Modern C++ for Scientific Computing

Victor Eijkhout

TACCster 2024

Introduction

1. Write modern C++

1. C++ is a more structured and safer variant of C:
   There are very few reasons not to switch to C++.
2. C++ has undergone many changes in the last dozen years
   Many mechanisms make programming easier and safer
   (not faster: you'll never beat writing C)

2. In this course

1. Minor enhancements
2. Looping and ranges
3. Smart pointers
4. OpenMP mechanisms
5. A nice looking MPI library
6. More.

I'm assuming that you know how to code C loops and functions and you understand what structures and pointers are!

3. About this course

Slides and codes are from my open source text book:

`https://theartofhpc.com/isp.html`

4. General note about standards

Many of the examples in this lecture use the C++17/20/23 (sometimes C++26) standard.

```
icpc    -std=c++20 yourprogram.cxx
icpx    -std=c++20 yourprogram.cxx
g++     -std=c++20 yourprogram.cxx
clang++ -std=c++20 yourprogram.cxx
```

There is no reason not to use that all the time:

```
alias icpc='icpc -std=c++20'
et cetera
```

5. Build with Cmake

```
cmake_minimum_required( VERSION 3.20 )
project( ${PROGRAM_NAME} VERSION 1.0 )

add_executable( ${PROGRAM_NAME} ${PROGRAM_NAME}.cpp )
target_compile_features( ${PROGRAM_NAME}
    PRIVATE cxx_std_20 )
```

6. C++ standard

- C++98/C++03: ancient.
  There was a lot wrong or not-great with this.
- C++11/14/17: 'modern' C++.
  What everyone uses.
- C++20/23: 'post-modern' C++.
  Ratified, but only partly implemented.
- C++26: 'feature freeze' in 2025; some features already available.

7. What is not (modern) C++?

Do not use:

- Parameter passing with `y=f(&x);`
  'star' pointers
- `malloc` and such
- Arrays and strings `a[5];`

It's legal, just not 'modern', and frankly not needed.
Explanations to follow ...

Minor enhancements

8. Conditional with initializer

Variable local to the conditional:

Code:

```
// basic/ifinit.cpp
if ( char c = getchar();
     c!='a' )
  cout << "Not an a, but: "
       << c << '\n';
else
  cout << "That was an a!"
       << '\n';
```

Output:

```
Script:
for c in d b a z ;
    ↪do \
   echo $c |
    ↪./ifinit ; \
done
Not an a, but: d
Not an a, but: b
That was an a!
Not an a, but: z
```

9. Initializer statement

Loop variable can be local (also in C99):

```
for (int i=0; i<N; i++) // do whatever
```

Similar in conditionals and switch:

```
// basic/ifinit.cpp
if ( char c = getchar(); c!='a' )
  cout << "Not an a, but: "
       << c << '\n';
else
  cout << "That was an a!"
       << '\n';
```

(strangely not in `while`)

10. Simple I/O

Headers:

```
#include <iostream>
using std::cin;
using std::cout;
```

Ouput:

```
int main() {
  int plan=4;
  cout << "Plan " << plan << " from outer space" << "\n";
```

11. Format library

printf-ish formatting:

```
#include <format>

string std::format( /* stuff */ );
void   std::print ( /* stuff */ ); // as of C++23
```

(also available as external `fmtlib`)

12. Simple example

The basic usage is:

```
int i=2;
format("string {} brace expressions",i);
```

Format string, and arguments.

13. Displaying the format result

Use `cout` or (C++23) *print*:

Code:

```
// iofmt/fmtbasic.cpp
cout << format("{}\n",2);
string hello_string = format
  ("{} {}!","Hello","world");
cout << hello_string << '\n';
cout << format
  ("{0}, {0} {1}!\n",
   "Hello","world");
// c++23 only:
// print("{0}, {0} {1}
    {1}!\n",
//       "Hello","world");
```

Output:

```
2
Hello world!
Hello, Hello world!
```

14. Right align

Right-align with > character and width:

Code:

```
// io/fmtlib.cpp
for (int i=10; i<200000000;
     i*=10)
  fmt::print("{:>6}\n",i);
```

Output:

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
```

15. Padding character

Other than space for padding:

Code:

```
// io/fmtlib.cpp
for (int i=10; i<200000000;
     i*=10)
  fmt::print("{0:.>6}\n",i);
```

Output:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
```

16. Number bases

Code:

```
// io/fmtlib.cpp
fmt::print
  ("{0} = {0:b} bin\n",17);
fmt::print
  ("    = {0:o} oct\n",17);
fmt::print
  ("    = {0:x} hex\n",17);
```

