High Level Programming

Sequential Containers

Stefano Quer

Dipartimento di Automatica e Informatica

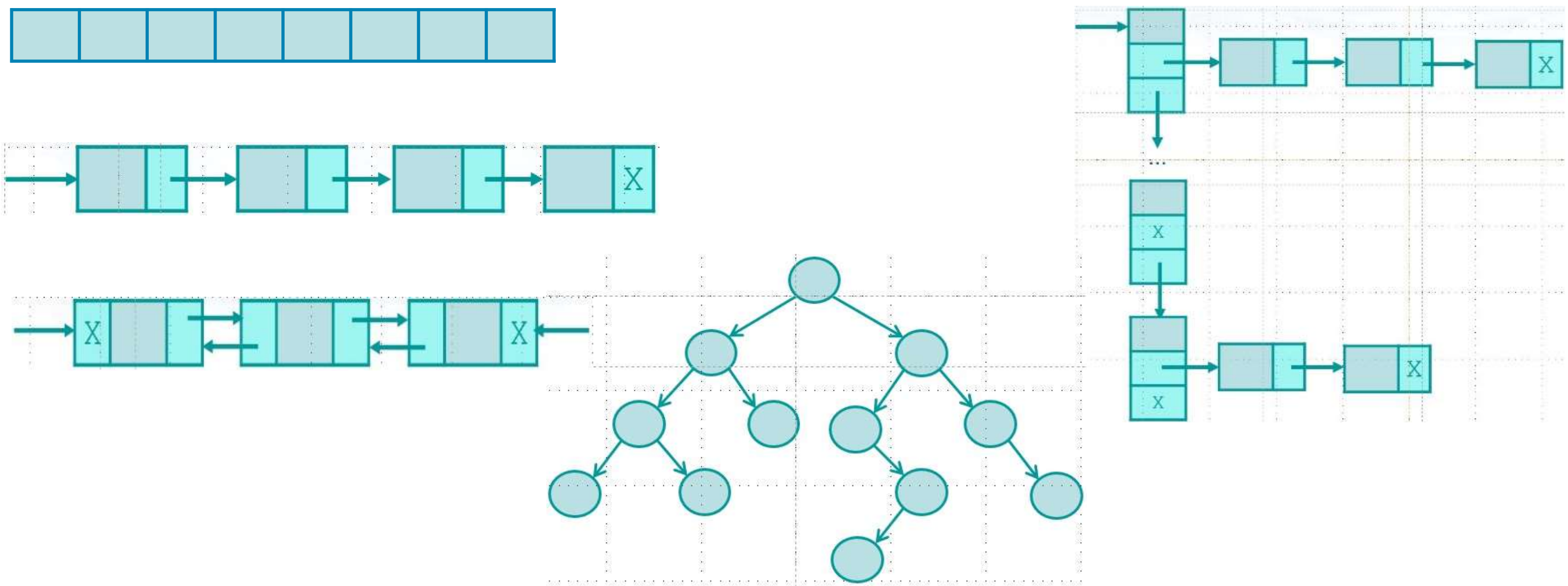
Politecnico di Torino

License Information

Premises

- For an introductions on basic data structures
  - Arrays, matrices, lists, stacks, queues, trees, heaps, hash-tables, etc.

and related algorithms, please see courses of "Algorithms and Data Structures"



C-like containers

- In C++ we can use C-like arrays
  - We can use **integer** (float) arrays
  - **Character** arrays are define similarly
  - **Strings** are a special type of character arrays
  - **Pointers** can be used to manage all sort of C-like arrays
  - We can define **multidimentional** arrays (matrices) as arrays of arrays

Let us analyze these throughout some examples

constexpr:
Think of constexpr as something you can figure out before you start running your program, like calculating 2 + 2 before you start your math homework.
When you declare something as constexpr, you're saying "Hey compiler, I know this value right now, so you can use it while you're building my program."
Example:
constexpr unsigned N1 = 10;
Here, N1 is known to be 10 even before the program starts running. It's like a pre-determined value that the compiler knows about.
const:
Think of const as something you know when your program is already running, like finding out the number of people in a room after you've entered it.
When you declare something as const, you're saying "Hey program, this value won't change while you're running."
const unsigned N2 = 3;
Here, N2 is set to 3, but the program figures this out while it's running. It's like a value that's decided upon after your program has started.
So, in simple terms, constexpr is about knowing things before the program starts running, while const is about knowing things while the program is already running.

