Imperial College London

Advanced Programming

Standard Template Library

Algorithms

Adriana Paluszny

[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

Imperial College London

Commonly used standard headers

`<iostream>`	I/O streams, cout, cin, …
`<fstream>`	file streams
`<algorithm>`	sort, copy, …
`<numeric>`	accumulate, inner_product, …
`<functional>`	function objects
`<string>`	
`<vector>`	
`<map>`	
`<unordered_map>`	hash table
`<list>`	
`<set>`	

Algorithms

- An STL-style algorithm
  - Takes one or more sequences
    - Usually as pairs of iterators
  - Takes one or more operations
    - Usually as function objects
    - Ordinary functions also work
  - Usually reports “failure” by returning the end of a sequence

Imperial College London

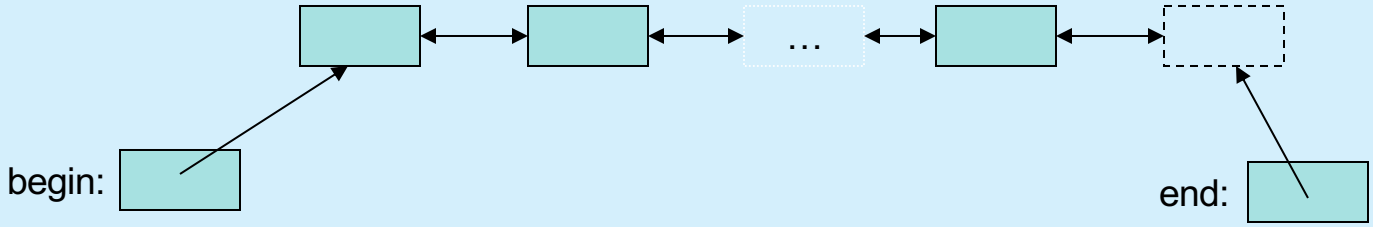
Some useful standard algorithms

`r=find(b,e,x)`	r points to the first occurrence of x in [b,e)
`r=find_if(b,e,p)`	r points to the first element x in [b,e) for which p(x)
`c=count(b,e,x)`	c is the number of occurrences of x in [b,e)
`c=count_if(b,e,p)`	c is the number of elements in [b,e) for which p(x)
`sort(b,e)`	sort [b,e) using <
`sort(b,e,p)`	sort [b,e) using p
`copy(b,e,b2)`	copy [b,e) to [b2,b2+(e-b)) there had better be enough space after b2
`unique_copy(b,e,b2)`	copy [b,e) to [b2,b2+(e-b)) but don’ t copy adjacent duplicates
`merge(b,e,b2,e2,r)`	merge two sorted sequence [b2,e2) and [b,e) into [r,r+(e-b)+(e2-b2))
`r=equal_range(b,e,x)`	r is the subsequence of [b,e) with the value x (basically a binary search for x)
`equal(b,e,b2)`	do all elements of [b,e) and [b2,b2+(e-b)) compare equal?

*coloured differently to improve readability

Imperial College London

The simplest algorithm: `find()`




```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val)
{
    while (first!=last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)  // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

We can ignore ("abstract away") the differences between containers

Imperial College London

find()

generic for both element type and container type

```
void f(vector<int>& v, int x)      // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(list<string>& v, string x)       // works for list of strings
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(set<double>& v, double x)        // works for set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

