Brief Tutorial on `std::vector`

The material in this handout is collected from the following references:

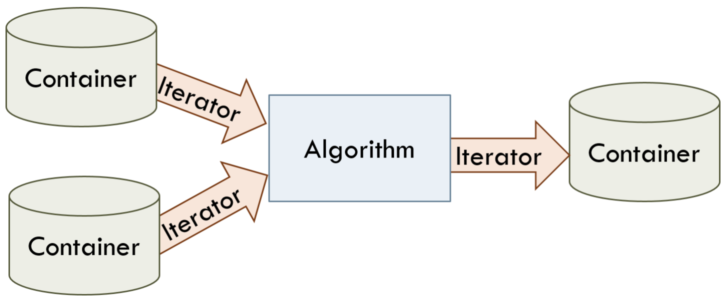
- Sections 3.3 and 3.4 of the text book C++ Primer.
- Section 7.3 of The C++ Standard Library: A Tutorial and Reference.

Overview of Standard Template Library

The heart of the C++ standard library is the *standard template library* (STL). The STL is a generic library that provides solutions to managing collections of data with modern and efficient algorithms. It allows C++ programmers to benefit from innovations in the area of data structures and algorithms without needing to learn how they work. The STL is based on the cooperation of various well-structured components, key of which are containers, iterators, and algorithms:

- ***Containers*** are used to manage collections of objects of a certain kind. Every kind of container has its own advantages and disadvantages, so having different container types reflect different requirements for collections in programs.
- ***Iterators*** are used to step through the elements of collections of objects. These collections may be containers or subsets of containers. The major advantage of iterators is that they offer a small but common interface for any arbitrary container type. For example, one operation of this interface lets the iterator step to the next element in the collection. This is done independently of the internal structure of the collection. Regardless of whether the collection is an array, a linked list, or a tree, it works. The interface for iterators is almost the same as for ordinary pointers. To increment an iterator, you call operator `++`. To access the value of an iterator, you use operator `*`. So, you might consider an iterator a kind of a smart pointer that translates the call "go to the next element" into whatever is appropriate.
- ***Algorithms*** are used to process the elements of collections. For example, algorithms can search, sort, modify, or simply use the elements for various purposes. Algorithms use iterators. Thus, because the iterator interface for iterators is common for all container types, an algorithm has to be written only once to work with arbitrary containers.

The concept of the STL is based on a *separation* of data and operations. The data is managed by container classes, and the separation are defined by configurable algorithms. Iterators are the glue between these two components. They let any algorithm interact with any container.



In a way, the STL concept contradicts the original idea of object-oriented programming: the STL *separates* data and algorithms rather than combining them. However, the reason for doing so is very important. In principle, you can combine every kind of container with every kind of algorithm, so the result is a very flexible but still rather small framework.

Background

Static arrays are used when programs require fixed-sized sequence of elements of a given type where the number of elements is specified at compile time. C/C++ provide the subscript operator `[]` to randomly access any element of the array. Such arrays are provided contiguous storage on the stack or in static storage. When used in an expression, the name of a static array evaluates to a pointer to the first element of the array. Thus, the use of the subscript operator in expressions is syntactic sugarcoating for pointer arithmetic involving the array name and the index of the element being accessed. Static arrays have three disadvantages:

- Indices are not checked before accessing an array. Such out-of-bound reads can generate incorrect program results while out-of-bound writes can cause undefined program behavior.
- The array size must be known at compile time. For instance, static arrays will not work when a sequence of unknown number of integers have to be read from a file.
- When the static array is passed to a function, the array size is lost because of the array-name-to-pointer-to-first-element conversion. This means that we need to pass functions not only the array (more correctly, a pointer) but also the number of elements that must be handled by the function.

The first and third problems are solved in C++ by the introduction of `array` type. This type has a member function `array::at` that performs runtime bounds checking on indices albeit at the cost of diminished program performance. Here is a simple program that illustrates the `array` type:

```cpp
#include <array>     // std::array
#include <string>    // std::string
#include <iostream>  // std::cout
#include <stdexcept> // std::out_of_range

template <std::size_t N>
int sum(std::array<int, N> const& a) {
  int total {0};

  // intentionally access out-of-bounds element
  for (std::size_t i{0}; i < a.size()+1; ++i) {
    // member function array::at automatically checks whether index i
    // is out-of-bounds; if so, an std::out_of_range exception is thrown.
    total += a.at(i);
  }
  return total;
}

int main() {
  try {
    // this is an alternative to static arrays in C++11
    std::array<int, 5> ai5 {11, 21, 31, 41, 51};
    // when passing array container to function, we don't need to
    // specify the array size
    int total = sum(ai5);
    std::cout << "total: " << total << "\n";
  } catch (std::out_of_range& e) {
    std::cerr << e.what() << " Oops!!! Range error\n";
  }
}
```

The second problem is solved by dynamically allocating contiguous memory on the free store using operator `new []`. Dynamically allocated arrays have the same danger as static arrays: accessing out of range memory can corrupt data (belonging to other objects) and subsequent program crashes. The other danger of dynamically allocated memory is memory leaks. Programmers must remember to return previously allocated memory back to the free store using operator `delete []`. Programmers may inadvertently forget to release the memory back to the free store. Or, even if the programmer has deleted the memory, program execution may not reach the statement `delete`ing the memory because of logic errors. In other cases, the memory may have been returned to the free store but the programmer may still use the pointer to access the previously allocated but now freed memory.