Output:

```
17 = 10001 bin
   = 21 oct
   = 11 hex
```

17. Float and fixed

Floating point or normalized exponential with `e` specifier
fixed: use decimal point if it fits, `m.n` specification

Code:

```
// iofmt/fmtfloat.cpp
x = 1.234567;
for (int i=0; i<6; ++i) {
  cout <<

    format("{0:.3e}/{0:7.4}\n",
           x);
  x *= 10;
}
```

Output:

```
1.235e+00/  1.235
1.235e+01/  12.35
1.235e+02/  123.5
1.235e+03/   1235
1.235e+04/1.235e+04
1.235e+05/1.235e+05
```

Functions

18. Parameter passing by reference

The function parameter `n` becomes a reference to the variable `i` in the main program:

```
1  void f(int &n) {
2    n = /* some expression */ ;
3  };
4  int main() {
5    int i;
6    f(i);
7    // i now has the value that was set in the function
8  }
```

19. Pass by reference example 1

Code:

```
// basic/setbyref.cpp
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << '\n';
```

Output:

```
5
```

Compare the difference with leaving out the reference.

20. Pass by reference example 2

```
bool can_read_value( int &value ) {
  // this uses functions defined elsewhere
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status==0;
}

int main() {
  int n;
  if (!can_read_value(n)) {
    // if you can't read the value, set a default
    n = 10;
  }
  ..... do something with 'n' ....
```

21. Const ref parameters

```
void f( const int &i ) { .... }
```

- Pass by reference: no copying, so cheap
- Const: no accidental altering.
- Especially useful for large objects.

22. Default arguments

Functions can have default argument(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

23. Useful idiom

Don't trace a function unless I say so:

```
void dosomething(double x,bool trace=false) {
  if (trace) // report on stuff
};
int main() {
  dosomething(1); // this one I trust
  dosomething(2); // this one I trust
  dosomething(3,true); // this one I want to trace!
  dosomething(4); // this one I trust
  dosomething(5); // this one I trust
```

24. Polymorphic functions

You can have multiple functions with the same name:

```
double average(double a,double b) {
  return (a+b)/2; }
double average(double a,double b,double c) {
  return (a+b+c)/3; }
```

Distinguished by type or number of input arguments: can not differ only in return type.

```
int f(int x);
string f(int x); // DOES NOT WORK
```

Lambda expressions

25. Introducing: lambda expressions

Traditional function usage:
explicitly define a function and apply it:

```
float sum(float x,float y) { return x+y; }
cout << sum( 1.2f, 3.4f );
```

New:
apply the function recipe directly:

Code:

```
// lambda/lambdaex.cpp
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Output:

```
3.8
```

26. Lambda syntax

```
[capture] ( inputs ) -> outtype { definition };
[capture] ( inputs ) { definition };
```

- The square brackets are how you recognize a lambda; we will get to the 'capture' later. For now it will often be empty.
- Inputs: like function parameters
- Result type specification `-> outtype`: can be omitted if compiler can deduce it;
- Definition: function body.

27. Lambdas as parameter: the problem

Lambdas have a type that is dynamically generated, so you can not write a function that takes a lambda as argument, because you can't write the type.

```
1  void apply_to_5( /* what? */ func ) {
2      func(5);
3  }
4  int main() {
5      apply_to_5
6        ( [] (double x) { cout << x; } );
7  }
```

28. Lambdas as parameter: the solution

Header:

```
#include <functional>
using std::function;
```

declare function parameters by their signature
(that is, types of parameters and output):

Code:

```
// lambda/lambdaex.cpp
void apply_to_5
    ( function< void(int) > f
    ) {
  f(5);
}
    /* ... */
  apply_to_5
    ( [] (int i) {
      cout << "Int: " << i
           << '\n'; } );
```

Output:

```
Int: 5
```

```
// newton/newton-lambda.cpp
double newton_root
    ( function< double(double) > f,
      function< double(double) > fprime ) {
```

29. Capture variable

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

Code:

```
// lambda/lambdacapture.cpp
int n;
cin >> n;
auto increment_by_n =
  [n] ( int input ) -> int {
    return input+n;
};
cout << increment_by_n (5)  << '\n';
cout << increment_by_n (12)  << '\n';
cout << increment_by_n (25)  << '\n';
```

Output:

```
(input value:
    ↪1)

6
13
26
```

30. Capture value is copied

Illustrating that the capture variable is copied once and for all:

Code:

```
// lambda/lambdacapture.cpp
int inc;
cin >> inc;
auto increment =
  [inc] ( int input ) -> int {
    return input+inc;
  };
cout << "increment by: "
     << inc << '\n';
cout << "1 -> "
     << increment(1) << '\n';
inc = 2*inc;
cout << "1 -> "
     << increment(1) << '\n';
```

Output:

```
increment by: 2
1 -> 3
1 -> 3
```

```
// newton/newton-lambda.cpp
  for (int n=2; n<=8; ++n) {
    cout << "sqrt(" << n << ") = "
         << newton_root(
    /* ... */
                        )
         << '\n';


// newton/newton-lambda.cpp
[n] (double x) { return x*x-n; },
[] (double x) { return 2*x; }


// newton/newton-lambda.cpp
double newton_root( function< double(double) > f, double
    h=.001 ) {
  cout << "gradient-free newton with h=" << h << '\n';
  return newton_root( f, [f,h] (double x) { return
    (f(x+h)-f(x))/h; } );
};
```

Vectors, loops, ranges

31. Short vectors

Short vectors can be created by enumerating their elements:

```
1 // array/shortvector.cpp
2 #include <vector>
3 using std::vector;
4
5 int main() {
6   vector<int> evens{0,2,4,6,8};
7   vector<float> halves = {0.5, 1.5, 2.5};
8   auto halfloats = {0.5f, 1.5f, 2.5f};
9   cout << evens.at(0)
10       << " from " << evens.size()
11       << '\n';
12  return 0;
13 }
```

32. Range over elements

A range-based for loop gives you directly the element values:

```
vector<float> my_data(N);
/* set the elements somehow */;
for ( float e : my_data )
  // statement about element e
```

Here there are no indices because you don't need them.

33. Range over elements, version 2

Same with auto instead of an explicit type for the elements:

```
for ( auto e : my_data )
  // same, with type deduced by compiler
```

34. Range over elements by reference

Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
// array/vectorrangeref.cpp
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2)
     << '\n';
```

Output:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

35. Range over vector denotation

Code:

```
// array/rangedenote.cpp
for ( auto i : {2,3,5,7,9} )
  cout << i << ",";
cout << '\n';
```

Output:

```
2,3,5,7,9,
```

36. Example: multiplying elements

Example: multiply all elements by two:

Code:

```
// array/vectorrangeref.cpp
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2)
     << '\n';
```

Output:

```
6.6
```

Exercise 1

Create a `vector` x of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in L_2 norm and check the correctness of your calculation, that is,

1. Compute the L_2 norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

37. Range-based iteration

You have seen

```
for ( auto n : set_of_integers )
  if ( even(n) )
    do_something(n);
```

Can we do

```
for ( auto n : set_of_integers
    and even ) // <= not actual syntax
  do_something(n);
```

or even

```
// again, not actual syntax
apply( set_of_integers and even,
    do_something ):
```

38. Loop algorithms

Algorithms: for-each, find, filter, ...

Ranges: iteratable things such as vectors

Views: transformations of ranges, such as picking only even numbers

C++20 ranges

39. Range over vector

With

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
// rangestd/range.cpp
#include <algorithm>
#include <ranges>
namespace rng = std::ranges;
    /* ... */
  rng::for_each
    ( v,
      [] (int i) {
        cout << i << " ";
      }
      );
```

Output:

```
2 3 4 5 6 7
```

40. Ranged algorithm

With

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
// rangestd/range.cpp
count = 0;
rng::for_each
  ( v,
    [&count] (int i) {
      count += (i<5); }
    );
cout << "Under five: "
     << count << '\n';
```

Output:

```
Under five: 3
```

41. Range composition

Pipeline of ranges and views:

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
// rangestd/range.cpp
count = 0;
rng::for_each
  ( v | rng::views::drop(1),
    [&count] (int i) {
      count += (i<5); }
    );
cout << "minus first: "
     << count << '\n';
```

Output:

```
minus first: 2
```

'pipe operator'

42. Iota and take

Code:

```
// rangestd/iota.cpp
#include <ranges>
namespace rng = std::ranges;
    /* ... */
  for ( auto n :

    rng::views::iota(2,6) )
    cout << n << '\n';
  cout << "===\n";
  for ( auto n :
          rng::views::iota(2)
          |
    rng::views::take(4) )
    cout << n << '\n';
```

Output:

```
2
3
4
5
===
2
3
4
5
```

Exercise 2: Iota and take

Rewrite the second loop of the previous slide using an algorithm, and no explicit loop.

43. Filter

Take a range, and make a new one of only the elements satisfying some condition:

Code:

