deep se

dependable evolvable pervasive software engineering group

Standard Template Library

Danilo Ardagna

Politecnico di Milano

danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- STL overview
- Sequential containers introduction
- How are vectors implemented?
- Sequential containers overview

Standard Template Library

- STL is a software library for the C++ programming language that provides four components:
  - ***algorithms***
  - ***containers***
  - ***functional*** (or functor)
  - ***iterators***

- STL provides a ready-made set of containers that can be used with **any built-in type** and with **any user-defined** type that supports some elementary operations (**copying** and **assignment**, which are synthetized for us by the compiler if we don’t define)

- Containers implement a **like-a-value semantic**

Container elements are copies

- When we use an object to initialize a container, or **insert an object into a container**, a **copy** of that object value is placed in the container, **not the object itself**

- Just as when we pass an object to a non-reference parameter (pass by value!), there is **no relationship between the element in the container and the object** from which that value originated

Subsequent changes to the element in the container have no effect on the original object, and vice versa

Standard Template Library

- STL **algorithms** are **independent** of **containers**, which significantly reduces the complexity of the library
  - This is obtained also thanks to **iterators**

- STL achieves its results through the use of **templates**
  - This approach provides **compile-time polymorphism** that is often more **efficient** than traditional run-time polymorphism
  - Modern C++ **compilers** are tuned to minimize abstraction penalty arising from heavy use of the STL

STL – Container Classes

- Container classes share a common interface, which each of the containers extends in its own way
  - This common interface makes the library easier to learn
  - It is also easy to change container type (limited changes in the remaining code)

- Each kind of container offers a different set of performance and functionality trade-offs (this is why we discussed about **complexity**)

- A container holds a collection of objects of a specified type
  - **Sequential containers:**
    - Let the programmer control the order in which the elements are stored and accessed
    - That order does not depend on the values of the elements but on their position
  - **Associative containers:**
    - Store their elements based on the value of a key
    - Elements are retrieved efficiently according to their key value

Sequential and Associative containers

Containers

Sequential

Ordered collection of elements whose position is independent from the value of the element.
vector, list, deque

Associative

Collections of elements (pairs Key-Value)
set, multiset, map, multimap

Ordered

A strict ordering relation has been defined through a specialization of the functor less< T> or by overloading **operator<** (). The position of an element depends on its value. set< T> and map< T,V> (no repetition), and multiset< T> and multimap< T,V> (with repetition).

Un-ordered

A **hashing function** should be provided together with an equivalence relation.
unordered set<T>, unordered multiset<T>, unordered map<T,V> and unordered multimap<T,V>

Sequential Containers

- The sequential containers provide fast sequential access to their elements
- However, they offer different **performance trade-offs** relative to:
  - the costs to **add** or **delete** elements to the container
  - the costs to perform **non-sequential access** to elements of the container

`vector`	Flexible-size array. Supports **fast random** access. Inserting or deleting elements other than the **back** is slow
`deque`	Double-ended queue. Supports **fast random access. Fast insert/delete at front or back**
`list`	Doubly linked list. Supports only **bidirectional sequential access**. **Fast insert/delete at any point**
`forward_list`	Singly linked list. Supports only **sequential access** in **one direction**. **Fast insert/delete at any point**

Sequential Containers

- The sequential containers provide fast sequential access to their elements
- However, they offer different **performance trade-offs** relative to:
  - the costs to **add** or **delete** elements to the container
  - the costs to perform **non-sequential access** to elements of the container

`array`	Fixed-size array. Supports fast random access. Cannot add or remove elements
`string`	Specialized container (characters only), similar to vector. Fast random access. Fast insert/delete at the back

We don’t cover in details this STL part, consider as **readings**

A running example - GoodReads

A running example - GoodReads

- You have to implement GoodReads a free platform to share reviews and opinions on books

- Books are **uniquely identified** by their **title** and have an author (for the sake of simplicity suppose books have a single author and there are no books with the same title)

- A review is characterized by the title of the book, the text and the rating (an integer between 1 and 5)

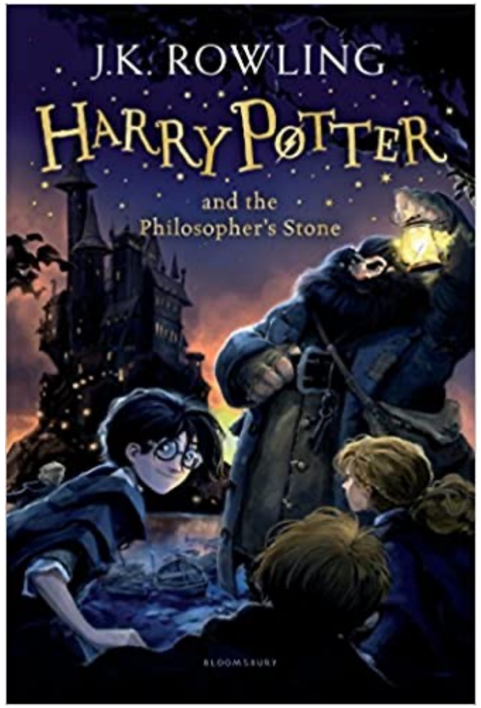
- Design goals:
  - Optimize the worst-case complexity
  - Favour as operation a book search
  - Optimize the computation of the average of the review scores

A running example - GoodReads

Title: Harry Potter and the Philosopher's stone
Author: J. K. Rowling
Publisher: Bloomsbury
pages: 223

1 stars: 121
2 stars: 4342
3 stars: 80012
4 stars: 199878
5 stars: 109010

Total reviews: 393363 **Average**: 4.05



A running example - GoodReads

- Within the class GoodReads:
  1. implement the method:
     - `void add_book(const string & title, unsigned pagesN, const string &publisher, const string &author)`
     - which adds a book and its relevant information to the system

  2. implement the method:
     - `void add_review(const string &bookTitle, const string &text, unsigned int rating)`
     - which adds a review to the system

A running example - GoodReads

3. implement the methods:

- `float get_avg_rating()`
- `float get_avg_rating(const string & title)`
- which provide the average ratings for all books and for the book with the specified title

4. implement the method:

- `void search_reviews(const vector<string> & keywords)`
- which, prints all the reviews including all the specified keywords

DEMO

GoodReads

- books []
- reviews []

+ add_book(const string & title, unsigned pagesNumber, const string &publisher, const string &author)
+ add_review(const string &bookTitle, const string &text, unsigned int rating)
+ get_avg_rating()
+ get_avg_rating(const string & title)
+ search_reviews(const vector<string> & keywords)
+ print_book(const string & title)
- find_book(const string &title)

- includes_all(const vector<string> &words, const vector<string> &keywords)
- includes_word (const vector<string> &words, const string &k)

Book

- ratings_distr[]
- pages_number
- publisher
- review_count
- author
- title;
- avg_rating;
- review_indexes[]

+ get_avg_rating()
+ add_review(unsigned index, unsigned stars)
+ to_string()
+ get_review_indexes()
+ get_title()

- compute_rating()

Review

- book_title
- text
- rating
- words []

+ to_string()
+ get_text()
+ get_words()

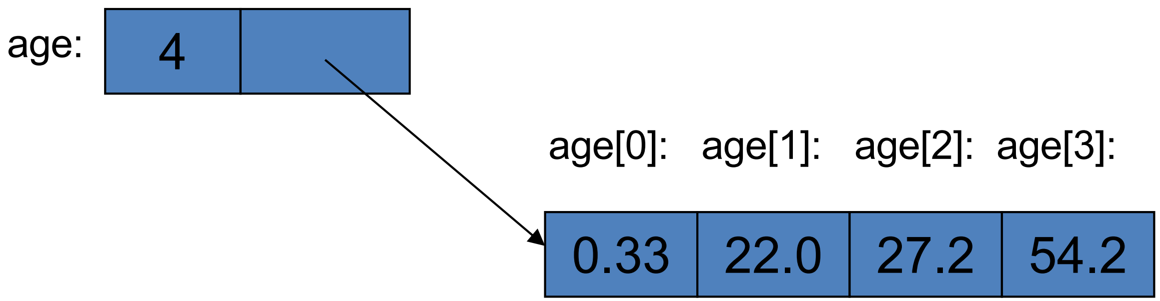
- find_in_words(const string & w)

How are `vector`s implemented?

Second part

Vector

- A vector
  - Can hold an arbitrary number of elements
    - Up to whatever physical memory and the operating system can handle
  - That number can vary over time
    - E.g. by using `push_back()`
  - Example
    - `vector<double> age(4);`
    - `age[0]=.33;  age[1]=22.0;  age[2]=27.2;  age[3]=54.2;`



Stored in the heap!

