## compare function to be used in sort(),and next\_permutation() in order to sort them in alphabetic order……

int order(char a)

{

if(isupper(a))

{

return (a-'A')\*2;

}

else

{

return (a-'a')\*2+1;

}

}

bool comp(const char &a,const char &b)

{

return order(a) < order(b);

}

##...........##

##.................##......................##.....................##

**Divisor Count through Prime Factorization**

We won’t be able to optimize our current algorithm more, so if we want a faster solution, we are going to have to think of a better way than checking each number.

Once again, let’s assume n = 24. We all know how to find the prime factors (aka. the prime factorization) of a number (and if not, take a look at [Wikipedia](http://en.wikipedia.org/wiki/Prime_factor)), and since we’re trying to find some patterns, we might as well factorize n. The prime factorization of 24is \displaystyle 24 = 2^3 \times 3^1. Now that we’ve factorized n, we might as well factorize it’s divisors (smiley-face).

\displaystyle 1 = 2^0 \times 3^0

\displaystyle 2 = 2^1 \times 3^0

\displaystyle 4 = 2^2 \times 3^0

\displaystyle 8 = 2^3 \times 3^0

\displaystyle 3 = 2^0 \times 3^1

\displaystyle 6 = 2^1 \times 3^1

\displaystyle 12 = 2^2 \times 3^1

\displaystyle 24 = 2^3 \times 3^1

Now we notice that the divisors have very similar prime factorizations to n. More precisely, they just have different combinations of the powers of the primes. The powers of range from 0to 3(same as the power in n), and the powers of 3range from 0to 1(same as the power in n). Now if we were to count the divisors of 24, we would use combinatorics and get (3   1) \times (1   1) = 8. Or more generally for any number n = p_0^{a_0} \times p_1^{a_1} \times \cdots \times p_n^{a_n}, where p_iis the i-th prime factor and a_iis the power of that prime factor, it’s divisor count is (a_0   1) \times (a_1   1) \times \cdots \times (a_n   1).

Our new algorithm finds the prime factorization of the number, and then computes the divisor count according to our new formula.  
Note that I use an optimized trial division, similar to what we did above, to find the prime factorization, but we could also have used a list of primes.

int count\_d(int a)

{

int count,pw,i;

// a counter for the number of divisors

    // intially 1 (the multiplication identity)

count=1;

// loop through 2 and the odd numbers up to sqrt(n)

for(i=2;i<=sqrt(a\*1.0);i=(i==2?3:i+2))

{

 // a counter for the power of the

        // current number in the prime factorization

pw=0;

  // while i is in n's prime factorization

while(a%i == 0)

{

pw++; // increment the power count

a/=i; // remove one i from the prime factorization of n

}

// change the divisor count according to our formula

count\*= (pw+1);

}

// if we've still not removed all factors from n,

    // then there is one prime factor left

if(a != 1)

{

count \*= 2;

}

return count;

}

##………………….##....................................##..............................##...............................##

// Funcion to sum the factors.

int sumf(int i)

{

int sum=1,j;

if(i==1)

return sum;

for(j=2;j<sqrt(i\*1.0);j++)

{

if(i%j == 0)

{

sum+=j;

sum+=(i/j);

}

}

if(i%j == 0 && i/j == j)

{

sum+=j;

}

return sum+i;

}

## check whether a number is power of 2 by bit shifting ##

bool is\_P2(int a)

{

int bit=0;

while(bit < 1)

{

if((a & 1))

{

bit++;

}

a >>= 1;

}

return (a==0);

}

## how can we tell if a number has an even or odd number of factors? For every factor *n* has below \sqrt{n}it has one above \sqrt{n}. This suggests that unless *n* is a perfect square, it has an even number of factors.

## fzort: This problem also has a O(1) solution. It's easy to compute the index *d* of the diagonal a given number *n* is in. The number of elements before diagonal *d* is

n={\frac  {d(d+1)}{2}}

, so the number *n* is in diagonal

d=\left\lfloor {\frac  {1+{\sqrt  {1+8n}}}{2}}\right\rfloor 

void sieve()

{

int i,j;

primes[0]=false;

primes[1]=false;

for(i=2;i<=1000;i++)

{

for(j=i\*i;j<=1000000;j+=i)

{

primes[j]=false;

}

}

}

int GCD(int a,int b)

{

if(b==0)

{

return a;

}

else

{

return GCD(b,a%b);

}

}

int count\_divisor(int a)

{

int count,pw,i;

count=1;

for(i=2;i<=sqrt(a\*1.0);i=(i==2?3:i+2))

{

pw=0;

while(a%i == 0)

{

pw++;

a/=i;

}

count\*= (pw+1);

}

if(a != 1)

{

count \*= 2;

}

return count;

}

int bigmod(long long x,int y, int z)

{

long long c;

if(y==0)

return 1;

if(y%2==0)

{

c=bigmod(x,y/2,z);

return ((c%z)\*(c%z))%z;

}

if(y%2==1)

{

return ((x%z)\*(bigmod(x,y-1,z)%z))%z;

}

}

int dfs\_find\_max\_depth\_of\_a\_node(int a)

{

chk[a] = true;

list <pair<int, int> >::iterator it;

int wgt;

for (it = adj[a].bgn; it != adj[a].end; it++)

{

if (!chk[it->first])

{

wgt = it->second + dfs(it->first);

if (wgt >= maxd1[a])

{

maxd2[a] = maxd1[a];

maxd1[a] = wgt;

}

else if (wgt > maxd2[a])

{

maxd2[a] = wgt;

}

}

}

mx = max(maxd1[a] + maxd2[a], mx);

return maxd1[a];

}

int dfs\_reverse\_max\_depth(int a)

{

chk[a] = true;

list <pair<int, int> >::iterator it;

int wgt;

for (it = adj[a].bgn; it != adj[a].end; it++)

{ if (!chk[it->first])

{

wgt = it->second + dfs(it->first);

if (wgt >= maxd1[a])

{

maxd2[a] = maxd1[a];

maxd1[a] = wgt;

}

else if (wgt > maxd2[a])

{

maxd2[a] = wgt;

}

}

}

return maxd1[a];

}

**int check\_palindrom\_string(string str)**

**{**

**int val=3;**

**for(int i=0,j=str.size()-1;i<=j;i++,j--)**

**{**

**if(str[i]!=str[j])**

**{**

**val&=1;**

**}**

**if(str[i]!=M[str[j]])**

**{**

**val&=2;**

**}**

**}**

**return val;**

**}**

**Val==1 mirror**

**Val==2 regular palindrom**

**Val==3 mirror palindrom**

**Val==0 not palindrom;**

**C++  priority queues**

A priority queue is an abstract data type that captures the idea of a container whose elements have "priorities" attached to them.  An element of highest priority always appears at the front of the queue.   If that element  is removed, the next  highest priority  element  advances to the front.

