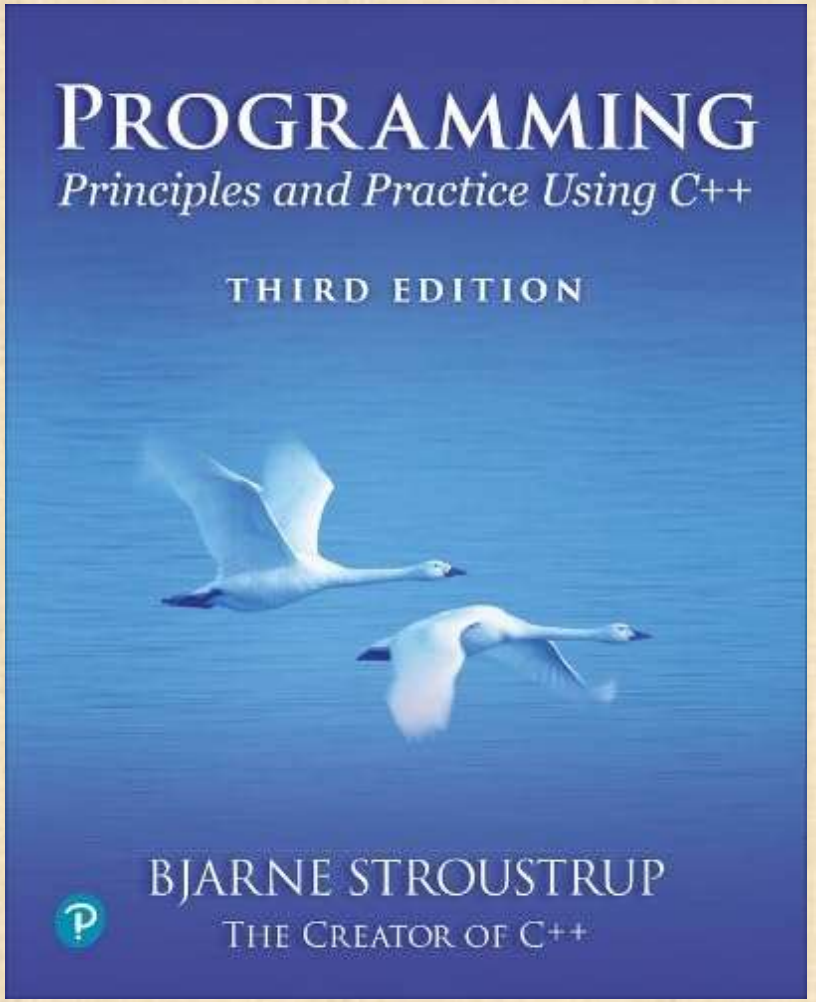
Chapter 21 - Algorithms



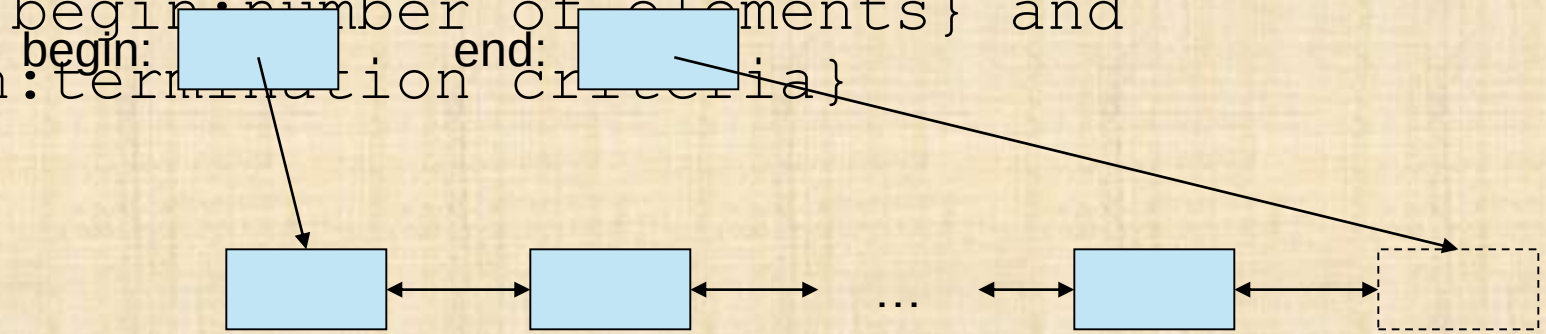
In theory, practice is simple.
– Trygve Reenskaug