- Memory blocks are reallocated as vector grows
- This is done efficiently (average case)
- We will compute worst case complexity

Changing `vector` size

- Given

```
vector v(n);   // v.size()==n
```

- We can change its size in three ways
  - Add an element
    - `v.push_back(7);` // add an element with the value 7 to the end of v
      // v.size() increases by 1
  - Resize it
    - `v.resize(10);` // v now has 10 elements
  - Assign to it
    - `v = v2;` // v is now a copy of v2
      // v.size() now equals v2.size()

How a `vector` grows

- To support **fast random access**, vector elements are **stored contiguously**

- Given that elements are contiguous, and that the size of the container is flexible, when we add an element if there is no room for the new element:
  - the container must allocate new memory to hold the existing elements plus the new one
  - copy the elements from the old location into the new space
  - add the new element
  - deallocate the old memory

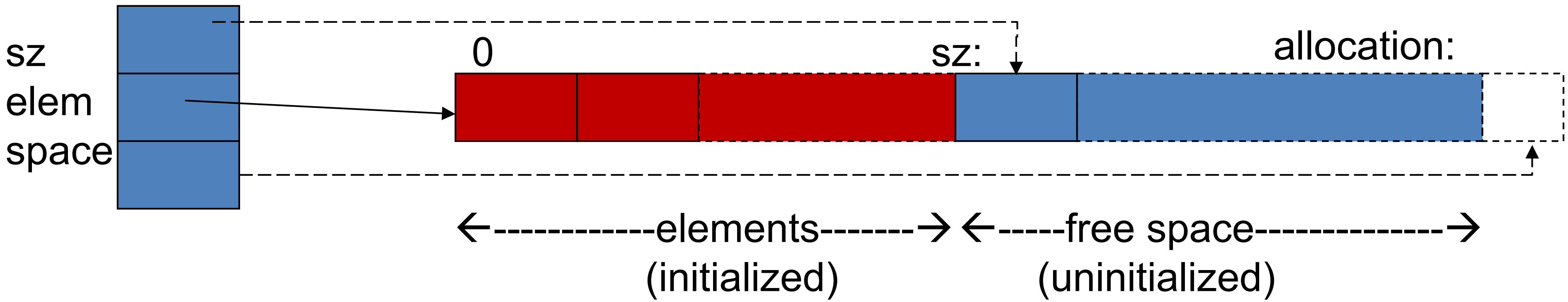
How a `vector` grows

- To avoid these costs, library implementors use allocation strategies that **reduce the number of times** the container is **reallocated**

- When new memory is allocated, **allocate capacity beyond** what is immediately **needed**
  - The container holds this storage in reserve and uses it to allocate new elements as they are added
  - This allocation strategy is dramatically more efficient than reallocating the container each time an element is added

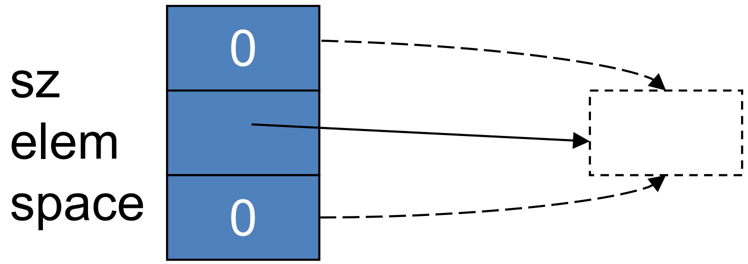
Representing vector

- If you `resize` or `push_back` once, you'll probably do it again;
  - Let's prepare for that by keeping a bit of free space for future expansion

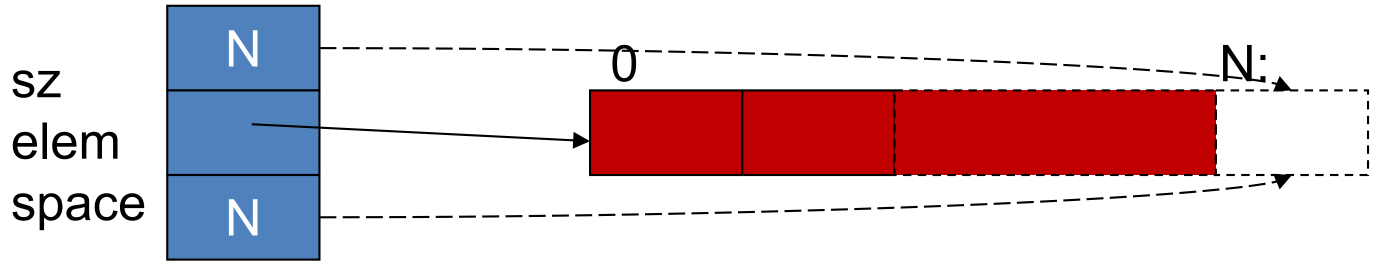


Representing vector

- An empty vector (no free store use):



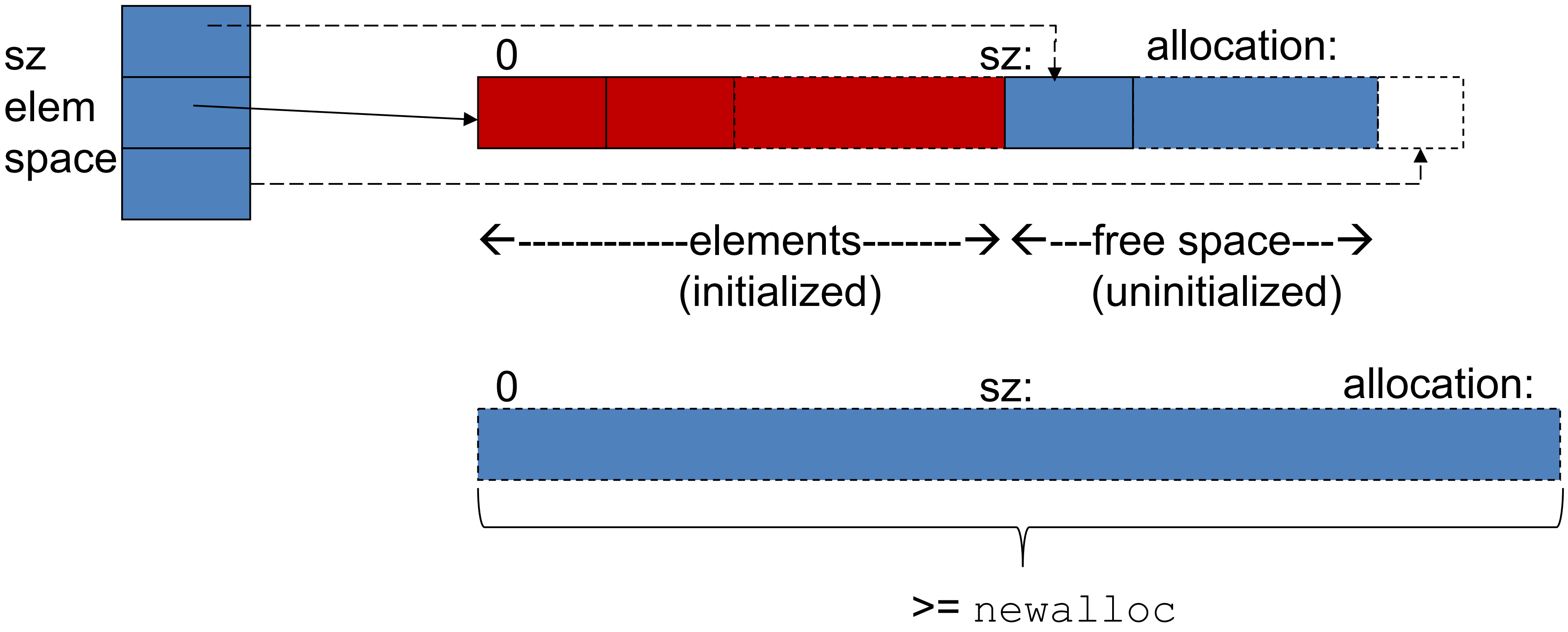
- A vector(N) (no free space):



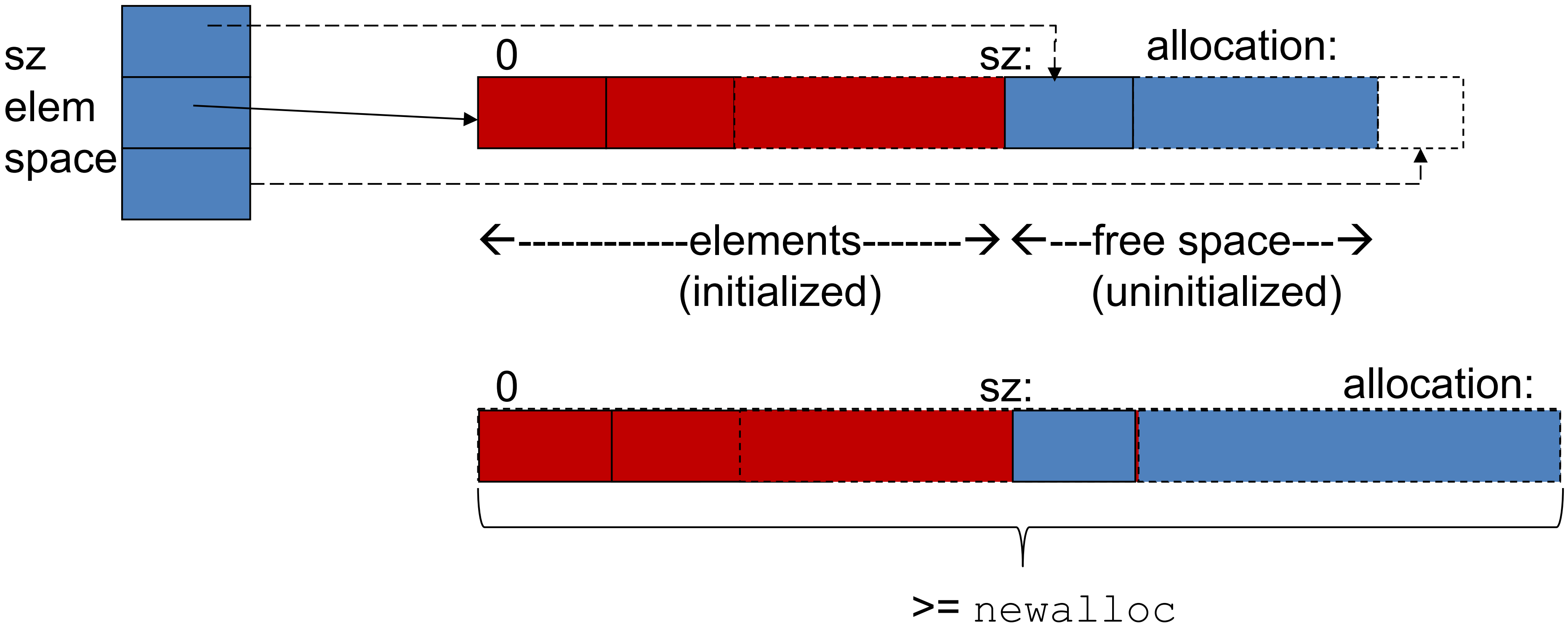
vector<T>::reserve(`unsigned newalloc`)

