נושאים מתקדמים בתכנות מונחה עצמים
הרצאה 2

פרופ' עפר שיר

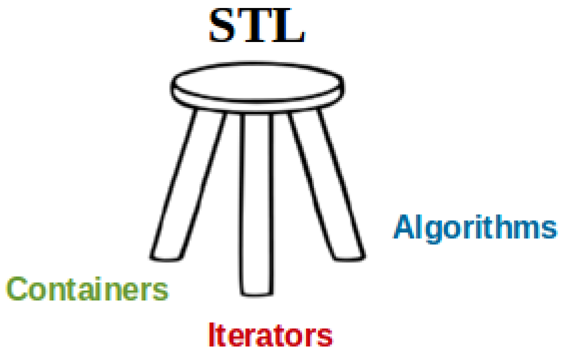
ofersh@telhai.ac.il

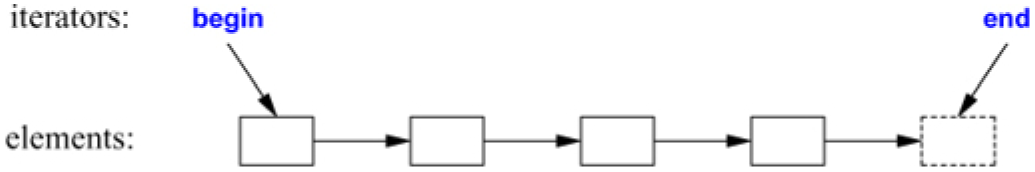
החוג למדעי המחשב

תל-חי
המכללה האקדמית

מבנה ההרצאה

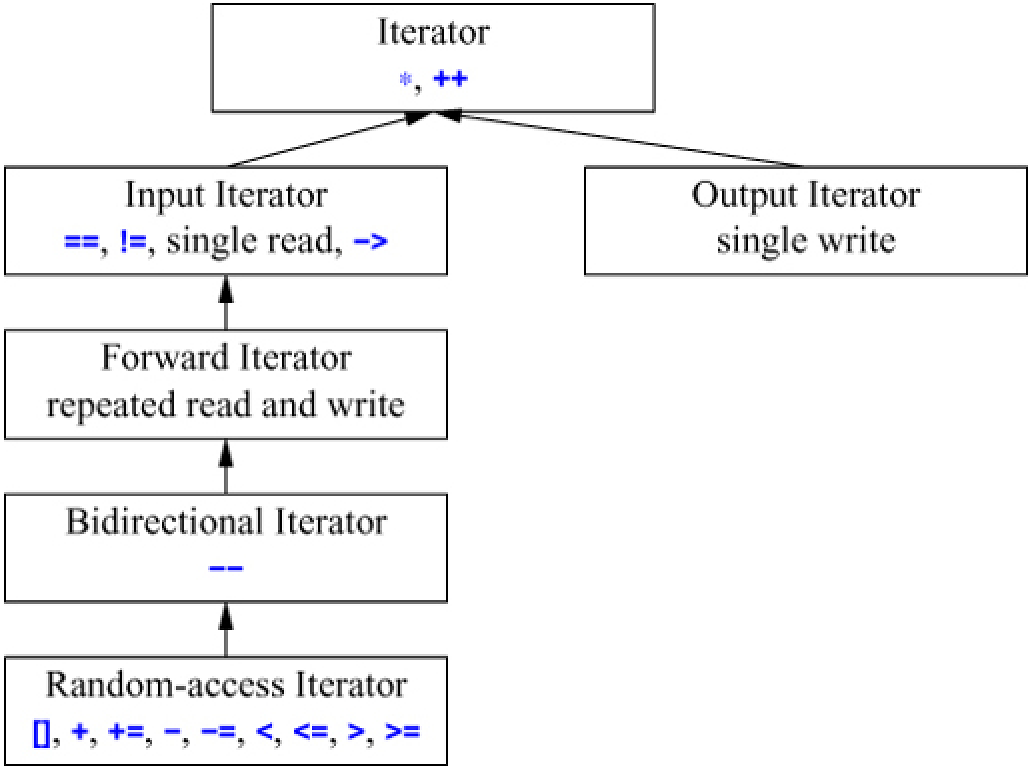
- Comparators
- Predicators
- Containers
- Algorithms