Examples

- C-like integer (float) arrays

Defines a compile-time object

Defines a value that cannot be changed

```
constexpr unsigned N1 = 10;

const unsigned N2 = 3;

int v1[N1];                   // Empty array of ten integers
int v2[ ] = {1,2,3,4,5};      // Explicit initialization
int v3[N1] = {1,2,3,4,5};     // Equivalent to
                              // {1,2,3,4,5,0,0,0,0,0}
//    int v6[]; // this will raise an error

int v4[N2] = {1,2,3,4,5};     // Error: Too many initializers
int v5[5] = {1,2,3,4,5};      // OK
```

// don't need to mention the size it will be calculated on its own

Examples

- C-like character arrays
  - Arrays of characters are equivalent to arrays of integers or floats

```
char s1[] = {‘C’,‘+’,‘+’};        // List initialization

char s2[] = {‘C’,‘+’,‘+’,‘\0’};   // List  initialization,
                                  // explicit NULL

char s3[] = “C++”;                // Same as before, but
                                  // NULL added automatically,
                                  // i.e., a C string
```

2nd equaltion with 0\ This line initializes another character array s2 with the same characters as s1, but it adds a null terminator '\0' explicitly at the end.
The null terminator '\0' is a special character used in C-style strings to mark the end of the string.
By adding '\0' explicitly, s2 becomes a valid C-style string because it's terminated with a null character.
The size of s2 will be 4 to accommodate the additional null terminator.

This line initializes yet another character array s3 with the string literal "C++".
In C++, when you initialize a character array with a string literal enclosed in double quotes, the compiler automatically adds a null terminator '\0' at the end of the string.
So, s3 will also be a valid C-style string, terminated with '\0'.
The size of s3 will be 4 to accommodate the characters 'C', '+', '+', and the null terminator '\0'

Examples

The library function use the '\0' to perform its duty

- C-like strings
  - C-like strings are not a type
  - They are arrays of characters, NULL terminated

```
char s1[] = {'C','+','+'};        // List initilization
cout << s1;                       // Error

char s2[] = "C--";
char s3[] = "C++";                // NULL terminated
cout << s2;                       // Correct

s2 = s3;                 // Error: Cannot copy strings
                         // Must use strcpy

if (s2==s3) ...          // Warning: It does not compare strings
                         // (must use strcmp)
                         // it compares unrelated addresses
```

When you attempt to print s1 using cout, it will keep printing characters until it encounters a null terminator. However, since s1 doesn't have a null terminator explicitly provided, cout will continue reading memory past the end of s1 until it finds a null terminator, causing undefined behavior.

When you use double quotes " " to enclose a string literal, C++ automatically adds a null terminator '\0' at the end of the string.
So, s2 will be initialized as {'C', '-', '-', '\0'}.
When you print s2 using cout, it will correctly print the characters 'C', '-', and '-', and then stop when it encounters the null terminator '\0'.

Examples

- C-like pointers are closely intertwined with arrays

```
int v[10];          // Array of ten integers
int *p, *b, *e;

p = &v[0];          // The pointer points to element 0

// Pointer have a pointer arithmetic
// Pointers are iterators
b = &v[0];
e = &v[10];
for (p=b; p<e; p++)
  cout << *p << endl;


p = v;              // Equivalent to p=&v[0],
                    // p points to element v[0]

int *p2 = p+4;      // p2 points to element v[4]
                    // (If it exists)
```

More on iterators at the end of this unit

Examples

- Multidimensional arrays
  - In C (C++) there are not multidimensional arrays
  - They are implemented as arrays of arrays

```
int m1[3][4];          // Uninitialized 2D matrix

int m2[3][4] = {{0, 1, 2, 3},
                {4, 5, 6, 7},
                {8, 9, 10, 11}};    // Initialized 2D matrix

int m3[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    // Equivalent initialization (due to row-major order,
    // i.e., matrix are stored row-by-row)
```

Examples

- Multidimensional arrays and range for

```
constexpr int R = 3;
constexpr int C 0 4;

int m[R][C];           // Uninitialized 2D matrix

// Standard nested loops
for (int i=0; i<R; i++) {
  for (int j=0; j<C; j++) {
    cin >> m[i][j];
  }
}

// Range nested loops
for (auto &r: m) {     // For every element in the outer array
  for (auro &c: r) {  // For every element in the inner array
    cin >> c;
  }
}
```

We need reference because we need to modify the element.
Anything else?

Examples

- Multidimensional arrays and range for

```
// Range nested loops
for (auto &r: m) {
   for (auro &c: r) {
      cin >> c;
   }
}
// Buggy range nested loops
for (auto r: m) {
   for (auro c: r) {
      cout << c;
   }
}
// Range nested loops
for (auto &r: m) {
   for (auro c: r) {
      cout << c;
   }
}
```

We need reference because we need to modify the element. Anything else?