The C++ standard library defines a class template **priority\_queue**, with the following operations:

* push:  Insert an element into the prioity queue.
* top: Return (without removing it) a highest priority element from the priority queue.
* pop: Remove a highest priority element from the priority queue.
* size: Return the number of elements in the priority queue.
* empty: Return true or false according to whether the priority queue is empty or not.

The following code snippet shows how to construct two priority queues, one that can contain integers and another one that can contain character strings:

#include <queue>  
  
priority\_queue<int> q1;  
priority\_queue<string> q2;  
  
The following is an example of priority queue usage:  
  
#include <string>  
#include <queue>  
#include <iostream>  
  
using namespace std;  // This is to make available the names of things defined in the standard library.  
  
int main()  
{  
    piority\_queue<string> pq; // Creates a priority queue pq to store strings, and initializes the queue to be empty.  
  
    pq.push("the quick");  
    pq.push("fox");  
    pq.push("jumped over");  
    pq.push("the lazy dog");  
  
    // The strings are ordered inside the priority queue in lexicographic (dictionary) order:  
    // "fox", "jumped over", "the lazy dog", "the quick"  
    //  The lowest priority string is "fox", and the highest priority string is "the quick"  
  
    while (!pq.empty()) {  
       cout << pq.front() << endl;  // Print highest priority string  
       pq.pop();                    // Remmove highest priority string  
    }  
  
    return 0;  
}  
  
  
The output of this program is:  
  
  
  
the quick  
the lazy dog  
jumped over  
fox  
  
Since a queue follows a priority discipline, the strings are printed from highest to lowest priority.  
  
Sometimes one needs to create a priority queue to contain user defined objects.  In this case, the priority queue needs to know the comparison criterion used to determine which objects have the highest priority.  This is done by means of a *function object* beloging to a class that overloads the operator ().  The overloaded () acts as < for the purpose of determining priorities.  For example, suppose we want to create a priority queue to store Time objects.  A Time object has three fields: hours, minutes, seconds:  
  
struct Time {  
    int h;   
    int m;   
    int s;  
};  
  
class CompareTime {  
    public:  
    bool operator()(Time& t1, Time& t2) // Returns true if t1 is earlier than t2  
    {  
       if (t1.h < t2.h) return true;  
       if (t1.h == t2.h && t1.m < t2.m) return true;  
       if (t1.h == t2.h && t1.m == t2.m && t1.s < t2.s) return true;  
       return false;  
    }  
}  
  
A priority queue to store times acoording the the above comparison criterion would be defined as follows:  
  
priority\_queue<Time, vector<Time>, CompareTime> pq;  
  
Here is a complete program:  
  
#include <iostream>  
#include <queue>  
#include <iomanip>  
  
using namespace std;  
  
struct Time {  
    int h; // >= 0  
    int m; // 0-59  
    int s; // 0-59  
};  
  
class CompareTime {  
public:  
    bool operator()(Time& t1, Time& t2)  
    {  
       if (t1.h < t2.h) return true;  
       if (t1.h == t2.h && t1.m < t2.m) return true;  
       if (t1.h == t2.h && t1.m == t2.m && t1.s < t2.s) return true;  
       return false;  
    }  
};  
  
int main()  
{  
    priority\_queue<Time, vector<Time>, CompareTime> pq;  
  
    // Array of 4 time objects:  
  
    Time t[4] = { {3, 2, 40}, {3, 2, 26}, {5, 16, 13}, {5, 14, 20}};  
   
    for (int i = 0; i < 4; ++i)  
       pq.push(t[i]);  
  
    while (! pq.empty()) {  
       Time t2 = pq.top();  
       cout << setw(3) << t2.h << " " << setw(3) << t2.m << " " <<  
       setw(3) << t2.s << endl;  
       pq.pop();  
    }  
  
    return 0;  
}  
  
The program prints the times from latest to earliest:  
      
    5  16  13

    5  14  20  
    3   2  40  
    3   2  26  
  
If we wanted earliest times to have the highest priority, we would redefine CompareTime like this:  
  
class CompareTime {  
public:  
    bool operator()(Time& t1, Time& t2) // t2 has highest prio than t1 if t2 is earlier than t1  
    {  
       if (t2.h < t1.h) return true;  
       if (t2.h == t1.h && t2.m < t1.m) return true;  
       if (t2.h == t1.h && t2.m == t1.m && t2.s < t1.s) return true;  
       return false;  
    }  
};

**###Sorting**

Sorting in general refers to various methods of arranging or ordering things based on criterias (numerical, chronological, alphabetical, heirarchial etc.). In Computer Science, due to obvious reasons, Sorting (of data) is of immense importance and is one of the most extensively researched subjects. It is one of the most fundamental algorithmic problems. So much so that it is also fundmental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc. It is estimated that around 25% of all CPU cycles are used to sort data. There are many approaches to sorting data and each has its own merits and demerits. This article discusses some of the common sorting algorithms.