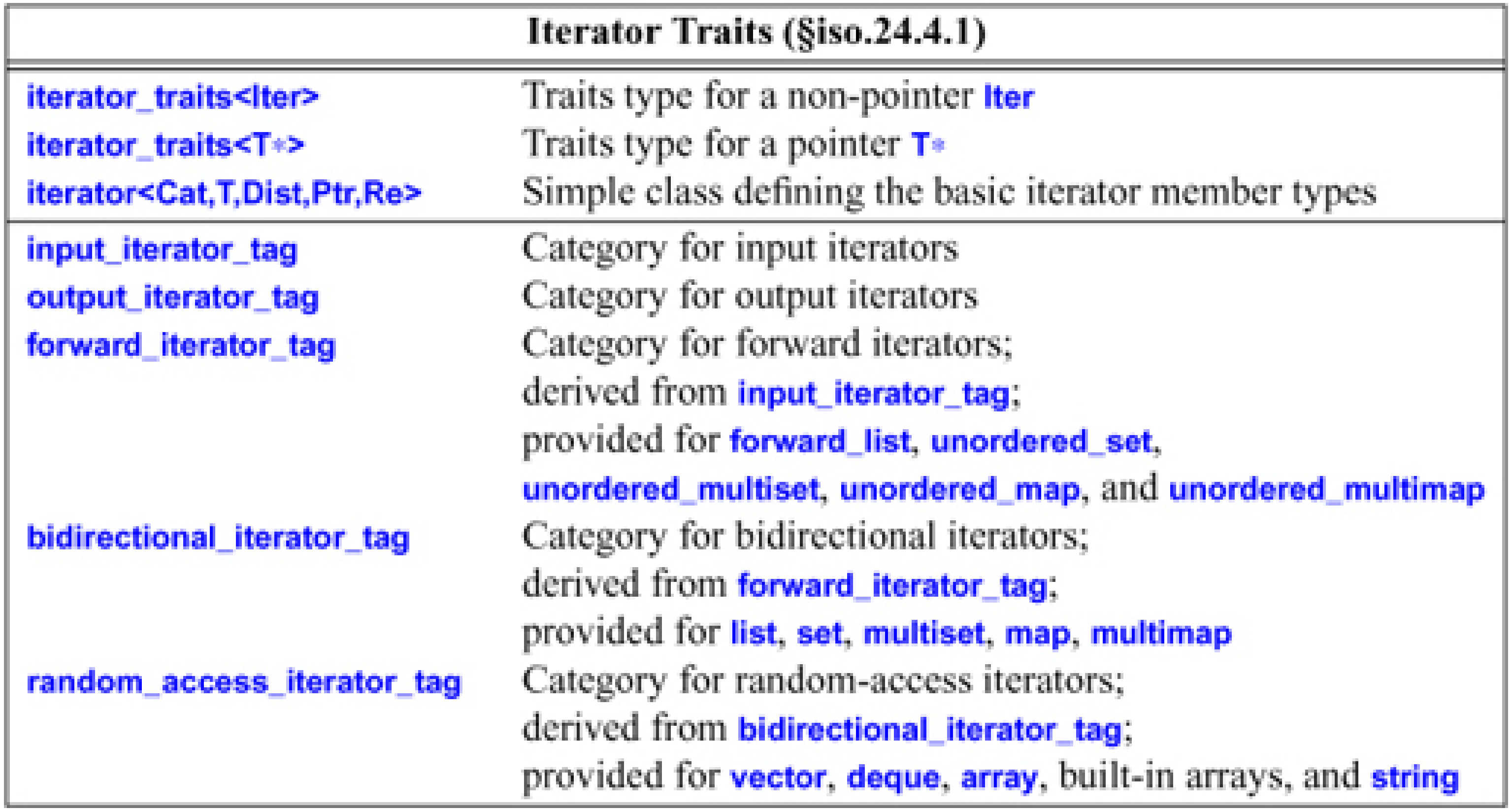


ITERATORS: RECAP

היררכיה


Iterator
*, ++
Input Iterator
==, !=, single read, ->
Output Iterator
single write
Forward Iterator
repeated read and write
Bidirectional Iterator
--
Random-access Iterator
[], +, +=, -, -=, <, <=, >, >=

Iterator Traits & Tags



Iterator Traits (§iso.24.4.1)	
iterator_traits<Iter>	Traits type for a non-pointer Iter
iterator_traits<T*>	Traits type for a pointer T*
iterator<Cat,T,Dist,Ptr,Re>	Simple class defining the basic iterator member types
input_iterator_tag	Category for input iterators
output_iterator_tag	Category for output iterators
forward_iterator_tag	Category for forward iterators; derived from input_iterator_tag; provided for forward_list, unordered_set, unordered_multiset, unordered_map, and unordered_multimap
bidirectional_iterator_tag	Category for bidirectional iterators; derived from forward_iterator_tag; provided for list, set, multiset, map, multimap
random_access_iterator_tag	Category for random-access iterators; derived from bidirectional_iterator_tag; provided for vector, deque, array, built-in arrays, and string

struct iterator

לסיכום, מבנה האיטרטור הכללי מאגד את התכונות המוזכרות
לכדי **struct** בסיסי עם ערכי ברירת מחדל:

```
template<typename Cat, typename T, typename Dist = ptrdiff_t,
typename Ptr = T*, typename Ref = T&>

struct iterator {

    using value_type = T;

    using difference_type = Dist ;    // type used by distance()

    using pointer = Ptr;              // pointer type

    using reference = Ref;            // reference type

    using iterator_category = Cat;    // category (tag)

};
```

Alias-declaration **in C++11 is equivalent to a** ***typedef-name***

Iterator Traits

לשם השגת גנריות מלאה, STL מספקת מחלקת תבנית לייצוג כל התכונות האפשריות של האיטרטור:

```
namespace std {
    template <class T>
    struct iterator_traits {
        typedef typename T::value_type            value_type;
        typedef typename T::difference_type       difference_type;
        typedef typename T::iterator_category     iterator_category;
        typedef typename T::pointer               pointer;
        typedef typename T::reference             reference;
    };
}
```

T מייצג אובייקט איטרטור, כך שהמבנה מבטיח שכל טיפוסי המשתנים הללו מוגדרים היטב

Specialization for Pointers

```
namespace std {
    template <class T>
    struct iterator_traits<T*> {
        typedef T                          value_type;
        typedef std::ptrdiff_t             difference_type;
        typedef random_access_iterator_tag iterator_category;
        typedef T*                         pointer;
        typedef T&                         reference;
    };
}
```

- הייחוד הנ"ל מאפשר לראות במצביעים למערך כאיטרטורים מטיפוס random-access
- כך הושגה עקביות עבור מצביעים פרימיטיביים (אשר אינם מכילים את הטיפוסים הנ"ל) ועבור אובייקטי איטרטור של השפה

כתיבת פונקציה גנרית עבור איטרטורים

```
template<typename Iter> // NOT GENERAL
typename Iter::value_type read(Iter p, int n) {
     // ... do some checking ...
     return p[n];
}
```

← הרעיון הוא לבדוק את תכונות האיטרטור, במבנה
iterator_traits, במקום את האיטרטור עצמו:

```
template<typename Iter> // More general
typename iterator_traits<Iter>::value_type read(Iter p, int n)
{
     // ... do some checking ...
     return p[n];
}
```

תכונות נוספות: מרחק בין איטרטורים

תכונת איטרטור כללית היא הגדרת המרחק במרחב הכתובות,
מטיפוס **difference_type**, באמצעות **std::distance** :

```
template<typename Iter>
void f(Iter p, Iter q) {

/* First attempt: SYNTAX ERROR: "typename" missing */
  Iter::difference_type d1 = std::distance(p,q);

/* Second attempt: wouldn’t work for pointers! */
  typename Iter::difference_type d2 = std::distance(p,q);

/* Third attempt: OKAY */
  typename iterator_traits<Iter>::difference_type d3 =
        std::distance(p,q);

    // ...

}
```

