Intermediate Programming

Day 24

Outline

- Exercise 8-1
- STL classes
- STL algorithms
- Review questions

Exercise 8-1 (part 2)

In the supplied file `sort.cpp`, read an integer from the standard input into `count`.

```
std::cin >> count;
```

Exercise 8-1 (part 3)

Modify the **vec** array so that it stores **count** random values.

```
vec.resize( count );
for( size_t i=0 ; i<count ; i++ ) vec[i] = rand();
```

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array { }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5, 5 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5, 5, 6 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5, 5, 6, 7 }

Exercise 8-1 (part 5)

Define the **`sort`** function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5, 5, 6, 7, 13 }

Exercise 8-1 (part 5)

Define the `sort` function that implements the merge sort algorithm.

Merge Sort:

Given an array of values {1, 27, 7, 5, -2, 6, 5, 3, 13}

- Split in two {1, 27, 7, 5, -2} {6, 5, 3, 13}
- Sort the two halves independently {-2, 1, 5, 7, 27} {3, 5, 6, 13}
- Merge the two sorted halves into a single sorted array {-2, 1, 3, 5, 5, 6, 7, 13, 27}

Exercise 8-1 (part 5)

Define the **sort** function that implements the merge sort algorithm.

```
void sort( std::vector< int > *v )
{
    if( v->size()>1 )
    {
        std::vector< int > left , right;
        left.resize( v->size()/2 ) ; right.resize( v->size() – v->size()/2 );
        for( size_t i=0 ; i<v->size()/2 ; i++ ) left[i] = (*v)[i];
        for( size_t i=v->size()/2 ; i<v->size() ; i++ ) right[ v->size()/2-i ] = (*v)[i];

        sort( &left );
        sort( &right );
```

Exercise 8-1 (part 5)

Define the **sort** function that implements the merge sort algorithm.

```
        size_t idx=0 , i=0 , j=0;
        while( i<left.size() || j<right.size() )
        {
            if      ( i>=  left.size() )  (*v)[idx++] = right[j++];
            else if( j>=right.size() )  (*v)[idx++] =  left[i++];
            else if( left[i]<right[j] ) (*v)[idx++] =  left[i++];
            else                        (*v)[idx++] = right[j++];
        }
    }
}
```

Outline

- Exercise 8-1
- STL classes
- STL algorithms
- Review questions

STL classes (std::pair)

- The std::pair class is a container storing two objects of (possibly) different types
  - Members:
    - first: the first object
    - second: the second object
  - Function std::make_pair
    - constructs a pair with the prescribed values

```
…
template< class T1 , class T2 >
struct pair
{
      T1 first;
      T2 second;
      …
};
template< class T1 , class T2 >
std::pair< T1 , T2 > make_pair( T1 t1 , T2 t2 );
```

STL classes (std::pair)

- The std::pair class is a container storing two objects of (possibly) different types
  - In C, if we wanted a function to return multiple objects, we would need to pass pointers to the function which would then be dereferenced

```
#include <stdio.h>
void divmod( int a , int b, int *quo , int *rem )
{
	*quo = a / b;
	*rem = a % b;
}
int main( void )
{
	int q , r;
	divmod( 10 , 3 , &q , &r );
	printf( "10 = 3 * %d + %d\n" ,  q , r );
	return 0;
}
```

STL classes (std::pair)

- The std::pair class is a container storing two objects of (possibly) different types
  - In C, if we wanted a function to return multiple objects, we would need to pass pointers to the function which would then be dereferenced
  - In C++ we can return a std::pair

```
#include <iostream>
using std::cout ; using std::endl;
std::pair< int , int > divmod( int a , int b )
{
	return std::make_pair( a/b , a%b );
}
int main( void )
{
	std::pair< int , int > qr = divmod( 10 , 3 );
	cout << "10 = 5 * " << qr.first << " + " << qr.second << endl;
	return 0;
}
```

STL classes (std::tuple)

- The std::tuple class is a more general version storing multiple objects of (possibly) different types

```
#include <iostream>
#include <tuple>
using std::cout ; using std::endl;
std::tuple< int , int , float > divmod( int a , int b )
{
      return std::make_tuple( a/b , a%b , (float)a/b );
}
int main( void )
{
       std::tuple< int , int , float > qr = divmod( 10 , 3 );
      cout << "10/3 quotient=" << std::get< 0 >( qr ) << endl;
      cout <<"         remainder=" << std::get< 1 >( qr ) << endl;
      cout << ", decimal quotient=" << std::get< 2 >( qr ) << endl;
      return 0;
}
```

STL classes (std::tuple)

- The std::tuple class is a more general version storing multiple objects of (possibly) different types
  - The number of objects is defined by the number of parameters

```
#include <iostream>
#include <tuple>
using std::cout ; using std::endl;
std::tuple< int , int , float > divmod( int a , int b )
{
      return std::make_tuple( a/b , a%b , (float)a/b );
}
int main( void )
{
       std::tuple< int , int , float > qr = divmod( 10 , 3 );
      cout << "10/3 quotient=" << std::get< 0 >( qr ) << endl;
      cout <<"         remainder=" << std::get< 1 >( qr ) << endl;
      cout << ", decimal quotient=" << std::get< 2 >( qr ) << endl;
      return 0;
}
```

STL classes (std::tuple)

- The std::tuple class is a more general version storing multiple objects of (possibly) different types
  - The number of objects is defined by the number of parameters
  - std::make_tuple constructs a std::tuple

```
#include <iostream>
#include <tuple>
using std::cout ; using std::endl;
std::tuple< int , int , float > divmod( int a , int b )
{
      return std::make_tuple( a/b , a%b , (float)a/b );
}
int main( void )
{
       std::tuple< int , int , float > qr = divmod( 10 , 3 );
      cout << "10/3 quotient=" << std::get< 0 >( qr ) << endl;
      cout <<"         remainder=" << std::get< 1 >( qr ) << endl;
      cout << ", decimal quotient=" << std::get< 2 >( qr ) << endl;
      return 0;
}
```

STL classes (std::tuple)

- The std::tuple class is a more general version storing multiple objects of (possibly) different types
  - The number of objects is defined by the number of parameters
  - std::make_tuple constructs a std::tuple
  - std::get< 0 >( ), etc. return access to the std::tuple's member objects
    - The indices cannot be variables

```
#include <iostream>
#include <tuple>
using std::cout ; using std::endl;
std::tuple< int , int , float > divmod( int a , int b )
{
      return std::make_tuple( a/b , a%b , (float)a/b );
}
int main( void )
{
       std::tuple< int , int , float > qr = divmod( 10 , 3 );
      cout << "10/3 quotient=" << std::get< 0 >( qr ) << endl;
      cout <<"         remainder=" << std::get< 1 >( qr ) << endl;
      cout << ", decimal quotient=" << std::get< 2 >( qr ) << endl;
      return 0;
}
```

STL classes (std::pair and std::tuple)

- Both std::pair and std::tuple define (overload) the "<" relation that compares two objects by comparing their objects lexicographically using the object-specific "<" relation*
  - It's OK to construct std::pairs whose parameter classes don't define a "<" relation
    (as long as you don't try to do stuff like sort the std::pairs)

```
…
template< class T1 , class T2 >
struct pair
{
        T1 first;
        T2 second;
        bool operator < ( const pair& p ) const
        {
                if( first<p.first ) return true;
                if( p.first<first ) return false;
                return second<p.second;
        }
        …
};
```

*More on this later

STL classes (std::pair and std::tuple)

- Both std::pair and std::tuple define (overload) the "<" relation that compares two objects by comparing their objects lexicographically using the object-specific "<" relation*
  - It's OK to construct std::pairs whose parameter classes don't define a "<" relation (as long as you don't try to do stuff like sort the std::pairs)

```
…
template< class T1 , class T2 >
struct pair
{
        T1 first;
        T2 second;
        operator < ( const pair& p ) const

        if( first<p.first ) return true;
        if( p.first<first ) return false;
        return second<p.second;

        …
};
```

See:
http://www.cplusplus.com/reference/utility/pair/
http://www.cplusplus.com/reference/tuple/
for more std::pair and std::tuple functionality

*More on this later

STL classes (std::map)

- A std::map is a list of key/value pairs -- each element (key) has a unique value
  - The template parameters specify the key / value types
    - Key can be any type for which the operator "<" compares two values
    - Value can be any type

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main( void )
{
	std::map< int , std::string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[92394] = "George Washington";
	std::cout << "size: " << i2n.size() << std::endl;
	std::cout << "name[92394] " << i2n[92394] << std::endl;
	return 0;
}
```

STL classes (std::map)

- A std::map is a list of key/value pairs -- each element (key) has a unique value
  - The template parameters specify the key / value types
  - [ ] operator:
    - accesses (and creates) an entry associated with a key

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main( void )
{
	std::map< int , std::string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[92394] = "George Washington";
	std::cout << "size: " << i2n.size() << std::endl;
	std::cout << "name[92394] " << i2n[92394] << std::endl;
	return 0;
}
```

STL classes (std::map)

- A std::map is a list of key/value pairs -- each element (key) has a unique value
  - The template parameters specify the key / value types
  - [ ] operator:
    - accesses (and creates) an entry associated with a key
  - size:
    - returns the number of pairs in the list

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main( void )
{
	std::map< int , std::string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[92394] = "George Washington";
	std::cout << "size: " << i2n.size() << std::endl;
	std::cout << "name[92394] " << i2n[92394] << std::endl;
	return 0;
}
```

STL classes (std::map)

- A std::map is a list of key/value pairs -- each element (key) has a unique value
  - The template parameters specify the key / value types
  - [ ] operator:
    - accesses (and creates) an entry associated with a key
  - size:
    - returns the number of pairs in the list

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main( void )
{
      std::map< int , std::string > i2n;
      i2n[92394] = "Alex Hamilton";
      i2n[13522] = "Ben Franklin";
      i2n[92394] = "George Washington";
      std::cout << "size: " << i2n.size() << std::endl;
      std::cout << "name[92394] " << i2n[92394] << std::endl;
      return 0;
}
```

```
>> ./a.out
size: 2
name[92394] George Washington
>>
```

STL classes (std::map)

- A std::map is a list of key/value pairs -- each element (key) has a unique value
  - The template parameters specify the key / value types
  - [ ] operator:
    - accesses (and creates) an entry associated with a key
  - size:
    - returns the number of pairs in the list

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main( void )
{
      std::map< int , std::string > i2n;
      i2n[92394] = "Alex Hamilton";
      i2n[13522] = "Ben Franklin";
      i2n[92394] = "George Washington";
      std::cout << "size: " << i2n.size() << std::endl;
      std::cout << "name[92394] " << i2n[92394] << std::endl;
      return 0;
```

```
>> ./a.out
size: 2
name[92394] George Washington
>>
```

See:
http://www.cplusplus.com/reference/map/map/
for more std::map functionality

STL classes (std::map)

- begin / end
  - return iterators to the first / last elements of the list

```
...
int main( void )
{
	std::map< int , string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[42345] = "George Washington";
	for( std::map< int , string >::iterator it=i2n.begin() ; it!=i2n.end() ; ++it )
		std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

STL classes (std::map)

- begin / end
  - return iterators to the first / last elements of the list
    - These are objects of class std::map< KeyT , ValueT >::iterator
      They act like pointers to objects of type std::pair< KeyT , ValueT >

```
...
int main( void )
{
	std::map< int , string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[42345] = "George Washington";
	for( std::map< int , string >::iterator it=i2n.begin() ; it!=i2n.end() ; ++it )
		std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

STL classes (std::map)

- begin / end
  - return iterators to the first / last elements of the list
    - These are objects of class std::map< KeyT , ValueT >::iterator
      They act like pointers to objects of type std::pair< KeyT , ValueT >
      - Access first / second members via "->"

```
...
int main( void )
{
	std::map< int , string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[42345] = "George Washington";
	for( std::map< int , string >::iterator it=i2n.begin() ; it!=i2n.end() ; ++it )
		std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

STL classes (std::map)

- begin / end
  - return iterators to the first / last elements of the list
    - These are objects of class std::map< KeyT , ValueT >::iterator
      They act like pointers to objects of type std::pair< KeyT , ValueT >
      - Access first / second members via "->"
      - Advance to the next iterator using "++"

```
...
int main( void )
{
	std::map< int , string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[42345] = "George Washington";
	for( std::map< int , string >::iterator it=i2n.begin() ; it!=i2n.end() ; ++it )
		std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

STL classes (std::map)

- begin / end
  - return iterators to the first / last elements of the list
    - These are objects of class std::map< KeyT , ValueT >::iterator
      They act like pointers to objects of type std::pair< KeyT , ValueT >
      - Access first / second members via "->"
      - Advance to the next iterator using "++"
  - Keys are stored in sorted order (using the "<" relation for the key)

```
>> ./a.out
13522: Ben Franklin
42345: George Washington
92394: Alex Hamilton
>>
```

```
…
int main( void )
{
        std::map< int , string > i2n;
        i2n[92394] = "Alex Hamilton";
        i2n[13522] = "Ben Franklin";
        i2n[42345] = "George Washington";
        for( std::map< int , string >::iterator it=i2n.begin() ; it!=i2n.end() ; ++it )
                std::cout << it->first << ": " << it->second << std::endl;
        return 0;
}
```

STL classes (std::map)

- begin / end
- find
  - returns an iterator to the entry with the specified key
    or returns an iterator equal to end if the key is not in the map

```
>> ./a.out
42345: George Washington
>>
```

```
…
using namespace std;
int main( void )
{
	std::map< int , std::string > i2n;
	i2n[92394] = "Alex Hamilton";
	i2n[13522] = "Ben Franklin";
	i2n[42345] = "George Washington";
	std::map< int , string >::iterator it = i2n.find( 42345 );
	if( it!=i2n.end() ) std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

STL classes (std::map)

- The iterator type can be rather complex
  - std::map< int , string >::iterator
    - iterator over single std::map
  - std::map< string , std::map< string , int > >::iterator
    - iterator over a std::map where the values are themselves maps

- typedefing can help by:
  - Reducing clutter
  - Bringing the iterator and object type declarations closer together in the code
    - Changing one usually requires changing the other

STL classes (std::map)

- The iterator type can be rather complex
  - std::map< int , string >::iterator
    - iterator over single std::map
  - std::map< string , std::map< st
    - iterator over a std::map where the
- typedefing can help by:
  - Reducing clutter
  - Bringing the iterator and object ty
    - Changing one usually requires chang

main.cpp

```
#include <iostream>
#include <map>
int main( void )
{
	typedef std::map< int , std::string > i2s_map;
	typedef i2s_map::iterator i2s_iter;
	i2s_map i2s;
	i2s[92394] = "Alex Hamilton";
	i2s[13522] = "Ben Franklin";
	i2s[42345] = "George Washington";
	for( i2s_iter it=i2s.begin() ; it!=i2s.end() ; it++ )
		std::cout << it->first << ": " << it->second << std::endl;
	return 0;
}
```

Outline

- Exercise 8-1
- STL classes
- STL algorithms
- Review questions

STL algorithms

STL defines a **std::sort** function in the **algorithm** header

```
template< class Iterator >
void sort( Iterator begin , Iterator end );
```

- Modifies the order of elements in a vector, arranging elements in ascending order according to the "<" relation
  - For numbers "<" means "less than"
  - For strings "<" means "earlier lexicographically"
  - For your class, "<" means (almost) whatever you want:
    - The **sort** function will arrange elements according to whatever rule you prescribe

Not all containers support sorting – the associated iterator needs to support “random access”

STL algorithms

STL defines a **std::sort** function in the **algorithm** header

```
template< c
void sort( I
```

- Modifies the order of el
  order according to the "
  - For numbers "<" means "
  - For strings "<" means "e
  - For your class, "<" means
    - The **sort** function will

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main( void )
{
        std::vector< float > grades;
        float grade;
        while( std::cin >> grade ) grades.push_back( grade );
        std::sort( grades.begin() , grades.end() );
        std::cout << "Median grade: " << grades[ grades.size()/2 ] << std::endl;
        return 0;
}
```

```
>> echo 1 5 3 9 3 9 | ./a.out
Median grade: 5
>>
```

STL algorithms

STL defines a **std::find** function in the **algorithm** header

```
template< class Iterator , class T >
Iterator find( Iterator first , Iterator last , const T &val );
```

- Returns an iterator to the first element in the range [**first**,**last**) that compares equal to **val**.

STL algorithms

STL defines a **std::find** function in the **algorithm** header

```
template< class Iterator , class T >
Iterato
```

- Returns ar
  compares

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main( void )
{
	std::vector< int > values;
	int v;
	while( std::cin >> v ) values.push_back( v );
	std::cout << std::find( values.begin() , values.end() , 9 ) - values.begin() << std::endl;
	return 0;
}
```

```
>> echo 1 5 3 9 3 9 | ./a.out
3
>>
```

STL algorithms

STL defines a **std::count** function in the **algorithm** header

```
template< class Iterator , class T >
typename iterator_traits< Iterator >::difference_type
count( Iterator first , Iterator last , const T &val );
```

- Returns the number of elements in the range [**first**,**last**) that compares equal to **val**.
  - The return type will depend on the particular type of iterator being used.

STL algorithms

STL defines a **std::count** function in the **algorithm** header

```
template< class Iterator , class T >
typename iterator
count( Iterator fi
```

- Returns the number o
  equal to **val**.
  - The return type will de

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main( void )
{
	std::vector< int > values;
	int v;
	while( std::cin >> v ) values.push_back( v );
	std::cout << std::count( values.begin() , values.end() , 9 ) << std::endl;
	return 0;
}
```

```
>> echo 1 5 3 9 3 9 | ./a.out
2
>>
```

STL algorithms

STL defines a **std::is_permutation** function in the **algorithm** header

```
template< class Iterator1 , class Iterator2 >
bool is_permutation( Iterator1 first1 , Iterator1 last1 ,
                     Iterator2 first2 , Iterator2 last 2 );
```

- Returns **true** if there exists a permutation of the elements in [**first1**,**last1**) that makes the range equal to the range [**first1**,**last2**).
  - Elements are compared using the == operator.

STL algorithms

STL defines a **std::is_permutation** function in the **algorithm** header

```
template< class Iterator1 , class Iterator2 >
bool
```

- Returns
  that ma
  - Elem

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main( void )
{
	std::vector< int > v1 = { 1 , 1 , 2 , 3 , 5 , 8 , 13 };
	std::vector< int > v2 = { 13 , 8 , 5 , 3 , 2 , 1 , 1 };
	std::vector< int > v3 = { 1 , 2 , 3 , 4 , 5 , 6 , 7 };
	std::cout << std::is_permutation( v1.begin() , v1.end() , v2.begin() , v2.end() ) << " ";
	std::cout << std::is_permutation( v1.begin() , v1.end() , v3.begin() , v3.end() ) << std::endl;
	return 0;
}
```

```
>> ./a.out
1 0
>>
```

Outline

- Exercise 8-1
- STL classes
- STL algorithms
- Review questions

Review questions

1. What is `std::map` in C++ STL? What is the difference between `std::pair` and `std::tuple`?

A `std::map` is a collection of unique keys, each with an associated value.

A `std::pair` is a heterogenous container storing exactly two values. A `std::tuple` stores an arbitrary number of values.

Review questions

2. How do you return multiple values in C++?

`std::pair` or `std::tuple`

Review questions

3. Name some useful templated data containers provided by STL

std::vector, std::map, std::pair, std::tuple, std::list

Review questions

4. Name some useful algorithms provided by <algorithm>.

std::sort, std::find, std::count

Review questions

5. What's the difference between an `iterator` and a `const_iterator`?

With a `const_iterator`, you are not allowed to change the contents.

Exercise 8-2

- Website -> Course Materials -> ex8-2