The standard C++ library provides the `std::vector` type as an alternative to dynamically allocating arrays on the free store using operator `new[]`. The `vector` data type is an abstraction of the dynamic array and is the simplest and most common way to represent data in C++. Since a `vector` holds many values, we call them *containers*. A `vector` has the ability to expand or shrink to hold as many elements as required. An element can be inserted at a specified position without the programmer having to shift other elements "down". An element can be removed from a specified position without writing code to shift other elements "up" to fill the resulting hole(s). The management of dynamically allocated memory is completely abstracted away from users - the appropriate memory is allocated and released automatically.

This introduction will expose you to the following basic concepts about `vector`s:

- Defining and initializing `vector`s using constructors
- Functions to determine size: `vector::empty`, `vector::size`, `vector::max_size`
- Accessing elements: `vector::operator[]`, `vector::front`, `vector::back`
- Using iterators to access elements: `vector::begin`, `vector::end`, `vector::cbegin`, `vector::cend`
- Using range-`for` loop to access `vector` elements in order
- Adding and removing elements: `vector::push_back`, `vector::pop_back`
- Assignment operations: `vector::operator=`, `vector::assign`
- Passing `vector`s to functions and returning vector from functions
- Using global functions declared in header file `<algorithm>`. Some of the functions that we'll look at include: `std::max`, `std::min`, `std::min_element`, `std::max_element`, `std::merge`, `std::swap`, `std::reverse`, `std::find`, and `std::sort`

Defining `vector`s

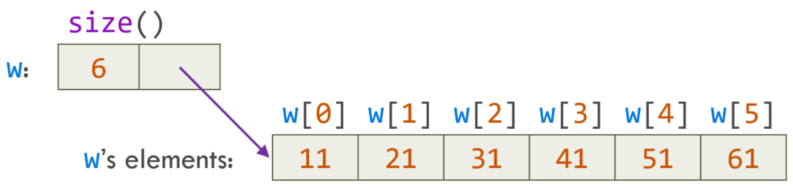
To define objects of type `vector`, programmers must include the appropriate header file, as in:

```cpp
// to use a vector, this header must be included
#include <vector>
```

Since `vector` data type is implemented as a class template, the element type must be specified in angle brackets:

```cpp
// t is an empty container of int values
// this is called default construction because the user has not provided
// any information about how to initialize the object
std::vector <int> t;
// u is container of 4 float elements whose values are initialized to 0.0f
std::vector<float> u(4);
// v is container of 5 double elements whose values are initialized to 1.1
std::vector <double> v(5, 1.1);
// w is a container of 6 int elements initialized with
// values in initialization list
std::vector<int> w {11, 21, 31, 41, 51, 1};
```

A `vector` is simply a sequence of elements that you can access by an index. For example, here is the `vector` called `w` that was defined above:



That is, the first element has index 0, the second index 1, and so on. We refer to an element by subscripting the name of the `vector` with the element's index, so here the value of `w[0]` is `11`, the value of `w[1]` is `21`, and so on. Indices for a `vector` always start with 0 and increase by 1. This should look familiar: the standard library `vector` is simply the C++ standard library's version of an old and well-known idea called *array* except that programmers using `vector`s don't have to worry about low-level details involving dynamic memory allocation and deallocation. The picture above is drawn so as to emphasize that it "knows its size"; that is, a `vector` doesn't just store its elements, it also stores its size.

We can copy construct `vector`s like this:

```cpp
// x is a container of ints and will have a copy of each
// element in w - this is called copy construction
std::vector<int> x (w); // also, x = w
// C++11 allows copy construction with initializer list syntax
std::vector<int> y {x};
```

The entire list of constructor functions are listed here. When a `vector` object goes out of scope, the destructor function is automatically invoked - the destructor will return the dynamically allocated memory that holds the `vector`'s elements back to the free store.

