STL

STL

- The **Standard Template Library** (**STL**) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*.
- The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment).
- STL algorithms are independent of containers, which significantly reduces the complexity of the library.

Containers

- From: http://www.cplusplus.com/reference/stl/
  - A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
  - The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
  - Containers replicate data structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

Operations on Containers

- Random access
- Sequential access (normal, reverse)
- Search
- Insert (at the end, at the front, anywhere)
- Delete (at the end, at the front, anywhere)

Properties of Containers

- Sorted?
- Contiguous?
- Sequence or Associative?
- Unique elements?

Containers

- From: http://www.cplusplus.com/reference/stl/
  - Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.
- Chosing the most appropriate data structure for the problem at hand can save lot of implementation time and introduce massive performance gains

Containers Adaptors

- From: http://www.cplusplus.com/reference/stl/
  - stack, queue and priority_queue are implemented as *container adaptors*. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the *container adaptor* independently of the underlying *container* class used.
- In simple words, they expose some functionality to the user, but implement this functionality using internally a proper container object as storage vehicle

Sequences

- A sequence is an enumerated collection of objects in which repetitions are allowed
- An example of sequences are arrays

```
int a[] = {1, 2, 3, 2, 2, -1};
```

- Elements of the array are enumerated by the index of the array

STL Sequence Containers

- array – like a static array
- vector – like a dynamic array
- list – a double linked list
- forward_list – a single linked list
- deque – a double ended queue

std::array

- An Array of fixed size
- It wraps a static C array in a class and adds some extra functionality (e.g. iterators, size query, checks on index validity)
- Declaration:

```
template < class T, size_t N > class array;
```

  - **T** is the type of the array (e.g. *double*)
  - **N** is the size of the array
- N is a special type of template argument, instead of a type it is a value.
- **size_t** is a typedef: an unsigned integer type of at least 16 bit
- Requires **#include <array>**

Example: std::array

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
  array<int, 4> x = {1, 2, 2, 3};
  for (size_t i = 0; i < x.size(); ++i)
      cout << x[i] << endl;
  return 0;
}
```

std::array properties

- Fixed size defined at compile time
- Allows random access in constant time, i.e. O(1)
- Guaranteed to be memory contiguous (i.e. we can cast to a pointer and use pointer arithmetic)
- Does not do dynamic allocation
- Let’s look at the user guide:

  http://www.cplusplus.com/reference/array/array/

  In particular:
  - properties
  - [] vs at, front, back, size
  - swap, fill

std::array summary

- A drop-in replacement for a static C-array
- Adds convenient functionality at no extra costs (e.g. the method ***size***)
- Adds functions useful for debug (e.g. checked indices)

Problem

- Declare an array of 10 integers
- Populate it with random numbers, using the function to check index range (at) to access elements
- Assign the memory address of the first element to a const int pointer
- Print the element of the array using the above pointer

Solution

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
    array<int, 10> x;

  for (size_t i = 0; i < x.size(); ++i)
      x.at(i) = rand();

  // declare in the loop to pointers:
  // p associated with the memory address of the 1st element
  // e associated with the memory address of the 1st element after
   the end of the vector
  for (const int *p = &x.front(), *e = p + x.size(); p != e; ++p)
      cout << *p << endl;

  return 0;
}
```

Array Initialization

- Using initializer lists

```
array<int, 3> x = {1, 2, 3};
```

- Using default constructor, then manually (warning: there is no automatic initialization as in std::vector)

```
array<int, 3> x;
X[0] = 1;
...
```

Iterators

- An iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (**++**) and dereference (*) operators).
- The most obvious form of iterator is a pointer: A pointer can point to elements in an array, and can iterate through them using the increment operator (++).
- Other kinds of iterators are possible. For example, each container type (such as a list) has a specific iterator type designed to iterate through its elements.
- They are declared as nested classes in containers classes

Example – Iterators with std::array

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<int, 10> x;

   for (array<int,10>::iterator i = x.begin(); i != x.end(); ++i)
   *i = rand();

   for (array<int,10>::const_iterator i = x.cbegin(); i != x.cend(); ++i)
       cout << *i << endl;

   return 0;
}
```

i is a class ot type **`array<int,10>::iterator`**, which is a nested class of **`array<int,10>`**

- We use **i** as if it was a pointer (*i, ++i, i != end), but note it is not a pointer!
- It is an iterator class, with operators ++, !=, *, overloaded
- **`array<int,10>`** implements the methods **begin**, **end**, **cbegin**, **cend**
- **end** points to the first element after the last

Example – Iterators with std::array

- Iterators types can be long and tedious to write. This is one example where the **auto** keyword comes handy

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<int, 10> x;

   for (auto i = x.begin(); i != x.end(); ++i)
        *i = rand();

   for (auto i = x.cbegin(); i != x.cend(); ++i)
        cout << *i << endl;

   return 0;
}
```

Example – Iterators with std::array

- Because for array an iterator and a pointer are equivalent, we could convert an iterator to a pointer, if we wanted to.

First the ***** operator overloaded in the iterator class dereferences to the actual pointed object, then the **&** operator gets the memory address

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<int, 10> x = {1, 2, 3};

   for (const int* i = &*x.cbegin(), *e = i + x.size(); i != e; ++i)
        cout << *i << endl;

   return 0;
}
```

std::vector

- An array of variable size
- It wraps a C dynamic array in a class and adds some extra functionality (e.g. iterators, size query, checks on index validity, automatic memory management)
- The class is similar to what we implemented in *InputNumberArrayStruct1.cpp*
- Declaration:

```
template < class T, class Alloc = allocator<T> >
  class vector;
```

  - **T** is the type of the array (e.g. *double*)
  - **Alloc** is a class which performs memory allocation
  - **allocator** is the default memory allocator provided by the STL. Usually we do not change it.
- Requires **#include <vector>**

Example: std::vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  // 2nd argument omitted: use default allocator
  vector<int> x;

  for (size_t i = 0; i < 3; ++i)
      x.push_back( rand() );  // append at the end, size increases

  cout << x.size() << endl;   // print size

  for (size_t i = 0; i < x.size(); ++i)
      cout << x[i] << endl;
  return 0;
}
```

std::vector properties

- Dynamic size
- Allows random access in constant time, i.e. O(1)
- Insertion of new elements at the end of the vector are O(1), assuming capacity is sufficient.
- Capacity is automatically increased when necessary
- Erase at the end of the vector is O(1).
- Insertion and deletion in random position are O(n)
- Guaranteed to be memory contiguous (i.e. we can cast to a pointer and use pointer arithmetic)
- Requires dynamic allocation, handled automatically
- Let's look at the user guide:

  http://www.cplusplus.com/reference/vector/vector/

std::vector initialization

- Using initializer lists

```
vector<int> x = {1, 2, 3};
```

- Using default constructor, then manually

```
vector<int> x;
x.push_back(1);  // append to the end
```

