Algorithms

Algorithms

- Set of generic functions working on containers
  - cca 90 functions, trivial or sophisticated (sort, make_heap, set_intersection, ...)
    - Some of the algorithms have parallel versions

```
#include <algorithm>
```

- Containers are accessed indirectly - using iterators
    - Typically a pair of iterator specifies a range inside a container
    - Algorithms may be run on complete containers or parts
    - Anything that looks like an iterator may be used
  - Some algorithms are read-only
    - The result is often an iterator
    - E.g., searching in non-associative containers
  - Most algorithms modify the contents of a container
    - Copying, moving (using std::move), or swapping (using std::swap) elements
    - Applying user-defined action on elements (defined by functors)
  - Iterators do not allow insertion/deletion of container elements
    - The space for "new" elements must be created before calling an algorithm
    - Removal of unnecessary elements must be done after returning from an algorithm

Algorithms

- Iterators do not allow insertion/deletion of container elements
  - The space for "new" elements must be created before calling an algorithm

```
my_container c2( c1.size(), 0);
std::copy( c1.begin(), c1.end(), c2.begin());
```

    - Note: This example does not require algorithms:

```
my_container c2( c1.begin(), c1.end());
```

  - Removal of unnecessary elements must be done after returning from an algorithm

```
auto my_predicate = /*...*/; // some condition

my_container c2( c1.size(), 0);  // max size
my_iterator it2 = std::copy_if( c1.begin(), c1.end(), c2.begin(),
my_predicate);
c2.erase( it2, c2.end());    // shrink to really required size

my_iterator it1 = std::remove_if( c1.begin(), c1.end(), my_predicate);
c1.erase( it1, c1.end());    // really remove unnecessary elements
```

STL – Algorithms

- Fake iterators
  - Algorithms can accept anything that works like an iterator
  - The required functionality is specified by iterator category
    - Input, Output, Forward, Bidirectional, RandomAccess
  - Every iterator must specify its category and some other properties
    - std::iterator_traits
    - Some algorithms change their implementation based on the category (std::distance)

  - Inserters

```
my_container c2;    // empty
auto my_inserter = std::back_inserter( c2);
std::copy_if( c1.begin(), c1.end(), my_inserter, my_predicate);
```

  - Text input/output

```
auto my_inserter2 = std::ostream_iterator< int>( std::cout, " ");
std::copy( c1.begin(), c1.end(), my_inserter2);
```

C++20 - ranges

- [C++20] – a pair of iterators replaced by a *range*
  - *range* is anything equipped with begin() and end()
    - Any container is a *range* – this kind of range is the owner of the data!
      - copying such a range copies the data
    - *view range* is a reference to the data – not the owner
      - view range may be copied in constant time
      - all_view(k) is a reference to all elements in a container
      - iota_view(10,20) is a virtual container containing [10,11,...,19]
  - *range adaptor* allows filtration or transformation of data
    - filter_view(range, pred) returns only the elements of range which satisfy pred
    - adapters may be also applied using unix-like pipe syntax:

```
range | filter_view(pred)
```

  - Existing algorithms will be presented also with *range* interfaces
  - *Range* fits into the [C++11] range-based for

```
for ( auto i : std::iota_view(0,n) )   // same as for(int i=0; i<n; ++i)
```

  - *ranges* require *concepts* that are a major language extension [C++20]
    - the implementation of *ranges* is not reliable and complete yet (as of 2020)

Functors

STL – Functors

- Example - for_each

```
template<class InputIterator, class Function>
Function for_each( InputIterator first, InputIterator last, Function f)
{
    for (; first != last; ++first)
        f( * first);
    return f;
}
```

  - f may be anything that supports the operation f(x)
    - a global function (pointer to function), or
    - a *functor*, i.e. a class containing operator()
  - The function f (its operator()) is called for each element in the given range
    - The element is accessed using the * operator which typically return a reference
    - The function f can modify the elements of the container

STL – Algorithms

- A simple application of for_each

```
void my_function( double & x)
{
    x += 1;
}
void increment( std::list< double> & c)
{
    std::for_each( c.begin(), c.end(), my_function);
}
```

- [C++11] Lambda
  - An expression that generates a functor

```
void increment( std::list< double> & c)
{
    for_each( c.begin(), c.end(), []( double & x){ x += 1;});
}
```

STL – Algorithms

- Passing parameters requires a functor

```
class my_functor {
public:
    double v;
    void operator()( double & x) const { x += v; }
    my_functor( double p) : v( p) {}
};
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), my_functor( value));
}
```

- Equivalent implementation using lambda

```
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), [value]( double & x){ x += value;});
}
```

- A functor may modify its contents

```
class my_functor {
public:
    double s;
    void operator()( const double & x) { s += x; }
    my_functor() : s( 0.0) {}
};
double sum( const std::list< double> & c)
{
    my_functor f = std::for_each( c.begin(), c.end(), my_functor());
    return f.s;
}
```

- Using lambda (the generated functor contains a reference to s)

```
double sum( const std::list< double> & c)
{   double s = 0.0;
    for_each( c.begin(), c.end(), [& s]( const double & x){ s += x;});
    return s;
}
```

Lambda [C++11]

Lambda expressions

- Lambda expression [C++11]
- [capture](params) mutable -> rettype { body }
  - Declares a class similar to this sketch:

```
class ftor {
public:
  ftor( TList ... plist) : vlist( plist) ... { }
  rettype operator()( params ) const { body }
private:
  TList ... vlist;
};
```