- `reserve(unsigned newalloc)`
  - Deals with space (allocation); given space all else is easy
  - Doesn't mess with size or element values
- If the requested size is less than or equal to the existing capacity, reserve does nothing
  - Calling reserve with a size smaller than capacity does not cause the container to give back memory
- After calling reserve, the capacity will be greater than or equal to the argument passed to reserve

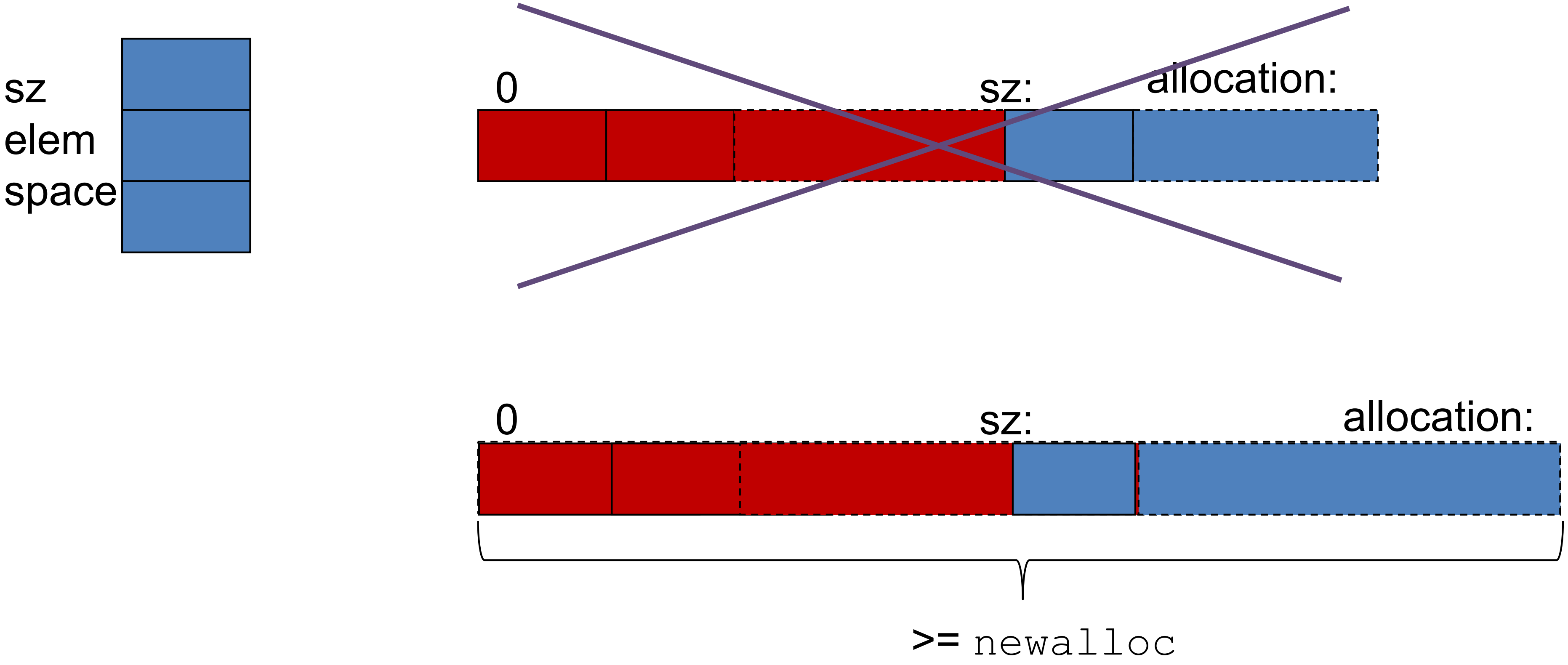
vector<T>::reserve(`unsigned newalloc`)



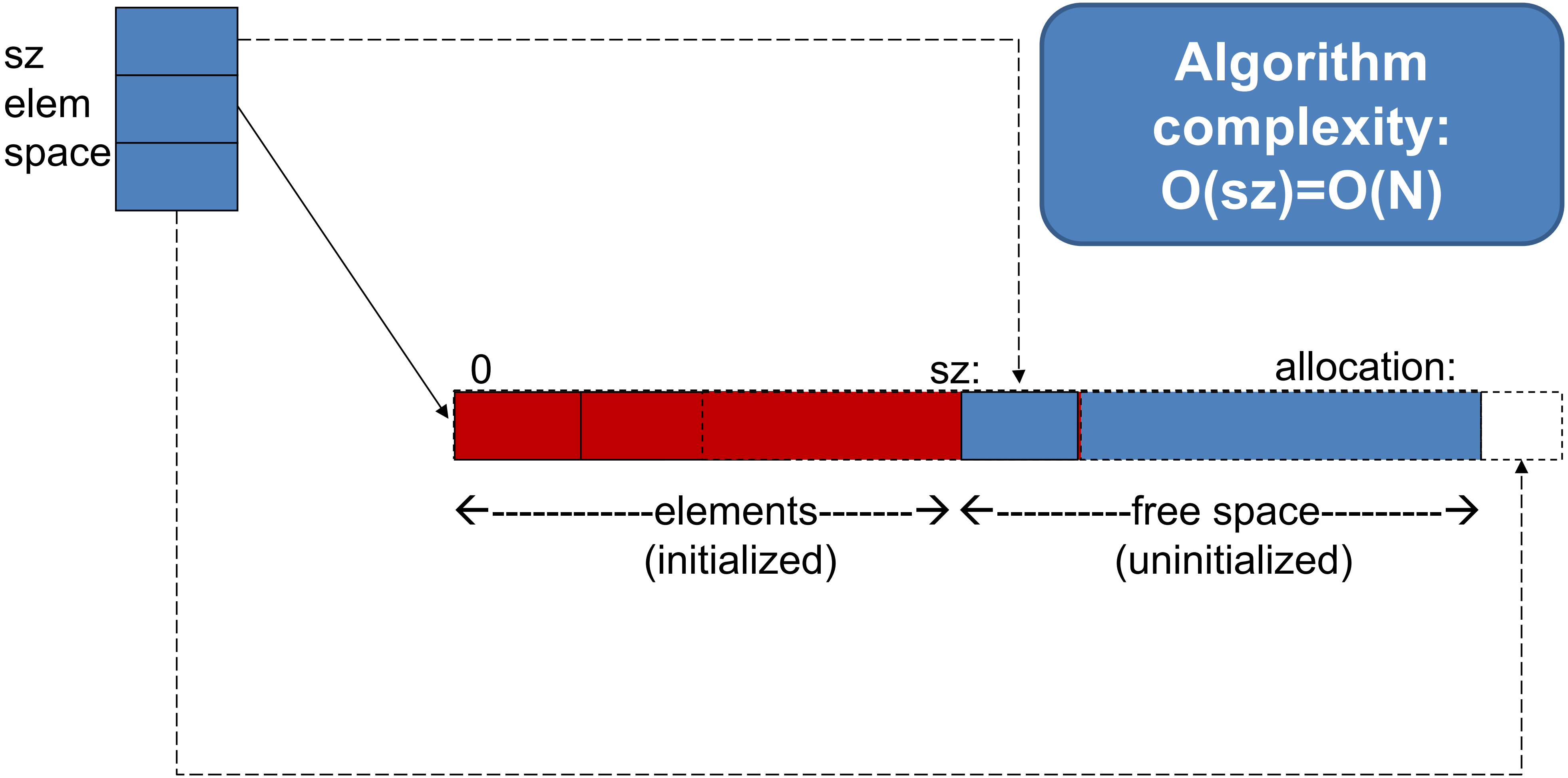
vector<T>::reserve(`unsigned newalloc`)



vector<T>::reserve(`unsigned newalloc`)



vector<T>::reserve(`unsigned newalloc`)



Algorithm complexity: O(sz)=O(N)