Simple algorithm: `find_if()`

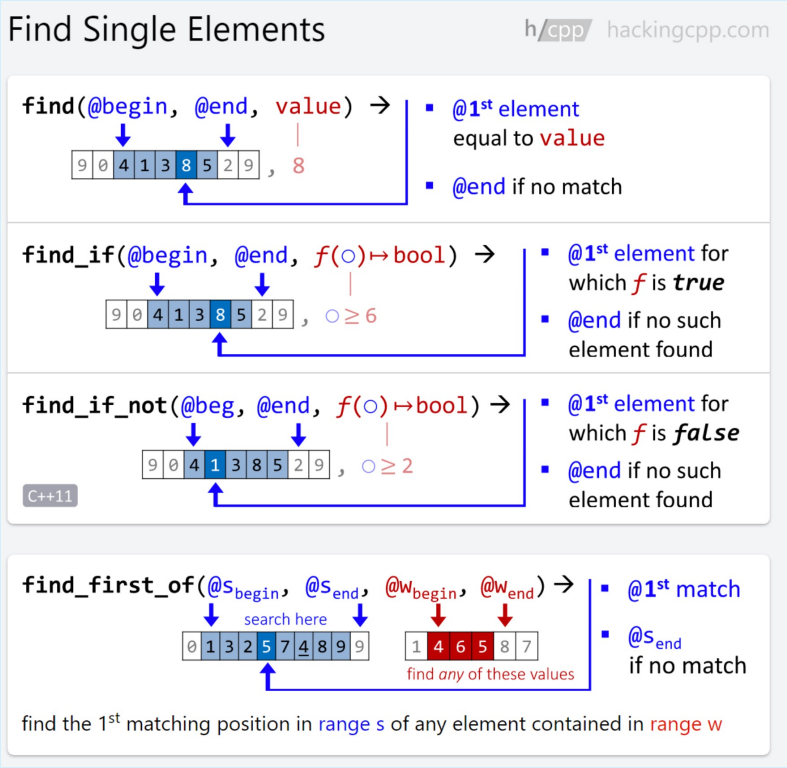
- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **`bool`**

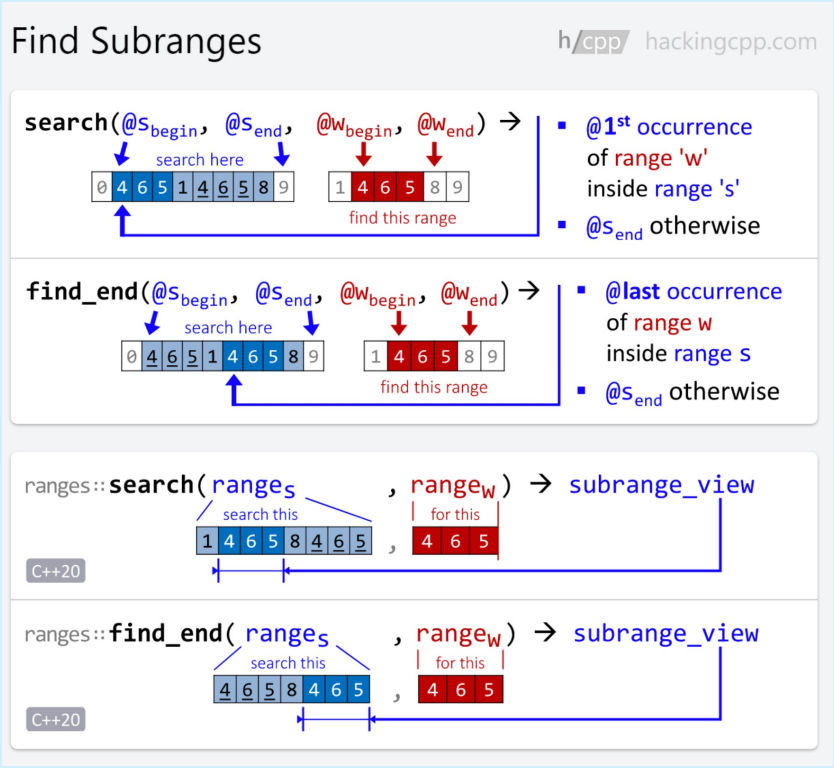
```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
   while (first!=last && !pred(*first)) ++first;
   return first;
}

void f(vector<int>& v)
{
   vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
   if (p!=v.end()) { /* we found an odd number */ }
   // …
}
```

A “predicate”