    - vlist determined by local variables used in the body
    - TList determined by their types and adjusted by capture
    - operator() is const if mutable not present
  - The lambda expression corresponds to creation of an anonymous object

```
ftor( vlist ...)
```

Lambda expressions – return types

- Lambda: The return type of the operator()
  - Explicitly defined

```
[]() -> int { /*...*/ }
```

  - Automatically derived
    - All the return statements inside must return the same type

```
[]() { /*...*/ return /*...*/; }
```

  - void otherwise

```
[]() { /*...*/ }
```

- This return type syntax is also available with named functions [C++14]
  - Explicitly defined
    - Useful when the return type depends on the arguments (via decltype etc.)
    - Or when the return type name is nested in the class scope of the (member) function

```
auto f() -> int;    // same as int f();
```

  - Automatically derived
    - Only in function definition, not in forward declarations
    - Therefore applicable to inline, i.e. short functions

```
auto f() { /*...*/ return /*...*/; }
```

    - Also available as (universal or const) reference

```
auto && g() { /*...*/ return /*...*/; }
```

Lambda expressions – generic parameter types

- Generic lambdas [C++14]
  - Lambda parameters may be declared generic using auto

```
[](auto x, auto && y) { /*...*/ }
```

  - It will generate a functor with generic (template) operator()

```
class ftor { public:
  template< typename T, typename U>
  void operator()(T x, U && y) const
  { /*...*/ }
};
```

  - Explicitly templated lambdas [C++20]
    - C++ has finally merged all the four kinds of parentheses into one syntactic construct!

```
[]< typename T, typename U>(T x, U && y){ /*...*/ }
```

- Conversely, named functions allow auto parameters [C++20]

```
void f(auto x, auto && y)
{ /*...*/ }
```

  - As a shortcut for

```
template< typename T, typename U>
void f(T x, U && y)
{ /*...*/ }
```

Lambda expressions – capture

- Capture
  - Defines which external variables are accessible and how
    - local variables in the enclosing function
    - *this*, if used in a member function
  - Determines the data members of the functor
  - Explicit capture
    - The external variables explicitly listed in *capture*

```
[a,&b,c,&d,this]
```

    - variables marked *&* passed by reference, the others by value
    - when returning lambdas from functions, beware of the lifetime of referenced variables
  - Implicit capture
    - The required external variables determined automatically by the compiler, *capture* defines the mode of passing

```
[=]
[=,&b,&d]
```

    - passed by value, the listed exceptions by reference

```
[&]
[&,a,c]
```

    - passed by reference, the listed exceptions by value
  - Explicitly initialized capture [C++14]

```
[x=std::move(a),&y=b[i]]
```

    - works as if the following declarations were visible inside the lambda

```
auto x=std::move(a); auto && y=b[i];
```

    - [x,&y] is actually a shortcut for [x=x,&y=y]

Lambda expressions

- Every lambda creates a new (anonymous) type
  - Functions accepting lambdas must be templates

```
template< typename F>
void g( F f) // usually passed by value but (F && f) is also used
{
  f(/*...*/);
}
```

  - The functionality of lambda is passed by the type (F), not by the value (f)
    - It is done at compile time – no run-time cost (same as if the function were called directly)
  - At runtime, the values of (or references to) the captured variables are passed (via f)

- Lambda may be held by a local auto variable
  - It acts as a nested function definition
    - With the ability to access the surrounding variables through capture

```
void f(std::ostream & out)
{
   auto print = [& out](auto && x){ out << x << std::endl; };

   for (auto && a : k) { print(a); }
   // same as
   std::for_each( k.begin(), k.end(), print);
}
```

Lambda expressions

- It is difficult (and dangerous) to return lambdas from functions

```
auto h(int p) { return [p](int & x){ x += p}; }
```

- It is difficult to use lambda outside of parameters or local variables
  - e.g. for defining comparison in containers

```
using M = std::map<K,T,decltype([](const K&a, const K&b){ return /*...*/; })>;
```

- It is impossible to directly mix different lambdas in one expression/container
  - Even if lambda functions have the same signature, the generated functors are of different unrelated types

```
cond ? [](int & x){ ++x; } : [](int & x){ --x; } // ERROR

k[0] = [](int & x){ ++x; }; k[1] = [](int & x){ --x; };// ERROR
```

Lambda expressions

- std::function
  - std::function can pack any lambdas of the same signature
    - the signature is specified as a (function) type argument

```
using fnc_t = std::function<void(int&)>;
std::vector<fnc_t> k(2);
```

    - implicit conversion from a lambda to std::function (of the same signature)

```
k[0] = [](int & x){ ++x; };
k[1] = [](int & x){ --x; };
```

  - std::function replaces the compile-time mechanism by a run-time mechanism
    - There is both memory and time cost for using std::function (wrt. lambda)
      - The functor/lambda is copied/moved into a dynamically allocated object
      - std::function acts as a shared owner of this object
    - Use std::function only if necessary
      - Containers containing different functors/lambdas
      - Non-local variables to store
    - std::function may make the lifetime of a lambda long/unpredictable
      - Do not use capture by reference in the lambda
  - std::function itself behaves like a functor
    - It may be used as an argument to algorithms etc.
      - It will be slower than a directly passed lambda