**Bubble Sort**  
Bubble Sort is probably one of the oldest, most easiest, straight-forward, inefficient sorting algorithms. It is the algorithm introduced as a sorting routine in most introductory courses on Algorithms. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list. An implementation in C.

void BubbleSort(int a[], int array\_size)  
{  
 int i, j, temp;  
 for (i = 0; i < (array\_size - 1); ++i)  
 {  
 for (j = 0; j < array\_size - 1 - i; ++j )  
 {  
 if (a[j] > a[j+1])  
 {  
 temp = a[j+1];  
 a[j+1] = a[j];  
 a[j] = temp;  
 }  
 }  
 }  
}

A single, complete "bubble step" is the step in which a maximum element is bubbled to its correct position. This is handled by the inner for loop.

for (j = 0; j < array\_size - 1 - i; ++j )  
{  
 if (a[j] > a[j+1])  
 {  
 temp = a[j+1];  
 a[j+1] = a[j];  
 a[j] = temp;  
 }  
}

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4 } pass 0  
6 8 3 1 2 5 4 10 } pass 1  
6 3 1 2 5 4 8 10 } pass 2  
3 1 2 5 4 6 8 10 } pass 3  
1 2 3 4 5 6 8 10 } pass 4  
1 2 3 4 5 6 8 10 } pass 5  
1 2 3 4 5 6 8 10 } pass 6  
1 2 3 4 5 6 8 10 } pass 7

The above tabulated clearly depicts how each bubble sort works. Note that each pass results in one number being bubbled to the end of the list.

**Selection Sort**  
The idea of Selection Sort is rather simple. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where its supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-1th index of the list. The below is an implementation of the algorithm in C.

void SelectionSort(int a[], int array\_size)  
{  
 int i;  
 for (i = 0; i < array\_size - 1; ++i)  
 {  
 int j, min, temp;  
 min = i;  
 for (j = i+1; j < array\_size; ++j)  
 {  
 if (a[j] < a[min])  
 min = j;  
 }  
  
 temp = a[i];  
 a[i] = a[min];  
 a[min] = temp;  
 }  
}

Consider the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4 } pass 0  
1 6 10 3 8 2 5 4 } pass 1  
1 2 10 3 8 6 5 4 } pass 2  
1 2 3 10 8 6 5 4 } pass 3  
1 2 3 4 8 6 5 10 } pass 4  
1 2 3 4 5 6 8 10 } pass 5  
1 2 3 4 5 6 8 10 } pass 6  
1 2 3 4 5 6 8 10 } pass 7

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).

**Insertion Sort**  
The Insertion Sort algorithm is a commonly used algorithm. Even if you haven't been a programmer or a student of computer science, you may have used this algorithm. Try recalling how you sort a deck of cards. You start from the begining, traverse through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following is an implementation in C.

void insertionSort(int a[], int array\_size)  
{  
 int i, j, index;  
 for (i = 1; i < array\_size; ++i)  
 {  
 index = a[i];  
 for (j = i; j > 0 && a[j-1] > index; j--)  
 a[j] = a[j-1];  
  
 a[j] = index;  
 }  
}

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4 } pass 0  
6 8 10 3 1 2 5 4 } pass 1  
6 8 10 3 1 2 5 4 } pass 2  
3 6 8 10 1 2 5 4 } pass 3  
1 3 6 8 10 2 5 4 } pass 4  
1 2 3 6 8 10 5 4 } pass 5  
1 2 3 5 6 8 10 4 } pass 6  
1 2 3 4 5 6 8 10 } pass 7

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. Now try out the deck-of-cards-sorting algorithm with this list and see if it matches with the tabulated data. For example, you start from 8 and the next card you see is 6. Hence you remove 6 from its current position and "insert" it back to the top. That constitued pass 1. Repeat the same process and you'll do the same thing for 3 which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its < (6,8,10) but > 1. As you carry on till you reach the end of the list you'll find that the list has been sorted. It didn't take a course to tell you how to sort a deck of cards, did it; you prolly figured it out on your own. Amazed at the computer scientist in you ? ;)

**Heap Sort**  
Heap sort algorithm, as the name suggests, is based on the concept of heaps. It begins by constructing a special type of binary tree, called heap, out of the set of data which is to be sorted. Note:

* A Heap by definition is a special type of binary tree in which each node is greater than any of its descendants. It is a complete binary tree.
* A semi-heap is a binary tree in which all the nodes except the root possess the heap property.
* If N be the number of a node, then its left child is 2\*N and the right child 2\*N+1.

The root node of a Heap, by definition, is the maximum of all the elements in the set of data, constituting the binary tree. Hence the sorting process basically consists of extracting the root node and reheaping the remaining set of elements to obtain the next largest element till there are no more elements left to heap. Elemetary implementations usually employ two arrays, one for the heap and the other to store the sorted data. But it is possible to use the same array to heap the unordered list and compile the sorted list. This is usually done by swapping the root of the heap with the end of the array and then excluding that element from any subsequent reheaping.

Significance of a semi-heap - A Semi-Heap as mentioned above is a Heap except that the root does not possess the property of a heap node. This type of a heap is significant in the discussion of Heap Sorting, since after each "Heaping" of the set of data, the root is extracted and replaced by an element from the list. This leaves us with a Semi-Heap. Reheaping a Semi-Heap is particularily easy since all other nodes have already been heaped and only the root node has to be shifted downwards to its right position. The following C function takes care of reheaping a set of data or a part of it.

void downHeap(int a[], int root, int bottom)  
{  
 int maxchild, temp, child;  
 while (root\*2 < bottom)  
 {  
 child = root \* 2 + 1;  
 if (child == bottom)  
 {  
 maxchild = child;  
 }  
 else  
 {  
 if (a[child] > a[child + 1])  
 maxchild = child;  
 else  
 maxchild = child + 1;  
 }  
  
 if (a[root] < a[maxchild])  
 {  
 temp = a[root];  
 a[root] = a[maxchild];  
 a[maxchild] = temp;  
 }  
 else return;  
  
 root = maxchild;  
 }  
}

In the above function, both root and bottom are indices into the array. Note that, theoritically speaking, we generally express the indices of the nodes starting from 1 through size of the array. But in C, we know that array indexing begins at 0; and so the left child is

child = root \* 2 + 1  
/\* so, for eg., if root = 0, child = 1 (not 0) \*/

In the function, what basically happens is that, starting from root each loop performs a check for the heap property of root and does whatever necessary to make it conform to it. If it does already conform to it, the loop breaks and the function returns to caller. Note that the function assumes that the tree constituted by the root and all its descendants is a Semi-Heap.