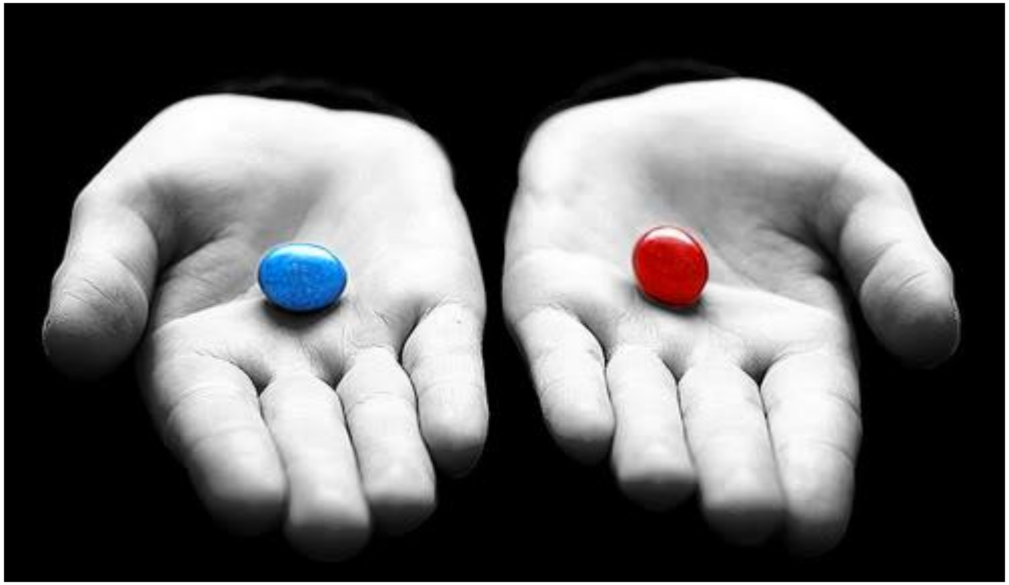
כתיבת פונקציה גנרית עבור איטרטורים

```
template <typename Itr>
inline void my_func (Itr begin, Itr end)
{
  func_helper (begin, end,
      std::iterator_traits<Itr>::iterator_category{}
);
}
```

```
template <typename BidectionalIterator>
void func_helper (BidectionalIterator begin,
                  BidirectionalIterator end,
                  std::bidirectional_iterator_tag)
 {
       //Bidirectional Iterator specific code is here
 }

template <typename RandomIterator>
void  func_helper(RandomIterator begin,
                  RandomIterator end,
                  std::random_access_iterator_tag)
 {

       // Random access Iterator specific code is here
 }
```



COMPARATORS

Comparators

- Boolean *functors* representing the comparison criterion among elements (keys)
- Essential in sorting algorithms
  - `std::sort`
- Essential in ordered containers
  - `std::set`, `std::multiset`
  - `std::map`, `std::multimap`
  - `priority_queue` (later today…)

Course #121503 Revisited

```
// Compare object: ordering by length.
class LessThanByLength
{
  public:
    bool operator()( const Rectangle & lhs,
                       const Rectangle & rhs ) const
      { return lhs.getLength( ) < rhs.getLength( ); }
};

// Compare object: ordering by area.
class LessThanByArea
{
  public:
    bool operator()( const Rectangle & lhs,
                       const Rectangle & rhs ) const
      { return lhs.getLength( ) * lhs.getWidth( ) <
               rhs.getLength( ) * rhs.getWidth( ); }
};
```

Generic **`findMax`**

```
/* Generic findMax, with a function object.
   Precondition: a.size( ) > 0. */
template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a,
                        const Comparator& isLessThan)
{
    int maxIndex = 0;
    for( int i = 1; i < a.size( ); i++ )
        if( isLessThan( a[maxIndex], a[i]) )
            maxIndex = i;
    return a[ maxIndex ];
}
```

test_mm.cpp

```
#include <iostream>
#include <map>
using std::cout;
using std::endl;
typedef std::multimap< int, double, std::less< int > > mmid;
int main(void) {
   mmid pairs;
   cout << "There are currently " << pairs.count( 15 )<< " pairs
with key 15 in the multimap\n";
   pairs.insert( mmid::value_type( 15, 2.7 ) );
   pairs.insert( mmid::value_type( 15, 99.3 ) );
   cout << "After inserts, there are " << pairs.count( 15 ) << "
pairs with key 15\n\n";
   pairs.insert( mmid::value_type( 30, 111.11 ) );
   pairs.insert( mmid::value_type( 10, 22.22 ) );
   pairs.insert( mmid::value_type( 25, 33.333 ) );
   pairs.insert( mmid::value_type( 20, 9.345 ) );
   pairs.insert( mmid::value_type( 5, 77.54 ) );
   cout << "Multimap pairs contains:\nKey\tValue\n";
   for ( mmid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
   cout << iter->first << '\t’<< iter->second << '\n';
   cout << endl;
   return 0;
} // end main
```

```
There are currently 0 pairs with
key 15 in the multimap
After inserts, there are 2 pairs
with key 15

Multimap pairs contains:
Key       Value
5         77.54
10        22.22
15        2.7
15        99.3
20        9.345
25        33.333
30        111.11
```

PREDICATORS

predicate (n.), predicate (v.), predicator (n.)

- Predicate (noun) [GRAMMER]
  1. the part of a sentence or clause containing a verb and stating something about the subject
- Predicate (verb) [GRAMMER] [LOGIC]
  1. state, affirm, or assert (something) about the subject of a sentence or an argument of proposition
  2. found or base something on

Predicators

- Boolean *functors* that assert a prescribed condition
  - Often to verify some conditioning on a given operation:

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

21.5.3 `remove`, `remove_if`, `remove_copy` & `remove_copy_if`

- **`remove`**
  - **`remove( iter1, iter2, value);`**
  - Removes all instances of **`value`** in range (**`iter1-iter2`**)
    - Moves instances of **`value`** towards end
    - Does not change size of container or delete elements
  - Returns iterator to "new" end of container
  - Elements after new iterator are undefined (**`0`**)
- **`remove_copy`**
  - Copies one vector to another while removing an element
  - **`remove_copy(iter1, iter2, iter3, value);`**
    - Copies elements not equal to **`value`** into **`iter3`** (output iterator)
    - Uses range **`iter1-iter2`**