```
// rangestd/filter.cpp
vector<float> numbers
  {1,-2.2,3.3,-5,7.7,-10};
auto pos_view =
  numbers
  | std::ranges::views::filter
    ( [] (int i) -> bool {
        return i>0; }
     );
for ( auto n : pos_view )
  cout << n << " ";
cout << '\n';
```

Output:

```
1 3.3 7.7
```

Exercise 3: Element counting

Change the filter example to let the lambda count how many elements were > 0.

44. Range composition

Code:

```
// range/filtertransform.cpp
vector<int> v{ 1,2,3,4,5,6 };
/* ... */
auto times_two_over_five = v
  | rng::views::transform
      ( [] (int i) {
        return 2*i; } )
  | rng::views::filter
      ( [] (int i) {
        return i>5; } );
```

Output:

```
Original vector:
  1, 2, 3, 4, 5, 6,
Times two over five:
  6 8 10 12
```

45. Quantor-like algorithms

Code:

```
// rangestd/of.cpp
vector<int>
    integers{1,2,3,5,7,10};
auto any_even =
  std::ranges::any_of
    ( integers,
      [=] (int i) -> bool {
        return i%2==0; }
     );
if (any_even)
  cout << "there was an
    even\n";
else
  cout << "none were even\n";
```

Output:

```
there was an even
```

Also *all_of*, *none_of*

46. Reductions

accumulate and *reduce*:
tricky, and not in all compilers.
See above for an alternative.

Exercise 4: Perfect numbers

A perfect number is the sum of its own divisors:

$$6 = 1 + 2 + 3$$

Output the perfect numbers.
(at least 4 of them)
Use only ranges and algorithms, no explicit loops.

Multi-dimensional arrays

47. Using subarrays

Form *subarray* as part of *array* that starts at the second element:

```
double *array = new double[N];
double *subarray = array+1;
subarray[1] = 5.; // same as: array[2] = 5.;
```

Using 'subarrays' would be useful, for instance in a quicksort algorithm:

```
// Warning: this is pseudo-code
void qs( data ) {
  if (data.size()>1) {
    // pivoting stuff omitted
    qs( data.lefthalf() ); qs( data.lefthalf() );
  }
}
```

48. Span

Create a `span` from a `vector`, starting at its second element:

```
#include <span>
vector<double> v;
std::span<double> v_span( v.data()+1,v.size()-1 );
```

49. mdspan

Header: mdspan

Create 2D `mdspan` from vector:

```
// mdspan/index2.cpp
// matrix in row major
vector<float> A(M*N);
md::mdspan
  Amd{ A.data(),md::extents{M,N} };
```

50. Two-d mdspan matrix

Construct a multi-dimensional span from a vector:

```
vector<float> ar10203040(10*20*30*40);
auto brick10203040 =
    std::mdspan< float, extents<10,20,30,40> >(
    ar10203040.data() );
auto midpoint = brick10203040[5,10,15,20];
```

51. Rowsum calculation

Given mdspan *mat*, find its sizes, extract each row, and the sum of its elements:

```
// mdspan/index2.cpp
int M = mat.extent(0); int N = mat.extent(1);
vector<float> rowsums(N);
for ( int row=0; auto& rs : rowsums ) {
  auto the_row =
    rng::iota_view(0,M)
    | rng::views::transform
      ( [mat,row] (int col) -> float {
        return mat[row,col]; } );
  rs = rng::accumulate( the_row, 0.f );
  row++;
}
```

Note that *the_row* is a view, not a data structure.

OpenMP parallel loops

Questions

1. Do regular OpenMP loops look different in C++?
2. Is there a relation between OpenMP parallel loops and iterators?
3. OpenMP parallel loops vs parallel execution policies on algorithms.

Range syntax

Parallel loops in C++ can use range-based syntax as of OpenMP-5.0:

```
// vecdata.cxx
vector<float> values(100);

#pragma omp parallel for
for ( auto& elt : values ) {
  elt = 5.f;
}

float sum{0.f};
#pragma omp parallel for reduction(+:sum)
for ( auto elt : values ) {
  sum += elt;
}
```

Tests show exactly the same speedup as the C code.

General idea

OpenMP can parallelize any loop over a C++ construct that has a 'random-access' iterator.

C++ ranges header

The C++20 ranges library is supported:

```
#pragma omp parallel for reduction(+:count)
for ( auto e : data
              | std::ranges::views::drop(1) )
        count += e;
#pragma omp parallel for reduction(+:count)
for ( auto e : data
              | std::ranges::views::transform
              ( []( auto e ) { return 2*e; } ) )
        count += e;
```

C++ ranges speedup

