**How to get element by index in List**

## std::list doesn’t have operator []

std::list does not have random access operator [] because std::list internally store elements in doubly linked list. So, to access an element at nth location we need to iterate one by one from beginning to nth element. So, its complexity will be O(n).

Suppose we have a list of strings i.e.

|  |  |
| --- | --- |
| 1  2  3  4 | std::list<std::string> listOfStrs = { "First", "Sec",  "Third", "Fourth",  "Fifth", "Sixth"  }; |

Now lets see how to access element at 3rd position from list using 2 different techniques.

## Accessing nth element in std::list using std::advance

STL provides and algorithm std::advance() i.e.

|  |  |
| --- | --- |
| 1  2 | template <class InputIterator, class Distance>  void advance (InputIterator& it, Distance n); |

It advances the given iterator by n positions. Now lets access element at 3rd position using std::advance.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Create iterator pointing to first element  std::list<std::string>::iterator it = listOfStrs.begin();    // Advance the iterator by 2 positions,  std::advance(it, 2); |

As iterator **it** was already pointing to first element, therefore we need to advance it by 2 to point it to 3rd position.

Complexity of std::advance for std::list is O(n) because it needs to iterate one by one from beginning to nth position.

In C++11 another algorithm has been introduced i.e std::next()

Accessing nth element in std::list using std::next()

|  |  |
| --- | --- |
| 1  2  3 | template <class ForwardIterator>  ForwardIterator next (ForwardIterator it,         typename iterator\_traits<ForwardIterator>::difference\_type n = 1); |

std::next() is introduced in C++11. It accepts an iterator and position to be advanced. It does not modify the passed iterator, but uses it to create a new iterator pointing to nth  -1 position from given iterator and returns it.

Lets see how to access element at 3rd position by std::next()

|  |  |
| --- | --- |
| 1 | auto it1 = std::next(listOfStrs.begin(), 2); |

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #include <iostream>  #include <list>  #include <string>  #include <iterator>  #include <algorithm>    int main()  {  std::list<std::string> listOfStrs =  { "First", "Sec", "Third", "Fourth", "Fifth", "Sixth" };    /\*\*\*\* Finding nth element using std::advance() \*\*\*\*\*\*/    // Find 3rd element from list    // Create iterator pointing to first element  std::list<std::string>::iterator it = listOfStrs.begin();    // Advance the iterator by 2 positions,  std::advance(it, 2);    // Now iterator it points to 3rd element  std::cout << "3rd element = " << \*it << std::endl;    /\*\*\*\* Finding nth element using std::next() \*\*\*\*\*\*/    // Find 3rd element from list    // It returns a new iterator pointing to n position after the  // base iterator given as first argument  auto it1 = std::next(listOfStrs.begin(), 2);    std::cout << "3rd element = " << \*it1 << std::endl;    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2 | 3rd element = Third  3rd element = Third |

As the above example uses std::next(), which is a c++11 feature. So, use following command to compile the example in linux i.e.

***g++ –std=c++11 example.cpp***

## Complexity of std::next() & std::advance() for std::list()

As std::list is internally implemented as doubly linked list. So, in linked list to access the nth element we need to go one by one from start to nth element.

Therefore, for both the APIs i.e. std::next() & std::advance() complexity to fetch nth element will be O(n).

# How to insert element in vector at specific position | vector::insert() examples

Vector provides different overloaded version of member function insert() , to insert one or more elements in between existing elements.  
Let’s discuss them in detail,

## Inserting a single element at specific position in vector

We are going to use first overloaded version of Vector’s insert() function i.e.

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator pos, const value\_type& val); |

It Inserts a copy of give element **“val”**, before the iterator position “**pos**” and also returns the iterator pointing to new inserted element.