Even if we do not modify the matrix, this code does not work; r is no a reference is an element; c cannot iterate over an element

This is OK

C++ containers

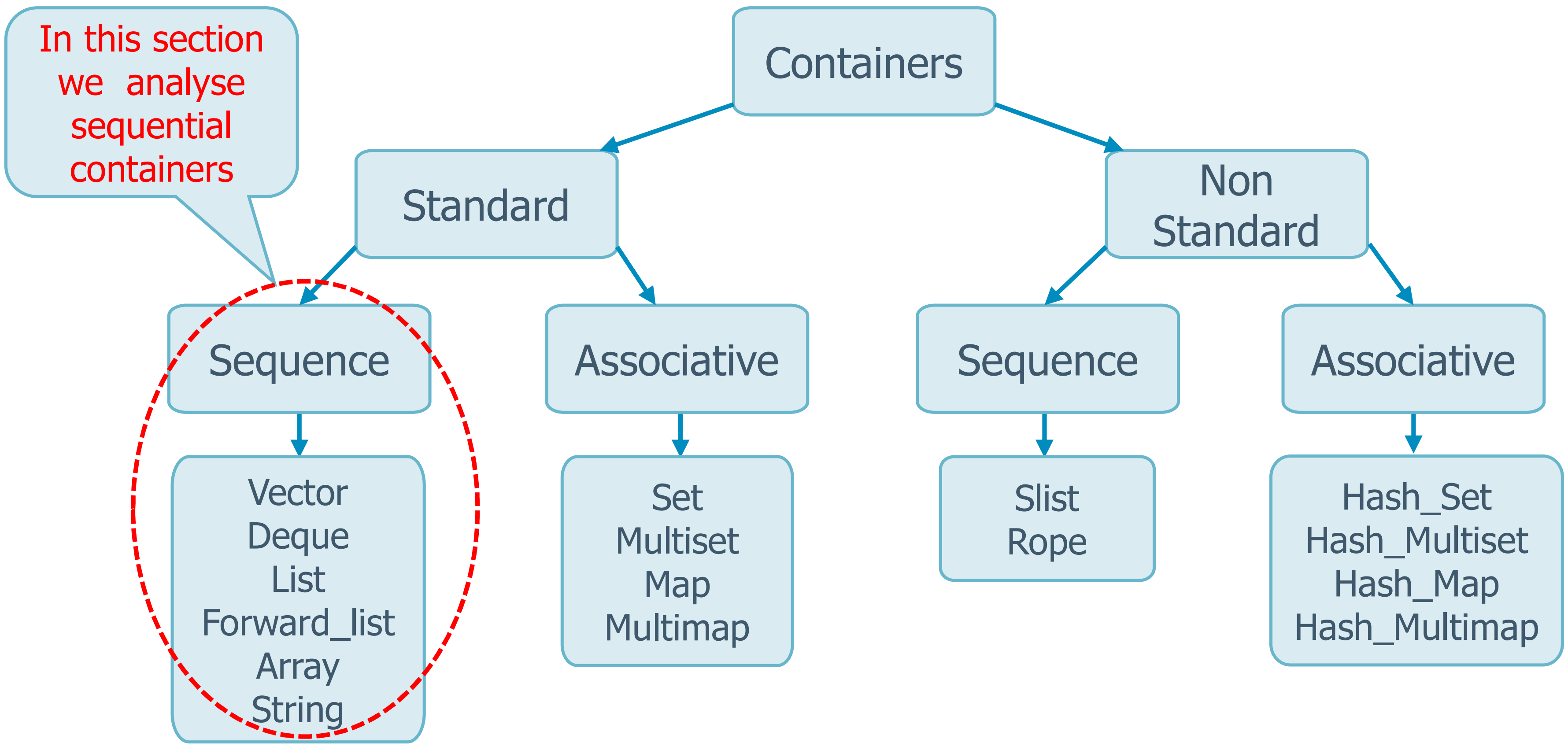
- A container is an object that
  - **Stores** (contains) other objects
    - Almost any type can be used as the element type of a sequential container
  - Manages the **storage space**
    - Elements are of generic type
    - Generic types in C++ are represented as **templates**

More on templates in section u04s07

  - Provides **member functions** for accessing elements
    - Access can be performed directly or through iterators
    - Member functions are often shared among different containers
  - Guarantee the **complexity** of all operations

C++ containers

- Containers organize their elements differently
  - Many operations are available on all of them
  - The efficiency varies



Sequential containers

- Sequential containers provide fast sequential access to their element
- They offer different performance to
  - Add or delete an element
  - Perform non-sequential access
  - With the exception of **array** (fixed size), they provide efficient and flexible memory management
    - We can add or remove elements and the container grows or shrinks
    - The strategy used to store elements influences the efficiency of the operations

Sequential containers

- Standard sequential containers available in C++

Type	Main characteristics
vector	Flexible-size array. Fast random access. Fast insert and delete at the back, slow elsewhere.
string	Similar to vector, specialized for characters. Fast random access. Fast insert and delete at the back.
list	Doubly-linked list. Bidirectional sequential access. Fast insert and delete in any position.
forwad_list	Singly-linked list. Sequential access in one direction. Fast insert and delete in any position.
deque	Double-ended queue. Fast random access. Fast insert and delete at front or back.
array	Fixed-size array. Fast random access. Cannot add or remove elements.