Overview

- Standard-library algorithms
- Function objects
  - Lambdas
- Numerical algorithms
- Copying
- Sorting and searching

Basic model of a sequence/range

- A pair of iterators defines a half-open sequence [begin:end)
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)
- Also {begin:number of elements} and {begin:termination criteria}



- An iterator is a type that supports the "iterator operations" of
  - ++ Point to the next element
  - * Get the element value
  - == Does this iterator point to the same element as that iterator?

Selected standard algorithms

- **p=find(b,e,v)** **p** points to the first occurrence of **v** in [**b:e**)
- **p=find_if(b,e,p)** **p** points to the first element x in [**b:e**) so that **p(x)** is true
- **x=count(b,e,v)** **x** is the number of occurrences of **v** in [**b:e**).
- **x=count_if(b,e,p)** **x** is the number of elements in [**b:e**) so that **p(x)** is true
- **sort(b,e)** Sort [**b:e**) using **<**
- **sort(b,e,p)** Sort [**b:e**) using **p**
- **x=is_sorted(b,e)** If [**b:e**) is sorted x is true
- **b2=copy(b,e,b2)** Copy [**b:e**) to [**b2:b2+(e-b)**)
- **b2=move(b,e,b2)** Move [**b:e**) to **[\{b2\}:\{b2+(e-b)\})**
- **b2=uninitialized_copy(b,e,b2)** Copy [**b:e**) to an uninitialized **[b2:b2+(e-b))**
- **b2=unique_copy(b,e,b2)** Copy [**b:e**) to **[\{b2\}:\{b2+(e-b)\});** don't copy adjacent duplicates.

- For **move(), copy(), unitialized_copy(),** and other

Selected standard algorithms

- **e2=merge(b,e,b2,e2,r)** Merge two sorted sequences **[\{b\}:\{e\})** and [**b2:e2**) into [**r:r+(e-b)+(e2-b2)**)
- **[p,q]=equal_range(b,e,v)** [**p:q**) is the subsequence of the sorted range [**b:e**) with the value **v**,
  basically, a binary search for **v**
- **equal(b,e,b2)** Do all elements of **[\{b\}:\{e\})** and **[b2:b2+(e-b))** compare equal?
  The sequences must be sorted.
- **x=accumulate(b,e,i)** **x** is the sum of **i** and the elements of **[\{b\}:\{e\}).**
- **r=max(x,y)** **r** is a reference to the larger of **x** and **y**
- **p=max_element(b,e)** **p** points to the largest element in [**b:e**)
- **iota(b,e,i)** [**b:e**) becomes the sequence **i, i+1, i+2, ...**

- Usually, a standard algorithm is faster than our handcrafted code
- Knowing relevant standard library algorithms saves us a lot of

The simplest algorithm: find()

- Traversing a sequence doing something to each element

```
template<input_iterator In, equality_comparable<In::value_type> T>
In find(In first, In last, const T& val)                    // find the first element in [first,last) that equals val
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}

if (auto p = find(begin(v), end(v),x); p!=v.end()) {
    // ... we found x in v; we can use *p ...
}
else {
    // ... no x in v; don't dereference p ...
}
//  ...
```

Yes, we can use an argument as a variable

Condition in if-statement

The simplest algorithm: find()

- We could almost equivalently have written

```
auto p = find(begin(v), end(v),x);
if (p!=v.end()) {
   // ... we found x in v; we can use *p ...
}
else {
   // ... no x in v; don't dereference p ...
}
//  ...
```