Let’s understand by example,

Suppose we have a vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfNums { 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Now we want to insert an element at index position 4th (In vector position index start from 0),

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Create Iterator pointing to 4th Position  auto itPos = vecOfNums.begin() + 4;    // Insert element with value 9 at 4th Position in vector  auto newIt = vecOfNums.insert(itPos, 9); |

Vector’s contents will be now,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 22 , 9 , 33 , 2 , 11 , 89 , 49 |

Inserting an element in vector will increase the vector size by 1.  
As in vector all elements are stored at continuous memory locations, so inserting an element in between will cause all the elements in right to shift or complete reallocation of all elements.

## Inserting multiple elements or a range at specific position in vector

Some times we encounter a situation where we want to insert multiple elements in vector at specific position. These multiple elements can from another vector , array or any other container.

For this, vector provides an overloaded version of insert() function to insert multiple elements i.e.

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator pos, InputIterator first, InputIterator last); |

It inserts the elements in range from **[first, end)** before iterator position**pos** and returns the iterator pointing to position first newly added element.

Let’s understand by an example,

Suppose we have 2 vectors of strings i.e.

|  |  |
| --- | --- |
| 1  2 | std::vector<std::string> vec1 { "at" , "hello", "hi", "there", "where", "now", "is", "that" };  std::vector<std::string> vec2 { "one" , "two", "two" }; |

Now insert all the elements in vec2 at position 3 in vec1 i.e.

|  |  |
| --- | --- |
| 1  2 | // Insert all the elements in vec2 at 3rd position in vec1  auto iter = vec1.insert(vec1.begin() + 3, vec2.begin(), vec2.end()); |

Contents of vec1 will be now,

|  |  |
| --- | --- |
| 1 | at , hello , hi , one , two , two , there , where , now , is , that , |

## Inserting multiple elements using Initializer list

Another overloaded version of vector’s insert() function is as follows,

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator position, initializer\_list<value\_type> list); |

It copies all the elements in given initializer list before given iterator position pos and also returns the iterator of first of the newly added elements.

Suppose we have vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfInts { 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Let’s add all elements in initialisation list to the existing vector i.e.

|  |  |
| --- | --- |
| 1  2 | // Insert all elements from initialization\_list to vector at 3rd position  auto iter2 = vecOfInts.insert(vecOfInts.begin() + 3, {34,55,66,77}); |

Contents of vecOfInts will be now,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 34 , 55 , 66 , 77 , 22 , 33 , 2 , 11 , 89 , 49 |

## vector.insert() and Iterator invalidation

Inserting elements in vector will cause existing elements to shift places or sometimes complete reallocation, which will invalidates all the existing iterators.

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | #include <iterator>  #include <iostream>  #include <vector>  #include <algorithm>  #include <string>    template <typename T>  void print(T & vecOfElements, std::string delimeter = " , ")  {  for(auto elem : vecOfElements)  std::cout<<elem<<delimeter;  std::cout << std::endl;  }  int main()  {    std::vector<int> vecOfNums { 1, 4, 5, 22, 33, 2, 11, 89, 49 };    /\*  \* Inserting an element at specific position in vector  \*/    // Create Iterator pointing to 4th Position  auto itPos = vecOfNums.begin() + 4;    // Insert element with value 9 at 4th Position in vector  auto newIt = vecOfNums.insert(itPos, 9);    std::cout << "Element added in vector is : " << \*newIt << std::endl;  std::cout << "Modified vecOfNums = ";  print(vecOfNums);    /\*  \* Inserting multiple elements / range at specific position in vector  \*/    std::vector<std::string> vec1 { "at" , "hello", "hi", "there", "where", "now", "is", "that" };  std::vector<std::string> vec2 { "one" , "two", "two" };    // Insert all the elements in vec2 at 3rd position in vec1  auto iter = vec1.insert(vec1.begin() + 3, vec2.begin(), vec2.end());    std::cout << "First of the newly added elements : " << \*iter << std::endl;  std::cout << "Modified vec1 = ";  print(vec1);    /\*  \* Inserting all elements in initialization\_list in another vector  \* at specific position.  \*/    std::vector<int> vecOfInts { 1, 4, 5, 22, 33, 2, 11, 89, 49 };    // Insert all elements from initialization\_list to vector at 3rd position  auto iter2 = vecOfInts.insert(vecOfInts.begin() + 3, {34,55,66,77});    std::cout << "First of the newly added elements : " << \*iter2 << std::endl;  std::cout << "Modified vecOfInts = ";  print(vecOfInts);    return 0;    } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Element added in vector is : 9  Modified vecOfNums = 1 , 4 , 5 , 22 , 9 , 33 , 2 , 11 , 89 , 49 ,  First of the newly added elements : one  Modified vec1 = at , hello , hi , one , two , two , there , where , now , is , that ,  First of the newly added elements : 34  Modified vecOfInts = 1 , 4 , 5 , 34 , 55 , 66 , 77 , 22 , 33 , 2 , 11 , 89 , 49 , |