Sequential containers

Type
vector
string
list
forwad_list
deque
array

- Vectors and strings
  - Hold their elements in contiguous memory cells
  - Fast access given an index
  - It is expensive to add or remove elements in the middle
    - We need to move many elements to maintain contiguity
  - Adding a new element may require re-allocation to a new storage area

What is random accesss? Random access refers to the ability to be able to access elements in a data structure directly, without needing to iterate through the preceding elements.
In the context of containers like arrays, vectors, and strings, random access means that you can access any element in the container directly by specifying its index or its position. This is typically achieved using subscript notation myVector[3]

Sequential containers

When we say that elements in a data structure are not stored contiguously in memory, it means that the memory allocated for each element is not adjacent to the memory allocated for the next element.
consider a simple array of integers: int arr[4] = {10, 20, 30, 40};
stored in mem like: | 10 | 20 | 30 | 40 |
stored one after the other.

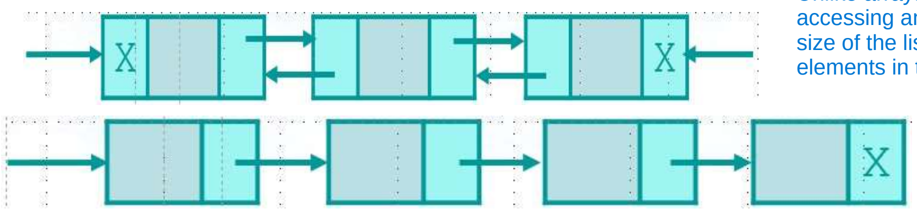
For example, if you have an array of integers where each integer takes up 4 bytes of memory, and you want to access the element at index 3, you can calculate its memory address like this:
Memory address of element at index 3 = Address of first element + (Index * Size of each element)
If the addr of first element is 1000 and element takes up to 4 bytes
Memory address of element at index 3 = 1000 + (3 * 4) = 1012

Type
vector
string
list
forwad_list
deque
array

Lists and forward_lists

- It is efficient to add or remove an element anywhere
  - Forward_list has been added with the newer standards
- They do not support random access
  - We need to iterate through the container to access elements
- The memory overhead is substantial when compared to the other containers

Each element (or node) in the list contains a reference (or pointer) to the next element in the sequence. Unlike arrays, where elements are stored one after another in contiguous memory, elements in a linked list can be scattered throughout memory, connected only by these references. Hence why they don't support random access and are not stored in contiguously.

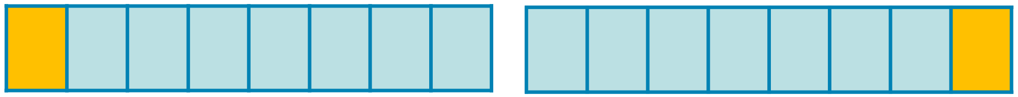


Unlike arrays, where accessing an element by index takes constant time O(1), accessing an element in a linked list takes linear time O(n) proportional to the size of the list. This is because you have to traverse through potentially all the elements in the list to reach the desired element

Sequential containers

Type
vector
string
list
forwad_list
deque
array

- Deques
  - Are more complicated
  - Like strings and vectors
    - Support random access
    - It is expensive to add and remove elements in the middle
  - Are fast for adding or removing elements at either end of the deque
    - The memory is not contiguous
    - Typical implementation use a sequence of individually allocated fixed size arrays



Sequential containers

Type
vector
string
list
forwad_list
deque
array

- Arrays
  - Are similar to vector
    - Are an alternative to C-like array
  - Have a **fixed size**
    - Do not support operations to add, remove, or resize a container
  - Have been added with C++11 for efficiency reasons
    - Adding elements is impossibile, thus there is no penalty for reallocations
    - It is possibile to use a C-like notation (and use pointers)

Selecting a container

- Use containers whenever possible, and when you need
  - Small elements and memory matter, don't use a **list** or **forward_list**
  - Random access use a **vector** or a **deque**
  - To insert or delete elements in the middle of the container, use **list** or **forward_list**
  - To insert or delete elements at the front and the back (but not in the middle) use a **deque**
- When in doubts, use a **vector**
  - Strings have a specific use (to store sequences of characters)

Definitions

- Don't forget to include the right header
  - vector, deque, list, etc.
- To create a new container it is possible to
  - Use the default constructor, make a copy of another container, use a list initializer

The copy constructor is a special type of constructor in C++ that is used to create a new object as a copy of an existing object of the same type. It's invoked when an object is initialized as a copy of another object, or when an object is passed by value to a function.

```
class MyClass {
public:
    // Copy constructor declaration
    MyClass(const MyClass& other);
};
```

The copy constructor is used to create a deep copy of an object, meaning that a new object with its own separate memory is created, and the contents of the original object are duplicated into the new object.

```
vector<int> v;
 vector<int> v8(v); // v8 is a vector of integers that is a copy of v.
```
Any changes in v8 will not be reflected in v because it's just a copy.