Imperial College London

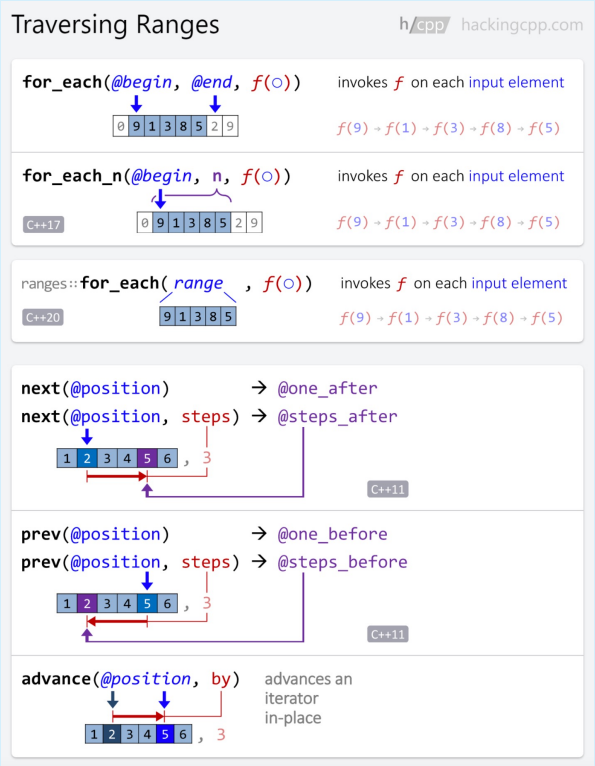
Find – visual summary

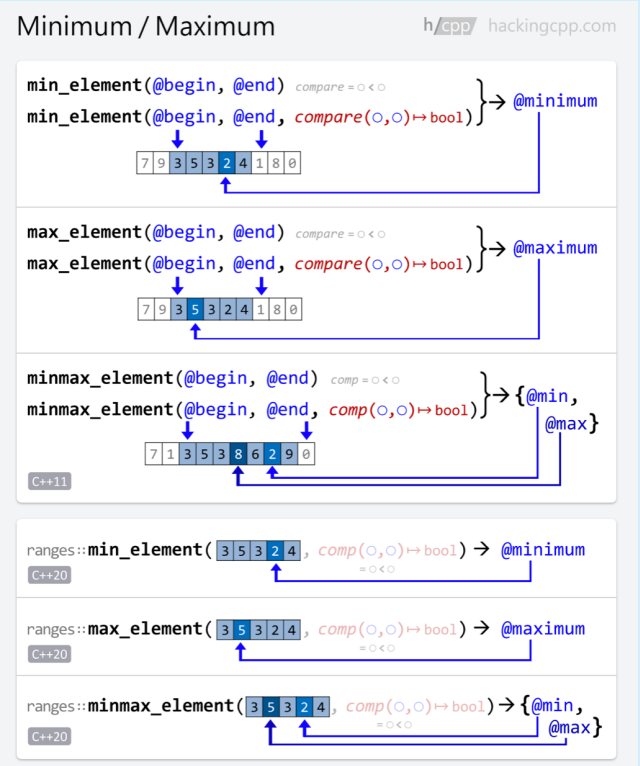




Imperial College London

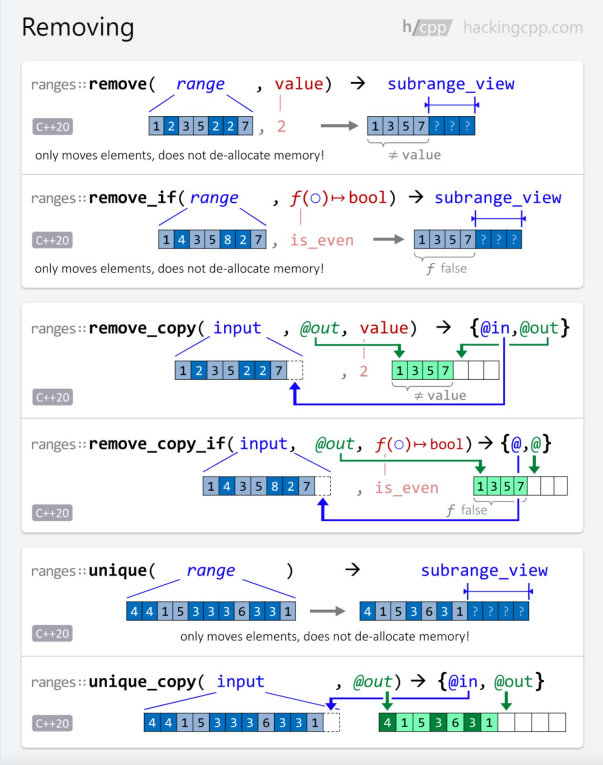
Minimum/Maximum and Traversing Ranges

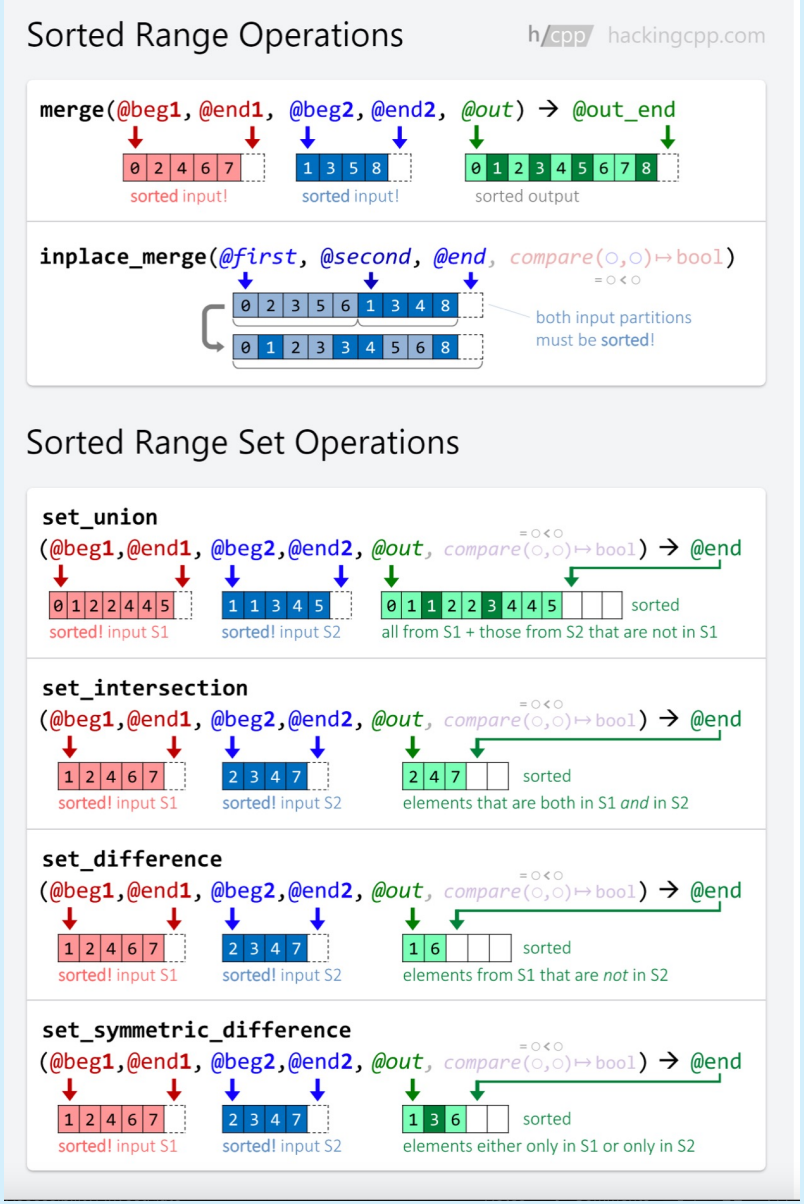




Imperial College London

Removing and Sorting



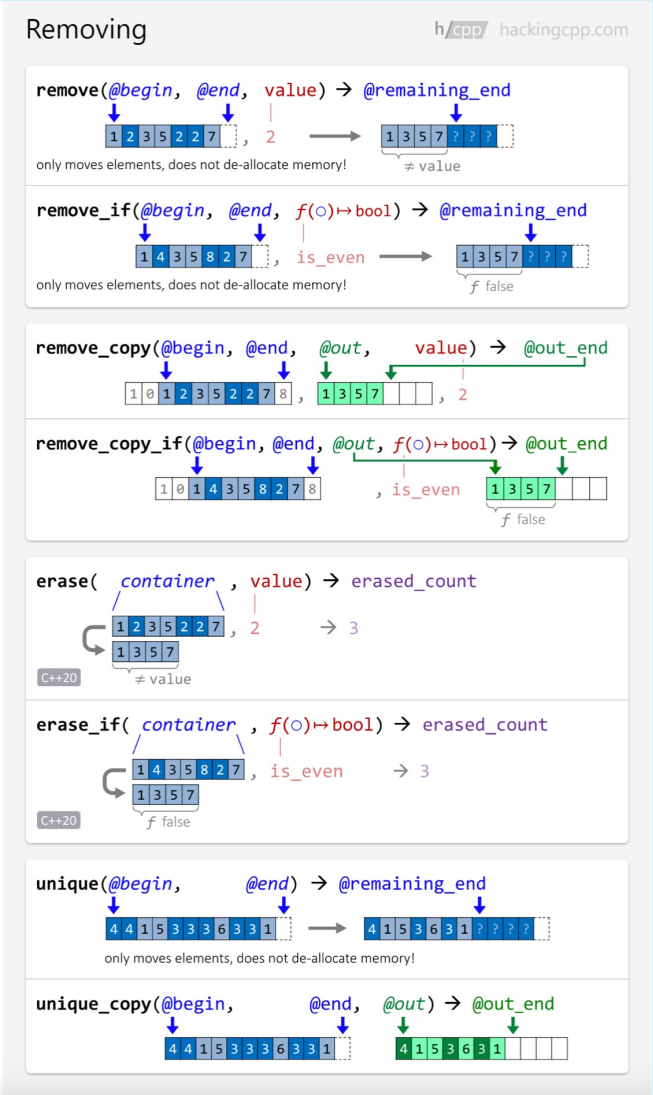


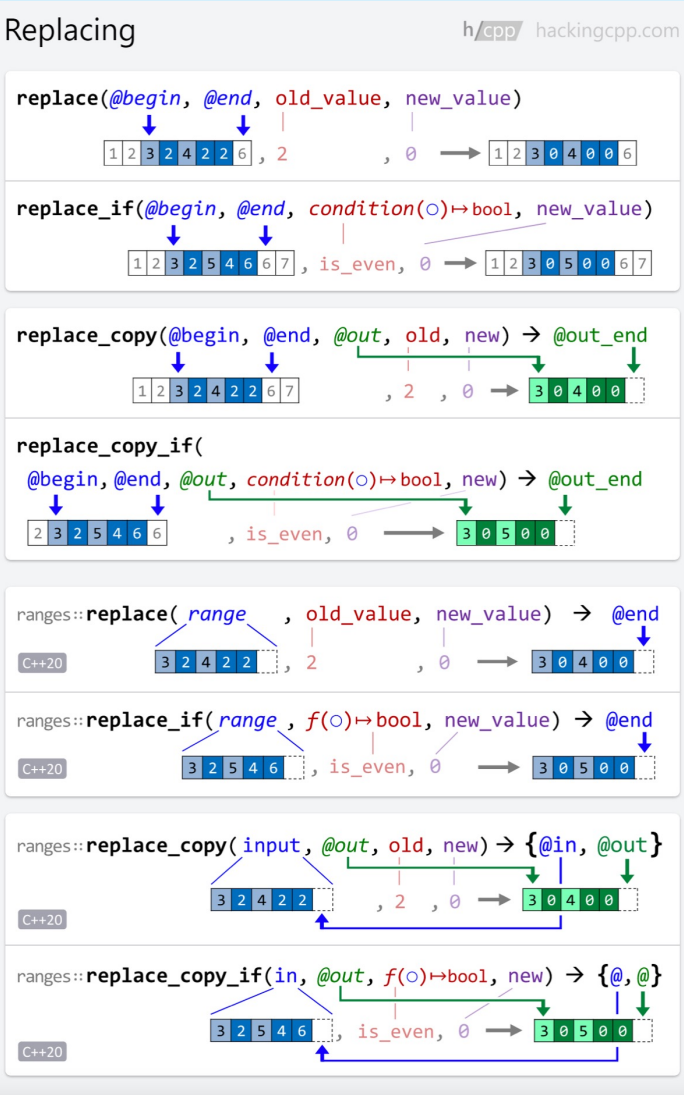
Imperial College London

Removing

and

Replacing





Imperial College London

EXERCISE

1. Create a vector with 1000 random three-letter words. Use the “find()” algorithm to search and count entries that have three equal letters.