21.5.3 remove, remove_if, remove_copy & remove_copy_if

- **remove_if**
  - Like **remove**
    - Returns iterator to last element
    - Removes elements that return **true** to the specified predicator

```
remove_if(iter1,iter2, predicat);
```

- **remove_copy_if**
  - Like **remove_copy** and **remove_if**
  - Copies range of elements to **iter3**, except those for which **predicator** returns **true**

```
remove_copy_if(iter1,iter2,iter3, predicat);
```

```cpp
// Fig. 21.28: fig21_28.cpp
// Standard library functions remove, remove_if,
// remove_copy and remove_copy_if.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>   // algorithm definitions
#include <vector>      // vector class-template definition

bool greater9( int );  // prototype

int main(void)
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };

   std::ostream_iterator< int > output( cout, " " );

   std::vector< int > v( a, a + SIZE );
   std::vector< int >::iterator newLastElement;

   cout << "Vector v before removing all 10s:\n   ";
   std::copy( v.begin(), v.end(), output );

   // remove 10 from v
   newLastElement = std::remove( v.begin(), v.end(), 10 );

   cout << "\nVector v after removing all 10s:\n
   std::copy( v.begin(), newLastElement, output );

   std::vector< int > v2( a, a + SIZE );
   std::vector< int > c( SIZE, 0 );

   cout << "\n\nVector v2 before removing all 10s "
        << "and copying:\n   ";
   std::copy( v2.begin(), v2.end(), output );

   // copy from v2 to c, removing 10s in the process
   std::remove_copy( v2.begin(), v2.end(), c.begin(), 10 );

   cout << "\nVector c after removing all 10s from v2:\n   ";
   std::copy( c.begin(), c.end(), output );

   std::vector< int > v3( a, a + SIZE );

   cout << "\n\nVector v3 before removing all elements"
        << "\ngreater than 9:\n   ";
   std::copy( v3.begin(), v3.end(), output );

```

Remove all **`10`**'s from **`v`**. Returns an iterator pointing to the new last element.

Use **`remove_copy`** to create a duplicate of **`v`**, with all the **`10`**'s removed.

```
   // remove elements greater than 9 from v3
   newLastElement =
      std::remove_if( v3.begin(), v3.end(), greater9 );

   cout << "\nVector v3 after removing all elements"
        << "\ngreater than 9:\n   ";
   std::copy( v3.begin(), newLastElement, output );

   std::vector< int > v4( a, a + SIZE );
   std::vector< int > c2( SIZE, 0 );

   cout << "\n\nVector v4 before removing all elements"
        << "\ngreater than 9 and copying:\n   ";
   std::copy( v4.begin(), v4.end(), output );

   // copy elements from v4 to c2, removing elements greater
   // than 9 in the process
   std::remove_copy_if(
      v4.begin(), v4.end(), c2.begin(), greater9 );

   cout << "\nVector c2 after removing all elements"
        << "\ngreater than 9 from v4:\n   ";
   std::copy( c2.begin(), c2.end(), output );

```

Use function **`greater9`** to determine whether to remove the element.

Note use of **`remove_copy_if`**.

```
   cout << endl;

   return 0;

} // end main

// determine whether argument is greater than 9
bool greater9( int x )
{
   return x > 9;

} // end greater9
```

```
Vector v before removing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v after removing all 10s:
   2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c after removing all 10s from v2:
   2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements
greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after removing all elements
greater than 9:
   2 4 6 8

Vector v4 before removing all elements
greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after removing all elements
greater than 9 from v4:
   2 4 6 8 0 0 0 0 0 0
```

21.5.4 `replace`, `replace_if`, `replace_copy` & `replace_copy_if`

- Functions

```
replace( iter1, iter2, value, newvalue );
```

  - Like **`remove`**, except replaces **`value`** with **`newvalue`**

```
replace_if( iter1, iter2, predicator,
  newvalue );
```

  - Replaces value if **`predicator`** returns **`true`**

```
replace_copy(iter1, iter2, iter3, value,
   newvalue);
```

  - Replaces and copies elements to **`iter3`**
  - Does not affect originals

```
replace_copy_if( iter1, iter2, iter3,
  predicator, newvalue );
```

  - Replaces and copies elements to **`iter3`** if **`predicator`** returns **`true`**

```
  // Fig. 21.29: fig21_29.cpp
  // Standard library functions replace, replace_if,
  // replace_copy and replace_copy_if.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>
#include <vector>

bool greater9( int );

int main(void)
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };

   std::ostream_iterator< int > output( cout, " " );

   std::vector< int > v1( a, a + SIZE );
   cout << "Vector v1 before replacing all 10s:\n   ";
   std::copy( v1.begin(), v1.end(), output );

   // replace 10s in v1 with 100
   std::replace( v1.begin(), v1.end(), 10, 100 );

   cout << "\nVector v1 after replacing 10s with 100s:\n   ";
   std::copy( v1.begin(), v1.end(), output );

   std::vector< int > v2( a, a + SIZE );
   std::vector< int > c1( SIZE );

   cout << "\n\nVector v2 before replacing all 10s "
        << "and copying:\n   ";
   std::copy( v2.begin(), v2.end(), output );

   // copy from v2 to c1, replacing 10s with 100s
   std::replace_copy(
      v2.begin(), v2.end(), c1.begin(), 10, 100 );

   cout << "\nVector c1 after replacing all 10s in v2:\n   ";
   std::copy( c1.begin(), c1.end(), output );

   std::vector< int > v3( a, a + SIZE );

   cout << "\n\nVector v3 before replacing values greater"
        << " than 9:\n   ";
   std::copy( v3.begin(), v3.end(), output );