- But now p is in scope after the if-statement
  - Do we want that?
  - Sometimes we do, and sometimes the scope ends right after the **if**-statement

The simplest algorithm: find()

- Traversing a range doing something to each element

```
template<ranges::range R, equality_comparable<R::value_type> T>
R::iterator find(R r, const T& val)          // find the first element in r that equals val
{
    return find(r.begin, r.end(), val);
}

if (auto p = find(v,x); p!=v.end()) {
    // ... we found x in v; we can use *p ...
}
else {
    // ... no x in v; don't dereference p ...
}
//  ...
```

The simplest algorithm: ranges::find()

- Traversing a range doing something to each element

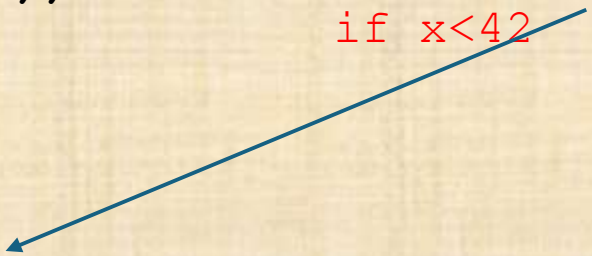
```
if (auto p = ranges::find(v,x); p!=v.end()) {
    // ... we found x in v; we can use *p ...
}
else {
    // ... no x in v; don't dereference p ...
}
//  ...
```

find_if()

- Often, we don't look for a value but for something that meets a criteria

```
template<input_iterator In, predicate<In::value_type> Pred>
In find_if(In first, In last, Pred pred)
{
  while (first!=last && !pred(*first))
        ++first;
  return first;
}
```

A predicate: true if x<42



```
if (auto p = find_if(begin(v), end(v), Greater_than{42}); p!=v.end()) {
      // ... we found something greater than 42 in v; we can use *p ...
}
else {
      // ... no value greater than 42 in v; don't dereference p ...
}
```

Predicates

- A predicate I something that returns **true** if a criteria is met and **false** otherwise
  - A function

```
template<typename T>
bool greater(const T& x, const T& y) {return x>y; }
```

    - Obvious, but clumbsy where we want to compare many elements against the same value
  - A function object

```
template<typename T>
struct Greater_than {
        T val;
        Greater_than(const  T& x) : val(x) {}
        bool operator()(const T& x) { return x>val; }
};
```

  - A lambda expression

```
template<typename T>
[](const T& x} {return x>42;}
```

    - Generates a function object (like **Greater_than**)

Lambda expressions

- A lambda is an expression, so we can use it as a function argument

```
if (auto p = find_if(begin(v), end(v),[](int x) {return x>42;}),
p!=v.end()) {
    // ... we found something greater than 42 in v; we can use *p
...
}
else {
    // ... no value greater than 42 in v; don't dereference p ...
}
```

- Or

```
if (auto p = ranges::find_if(v, [](int x) {return x>42;}),
p!=v.end()) {
    // ... we found something greater than 42 in v; we can use *p
...
}
```

Function objects

- A function object is an object that can be called like a function
  - A generalization of a function

```
class F {       // abstract example of a function object
    S s;                                    // state
public:
    F(const S& ss) :s(ss) { /* establish initial state*/ }
    T operator() (const S& ss) const
    {
            // do something with ss to s
            // return a value of type T (T is often void, bool, or S)
    }
    const S& state() const { return s; }                // reveal state
    void reset(const S& ss) { s = ss; }         // reset state
};
```

Lambdas

- A lambda
  - generates a function object
  - can be used wherever an expression is allowed
  - can carry values (since they are objects) yielding more compact code
  - Can access its enclosing scope
  - can be used to avoid defining a function in on place and then using it elsewhere (yielding more compact code)
  - Should not be used where a named function yields clearer and less repetitive code
  - Keep lambdas short and simple