C is the container type

c is the object

Operation	Meaning
C c;	Default constructor; c is empty (but for arrays).
C c1(c2); C c1=c2;	Copy constructor: c1 is a copy of c2. They must have the same type.
C c{a,b,c…}; C c={a,b,c,…};	List constructor. Types must be compatibles. For array the list must have the same or fewer number of elements.
C c(b,e);	Container c is a copy of the element denoted by the iterators b and e.

Examples

Don't forget the right header files

```
#include <vector>
#include <list>

...

using std::vector;
using std::list
...
```

Containers!
Thus, they can contain different element types

```
vector<int> v1(10);        // 10 values equal to 0
vector<int> v2{10};        // One element with value 10
vector<int> v3(10, 1);     // 10 values equal to 1
vector<int> v4{10, 1};     // Two elements with values 10 and 1


vector<string> vs1{"a", "b", "d"};
vector<string> vs2{"a", "b", "d"};      // List initialization
vector<string> vs3("a", "b", "d");      // Error
```

Examples

```
list<float> myconst = {3.14159, 2.71828, 9.80655};
list<string> authors = { "Leopardi", "Manzoni", "Verga"};


deque<string> sd(10);          // 10 empty strings

array<int, 10> ia;             // 10 integer (default initializer)

array<int, 3> a1{ {1, 2, 3} };
    // Double-braces required before C++11
array<int, 10> a2 = {1,2,3};
    // double braces not required after C++11
    // 3 integers list intializer with he values 1, 2, 3


array<string, 2> a3 = { string("a"), "b" };


vector<vector<int>> vvi = {{1,2,3},{4,5,6}};
vector<vector<string>> vvs = {{ "one","two"}
                              { "three","four"}};
```

A container can include another containter

Assignment

- Assignment related operators act on the entire container
  - All elements are replaced with copies of the elements from the right-hand operand

Operation	Meaning
c1=c2;	Replace elements of c1 with elements of c2. c1 and c2 must have the same type.
c={a,b, …};	Replace the element of c with a copy of the elements of the list.
swap(c1,c2) c1.swap(c2)	Exchange elements in c1 with those in c2.

Examples

```
vector<string> vs1(10);       // vector with 10 elements
vector<string> vs2(20);       // vector with 20 elements

swap(sv1,sv2);                // Now sv1 containes 20 elements
                              // and sv2 10
```

Possible but with arrays
It is supposed to be fast:
Elements are not swapped;
internal structure is swapped

Vector Internals:
Each vector has three main components:
Pointer to the beginning of the allocated memory block (usually denoted as begin).
Pointer to the end of the used portion of the allocated memory block (usually denoted as end).
Pointer to the end of the allocated memory block (usually denoted as capacity).
Swapping Pointers:
When you call swap on two vectors, the begin, end, and capacity pointers of the two vectors are exchanged.
This means that vs1 starts pointing to the memory block of vs2, and vice versa.
However, the actual memory blocks remain unchanged.
Efficiency:
Swapping internal pointers is extremely fast compared to copying elements.
It's a constant-time operation, regardless of the size of the vectors, because it only involves exchanging a few pointers.
Effect on Size and Capacity:

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

if (v1 == v4) ...      // True
if (v1 < v2) ...       // True
if (v2 < v3) ...       // True
```

Adding elements

- All containers (but arrays) provide flexible memory management
  - We can add or remove elements at run time

only push_back is available in std::vector

In a std::vector in C++, the push_back method inserts a new element at the end of the vector, after its current last element.

Operation	Meaning
c.push_back(t) c.emplace_back(args)	Creates an element at the end of c (with value t or created from args). so after push back of 2 in 1,3,4 => 1,3,4,2
c.push_front(t) c.emplace_front(args)	Creates an element an the front of c (with value t or created from args). so after push front of 2 in 1,3,4 => 2, 1,3,4
c.insert(n,t) c.emplace(n,args)	Creates an element in position n (with value t or created from args).
c.insert(it,t,n)	Insert n elements with value t before the element denoted by iterator it.
c.insert(it,b,e)	Insert the elements from iterator b to iterator e before the element denoted by iterator it.

std::deque: Both push_back and push_front are available. push_back adds a new element at the end of the deque, after its current last element. push_front inserts a new element at the beginning of the deque, before its current first element.

Examples

```
struct student_t {
  int rn;
  string last_name, first_name;
  int mark;
} myt;
vector<student_t> sv;

sv.push_back(student_t(123456, "Potter", "Harry", 28));

sv.emplace_back(123456, "Wisley", "Ronald", 26);

sv.push_back(123457, "Granger", "Hermione", 30);
```

Correct:
Construct a student_t object and insert it into sv

"emplace" does that atutomatically

Error:
There is no version of push_back with 3 argouments

Examples