```
==== Run range on 1 threads ====
sum of vector: 50000005000000 in 6.148
sum w/ drop 1: 50000004999999 in 6.017
sum times 2  : 100000010000000 in 6.012
==== Run range on 25 threads ====
sum of vector: 50000005000000 in 0.494
sum w/ drop 1: 50000004999999 in 0.477
sum times 2  : 100000010000000 in 0.489
==== Run range on 51 threads ====
sum of vector: 50000005000000 in 0.257
sum w/ drop 1: 50000004999999 in 0.248
sum times 2  : 100000010000000 in 0.245
==== Run range on 76 threads ====
sum of vector: 50000005000000 in 0.182
sum w/ drop 1: 50000004999999 in 0.184
sum times 2  : 100000010000000 in 0.185
==== Run range on 102 threads ====
sum of vector: 50000005000000 in 0.143
sum w/ drop 1: 50000004999999 in 0.139
sum times 2  : 100000010000000 in 0.134
==== Run range on 128 threads ====
sum of vector: 50000005000000 in 0.122
sum w/ drop 1: 50000004999999 in 0.11
```

Ranges and indices

Use *iota_view* to obtain indices:

```
// iota.cxx
vector<long> data(N);
# pragma omp parallel for
for ( auto i : std::ranges::iota_view( 0UZ,data.size() ) )
  data[i] = f(i);
```

Note that this uses C++23 suffix for `unsigned size_t`. For older versions:

```
iota_view( static_cast<size_t>(0),data.size() )
```

Custom iterators, 0

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
   ... e ...
```

for:

```
for ( vector<float>::iter
      e=v.begin();
      e!=v.end(); e++ )
  ... *e ...
```

If we want

```
for ( auto e : my_object )
    ... e ...
```

we need a sub-class for the iterator with methods such as *begin*, *end*, * and ++.
Probably also += and -

Custom iterators, 1

OpenMP can parallelize any range-based loop with a random-access iterator.

Class:

```
// iterator.cxx
template<typename T>
class NewVector {
protected:
  T *storage;
  int s;
public:
// iterator stuff
  class iter;
  iter begin();
  iter end();
};
```

Main:

```
NewVector<float> v(s);
#pragma omp parallel for
for ( auto e : v )
  cout << e << " ";
```

Custom iterators, 2

Required iterator methods:

```
NewVector<T>::iter& operator++();
T& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-
    ( const NewVector::iter& other ) const;
NewVector<T>::iter& operator+=( int add );
```

This is a little short of a full random-access iterator; the difference depends on the OpenMP implementation.

Custom iterators, exercise

Write the missing iterator methods.

Here's something to get you started.

```
template<typename T>
class NewVector<T>::iter {
private: T *searcher;
};
template<typename T>
NewVector<T>::iter::iter( T* searcher )
  : searcher(searcher) {};
template<typename T>
NewVector<T>::iter NewVector<T>::begin() {
  return NewVector<T>::iter(storage); };
template<typename T>
NewVector<T>::iter NewVector<T>::end()   {
  return NewVector<T>::iter(storage+NewVector<T>::s); };
```

Custom iterators, solution

```
template<typename T>
bool  NewVector<T>::iter::operator==
    ( const NewVector<T>::iter &other ) const {
  return searcher==other.searcher; };
template<typename T>
bool  NewVector<T>::iter::operator!=
    ( const NewVector<T>::iter &other ) const {
  return searcher!=other.searcher; };
template<typename T>
NewVector<T>::iter& NewVector<T>::iter::operator++() {
  searcher++; return *this; };
template<typename T>
NewVector<T>::iter& NewVector<T>::iter::operator+=( int add )
    {
  searcher += add; return *this; };
```

Custom iterators, solution

```
template<typename T>
T&  NewVector<T>::iter::operator*() {
  return *searcher; };
// needed for OpenMP
template<typename T>
int   NewVector<T>::iter::operator-
    ( const NewVector<T>::iter& other ) const {
  return searcher-other.searcher; };
```

OpenMP vs standard parallelism

Application: prime number marking (load unbalanced)

```
#pragma omp parallel for \
  schedule(static)
 for ( int i=0; i<nsize; i++) {
   results[i] =
     one_if_prime( number(i) );
 }

// primepolicy.cpp
transform
  ( std::execution::par,
    numbers.begin(),numbers.end(),
    results.begin(),
    [] (int n ) -> int {
      return one_if_prime(n); }
    );
```