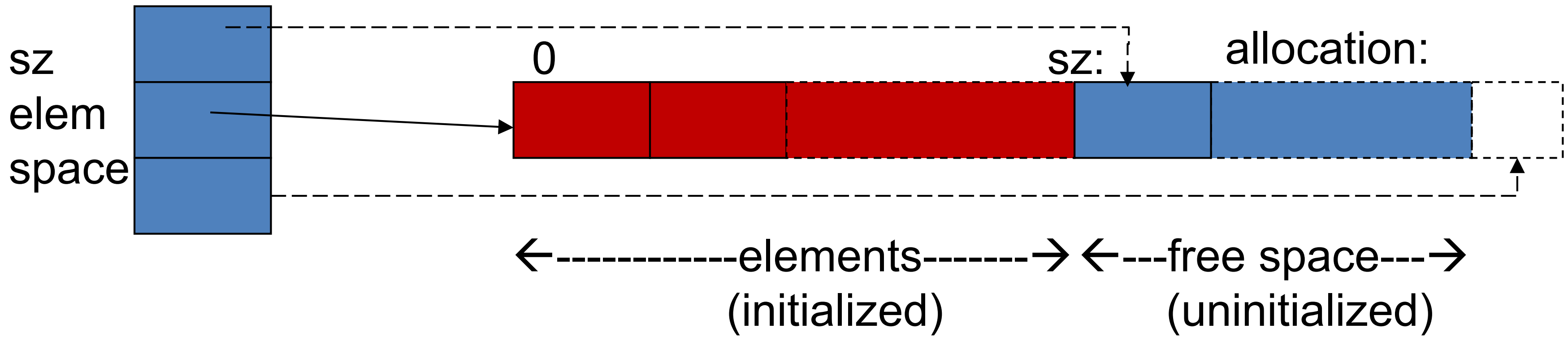
vector<T>::resize(`unsigned newsize`)

- Given `reserve`, `resize` is easy
  - `reserve` deals with space/allocation
  - `resize` deals with element values

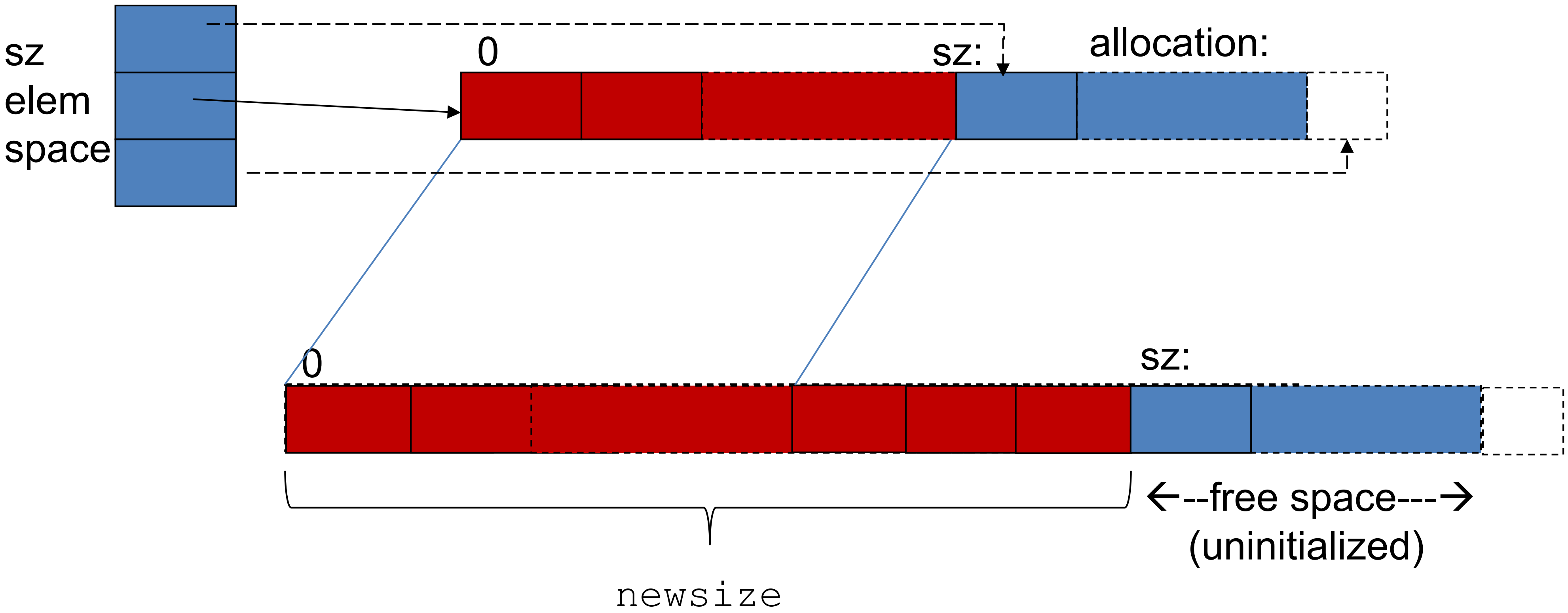
- `resize()` goal is to:
  - Reserve `newsize` elements
  - Fill the the elements with indeces between sz and `newsize-1` with a default value

Algorithm complexity: O(newsize)

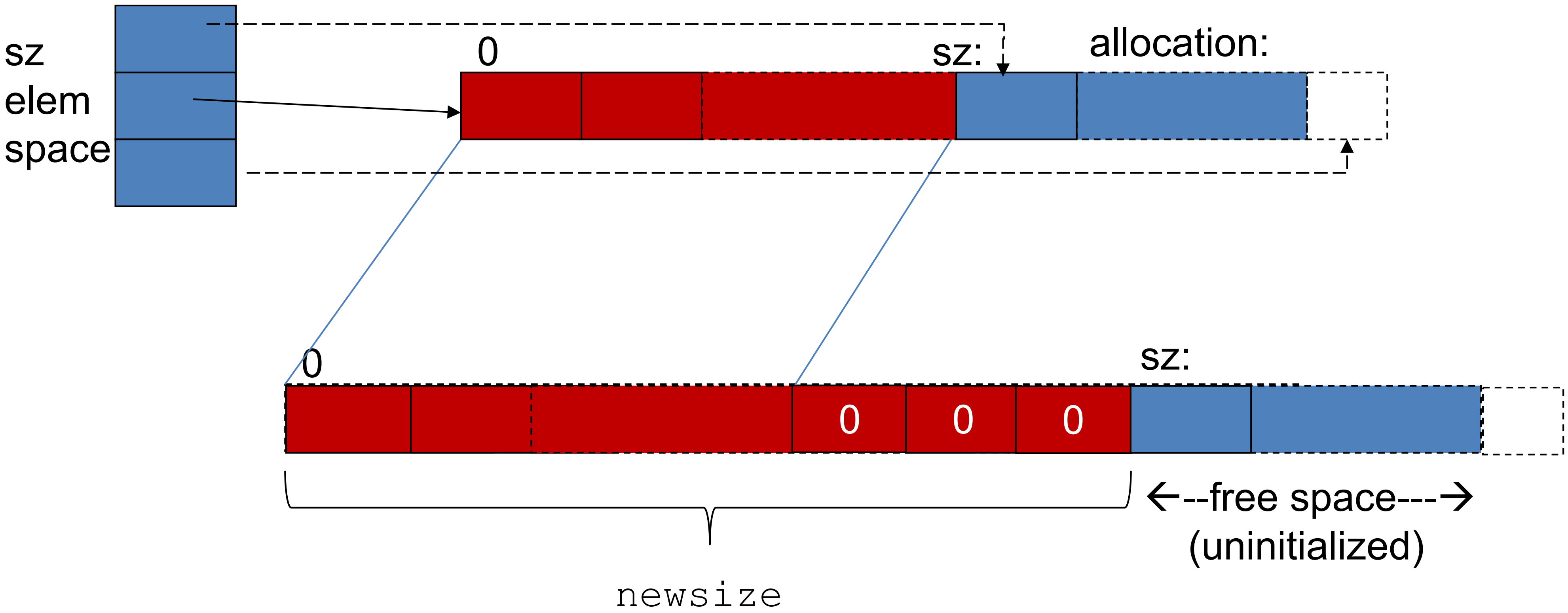
vector<T>::resize(`unsigned newsize`)