```

```
     // replace values greater than 9 in v3 with 100
     std::replace_if( v3.begin(), v3.end(), greater9, 100 );

     cout << "\nVector v3 after replacing all values greater"
          << "\nthan 9 with 100s:\n   ";
     std::copy( v3.begin(), v3.end(), output );

     std::vector< int > v4( a, a + SIZE );
     std::vector< int > c2( SIZE );

     cout << "\n\nVector v4 before replacing all values greater "
          << "than 9 and copying:\n   ";
     std::copy( v4.begin(), v4.end(), output );

     // copy v4 to c2, replacing elements greater than 9 with 100
     std::replace_copy_if(
        v4.begin(), v4.end(), c2.begin(), greater9, 100 );

     cout << "\nVector c2 after replacing all values greater "
          << "than 9 in v4:\n   ";
     std::copy( c2.begin(), c2.end(), output );

     cout << endl;

     return 0;
} // end main
```

```

// determine whether argument is greater than 9
bool greater9( int x )
{
   return x > 9;

} // end function greater9
```

```
Vector v1 before replacing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing 10s with 100s:
   100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
   100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
   100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater than 9 in v4:
   100 2 100 4 100 6 100 8 100 100
```

MORE CONTAINERS

Container Adapters

- Container adapters
  - **`stack`**, **`queue`** and **`priority_queue`**
  - Not first class containers
    - Do not support iterators
    - Do not possess in-house data structure
  - Programmer can select implementation
  - Member functions **`push`** and **`pop`**

The stack Adapter

- **`stack`**
  - Header **`<stack>`**
  - Last-in, first-out (LIFO) data structure:
    - Insertions and deletions at one end
  - Can use **`vector`**, **`list`**, or **`deque`** (default)
  - Declarations

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // default: deque
```

    - **`vector`**, **`list`**
      - Implementation of **`stack`** (default **`deque`**)
      - Does not change behavior, just performance (**`deque`** and **`vector`** are fastest)

```cpp
 // Fig. 21.23: fig21_23.cpp
 // Standard library adapter stack test program.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <stack>   // stack adapter definition
 #include <vector>  // vector class-template definition
#include <list>    // list class-template definition

// popElements function-template prototype
template< class T >
void popElements( T &stackRef );

int main(void)
{
   // stack with default underlying deque
   std::stack< int > intDequeStack;

   // stack with underlying vector
   std::stack< int, std::vector< int > > intVectorStack;

   // stack with underlying list
   std::stack< int, std::list< int > > intListStack;

   // push the values 0-9 onto each stack
   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );

   } // end for

   // display and remove elements from each stack
   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );

   cout << endl;

   return 0;

} // end main

```

```
// pop elements from stack object to which stackRef refers
template< class T >
void popElements( T &stackRef )
{
   while ( !stackRef.empty() ) {
      cout << stackRef.top() << ' ';   // view top element
      stackRef.pop();                  // remove top element

   } // end while

} // end function popElements
```

```
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

The `queue` Adapter

- **`queue`**
  - Header **`<queue>`**
  - First-in-first-out (FIFO) data structure:
    - Insertions at back, deletions at front
  - Implemented with **`list`** or **`deque`** (default)
    - **`std::queue<double> values;`**
- Functions
  - **`push( element )`**
    - Same as **`push_back`**, add to end
  - **`pop( element )`**
    - Implemented with **`pop_front`**, remove from front
  - **`empty()`**
  - **`size()`**

```cpp
 // Fig. 21.24: fig21_24.cpp
 // Standard library adapter queue test program.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <queue>  // queue adapter definition

int main(void)
{
   std::queue< double > values;

   // push elements onto queue values
   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";

   while ( !values.empty() ) {
      cout << values.front() << ' ';  // view front element
      values.pop();                   // remove element

   } // end while

   cout << endl;

   return 0;

} // end main
```

```
Popping from values: 3.2 9.8 5.4
```

The `priority_queue` Adapter

- **`priority_queue`**
  - Header **`<queue>`**
  - Insertions occur in a sorted fashion; deletions from front
  - Implemented with **`vector`** (default) or **`deque`**
  - *Highest priority* element always removed first
    - Heapsort algorithm puts largest elements at front
    - **`less<T>`** default, programmer can specify other comparator
  - Functions
    - **`push(value), pop(value)`**
    - **`top()`**
      - View top element
    - **`size()`**
    - **`empty()`**

```cpp
 // Fig. 21.25: fig21_25.cpp
 // Standard library adapter priority_queue test program.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <queue>  // priority_queue adapter definition

int main(void)
{
   std::priority_queue< double > priorities;

   // push elements onto priorities
   priorities.push( 3.2 );
   priorities.push( 9.8 );
   priorities.push( 5.4 );

   cout << "Popping from priorities: ";

   while ( !priorities.empty() ) {
      cout << priorities.top() << ' ';  // view top element
      priorities.pop();                 // remove top element

   } // end while

   cout << endl;

   return 0;

} // end main
```

```
Popping from priorities: 9.8 5.4 3.2
```

MORE ALGORITHMS

21.5 Algorithms

- Before STL
  - Class libraries incompatible among vendors
  - Algorithms built into container classes
- STL separates containers and algorithms
  - Easier to add new algorithms
  - More efficient, avoids **`virtual`** function calls
  - **`<algorithm>`**

21.5.1 fill, fill_n, generate and generate_n

- Functions to modify containers
  - `fill(iterator1, iterator2, value);`
    - Sets range of elements to **`value`**
  - `fill_n(iterator1, n, value);`
    - Sets **`n`** elements to **`value`**, starting at **`iterator1`**
  - `generate(iterator1, iterator2, function);`
    - Like **`fill`**, but calls **`function`** to set each value
  - `generate(iterator1, quantity, function)`
    - Like **`fill_n, ""`**

```cpp
 // Fig. 21.26: fig21_26.cpp
 // Standard library algorithms fill, fill_n, generate
 // and generate_n.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <algorithm>  // algorithm definitions
#include <vector>     // vector class-template definition

char nextLetter();    // prototype

int main(void)
{
   std::vector< char > chars( 10 );
   std::ostream_iterator< char > output( cout, " " );

   // fill chars with 5s
   std::fill( chars.begin(), chars.end(), '5' );

   cout << "Vector chars after filling with 5s:\n";
   std::copy( chars.begin(), chars.end(), output );