Standard parallelism uses Threading Building Blocks (Intel) (TBB) as backend

Timing

```
Threads:  1
TBB:  Time:       392 msec
Stat: Time:       389 msec
Dyn:  Time:       390 msec

Threads: 25
TBB:  Time:        20 msec
Stat: Time:        32 msec
Dyn:  Time:        17 msec

Threads: 51
TBB:  Time:        13 msec
Stat: Time:        15 msec
Dyn:  Time:         9 msec

Threads: 76
TBB:  Time:        29 msec
Stat: Time:        11 msec
Dyn:  Time:         6 msec

Threads: 102
TBB:  Time:        61 msec
Stat: Time:         8 msec
```

Reductions vs standard parallelism

Application: prime number counting (load unbalanced)

```
#pragma omp parallel for \
  schedule(guided,8) \
  reduction(+:prime_count)
for ( auto n : numbers ) {
  prime_count += one_if_prime( n );
}

// reducepolicy.cpp
prime_count = transform_reduce
  ( std::execution::par,
    numbers.begin(),numbers.end(),
    0,
    std::plus<>{},
    [] ( int n ) -> int {
        return one_if_prime(n); }
    );
```

Timing

```
Threads:  1
TBB:  Time:       391 msec
Stat: Time:       390 msec
Dyn:  Time:       389 msec

Threads: 25
TBB:  Time:        20 msec
Stat: Time:        17 msec
Dyn:  Time:        17 msec

Threads: 51
TBB:  Time:        13 msec
Stat: Time:         9 msec
Dyn:  Time:         8 msec

Threads: 76
TBB:  Time:        14 msec
Stat: Time:         8 msec
Dyn:  Time:         5 msec

Threads: 102
TBB:  Time:        76 msec
Stat: Time:         5 msec
```

OpenMP reductions

Questions

1. Are simple reductions the same as in C?
2. Can you reduce `std::vector` like an array?
3. Precisely *what* can you reduce?
4. Any interesting examples?
5. Compare reductions to native C++ mechanisms.

Scalar reductions

Same as in C,
you can now use range syntax for the loop.

```
// range.cxx
#pragma omp parallel for reduction(+:count)
for ( auto e : data )
        count += e;
```

Reductions on vectors

Use the *data* method to extract the array on which to reduce. However, this does not work:

```
vector<float> x;
#pragma omp parallel reduction(+:x.data())
```

because the reduction clause wants a variable, not an expression, for the array, so you need an extra bare pointer:

```
// reductarray.cxx
vector<int> data(nthreads,0);
int *datadata = data.data();
#pragma omp parallel for schedule(static,1) \
  reduction(+:datadata[:nthreads])
```