2. Create a set with 1000 random three-letter words. Use the “replace()” algorithm to search words that start with the letter “a” and replace them by the same string but now starting with a ‘z’.

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **`bool`**
- For example
  - A function

```
bool odd(int i) { return i%2; }  // % is the remainder (modulo) operator
odd(7);                          // call odd: is 7 odd?
```

  - A function object

```
struct Odd {
     bool operator()(int i) const { return i%2; }
};
Odd odd;       // make an object odd of type Odd
odd(7);        // call odd: is 7 odd?
```

Function objects

A specific example

```
template<class T> struct Less_than {
   T val; // value to compare with
   Less_than(T& x) :val(x) { }
   bool operator()(const T& x) const { return x < val; }
};

// find x<43 in vector<int> :
p=find_if(v.begin(), v.end(), Less_than(43));

// find x<"perfection" in list<string>:
q=find_if(ls.begin(), ls.end(), Less_than("perfection"));
```

Function objects

- A very efficient technique
  - inlining very easy
    - and effective with current compilers
  - Faster than equivalent function
    - And sometimes you can't write an equivalent function
- The main method of policy parameterization in the STL
- Key to emulating functional programming techniques in C++

Policy parameterisation

- Whenever you have a useful algorithm, you eventually want to parameterize it by a “policy”.
  - For example, we need to parameterize sort by the comparison criteria

```
struct Record {
   string name;         // standard string for ease of use
   char addr[24];       // old C-style string to match database layout
   // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());     // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr());     // sort by addr
```