- An abstract lambda: **`[c](arg){ code }`** becomes a function object "F"

Values to be captured
Becomes F's member variables,
Initialized by F's constructor

Arguments for F's operator()()

Code to be executed.
Becomes the body of F's operator

Lambda usage examples

```
struct Record {
      string name;              // standard string for ease of use
      char addr[24];            // old style to match database layout
      //  ...
};
vector<Record> vr;


ranges::sort(vr, [](const Record& a, const Record& b) { return a.name < b.name; });
ranges::sort(vr, [](const Record& a, const Record& b) { return strncmp(a.addr, b.addr, 24) < 0; });
```

Lambda usage examples

- Lambdas are very flexible

```
vector<int> vi;
list<string> ls;
string s = "Hello!"
int answer = 42;
// …
auto p1 = ranges::find_if(vi, [](int a) { return a>31; });
if (p1!=v.end()) {
    // ... we found a int value > 31 ...
}
auto p2 = ranges::find_if(ls, [&](const char* a) { return a>s; });
if (p2!=ls.end()) {
    // ... we found a C-style string > s ...
}
auto p3 = ranges::find_if(vi, [=answer](double a) { return a>answer; });
if (p3!=v.end()) {
    // ... we found a double > answer ...
}
```

Lambdas

- Lambda captures :
  - **[]**: If there is nothing between **[** and **]**, the lambda is just like an ordinary function: it can access its arguments, its own local variables, and names in the global (namespace) scope
  - **[&]**: If we use **[&]**, the lambda can also use names from the scope in which it is defined, its enclosing scope. References to local objects are stored in the lambda object. Now the lambda acts like a local function
  - **[=]**: You can even ask to access copies of variables in the enclosing scope. Copies of the objects in the enclosing scope are stored in the lambda object

Numerical algorithms

- **x=accumulate(b,e,i)** Add a sequence of values; e.g., for **{a,b,c,d}** produce **i+a+b+c+d.**
  The type of the result **x** is the type of the initial value **i.**
- **x=inner_product(b,e,b2,i)** Multiply pairs of values from two sequences and sum the results;
  e.g., for **{a,b,c,d}** and **{e,f,g,h}** produce **i+a*e+b*f+c*g+d*h.**
  The type of the result **x** is the type of the initial value **i.**
- **r=partial_sum(b,e,r)** Produce the sequence of sums of the first \{n\} elements of [**b:e**);
  e.g., for {**a,b,c,d}** produce {**a, a+b, a+b+c, a+b+c+d}.**
- **r=adjacent_difference(b,e,b2,r)** Produce the sequence of differences between elements
  of [**b:e**); e.g., for **{a,b,c,d}** produce **{a,b-a,c-b,d-c}.**
- **iota(b,e,x)** Fill the range [**b:e**) with the values **x, x+1, x+2**

Simple **accumulate()**

```
template<input_iterator In, Number T>
T accumulate(In first, In last, T init)
{
      while (first!=last) {
            init = init + *first;
            ++first;
      }
      return init;
}

int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(*a), 0);      // yes, that's how
to get the size of a built-in array
```

Ranges numerical algorithms

- For lack of time, the ranges versions of the numerical algorithms didn't make it into C++20, but they are not hard to define. For example:

```
template<input_range R, output_iterator Out, typename T>
T accumulate(R r, Out oo, T init)
{
   return accumulate(begin(r),end(r),oo,init);
}
```

accumulate()

- The type of the result (the sum) is the type of the variable that **accumulate()** uses to hold the accumulator.
  - This gives a degree of flexibility that can be important.

```
void g(vector<int>& v)
{
    int s1 = accumulate(v, 0);          // sum into an int
    long sl = accumulate(v, long{0});       // sum the ints into a long
    double s2 = accumulate(v, 0.0);         // sum the ints into a double
}
```

Generalized **accumulate()**

```
template<input_iterator In, typename T, invocable<T,In::value_type> BinOp>
[[nodiscard]] // warn if the return value isn't used by a caller
T accumulate(In first, In last, T init, BinOp op)
{
        while (first!=last) {
                init = op(init, *first);
                ++first;
        }
        return init;
}

vector<double> a = { 1.1, 2.2, 3.3, 4.4 };
cout << accumulate(a.begin(),a.end(), 1.0, multiplies<double>());
```

accumulate()

```
struct Record {
	double unit_price;
	int units;		// number of units sold
	//  ...
};

double price(double v, const Record& r)
{
	return v + r.unit_price * r.units;		// extract values, calculate price, and accumulate
}

void f(const vector<Record>& vr)
{
	double total = accumulate(vr.begin(), vr.end(), 0.0, price);
	//  ...
```

Copy operations

b2=copy(b,e,b2) Copy [**b:e**) to [**b2:b2+(e-b)**)

b2=unique_copy(b,e,b2) Copy [**b:e**) to [**b2:b2+(e-b)**); suppress adjacent copies

b2=copy_if(b,e,b2,p) Copy [**b:e**) that meets the predicate p to [**b2:b2+(e-b)**)

```
template<input_iterator In, output_iterator Out>
Out copy(In first, In last, Out res)    // The simplest copy
{
        while (first!=last) {
                *res = *first;      // copy element
                ++res;
                ++first;
        }
        return res;
}
```

Make sure that there is enough sp
For the result in the target

copy_if()

```
template<input_iterator In, output_operator Out, predicate<In::value_type>
Pred>
Out copy_if(In first, In last, Out res, Pred p)       // copy elements that
fulfill the predicate p into res
{
    while (first!=last) {
       if (p(*first)) {
              *res = *first;
              ++res;
       }
       ++first;
    }
    return res;
}
```

```
void f(const vector<int>& v)  // copy all elements with a value
{
       vector<int> v2(v.size());
       ranges::copy_if(v, v2.begin(), [](int x){ return x>6;});
       //  ...
}
```

Sorting: **sort()**

```
template<random_access_iterator Ran>
void sort(Ran first, Ran last);


template<random_access_iterator Ran,
less_than_comparable<Ran::value_type> Cmp>
void sort(Ran first, Ran last, Cmp cmp);
```

Searching: **binary_search()**

```
template<random_access_range Ran, typename T>                    // compare using <
      bool binary_search(Ran r, const T& val);

template<random_access_range Ran, typename T,
            predicate<Ran::value_type,Ran::value_type> Cmp>      // compare using cmp
      bool binary_search(Ran r, const T& val, Cmp cmp);


void f(vector<string>& vs)              // vs is sorted
{
      if (ranges::binary_search(vs,"starfruit")) {
            // we have a starfruit (but we don't know where it is)
      }
      //  ...
}
```

Searching: **equal_range()**

- It is often useful to know where a matching term is:

```
template<forward_iterator Iter, typename T >
   equal_range( Iter first, Iter last, const T& value );                    //
compare using <

template<forward_iterator Iter, typename T,
         predicate<Ran::value_type,Ran::value_type> Cmp>                  //
compare using cmp
   equal_range( Iter first, Iter last, const T& value );
```

```
template<typename T>
void print same(const vector<T>& v, const T& x)
{
   for (const auto& x : ranges::equal_range(v,x))  // equal_range()
returns a sub-range
```

And there is more!

- Have a look at the "Software ideals and history" from PPP2:
  - https://www.stroustrup.com/PPP3_slides/22-ideals.pptx
- Also, you can find the more specialized "broadening the view" chapters on the Web:
  - Chapter 1: Computers, People, and Programming
  - Chapter 11: Customizing Input and Output
  - Chapter 22: Ideal and History
  - Chapter 23: Text Manipulation
  - Chapter 24: Numerics
  - Chapter 25: Embedded Systems Programming
  - Chapter 26: Testing
  - Chapter 27: The C Programming Language
- And lecture slides for those: https://www.stroustrup.com/PPP2slides.html