Now that we have a downheaper, what we need is the actual sorting routine.

void heapsort(int a[], int array\_size)  
{  
 int i;  
 for (i = (array\_size/2 -1); i >= 0; --i)  
 {  
 downHeap(a, i, array\_size-1);  
 }  
  
 for (i = array\_size-1; i >= 0; --i)  
 {  
 int temp;  
 temp = a[i];  
 a[i] = a[0];  
 a[0] = temp;  
 downHeap(a, 0, i-1);  
 }  
}

Note that, before the actual sorting of data takes place, the list is heaped in the for loop starting from the mid element (which is the parent of the right most leaf of the tree) of the list.

for (i = (array\_size/2 -1); i >= 0; --i)  
{  
 downHeap(a, i, array\_size-1);  
}

Following this is the loop which actually performs the extraction of the root and creating the sorted list. Notice the swapping of the ith element with the root followed by a reheaping of the list.

for (i = array\_size-1; i >= 0; --i)  
{  
 int temp;  
 temp = a[i];  
 a[i] = a[0];  
 a[0] = temp;  
 downHeap(a, 0, i-1);  
}

The following are some snapshots of the array during the sorting process. The unodered list -

8 6 10 3 1 2 5 4

After the initial heaping done by the first for loop.

10 6 8 4 1 2 5 3

Second loop which extracts root and reheaps.

8 6 5 4 1 2 3 10 } pass 1  
6 4 5 3 1 2 8 10 } pass 2  
5 4 2 3 1 6 8 10 } pass 3  
4 3 2 1 5 6 8 10 } pass 4  
3 1 2 4 5 6 8 10 } pass 5  
2 1 3 4 5 6 8 10 } pass 6  
1 2 3 4 5 6 8 10 } pass 7  
1 2 3 4 5 6 8 10 } pass 8

Heap sort is one of the preferred sorting algorithms when the number of data items is large. Its efficiency in general is considered to be poorer than quick sort and merge sort.

**###STL**

library

<algorithm>

**<algorithm>**

Standard Template Library: Algorithms

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.  
  
A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the [STL containers](http://www.cplusplus.com/stl). Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

**Functions in <algorithm>**

**Non-modifying sequence operations**:

[**all\_of**](http://www.cplusplus.com/reference/algorithm/all_of/)

Test condition on all elements in range (function template )

[**any\_of**](http://www.cplusplus.com/reference/algorithm/any_of/)

Test if any element in range fulfills condition (function template )

[**none\_of**](http://www.cplusplus.com/reference/algorithm/none_of/)

Test if no elements fulfill condition (function template )

[**for\_each**](http://www.cplusplus.com/reference/algorithm/for_each/)

Apply function to range (function template )

[**find**](http://www.cplusplus.com/reference/algorithm/find/)

Find value in range (function template )

[**find\_if**](http://www.cplusplus.com/reference/algorithm/find_if/)

Find element in range (function template )

[**find\_if\_not**](http://www.cplusplus.com/reference/algorithm/find_if_not/)

Find element in range (negative condition) (function template )

[**find\_end**](http://www.cplusplus.com/reference/algorithm/find_end/)

Find last subsequence in range (function template )

[**find\_first\_of**](http://www.cplusplus.com/reference/algorithm/find_first_of/)

Find element from set in range (function template )

[**adjacent\_find**](http://www.cplusplus.com/reference/algorithm/adjacent_find/)

Find equal adjacent elements in range (function template )

[**count**](http://www.cplusplus.com/reference/algorithm/count/)

Count appearances of value in range (function template )

[**count\_if**](http://www.cplusplus.com/reference/algorithm/count_if/)

Return number of elements in range satisfying condition (function template )

[**mismatch**](http://www.cplusplus.com/reference/algorithm/mismatch/)

Return first position where two ranges differ (function template )

[**equal**](http://www.cplusplus.com/reference/algorithm/equal/)

Test whether the elements in two ranges are equal (function template )

[**is\_permutation**](http://www.cplusplus.com/reference/algorithm/is_permutation/)

Test whether range is permutation of another (function template )

[**search**](http://www.cplusplus.com/reference/algorithm/search/)

Search range for subsequence (function template )

[**search\_n**](http://www.cplusplus.com/reference/algorithm/search_n/)

Search range for elements (function template )

**Modifying sequence operations**:

[**copy**](http://www.cplusplus.com/reference/algorithm/copy/)

Copy range of elements (function template )

[**copy\_n**](http://www.cplusplus.com/reference/algorithm/copy_n/)

Copy elements (function template )

[**copy\_if**](http://www.cplusplus.com/reference/algorithm/copy_if/)

Copy certain elements of range (function template )

[**copy\_backward**](http://www.cplusplus.com/reference/algorithm/copy_backward/)

Copy range of elements backward (function template )

[**move**](http://www.cplusplus.com/reference/algorithm/move/)

Move range of elements (function template )

[**move\_backward**](http://www.cplusplus.com/reference/algorithm/move_backward/)

Move range of elements backward (function template )

[**swap**](http://www.cplusplus.com/reference/algorithm/swap/)

Exchange values of two objects (function template )

[**swap\_ranges**](http://www.cplusplus.com/reference/algorithm/swap_ranges/)

Exchange values of two ranges (function template )

[**iter\_swap**](http://www.cplusplus.com/reference/algorithm/iter_swap/)

Exchange values of objects pointed by two iterators (function template )

[**transform**](http://www.cplusplus.com/reference/algorithm/transform/)

Transform range (function template )

[**replace**](http://www.cplusplus.com/reference/algorithm/replace/)

Replace value in range (function template )

[**replace\_if**](http://www.cplusplus.com/reference/algorithm/replace_if/)

Replace values in range (function template )

[**replace\_copy**](http://www.cplusplus.com/reference/algorithm/replace_copy/)