To Compile the above example in linux use following command,

***g++ –std=c++11 example.cpp***

# How to get element by index in vector | at() vs operator []

In this article we will discuss different techniques to get an element from vector by index or position.

In vector elements are indexed from 0 to size() – 1. To access any element in vector by index vector provides two member functions i.e.

* at()
* operator[]

Let’s discuss them in detail,

## Access an element in vector using operator []

std::vector provides [] operator i.e.

|  |  |
| --- | --- |
| 1 | element\_reference operator[] (size\_type n); |

It returns the reference of element in vector at index n.

Suppose we have a vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfNums{ 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Let’s access element at index 3 using operator [] i.e.

|  |  |
| --- | --- |
| 1  2 | // Access element at index 3  int & element = vecOfNums[3]; |
| 1  2 | // Access element at index 3  int & element = vecOfNums[3]; |

### Access and update element in vector using []

As, operator [] returns a reference to the element in vector, so we can change the content of vector too using operator [] i.e.

|  |  |
| --- | --- |
| 1  2 | // Access and change the value of element at index 3  vecOfNums[3] = 10; |

It will update the value of element at index 3. New contents of vector will be,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 10 , 33 , 2 , 11 , 89 , 49 |

We can also keep the returned reference in a reference variable and use later to modify the vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Get the reference of element at index 3  int & elemRef = vecOfNums[3];    // Modifying the vector using reference to element at index 3  elemRef = 22; |

New contents of vector will be,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 |

### Accessing out of index element through operator []

While accessing any element through operator [] we need to make sure that given index is in range i.e. less than the size of vector, otherwise it will result in undefined behaviour and can also crash application.

Therefore we should always check the size before accessing element using operator [] i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Accessing out of range element using [] results in undefined behaviour  int index = 100;  if (index < vecOfNums.size())  {  element = vecOfNums[index];  }  else  std::cout << "index out of bound" << std::endl; |

## Access an element in vector using vector::at()

std::vector provides an another member function at() i.e.

|  |  |
| --- | --- |
| 1 | reference at(size\_type n); |

It returns the reference of element at index n in vector. If index n is out of range i.e. greater then size of vector then it will throw out\_of\_range exception.

Let’s access element at index 3 using at() i.e.

|  |  |
| --- | --- |
| 1  2 | // Access element at index 3 using at()  int & numRef = vecOfNums.at(3); |

As at() returns a reference, so we can modify the value of element too i.e.

|  |  |
| --- | --- |
| 1  2 | // Modifying the element in vector using reference  numRef = 96; |

### Accessing out of range element using at()

vector::at() will throw out\_of\_range exception in case we try to access the out of range element i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Accessing out of range element using at() will throw out\_of\_range exception  try  {  int index = 100;  element = vecOfNums.at(index);  }  catch (const std::out\_of\_range & ex)  {  std::cout << "out\_of\_range Exception Caught :: " << ex.what() << std::endl;  } |

**Output:**

|  |  |
| --- | --- |
| 1 | out\_of\_range Exception Caught :: vector::\_M\_range\_check: \_\_n (which is 100) >= this->size() (which is 9) |

## vector::operator[] vs vector::at()

Both operator[] & at() provides random access to elements in vector in O(1) Complexity. But in case of out of range access operator[] causes undefined behaviour, whereas at() returns proper out\_of\_range exception. So, at() is more safe to use as compared to operator[].

**Complete example is as follows,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | #include <iostream>  #include <vector>  #include <algorithm>  #include <functional>  #include <string>    template <typename T>  void print(T & vecOfElements, std::string delimeter = " , ")  {  for(auto elem : vecOfElements)  std::cout<<elem<<delimeter;  std::cout << std::endl;  }  int main()  {  std::vector<int> vecOfNums{ 1, 4, 5, 22, 33, 2, 11, 89, 49 };    /\*  Accessing element by index in vector using operator[]  \*/    // Access element at index 3  int & element = vecOfNums[3];    std::cout << "Original vecOfNums = ";  print(vecOfNums);    std::cout << "Element at index 3 is : " << element << std::endl;    // Access and change the value of element at index 3  vecOfNums[3] = 10;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Get the reference of element at index 3  int & elemRef = vecOfNums[3];    // Modifying the vector using reference to element at index 3  elemRef = 22;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Accessing out of range element using [] results in undefined behaviour  int index = 100;  if (index < vecOfNums.size())  {  element = vecOfNums[index];  }  else  std::cout << "index out of bound" << std::endl;    /\*  Accessing element by index in vector using at()  \*/    // Access element at index 3 using at()  int & numRef = vecOfNums.at(3);    std::cout << "Element at index 3 is : " << numRef << std::endl;    // Modifying the element in vector using reference  numRef = 96;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Accessing out of range element using at() will throw out\_of\_range exception  try  {  int index = 100;  element = vecOfNums.at(index);  }  catch (const std::out\_of\_range & ex)  {  std::cout << "out\_of\_range Exception Caught :: " << ex.what() << std::endl;  }      return 0;    } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Original vecOfNums = 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 ,  Element at index 3 is : 22  Modified vecOfNums = 1 , 4 , 5 , 10 , 33 , 2 , 11 , 89 , 49 ,  Modified vecOfNums = 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 ,  index out of bound  Element at index 3 is : 22  Modified vecOfNums = 1 , 4 , 5 , 96 , 33 , 2 , 11 , 89 , 49 ,  out\_of\_range Exception Caught :: vector::\_M\_range\_check: \_\_n (which is 100) >= this->size() (which is 9) |

To compile the above code use following command,

***g++ –std=c++11 example.cpp***

# Different ways to insert elements in Set

Set provides 3 different overloaded version of insert() function to add element in set. Let’s discuss them one by one,

## Inserting a Single element in Set and checking the result

Set is an associative container and contains only unique elements, therefore an insertion in set can fail if  
a similar element is already present in set. Keeping that in mind set provides a version of member function  
insert() i.e.

|  |  |
| --- | --- |
| 1 | pair<iterator,bool> insert (const value\_type& val); |

It accepts the element to be inserted and returns a pair of Iterator & bool flag.

If insertion is successful then value of bool flag in returned pair will be true and iterator in it will  
point to the newly inserted element.  
Whereas, if insertion is failed because it was a duplicate element then bool flag in returned pair will be false.

Suppose we have a set of strings i.e.

|  |  |
| --- | --- |
| 1 | std::set<std::string> setOfStrs; |

Inserting an element with insert() i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // A pair of set iterator and bool  std::pair<std::set<std::string>::iterator, bool> result;    // Insert Returns a pair of iterator and bool  result = setOfStrs.insert("Hi");    // Check if element added sucessfuly  if(result.second)  std::cout<<"Hi"<<" - Inserted sucessfuly"<<std::endl; |

Check out complete example as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <iostream>  #include <set>  #include <iterator>  #include <string>    void insertInSet(std::set<std::string> & setOfStrs, std::string str)  {  // A pair of set iterator and bool  std::pair<std::set<std::string>::iterator, bool> result;    // Insert Returns a pair of iterator and bool  result = setOfStrs.insert(str);    // Check if element added sucessfuly  if(result.second)  std::cout<<str<<" - Inserted sucessfuly"<<std::endl;  else  std::cout<<str<<" - Not Inserted sucessfuly"<<std::endl;    }    int main()  {  std::set<std::string> setOfStrs;      insertInSet(setOfStrs, "Hi");  // Try to insert the duplicate element.  insertInSet(setOfStrs, "Hi");    insertInSet(setOfStrs, "the");  insertInSet(setOfStrs, "is");  insertInSet(setOfStrs, "Hello");    std::cout<<"\*\*Map Contents\*\*\*"<<std::endl;    for(auto elem : setOfStrs)  std::cout<<elem<<std::endl;    return 0;  } |

Output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Hi - Inserted sucessfuly  Hi - Not Inserted sucessfuly  the - Inserted sucessfuly  is - Inserted sucessfuly  Hello - Inserted sucessfuly  \*\*Map Contents\*\*\*  Hello  Hi  is  the |

## Inserting an Iterator Range into a Set

std::set provides an another overloaded version of insert() i.e.

|  |  |
| --- | --- |
| 1  2 | template <class InputIterator>  void insert (InputIterator first, InputIterator last); |

It accepts a range of input iterators and one by one all the elements in set while traversing through the range( first to last-1).

Check out the example, we are inserting elements from a vector to set by giving its iterator range to insert() function i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <iostream>  #include <set>  #include <iterator>  #include <string>  #include <vector>    int main()  {  std::vector<std::string> vecOfStrs = {"Hi", "Hello", "is", "the", "at", "Hi", "is"};    std::set<std::string> setOfStrs;    // Insert a Range in set  // Range here is start and end iterators of a vector  setOfStrs.insert(vecOfStrs.begin(), vecOfStrs.end());    // It will insert all the elements in vector to set, but as  // set contains only unique elements, so duplicate elements will  // be automatically rejected.    // But there is no way to find out how many actually inserted  // because it doesn't return any value.    std::cout<<"\*\*Map Contents\*\*\*"<<std::endl;    for(auto elem : setOfStrs)  std::cout<<elem<<std::endl;    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \*\*Map Contents\*\*\*  Hello  Hi  at  is  the |

It will only add unique elements in set and reject the duplicate ones. As this version of overloaded insert() function doesn’t return any value. Hence, we have no information after the call, that what elements were actually inserted and what got rejected.

Similarly, we can insert elements from other containers too by providing a range of iterators to the insert() function.

## Inserting a Initializer List in Set

std::set provides an another overloaded version of insert() i.e.

|  |  |
| --- | --- |
| 1 | void insert (initializer\_list<value\_type> il); |

It accepts a initializer list and insert all elements in it to the set.

Check out the example, we are inserting elements from an initiazer list to the set i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <iostream>  #include <set>  #include <iterator>  #include <string>    int main()  {  std::set<std::string> setOfStrs;    // Insert a Initializer list in the set  setOfStrs.insert({"Hi", "Hello", "is", "the", "at", "Hi", "is"});    // It will insert all the elements in initializer\_list to set, but as  // set contains only unique elements, so duplicate elements will  // be automatically rejected.    // But there is no way to find out how many actually inserted  // because it doesn't return any value.    std::cout<<"\*\*Map Contents\*\*\*"<<std::endl;    for(auto elem : setOfStrs)  std::cout<<elem<<std::endl;    return 0;  } |

Output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \*\*Map Contents\*\*\*  Hello  Hi  at  is  the |

It will only add unique elements in set and reject the duplicate ones. As this version of overloaded insert() function doesn’t return any value.  
Hence, we have no information after the call, that what elements were actually inserted and what got rejected.