In addition to primitive types, `vector`s can hold elements of other types, as long as the element type is copy constructible and copy assignable.

```cpp
// s1 is default constructed as empty container of string elements
std::vector <std::string> s1;

// s2 is container of 4 string elements. First, appropriate vector ctor is
// automatically invoked - this ctor will use operator new [] to allocate
// allocate free store memory to store 4 string objects
// Second, string default ctor: string::string() is automatically invoked
// to intialize each each of these 4 string objects. Recall, the string
// default ctor will dynamically allocate memory to encapsulate the null
// C-style string ""
```

```cpp
std::vector<std::string> s2(4);

// s3 is container of 4 string elements. First, appropriate vector ctor is
// automatically invoked - this ctor will use operator new [] to allocate
// free store memory to store 4 string objects.
// Next, string ctor string::string(char const*) is automatically called to
// intialize each of 4 string objects to encapsulate C-style string "hello"
std::vector<std::string> s3(4, "hello");

// C++11 allows initialization with initialization list
// s4 is vector with 5 elements with each element initialized with ctor
// that takes corresponding C-style string in initialization list
std::vector<std::string> s4 {"today", "is", "a", "good", "day"};

// s5 is a container of strings that is a copy of s4
// First, vector copy ctor is automatically invoked to dynamically
// allocate memory for as many string elements as contained in s4
// Next, each string element of type string is initialized with
// corresponding string element in s4 using string copy ctor
std::vector<std::string> s5(s4);

// C++11 allows copy construction with initializer list syntax
std::vector<std::string> s6 {s5};
```

Recall that static arrays are not first-class objects in both C and C++. Therefore, static arrays cannot be specified as class template types. However, defining and accessing two-dimensional *arrays* is straightforward with `vector`:

```cpp
// this definition is illegal since static arrays are not first-class
// objects because of the array-name-to-pointer-to-first-element conversion
std::vector<int [4]> d1; // ERROR

// if you want two-dimensional array of ints, then specify template
// parameter type as vector<int> here, d2 is a container of 4 elements with
// each element itself being a vector<int>
std::vector<std::vector<int>> d2(4);

// C++11 library introduces a new type std::array that is "better" than a
// plain-old static array because std::array objects don't convert into
// pointers like static arrays do d3 is a container of 4 elements - each
// element is a static array of 5 ints
std::vector<std::array<int,5>> d3(4);
```

Accessing a `vector`s elements

Just as with arrays, the elements of a `vector` can be accessed with overloaded `operator[]` operator. A `vector` knows the number of elements in the dynamic array it encapsulates - member function `vector::size` returns the number of elements in the `vector`. Thus, the elements of a `vector` can be printed like this:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};
for(size_t i{0}; i < cities.size(); ++i) {
  std::cout << cities [i] << "\n";
}
```

The overloaded `operator[]` operator returns a reference to the indexed element. So, we can change the fourth element of `cities` from `"Shanghai"` to `"Santiago"`, as in:

```
std::string south_american_city {"Santiago"};
cities[3] = south_american_city;
```

Just as with static arrays, out-of-bounds reads and writes will result in undefined behaviors at runtime:

```
// ERROR - out-of-bounds write will cause undefined behavior at runtime
cities[10] = south_american_city;
```

Member functions `vector::front` and `vector::back` return references to the first and last elements of the `vector`, respectively. Here, we use `std::swap` standard library function (declared in `<utility>`) to swap a `vector`'s first and last elements:

```
// swap first and last elements of vector container cities
std::swap(cities.front(), cities.back()); // include <utility>
```

We can obtain pointers to the first and last elements of a `vector` by applying the address-of operator to the reference values returned by `front` and `back`, respectively. Here, we use these pointers to iterate over the container from first-to-last and last-to-first orders:

```
// print elements of container cities in front to back order:
for (std::string *p {&cities.front()}; p <= &cities.back(); ++p) {
  std::cout << *p << "\n";
}
// here, we print elements of the container cities in back to front order:
for (std::string *p = &cities.back(); p >= &cities.front(); --p) {
  std::cout << *p << "\n";
}
```

Using a `vector`, we can initialize a two-dimensional array of `int`s and print the table using the overloaded subscript operator:

```
std::vector<std::vector<int>> d2 {{1}, {11, 22}, {33, 44, 55},
                                  {66, 77, 88, 99}};
for (size_t i{0}; i < d2.size(); ++i) {
  for (size_t j{0}; j < d2[i].size(); ++j) {
    std::cout << d2[i][j] << ' ';
  }
  std::cout << "\n";
}
```

The output would look like this:

```
1
11 22
33 44 55
66 77 88 99

```

Using `begin()` and `end()`

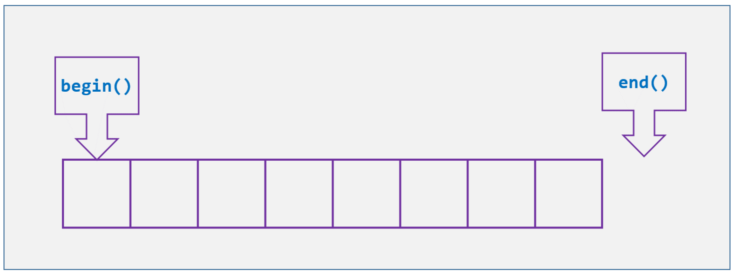
We've used subscripts to access elements of a `vector` and individual `char` elements of a `string`. We can get pointers to first and last members of `vector`s and `string`s using member functions `front` and `back`, respectively. These pointers can be used to access individual elements of `vector`s and `string`s:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};

// access char elements of a string using pointers
char *ptr_char = &cities[0].front(), *ptr_last_char = &cities[0].back();
while (ptr_char <= ptr_last_char) {
  std::cout << *ptr_char++;
}
std::cout << "\n";

// access string elements of a vector using pointers:
for (std::string *p = &cities.front(); p <= &cities.back(); ++p) {
  std::cout << *p << "\n";
}
```