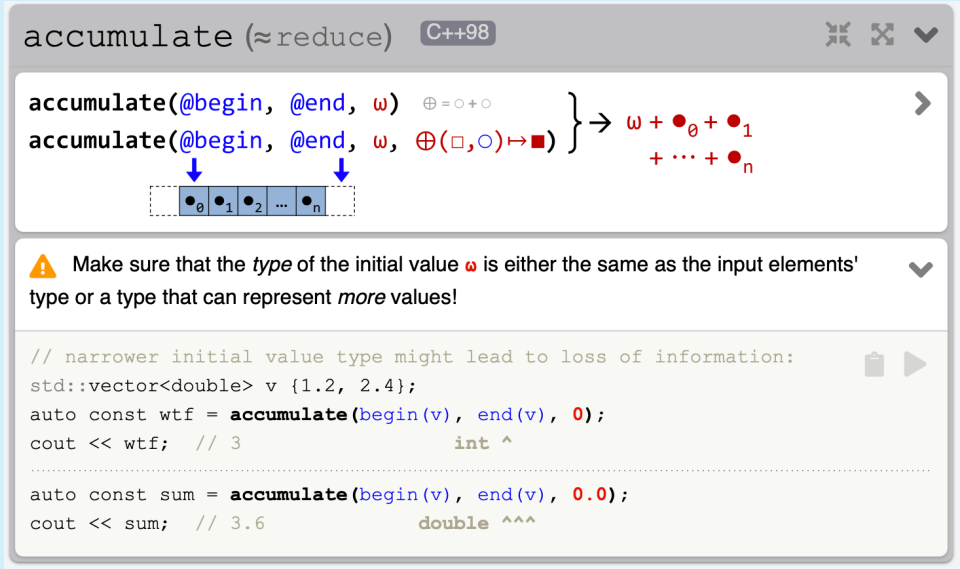
Some standard function objects

- From `<functional>`
  - Binary
    - plus, minus, multiplies, divides, modulus
    - equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or
  - Unary
    - negate
    - logical_not
  - Unary (missing, write them yourself)
    - less_than, greater_than, less_than_or_equal, greater_than_or_equal

Imperial College London

accumulate (sum the elements of a sequence)

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



v:

1	2	3	4

```
int sum = accumulate(v.begin(), v.end(), 0);      // sum becomes 10
```

Imperial College London

accumulate (sum the elements of a sequence)

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0); // add the elements of vd
    // note: the type of the 3rd argument, the initializer, determines the precision used

    int si = accumulate(p, p+n, 0);    // sum the ints in an int (danger of overflow)
                                       // p+n means (roughly) &p[n]

    long sl = accumulate(p, p+n, long(0));// sum the ints in a long
    double s2 = accumulate(p, p+n, 0.0);  // sum the ints in a double

    // popular idiom, use the variable you want the result in as the initializer:
    double ss = 0;
    ss = accumulate(vd.begin(), vd.end(), ss);  // do remember the assignment
}
```

Imperial College London

accumulate

(generalize: process the elements of a sequence)

```
// we don't need to use only +, we can use any binary operation (e.g., *)
// any function that "updates the init value" can be used:

template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
     init = op(init, *first);        // means "init op *first"
     ++first;
    }
    return init;
}
```

Imperial College London

accumulate

```
// often, we need multiplication rather than addition:
#include <numeric>
#include <functional>

void f(list<double>& ld)
{
    double product = accumulate(ld.begin(), ld.end(), 1.0, multiplies<double>());
    // …
}

// multiplies is a standard library function object for multiplying
```

Note: multiplies for *

Note: initializer 1.0

Imperial College London

accumulate (what if the data is part of a record?)

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

// let the "update the init value" function
// extract data from a Record element:
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units;
}

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

Imperial College London

EXERCISE

1. Consider this code. What is the role of myfun()? What is being printed?
2. What is the difference between sum(), myfun() and minus()?

```
// C++ program to demonstrate working of accumulate()
#include <iostream>
#include <numeric>
using namespace std;

// User defined function
int myfun(int x, int y)
{
	// for this example we have taken product
	// of adjacent numbers
	return x * y;
}

int main()
{
	// Initialize sum = 1
	int sum = 1;
	int a[] = { 5, 10, 15 };

	// Simple default accumulate function
	cout << "\nResult using accumulate: ";
	cout << accumulate(a, a + 3, sum);

	// Using accumulate function with
	// defined function
	cout << "\nResult using accumulate with"
		"user-defined function: ";
	cout << accumulate(a, a + 3, sum, myfun);