- Using one of the many constructors, e.g.

```
vector<int> x(4, 10);  // 4 elements of
  value 10
```

Example - vector

- Declare a vector of 3 identical elements with value 5, using the class constructor
- Insert at the end 3 more elements with value 10
- Print all the elements using iterators

Solution

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> x(3, 5);  // x(size, value) is one of the constructors

  for (size_t i = 0; i < 3; ++i)
      x.push_back( 10 );

  cout << "size: " << x.size() << endl;

  for (auto i = x.cbegin(); i != x.cend(); ++i)
      cout << *i << endl;

  return 0;
}
```

std::vector methods

- Let’s look at the user guide:

  http://www.cplusplus.com/reference/vector/vector/

  - constructors, destructor
  - resize, size, capacity
  - push_back
  - assign
  - etc...

Example - vectors

- Declare a vector with 5 entries of type double
- Initialize the first 2 entries with some values using the [] operator
- Print the entries of the vector to screen

Solution

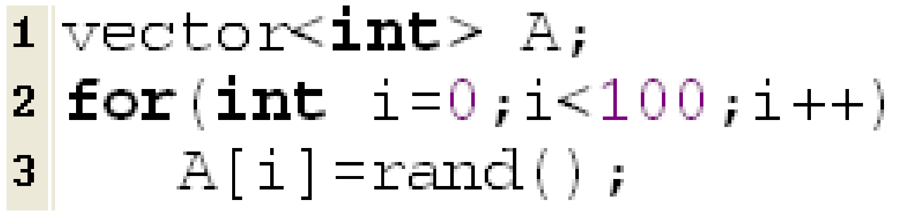
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  // x(size) is one of the constructors
  // all elements are initialized to zero
  vector<int> x(5);

  x[0] = rand();
  x[1] = rand();

  for (auto i = x.cbegin(); i != x.cend(); ++i)
      cout << *i << endl;

  return 0;
}
```

Problem

- Declare 2 vectors with 10 entries of type int
- Init them to random values
- Print them to the screen
- Swap them
- Print them to the screen

Vectors: Example

- Create a vector which contains 100 random numbers



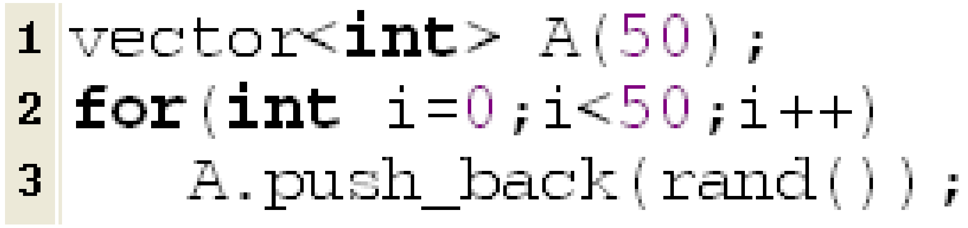
```
1 vector<int> A;
2 for(int i=0;i<100;i++)
3     A[i]=rand();
```

What is wrong?

After line 1, the vector still is empty. The entries A[i] used in line 3 don’t exist (forbidden indices)

Vectors: Example

Create a vector of length 50 whose entries are random numbers.



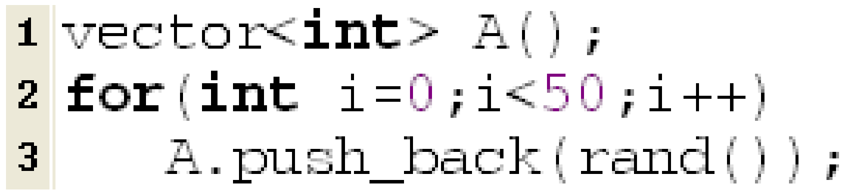
```
1 vector<int> A(50);
2 for(int i=0;i<50;i++)
3     A.push_back(rand());
```

What is wrong?

After line 1, the vector already contains 50 entries 0. The random number are appended to them. At the end, the vector will have a total of 100 entries.

Vectors: Example

Create a vector of length 50 whose entries are random numbers.



```
1 vector<int> A();
2 for(int i=0;i<50;i++)
3     A.push_back(rand());
```

What is wrong?

Line 1 does not create an empty vector. Possible would be `vector<int> A;` or `vector<int> A(0);`
Actually, line 1 is a function declaration!

Problem

After each of the following operations, print all entries of the vector to the screen:

- Create a vector of length 10 and entries of type `int`
- Append two new values to the end of the vector (*push_back*)
- Remove the last element of the vector (*pop_back*)
- Change the size of the vector to 20 (*resize*)
- Remove all entries of the vector (*clear*)

Problem

- Modify InputNumberArrayStruct1.cpp to use a std::vector

Other Iterators Operations

- Depending on the container on which they operate, some iterators may support other operators in addition to (++), (*).
- Examples:
  - (--): iterate backward
  - (-): decrement by given amount
  - (+): increment by given amount
  - (->): iterator member access

Problem

Construct vector with 10 numbers and print them to the screen in reverse order iterating on them in reverse order using the (--) operator

Problem

Construct vector with 10 numbers and print them to the screen in reverse order iterating on them in reverse order using reverse iterators

Problem

Read all lines of a file of arbitrary length as strings and store them in a vector

Problem

Implement a class which computes n factorial

The class has a cache, so at every request it stores in memory all factorials up to the highest number queried up to this point

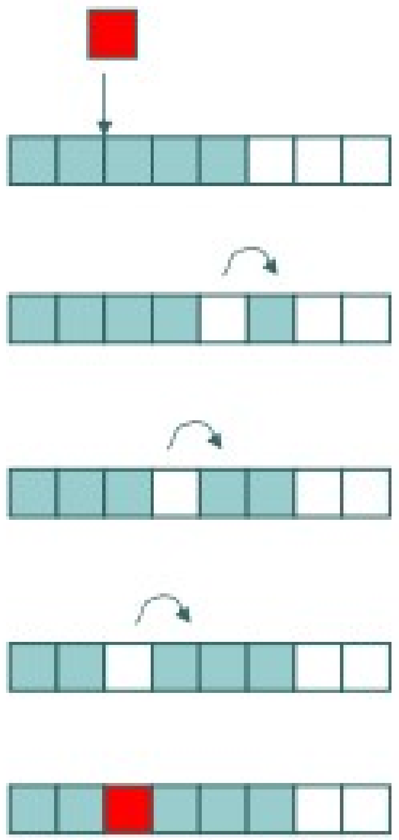
When a new request arrives for factorial of n, we check if it the value is already in the cache or not: if yes we return it, if not, we compute all new factorial up to n

Solution in: FactVec.cpp

Insertion in a Vector

- Inserting in a Vector has cost O(n)
- Suppose a vector contains {1, 4, 7, 3, 4} and we want to insert the element 5 at index 1. We need to:
  - Check if we have capacity for one extra elements and, if not, grow the vector (this all happens under the hood, inside the std::vector class)
  - If there is enough space, we need to move to the right by one position all elements from index 1 to index 4, to make space, i.e. {1, _, 4, 7, 3, 4} . We do this starting from the last one.
  - Then we can insert the new element: {1, 5, 4, 7, 3, 4}

Understanding Complexity



What happens when the vector is large?

std::list

- An std::list is double linked list of elements
- Each element keeps information on how to locate the next and the previous elements, so from the i-th element, we can access the (i-1)-th and (i+1)-th elements
- Declaration:

```
template < class T, class Alloc = allocator<T> >
  class list;
```

  - **T** is the type of the array (e.g. *double*)
  - **Alloc** is a class which performs memory allocation
  - **allocator** is the default memory allocator provided by the STL. Usually we do not change it.
- Requires **#include <list>**

std::list

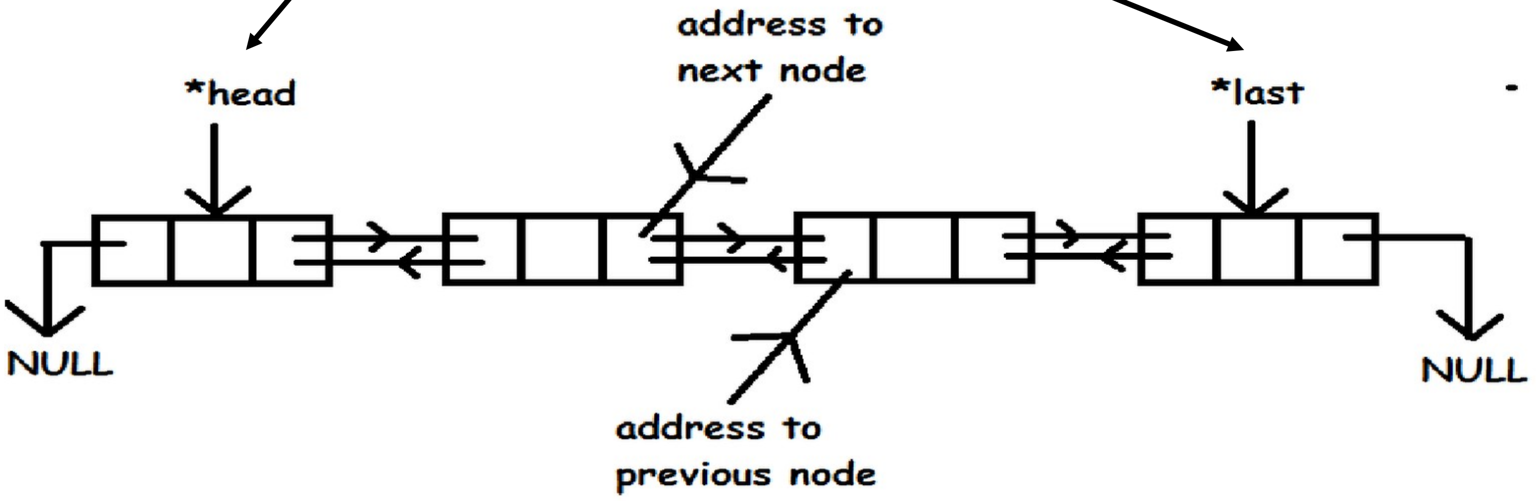
- Allow constant time, i.e. O(1), insert and erase operations anywhere within the sequence, and iteration in both directions.
- Entry points are the first element (head) and the last element (tail) of the list
- Random access to elements by index is not available: we need to traverse the list starting from head or tail up to element i-th
- Memory is not contiguous, so no pointer arithmetic available

std::list

The node in a list of types T is conceptually like this

```
template <class T>
struct ListNode {
   ListNode *prev, *next;
   T data;
};
```

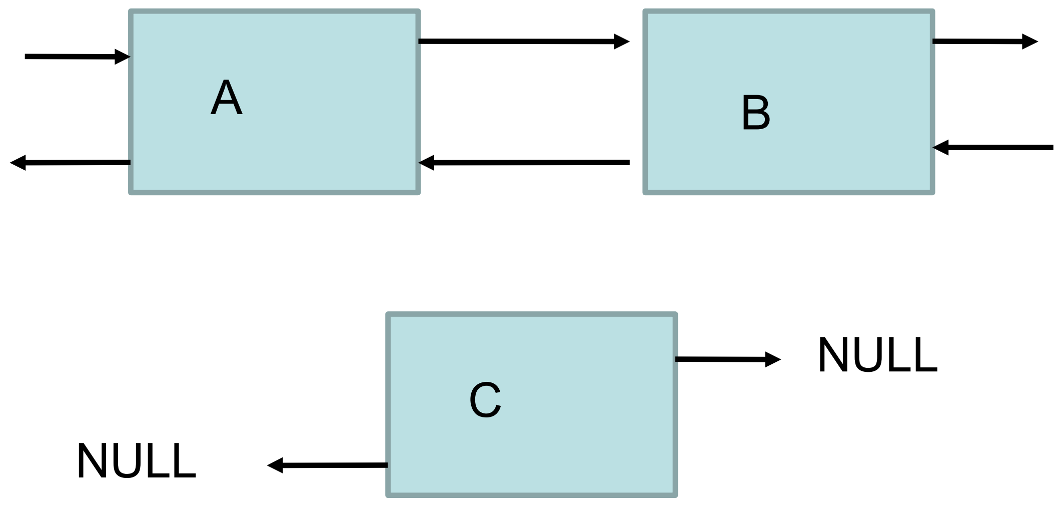
The list has pointers to the first and last element (head and tail)



Example – std::list

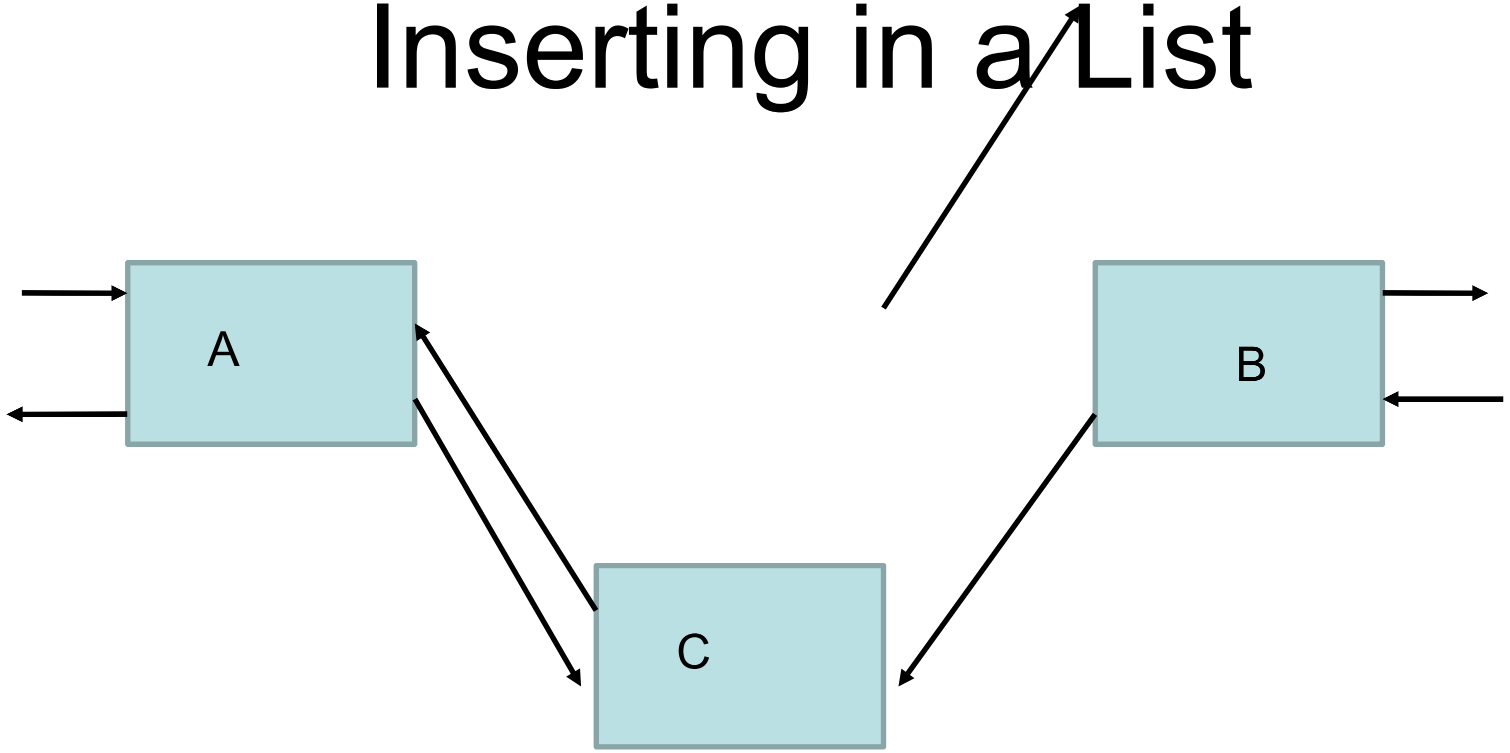
```
#include <iostream>
#include <list>

int main ()
{
    std::list<int> mylist(2,100); // two ints with a value of 100

    mylist.push_front(200);
    mylist.push_front(300);

    std::cout << "mylist contains:";
    for (auto it=mylist.cbegin(); it!=mylist.cend(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Inserting in a List



- We want to insert the new node C between A and B

Inserting in a List



- Insertion is easy and fast in a list, we just allocate a tiny bit of memory for the new element and update the pointers
- This is done internally by std::list::insert

std::list methods

- Let’s look at the user guide:
  http://www.cplusplus.com/reference/list/list/

- How does it compare with a vector? E,g, does a list have the *operator[]*? Does a vector have the method *push_front*?

Problem

- Create a list with 10 random elements in the range 0-5
- Print the list
- Print the list in reverse order
- Print the 4th element (inefficient, because access is sequential)
- Remove the first and the last element
- Print the list
- Find the first element greater than 2 and remove it
- Find the next element greater than 2 and insert the number 100 just before it
- Print the list
- Sort the list
- Print the list

std::list vs std::vector

- We gain fast insertion / removal
- We loose fast random access
- Storage is not contiguous
  - no pointer arithmetic
  - no need for large memory blocks (use many tiny memory blocks instead)
- Consumes extra memory (in addition to the data itself, we need to store also the pointers to the previous and to the next element)

std::forward_list

- A single linked list
- Very similar to std::list, except that links are uni-directional, so from element i-th we can only access element (i+1)-th
- As a result we can iterate only in one direction
- Insertions and deletion can happen only at the front
- http://www.cplusplus.com/reference/forward_list/forward_list/

std::deque

- Similar to a std::vector
- Not memory contiguous (i.e. no pointer arithmetic)
- Random access in O(1)
- Insertion and deletion in O(1) at front and back, in O(n) otherwise
- http://www.cplusplus.com/reference/deque/deque/

Choose a Sequence

- Depends on the use case
  - Is contiguous memory necessary (e.g. if we need to pass the data to a linear algebra routine)?
  - Random access needed?
  - Fixed or variable size?
  - Insertions/deletion at the end and at the front only, or everywhere?

Sequence Container Comparison

- See members map at

  http://www.cplusplus.com/reference/stl/

Problem

- We need to describe matrices of variable size.
- Use case: we set the matrix size at inception, when we create the matrix, then we access its elements by index in random position many times (for instance, for doing a matrix multiplication)
- The storage scheme needs to be memory contiguous, because we will pass the pointer to the storage to matrix operation function in the C-BLAS library
- We do not know the size of the matrix in advance
- What container shall we choose?

Problem

- We need to describe a deck of poker cards
- Initially we define the size of the deck (how many cards we have) and we shuffle the cards
- Then we deal the cards only from the top
- What container shall we choose?
- Hint: think about the shuffling operation, how do we do it?

Problem

- We need to store customer orders queue in a restaurant
- Orders are served in order of arrival (FIFO), i.e. new orders are added on one side of the queue and extracted for processing on the other side of the queue
- Sometimes a customer will ask how long it takes, i.e. about the position of his order in the queue
- What container shall we choose?

Predicates

- Let’s look at the declaration of the method **remove_if** of **std::list**

```
template <class Predicate>
void remove_if (Predicate pred);
```

- The template argument is a predicate, i.e. a function which returns true or false, depending if the element is to be removed from the list or not
- Predicate could be implemented via function pointers, but, in the STL, we use templates instead, which are more generic and faster

Function Objects

- Predicates can be implemented with **function objects**
- *Function objects* are objects specifically designed to be used with a syntax similar to that of functions, i.e. with round brackets. E.g.

```
mySquareRootFunction(3.5);
```

- In C++, this is achieved by defining member function **operator()** as part of the struct (or class) definition
- Function objects are typically passed as arguments to STL functions (like *remove_if*)

Function Objects - Example

```
struct myclass
{
  int operator()(int a) {return a*a;}
};

int main()
{
   myclass f;
   // we use f as if it was a function
   cout << f(3) << endl;
   return 0
};
```

Function Objects as Predicate - Example

```
#include <iostream>
#include <list>

// a predicate implemented as a class, i.e. as a function object:
struct is_odd {
  bool operator() (const int& value) { return (value%2)==1; }
};

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1

  // is_odd() is a constructor, i.e. it constructs a temporary object
  mylist.remove_if(is_odd());                // 36 20 4

  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Functions as STL Predicates

- Predicates can be implemented also as functions

```
// a predicate implemented as a function:
bool is_odd(const int& value) { return (value%2)==1; }

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);    // 15 36 7 17 20 39 4 1

  // here we pass the function pointer
  mylist.remove_if(&is_odd);                 // 36 20 4

  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

More Complex Predicates

- It would be nice if our predicates could take extra arguments
- For instance, if we want to remove from the list elements greater than 20, later on we want to remove also elements greater than 10, we can write two predicates:

```
bool greaterThan10(int v) { return v > 10; }
bool greaterThan20(int v) { return v > 20; }
```

- That is boring! Can we write the predicate only once and pass 10 or 20 as an extra argument?
- There are many ways to do it... let’s see some examples!

Extra Arguments as Data Members of Function Objects - Example

```
struct GraterThan {
  GreaterThan( int value ) : m_value(value) {}  // constructor
  bool operator() (const int& v) { return v > m_value; }
  int m_value;  // m_value is a data member of the function object
};

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1

  // GreaterThan(20) is a constructor, i.e. we construct a temporary object
  mylist.remove_if(GreaterThan(20));                 // 15 7 17 20 4 1
  mylist.remove_if(GreaterThan(10));                 // 7 4 1

  for (auto it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Extra Arguments as Non-Type Template Argument of Function Objects - Example

```
template <int V> // V is a non-type template parameter of the function object
struct GreaterThan {
  bool operator() (const int& v) { return v > V; }
};

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);    // 15 36 7 17 20 39 4 1

  // GreaterThan<20> is a constructor, i.e. we construct a temporary object
  mylist.remove_if(GreaterThan<20>{});                    // 15 7 17 20 4 1
  mylist.remove_if(GreaterThan<10>{});                    // 7 4 1

  for (auto it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

This only work if the type of the argument is a valid for a non-type template argument (e.g. int is ok, double is not ok)

Extra Arguments as Non-Type Template Argument of Functions - Example

```
template <int V> // V is a non-type template parameter of the function
bool GreaterThan (const int& v) { return v > V; }

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1

  // GreaterThan<20> is a constructor, i.e. we construct a temporary object
  mylist.remove_if(&GreaterThan<20>);                  // 15 7 17 20 4 1
  mylist.remove_if(&GreaterThan<10>);                  // 7 4 1

  for (auto it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

This only work if the type of the argument is a valid for a non-type template parameter (e.g. int is ok, double is not ok)

std::bind

- **std::bind** is an advanced function object which allows to bind arguments of a function to specific values
- It requires

```
#include <functional>
using namespace std::placeholders; // to use _1, _2, _3, ...
```

- Example:

```
int foo(int a, int b, int c, int d) { return a + b * c - d; }
using namespace std::placeholders;
// create a function of two arguments from foo binding a=5 and c=3
auto g = std::bind(&foo, 5, _1, 3, _2);
int x = g(10,20);  // equivalent to x = foo(5, 10, 3, 20);
```

std::bind - Example

```
#include <iostream>
#include <list>
#include <functional>
using namespace std::placeholders;    // adds visibility of _1, _2, _3,...

// GreaterThan is a function of 2 arguments.
bool GreaterThan (const int& v1, const int& v2) { return v1 > v2; }

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1

  // Here std::bind creates a function object based on a function
  // of 2 arguments and binds the 2nd argument to a specific value
  // The final result is a function object of just one variable
  mylist.remove_if(std::bind(&GreaterThan, _1, 10));     // 7 4 1

  for (auto it=mylist.begin(); it!=mylist.end(); ++it) std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Problem

- Given the function of 4 arguments

```
double f( double a, double b, double c, double x )
{
   return (a * x + b) * x + c;  // axx+bx+c
}
```

- use std::bind to create a function *g* which binds a=1, b=-2, c=1
- use it to compute g(2.5)

Solution

```
#include <iostream>
#include <functional>
using namespace std::placeholders;    // adds visibility of _1, _2, _3,...

double f(double a, double b, double c, double x) { return (a*x+b)*x+c; }

int main ()
{
  // Here std::bind creates a function object based on a function
  // of 4 arguments and binds the 1st, 2nd and 3rd argument to specific values
  // The final result g is a function object of just one variable
  // We use auto for the type of g as it is very complicated
  auto g = std::bind(&f, 1.0, -2.0, 1.0, _1);

  std::cout << g(2.5) << '\n';  // 2.25

  return 0;
}
```

Functional

- In the previous examples we implemented the function object GreaterThan
- This seems like a function object of very common use
- Is it already available from some library?
- The STL implements a number of function objects ready to use for us
- Like std::bind, it requires **#include <functional>**
- See

  http://www.cplusplus.com/reference/functional/

Functional - Example

```
#include <iostream>
#include <functional>

int main()
{
   std::greater<double> gt;
   // we use gt as if it was a function
   std::cout << gt(4.2, 5.0) << std::endl;  // false
   std::cout << gt(5.0, 4.2) << std::endl;  // true
   return 0;
}
```

Binding Arguments in a Function Object

- std::bind works also with function objects, not only with functions
- For function objects with just two arguments we can also use **bind1st** and **bind2nd**

bind1st, bind2nd with function objects - Example

```
#include <iostream>
#include <functional>

int main()
{
    using namespace std::placeholders;
   auto gt = std::greater<int>{};
   auto gta = std::bind1st(gt, 0);  // 0 > x ?
   auto gtb = std::bind2nd(gt, 0);  // x > 0 ?
   // we use gta and gtb as if they were functions
   std::cout << gta(2) << std::endl;  // false
   std::cout << gta(-2) << std::endl; // true
   std::cout << gtb(2) << std::endl;  // true
   std::cout << gtb(-2) << std::endl;  // false
   return 0;
}
```

bind with function objects- Example

```
#include <iostream>
#include <functional>

int main()
{
    using namespace std::placeholders;
   auto gt = std::greater<int>{};
   auto gta = std::bind(gt, 0, _1);  // 0 > x ?
   auto gtb = std::bind(gt, _1, 0);  // x > 0 ?
   // we use gta and gtb as if they were functions
   std::cout << gta(2) << std::endl;  // false
   std::cout << gta(-2) << std::endl; // true
   std::cout << gtb(2) << std::endl;  // true
   std::cout << gtb(-2) << std::endl;  // false
   return 0;
}
```

Stack

- A stack is a container which implements a LIFO policy.
- An example is the tower of the Hanoi puzzle, where disks can be inserted or removed only from the top
- We could use a **vector** or a **list**.
- Adding an element to the top of the stack is called a **push** operation, while removing it is a **pop** operation
- For example, in a vector the *push* operation could correspond to a *push_back* and the *pop* operation to a *pop_back*.
- Both have more functionality than strictly needed. For example, both allow insertion.

std::stack

- The STL implements a **std::stack** container, which has strictly the functionality needed.
- Requires **#include <stack>**
- This is implemented merely as an abstraction on top of a list or a vector, which is why this is called an adaptor container.
- I.e., the container has a data member of type vector (or list), but exposes to the user only a subset of its functionality
- http://www.cplusplus.com/reference/stack/stack/

std::stack

- Declaration:

```
template <class T, class Container = deque<T> >
class stack;
```

- T is the type contained
- The second argument, Container, allows the user to choose which actual container class should be used internally, and, if not specified, it proposes to use a deque. The user guide states that we can use any container which implements the methods: *empty, size, back, push_back, pop_back*

std::stack declarations- Examples

```
// a stack implemented internally as a deque
stack<int> s1;

// a stack implemented internally as a vector
stack<int, vector<int>> s2;

// a stack implemented internally as a list
stack<int, list<int>> s3;

// compiler error: forward_list does not implement
// the method push_back
stack<int, forward_list<int>> s4;
```

std::stack declarations- Examples

```
// a stack implemented internally as a deque
stack<int> s1;

// a stack implemented internally as a vector
stack<int, vector<int>> s2;

// a stack implemented internally as a list
stack<int, list<int>> s3;

// compiler error: forward_list does not implement
// the method push_back
stack<int, forward_list<int>> s4;
```

std::stack example

```
#include <stack>
#include <iostream>
using namespace std;

int main()
{
   stack<int> s;
   s.push(2);
   s.push(3);
   cout << "The stack contains " << s.size() << " entries\n";
   cout << "The top entry is " << s.top() << "\n";
   s.pop();
   cout << "The top entry is " << s.top() << "\n";
   s.pop();
   cout << "The empty function returns " << s.empty() << "\n";
   return 0;
}
```

Problem

- Modify the source code for the hanoi tower example to use a std::stack

std::queue

- **std::queue** implements a FIFO policy.
- It is implemented as an adaptor container.
- Declaration:

```
template <class T, class Container = deque<T> > class queue;
```

- http://www.cplusplus.com/reference/queue/queue/
- Requires the underlying container to implement the methods: *empty, size, front, back, push_back, pop_front*

Associative Containers

- In an associative container elements are referenced by key, rather than by the index of their position in the container
- Two commonly used STL associative containers are **set** and **map**

std::set

- A set is a collection of keys
- The entries are kept sorted according with some strict ordering criteria
- Entries are unique

std::set

- **std::set** is a collection of keys
- The entries are kept sorted according with some strict ordering criteria
- Entries are unique
- Declaration:

```
template <
   class T,                        // set::key_type/value_type
   class Compare = less<T>,        // set::key_compare/value_compare
   class Alloc = allocator<T>      // set::allocator_type
  > class set;
```

- http://www.cplusplus.com/reference/set/set/
- The user can pass a custom ordering criteria. The default is the function object **less<T>**

std::set - example

```
#include <set>
#include <iostream>
using namespace std;

int main()
{
    set<int> s;     // a set of integers
    s.insert(10);   // insert 10 in the set
    s.insert(4);    // insert 4 in the set
    s.insert(9);    // insert 9 in the set
    s.insert(4);    // does nothing, because 4 is already there
    // print elements sequentially (note they are ordered)
    for (auto i = s.begin(); i != s.end(); ++i)
        cout << *i << endl;  // we use the iterator i
    return 0;
}
```

std::set with custom order - example

```
#include <set>
#include <iostream>
#include <functional>
using namespace std;

int main()
{
    set<int, greater<int>> s;  // a set of integers in reverse order
    s.insert(10);  // insert 10 in the set
    s.insert(4);   // insert 4 in the set
    s.insert(9);   // insert 9 in the set
    s.insert(4);   // does nothing, because 4 is already there
    // print elements sequentially (note the reverse ordered)
    for (auto i = s.begin(); i != s.end(); ++i)
        cout << *i << endl;  // we use the iterator i
    return 0;
}
```

std::set with user class - example

```
#include <set>
#include <iostream>
using namespace std;

// Note there is no constructor: how does YearMonth{12,3} works?
struct YearMonth {
   int m_y, m_m;
};

bool operator<(const YearMonth& a, const YearMonth& b)
{ return a.m_y < b.m_y || a.m_y == b.m_y && a.m_m < b.m_m; }

ostream& operator<<(ostream& os, const YearMonth& a)
{ os << a.m_y << "-" << a.m_m; return os; }

int main() {
   set<YearMonth> s;    // relies on custom overload of operator<
   s.insert(YearMonth{12,3});  // insert March-2012
   s.insert(YearMonth{11,4});  // insert April-2011
   for (auto i = s.begin(); i != s.end(); ++i)
       cout << *i << endl; // relies on custom overload of operator<<
   return 0;
}
```

std::set finding entries

- In addition to sequential access, we can test if an element is contained in a set using the *find* method
- Since a set is usually implemented like a binary tree, the cost for searching an element in the tree is O(log n)

std::set find - example

```
// set::find
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; i++)
      myset.insert(i*10);    // set: 10 20 30 40 50

  it = myset.find(10);
  if (it != myset.end())
      cout << "number found\n";
  else
      cout << "number not found\n";

  return 0;
}
```

std::set many other methods

- std::set implements many other methods.
- See http://www.cplusplus.com/reference/set/set/

std::map

- A map is a collection of key-value pairs
- The entries are kept sorted according with some strict ordering criteria based on the key
- Entries are unique (i.e. keys are unique)

std::map

- **std::map** is a collection of (key,value) pairs
- The entries are kept sorted according with some strict ordering criteria
- Entries are unique by key
- Declaration:

```
template <
   class K,                         // set::key_type
   class T,                         // map::mapped_type
   class Compare = less<K>,         // set::key_compare/value_compare
   class Alloc = allocator<T>       // set::allocator_type
  > class map;
```

- http://www.cplusplus.com/reference/set/set/
- The user can pass a custom ordering criteria. The default is the function object **less<K>**

std::map - example

```
#include <map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    map<string, int> m;    // a map of (string,int)
    m["Jim"] = 1967;
    m["Bob"] = 1985;
    m["Tom"] = 2014;
    for (auto i = m.begin(); i != m.end(); ++i)
        cout << i->first << " was born on " << i->second << endl;
    return 0;
}
```

std::map - example

- Note that an iterator on a map points to a (key,value) pair
- In fact, when we print, we use *first* and *second* to access key and value

std::map - example

- Note that to insert elements we used the syntax:

```
m["Jim"] = 1967;
```

- In a map the operator[] is overloaded as follows: it takes the ket as argument, if the key does not exist in the map, add it to it and initialize the value using the default constructor, then, in any case, return a reference to the element
- So the instructions above correspond to the following steps:
  - Check if the key “Jim” in the map
  - Since Jim is not there, add it
  - Assign 1967 to the correspondent value (if the value was already in the map, it would be overridden)
- operator[] should not be used if we just want to test if an element is in the map (because if it is not there it will add it!)

std::map – double search

- Sometime we want to test if an element is in a map and, if it is not there we add it, if it is already there we do nothing.
- Note that this behaviour is different from that of the operator[]
- A common mistake is to implement this as:

```
if (m.find(key)==m.end())  // if not found
   m[key]=somevalue;
```

- This is not a bug, but it is very inefficient because we search for the key twice!

std::map double search - solution

```
#include <map>
#include <iostream>
#include <string>
using namespace std;
void verboseInsert(map<string, int>& m, const string& name, int year)
{
   pair<map<string, int>::iterator, bool> ins = m.insert(pair<string,int>(name,
   year));
   cout << (ins.second ? "new: " : "old: ")
        << ins.first->first << " " << ins.first->second << endl;
}
int main()
{
   map<string, int> m;
   m["Jim"] = 1967; m["Bob"] = 1985; m["Tom"] = 2014;
   verboseInsert(m, "Jim", 1960);  // Jim already exist and is not modified
   verboseInsert(m, "Tim", 1970);  // Tim does not exits and is inserted
   for (auto i : m) // using range syntax in the for loop
      cout << i.first << " was born on " << i.second << endl;
   return 0;
}
```

Std::map – ordering criteria

- Custom ordering criteria can be defined exactly as for a set
- User classes can be used as keys, like we did for *set*

std::map's other methods

- std::map implements many other methods.
- See http://www.cplusplus.com/reference/map/map/

Algorithms

- The algorithms library contains general purpose algorithms which work on iterators
- Because all containers expose iterators, they work on all containers
- They also work on raw pointers, as raw pointers are conceptually simple iterators
- Require **#include <algorithm>**

Algorithm Example: find

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

- Returns the first element in the range [first, last) which is equal to value
- If the sought value is not found it returns *last*
- It searches linearly, i.e. it checks the first element, then the second, etc. until it find the sought value
- Complexity is O(n)
- It only requires that iterators can traverse the container sequentially, which is a basic iterator feature, therefore it works on any container

find – possible implementation

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value)
{
  for (; first != last; ++first)
     if (*first == value)
        return first;
  return last;
}
```

- This possible implementation uses the following iterator operators: (!=), (++), (*)
- and (==) on the value type T

find – Example with vector

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

int main()
{
    int n = 3;

    std::vector<int> v  = {0, 1, 2, 3, 4};

    std::vector<int>::iterator result = std::find(v.begin(), v.end(), n);

    if (result != v.end()) {
        std::cout << "v contains: " << n << '\n';
    } else {
        std::cout << "v does not contains: " << n << '\n';
    }
}
```

find – Example with pointers

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

int main()
{
    int n = 3;

    int v[] = {0, 1, 2, 3, 4};
    const int nElems = sizeof(v)/sizeof(*v);

    int *result = std::find(v, v+nElems, n);

    if (result != v+nElems) {
        std::cout << "v contains: " << n << '\n';
    } else {
        std::cout << "v does not contains: " << n << '\n';
    }
}
```

find – Example with set

```
#include <iostream>
#include <algorithm>
#include <set>

int main()
{
    int n = 3;

    int sv[] = {0, 1, 2, 3, 4};
    std::set<int> v(sv, sv+sizeof(sv)/sizeof(*sv)); // note the constructor

    std::set<int>::iterator result = std::find(v.begin(), v.end(), n);

    if (result != v.end()) {
        std::cout << "v contains: " << n << '\n';
    } else {
        std::cout << "v does not contains: " << n << '\n';
    }
}
```

Algorithm Example: find_if

```
template< class InputIt, class UnaryPredicate >
InputIt find_if( InputIt first, InputIt last,
                 UnaryPredicate p );
```

- Returns the first element in the range [first, last) such that the unary predicate p is true
- It searches linearly, i.e. it checks the first element, then the second, etc. until it find the sought value
- If the sought value is not found it returns *last*
- It only requires that iterators can traverse the container sequentially, which is a basic iterator feature, therefore it works on any container

find_if – possible implementation

```
template<class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last,
  UnaryPredicate p)
{
    for (; first != last; ++first)
        if (p(*first))
            return first;
    return last;
}
```

- This possible implementation uses the following iterator operators: (!=), (++), (*)

find_if – Example with pointers

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    int v[] = {0, 1, 2, 3, 4};
    const int n = sizeof(v)/sizeof(*v);

    // find first element such that v[i]>2
    // we use functional to describe the predicate
    int *result = find_if(v, v+n, bind2nd(greater<int>{}, 2) );

    if (result != v+n) {
        cout << "value greater than 2 found: " << *result << '\n';
    } else {
        cout << "value greater than 2 not found" << '\n';
    }
}
```

Algorithm Example: transform

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1,
   OutputIt d_first, UnaryOperation unary_op );
```

- **transform** applies the given function to a range and stores the result in another range
- If the sought value is not found it returns *last*
- It only requires that iterators can traverse the container sequentially, and that iterators are modifiable (i.e. non const operators)

transform – possible implementation

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
   OutputIt d_first, UnaryOperation unary_op)
{
    while (first1 != last1)
        *d_first++ = unary_op(*first1++);
    return d_first;
}
```

- This possible implementation uses the following iterator operators: !=, ++, *

transform – Example with vector

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int square(int x) {return x*x;}

int main()
{
    int sv[5] = {0, 1, 2, 3, 4};
    vector<int> v1(sv, sv+5);
    vector<int> v2(5);  // allocate space

    // we transform all elements and write the result to v2
    // note we could have used as destination v1 itself
    transform(v1.begin(), v1.end(), v2.begin(), square );

    for (auto i : v2)
        cout << i << ' ';
    cout << endl;
}
```

transform – example with 2 vectors

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1 = {0, 1, 2, 3, 4};
    vector<int> v2(v1.size());  // allocate space

    // we transform all elements and write result to v2
    // performs the operation v2[i] = v1[i] + v1[n-i]
    transform(v1.begin(), v1.end(), v1.rbegin(), v2.begin(), plus<int>() );

    for (vector<int>::iterator i = v2.begin(); i != v2.end(); ++i)
        cout << *i << ' ';
    cout << endl;
}
```

<algorithm>

- Let’s look at what’s in it:
- http://en.cppreference.com/w/cpp/algorithm

predicates: lambda expressions

```
[ captures ] ( params ) -> ret { body }
```

- captures variables from the context
- params are arguments
- ret is the return type (optional)

```
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    auto f = [x](int y) -> int { return x + y * 2; };  // return type omitted

    cout << f(5) << endl;

    return 0;
}
```

transform – example with lambda

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1 = {0, 1, 2, 3, 4};
    vector<int> v2(v1.size());  // allocate space

    // we transform all elements and copy to v2
    // we could have used as destination v1
    transform(v1.begin(), v1.end(), v2.begin(), [](int x){return x*x;} );

    for (auto i : v2)
        cout << i << ' ';
    cout << endl;
}
```

transform – example with lambda (2 vectors)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1 = {0, 1, 2, 3, 4};
    vector<int> v2(v1.size());  // allocate space

    // we transform all elements and write result to v2
    // performs the operation v2[i] = v1[i] + v1[n-i]
    transform(v1.begin(), v1.end(), v1.rbegin(), v2.begin(),
              [](int x, int y) { return x + y; } );

    for (vector<int>::iterator i = v2.begin(); i != v2.end(); ++i)
        cout << *i << ' ';
    cout << endl;
}
```

Problem

- Create a vector of 20 random int numbers in the range [1, 15] and print it
- Sort the vector and print it
- Remove duplicates an print it

(Source code: Algo1.cpp)

Problem

- Create a vector of 20 random int numbers in the range [1, 15] and print it
- Copy them to a set and print it
- Copy back to the vector and print it
- Modify the print routine from Algo1.cpp to work either with a vector or a set

(Source code: Algo2.cpp)

Problem

- You re-evaluated your portfolio in 1000 possible different scenarios (create a vector with random double numbers in -100, +100)
- Find the value at risk at the portfolio with 95% confidence
- Hint: you could sort the vector, but there may be a more appropriate function

Problem

- Two payoffs requires price observations respectively on the dates ($t_{1,0}$, $t_{1,1}$, ..., $t_{1,n}$) and ($t_{2,0}$, $t_{2,1}$, ..., $t_{2,m}$). To evaluate both products in the same Monte Carlo simulation, you need to merge the two set of dates
- Generate two vectors of strictly increasing times, both starting from 0, and containing an arbitrary number of strictly increasing times.
- Merge them
- Hint: to generate the vector you can build the vectors incrementally, adding repeatedly a random positive time step
- Source code: *MergeDates.cpp*

Problem

- Given a std::vector of pair<double,double>, which describes the marks of a volatility smile, implement the following boolean functions which check that the set of marks is non arbitragebile:
  - The strikes must be in increasing order (use adjacent_find)
  - Compute a vector of call option prices (use transform). Hint: implement the cumulative normal based on std::erf (cmath)
  - Verify that the call option prices are strictly decreasing (use adjacent_find)
  - Compute the vector of slopes between every pair of points using std::transform
  - Check that the first slope is >-1 (use *front*), the last slop is >0 (use *back*) and the sequence of slopes is strictly increasing (use adjacent_find)