```
list<string> sl;
string word;

while (cin >> word)
  sl.push_back (word);

while (cin >> word)
  sl.push_front (word);

sl.insert(sl.begin(), "Start");
```

List, forward_list, and deque support analogous operation in front of the data stucture

begin() is an iterator. Equivalent to push_front (insert an element before begin)

```
vector<string> vs;

sl.insert(vs.begin(), "Start");
```

There is no push_front on array; thus, we can insert before begin(); however, it is slow on vectors !!!

Accessing elements

- Access operations are usually undefined when the container is empty
  - Each container has a front element
  - Each container, but forward_list, has a back member

Operation	Meaning
c.back()	Return a reference to the last element in c. Undefined if c is empty.
c.front()	Return a reference to the first element in c. Undefined if c is empty.
c[n]	Return a reference to the element indexed by n in c. Undefined if n>=c.size().
c.at(n)	Return a reference to the element indexed by n in c. If n is out of range, throws an exception.

Examples

```
vector<string> sv;

cout << sv[0];                  // Run time error: No element

cout << sv.at[0];               // Throws an exception
                                // (out_of_range)
```

```
deque<int> id;

...

if (!id.empty()) {
  id.front() = 10;              // assign 10 to the first element
  auto &v1 = id.back();         // v1 is a reference
  v1 = 100;                     // change element value

  auto v2 = id.back();          // v2 s a copy
  v2 = 1000;                    // does not change the value
                                // of the element in c
}
```

Erasing elements

- As is it possible to add elements, it is also possibile to remove them
  - Pop operations remove the first or last element
  - Erase operates on specific elements

Operation	Meaning
c.pop_back()	Remove last element in c. Undefined if c is empty. Returns void.
c.pop_front()	Remove first element in c. Undefined if c is empty. Returns void.
c.erase(it)	Remove element denoted by the iterator it.
c.erase(b,e)	Remove all elements from the iterators b ato the iterator e.
c.clear()	Remove all element in c.

Examples

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};

while (!lst.empty()) {
   ... Manipulate lst.front() ...
   lst.pop_front();
}
```

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();

while (it != lst.end()) {
  if (*it % 2 == 0)        // If the element is odd
    it = lst.erase(it);   // erate it
  else
    it++;                 // othewise move to the next element
}
```

begin() and end() are iterators.
See forward

Examples

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};

auto it = lst.begin();
auto it1=it+4;
auto it2=it+6;

it1 = lst.erase(it1, it2);
// Erase all elements from iterator it1 and iterator it2
// After the call it1==it2
```

It, it1, it2 are iterators.
See forward

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};

lst.clear ();                         // Erase all elements
lst.erase(lst.begin(), lst.end());    // Euaivalent instruction
```

Iterators

- We can use subscripts to access elements of a vector or a string
  - Subscripts are not general, i.e., they are not applicable to all other containers
- However, all containers support iterators
  - An iterator is an objects that can be thought of as pointer abstractions, i.e., it gives an direct access to the elements
  - The standard library defines multiple iterator types as containers have different capabilities
    - Random access, traversable in both directions, etc.

Iterators

- We can use iterators to
  - Access elements
  - Move from one element to another
- Iterators are returned by member functions, not by pointers
  - They are included in the **<iterator>** header

Iterators

- The standard library provides 5 iterator categories
  - Input, output, forward, bidirectional, random-access

Type	Iterator Type
vector	Random access
deque	Random access
list	Bidirectional
forwad_list	Forward
array	Random access
string	Random access

Iterator operations

Operation	Meaning
auto b=v.begin(); auto e=v.end();	b denotes the first element. e denotes one past the last element.
auto b=v.rbegin(); auto e=v.rend();	Reverse interators: From one element past the last element to the first one.
*b	Reference to the element denoted by b.
b->mem	Fetch the member mem referenced by b, equivalent to (*b).mem
++b --e	b (e) points to the next (previous) element
b+n e-n	Move the iterator b (e) to denote n elements forward (backward) within the container (possibly outside)
b != e b==e	Compare iterators

Examples

Iterators and strings

```
string s("this is a string");
if (s.begin() != s.end()) {        // Make sure s is not empty
  auto it = s.begin();
  *it = tupper(*it);               // Make first character
                                   // uppercase
}
// Make all characters uppercase
for (auto it = s.begin(); it!=s.end(); it++)
  *it = toupper(*it);
```

Iterators and lists

```
list<int> l={0,1,2,3,4,5,6,7,8,9};

auto it=l.begin();
while (it!=l.end()) {
  if (*it % 2)
    it = l.erase(it);       // Erase odd elements in the list
  else
    ++it;
}
```

No ++it !!!

Examples

- When we modify a container, an existing iterator may become invalid
  - For example, if we add add an element, the existing iterator may be invalidated and mujst be used with care

```
vector<int> v= {0,1,2,3,4,5,6,7,8,9};
auto it = v.begin();

while (it != v.end()) {
  if (*it %2) {
    it = v.insert(it, *it);
    it += 2;
  } else {
    it = v.erase(it);
  }
}
```

Insert, insert before and set it to the added element; thus, after insert it must be incremented by 2

Erase, cancel the element; thus, after erase there is no need to increment it

Examples

Iterators and vectors of strings

```
vector<std::string> v = {"one", "two", "three", "four"};

vector<std::string>::iterator it = v.begin();
vector<std::string>::const_iterator itc = v.begin();
```

A **const_interator** can be used to read not to write an element

Iterators have iterator type

Examples

Iterators and vectors of strings

```
vector<std::string> v = {"one", "two", "three", "four"};

vector<std::string>::iterator it = v.begin();
auto end = v.end();




if (!(*it).empty())   // Checks whether the string is empty
  ...
if (!(*it.empty()))   // Error
  ...
```

It tries to access the member empty() of it, but it is an iterator and does not have a member emoty()

Examples

Iterators and vectors of strings

```
vector<std::string> v = {"one", "two", "three", "four"};

vector<std::string>::iterator it = v.begin();
auto end = v.end();




cout << *it;                    // prints "one"
cout << *end;                   // undefined behavior

++it;                           // Prefer to use pre-increment
std::cout << *it;               // prints "two”

// To print "three,four,"
while (it != end) {
  std::cout << *it << ",";
  it++;
}
```

Examples

Iterators and vectors of strings

```
std::vector<std::string> v = {"one","two","three","four"};

for (auto it = v.begin(); it != v.end(); ++it) {
  if (it->size == 3) {
    it = v.insert(it,"foo");
    // it now points to the newly inserted element
    ++it;
  }
}
```

v is now {"foo", "one", "foo", "two", "three", "four"}

```
for (it = v.begin(); it != v.end(); ++it) {
  if (it->size == 3) {
    it = v.erase(it);
    // erase returns a new, valid iterator
    // pointing to the next element
  }
}
```

v is now { "three", "four"}

More on … vectors

- Vectors are constructed just like arrays
- Vectors are
  - Defined in the header **<vector>**
  - Collections of contiguous objects of the same type
  - Arrays that can dynamically grow
- The memory is
  - Pre-allocated for a certain amount of elements
    - Can be reserved for a given amount of elements with reserve
  - Re-allocated when exhausted
    - Moved to a larger chunk of memory
    - All elements are copied

Expensive

More on … vectors

- Time complexity of the main operations
  - Random access
    - O(1)
  - Back insertion
    - Typically: O(1)
    - Worst case: O(n) due to possible reallocation

Vectors and strings typically allocate capacity beyond what it is immediately needed

Vectors and strings have methods: shink_to_fit(), capacity(), reserve() to optimize reallocation performances

  - Insertion and removal at any other position
    - O(n)

More on … vectors

- Initialization

T is the generic type

Operation	Meaning
`#include <vector>` `using std::vector;`	Include the appropriate header.
`vector<T> v;`	Default initialization; v is empty.
`vector<T> v2(v1);` `vector<T> v2=v1;`	Initialization by copying all elements of v1 into v2.
`vector<T> v{n};` `vector<T> v{n,val};`	Initialization with n values equal to the initialization value for that type or the value val.
`vector<T> v{a,b,c…};` `vector<T> v={a,b,c…};`	Esplicit initialization with a list initializer.

More on … vectors

- Management

Operation	Meaning
v.empty()	Return true if v is empty.
v.size()	Return the number of elements.
v.push_back(a)	Add value a to the end of v.
v[n]	Return a reference to element n.
v1=v2	Replace elements of v1 with a copy of the element of v2.
v1={a, b, c, …}	Replace elements of v1.
v1==v2 v1!=v2, v1<=v2 etc.	Normal comparison operation using dictionaly ordering.

Examples

Vector initialization

```
#include <vector>

using std::vector;

vector<int> v;              // Default initialization
                            // v has no elements

vector<int> v{10};          // one element with value 10

vector<int> v(10,1);       // ten element with value 1
vector<int> v{10,1};       // two elements with value 10 and 1
```

Access to specific elements

```
vector<int> fib = {1,1,2,3};  // values 1,1,2,3

if (fib[1]==1) ...            // True
fib[3] = 43;                  // fib is now 1,1,2,43

fib[4] = 12;                  // Error: There is no element 4
```

Examples

Dynamic creation

```
vector<int> v;
for (int i=0; i<100; i++)
  v.push_back(i);

for (auto i : v)
  cout << i << " ";     // print the element

for (auto &i : v)
  i *= i;               // square the element value
```

2D-Vector