	// Using accumulate function with
	// pre-defined function
	cout << "\nResult using accumulate with "
		"pre-defined function: ";
	cout << accumulate(a, a + 3, sum, std::minus<int>());

	return 0;
}
```

Imperial College London

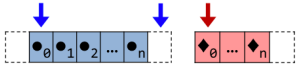
number of units	1	2	3	4	…
*	*	*	*	*	
unit price	4	3	2	1	…

inner_product

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
    // This is the way we multiply two vectors
    //(yielding a scalar)
{
    while(first!=last)
    {// multiply pairs of elements and sum
      init  = init + (*first) * (*first2);
      ++first;
      ++first2;
    }
    return init;
}
```

inner_product (≈transform_reduce) C++98

inner_product(@●$_{beg}$, @●$_{end}$, @◆, ω) , ⊕ = □ + Δ , ⊗ = ○ × ◊
inner_product(@●$_{beg}$, @●$_{end}$, @◆, ω, ⊕(□,Δ)↦■, ⊗(○,◊)↦▲) } → Π



$\Pi = \omega + (\bullet_0 \times \blacklozenge_0) + (\bullet_1 \times \blacklozenge_1) + \cdots + (\bullet_n \times \blacklozenge_n)$

Prefer C++17's `std::transform_reduce` because it can also be executed in parallel.

cppreference

```
std::vector<int> v {4,3,2,1};
std::vector<int> w {10,20,30,40};

auto const ip = inner_product(begin(v), end(v), begin(w), 50);
// ip = 50 + (4·10)+(3·20)+(2·30)+(1·40) = 250

std::vector<double> num {1.0, 3.0, 5.0};
std::vector<double> den {2.0, 4.0, 8.0};

auto const res = inner_product(
    begin(num), end(num), begin(den), 0.0,
    std::plus<>{}, std::divides<>{} );
// res = 0.0 + (1/2)+(3/4)+(5/8) = 1.875
```

Imperial College London

inner_product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price; // share price for each company
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);
// …
vector<double> dow_weight;     // weight in index for each company
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
// …
double dj_index = inner_product(    // multiply (price,weight) pairs and add
    dow_price.begin(), dow_price.end(),
    dow_weight.begin(),
    0.0);
```

Imperial College London

inner_product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price = {   // share price for each company
    81.86, 34.69, 54.45,
    // …
};
vector<double> dow_weight = {      // weight in index for each company
    5.8549, 2.4808, 3.8940,
    // …
};

double dj_index = inner_product(    // multiply (price,weight) pairs and add
     dow_price.begin(), dow_price.end(),
     dow_weight.begin(),
     0.0);
```

Imperial College London

inner_product (generalise!)

```
// we can supply our own operations for combining element values with“init”:
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
   while(first!=last) {
    init  = op(init, op2(*first, *first2));
    ++first;
    ++first2;
   }
   return init;
}
```

Imperial College London

copy example

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first!=last) *res++ = *first++;
                // conventional shorthand  for:
                //  *res = *first; ++res; ++first
    return res;
}

void f(vector<double>& vd, list<int>& li)
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin()); // note: different container types
                            // and different element types
                            // (vd needs tp have enough elements
                            // to hold copies of li's elements)
    sort(vd.begin(), vd.end());
    // …
}
```

Imperial College London

Input and output iterators

```
// we can provide iterators for output streams

   ostream_iterator<string> oo(cout);  // assigning to *oo is to write to cout

   *oo = "Hello, ";   // meaning cout << "Hello, "
   ++oo;              // “get ready for next output operation”
   *oo = "world!\n";  // meaning cout << "world!\n"


// we can provide iterators for input streams:

   istream_iterator<string> ii(cin); // reading *ii is to read a string from cin

   string s1 = *ii;   // meaning cin>>s1
   ++ii;              // “get ready for the next input operation”
   string s2 = *ii;   // meaning cin>>s2
```

Imperial College London

copy_if()

```
// a very useful algorithm (missing from the standard library):

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

Imperial College London

copy_if()

```
void f(const vector<int>& v)   // “typical use” of predicate with data
                       // copy all elements with a value less than 6
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), [](int x) { return x<6; } );
    // …
}
```

Imperial College London

EXERCISE

1. Create a vector with 10 strings. Copy the entries to a set using an algorithm.

2. Create a vector with 10,000 entries of numbers between 0.1 and 0.001. Find the square of its magnitude (the inner product of the vector with itself).