```

Function **`fill`**.

```
   // fill first five elements of chars with As
   std::fill_n( chars.begin(), 5, 'A' );

   cout << "\n\nVector chars after filling five elements"
        << " with As:\n";
   std::copy( chars.begin(), chars.end(), output );

   // generate values for all elements of chars with nextLetter
   std::generate( chars.begin(), chars.end(), nextLetter );

   cout << "\n\nVector chars after generating letters A-J:\n";
   std::copy( chars.begin(), chars.end(), output );

   // generate values for first five elements of chars
   // with nextLetter
   std::generate_n( chars.begin(), 5, nextLetter );

   cout << "\n\nVector chars after generating K-O for the"
        << " first five elements:\n";
   std::copy( chars.begin(), chars.end(), output );

   cout << endl;

   return 0;

} // end main
```

Functions **generate** and **generate_n** both use functor **nextLetter**.

```cpp

   // returns next letter in the alphabet (starts with A)
   char nextLetter()
   {
      static char letter = 'A';
      return letter++;

   } // end function nextLetter
```

```
Vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with A’s:
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:
A B C D E F G H I J

Vector chars after generating K-O for the first five elements:
K L M N O F G H I J
```

21.5.2 `equal`, `mismatch` & `lexicographical_compare`

- Functions to compare sequences of values
  - **`equal`**
    - Returns **`true`** if sequences are equal (uses **`==`**)
    - Can return **`false`** if of unequal length

```
equal(iterator1, iterator2, iterator3);
```

    - Compares sequence from **`iterator1`** to **`iterator2`** with sequence beginning at **`iterator3`**
  - **`mismatch`**
    - Arguments same as **`equal`**
    - Returns a **`pair`** object with iterators pointing to mismatch
      - If no mismatch, **`pair`** iterators equal to last item

```
pair < iterator, iterator > myPairObject;
myPairObject = mismatch( iter1, iter2, iter3);
```

21.5.2 `equal`, `mismatch` & `lexicographical_compare`

- Functions to compare a sequence of characters
  - **`lexicographical_compare`**
    - Compare contents of two character arrays
    - Returns **`true`** if element in first sequence smaller than corresponding element in second

```
bool result = lexicographical_compare(iter1, iter2,
                                      iter3);
```

```
 // Fig. 21.27: fig21_27.cpp
 // Standard library functions equal,
 // mismatch and lexicographical_compare.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <algorithm>  // algorithm definitions
#include <vector>     // vector class-template definition

int main(void)
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };

   std::vector< int > v1( a1, a1 + SIZE );
   std::vector< int > v2( a1, a1 + SIZE );
   std::vector< int > v3( a2, a2 + SIZE );

   std::ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   std::copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   std::copy( v2.begin(), v2.end(), output );
   cout << "\nVector v3 contains: ";
   std::copy( v3.begin(), v3.end(), output );

   // compare vectors v1 and v2 for equality
   bool result =
      std::equal( v1.begin(), v1.end(), v2.begin() );

   cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v2.\n";

   // compare vectors v1 and v3 for equality
   result = std::equal( v1.begin(), v1.end(), v3.begin() );
   cout << "Vector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v3.\n";

   // location represents pair of vector iterators
   std::pair< std::vector< int >::iterator,
              std::vector< int >::iterator > locatio

   // check for mismatch between v1 and v3
   location = std::mismatch( v1.begin(), v1.end(), v3.begin() );

```

Use function **equal**.
Compares all of **v1** with **v2**.

Note use of function **mismatch**.

נושאים מתקדמים בתכנות מונחה עצמים, אביב 2022

```
   cout << "\nThere is a mismatch between v1 and v3 at "
        << "location " << ( location.first - v1.begin() )
        << "\nwhere v1 contains " << *location.first
        << " and v3 contains " << *location.second
        << "\n\n";

   char c1[ SIZE ] = "HELLO";
   char c2[ SIZE ] = "BYE BYE";

   // perform lexicographical comparison of c1 and c2
   result = std::lexicographical_compare(
      c1, c1 + SIZE, c2, c2 + SIZE );

   cout << c1
        << ( result ? " is less than " :
           " is greater than or equal to " )
        << c2 << endl;

   return 0;

} // end main
```

Use **lexicographical_compare**.

```
Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v3 contains 1000

HELLO is greater than or equal to BYE BYE
```

21.5.5 Mathematical Algorithms

- **`random_shuffle(iter1, iter2)`**
  - Randomly mixes elements in range
- **`count(iter1, iter2, value)`**
  - Returns number of instances of **`value`** in range
- **`count_if(iter1, iter2, function)`**
  - Counts number of instances that return **`true`**
- **`min_element(iter1, iter2)`**
  - Returns iterator to smallest element
- **`max_element(iter1, iter2)`**
  - Returns iterator to largest element

21.5.5 Mathematical Algorithms

- **`accumulate(iter1, iter2)`**
  - Returns sum of elements in range
- **`for_each(iter1, iter2, function)`**
  - Calls **`function`** on every element in range
  - Does not modify element
- **`transform(iter1, iter2, iter3, function)`**
  - Calls **`function`** for all elements in range of **`iter1-iter2`**, copies result to **`iter3`**

```
 // Fig. 21.30: fig21_30.cpp
 // Mathematical algorithms of the standard library.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <algorithm>  // algorithm definitions
 #include <numeric>    // accumulate is defined here
#include <vector>

bool greater9( int );
void outputSquare( int );
int calculateCube( int );

int main(void)
{
   const int SIZE = 10;
   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   std::vector< int > v( a1, a1 + SIZE );
   std::ostream_iterator< int > output( cout, " " );

   cout << "Vector v before random_shuffle: ";
   std::copy( v.begin(), v.end(), output );

   // shuffle elements of v
   std::random_shuffle( v.begin(), v.end() );

   cout << "\nVector v after random_shuffle: ";
   std::copy( v.begin(), v.end(), output );

   int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   std::vector< int > v2( a2, a2 + SIZE );

   cout << "\n\nVector v2 contains: ";
   std::copy( v2.begin(), v2.end(), output );

   // count number of elements in v2 with value 8
   int result = std::count( v2.begin(), v2.end(), 8 );

   std::cout << "\nNumber of elements matching 8: " << result;

   // count number of elements in v2 that are greater than 9
   result = std::count_if( v2.begin(), v2.end(), greater9 );

   cout << "\nNumber of elements greater than 9: " << result;

   // locate minimum element in v2
   cout << "\n\nMinimum element in Vector v2 is: "
        << *( std::min_element( v2.begin(), v2.end() ) );

   // locate maximum element in v2
   cout << "\nMaximum element in Vector v2 is: "
        << *( std::max_element( v2.begin(), v2.end() ) );

   // calculate sum of elements in v
   cout << "\n\nThe total of the elements in Vector v is: "
        << std::accumulate( v.begin(), v.end(), 0 );

   cout << "\n\nThe square of every integer in Vector v is:\n";

   // output square of every element in v
   std::for_each( v.begin(), v.end(), outputSquare );

   std::vector< int > cubes( SIZE );

   // calculate cube of each element in v;
   // place results in cubes
   std::transform(
      v.begin(), v.end(), cubes.begin(), calculateCube );
```

```cpp

   cout << "\n\nThe cube of every integer in Vector v is:\n";
   std::copy( cubes.begin(), cubes.end(), output );

   cout << endl;

   return 0;

} // end main

// determine whether argument is greater than 9
bool greater9( int value )
{
   return value > 9;

} // end function greater9

// output square of argument
void outputSquare( int value )
{
   cout << value * value << ' ';

} // end function outputSquare