The C++ standard library provides the concept of iterators that abstract the interface of ordinary pointers. Just like pointers, iterators give us indirect access to an object. As with pointers, we can use operators `++` and `--` to step forward and backward, respectively. We can use operators `==` and `!=` to determine whether two iterators represent the same position. We can assign iterators using operator `=`. We'll study iterators in more detail at a later date in the semester. For now, we'll use iterators to traverse `string`s since they simply encapsulate a `char *`. Likewise, we use iterators to traverse `vector`s since they encapsulate an ordinary pointer.

All container classes provide the same member functions that enable them to use iterators to navigate over their elements. The most important of these functions are as follows:

- `begin` returns an iterator that points to the first element in the container.
- `end` returns an iterator that points to the position after the last element of the container.



Thus, as shown in the picture, member functions `begin` and `end` define a *half-open range* that *includes the first element* but *excludes the last*. A half-open range has two advantages:

- There is a simple criterion to terminate loops that iterate over elements of a container. Loops simply continue as long as function `end` is not reached.
- Empty containers will not require special handling since function `begin` is equal to function `end`.

Rather than using subscripts, the following example demonstrates the use of iterators to traverse the elements of a `string` and convert every alphabetic character to uppercase:

```cpp
#include <cctype> // for std::isalpha, std::islower, std::toupper

std::string s {"Runtime error!!!"};
for (std::string::iterator it = s.begin(); it != s.end(); ++it) {
  if (std::isalpha(*it) && std::islower(*it)) {
    *it = std::toupper(*it);
  }
}
```

Every container defines two iterator types:

- `container::iterator` is provided to iterate over elements in read/write mode.
- `container::const_iterator` is provided to iterator over elements in read-only mode.

The following example demonstrates the use of read-only iterators to traverse the elements of a `vector`:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};

// iterate over the elements of container cities: vector::begin() is
// overloaded to return either read/write or read-only iterator. Because
// cit has type read-only iterator, cities.begin() will return read-only
// iterator
for (std::vector<std::string>::const_iterator cit = cities.begin();
     cit != cities.end(); ++cit) {
  std::cout << *cit << "\n";
}

// here, we explicitly call cbegin() and cend()
for (std::vector<std::string>::const_iterator cit = cities.cbegin();
     cit != cities.cend(); ++cit) {
  std::cout << *cit << "\n";
}
```

Using range-`for` loop

For simple loops that require visiting each element of the `vector`, the range-`for` loop is more preferable:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};
// read the following range-for loop as:
// "for each (reference to read-only) string x in container cities"
for (std::string const& x : cities) {
  std::cout << x << "\n";
}
```

The range-`for` will automatically iterate only over elements in a container. On the other hand, buffer overflow errors will arise when incorrect subscripts are inadvertently used with the overloaded `operator[]`. Therefore, a good way to ensure that subscripts are in range is to avoid subscripting altogether by using a range-`for` whenever possible. In fact, range-`for` loops for a container are implemented using the container's `begin` and `end` member functions. In general, the following range-`for` loop

```cpp
for (decl : coll) {
  statement
}
```

is equivalent to the following, if `container` provides member functions `begin` and `end`:

```cpp
for (auto _pos=coll.begin(), _end=coll.end(); _pos != _end; ++_pos) {
  decl = *pos_;
  statement
}
```

Growing and shrinking `vector`s

Consider the definition of a `vector` object `cities`:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};
```

Member function `vector::push_back` adds an additional element to the end of object `cities`, thereby increasing the size of `cities` by 1:

```cpp
// there are two things going on. first, a temporary unnamed string object
// is constructed from string literal "Santiago"
// next, the temporary string object is used to construct element 5 of cities
cities.push_back("Santiago");

// the two steps are more explicitly specified in this push_back statement
// that creates a temporary string object to initialize element 6 of cities
cities.push_back(std::string("Stockholm"));
```

Member function `vector::push_back` is useful for filling a `vector` from an input stream, as in:

```cpp
std::vector<std::string> cities;

// continue adding strings to vector container until end-of-file is signalled
// use CTRL-D to signal end-of-file to standard input stream
for (std::string tmp_str; std::cin >> tmp_str; cities.push_back(tmp_str)) {
  // empty by design
}
```

Member function `vector::pop_back` removes the last element of the `vector` effectively reducing the `vector`'s size by 1:

```cpp
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore","Shanghai"};
cities.pop_back();
// now the size of cities is reduced from 4 to 3 with element encapsulating
// "Shanghai" deleted. this means that destructor string::~string() is
// automatically invoked to ensure dynamic memory allocated to the last
// string object of cities is returned to the free store
```

As with arrays, it is up to the programmer to ensure that the `vector` is not empty when `pop_back` is called:

```cpp
std::vector<std::string> names; // empty container
names.pop_back(); // RUNTIME error - causes undefined behavior
```

Before calling member function `pop_back` on a potentially empty `vector`, it is better to check whether the container is empty or not by calling member function `vector::empty`:

```cpp
std::vector<std::string> names; // empty container
// this is better ...
if (names.empty()) {
  names.pop_back();
}
```

Assignments

The `vector` class overloads the assignment operator `=`:

```cpp
std::vector<std::string> cities1 {"Seattle", "San Jose",
                                  "Singapore", "Shanghai"};
std::vector<std::string> cities2 {"Toronto","Vancouver","Montreal"};
cities1 = cities2;
// First, the dtor for each element of cities1 is invoked.
// Second, the dynamic array previously allocated to store the 4 elements of
// cities1 is deleted.
// Third, cities1 calls new [] operator to allocate memory to store 3 string
// elements - each of the newly allocated string elements is initialized to
// encapsulate the null C-style string.
// Fourth, each string element of cities2 is assigned to the corresponding
// string element of cities1.
```

Member function `vector::assign` can be used to assign a range of elements:

```cpp
std::vector<std::string> cities1 {"Seattle", "San Jose",
                                  "Singapore", "Shanghai"};
std::vector<std::string> cities2 {"Toronto", "Vancouver", "Montreal",
                                  "Ottawa", "Calgary"};
cities1.assign(cities2.begin()+2, cities2.end());
// First, the dtor for each element of cities1 is invoked.
// Second, the dynamic array previously allocated to store the 4 elements of
// cities1 is deleted.
// Third, cities1 calls new [] operator to allocate memory to store 3 string
// elements - each of the newly allocated string elements is initialized to
// encapsulate the null C-style string.
// Fourth, each of 3 string elements of cities2 - starting from third
// element - are assigned to the corresponding string element of cities1.
```

Passing `vector`s to functions

With one caveat, `vector` objects can be used in function arguments in exactly the same way as other values:

```cpp
// pass-by-value semantics: vs is copy constructed
void print_vec_str(std::vector<std::string> vs) {
  // pass-by-value semantics: x is copy constructed/assigned
  for (std::string x : vs) {
    std::cout << x << "\n";
  }
}

std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};
print_vec_str(cities);
```

Recall that C and C++ functions implement *pass-by-value* semantics - that is, the value resulting from argument evaluation is copied on the stack and is then used to initialize the parameter. This means that argument `cities` in function call `print_vec_str()` is used to initialize parameter `vs` in the definition of function `print_vec_str`. We have previously learnt that *copy constructor* `vector::vector(vector const&)` is used to initialize `vs` with a copy of argument `cities`. We've also learnt that *destructor* `vector::~vector()` is automatically invoked when parameter `vs` goes out of scope when function `print_vec_str` terminates. Since `vector`s can encapsulate large amounts of data, making copies of arguments and their subsequent destruction can significantly impair the program's performance. Therefore, *pass-by-reference* semantics must be used when passing containers to functions to remove unnecessary calls to constructors and destructors:

```cpp
// pass-by-reference: vs is reference to container specified as function
// argument and therefore there is no unnecessary copy construction
void print_vec_str(std::vector<std::string>& vs) {
  // x is now reference to an element of vs - there is no need for copy
  // construction nor copy assignment
  for (std::string& x : vs) {
    std::cout << x << "\n";
  }
// since there are no objects constructed, destructor is not invoked
}

std::vector<std::string> cities {"Seattle","S an Jose",
                                 "Singapore", "Shanghai"};
print_vec_str(cities);
```

We can still do better. In function `print_vec_str`, we iterate over each element of the `vector` container in a read-only mode. We make this intent clear to the compiler and to clients of the function by using *pass-by-read-only-reference* semantics:

```cpp
// pass-by-read-only-reference: vs is read-only reference to container
// specified as function argument and therefore there is no unnecessary copy
// construction nor can there be inadvertent changes to container
void print_vec_str(std::vector<std::string> const& vs) {
  // x is now a read-only reference to an element of vs -
  // there is no need for copy construction nor copy assignment nor
  // can there be inadvertent changes to any element of vs
  for (std::string const& x : vs) {
    std::cout << x << "\n";
  }
// since there are no objects constructed, destructor is not invoked
}

std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};
print_vec_str(cities);
```

Unlike arrays which evaluate to a pointer-to-first-element-of-array, the size of a `vector` is encapsulated by the object. Therefore, addition of two `vector`s can be reliably implemented:

```
 1  #include <cassert> // for assert()
 2  
 3  // add left.size() number of values in left and right to result
 4  void add(std::vector<double> const& left, std::vector<double> const& right,
 5           std::vector<double>& result) {
 6    // C macros don't require std namespace
 7    assert(left.size() == right.size());
 8    assert(left.size() == result.size());
 9  
10    for (size_t i {0}; i < left.size(); ++i) {
11      result[i] = left[i] + right[i];
12    }
13  }
14  
15  // in calling function ...
16  std::vector<double> vone {1.1, 2.2, 3.3, 4.4};
17  std::vector<double> vtwo {10.10, 11.11, 12.12, 13.13};
18  std::vector<double> vthr(vone.size());
19  add(vone, vtwo, vthr);
```

It makes mathematical sense that to add two `vector`s, both must have the same number of elements. Proper continuation of the program is not possible if function `add` receives two `vector`s with different sizes. The principal ways to deal with unexpected behavior in C++ are exceptions while assertions are C's way of dealing with such behavior. Today, we'll describe the `assert` macro and leave C++ exceptions for a later date. The `assert` macro is declared in the C header `<cassert>`. The macro evaluates an expression and when the result is `false` then the program is terminated immediately. On lines 7 and 8, if assertions that the three `vector`s have the same lengths is false, the program will be terminated immediately and a diagnostic message is printed to standard error stream `cerr`.

A great advantage of `assert` is that we can let it disappear entirely by a simple macro declaration. Before including `<cassert>` define macro `NDEBUG`:

```
// if NDEBUG is defined, then assert is disabled
//#define NDEBUG
#include <cassert>
```

and all assertions are disabled if we pass macro `NDEBUG` to the compiler. For `gcc`, `clang`, `g++`, and `clang++` compilers, we use the `-D` flag to pass the macro, as in `-NDEBUG`. For Microsoft C/C++ compiler, we use the `/D` flag, as in `/DNDEBUG`.

Returning `vector`s from functions

`vector`s are copyable and can be returned by functions. This allows us to use a more natural notation. For example, the previously defined function `add` can make its intent clearer by returning the resultant `vector`:

```cpp
std::vector<double> add(std::vector<double> const& left,
                        std::vector<double> const& right) {
  // macros are replaced by preprocessor which knows nothing about
  // namespaces. hence, C/C++ macros are defined outside std namespace
  assert(left.size() == right.size());
  std::vector<double>::size_type sz = left.size();
  std::vector<double> result(sz);

  for (std::vector<double>::size_type i {0}; i < sz; ++i) {
    result[i] = left[i] + right[i];
  }
  return result;
}

// in the calling function ...
std::vector<double> vone {1.1, 2.2, 3.3, 4.4};
std::vector<double> vtwo {10.10, 11.11, 12.12, 13.13};
std::vector<double> vthr = add(vone, vtwo);
```

For well designed containers and types, C++ compilers since C++11 can optimize away unnecessary constructors and destructors even though the function returns a value. Both `string`s and `vector`s fall into the category of well designed types that will not generate unnecessary constructors and destructors.

Using C++ standard library algorithms

The standard library provides several standard global functions for processing elements of containers including sorting, searching, copying,modifying, reordering, and numerical processing. They are implemented once and using iterators are able to operate on any container type including `string`s, `vector`s, `list`s, and so on. We'll exercise a few algorithms including `min`, `max`, `min_element`, `max_element`, `sort`, `swap`, `reverse`, and `find` to process `string` and `vector` elements in the following examples. To use these global functions in source code, header file `<algorithm>` must be included.

Let's begin by authoring overloaded functions `PRINT` to print `string` and `vector<string>` containers:

```cpp
// print elements of string container
void PRINT(std::string const& cont, std::string const& prefix,
           std::string const& break_str = " ", // to print as separator
           std::string const& postfix = "\n") { // after values are printed
  std::cout << prefix;
  for (char const& x : cont) {
    std::cout << x << break_str;
  }
  std::cout << postfix;
}

```

```cpp
// print elements of vector container
void PRINT(std::vector<std::string> const& cont,
           std::string const& prefix,    // text to print before values
           std::string const& break_str = " ", // to print as separator
           std::string const& postfix = "\n") { // after values are printed
  std::cout << prefix;
  for (auto const& x : cont) {
    std::cout << x << break_str;
  }
  std::cout << postfix;
}
```

Merging two `vector<std::string>` containers is straightforward. The `merge` algorithm requires a half-open range for each container and a fifth iterator pointing to the element where the merged elements are to be stored.

