Algorithms

The algorithms category of the C++ STL provides a variety of common programming solutions that operate on ranges of elements within containers.  These algorithms are expressed in the form of functions within the std namespace and are usable with our own collections of types as well as with the STL containers.  Most algorithms include overloads that accept execution policies.  The three libraries in this category are:

* functional - standard function objects
* algorithm - standard algorithms
* numeric - standard numeric operations

Each library is independent of any other library.

This chapter introduces each library of the algorithms category in turn and describes some commonly used features.  This introduction includes descriptions of selected functions with examples.

Functional Library

The functional library defines templated function objects that can be passed as arguments to other functions, including the algorithm templates.  This library is defined in the header file

#include <functional>

and consists of three distinct parts:

* Wrapper Class Templates
* Function Templates
* Operator Classes

Wrapper Class Templates

The wrapper class templates are templates generate callable objects.  These templates include:

* function - creates a function object wrapper for a function, another function object or a lambda expression
* reference\_wrapper - creates a reference object

std::function Wrapper Template

The program on the left creates 3 function objects: one for the function multiply(), one for the functor Multiply, and one for a lambda expression that performs a multiply operation.  The results of calling each wrapper object with the same arguments are listed on the right:

|  |  |
| --- | --- |
| // Functional - function wrapper  // function\_wrapper.cpp  #include <iostream>  #include <functional>  // a simple function  long multiply(long x, long y) { return x \* y; }  // a functor  struct Multiply {  long operator()(long x, long y) { return x \* y; }  };  int main() {  std::function<long(long, long)> f1 = multiply;  std::function<long(long, long)> f2 = Multiply();  std::function<long(long, long)> f3 = [](long x, long y)  { return x \* y; };  std::cout << "f1(10, 2) = " << f1(10, 2) << std::endl;  std::cout << "f2(10, 2) = " << f2(10, 2) << std::endl;  std::cout << "f3(10, 2) = " << f3(10, 2) << std::endl;  } | f1(10, 2) = 20  f2(10, 2) = 20  f3(10, 2) = 20 |

reference Wrapper Template

The reference wrapper template creates a class that facilitates the copying and assigning of references for functions as well as objects.  This template allows utilization of references where the C++ standard forbid normal references (for example, in container classes).

A reference wrapper facilitates the storage of references inside container objects.

The program on the left creates a vector of double objects and a parallel vector of references to those double objects.  It multiplies the values in the original vector by 3 and displays the values referred to by the reference wrapper:

|  |  |
| --- | --- |
| // Functional - reference wrapper  // reference\_wrapper\_vector.cpp  #include <iostream>  #include <vector>  #include <functional>  int main() {  std::vector<double> original(5, 10.3);  std::vector<std::reference\_wrapper<double>>  references(original.begin(), original.end());  for (auto& e : original)  e \*= 3;  for (auto e : references)  std::cout << e << " ";  std::cout << std::endl;  } | 30.9 30.9 30.9 30.9 30.9 |

std::reference\_wrapper can be used anywhere a regular reference can be used.

The following program creates references to 3 longs, resets the referenced values, and displays the value of each reference:

|  |  |
| --- | --- |
| // Functional - reference wrapper  // reference\_wrapper.cpp  #include <iostream>  #include <functional>  int main() {  long v1 = 1L, v2 = 2L, v3 = 3L;  std::reference\_wrapper<long> r1 = v1;  std::reference\_wrapper<long> r2 = v2;  std::reference\_wrapper<long> r3 = v3;  v1 = 10L, v2 = 20L, v3 = 30L;  std::cout << "r1 = " << r1 << std::endl;  std::cout << "r2 = " << r2 << std::endl;  std::cout << "r3 = " << r3 << std::endl;  } | r1 = 10  r2 = 20  r3 = 30 |

Function Templates

The function templates include:

* bind(Fn&& fn, Args&&... args) - binds one or more arguments to a function object
* reference\_wrapper<T> ref(T& t) - creates a reference wrapper on object t of type T
* reference\_wrapper<const T> cref(const T& t) - creates a reference wrapper on unmodifiable object t of type T

bind

The program on the left binds the function multiply() to its arguments and returns the corresponding function object.  When we call the function object, it returns the result for the specified arguments:

|  |  |
| --- | --- |
| // Functional - bind a function to its arguments  // bind.cpp  #include <iostream>  #include <functional>  double multiply(double x, double y) { return x \* y; }  int main() {  auto p = std::bind(multiply, 10, 3);  std::cout << "Product = " << p() << std::endl;  } | Product = 30 |

std::bind stores the arguments passed in by copying their values.

ref

The function template std::ref() returns an std::reference\_wrapper instance for the argument supplied.