```
vector<vector<int>> m;

for (int r=0; r<R; r++) {
  vector<int> tmp;
  for (int c=0; c<C; c++) {
    tmp.push_back(c);
  }
  m.push_back(tmp)
}
```

More on … strings

- A string is a variable-length sequence of characters
- Strings are
  - Defined in the header **<string>**
  - Are provided with additional operations compared to the ones available for the other containers

More on ... strings

- Initialization

Operation	Meaning
#include <string> using std::string;	Include the appropriate header.
string s;	Defaul initialization; s is the empty string.
string s2(s1); string s2=s1;	String s2 is defined and it is a copy of s1.
string s("value"); string s = "value";	String s is defined and it is a copy of the string literal "value".
string s(n,'c');	Define and initialize s with n copies of character 'c'.

More on … strings

- Management

Operation	Meaning
ostream << s	Write the string s on the output stream.
istream >> s	Read the string s from the input stream.
getline(istream, s)	Read an intere line into s from the input stream.
s.empty()	Return true if the string is empty.
s.size()	Return the number of characters.
s[i]	Refernce to character in position i (from zero).
s1+s2	Returns a string which is the concatenation of strings s1 and s2.
s1=s2	Replace s1 with a copy of s2.
s1==s2, s1!=s2, s1<s2, etc.	Comparison using dictionary ordering and case-sensitive.

More on ... strings

Operation	Meaning
string s2(s1,pos);	String s1 is a copy of the characters of string s2 starting at index pos.
s.substr(pos,n)	Return a string containing n characters from s starting at pos.
s.insert(pos,args)	Insert characters args before (position or iterator) pos. Args can be a string, a triple (string, pos, len), etc.
s.erase(pos,len)	Remove len characters starting at position pos.
s.assign(args)	Replace characters in s according to args (defined as before).
s.append(args)	Append args (defined as before) to s.
s.replace(range,args)	Remove range (index or a pair of iterators) of characters from s and replace them with the characters formed by args (defined as before).
s.find(args)	Find the occurrence args (defined in several ways) in s.

More on ... characters

- Dealing with characters in a string

c is a character in this case not a container

Operation	Meaning
isalnum(c)	True is c is a letter or a digit.
isalpha(c)	True is c is a letter.
iscntrl(c)	True is c is a control character.
isdigit(c)	True is c is a digit.
islower(c)	True is c is a lowercase letter.
ispunct(c)	True is c is a punctuation character.
isspace(c)	True is c is a space.
tolower(c)	If c is an uppercase letter, returns its lowercase equivalent; otherwise, returns c unchanged.
toupper(c)	If c is a lowercase letter, returns its uppercase equivalent; otherwise, returns c unchanged.

Examples

String initialization

```
#include <string>
using std::string;

string s1;
string s2 = "foo";
string s3(10, 'a');   // s3 is "aaaaaaaaaa"

s1 = s3;              // replace content of s1 with
                      // content of s3
```

String IO

Stop with:
ctrl-D (UNIX), ctrl-Z (Windows)

```
string word, line;

while (cin >> word)           // Input is broken by spaces
  cout << word << endl;

while (getline (cin, line))   // Read up to the newline
  if (!line.empty())
    // Display only lines that are not empty
    cout << line << endl;
```

Examples

String concatenations

```
string s1 = "hello";
string s2 = "word";

string s3 = s1+s2;                // String concatenation
                                  // "helloword"

// Adding literals to a string
string s4 = s1 + ' ' + s2 + '\n';    // "hello word"

s1 += s2;                         // String concatenation
                                  // "helloword"
```

The range-for statement of strings

```
string str("some string");

for (auto c : str)        // To access a string
  cout << c << endl;

for (auto &c : str)       // To modify a string
  c = toupper (c);
```

Examples

Using subscripts

```
// Process the entire string
for (decltype(str.size()) i=0; i!=str.size(); i++)
  str[i] = toupper (str[i]);

// Process a string until we hit a space
for (decltype(str.size()) i=0;
  i!=str.size() && !isspace(str[index]); i++)
  str[i] = toupper (str[i]);
```

Vectors of strings

```
string word;
vector<string> text;

while (cin >> word)
  text.push_back(word);
```

Examples

Push versus emplace

```
vector<my_type> c;

c.push_back(my_type("string", 12, 24.50));

// Use the constructor of my_type
c.emplace_back("string", 12, 24.50);

// Error
c.push_back("string", 12, 24.50);
```

Push back takes an object as an argument and adds a copy of that object to the end of the constructor. It's useful when you have an existing object and you want to add a copy of it to the container.
vector.push_back(num)

Emplace back constructs the new element in place at the end of the container, directly within the container's memory.
It takes the constructor arguments for the new element as arguments.
It's useful when you want to construct an object directly in the container without first creating a separate object.
std::vector<std::string> vec;
vec.emblace_back("hello")