Copy range replacing value (function template )

[**replace\_copy\_if**](http://www.cplusplus.com/reference/algorithm/replace_copy_if/)

Copy range replacing value (function template )

[**fill**](http://www.cplusplus.com/reference/algorithm/fill/)

Fill range with value (function template )

[**fill\_n**](http://www.cplusplus.com/reference/algorithm/fill_n/)

Fill sequence with value (function template )

[**generate**](http://www.cplusplus.com/reference/algorithm/generate/)

Generate values for range with function (function template )

[**generate\_n**](http://www.cplusplus.com/reference/algorithm/generate_n/)

Generate values for sequence with function (function template )

[**remove**](http://www.cplusplus.com/reference/algorithm/remove/)

Remove value from range (function template )

[**remove\_if**](http://www.cplusplus.com/reference/algorithm/remove_if/)

Remove elements from range (function template )

[**remove\_copy**](http://www.cplusplus.com/reference/algorithm/remove_copy/)

Copy range removing value (function template )

[**remove\_copy\_if**](http://www.cplusplus.com/reference/algorithm/remove_copy_if/)

Copy range removing values (function template )

[**unique**](http://www.cplusplus.com/reference/algorithm/unique/)

Remove consecutive duplicates in range (function template )

[**unique\_copy**](http://www.cplusplus.com/reference/algorithm/unique_copy/)

Copy range removing duplicates (function template )

[**reverse**](http://www.cplusplus.com/reference/algorithm/reverse/)

Reverse range (function template )

[**reverse\_copy**](http://www.cplusplus.com/reference/algorithm/reverse_copy/)

Copy range reversed (function template )

[**rotate**](http://www.cplusplus.com/reference/algorithm/rotate/)

Rotate left the elements in range (function template )

[**rotate\_copy**](http://www.cplusplus.com/reference/algorithm/rotate_copy/)

Copy range rotated left (function template )

[**random\_shuffle**](http://www.cplusplus.com/reference/algorithm/random_shuffle/)

Randomly rearrange elements in range (function template )

[**shuffle**](http://www.cplusplus.com/reference/algorithm/shuffle/)

Randomly rearrange elements in range using generator (function template )

**Partitions**:

[**is\_partitioned**](http://www.cplusplus.com/reference/algorithm/is_partitioned/)

Test whether range is partitioned (function template )

[**partition**](http://www.cplusplus.com/reference/algorithm/partition/)

Partition range in two (function template )

[**stable\_partition**](http://www.cplusplus.com/reference/algorithm/stable_partition/)

Partition range in two - stable ordering (function template )

[**partition\_copy**](http://www.cplusplus.com/reference/algorithm/partition_copy/)

Partition range into two (function template )

[**partition\_point**](http://www.cplusplus.com/reference/algorithm/partition_point/)

Get partition point (function template )

**Sorting**:

[**sort**](http://www.cplusplus.com/reference/algorithm/sort/)

Sort elements in range (function template )

[**stable\_sort**](http://www.cplusplus.com/reference/algorithm/stable_sort/)

Sort elements preserving order of equivalents (function template )

[**partial\_sort**](http://www.cplusplus.com/reference/algorithm/partial_sort/)

Partially sort elements in range (function template )

[**partial\_sort\_copy**](http://www.cplusplus.com/reference/algorithm/partial_sort_copy/)

Copy and partially sort range (function template )

[**is\_sorted**](http://www.cplusplus.com/reference/algorithm/is_sorted/)

Check whether range is sorted (function template )

[**is\_sorted\_until**](http://www.cplusplus.com/reference/algorithm/is_sorted_until/)

Find first unsorted element in range (function template )

[**nth\_element**](http://www.cplusplus.com/reference/algorithm/nth_element/)

Sort element in range (function template )

**Binary search** (operating on partitioned/sorted ranges):

[**lower\_bound**](http://www.cplusplus.com/reference/algorithm/lower_bound/)

Return iterator to lower bound (function template )

[**upper\_bound**](http://www.cplusplus.com/reference/algorithm/upper_bound/)

Return iterator to upper bound (function template )

[**equal\_range**](http://www.cplusplus.com/reference/algorithm/equal_range/)

Get subrange of equal elements (function template )

[**binary\_search**](http://www.cplusplus.com/reference/algorithm/binary_search/)

Test if value exists in sorted sequence (function template )

**Merge** (operating on sorted ranges):

[**merge**](http://www.cplusplus.com/reference/algorithm/merge/)

Merge sorted ranges (function template )

[**inplace\_merge**](http://www.cplusplus.com/reference/algorithm/inplace_merge/)

Merge consecutive sorted ranges (function template )

[**includes**](http://www.cplusplus.com/reference/algorithm/includes/)

Test whether sorted range includes another sorted range (function template )

[**set\_union**](http://www.cplusplus.com/reference/algorithm/set_union/)

Union of two sorted ranges (function template )

[**set\_intersection**](http://www.cplusplus.com/reference/algorithm/set_intersection/)

Intersection of two sorted ranges (function template )

[**set\_difference**](http://www.cplusplus.com/reference/algorithm/set_difference/)

Difference of two sorted ranges (function template )

[**set\_symmetric\_difference**](http://www.cplusplus.com/reference/algorithm/set_symmetric_difference/)

Symmetric difference of two sorted ranges (function template )

**Heap**:

[**push\_heap**](http://www.cplusplus.com/reference/algorithm/push_heap/)

Push element into heap range (function template )

[**pop\_heap**](http://www.cplusplus.com/reference/algorithm/pop_heap/)

Pop element from heap range (function template )

[**make\_heap**](http://www.cplusplus.com/reference/algorithm/make_heap/)

Make heap from range (function template )

[**sort\_heap**](http://www.cplusplus.com/reference/algorithm/sort_heap/)

Sort elements of heap (function template )

[**is\_heap**](http://www.cplusplus.com/reference/algorithm/is_heap/)

Test if range is heap (function template )

[**is\_heap\_until**](http://www.cplusplus.com/reference/algorithm/is_heap_until/)