```cpp
std::vector<std::string> cities1 {"Seattle", "San Jose", "Singapore",
                                  "Shanghai", "Stockholm", "Santiago"};
std::vector<std::string> cities2 {"Toronoto", "Vancouver", "Ottawa",
                                  "Calgary", "Montreal"};

// allocate enough elements to merge cities1 & cities2 into single vector
std::vector<std::string> cities(cities1.size() + cities2.size());

// merge ALL elements in cities1 and cities2 into container cities ...
// the first two arguments specify half-open range of elements
// in 1st source container cities1
// the next two arguments specify half-open range of elements
// in 2nd source container cities2
// the last argument specifies the start position at which merged elements
// will be copied into destination container cities ...
std::merge(cities1.begin(), cities1.end(),
           cities2.begin(), cities2.end(),
           cities.begin());
PRINT(cities, "cities:\n", "\n");
```

`min` and `max` algorithms can be used to determine the smaller and larger of two values, respectively:

```cpp
// compare first and last elements of container cities. recall that
// vector::front() and vector::back() return references to elements located
// at first and last positions, respectively in container ...
std::string const& sf = cities.front();
std::string const& sb = cities.back();

// min returns smaller of two values
std::cout << "Min of (" << sf << ", " << sb
          << "): " << std::min(sf, sb) << "\n";

// max returns the larger of two values
std::cout << "Max of (" << sf << ", " << sb
          << "): " << std::max(sf, sb) << "\n";
```

The following values are printed to standard output by the `min` and `max` algorithms on the merged container `cities`:

```
Min of (Seattle, Montreal): Montreal
Max of (Seattle, Montreal): Seattle
```

Unlike `min` and `max` algorithms that return values, `min_element` and `max_element` return an iterator pointing to the smallest and largest values in a half-open range:

```
// min_element returns iterator (pointer) to "smallest" element in range
// here we want pointer to smallest element (in value) of all elements
// in cities ...
std::vector<std::string>::const_iterator min_citer
                  = std::min_element(cities.cbegin(), cities.cend());
std::cout << "Minimum value in cities: " << *min_citer << "\n";

// max_element returns an iterator (pointer) to "largest" element in range
// here we want pointer to largest element (in value) of all elements
// in cities ...
std::vector<std::string>::const_iterator max_citer
                 = std::max_element(cities.cbegin(), cities.cend());
std::cout << "Maximum value in cities: " << *max_citer << "\n";
```

The following values are printed to standard output by the `min_element` and `max_element` algorithms on the merged container `cities`:

```
Minimum value in cities: Calgary
Maximum value in cities: Vancouver
```

The `swap` algorithm (now declared in `<utility>`) can be used to swap elements between two containers:

```
// an expensive swap of containers cities1 and cities2
std::swap(cities1, cities2);
PRINT(cities1, "Cities1:\n", "\n");
PRINT(cities2, "Cities2:\n", "\n");
```

As the name implies, the `reverse` algorithm reverses the order of elements in a half-open range:

```
// reverses order of all elements in cities ...
std::reverse(cities.begin(), cities.end());
PRINT(cities, "Cities:\n", "\n");
```

The `find` algorithm returns an iterator to the first element in a half-open range equal to a search value:

```
std::string search_str("Vancouver"); // search for "Vancouver"
// find returns iterator (pointer) to 1st element that matches search value
// if search value is not found, find returns iterator to 2nd argument
std::vector<std::string>::const_iterator cit =
                      std::find(cities.begin(), cities.end(), search_str);
if (cit != cities.end()) {
  std::cout << search_str << " exists in cities!!!" << "\n";
} else {
  std::cout << search_str << " doesn't exist in cities!!!" << "\n";
}
```

```cpp

// search for "Hong Kong" in cities ...
search_str = "Hong Kong";
cit = std::find(cities.begin(), cities.end(), search_str);
if (cit != cities.end()) {
  std::cout << search_str << " exists in cities!!!" << "\n";
} else {
  std::cout << search_str << " doesn't exist in cities!!!" << "\n";
}
```

The `sort` algorithm sorts the elements in a range in ascending order:

```cpp
// sort all elements in cities in the default ascending order ...
std::sort(cities.begin(), cities.end());
PRINT(cities, "Cities:\n", "\n");
```

To change the default sorting criterion from ascending to descending order, we author a comparison binary function that accepts two values from the range as parameters and returns a `bool` value if the first parameter is *greater-than* the second parameter:

```cpp
// this comparison function between two strings returns true if
// 1st parameter is lexicographically greater than 2nd parameter ...
// we use this as a call-back function to std::sort() when we wish to change
// sorting criterion from default ascending order to descending order
bool cmp_str_greater_than(std::string const& lhs, std::string const& rhs) {
  return (lhs > rhs) ? true : false;
}

// we pass a pointer to string comparison function cmp_str_greater_than()
// that will be used by std::sort() to change the default sorting criterion
// from ascending to descending order ...
std::sort(cities.begin(), cities.end(), cmp_str_greater_than);
PRINT(cities, "Cities:\n", "\n");
```

Size and capacity

A `vector` is an abstraction that manages its elements using a dynamic C-style array. It automatically grows to accommodate as many elements as are put into it, provided only that its *maximum size* is not exceeded. This maximum size is obtained by calling member function `vector::max_size`. Growth is handled by doing the equivalent of calling the C standard library function `std::realloc` whenever more memory is required. This `realloc`-like change in size of previously allocated memory block has four parts:

1. Allocate a new memory block that is some multiple of the `vector`'s current capacity. In most implementations, the capacity increases by a factor of two each time, that is, the capacity is doubled each time the `vector` must be expanded.
2. Copy all elements from the `vector`'s old memory into its new memory.
3. Destroy objects in the old memory.
4. Deallocate old memory.

Given all of this allocation, deallocation, copying, and destruction, it shouldn't be of surprise to learn that these steps can be expensive. Naturally, we don't want to perform these steps any more than frequently than we have to. Member function `vector::reserve` can be used to minimize the number of reallocations that must be performed. Before understanding how `vector::reserve` can help reduce reallocations, we must understand four interrelated and confusing member functions that manage the size and capacity of `vector`s. Among the standard containers, only `vector` and `string` provide all of these four functions. Although we limit our discussion to `vector`s, the principles are also relevant to the efficient and correct use of `string`s.

- `vector::size` returns the number of elements currently contained in the `vector` container. It does not specify how much memory the container has allocated for the elements it holds.
- `vector::capacity` returns the number of elements the container can hold in the memory it has already allocated. This is how many total elements the container can hold in that memory, not how many more elements it can hold. To find out how much unoccupied memory a `vector` has, the value returned by `vector::size` must be subtracted from the value returned by `vector::capacity`. If `vector::size` and `vector::capacity` return the same value, there is no room in the memory block allocated to the container for additional elements, and the next insertion (via `vector::insert` or `vector::push_back`) will trigger the four reallocation steps described earlier.
- `vector::resize(size_t n)` forces the container to change the number of elements it holds to `n`. After the call to `vector::resize`, `vector::size` will return `n`. If `n` is smaller than the current size, elements at the end of the container will be destroyed. If `n` is larger than the current size, new default-constructed elements will be added to the end of the container. If `n` is larger than the current capacity, a reallocation will take place before the elements are added.
- `vector::reserve(size_t n)` forces the container to change its capacity to at least `n`, provided `n` is not less than the current size. This typically forces a reallocation, because the capacity needs to be increased. If `n` is less than the current capacity, `vector` ignores the call and does nothing.

From these descriptions, it should be clear that reallocations (including raw memory allocations and deallocations, object copying and destruction) will occur whenever an element needs to be inserted and the container's capacity is insufficient. The key to avoiding reallocations, then, is to use `vector::reserve` to set a `vector`'s capacity to a sufficiently large value as soon as possible, ideally right after the `vector` container is constructed. For example, to create a `vector<int>` holding the values 1 through 1000 without using `vector::reserve`, we might do it like this:

```cpp
std::vector<int> v;
for (size_t i {1}; i <= 1000; ++i) {
  v.push_back(i);
}
```

In `g++`, this code will result in 12 reallocations during the course of the loop while Microsoft's `cl` will result in about 20 reallocations. You can check the progress of memory reallocations using this code:

```cpp
std::vector<int> v;
std::vector<int>::size_type curr_cap = v.capacity();
std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
for (size_t i {1}; i <= 1000; ++i) {
  v.push_back(i);
  if (v.capacity() != curr_cap) {
    curr_cap = v.capacity();
    std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
  }
}
```

Modifying the previous code to use `vector::reserve` gives us:

```cpp
std::vector<int> v;
v.reserve(1000); // pre-allocate memory for 1000 elements
std::vector<int>::size_type curr_cap = v.capacity();
std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
for (size_t i {1}; i <= 1000; ++i) {
  v.push_back(i);
  if (v.capacity() != curr_cap) {
    curr_cap = v.capacity();
    std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
  }
}
```

The use of `vector::reserve` results in zero reallocations during the loop.

There are two common ways to use `vector::reserve` to avoid unneeded reallocations. The first is applicable when the size of the `vector` container is known exactly or approximately. In that case, as shown in the code above, the appropriate amount of space can be reserved in advance. The second way is to reserve the maximum space the program might ever need, then, once all the data has been added, excess capacity is trimmed off using member function `shrink_to_fit`:

```cpp
std::vector<int> v;
v.reserve(10'000); // pre-allocate memory for 10,000 elements
std::vector<int>::size_type curr_cap = v.capacity();
std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
for (size_t i {1}; i <= 1000; ++i) {
  v.push_back(i);
  if (v.capacity() != curr_cap) {
    curr_cap = v.capacity();
    std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
  }
}
// we trim off excess capacity ...
v.shrink_to_fit();
std::cout << "size: " << v.size() << " | cap: " << v.capacity() << "\n";
```