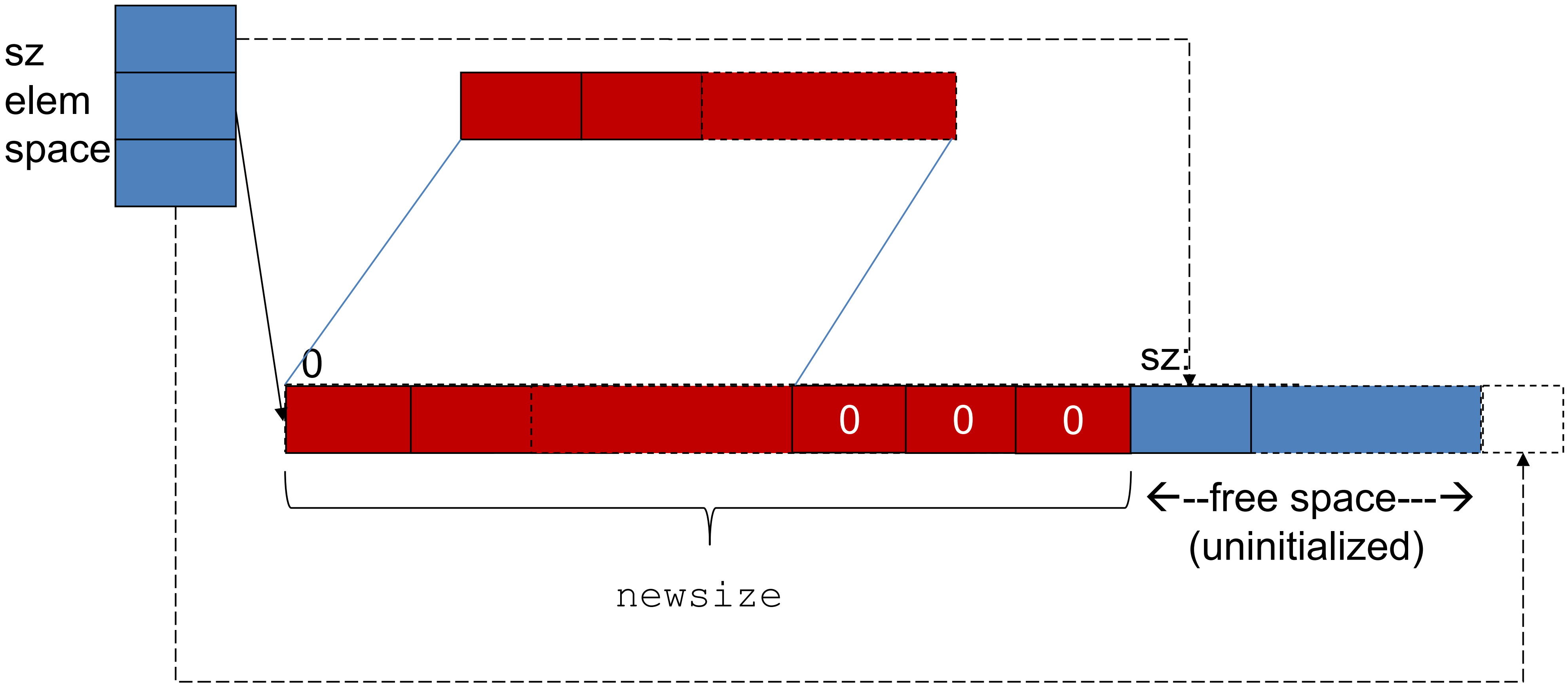
vector<T>::resize(`unsigned newsize`)



vector<T>::resize(`unsigned newsize`)



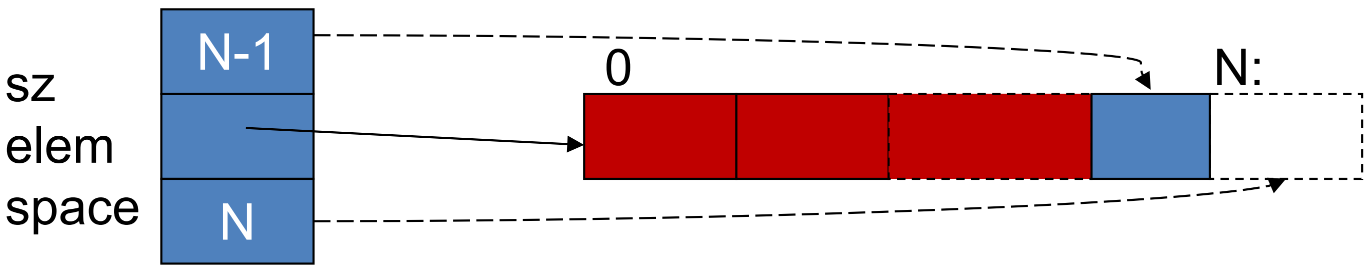
vector<T>::resize(`unsigned newsize`)



Algorithm complexity: O(newsize)

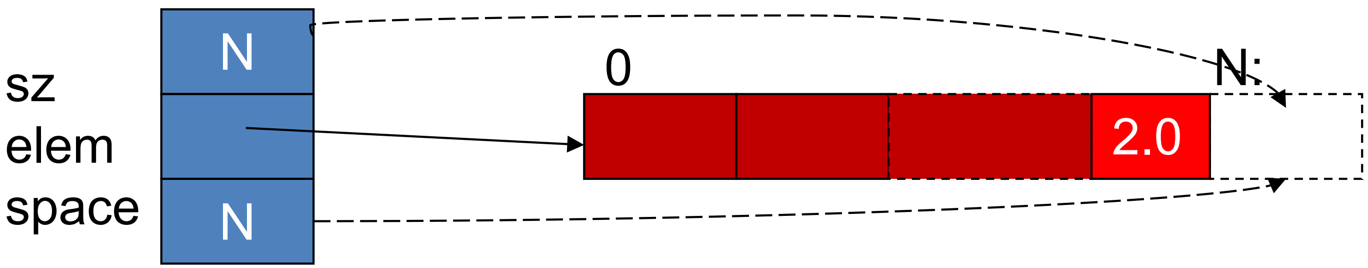
vector<T>::push_back(`T val`)

- If there is enough room simply increment sz and store the element `val`
- Otherwise `reserve` twice sz elements and add `val`



vector<T>::push_back(`T val`)

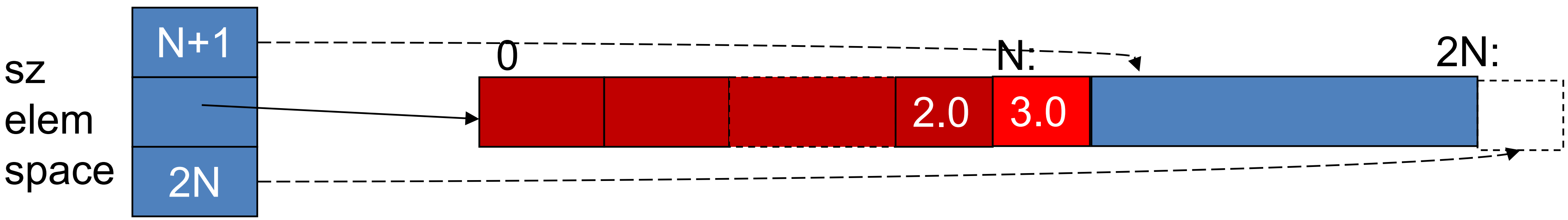
- If there is enough room simply increment sz and store the element `val`
- Otherwise `reserve` twice sz elements and add `val`



`v.push_back(2.0);`

vector<T>::push_back(`T val`)

- If there is enough room simply increment sz and store the element `val`
- Otherwise `reserve` twice sz elements and add `val`




```
v.push_back(2.0);
v.push_back(3.0);
```

vector<T>::push_back(`T val`)

- If there is enough room simply increment sz and store the element `val`

- Otherwise `reserve` twice sz elements and add `val`

Algorithm complexity: O(sz)=O(N)

Vectors complexity final considerations

- Working with vectors implies push_back worst case complexity O(N), but the average case (also called amortized complexity) is O(1) (push_back is efficient in general)
- Random access is O(1)
- Insert in the middle worst and average cases are O(N)

GoodReads method complexity

- `Book::add_review()`
  - `push_back` in a vector
  - Worst case complexity O(n_reviews)
- `GoodReads::find_book()`
  - Sequential search in a vector
  - Worst case complexity O(n_books)
- `GoodReads::add_book()`
  - `find_book` and `push_back` in a vector
  - Worst case complexity O(n_books)
- `GoodReads::get_avg_rating()`
  - Sequential access in a vector
  - Worst case complexity O(n_books)
- `GoodReads::get_avg_rating(const string & title)`
  - `find_book`
  - Worst case complexity O(n_books)

- In what follows we try to improve `GoodReads` implementation with a focus on the worst case complexity

Range checking

- Ideal: we would like that the STL implementation checks if index is within vector range

- STL doesn't guarantee range checking. Why?
  - Checking cost in speed and code size
  - Some projects need optimal performance
    - Think huge (e.g., Google) and tiny (e.g., cell phone)
  - The standard must serve everybody
    - You can build checked on top of optimal
    - You can't build optimal on top of checked

Sequential Containers Overview

Sequential Containers

- Provide efficient, flexible memory management:
  - We can add and remove elements, grow and shrink the size of the container….
  - …with an exception, array (fixed-size container)

- The strategies that containers use for storing their elements have inherent, and sometimes significant, impact on the efficiency of these operations
  - In some cases, these strategies also affect whether a particular container supplies a **particular operation**

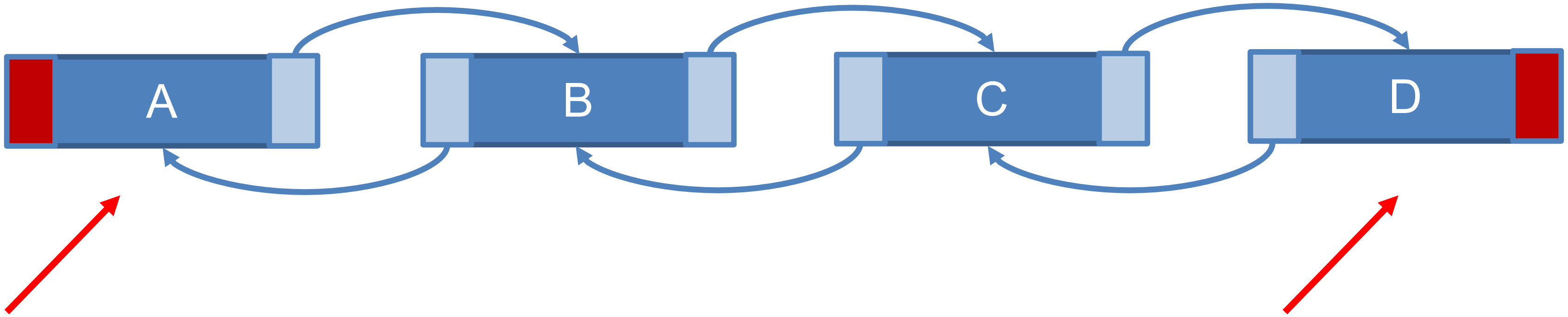
Sequential Containers Comparison

	Random Access	Add element at back	Add element in front	Add element in the middle
`vector`	+	+	N.A. (-)	-
`deque`	+	+	+	-
`list`	N.A.	++	++	++
`forward_list`	N.A.	N.A. (-)	++	++
`array`	+	N.A.	N.A.	N.A.