The program on the left binds two arguments to function std::increment(), one by reference and the other by value and displays their values after inc() has updated them:

|  |  |
| --- | --- |
| // Functional - create a reference wrapper  // ref.cpp  #include <iostream>  #include <functional>  void increment(int& x, int& y) { ++x, ++y; }  int main() {  int a = 10, b = 20;  auto inc = bind(increment, std::ref(a), b);  inc();  std::cout << "a = " << a << std::endl;  std::cout << "b = " << b << std::endl;  } | a = 11  b = 20 |

Operator Class Templates

The operator class templates define function object equivalents for most of the operators present in the core language.  The templates include:

* bit\_and - result of x & y
* bit\_or - result of x | y
* bit\_xor - result of x ^ y
* divides - result of x / y
* equal\_to - result of x == y
* greater - result of x > y
* greater\_equal - result of x >= y
* less - result of x < y
* less\_equal - result of x <= y
* logical\_and - result of x && y
* logical\_or - result of x || y
* minus - result of x - y
* multiplies - result of x \* y
* negate - result of - x
* not\_equal - result of x != y
* plus - result of x + y

These objects specify the operation to be performed by an algorithm and can be passed to an algorithm in place of a lambda expression or full-blown function object.

Algorithm Library

The algorithm function templates perform common operations on ranges of elements in a sequence.  The function calls accept these ranges as arguments in the form of iterators.  These functions apply, not only to containers, but also to strings and built-in arrays.  Thy do not change the size or storage allocation of any sequence.

The function templates are defined in header file:

#include <algorithm>

The templates consist of:

* Queries
* Modifiers
* Manipulators

The template parameter types include:

* Iterators from one of the six categories that define the operations to be performed:
  + InputIterator - an input iterator type (!=, \*, ->, ++, \*i++)
  + OutputIterator - an output iterator type (write and ++, \*i++)
  + ForwardIterator - an output iterator type (!=, \*, ->, ++, \*i++)
  + BiDirectionalIterator - an output iterator type (!=, \*, ->, ++, \*i++, --)
  + RandomAccessIterator - a random access iterator type (!=, \*, ->, ++, \*i++, --, \*i--, +, i, i[n], <, >, <=, >=)
  + ContiguousIterator - an output iterator type (!=, \*, ->, ++, \*i++, --, \*i--, +, i, i[n], <, >, <=, >=, contiguous storage)
* Fn - a function object type

The notation [f, l) refers to the range starting at the position identified by iterator f and extending to the position immediately before that identified by iterator l.

The admissible operations defined on each type of iterator may be found at [cppreference.com](http://en.cppreference.com/w/cpp/iterator).

Queries

The query class templates include:

* bool all\_of(InputIterator f, InputIterator l, Fn predicate) - true if all within range [f,l) satisfy predicate
* bool any\_of(InputIterator f, InputIterator l, Fn predicate) - true if any within range [f,l) satisfy predicate
* bool none\_of(InputIterator f, InputIterator l, Fn predicate) - true if none within range [f,l) satisfy predicate
* Fn for\_each(InputIterator f, InputIterator l, Fn predicate) - for each within range [f,l) apply predicate
* InputIterator find(InputIterator f, InputIterator l, const T& t) - find 1st element within range [f,l) equal to t
* InputIterator find\_if(InputIterator f, InputIterator l, Fn predicate) - find 1st element within range [f,l) that satisfies predicate
* InputIterator find\_if\_not(InputIterator f, InputIterator l, Fn predicate) - find 1st element within range [f,l) that does not satisfy predicate
* int count(InputIterator f, InputIterator l, const T& t) - count the occurrences of t within range [f,l)
* int count\_if(InputIterator f, InputIterator l, Fn predicate) - count the occurrences of t within range [f,l) that satisfy predicate

count

The program on the left counts the number of occurences of the value 12 in array a:

|  |  |
| --- | --- |
| // Algorithms - Count  // count.cpp  #include <array>  #include <algorithm>  #include <iostream>  int main() {  std::array<int, 11> a = {1, 12, 4, 5, 8, 9, 12, 13,  16, 18, 12};  int n = std::count(a.begin(), a.end(), 12);  std::cout << "12 occurs "<< n << " times" << std::endl;  } | 12 occurs 3 times |

count\_if

The program on the left counts the number of occurences of an even number in the array a:

|  |  |
| --- | --- |
| // Algorithms - Count If  // count\_if.cpp  #include <algorithm>  #include <iostream>  int main() {  int a[] = {1, 2, 4, 5, 8, 9, 12, 13, 16, 18, 22};  int n = std::count\_if(a, a + 11, [](int i)  { return !(i & 1); });  std::cout << "Even Numbers = "<< n << std::endl;  } | Even Numbers = 7 |