```

```cpp
 // return cube of argument
 int calculateCube( int value )
 {
    return value * value * value;

} // end function calculateCube
```

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8
```

21.5.10 Set Operations

- **`includes(iter1, iter2, iter3, iter4)`**
  - Returns **`true`** if **`iter1-iter2`** contains **`iter3-iter4`**
  - Both ranges must be sorted

```
a1:  1 2 3 4
a2:  1 3
a1 includes a3
```

- **`set_difference(iter1, iter2, iter3, iter4,`**
  `iter5)`
  - Copies elements in first set (1-2) that are not in second set (3-4) into **`iter5`**
- **`set_intersection(iter1, iter2, iter3,`**
  `iter4, iter5)`
  - Copies common elements from the two sets (1-2, 3-4) into **`iter5`**

21.5.10 Set Operations

- **`set_symmetric_difference(iter1, iter2, iter3, iter4, iter5)`**
  - Copies elements in set (1-2) but not set (3-4), and vice versa, into **`iter5`**
    - **`a1: 1 2 3 4 5 6 7 8 9 10`**
    - **`a2: 4 5 6 7 8`**
    - **`set_symmetric_difference: 1 2 3 9 10`**
  - Both sets must be sorted

- **`set_union( iter1, iter2, iter3, iter4, iter5)`**
  - Copies elements in either or both sets to **`iter5`**
  - Both sets must be sorted

```
  // Fig. 21.35: fig21_35.cpp
  // Standard library algorithms includes, set_difference,
  // set_intersection, set_symmetric_difference and set_union.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>  // algorithm definitions

int main(void)
{
    const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
    int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
    int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
    std::ostream_iterator< int > output( cout, " " );

    cout << "a1 contains: ";
    std::copy( a1, a1 + SIZE1, output );
    cout << "\na2 contains: ";
    std::copy( a2, a2 + SIZE2, output );
    cout << "\na3 contains: ";
    std::copy( a3, a3 + SIZE2, output );

```

```
   // determine whether set a2 is completely contained in a1
   if ( std::includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
      cout << "\n\na1 includes a2";
   else
      cout << "\n\na1 does not include a2";

   // determine whether set a3 is completely contained in a1
   if ( std::includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
      cout << "\na1 includes a3";
   else
      cout << "\na1 does not include a3";

   int difference[ SIZE1 ];

   // determine elements of a1 not in a2
   int *ptr = std::set_difference( a1, a1 + SIZE1,
      a2, a2 + SIZE2, difference );

   cout << "\n\nset_difference of a1 and a2 is: ";
   std::copy( difference, ptr, output );

   int intersection[ SIZE1 ];

   // determine elements in both a1 and a2
   ptr = std::set_intersection( a1, a1 + SIZE1,
      a2, a2 + SIZE2, intersection );
   cout << "\n\nset_intersection of a1 and a2 is: ";
   std::copy( intersection, ptr, output );

   int symmetric_difference[ SIZE1 ];

   // determine elements of a1 that are not in a2 and
   // elements of a2 that are not in a1
   ptr = std::set_symmetric_difference( a1, a1 + SIZE1,
      a2, a2 + SIZE2, symmetric_difference );

   cout << "\n\nset_symmetric_difference of a1 and a2 is: ";
   std::copy( symmetric_difference, ptr, output );

   int unionSet[ SIZE3 ];

   // determine elements that are in either or both sets
   ptr = std::set_union( a1, a1 + SIZE1,
      a3, a3 + SIZE2, unionSet );

   cout << "\n\nset_union of a1 and a3 is: ";
   std::copy( unionSet, ptr, output );

   cout << endl;

   return 0;
} // end main
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15

a1 includes a2
a1 does not include a3

set_difference of a1 and a2 is: 1 2 3 9 10

set_intersection of a1 and a2 is: 4 5 6 7 8

set_symmetric_difference of a1 and a2 is: 1 2 3 9 10

set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

21.5.12 Heapsort

- Heapsort - sorting algorithm
  - A binary heap (= heap in the form of a binary tree)
  - Largest element at top of heap
  - Children always less than parent node
  - **`make_heap(iter1, iter2)`**
    - Creates a heap in the range of the iterators
    - Must be random access iterators (arrays, **`vector`**s, **`deque`**s)
  - **`sort_heap(iter1, iter2)`**
    - Sorts a heap sequence from **`iter1`** to **`iter2`**

21.5.12 Heapsort

- Functions
  - **`push_heap(iter1, iter2)`**
    - The iterators must specify a heap
    - Adds last element in object to heap
      - Assumes other elements already in heap order
  - **`pop_heap(iter1, iter2)`**
    - Removes the top element of a heap and puts it at the end of the container.
    - Function checks that all other elements still in a heap
    - Range of the iterators must be a heap.
    - If all the elements popped, sorted list

```cpp
// Fig. 21.37: fig21_37.cpp
// Standard library algorithms push_heap, pop_heap,
// make_heap and sort_heap.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>
#include <vector>

int main(void)
{
   const int SIZE = 10;
   int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   std::vector< int > v( a, a + SIZE ), v2;
   std::ostream_iterator< int > output( cout, " " );

   cout << "Vector v before make_heap:\n";
   std::copy( v.begin(), v.end(), output );

   // create heap from vector v
   std::make_heap( v.begin(), v.end() );

   cout << "\nVector v after make_heap:\n";
   std::copy( v.begin(), v.end(), output );
```