Reduction on class objects

Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

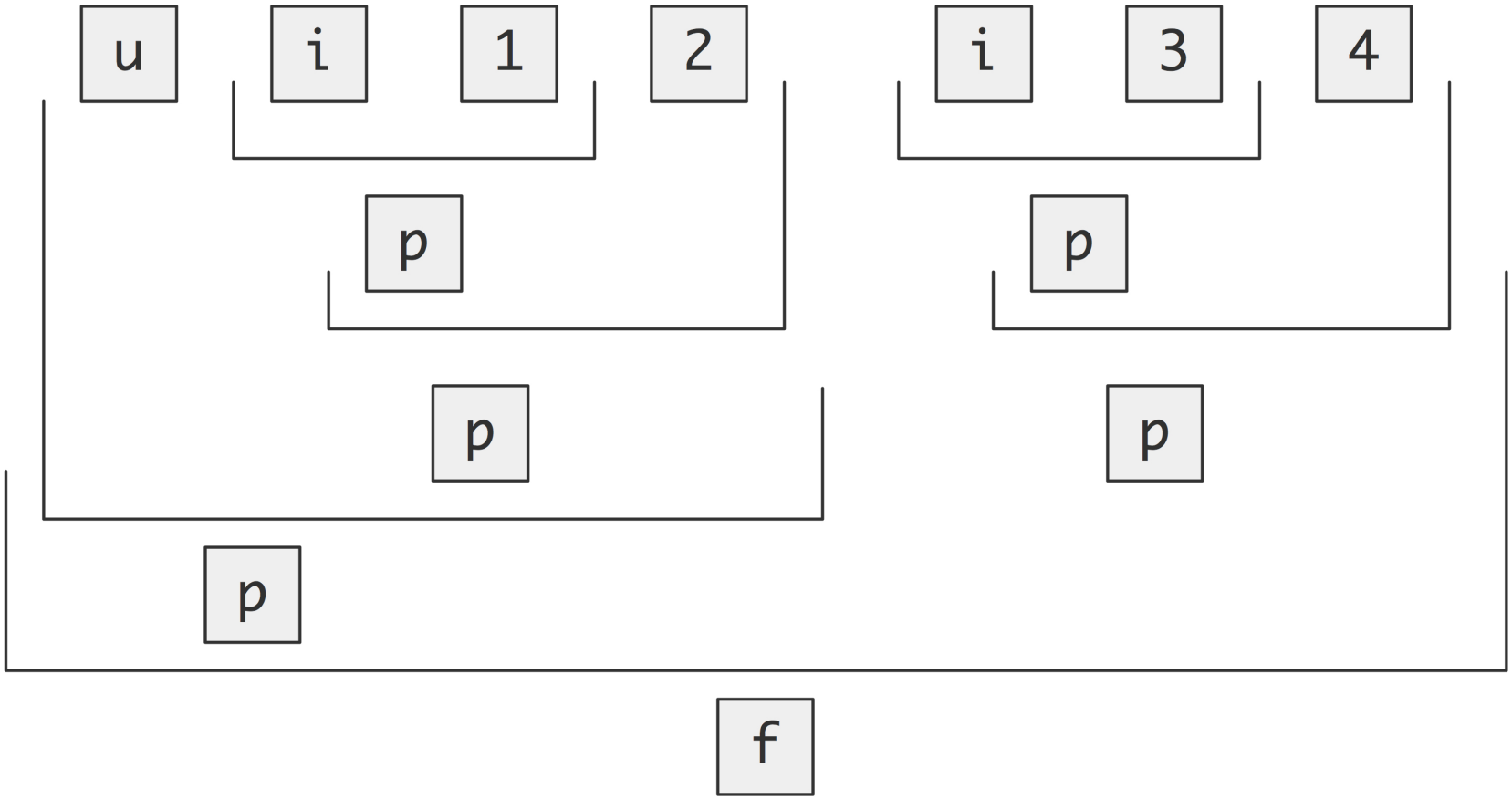
```
// reductclass.cxx
class Thing {
private:
  float x{0.f};
public:
  Thing() = default;
  Thing( float x ) : x(x) {};
  Thing operator+
      ( const Thing& other )
    {
    return Thing( x +
    other.x );
  };
};
```

```
vector< Thing >
  things(500,Thing(1.f) );
Thing result(0.f);
#pragma omp parallel for \
    reduction( +:result )
for ( const auto& t : things
    )
  result = result + t;
```

A default constructor is required for the internally used init value; see figure 86

Reduction illustrated

Reduction of four items on two threads. `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value.



User-defined reductions, syntax

```
#pragma omp declare reduction
( identifier : typelist : combiner )
[initializer(initializer-expression)]
```

Reduction over iterators

Support for *C++ iterators*

```
#pragma omp declare reduction \
    (merge                  // identifier
    : std::vector<int> // typelist
    : omp_out.insert(omp_out.end(), omp_in.begin(),
                     omp_in.end()) // combiner
    )
```

Lambda expressions in declared reductions

You can use lambda expressions in the explicit expression for a declared reduction:

```
// reductexpr.cxx
#pragma omp declare reduction\
  (minabs : int : \
   omp_out = \
       [] (int x,int y) -> int { \
          return abs(x) > abs(y) ? abs(y) : abs(x); } \
       (omp_in,omp_out) ) \
  initializer (omp_priv=limit::max())
```

You can not assign the lambda expression to a variable and use that, because *omp_in/out* are the only variables allowed in the explicit expression.

Example category: histograms

Count which elements fall into what bin:

```
for ( auto e : some_range )
  histogram[ value(e)]++;
```

Collisions are possible, but unlikely, so critical section is very inefficient

Histogram: intended main program

Declare a reduction on a histogram object;
each thread gets a local map:

```
/*
 * Reduction loop in main program
 */
bincounter<char> charcount;
#pragma omp parallel for reduction(+ : charcount)
for ( int i=0; i<text.size(); i++ )
  charcount.inc( text[i] );
```

Q: why does the *inc* not have to be atomic?

Histogram solution: reduction operator

Give the class a += operator to do the combining:

```
// charcount.cxx
template<typename key>
class bincounter : public map<key,int> {
public:
// merge this with other map
  void operator+=
       ( const bincounter<key>& other ) {
    for ( auto [k,v] : other )
      if ( map<key,int>::contains(k) )
        this->at(k) += v;
      else
        this->insert( {k,v} );
  };
// insert one char in this map
  void inc(char k) {
    if ( map<key,int>::contains(k) )
      this->at(k) += 1;
    else
      this->insert( {k,1} );
};
```

Histogram in native C++

Use atomics because there is no reduction mechanism:

```
// mapreduceatomic.cxx
class CharCounter : public array<atomic<int>,26> {
public:
  CharCounter() {
    for ( int ic=0; ic<26; ic++ )
      (*this)[ic] = 0;
  };
// insert one char in this map
  void inc(char k) {
    if (k==' ') return;
    int ik = k-'a';
    (*this)[ik]++;
  };
};
```

Histogram in native C++, comparison

OpenMP reduction on `array<int,26>`:

```
Using atomics    on 1 threads: time= 20.19 msec
OpenMP reduction on 1 threads: time= 1.966 msec
Using atomics    on 5 threads: time= 315.855 msec
OpenMP reduction on 5 threads: time= 0.52 msec
Using atomics    on 10 threads: time= 91.968 msec
OpenMP reduction on 10 threads: time= 0.364 msec
Using atomics    on 30 threads: time= 249.171 msec
OpenMP reduction on 30 threads: time= 0.556 msec
Using atomics    on 50 threads: time= 164.177 msec
OpenMP reduction on 50 threads: time= 0.904 msec
```

Exercise: mapreduce

Make an OpenMP parallel version of:

```
intcounter primecounter;
for ( auto n : numbers )
  if ( isprime(n) )
    primecounter.add(n);
```

where *primecounter* contains a *map*<int,int>.

Use skeleton: `mapreduce.cxx`

Example category: list filtering

The sequential code is as follows:

```
vector<int> data(100);
// fil the data
vector<int> filtered;
for ( auto e : data ) {
  if ( f(e) )
    filtered.push_back(e);
}
```

List filtering, solution 1

Let each thread have a local array, and then to concatenate these:

```
#pragma omp parallel
{
  vector<int> local;
# pragma omp for
  for ( auto e : data )
    if ( f(e) ) local.push_back(e);
  filtered += local;
}
```

where we have used an append operation on vectors:

```
// filterreduct.cxx
template<typename T>
vector<T>& operator+=( vector<T>& me, const vector<T>& other
    ) {
  me.insert( me.end(),other.begin(),other.end() );
  return me;
};
```

List filtering, not quite solution 2

We could use the plus-is operation to declare a reduction:

```
#pragma omp declare reduction\
  (     \
        +:vector<int>:omp_out += omp_in \
        ) \
  initializer( omp_priv = vector<int>{} )
```

Problem: OpenMP reductions can not be declared non-commutative, so the contributions from the threads may not appear in order.

Code:

```
#pragma omp parallel \
  reduction(+ : filtered)
  {
    vector<int> local;
#   pragma omp for
    for ( auto e : data )
      if ( f(e) )
        local.push_back(e);
```

Output:

```
Mod 5: 80 85 90 95
    ↪100 5 10 15 20
    ↪25 30 35 40 45
    ↪50 55 60 65 70
    ↪75
```

List filtering, task-based solution

Parallel region, without for:

Code:

```
// filtertask.cxx
vector<int> filtered;
int ithread=0;
#pragma omp parallel
{
  vector<int> local;
  int threadnum =
    omp_get_thread_num();
#   pragma omp for
  for ( auto e : data )
    if ( e%5==0 )
      local.push_back(e);
// create task to add local
    to filtered
```

Output:

```
Mod 5: 5 10 15 20 25
    ↪30 35 40 45 50
    ↪55 60 65 70 75
    ↪80 85 90 95 100
```

List filtering, task-based solution'

The task spins until it's its turn:

Code:

```
#   pragma omp task \
      shared(filtered,ithread)
    {
// wait your turn
      while
     (threadnum>ithread) {
#       pragma omp taskyield
      }
// merge
      filtered += local;
      ithread++;
    }
```

Output:

```
Mod 5: 5 10 15 20 25
    ↪30 35 40 45 50
    ↪55 60 65 70 75
    ↪80 85 90 95 100
```

Templated reductions

You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```
template<typename T>
T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction
                \

    (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))
    \
  initializer(omp_priv=-1.f)

  T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
  for (int id=0; id<tdata.size(); id++)
    tmin = reduce_without_zero<T>(tmin,tdata[id]);
  return tmin;
};
```

which is then called with specific data:

```
tmin = generic_reduction<float>(fdata);
```