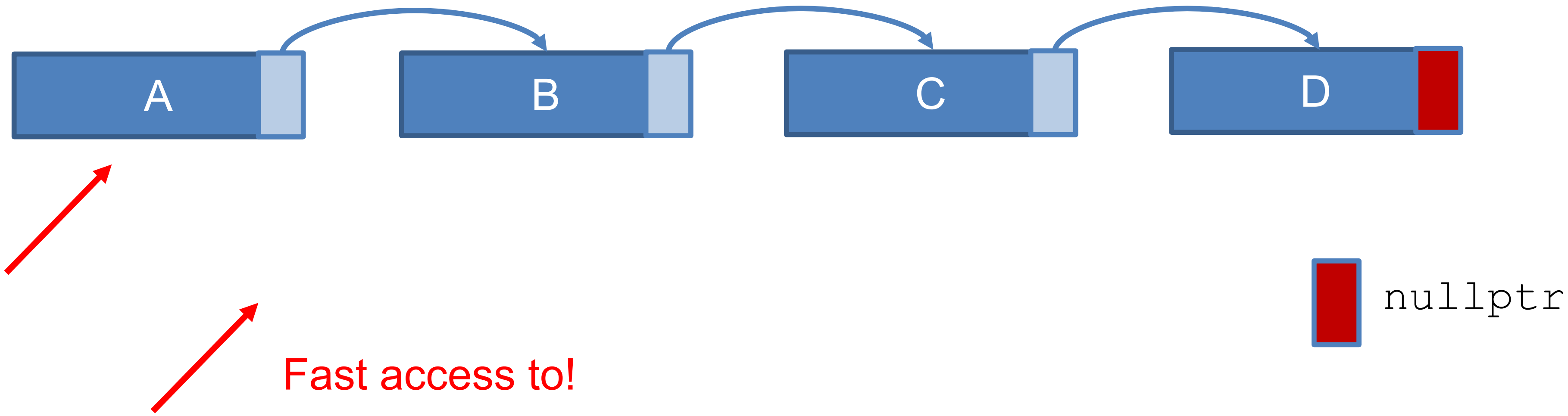
- Ranking based on amortized (average) complexity, worst case complexity can lead to a different ranking
- Add in the middle, assumes you have access (an iterator) to the element before the one you will insert

Sequential Containers - `list` and `forward_list`

- `list` implements a doubly-linked list



- `forward_list` implements a singly-linked list



Fast access to!

Sequential Containers

- The `list` and `forward_list` containers are designed to make it fast to add or remove an element anywhere in the container
  - In exchange, these types do **not** support **random access** to elements
  - The **memory overhead** for these containers is significant

- A `deque` is a more complicated data structure
  - Like string and vector, supports **fast random access** and **adding** or **removing** elements in the **middle** of a deque is an **expensive** operation
  - Adding or removing elements at either front or end is a fast operation, comparable to a `list` or `forward_list`

Sequential Containers

- The `forward_list` and `array` types were added by C++ 11
- A `forward_list` comparable with `list`
  - Does not have the `size` operation and more memory efficient than `list`
  - But can be accessed from the **begin to end only** (cannot move backwards)
- An `array` is a safer, easier-to-use alternative to built-in arrays and has fixed size
  - Does not support operations to add and remove elements or to resize

Which Sequential Container to Use?

- It is best to use `vector` unless there is a good reason to prefer another container
- If you need lots of small elements and space overhead matters, don't use `list` or `forward_list`
- If the program requires random access to elements, use a `vector` or a `deque`
- If the program needs to insert or delete elements in the middle of the container, use a `list` or `forward_list`

Which Sequential Container to Use?

- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a `deque`

- If the program needs to insert elements in the middle of the container only while reading input, and subsequently needs random access to the elements:
  - First, decide whether you actually need to add elements in the middle of a container. It is often easier to append to a vector and then call the library `sort` function to reorder the container when you’re done with input
  - If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete, copy the `list` into a `vector`

Which Sequential Container to Use?

- If the program needs random access and needs to insert and delete elements in the middle:
  - Evaluate the relative cost of accessing the elements in a `list` or `forward_list` versus the cost of inserting or deleting elements in a `vector` or `deque`

Which Sequential Container to Use?

- In general, the predominant operation of the application (whether it does more access or more insertion or deletion) will determine the choice of container type

- Application performance testing usually needed

If **not sure** which container to use, write your code **using only operations common** to both vectors and lists:

- **use iterators, not subscripts**
- **avoid random access** to elements

This way code changes will be easy!

Container common types

`iterator`	Type of the iterator for the considered container type
`const_iterator`	Iterator type that can read but cannot change its elements
`size_type`	Uns. int. large enough to hold the largest possible container size
`difference_type`	Sign. int. large enough to hold the distance between two iterators
`value_type`	Element type
`reference`	Element lvalue reference type, synonymous for `value_type &`
`const_reference`	Element const lvalue type, (i.e., `const value_type &`)

Container common operations

C c;	Default constructor, empty container
C c1(c2);	Construct c1 as a copy of c2
C c(b, e);	Copy elements from the range denoted by the iterators b and e (no for `array`)
C c{a, b, c, …};	List initialize c

c.size()	Number of elements in c (no for `forward_list`)
c.max_size()	Maximum number of elements c can hold
c.empty()	`true` if c has no elements, false otherwise

Container common operations

c.insert(args)	Copy element(s) as specified by *args* in c
c.emplace(inits)	Use *inits* to construct an element in c
c.erase(args)	Remove element(s) specified by *args*
c.clear()	Remove all elements from c

==, !=	Equality
<,<=,>,>=	Relationals (no for unordered associative containers)

Container common operations

c.begin(), c.end()	Return iterator to the first/one past element in c
c.cbegin(), c.cend()	Return `const_iterator`

`reverse_iterator`	Iterator that addresses elements in reverse order
`const_reverse_iterator`	Reverse iterator read only
c.rbegin(), c.rend()	Iterator to the last, one past the first element in c
c.crbegin(), c.crend()	Return `const_reverse_iterator`

No for `forward_list`

Creating a container

- Each container is defined in a header file with the same name as the type
- Containers are class templates
  - We must supply additional information to generate a particular container type, usually at least element type

```
list<Sales_data> l; // list that holds Sales_data objects
deque<double> d;  // deque that holds doubles
```

Constraints on types that a container can hold

- Almost any type can be used as the element type of a sequential container

```
vector<vector<string>> lines; // vector of vectors of strings
```

- Some container operations impose requirements of their own on the element type
  - We can define a container for a type that does not support an operation-specific requirement, but we can use an operation only if the element type meets that operation requirements

```
// assume noDefault is a type without a default constructor
vector<noDefault> v1(10, init); // ok: element initializer supplied
vector<noDefault> v2(10); // error: must supply an
                          // element initializer
```

Iterators

- Iterators have also a common interface:
  - All the iterators let access an element from a container providing the dereference operator and allow to move from one element to the next through the increment operator

Iterators

*iter	Returns a reference to the element denoted by the iterator *iter*
iter->memb (*iter).memb	Dereferences *iter* and fetches the member *memb* from the underlying element
++iter	Increments *iter* to refer to the next element in the container
--iter	Decrements *iter* to refer to the previous element in the container
iter1==iter2 iter1!=iter2	Compares two iterators. Two iterators are equal if they denote the same element or if they are the **off-the-end** iterator for the **same container**

- `forward_list` iterators do not support the decrement (--) operator.