Create a new heap.

```

   // sort elements of v with sort_heap
   std::sort_heap( v.begin(), v.end() );

   cout << "\nVector v after sort_heap:\n";
   std::copy( v.begin(), v.end(), output );

   // perform the heapsort with push_heap and pop_heap
   cout << "\n\nArray a contains: ";
   std::copy( a, a + SIZE, output );

   cout << endl;

   // place elements of array a into v2 and
   // maintain elements of v2 in heap
   for ( int i = 0; i < SIZE; ++i ) {
      v2.push_back( a[ i ] );
      std::push_heap( v2.begin(), v2.end() );
      cout << "\nv2 after push_heap(a[" << i << "]): ";
      std::copy( v2.begin(), v2.end(), output );

   } // end for

   cout << endl;

```

Add elements one at a time.

```
   // remove elements from heap in sorted order
   for ( int j = 0; j < v2.size(); ++j ) {
      cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
      std::pop_heap( v2.begin(), v2.end() - j );
      std::copy( v2.begin(), v2.end(), output );

   } // end for

   cout << endl;

   return 0;

} // end main
```

```
Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22
```

```
v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100
```

21.5.13 `min` & `max`

- **`min(value1, value2)`**
  - Returns smaller element
- **`max(value1, value2)`**
  - Returns larger element

```
 // Fig. 21.38: fig21_38.cpp
 // Standard library algorithms min and max.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <algorithm>

int main(void)
{
   cout << "The minimum of 12 and 7 is: "
        << std::min( 12, 7 );
   cout << "\nThe maximum of 12 and 7 is: "
        << std::max( 12, 7 );
   cout << "\nThe minimum of 'G' and 'Z' is: "
        << std::min( 'G', 'Z' );
   cout << "\nThe maximum of 'G' and 'Z' is: "
        << std::max( 'G', 'Z' ) << endl;

   return 0;

} // end main
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

21.5.14 Algorithms Not Covered in This Chapter

- **`adjacent_difference`**
- **`inner_product`**
- **`partial_sum`**
- **`nth_element`**
- **`partition`**
- **`stable_partition`**
- **`next_permutation`**
- **`prev_permutation`**
- **`rotate`**
- **`rotate_copy`**
- **`adjacent_find`**
- **`partial_sort`**
- **`partial_sort_copy`**
- **`stable_sort`**

21.6 Class `bitset`

- Class **`bitset`**
  - Represents a set of bit flags
  - Can manipulate bit sets
  - Compare to the specialization of **`std::vector<bool>`**
- Operations

`bitset <size> b;`	create bitset
`b.set( bitNumber)`	set bit bitNumber to on
`b.set()`	all bits on
`b.reset(bitNumber)`	set bit bitNumber to off
`b.reset()`	all bits off
`b.flip(bitNumber)`	flip bit (on to off, off to on)
`b.flip()`	flip all bits
`b[bitNumber]`	returns reference to bit
`b.at(bitNumber)`	range checking, returns reference

21.6 Class `bitset`

- Operations
- **`b.test(bitNumber)`** has range checking; if bit on, returns **`true`**
- **`b.size()`** size of bitset
- **`b.count()`** number of bits set to on
- **`b.any()`** true if any bits are on
- **`b.none()`** true if no bits are on
- **`can use &=, |=, !=, <<=, >>=`**
  - **`b &= b1`**
  - Logical AND between **`b`** and **`b1`**, result in **`b`**
- **`b.to_string()`** convert to string
- **`b.to_ulong()`** convert to long

```cpp
 // Fig. 21.40: fig21_40.cpp
 // Using a bitset to demonstrate the Sieve of Eratosthenes.
 #include <iostream>

 using std::cin;
 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

#include <bitset>  // bitset class definition
#include <cmath>   // sqrt prototype

int main(void)
{
   const int size = 1024;
   int value;
   std::bitset< size > sieve;

   sieve.flip();

   // perform Sieve of Eratosthenes
   int finalBit = sqrt( sieve.size() ) + 1;

   for ( int i = 2; i < finalBit; ++i )

      if ( sieve.test( i ) )

         for ( int j = 2 * i; j < size; j += i )
            sieve.reset( j );

   cout << "The prime numbers in the range 2 to 1023 are:\n";

   // display prime numbers in range 2-1023
   for ( int k = 2, counter = 0; k < size; ++k )

      if ( sieve.test( k ) ) {
         cout << setw( 5 ) << k;

         if ( ++counter % 12 == 0 )
            cout << '\n';

      } // end outer if

   cout << endl;

```

Sieve of Eratosthenes: turn off bits for all multiples of a number. What bits remain are prime.

```
   // get value from user to determine whether value is prime
   cout << "\nEnter a value from 1 to 1023 (-1 to end): ";
   cin >> value;

   while ( value != -1 ) {

      if ( sieve[ value ] )
         cout << value << " is a prime number\n";
      else
         cout << value << " is not a prime number\n";

      cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
      cin >> value;

   } // end while

   return 0;

} // end main
```

```
The prime numbers in the range 2 to 1023 are:
    2    3    5    7   11   13   17   19   23   29   31   37
   41   43   47   53   59   61   67   71   73   79   83   89
   97  101  103  107  109  113  127  131  137  139  149  151
  157  163  167  173  179  181  191  193  197  199  211  223
  227  229  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349  353  359
  367  373  379  383  389  397  401  409  419  421  431  433
  439  443  449  457  461  463  467  479  487  491  499  503
  509  521  523  541  547  557  563  569  571  577  587  593
  599  601  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733  739  743
  751  757  761  769  773  787  797  809  811  821  823  827
  829  839  853  857  859  863  877  881  883  887  907  911
  919  929  937  941  947  953  967  971  977  983  991  997
 1009 1013 1019 1021

Enter a value from 1 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1
```