Find first element not in heap order (function template )

**Min/max**:

[**min**](http://www.cplusplus.com/reference/algorithm/min/)

Return the smallest (function template )

[**max**](http://www.cplusplus.com/reference/algorithm/max/)

Return the largest (function template )

[**minmax**](http://www.cplusplus.com/reference/algorithm/minmax/)

Return smallest and largest elements (function template )

[**min\_element**](http://www.cplusplus.com/reference/algorithm/min_element/)

Return smallest element in range (function template )

[**max\_element**](http://www.cplusplus.com/reference/algorithm/max_element/)

Return largest element in range (function template )

[**minmax\_element**](http://www.cplusplus.com/reference/algorithm/minmax_element/)

Return smallest and largest elements in range (function template )

**Other**:

[**lexicographical\_compare**](http://www.cplusplus.com/reference/algorithm/lexicographical_compare/)

Lexicographical less-than comparison (function template )

[**next\_permutation**](http://www.cplusplus.com/reference/algorithm/next_permutation/)

Transform range to next permutation (function template )

[**prev\_permutation**](http://www.cplusplus.com/reference/algorithm/prev_permutation/)

Transform range to previous permutation (function template )

class template

<string>

**std::basic\_string**

template < class charT,

class traits = char\_traits<charT>, // basic\_string::traits\_type

class Alloc = allocator<charT> // basic\_string::allocator\_type

> class basic\_string;

Generic string class

The basic\_string is the generalization of class [string](http://www.cplusplus.com/string) for any character type (see [string](http://www.cplusplus.com/string) for a description).

**Template parameters**

charT

Character type.  
The string is formed by a sequence of characters of this type.  
This shall be a non-array [POD type](http://www.cplusplus.com/is_pod).

traits

[Character traits](http://www.cplusplus.com/char_traits) class that defines essential properties of the characters used by [basic\_string](http://www.cplusplus.com/basic_string) objects (see [char\_traits](http://www.cplusplus.com/char_traits)).  
traits::char\_type shall be the same as charT.  
Aliased as member type basic\_string::traits\_type.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type basic\_string::allocator\_type.

Note: Because the first template parameter is not aliased as any member type, charT is used throughout this reference to refer to this type.

**Template instantiations**

[**string**](http://www.cplusplus.com/reference/string/string/)

String class (class )

[**wstring**](http://www.cplusplus.com/reference/string/wstring/)

Wide string (class )

[**u16string**](http://www.cplusplus.com/reference/string/u16string/)

String of 16-bit characters (class )

[**u32string**](http://www.cplusplus.com/reference/string/u32string/)

String of 32-bit characters (class )

**Member types**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| traits\_type | The second template parameter (traits) | defaults to: [char\_traits](http://www.cplusplus.com/char_traits)<charT> |
| allocator\_type | The third template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<charT> |
| value\_type | traits\_type::char\_type | shall be the same as charT |
| reference | allocator\_type::reference | for the default [allocator](http://www.cplusplus.com/allocator): charT& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const charT& |
| pointer | allocator\_type::pointer | for the default [allocator](http://www.cplusplus.com/allocator): charT\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const charT\* |
| iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to charT | convertible to const\_iterator |
| const\_iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const charT |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | allocator\_type::difference\_type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | allocator\_type::difference\_type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

**Member functions**

[**(constructor)**](http://www.cplusplus.com/reference/string/basic_string/basic_string/)

Construct basic\_string object (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/string/basic_string/%7Ebasic_string/)

String destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/string/basic_string/operator=/)

String assignment (public member function )

**Iterators**:

[**begin**](http://www.cplusplus.com/reference/string/basic_string/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/string/basic_string/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/string/basic_string/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/string/basic_string/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/string/basic_string/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/string/basic_string/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/string/basic_string/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/string/basic_string/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**size**](http://www.cplusplus.com/reference/string/basic_string/size/)

Return size (public member function )

[**length**](http://www.cplusplus.com/reference/string/basic_string/length/)

Return length of string (public member function )

[**max\_size**](http://www.cplusplus.com/reference/string/basic_string/max_size/)

Return maximum size (public member function )

[**resize**](http://www.cplusplus.com/reference/string/basic_string/resize/)

Resize string (public member function )

[**capacity**](http://www.cplusplus.com/reference/string/basic_string/capacity/)

Return size of allocated storage (public member function )

[**reserve**](http://www.cplusplus.com/reference/string/basic_string/reserve/)

Request a change in capacity (public member function )

[**clear**](http://www.cplusplus.com/reference/string/basic_string/clear/)

Clear string (public member function )

[**empty**](http://www.cplusplus.com/reference/string/basic_string/empty/)

Test whether string is empty (public member function )

[**shrink\_to\_fit**](http://www.cplusplus.com/reference/string/basic_string/shrink_to_fit/)

Shrink to fit (public member function )

**Element access**:

[**operator[]**](http://www.cplusplus.com/reference/string/basic_string/operator%5b%5d/)

Get character of string (public member function )

[**at**](http://www.cplusplus.com/reference/string/basic_string/at/)

Get character of string (public member function )

[**back**](http://www.cplusplus.com/reference/string/basic_string/back/)

Access last character (public member function )

[**front**](http://www.cplusplus.com/reference/string/basic_string/front/)

Access first character (public member function )

**Modifiers**:

[**operator+=**](http://www.cplusplus.com/reference/string/basic_string/operator+=/)

Append to string (public member function )

[**append**](http://www.cplusplus.com/reference/string/basic_string/append/)

Append to string (public member function )

[**push\_back**](http://www.cplusplus.com/reference/string/basic_string/push_back/)

Append character to string (public member function )

[**assign**](http://www.cplusplus.com/reference/string/basic_string/assign/)

Assign content to string (public member function )

[**insert**](http://www.cplusplus.com/reference/string/basic_string/insert/)

Insert into string (public member function )

[**erase**](http://www.cplusplus.com/reference/string/basic_string/erase/)

Erase characters from string (public member function )

[**replace**](http://www.cplusplus.com/reference/string/basic_string/replace/)

Replace portion of string (public member function )

[**swap**](http://www.cplusplus.com/reference/string/basic_string/swap/)

Swap string values (public member function )

[**pop\_back**](http://www.cplusplus.com/reference/string/basic_string/pop_back/)

Delete last character (public member function )

**String operations**:

[**c\_str**](http://www.cplusplus.com/reference/string/basic_string/c_str/)

Get C-string equivalent

[**data**](http://www.cplusplus.com/reference/string/basic_string/data/)

Get string data (public member function )

[**get\_allocator**](http://www.cplusplus.com/reference/string/basic_string/get_allocator/)

Get allocator (public member function )

[**copy**](http://www.cplusplus.com/reference/string/basic_string/copy/)

Copy sequence of characters from string (public member function )

[**find**](http://www.cplusplus.com/reference/string/basic_string/find/)

Find first occurrence in string (public member function )

[**rfind**](http://www.cplusplus.com/reference/string/basic_string/rfind/)

Find last occurrence in string (public member function )

[**find\_first\_of**](http://www.cplusplus.com/reference/string/basic_string/find_first_of/)

Find character in string (public member function )

[**find\_last\_of**](http://www.cplusplus.com/reference/string/basic_string/find_last_of/)

Find character in string from the end (public member function )

[**find\_first\_not\_of**](http://www.cplusplus.com/reference/string/basic_string/find_first_not_of/)

Find non-matching character in string (public member function )

[**find\_last\_not\_of**](http://www.cplusplus.com/reference/string/basic_string/find_last_not_of/)

Find non-matching character in string from the end (public member function )

[**substr**](http://www.cplusplus.com/reference/string/basic_string/substr/)

Generate substring (public member function )

[**compare**](http://www.cplusplus.com/reference/string/basic_string/compare/)

Compare strings (public member function )

**Non-member function overloads**

[**operator+**](http://www.cplusplus.com/reference/string/basic_string/operator+/)

Concatenate strings (function template )

[**relational operators**](http://www.cplusplus.com/reference/string/basic_string/operators/)

Relational operators for basic\_string (function template )

[**swap**](http://www.cplusplus.com/reference/string/basic_string/swap-free/)

Exchanges the values of two strings (function template )

[**operator>>**](http://www.cplusplus.com/reference/string/basic_string/operator%3E%3E/)

Extract string from stream (function template )

[**operator<<**](http://www.cplusplus.com/reference/string/basic_string/operator%3C%3C/)

Insert string into stream (function template )

[**getline**](http://www.cplusplus.com/reference/string/basic_string/getline/)

Get line from stream into string (function template )

**Member constants**

[**npos**](http://www.cplusplus.com/reference/string/basic_string/npos/)

Maximum value of size\_type (public static member constant )

class template

<map>

**std::map**

template < class Key, // map::key\_type

class T, // map::mapped\_type

class Compare = less<Key>, // map::key\_compare

class Alloc = allocator<pair<const Key,T> > // map::allocator\_type

> class map;

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.  
  
In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value\_type, which is a [pair](http://www.cplusplus.com/pair) type combining both:

|  |  |
| --- | --- |
|  | *typedef* pair<*const* Key, T> value\_type; |

Internally, the elements in a map are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/map::key_comp) (of type Compare).  
  
map containers are generally slower than [unordered\_map](http://www.cplusplus.com/unordered_map) containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.  
  
The mapped values in a [map](http://www.cplusplus.com/map) can be accessed directly by their corresponding key using the *bracket operator* (([operator[]](http://www.cplusplus.com/map::operator%5b%5d)).  
  
Maps are typically implemented as *binary search trees*.

**Container properties**

Associative

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

Map

Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

Unique keys

No two elements in the container can have equivalent *keys*.

Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

**Template parameters**

Key

Type of the *keys*. Each element in a map is uniquely identified by its key value.  
Aliased as member type map::key\_type.

T

Type of the mapped value. Each element in a map stores some data as its mapped value.  
Aliased as member type map::mapped\_type.

Compare

A binary predicate that takes two element keys as arguments and returns a bool. The expression comp(a,b), where *comp* is an object of this type and *a* and *b* are key values, shall return true if *a* is considered to go before *b* in the *strict weak ordering* the function defines.  
The map object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if !comp(a,b) && !comp(b,a)). No two elements in a map container can have equivalent keys.  
This can be a function pointer or a function object (see [constructor](http://www.cplusplus.com/map::map) for an example). This defaults to [less](http://www.cplusplus.com/less)<T>, which returns the same as applying the *less-than operator* (a<b).  
Aliased as member type map::key\_compare.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type map::allocator\_type.

**Member types**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| key\_type | The first template parameter (Key) |  |
| mapped\_type | The second template parameter (T) |  |
| value\_type | [pair](http://www.cplusplus.com/pair)<const key\_type,mapped\_type> |  |
| key\_compare | The third template parameter (Compare) | defaults to: [less](http://www.cplusplus.com/less)<key\_type> |
| value\_compare | *Nested function class to compare elements* | see [value\_comp](http://www.cplusplus.com/map::value_comp) |
| allocator\_type | The fourth template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<value\_type> |
| reference | allocator\_type::reference | for the default [allocator](http://www.cplusplus.com/allocator): value\_type& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type& |
| pointer | allocator\_type::pointer | for the default [allocator](http://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to: iterator\_traits<iterator>::difference\_type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

**Member functions**

[**(constructor)**](http://www.cplusplus.com/reference/map/map/map/)

Construct map (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/map/map/%7Emap/)

Map destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/map/map/operator=/)

Copy container content (public member function )

**Iterators**:

[**begin**](http://www.cplusplus.com/reference/map/map/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/map/map/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/map/map/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/map/map/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/map/map/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/map/map/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/map/map/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/map/map/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**empty**](http://www.cplusplus.com/reference/map/map/empty/)

Test whether container is empty (public member function )

[**size**](http://www.cplusplus.com/reference/map/map/size/)

Return container size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/map/map/max_size/)

Return maximum size (public member function )

**Element access**:

[**operator[]**](http://www.cplusplus.com/reference/map/map/operator%5b%5d/)

Access element (public member function )

[**at**](http://www.cplusplus.com/reference/map/map/at/)

Access element (public member function )

**Modifiers**:

[**insert**](http://www.cplusplus.com/reference/map/map/insert/)

Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/map/map/erase/)

Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/map/map/swap/)

Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/map/map/clear/)

Clear content (public member function )

[**emplace**](http://www.cplusplus.com/reference/map/map/emplace/)

Construct and insert element (public member function )

[**emplace\_hint**](http://www.cplusplus.com/reference/map/map/emplace_hint/)

Construct and insert element with hint (public member function )

**Observers**:

[**key\_comp**](http://www.cplusplus.com/reference/map/map/key_comp/)

Return key comparison object (public member function )

[**value\_comp**](http://www.cplusplus.com/reference/map/map/value_comp/)

Return value comparison object (public member function )

**Operations**:

[**find**](http://www.cplusplus.com/reference/map/map/find/)

Get iterator to element (public member function )

[**count**](http://www.cplusplus.com/reference/map/map/count/)

Count elements with a specific key (public member function )

[**lower\_bound**](http://www.cplusplus.com/reference/map/map/lower_bound/)

Return iterator to lower bound (public member function )

[**upper\_bound**](http://www.cplusplus.com/reference/map/map/upper_bound/)

Return iterator to upper bound (public member function )

[**equal\_range**](http://www.cplusplus.com/reference/map/map/equal_range/)

Get range of equal elements (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/map/map/get_allocator/)

Get allocator (public member function )

class template

<vector>

**std::vector**

template < class T, class Alloc = allocator<T> > class vector; // generic template

Vector

Vectors are sequence containers representing arrays that can change in size.  
  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.  
  
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](http://www.cplusplus.com/vector::capacity) greater than the storage strictly needed to contain its elements (i.e., its [size](http://www.cplusplus.com/vector::size)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](http://www.cplusplus.com/vector::size) so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see [push\_back](http://www.cplusplus.com/vector::push_back)).  
  
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.  
  
Compared to the other dynamic sequence containers ([deques](http://www.cplusplus.com/deque), [lists](http://www.cplusplus.com/list) and [forward\_lists](http://www.cplusplus.com/forward_list)), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its [end](http://www.cplusplus.com/vector::end). For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than [lists](http://www.cplusplus.com/list) and [forward\_lists](http://www.cplusplus.com/forward_list).

**Container properties**

Sequence

Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array

Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

**Template parameters**

T

Type of the elements.  
Only if T [is guaranteed to not throw while moving](http://www.cplusplus.com/is_nothrow_move_constructible), implementations can optimize to move elements instead of copying them during reallocations.  
Aliased as member type vector::value\_type.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type vector::allocator\_type.

**Member types**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| value\_type | The first template parameter (T) |  |
| allocator\_type | The second template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<value\_type> |
| reference | allocator\_type::reference | for the default [allocator](http://www.cplusplus.com/allocator): value\_type& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type& |
| pointer | allocator\_type::pointer | for the default [allocator](http://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to: iterator\_traits<iterator>::difference\_type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

**Member functions**

[**(constructor)**](http://www.cplusplus.com/reference/vector/vector/vector/)

Construct vector (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/vector/vector/%7Evector/)

Vector destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/vector/vector/operator=/)

Assign content (public member function )

**Iterators**:

[**begin**](http://www.cplusplus.com/reference/vector/vector/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/vector/vector/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/vector/vector/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/vector/vector/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/vector/vector/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/vector/vector/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/vector/vector/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/vector/vector/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**size**](http://www.cplusplus.com/reference/vector/vector/size/)

Return size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/vector/vector/max_size/)

Return maximum size (public member function )

[**resize**](http://www.cplusplus.com/reference/vector/vector/resize/)

Change size (public member function )

[**capacity**](http://www.cplusplus.com/reference/vector/vector/capacity/)

Return size of allocated storage capacity (public member function )

[**empty**](http://www.cplusplus.com/reference/vector/vector/empty/)

Test whether vector is empty (public member function )

[**reserve**](http://www.cplusplus.com/reference/vector/vector/reserve/)

Request a change in capacity (public member function )

[**shrink\_to\_fit**](http://www.cplusplus.com/reference/vector/vector/shrink_to_fit/)

Shrink to fit (public member function )

**Element access**:

[**operator[]**](http://www.cplusplus.com/reference/vector/vector/operator%5b%5d/)

Access element (public member function )

[**at**](http://www.cplusplus.com/reference/vector/vector/at/)

Access element (public member function )

[**front**](http://www.cplusplus.com/reference/vector/vector/front/)

Access first element (public member function )

[**back**](http://www.cplusplus.com/reference/vector/vector/back/)

Access last element (public member function )

[**data**](http://www.cplusplus.com/reference/vector/vector/data/)

Access data (public member function )

**Modifiers**:

[**assign**](http://www.cplusplus.com/reference/vector/vector/assign/)

Assign vector content (public member function )

[**push\_back**](http://www.cplusplus.com/reference/vector/vector/push_back/)

Add element at the end (public member function )

[**pop\_back**](http://www.cplusplus.com/reference/vector/vector/pop_back/)

Delete last element (public member function )

[**insert**](http://www.cplusplus.com/reference/vector/vector/insert/)

Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/vector/vector/erase/)

Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/vector/vector/swap/)

Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/vector/vector/clear/)

Clear content (public member function )

[**emplace**](http://www.cplusplus.com/reference/vector/vector/emplace/)

Construct and insert element (public member function )

[**emplace\_back**](http://www.cplusplus.com/reference/vector/vector/emplace_back/)

Construct and insert element at the end (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/vector/vector/get_allocator/)

Get allocator (public member function )

**Non-member function overloads**

[**relational operators**](http://www.cplusplus.com/reference/vector/vector/operators/)

Relational operators for vector (function template )

[**swap**](http://www.cplusplus.com/reference/vector/vector/swap-free/)

Exchange contents of vectors (function template )