Iterators – Random access

iter + n	Adding (subtracting) an integral value *n* from the iterator *iter* yields an iterator *n* elements forward of backward than *iter* within the container
iter - n	
iter1 +=n	Assign to *iter1* the value of adding (subtracting) n to *iter1*
iter1 -=n	
iter1-iter2	Compute the number of elements between *iter1* and *iter2*
>,>=,<,<=	One iterator is less than another if it denotes an element that appears in the container before the one referred to

- The iterator arithmetic operations listed above apply only to iterators for **`string`**, **`vector`**, **`deque`**, and **`array`**

Iterator Ranges

- Denoted by a pair of iterators each of which refers to an element, or to one past the last element, in the same container
- Often referred to as begin and end or (somewhat misleadingly) as first and last
- We have a **left-inclusive interval:** [begin, end)
- Nice properties:
  - If begin equals end, the range is empty
  - If begin is not equal to end, there is at least one element in the range, and begin refers to the first element in that range
  - We can increment begin some number of times until begin == end

Iterator Ranges

```
while (begin != end) {
        *begin = val;  // ok: range isn't empty so begin denotes
                       //  an element
        ++begin;  // advance the iterator to get the next element
}
```

Reverse Iterators

- Most containers provide reverse iterators, i.e., an iterator that goes backward through a container and inverts the meaning of the iterator operations
  - Saying ++ on a reverse iterator yields the previous element
  - Standard way to write iterators code independent of iterator direction

DEMO

Assignment operator

- The assignment operator replaces the entire range of elements in the left-hand container with **copies** of the elements from the right-hand operand
- After an assignment, the left- and right-hand containers are equal
  - If the containers had been of unequal size, after the assignment both containers would have the size of the right-hand operand

Container Assignment operator

c1=c2	Replace the elements in c1 with copies form c2. c1 and c2 must have the same type
c={a, b, c, …}	Replace the elements in c1 with copies of elements in the initializer list
swap(c1,c2)	Exchanges elements in c1 with those in c2. c1 and c2 must have the same type
c1.swap(c2)	
seq.assign(b,e)	Replaces elements in seq with those in the range denoted by the iterators b and e. b and e **must not be** iterators belonging to seq
seq.assign(i1)	Replaces elements in seq with those in the initializer list i1
seq.assign(n,t)	Replaces elements in seq with n elements with value t

Using `swap`

- Exchanges the contents of two containers of the same type
- After the call to `swap`, the elements in the two containers are interchanged

```
vector<string> svec1(10);  // vector with ten elements
vector<string> svec2(24);  // vector with 24 elements
swap(svec1, svec2);
```

- After the swap, svec1 contains 24 string elements and svec2 contains 10
- With the exception of arrays, swapping two containers is guaranteed to be fast, the elements themselves are not swapped; internal data structures are swapped (O(1) complexity)
- Swapping two arrays does exchange the elements
  - Requires time proportional to the number of elements in the array (complexity O(N))

Pointers invalidation and `swap`

- The fact that elements are not moved means that **iterators, references, and pointers** into the containers **are not invalidated**
  - They refer to the same elements as they did before the swap
  - After the swap, those elements are in a different container
  - For example, if *iter* denoted the string at position `svec1[3]` before the swap, it will denote element at position `svec2[3]` after the swap

Using `swap`

- In C++ 11, the containers offer both a member and non-member version of swap
- Earlier versions of the library defined only the member version of swap (e.g., `v1.swap(v2)`)
- The non-member swap (e.g., **`swap(v1,v2)`**) is of most importance in generic programs
- As a matter of habit, it is best to use the non-member version of swap

Container Size Operations

- Container types have three size-related operations
  - `size()` returns the number of elements in the container
  - `empty()` returns a bool that is true if size is zero and false otherwise
  - `max_size()` returns a number that is greater than or equal to the number of elements a container of that type can contain

- `forward_list` provides `max_size()` and `empty()`, but not `size()`

Relational Operators

- Every container type supports the equality operators (== and !=)
- All the containers except the unordered associative containers also support the relational operators (>, >=, <, <=)
- The right- and left-hand operands must be the **same** kind of **container** and must hold **elements** of the **same type**
  - We can compare a vector<int> only with another vector<int>
  - We cannot compare a vector<int> with a list<int> or a vector<double>

Relational Operators

- Comparing two containers performs a pairwise comparison of the elements (similarly to the string relationals)
  - If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise, they are unequal
  - If the containers have different sizes but every element of the smaller one is equal to the corresponding element of the larger one, then the smaller one is less than the other
  - If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements

Relational Operators

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
```

```
v1 < v2
v1 < v3
v1 == v4

v1 == v2
```

Relational Operators

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
```

`v1 < v2`	
`v1 < v3`	
`v1 == v4`	
`v1 == v2`	

Relational operators use their element relational operator

- The container equality operators use the element == operator, and the relational operators use the element < operator
- If the element type doesn't support the required operator, then we cannot use the corresponding operations on containers holding that type
- `Sales_data` type does not define either the == or the < operation, we cannot compare two containers that hold `Sales_data` elements

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // error: Sales_data has no less-than
                     // operator
```

Relational operators use their element relational operator

- Note that instead of relying on ==, you can use the `equal()` function defined within the STL header `algorithm`

- In that case you might specify also a binary function that establishes equality

- This will be discussed in a next class

Adding Elements to a Sequential Container

- Excepting `std::array`, all the library containers provide flexible memory management
  - We can add or remove elements dynamically changing the size of the container at run time

c.push_back(t) c.emplace_back(args)	Creates an element with value t or constructed from args at the end of c
c.push_front(t) c.emplace_front(args)	Creates an element with value t or constructed from args on the front of c
c.insert(p,t) c.emplace(p,args)	Creates an element with value t or constructed from args before the element denoted by iterator p
c.insert(p,n,t)	Creates n element with value t before the element denoted by iterator p

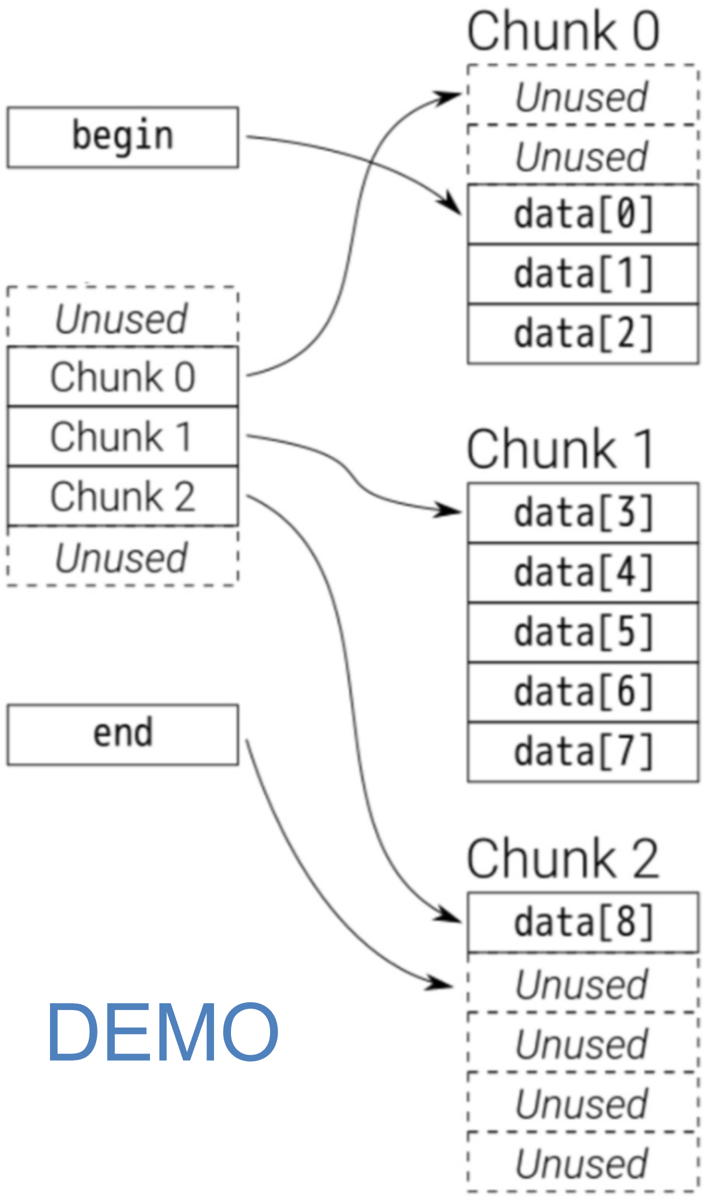
Adding Elements to a Sequential Container

c.insert(p,b,e)	Inserts the elements from the range denoted by the iterators b and e before the element denoted by the iterator p. **b and e may not be in c**
c.insert(p,i1)	i1 is a braced list of element values. Inserts the elements before the element denoted by the iterator p

- When we use these operations, we must remember that the containers use different strategies for allocating elements and that these strategies affect performance
  - Adding elements anywhere but at the end of a `vector` or `string`, or anywhere but the beginning or end of a `deque`, requires elements to be moved
  - Adding elements to a `vector` or a `string` may cause the entire object to be reallocated
  - Reallocating an object requires allocating new memory and moving elements from the old space to the new

deque

- Organizes data in chunks of memory referred by a sequence of pointers
- Like `vector` offers fast random access to its elements, provides the `push_front` member even though vector does not
- Guarantees (amortized) constant-time insert and delete of elements at the beginning and end of the container
- Inserting elements other than at the front or back of a `deque` is an expensive operation
- push_back and push_front worst case complexity O(N)



DEMO

Adding elements at a specified point in the container

- `insert` members
  - Let us insert one or more elements at any point in the container
  - Are supported for `vector`, `deque`, `list`, and `string`. `forward_list` provides specialized versions

- How `insert` works:
  - Takes an iterator as its first argument which indicates where in the container to put the element(s) (any position, including one past the end)
  - Element(s) are inserted before the position denoted by the iterator (the iterator might refer to a nonexistent element off the end)
  - **Returns an iterator** to the inserted element

DEMO

`insert`

- `slist.insert(iter, "Hello!");` *// insert "Hello!" just before iter*

- We can insert elements at the beginning of a container without worrying about whether the container has `push_front`

```
vector<string> svec;
list<string> slist;

// equivalent to calling slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");

// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector might be slow
svec.insert(svec.begin(), "Hello!");
```

- It is legal to insert anywhere in a `vector`, `deque`, or `string`. However, doing so is an expensive operation

Using the Return from `insert`