Modifiers

The modifier class templates include:

* OutputIterator copy(InputIterator f, InputIterator l, OutputIterator o) - copy all of the elements within range [f,l) into the range starting at o
* OutputIterator copy\_if(InputIterator f, InputIterator l, OutputIterator o, Fn predicate) - copy all of the elements within range [f,l) into the range starting at o that satisfy predicate
* OutputIterator transform(InputIterator f, InputIterator l, OutputIterator o, Fn u) - apply the unary operation u to all of the elements within range [f,l) and store the result starting at o
* OutputIterator transform(InputIterator f, InputIterator l, InputIterator g, OutputIterator o, Fn b) - apply the binary operation b to all of the elements within range [f,l) and corresponding element starting at g and store the result starting at o
* void fill(ForwardIterator f, ForwardIterator l, const T& old\_t, const T& new\_t) - fill every element within range [f,l) with t
* OutputIterator fill\_n(OutputIterator f, size n, const T& t) - fill the 1st n elements starting at element f with t
* void replace(ForwardIterator f, ForwardIterator l, const T& s, const T& t) - replace every occurrence of s with t for all of the elements within range [f,l)
* void replace\_if(ForwardIterator f, ForwardIterator l, Fn predicate, const T& t) - replace every element that satisfies predicate with t for all of the elements within range [f,l)

copy

The program on the left copies the first 2 elements of vector v into vector c, starting at the second element of vector c and displays the updated contents of vector c:

|  |  |
| --- | --- |
| // Algorithms - Copy  // copy\_.cpp  #include <vector>  #include <algorithm>  #include <iostream>  int main() {  std::vector<double> v(4, 10.34);  std::vector<double> c(4, 20.68);  std::copy(v.begin(), v.begin() + 2, c.begin() + 1);  for (auto e : c)  std::cout << e << std::endl;  } | 20.68  10.34  10.34  20.68 |

copy\_if

The program on the left copies from the first 10 elements of vector v those elements that are odd into vector c and displays the all of the elements in c:

|  |  |
| --- | --- |
| // Algorithms - Copy If  // copy\_if.cpp  #include <vector>  #include <algorithm>  #include <iostream>  int main() {  std::vector<int> v = {1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43};  std::vector<int> c(15);  std::copy\_if(v.begin(), v.begin() + 10, c.begin(),  [](int i) -> bool { return i % 2; });  for (auto e : c)  if (e) std::cout << e << std::endl;  } | 1  5  7  13  17  21 |

transform

The transform function templates perform programmer-specified transformations on the elements of a sequence.  A function object defines the transformation.

The program on the left multiplies each element in vector v by 3, stores the result in vector c and displays the contents of vector c:

|  |  |
| --- | --- |
| // Algorithms - Transform - Unary Operation  // transform\_u.cpp  #include <vector>  #include <algorithm>  #include <iostream>  int main() {  std::vector<int> v = {1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43};  std::vector<int> c(11);  std::transform(v.begin(), v.end(), c.begin(),  [](int i) { return 3 \* i; });  for (auto e : c)  std::cout << e << std::endl;  } | 3  6  12  15  21  24  30  39  51  63  129 |

The program on the left adds each element in vector a to the corresponding element in vector b, stores the results in vector c and displays the contents of vector c:

|  |  |
| --- | --- |
| // Algorithms - Transform - Binary Operation  // transform\_b.cpp  #include <vector>  #include <algorithm>  #include <functional>  #include <iostream>  int main() {  std::vector<int> a = {1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43};  std::vector<int> b = {2, 1, 0, 1, 2, 3, 16, 23, 21, 17, 32};  std::vector<int> c(11);  std::transform(a.begin(), a.end(), b.begin(), c.begin(),  std::plus<int>());  for (auto e : c)  std::cout << e << std::endl;  } | 3  3  4  6  9  11  26  36  38  38  75 |

Manipulators

The manipulator class templates include:

* void sort(RandomAccessIterator f, RandomAccessIterator l) - sorts the elements in ascending order
* void sort(RandomAccessIterator f, RandomAccessIterator l, Fn compare) - sorts the elements using compare as the comparator
* OutputIterator merge(InputIterator f1, InputIterator l1, InputIterator f2, InputIterator l2, OutputIterator o) - combine elements in sorted ranges [f1,l1) and [f2,l2) and store the merged results in o in ascending order
* OutputIterator merge(InputIterator f1, InputIterator l1, InputIterator f2, InputIterator l2, OutputIterator o, Fn compare) - combine elements in sorted ranges [f1,l1) and [f2,l2) and store the merged results in o using comparator compare

sort

The program on the left sorts the elements in array a in descending order and displays the sorted result:

|  |  |
| --- | --- |
| // Algorithms - Sort Descending Order  // sort.cpp  #include <iostream>  #include <algorithm>  #include <functional>  int main() {  int a[] = {3, 2, 4, 1};  std::sort(a, &a[4], greater);  for(int e : a)  std::cout << e << std::endl;  } | 4  3  2  1 |

Numeric Library

The numeric library provides standard templated functions for performing numeric operations on ranges of elements in a sequence.

The numeric function templates include:

* T accumulate(InputIterator f, InputIterator l, T init) - accumulate the values in the range [f,l) to init and return the result
* T accumulate(InputIterator f, InputIterator l, T init, Fn boper) - accumulate the values in the range [f,l) to init using the binary operation boper and return the result
* T inner\_product(InputIterator f1, InputIterator l1, InputIterator f2, T init) - accumulate the products of each pair in the range [f,l) to init and return the result
* T inner\_product(InputIterator f1, InputIterator l1, InputIterator f2, T init, Fn boper1, Fn boper2) - accumulate the results of binary operation boper2 on each pair in the range [f,l) to init using binary operation boper1 and return the result
* OutputIterator partial\_sum(InputIterator f1, InputIterator l1, OutputIterator partialSum) - calculate the partial sums for vector v and store them in vector partialSum
* OutputIterator partial\_sum(InputIterator f1, InputIterator l1, OutputIterator partialSum, Fn boper) - calculate the partial expressions for vector v using binary operation boper and store them in vector partialSum

Examples

accumulate

The program on the left sums the elements in array a, displays the result, then sums twice their values and displays that result:

|  |  |
| --- | --- |
| // Algorithms - Accumulate  // accumulate.cpp  #include <iostream>  #include <numeric>  int main() {  int a[] = {3, 2, 4, 1}, s;  s = std::accumulate(a, &a[4], (int)0);  std::cout << "sum = " << s << std::endl;  s = std::accumulate(a, &a[4], (int)0,  [](int x, int y) { return x + 2 \* y; });  std::cout << "2 \* sum = " << s << std::endl;  } | sum = 10  2 \* sum = 20 |

inner\_product

The program on the left calculate the inner or dot product of array a with array b, displays the result, then calculates the sum of the squares of the differences between the elements of the arrays and displays that result:

|  |  |
| --- | --- |
| // Algorithms - Inner Product  // inner\_product.cpp  #include <iostream>  #include <numeric>  #include <functional>  int main() {  int a[] = {3, 2, 4, 1}, b[] = {1, 2, 3, 4}, s;  s = std::inner\_product(a, &a[4], b, (int)0);  std::cout << "dot product = " << s << std::endl;  s = std::inner\_product(a, &a[4], b, (int)0,  std::plus<int>(),  [](int x, int y) { return (x - y) \* (x - y); }  );  std::cout << "sum of "  "(a[i]-b[i])^2 = " << s << std::endl;  } | dot product = 23  sum of  (a[i]-b[i])^2 = 14 |

partial\_sum

The program on the left calculates the partial sums of the elements in vector v, displays them, then calculates their partial products and displays them:

|  |  |
| --- | --- |
| // Algorithms - Partial Sum  // partial\_sum.cpp  #include <iostream>  #include <vector>  #include <functional>  #include <numeric>  int main() {  std::vector<int> v = {1, 2, 3, 4}, p(4);  std::partial\_sum(v.begin(), v.end(), p.begin());  for (auto i : p)  std::cout << i << std::endl;  std::partial\_sum(v.begin(), v.end(), p.begin(),  std::multiplies<int>());  for (auto i : p)  std::cout << i << std::endl;  } | 1  3  6  10  1  2  6  24 |