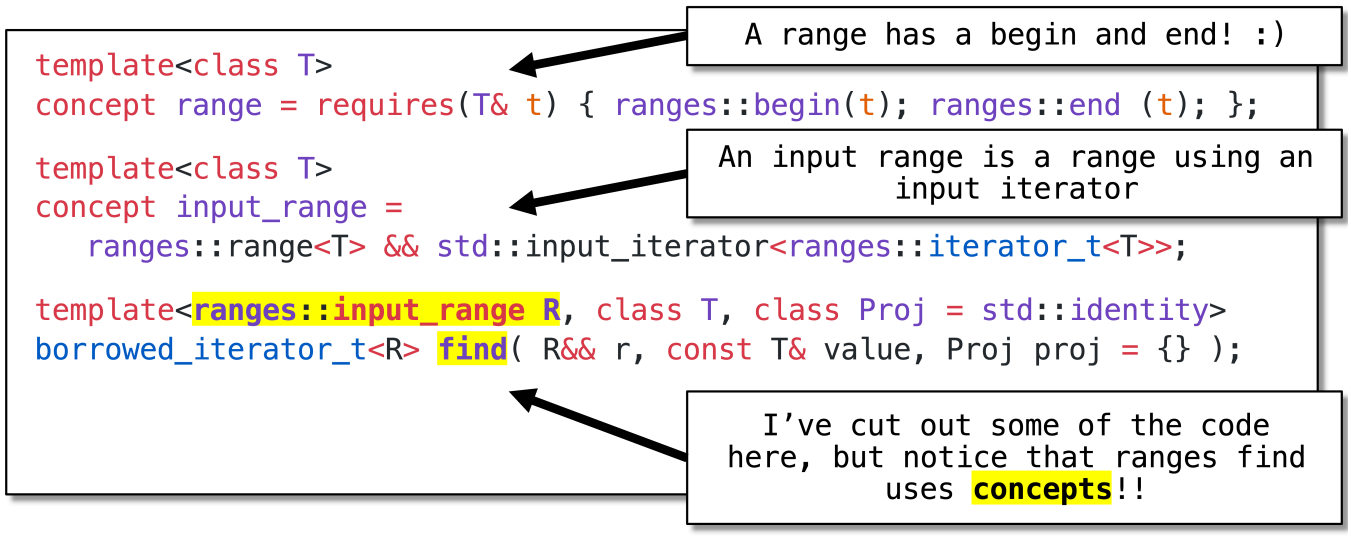
We can still work with iterators if we need to

```
int main() {
  std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};

  // Search from 'b' to 'd'
  auto first = v.begin() + 1;
  auto last = v.end() - 1;
  auto it = std::ranges::find(first, last, 'c');
}
```

range是受约束的（concept）

Range algorithms are constrained

That just means they make use of the new STL concepts! Remember them?



view

> 2025Fall-11-FunctionsAndLambdas, p.97
>
> Views: a way to compose algorithms

定义：

> 2025Fall-11-FunctionsAndLambdas, p.98
>
> Definition: A view is a range that lazily adapts another range

views有composable的能力，可以组合多种不同的range

先看之前旧版本的stl

Filter and transform in the old STL

This code is a bit awkward in the current STL

```
std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};

// Filter -- Get only the vowels
std::vector<char> f;
std::copy_if(v.begin(), v.end(), std::back_inserter(f), isVowel);

// Transform -- Convert to uppercase
std::vector<char> t;
std::transform(f.begin(), f.end(), std::back_inserter(t), toupper);

// { 'A', 'E' }
```

可以用view组合

Filter and transform with views!

A **view** is a range that lazily transforms its underlying range, one element at a time

```
std::vector<char> letters = {'a', 'b', 'c', 'd', 'e'};

auto f = std::ranges::views::filter(letters, isVowel);
auto t = std::ranges::views::transform(f, toupper);

auto vowelUpper = std::ranges::to<std::vector<char>>(t);
```

Views are composable

```
auto f = std::ranges::views::filter(letters, isVowel);
// f is a view! It takes an underlying range letters
// and yields a new range with only vowels!

auto t = std::ranges::views::transform(f, toupper);
// t is a view! It takes an underlying range f
// and yields a new range with uppercase chars!

auto vowelUpper = std::ranges::to<std::vector<char>>(t);
// Here we materialize the view into a vector!
// Nothing actually happens until this line!
```

结合操作符 | 更有条理

We can chain views together use operator |

```
std::vector<char> letters = {'a','b','c','d','e'};
std::vector<char> upperVowel = letters
   | std::ranges::views::filter(isVowel)
   | std::ranges::views::transform(toupper)
   | std::ranges::to<std::vector<char>>();

// upperVowel = { 'A', 'E' }
```

range还可以和sort组合

Remember: range algorithms are eager

`std::ranges` are a reskin of the old STL algorithms

```
// This actually sorts vec, RIGHT NOWWW!!!!
std::ranges::sort(v);
```

view是一种惰性的

Remember: views are lazy

`std::ranges::views` are a lazy way of composing algorithms

```
auto view = letters
   | std::ranges::views::filter(isVowel)
   | std::ranges::views::transform(toupper);

std::vector<char> upperVowel =
   std::ranges::to<std::vector<char>>(view);
```

views像是python里的generators

Pro tip: Views are like Python generators

This code in C++ works *exactly the same* as this Python code

```cpp
auto view = letters
   | std::ranges::views::filter(isVowel)
   | std::ranges::views::transform(toupper);
auto upperVowel = std::ranges::to<std::vector<char>>(view);
```

```python
view = (l for l in letters if isVowel(l))    # Lazy evaluation
view = (l.upper() for l in view)             # Lazy evaluation
upperVowel = list(view)
```

Recap：

Ranges and view recap

- Why you might like ranges/views?
  - ✅ Worry less about iterators
  - ✅ Constrained algorithms mean better error messages
  - ✅ Super readable, functional syntax
- Why you might dislike ranges/views?
  - ❌ They are extremely new, not fully feature complete yet
  - ❌ Lack of compiler support
  - ❌ Loss of performance compared to hand-coded version
  - ▪ ℹ️ For more info, see The Terrible Problem of Incrementing a Smart Iterator

可以这样实现soundex

Soundex: C++26?

Once views are fully implemented, our Soundex code might look like this

```
namespace rng = std::ranges;
namespace rv = std::ranges::views;

auto ch = *rng::find_if(s, isalpha);              // Get first letter
auto sx = s  | rv::filter(isalpha)              // Discard non-letters
             | rv::transform(soundexEncode)     // Encode letters
             | rv::unique                       // Remove duplicates
             | rv::filter(notZero)              // Remove zeros
             | rv::concat("0000")               // Ensure length >= 4
             | rv::drop(1)                      // Skip first digit
             | rv::take(3)                      // Take next three
             | rng::to<std::string>();          // Convert to string
return toupper(ch) + v;
```