```
list<string> first;
auto iter = first.begin();
while (cin >> word)
      iter = first.insert(iter, word);// same as calling
                                      // push_front
```

Book

- ratings_distr[]
- pages_number
- publisher
- review_count
- author
- title;
- avg_rating;
- review_indexes[]

+ get_avg_rating()
+ add_review(unsigned index, unsigned stars)
+ to_string()
+ get_review_indexes()
+ get_title()

- compute_rating()

GoodReads

- books []
- reviews []

+ add_book(const string & title, unsigned pagesNumber, const string &publisher, const string &author)
+ add_review(const string &bookTitle, const string &text, unsigned int rating)
+ get_avg_rating()
+ get_avg_rating(const string & title)
+ search_reviews(const vector<string> & keywords)
+ print_book(const string & title)
- find_book(const string &title)

- includes_all(const vector<string> &words, const vector<string> &keywords)
- includes_word (const vector<string> &words, const string &k)

Review

- book_title
- text
- rating
- words []

+ to_string()
+ get_text()
+ get_words()

- find_in_words(const string & w)

How to improve Book::add_review() worst case complexity? O(n_reviews)!

Book

- ratings_distr[]
- pages_number
- publisher
- review_count
- author
- title;
- avg_rating;
- list<unsigned> review_indexes

+ get_avg_rating()
+ add_review(unsigned index, unsigned stars)
+ to_string()
+ get_review_indexes()
+ get_title()

- compute_rating()

GoodReads

- books []
- reviews []

+ add_book(const string & title, unsigned pagesNumber, const string &publisher, const string &author)
+ add_review(const string &bookTitle, const string &text, unsigned int rating)
+ get_avg_rating()
+ get_avg_rating(const string & title)
+ search_reviews(const vector<string> & keywords)
+ print_book(const string & title)
- find_book(const string &title)

- includes_all(const vector<string> &words, const vector<string> &keywords)
- includes_word (const vector<string> &words, const string &k)

Review

- book_title
- text
- rating
- words []

+ to_string()
+ get_text()
+ get_words()

- find_in_words(const string & w)

Use a list instead of a vector for Book::review_indexes

A running example - GoodReads

- add_review() based on vectors

```
void Book::add_review(unsigned int index, unsigned int stars)
{
    review_indexes.push_back(index);
    ratings_distr[stars-1]++;
    review_count++;
    avg_rating = compute_rating();
}
```

A running example - GoodReads

```
class Book {
    vector<unsigned> ratings_distr;
    unsigned pages_number;
    string publisher;
    unsigned review_count;
    string author;
    string title;
    float avg_rating;
    list<unsigned> review_indexes;
public:
    BookData(unsigned int pagesNumber, const string &publisher,
             const string &author);
    float get_avg_rating() const;
    void add_review(unsigned index, unsigned stars);
    string to_string() const;
    list<unsigned> get_review_indexes()const ;
private:
    float compute_rating();
};
```

A running example - GoodReads

- add_review() based on list

```
void Book::add_review(unsigned int index, unsigned int stars)
{
    review_indexes.push_back(index);
    ratings_distr[stars-1]++;
    review_count++;
    avg_rating = compute_rating();
}
```

- New worst case complexity O(1)
- Code is same!

Accessing Elements

c.back()	Returns a reference to the last element in c. Undefined if c is empty
c.front()	Returns a reference to the first element in c. Undefined if c is empty

Accessing Elements

<table>
<tr><td>c.back()</td><td>Returns a reference to the last element in c. Undefined if c is empty</td></tr>
<tr><td>c.front()</td><td>Returns a reference to the first element in c. Undefined if c is empty</td></tr>
<tr><td>c[n]</td><td rowspan="2">Returns a reference to the element indexed by n. Undefined if i>=c.size()</td></tr>
<tr><td>c.at(n)</td></tr>
</table>

Accessing Elements

```
// check that there are elements before dereferencing an
// iterator or calling front or back
if (!c.empty()) {
      // val and val2 are copies of the value of the first
      // element in c
      auto val = *c.begin(), val2 = c.front();

      // val3 and val4 are copies of the of the last element
      // in c
      auto last = c.end();
      auto val3 = *(--last); // can't decrement forward_list
                             // iterators
      auto val4 = c.back();  // not supported by forward_list
}
```

Accessing Elements

```
// check that there are elements before dereferencing an
// iterator or calling front or back
if (!c.empty()) {
        // val and val2 are copies of the value of the first
        // element in c
        auto val = *c.begin(), val2 = c.front();

        // val3 and val4 are copies of the of the last element
        // in c
        auto last = c.end();
        auto val3 = *(--last); // can't decrement forward_list

        auto val4 = c.back();                              rd_list
}
```

Consider to use rbegin instead of this!

The Access Members Return References

- The members that access elements in a container return references
- If the container is not const, the return is an ordinary reference that we can use to change the value of the fetched element

```
if (!c.empty()) {
      c.front() = 42;        // assigns 42 to the first element in c
      auto &v = c.back();   // get a reference to the last element
      v = 1024;              // changes the element in c
      auto v2 = c.back();   // v2 is not a reference;
                             // it's a copy of c.back()
      v2 = 0;                // no change to the element in c
}
```

Erasing elements

c.pop_back()	Removes the last element in c. Undefined if c is empty
c.pop_front()	Removes the first element in c. Undefined if c is empty
c.erase(p)	Removes the element denoted by the iterator p. Undefined if p is the off-the-end iterator
c.erase(b,e)	Removes the range of elements denoted by the iterators b and e
c.clear()	Removes all elements in c

The `pop_front` and `pop_back` members

- Functions that remove the first and last elements, respectively (return `void`)
- No `pop_front` for `vector` and `string`, `forward_list` does not have `pop_back`
- We cannot use a pop operation on an empty container

```
while (!ilist.empty()) {
      process(ilist.front()); // do something with the
                              // current top of ilist
      ilist.pop_front(); // done; remove the first element
}
```

erase

- Removes element(s) at a specified point in the container
  - We can delete a single element or a range of elements
  - Both forms of erase **return an iterator** referring to the location **after the** (last) **element that was removed**

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
      if (*it % 2)    // if the element is odd
            it = lst.erase(it);  // erase this element
      else
            ++it;
```

erase

```
// delete the range of elements between two iterators
// returns an iterator to the element just after the last removed element
elem1 = slist.erase(elem1, elem2);  // after the call
                                    // elem1 == elem2

slist.clear();  // delete all the elements within the container
slist.erase(slist.begin(), slist.end());  // equivalent
```

Resizing a Container

- We can use resize to make a container larger or smaller
- If the current size is greater than the requested size, elements are deleted from the back of the container
- If the current size is less than the new size, elements are added to the back of the container

c.resize(n)	Resize c so that it has n elements. If n<c.size(), the excess elements are discarded. If new elements must be added they are **value initialized**
c. resize(n,t)	Resize c to have n elements. Any elements added have value t

DEMO

Container operations may invalidate iterators

- Operations that **add** or **remove** elements from a container can **invalidate** pointers, references, or iterators to container elements
  - An invalidated pointer no longer denotes an element
  - Using an **invalidated pointer** is a **serious programming error**

- After an operation that adds elements to a container
  - Iterators, pointers, and references to a vector or string are invalid if the **container was reallocated**
  - If no reallocation happens, indirect references to elements before the insertion remain valid; those to elements after the insertion are invalid
  - **Very risky to rely on this!!!**

Container operations may invalidate iterators

- Iterators, pointers, and references to a `deque` are invalid if we add elements anywhere but at the front or back

- If we add at the front or back, iterators are invalidated, but references and pointers to existing elements are not

- Iterators, pointers, and references (including the off-the-end and the before- the-beginning iterators) to a `list` or `forward_list` remain valid

Writing Loops That Change a Container

- Loops that add or remove elements of a `vector`, `string`, or `deque` must cater to the fact that iterators, references, or pointers might be invalidated
- The program must ensure that the iterator, reference, or pointer is refreshed on each trip through the loop

```
// silly loop to remove even-valued elements and insert a duplicate of odd-
// valued elements
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // call begin, not cbegin because we're changing vi
while (iter != vi.end()) {
        if (*iter % 2) {
                iter = vi.insert(iter, *iter);// duplicate the
                                              // current (odd) element
                iter += 2; // advance past this element and
                           // the one inserted before it }
        else
                 iter = vi.erase(iter); // remove even elements
                                        // don't advance the iterator; iter denotes the
                                        // element after the one we erased
}
```

Avoid storing the iterator returned from end

- When we add or remove elements in a `vector` or `string`, or add elements or remove any but the first element in a `deque`, the iterator returned by `end()` is always invalidated
- Thus, loops that add or remove elements should always call `end()` rather than use a stored copy
- Partly for this reason, C++ standard libraries are usually implemented so that calling `end()` is a very fast operation

```
// disaster: the behavior of this loop is undefined
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // call begin, not cbegin because we're changing vi
auto end = vi.end();
while (iter != end) {
        if (*iter % 2) {
                iter = vi.insert(iter, *iter);// duplicate the
                                              // current (odd) element
                iter += 2; // advance past this element and
                           // the one inserted before it }
        else
                iter = vi.erase(iter); // remove even elements
                                       // don't advance the iterator; iter denotes the
                                       // element after the one we erased
}
```

No!!!! Disaster!!!!!!!!

References

- Lippman Chapter 9
- http://www.cplusplus.com/reference/stl/

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming