Rationale behind using vectors

The final technical vote of the **C++ Standard** took place on November 14th, 1997; that was more than five years ago. However, significant parts of the Standard, especially the Standard Library, are still not very popular among many C++ users. A constant reader of CodeGuru's C++ forums will soon notice that many questions and answers still imply hand-crafted solutions that could be very elegantly solved by using the Standard Library.

One issue that comes up very often is the use of C-style arrays, with all their problems and drawbacks. People seem to be scared of the standard *vector* and its brother *deque*. One reason might be that the Standard Library documentations are mostly pretty elliptic and esoteric.

In this article, I will take *vector* and try to explain it in a way that is more accessible and understandable. I do not claim that this article is by any means complete. It is meant to give you a start in using *vector* and to help you avoid the most common pitfalls. We will start small and will not try to handle the topic very academically.

Introduction to vector

Vector is a template class that is a perfect replacement for the good old C-style arrays. It allows the same natural syntax that is used with plain arrays but offers a series of services that free the **C++ programmer** from taking care of the allocated memory and help operating consistently on the contained objects.

The first step using *vector* is to include the appropriate header:

```
1. #include <vector>
```

Note that the header file name does not have any extension; this is true for all of the Standard Library header files. The second thing to know is that all of the Standard Library lives in the namespace *std*. This means that you have to resolve the names by prepending *std::* to them:

```
1. std::vector<int> v;    // declares a vector of integers
```

For small projects, you can bring the entire namespace *std* into scope by inserting a using directive on top of your **cpp** file:

```
1. #include <vector>
2. using namespace std;
3. //...
4. vector<int> v;         // no need to prepend std:: any more
```

This is okay for small projects, as long as you write the using directive in your cpp file. Never write a using directive into a header file! This would bloat the entire namespace *std* into each and every cpp file that includes that header. For larger projects, it is better to explicitly qualify every name accordingly. I am not a fan of such shortcuts. In this article, I will qualify each name accordingly. I will introduce some *typedef*s in the examples where appropriate—for better readability.

Now, what is **std::vector<T> v;**? It is a template class that will wrap an array of Ts. In

this widely used notation, 'T' stands for any data type, built-in, or user-defined class. The *vector* will store the Ts in a contiguous memory area that it will handle for you, and let you access the individual Ts simply by writing *v[0]*, *v[1]*, and so on, exactly like you would do for a C-style array.

Note that for bigger projects it can be tedious to repeatedly write out the explicit type of the vectors. You may use a *typedef* if you want:

```
1. typedef std::vector<int> int_vec_t;    // or whatever you
2.                                        // want to name it
3. //...
4. int_vec_t v;
```

Do **not** use a macro!

```
1. #define int_vec_t std::vector<int> ;    // very poor style!
```

For the beginning, let's see what a *vector* can do for us. Let's start small and take the example of an array of integers. If you used plain arrays, you had either a static or a dynamic array:

```
1. size_t size = 10;
2. int sarray[10];
3. int *darray = new int[size];
4. // do something with them:
5. for(int i=0; i<10; ++i){
6.     sarray[i] = i;
7.     darray[i] = i;
8. }
9. // don't forget to delete darray when you're done
10. delete [] darray;
```

Let's do the same thing using a *vector*:

```
1. #include <vector>
2. //...
3. size_t size = 10;
4. std::vector<int> array(size);    // make room for 10 integers,
5.                                  // and initialize them to 0
6. // do something with them:
7. for(int i=0; i<size; ++i){
8.     array[i] = i;
9. }
10. // no need to delete anything
```

As you see, *vector* combines the advantages of both the static and the dynamic array because it takes a non-const size parameter such as the dynamic one and automatically deletes the used memory like the static one.

The standard *vector* defines the *operator []*, to allow a "natural" syntax. For the sake of performance, the *operator []* does not check whether the index is a valid one. Similar to a C-style array, using an invalid index will mostly buy you an access violation.

In addition to *operator []*, *vector* defines the member function *at()*. This function does the same thing as the *operator []*, but checks the index. If the index is invalid, it will throw an object of class *std::out_of_range*.

```
1. std::vector<int> array;
2. try{
3.     array.at(1000) = 0;
4. }
5. catch(std::out_of_range o){
```

```
6.     std::cout<<o.what()<<std::endl;
7. }
```

Depending on the implementation of the **C++ Standard Library** you use, the above snippet will print a more or less explicit error message. STLPort prints the word "vector", the Dinkumware implementation that comes with Visual C++ prints "invalid vector<T> subscript". Other implementations may print something else.

Note that *vector* is a standard container. The controlled sequence also can be accessed using iterators. More on iterators later in this article. For now, let's keep it simple.

Now, what if you don't know how many elements you will have? If you were using a C-style array to store the elements, you'd either need to implement a logic that allows to grow your array from time to time, or you would allocate an array that is "big enough." The latter is a poor man's approach and the former will give you a headache. Not so *vector*:

```
1. #include <vector>
2. #include <iostream>
3. //...
4. std::vector<char> array;
5. char c = 0;
6. while(c != 'x'){
7.   std::cin>>c;
8.   array.push_back(c);
9. }
```

In the previous example, *push_back()* appends one element at a time to the array. This is what we want, but it has a small pitfall. To understand what that is, you have to know that a *vector* has a so-called 'controlled sequence' and a certain amount of allocated storage for that sequence. The controlled sequence is just another name for the array in the guts of the *vector*. To hold this array, *vector* will allocate some memory, mostly more than it needs. You can *push_back()* elements until the allocated memory is exhausted. Then, *vector* will trigger a reallocation and will grow the allocated memory block. This can mean that it will have to move (that means: copy) the controlled sequence into a larger block. And copying around a large number of elements can slow down your application dramatically. Note that the reallocation is absolutely transparent for you (barring catastrophic failure—out of memory). You need to do nothing; *vector* will do all what that takes under the hood. Of course, there is something you can do to avoid having *vector* reallocate the storage too often. Just read on.

In the previous example, we declared the *vector* using its default constructor. This creates an empty *vector*. Depending on the implementation of the Standard Library being used, the empty *vector* might or might not allocate some memory "just in case." If we want to avoid a too-often reallocation of the *vector*'s storage, we can use its *reserve()* member function:

```
1. #include <vector>
2. #include <iostream>
3. //...
4. std::vector<char> array;
5. array.reserve(10);    // make room for 10 elements
6. char c = 0;
7. while(c != 'x'){
8.   std::cin>>c;
9.   array.push_back(c);
10. }
```

The parameter we pass to *reserve()* depends on the context, of course. The function

reserve() will ensure that we have room for at least 10 elements in this case. If the *vector* already has room for the required number of elements, *reserve()* does nothing. In other words, *reserve()* **will grow the allocated storage of the *vector*, if necessary, but will never shrink it.**

As a side note, the following two code snippets are not the same thing:

```
1. // snip 1:
2. std::vector<int> v(10);
3. // snip 2:
4. std::vector<int> v;
5. v.reserve(10);
```

The first snippet defines a *vector* containing 10 integers, and initializes them with their default value (0). If we hadn't integers but some user-defined class, *vector* would call the default ctor 10 times and contain 10 readily constructed objects. The second snippet defines an empty *vector*, and then tells it to make room for 10 integers. The *vector* will allocate enough memory to hold at least 10 integers, but will not initialize this memory. If we had no integers, but some user-defined class, the second snippet wouldn't construct any instance of that class.

To find out how many elements would fit in the currently allocated storage of a *vector*, use the *capacity()* member function. To find out how many elements are currently contained by the *vector*, use the *size()* member function:

```
1. #include <vector>
2. #include <iostream>
3. //...
4. std::vector<int> array;
5. int i = 999;          // some integer value
6. array.reserve(10);    // make room for 10 elements
7. array.push_back(i);
8. std::cout<<array.capacity()<<std::endl;
9. std::cout<<array.size()<<std::endl;
```

This will print

```
1. 10
2. 1
```

That means that the number of elements that can be added to a *vector* without triggering a reallocation always is *capacity() - size()*.

Note that, for the previous example, only 0 is a valid index for *array*. Yes, we have made room for at least 10 elements with *reserve()*, but the memory is not initialized. Because *int* is a built-in type, writing all 10 elements with *operator []* would actually work, but we would have a *vector* that is in an inconsistent state, because *size()* would still return 1. Moreover, if we tried to access the other elements than the first using *array.at()*, a ***std::out_of_range*** would be thrown. At a first sight, this may seem inconvenient, but a closer look reveals why this is so: If the *vector* contained objects of a user-defined class, *reserve()* wouldn't call any ctor. Accessing a not-yet-constructed object has undefined results and is a no-no in any case. The important thing to remember is that the role of *reserve()* is to minimize the number of potential reallocations and that it will not influence the number of elements in the controled sequence. A call to *reserve()* with a parameter smaller than the current *capacity()* is benign—it simply does nothing.

The correct way of enlarging the number of contained elements is to call *vector*'s member function *resize()*. The member function *resize()* has following properties:

- If the new size is larger than the old size of the *vector*, it will preserve all elements already present in the controlled sequence; the rest will be initialized according to the second parameter. If the new size is smaller than the old size, it will preserve only the first new_size elements. The rest is discarded and shouldn't be used any more—consider these elements invalid.
- If the new size is larger than *capacity()*, it will reallocate storage so all new_size elements fit. *resize()* will never shrink *capacity()*.

Example:

```
std::vector<int> array;   // create an empty vector
array.reserve(3);         // make room for 3 elements
                          // at this point, capacity() is 3
                          // and size() is 0
array.push_back(999);     // append an element
array.resize(5);          // resize the vector
                          // at this point, the vector contains
                          // 999, 0, 0, 0, 0
array.push_back(333);     // append another element into the vector
                          // at this point, the vector contains
                          // 999, 0, 0, 0, 0, 333
array.reserve(1);         // will do nothing, as capacity() > 1
array.resize(3);          // at this point, the vector contains
                          // 999, 0, 0
                          // capacity() remains 6
                          // size() is 3
array.resize(6, 1);       // resize again, fill up with ones
                          // at this point the vector contains
                          // 999, 0, 0, 1, 1, 1
```

Another way to enlarge the number of controlled elements is to use *push_back()*. In certain cases, this might be more efficient than calling *resize()* and then writing the elements. Let's have a closer look under the hood of *vector*, by looking at the following example:

```
class X
{
public:
    X():val_(0){}
    X(int val):val_(val){}
    int get(){return val_;}
    void set(int val){val_=val;}
private:
    int val_;
};
//....
std::vector<X> ax;         // create an empty vector containing
                           // objects of type class X
// version 1:
ax.resize(10);             // resize the controlled sequence
for(int i=0; i<10; ++i){
    ax[i].set(i);          // set each element's value
}
//...
// version 2:
ax.reserve(10);            // make room for 10 elements
for(int i=0; i<10; ++i){
    ax.push_back(X(i));    // insert elements using the second ctor
}
```

The two versions are equivalent, meaning that they will produce the same result. In both cases, we start with an empty *vector*. In the first version, we use *resize()* to grow the size of the controlled sequence to 10 elements. This will not only reallocate the *vector*s

storage, but will also construct a sequence of 10 elements, using the default ctor of X. When *resize()* is finished, we will have 10 valid objects of type X in our *vector*, all of them having *val_ == 0*, because that's what the default ctor of X does. In a second step, we pick every X in the sequence and use *X::set()* to change its *val_*.

In the second version, we call *reserve()* to make room for 10 elements. The *vector* will reallocate its storage and do nothing more than that. No element is constructed yet. In a second step, we create 10 objects of type X using the second ctor, thus giving them directly the correct value, and *push_back()* them into the vector.

Which method is more efficient? That probably also depends on the implementation of the Standard Library, but the second version is likely to be slightly more efficient because it doesn't call *X::set()* for each element.

Now that we have seen how to declare a *vector* and how to fill it up, let's see how we can operate on it. We will start with an analogy to C-style arrays and will progressively discover other possibilities, that are better or safer.

There are two ways of accessing a C-style array: either by using the subscript operator, or by using pointers. Also, passing a C-style array to a function means passing a pointer to the first element. Can we do the same thing with a *vector*? The answer is yes. Let's take a small example:

```
 1. #include <iostream>
 2.
 3. double mean(double *array, size_t n)
 4. {
 5.     double m=0;
 6.     for(size_t i=0; i<n; ++i){
 7.         m += array[i];
 8.     }
 9.     return m/n;
10. }
11.
12. int main()
13. {
14.     double a[] = {1, 2, 3, 4, 5};
15.     std::cout<<mean(a, 5)<<std::endl;    // will print 3
16.     return 0;
17. }
```

When we say *mean(a, 5)*, the first parameter actually is the address of the first element in the array *&a[0]*. We know that a *vector* is required to keep its elements in a contiguous block of memory, in order. That means that we can pass the address of the first element of a *vector* to the function *mean()* and it will work:

```
 1. int main()
 2. {
 3.     std::vector<double> a;
 4.     a.push_back(1);
 5.     a.push_back(2);
 6.     a.push_back(3);
 7.     a.push_back(4);
 8.     a.push_back(5);
 9.     std::cout<<mean(&a[0], 5)<<std::endl;    // will print 3
10.     return 0;
11. }
```

That's nice, but it's still not quite the same. We were able to directly initialize the C-style array, but we had to *push_back()* the elements into the vector. Can we do better? Well, yes. We cannot **directly** use an initializer list for the *vector*, but we can use an

intermediary array:

```
1. double p[] = {1, 2, 3, 4, 5};
2. std::vector<double> a(p, p+5);
```

Here we use another constructor provided by *vector*. It takes two parameters: a pointer to the first element of a C-style array and a pointer to one past the last element of that array. It will initialize the *vector* with a copy of each element in the array. Two things are important to note: The array is copied and it does not somehow go into the possession of the newly created *vector*, and the range we supply is from the first element to one past the last element in the array.

Understanding the second point is crucial when working with *vector*s or any other standard containers. The controlled sequence is always expressed in terms of **[first, one-past-last)**—not only for ctors, but also for every function that operates on a range of elements.

When taking the address of elements contained in a *vector*, there is something you have to watch out for: an internal reallocation of the *vector* will invalidate the pointers you hold to its elements.

```
1. std::vector<int> v(5);
2. int *pi = &v[3];
3. v.push_back(999); // <-- may trigger a reallocation
4. *pi = 333;        // <-- probably an error, pi isn't valid any more
```

In the previous example, we take the address of the fourth element of the *vector* and store it in *pi*. Then we *push_back()* another element to the end of the *vector*. Then we try to use *pi*. Boom! The reason is that *push_back()* may trigger a reallocation of *v*'s internal storage if this is not large enough to hold the additional element, too. *pi* will then point to a memory address that has just been deleted, and using it has undefined results. The bad news is that the *vector* might or might not reallocate the internal storage—you can't tell on the general case. The solution is either not to use pointers that might have been invalidated, or to make sure that the *vector* won't reallocate. The latter means to use *reserve()* wisely in order to have the *vector* handle memory (re)allocation at defined times.

From the member functions we have seen so far, only *push_back()* and *resize()* can invalidate pointers into the *vector*. There are other member functions that invalidate pointers; we will discuss them later in this tutorial.

Note that both the subscript operator and the member function *at()* never invalidate pointers into the *vector*.

Speaking of pointers into the *vector*, we can introduce a standard concept at this point: *iterators*. Iterators are the way the Standard Library models a common interface for all containers—*vector, list, set, deque*, and so on. The reason is that operations that are "natural" for one container (like subscripting for *vector*) do not make sense for other containers. The Standard Library needs a common way of applying algorithms like iterating, finding, sorting to all containers—thus the concept of iterators.

An iterator is a handle to a contained element. You can find an exact definition in your favorite textbook, if you want. The internal representation of an iterator is irrelevant at this point. Important is that if you have an iterator, you can dereference it to obtain the element it "points" to (for *vector* the most natural implementation of an iterator is indeed a plain vanilla pointer—but don't count on this). Let's get a grip on iterators with a small example:

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

int main()
{
    std::vector<double> a;
    std::vector<double>::const_iterator i;
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);
    for(i=a.begin(); i!=a.end(); ++i){
        std::cout<<(*i)<<std::endl;
    }
    return 0;
}
```

Let's take this small program step by step:

```
std::vector<double>::const_iterator i;
```

This declares a const iterator *i* for a *vector<double>*. We are using a const iterator because we do not intend to modify the contents of the *vector*.

```
...i=a.begin();...
```

The member function *begin()* returns an iterator that "points" to the first element in the sequence.

```
...i!=a.end();...
```

The member function *end()* returns an iterator that "points" to one-past-the-last-element in the sequence. Note that dereferencing the iterator returned by *end()* is illegal and has undefined results.

```
...++i
```

You can advance from one element to the next by incrementing the iterator.

Note that the same program, but using pointers instead of iterators, leads to a very similar construct:

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

int main()
{
    std::vector<double> a;
    const double *p;
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);
    for(p=&a[0]; p!=&a[0]+5; ++p){
        std::cout<<(*p)<<std::endl;
    }
    return 0;
}
```

So, if we can use pointers to basically achieve the same thing in the same way, why bother with iterators at all? The answer is that we have to use iterators if we want to apply some standard algorithm, like sorting, to the *vector*. The Standard Library does

not implement the algorithms as member functions of the various containers, but as free template functions that can operate on many containers.

The combination of standard containers in general (and *vector* in particular) and standard algorithms, is a very powerful tool; unfortunately, much too often neglected by programmers. By using it you can avoid large portions of hand crafted, error-prone code, and it enables you to write compact, portable, and maintainable programs.

Let's have a look at the member functions *vector* provides:

Constructors

A complete set of **C++ constructors** , **C++ destructor**, and copy operator is provided. Let's have a look at them on the example of a vector of standard strings:

```
1. typedef std::vector<std::string> str_vec_t;
2. str_vec_t v1;                        // create an empty vector
3. str_vec_t v2(10);                    // 10 copies of empty strings
4. str_vec_t v3(10, "hello");           // 10 copies of the string
5.                                      // "hello"
6. str_vec_t v4(v3);                    // copy ctor
7.
8.     std::list<std::string> sl;       // create a list of strings
9.                                      // and populate it
10.    sl.push_back("cat");
11.    sl.push_back("dog");
12.    sl.push_back("mouse");
13.
14. str_vec_t v5(sl.begin(), sl.end()); // a copy of the range in
15.                                     // another container
16.                                     // (here, a list)
17.
18. v1 = v5;                            // will copy all elements
19.                                     // from v5 to v1
```

The *assign()* function

The *assign()* function will reinitialize the vector. We can pass either a valid element range using the [first, last) iterators or we can specify the number of elements to be created and the element value.

```
1. v1.assign(sl.begin(), sl.end());    // copies the list into
2.                                     // the vector
3. v1.assign(3, "hello");              // initializes the vector
4.                                     // with 3 strings "hello"
```

The assignment completely changes the elements of the *vector*. The old elements (if any) are discarded and the size of the *vector* is set to the number of elements assigned. Of course, *assign()* may trigger an internal reallocation.

Stack operations

We have seen the function *push_back()*. It appends an element to the end of the controlled sequence. There is a counterpart function, *pop_back()*, that removes the last element in the controlled sequence. The removed element becomes invalid, and *size()* is decremented. Note that *pop_back()* does not return the value of the popped element. You have to peek it before you pop it. The reason why this is so is exception safe. Popping on an empty vector is an error and has undefined results.

```
1. std::vector<int> v;
2. v.push_back(999);
3. v.pop_back();
```

Note that *pop_back()* does not shrink the *capacity()*.

Predefined iterators

We have seen the iterators *begin()* and *end()*. They point to the first, respectively, to one-past-the-last element in the controlled sequence. There also are *rbegin()* and *rend()* which point to the first, respectively, to the one-past-the-last element of the reverse sequence. Note that both *rbegin()* and *rend()* return the type *reverse_iterator* (or *const_reverse_iterator* for their const versions)—which is not the same as *iterator*, (respectively *const_iterator*). To obtain a "normal" iterator from a reverse iterator, use *reverse_iterator*'s *base()* member function:

```
1. std::vector<int> v;
2. v.push_back(999);
3. std::vector<int>::reverse_iterator r = v.rbegin();
4. std::vector<int>::iterator i = r.base(); // will point to the last
5.                                          // element in the sequence
```

Element access

We have seen the subscript *operator []* that provides unchecked access and the member function *at()*, which will throw an object of type *std::out_of_range* if the index passed is invalid. Two other member functions exist, *front()* and *back()*, which return a reference to the first, respectively the last element in the controlled sequence. Note that they do not return iterators!

```
1. std::vector<int> v;
2. v.push_back(999);
3. // fill up the vector
4. //...
5. // following statements are equivalent:
6. int i = v.front();
7. int i = v[0];
8. int i = v.at(0);
9. int i = *(v.begin());
10. // following statements are equivalent:
11. int j = v.back();
12. int j = v[v.size()-1];
13. int j = v.at(v.size()-1);
14. int j = *(v.end()-1);
```

Note that we cannot write **(--v.end())* because *v.end()* is not a l-value.

List operations

A few operations provided by *vector* are actually native for *list*. They are provided by the most containers and deal with inserting and erasing elements in the middle of the controlled sequence. Let's demonstrate them by some examples:

```
1. #include <vector>
2. #include <iostream>
3. int main()
4. {
5.     std::vector<int> q;
6.     q.push_back(10); q.push_back(11); q.push_back(12);
```

```
 7.
 8.     std::vector<int> v;
 9.     for(int i=0; i<5; ++i){
10.         v.push_back(i);
11.     }
12.     // v contains 0 1 2 3 4
13.
14.     std::vector<int>::iterator it = v.begin() + 1;
15.     // insert 33 before the second element:
16.     it = v.insert(it, 33);
17.     // v contains 0 33 1 2 3 4
18.     // it points to the inserted element
19.
20.     //insert the contents of q before the second element:
21.     v.insert(it, q.begin(), q.end());
22.     // v contains 0 10 11 12 33 1 2 3 4
23.     // iterator 'it' is invalid
24.
25.     it = v.begin() + 3;
26.     // it points to the fourth element of v
27.     // insert three time -1 before the fourth element:
28.     v.insert(it, 3, -1);
29.     // v contains 0 10 11 -1 -1 -1 12 33 1 2 3 4
30.     // iterator 'it' is invalid
31.
32.     // erase the fifth element of v
33.     it = v.begin() + 4;
34.     v.erase(it);
35.     // v contains 0 10 11 -1 -1 12 33 1 2 3 4
36.     // iterator 'it' is invalid
37.
38.     // erase the second to the fifth element:
39.     it = v.begin() + 1;
40.     v.erase(it, it + 4);
41.     // v contains 0 12 33 1 2 3 4
42.     // iterator 'it' is invalid
43.
44.     // clear all of v's elements
45.     v.clear();
46.
47.     return 0;
48. }
```

Note that both *insert()* and *erase()* may invalidate any iterators you might hold. The first version of *insert()* returns an iterator that points to the inserted element. The other two versions return *void*. Inserting elements may trigger a reallocation. In this case, all iterators in the container become invalid. If no reallocation occurs (for example, by a call to *reserve()* prior to inserting), only iterators printing between the insertion point and the end of the sequence become invalid.

Erasing elements never triggers a reallocation, nor does it influence the *capacity()*. However, all iterators that point between the first element erased and the end of the sequence become invalid.

Calling *clear()* removes all elements from the controlled sequence. The memory allocated is not freed, however. All iterators become invalid, of course.

Note that both *insert()* and *erase()* are not very efficient for *vector*s. They are expected to perform in amortized linear time, O(n)+. If your application often uses insertion and erasure, *vector* probably isn't the best choice of a container for you.

Comparison operations

You can compare the contents of two vectors on an element-by-element basis using the operators ==, != and <. Two *vector*s are equal if both have the same *size()* and the elements are correspondingly equal. Note that the *capacity()* of two equal *vector*s need not to be the same. The operator < orders the *vector*'s lexicographically.

```
1. std::vector<int> v1, v2;
2. //...
3. if(v1 == v2) ...
```

Swapping contents

Sometimes, it is practical to be able to *swap()* the contents of two vectors. A common application is forcing a *vector* to release the memory it holds. We have seen that erasing the elements or clearing the *vector* doesn't influence its *capacity()* (in other words, the memory allocated). We need to do a small trick:

```
1. std::vector<int> v;
2. //...
3. v.clear();
4. v.swap(std::vector<int>(v));
```

Normally (see below), *vector*s simply swap their guts. In the previous example, we create a temporary *vector* by using the copy ctor and *swap()* its contents with *v*. The temporary object will receive the entire memory held by *v* and *v* will receive the memory held by the temporary object—which is likely to allocate nothing on creation. The temporarily created *vector* is destroyed at the end of the above statement, and all the memory formally held by *v* is freed.

The *vector* class template has a second, default template parameter:

```
1. template<class T, class A = allocator<T> >
2.         class vector ...
```

The *allocator* is a class that supplies the functions used by the container to allocate and deallocate memory for its elements. In this tutorial, we assumed that we have a default *allocator* and we will continue to assume this. For the sake of completeness, note that *swap()* will perform in constant time (simply swap the guts of the two *vector*s) if both *allocator*s are the same. This is in most cases so.

In this first tutorial, we have scratched the surface of the Standard Library and met *std::vector*. In the next tutorial, we will have a look at more advanced topics related to *vector*, respectively applying standard algorithms to it, and will discuss design decisions, such as the question of when to store objects and when to store pointers to objects in the *vector*. We also will introduce a close relative of the *vector*, that is, *std::deque*.

Related Articles

- Allocators (STL)
- Function Objects (STL)

Comments

- **Undoubtedly the best on Vectors**

  Posted by *Nick* on *02/05/2016 09:27pm*

  The best article you can come across. Keep up the good work pal.

  Reply

- **Thanks**

  Posted by *froggy25* on *02/03/2016 10:32pm*

  Thank you very much for this tutorial. Great introduction to the vector library.

  Reply

- **Thank you**

  Posted by *Pouria* on *01/30/2016 07:10am*

  I needed some touch ups on vector and your informative article helped me a lot. Thank you :) (you can remove the following text if you want) --> Just a quick and mostly technically irrelevant note: there are few areas that vector has been written as ctors. not a big deal at all comparing to the length and informativeness of your article. Thank you :)

  - **reply to Pouria**

    Posted by *froggy25* on *02/03/2016 10:34pm*

    ctor = constructor, that's not a typo

    Reply

  Reply

- **Great explanation**

  Posted by *Arpit* on *01/14/2016 12:20pm*

  Thanks a lot for such a descriptive document. Indeed a good read to clear out the specifics regarding vectors.

  Reply