Searching

Algoritmos e Estruturas de Dados,
L.EIC, 2021/2022

AP Rocha, P Ribeiro,
R Rossetti, F Ramos, L Grácio, A Costa, S Martins

The Search problem

Problem: given an array ***v*** storing ***n*** elements, and a target element ***el***, locate the position in ***v*** (if it exists) where *v[i] = el*

- variants for the case of arrays with repeated values:
  a) indicate the position of the first occurrence
  b) indicate the position of the last occurrence
  c) indicate the position of any occurrence

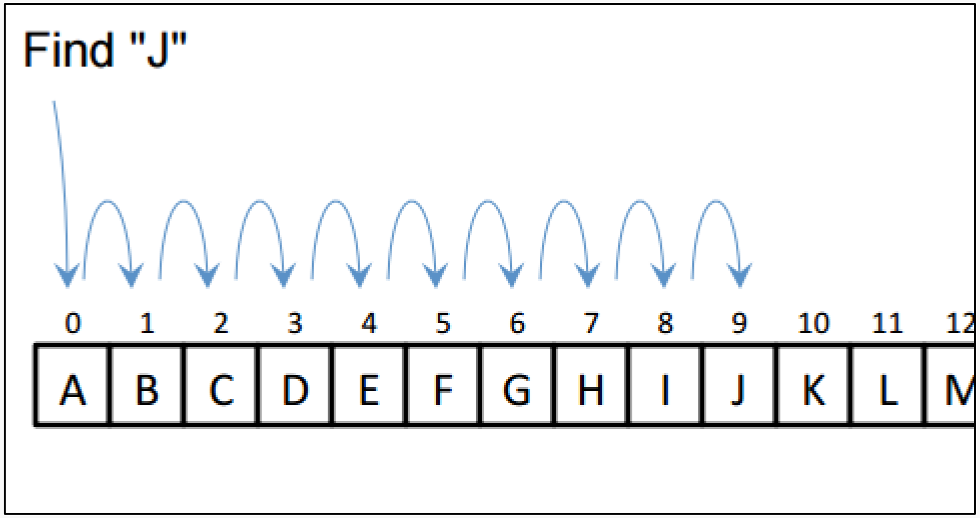
- when the target ***el*** does not exist, return an undefined position, such as *-1*

Sequential Search

- Algorithm *(sequential search)*

  sequentially checks each element of the array, from the first to the last [(a)] or from the last to the first [(b)], until a match is found or the end of the array is reached

  - [(a)] if you want to know the position of the first occurrence
  - [(b)] if you want to know the position of the last occurrence



suitable for unordered or small arrays

Sequential Search

variant a)

```
/* Search for an element el in a vector v of comparable
elements with the comparison operators. Returns the
index of the first occurrence of el in v, if found;
otherwise, returns -1 */

template <class T>
int SequentialSearch(const vector<T> &v, T el)
{
   for (unsigned i = 0; i < v.size(); i++)
      if (v[i] == el)
         return i; // found

   return -1; // not found
}
```

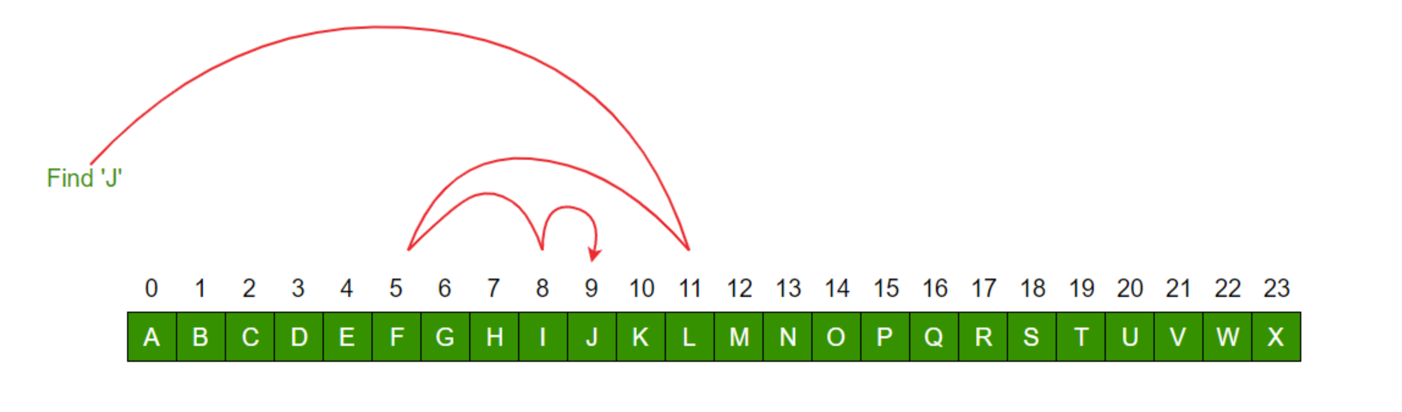
Sequential Search complexity

- Sequential Search time complexity
  - the operation performed most often is the test "`if (v[i] == el)`", at most **n+1** times (in case it doesn't find the target element).
  - if the target element exists in the vector, the test is performed approximately **n/2** times on average (1 time in the best case)
  - ***T(n) = O(n)*** in worst case and average case

- Sequential Search space complexity
  - space on local variables (including arguments)
  - since vectors are passed "by reference“, the space taken up by the local variables is constant and independent of the vector size.
  - ***S(n) = O(1)***

Searching in sorted arrays

- Suppose the array is ordered (arranged in increasing or non-decreasing order)
  - Sequential search on a sorted array still yields the same analysis *T(n) = O(n)*
  - Can exploit sorted structure by performing **binary search**
  - *Strategy:* inspect middle of the structure so that half of the structure is discarded at every step



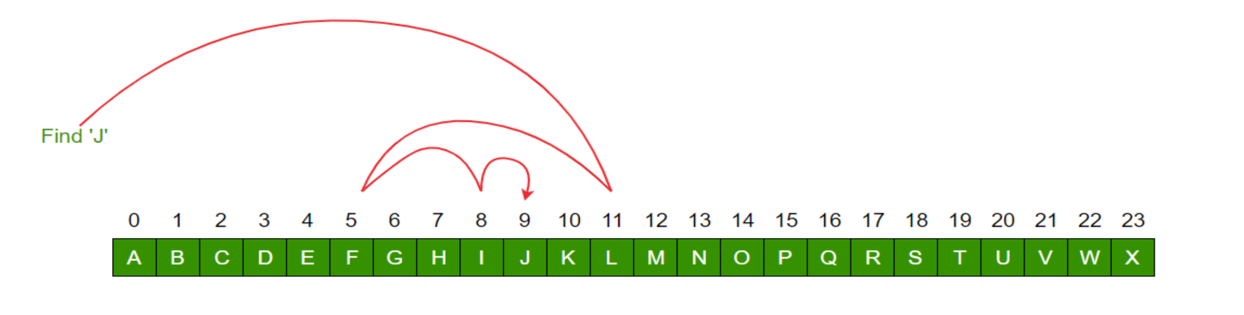
Binary Search

- Algorithm *(binary search)*

compares the element in the middle of the array with the target element:

- is equal to the target element → found
- is greater than the target element → continue searching (in the same way) in the sub-array to the left of the inspected position
- is less than the target element → continue searching (in the same way) in the sub-array to the right of the inspected position

if the sub-array to be inspected reduces to an empty vector, it is concluded that the target element does not exist

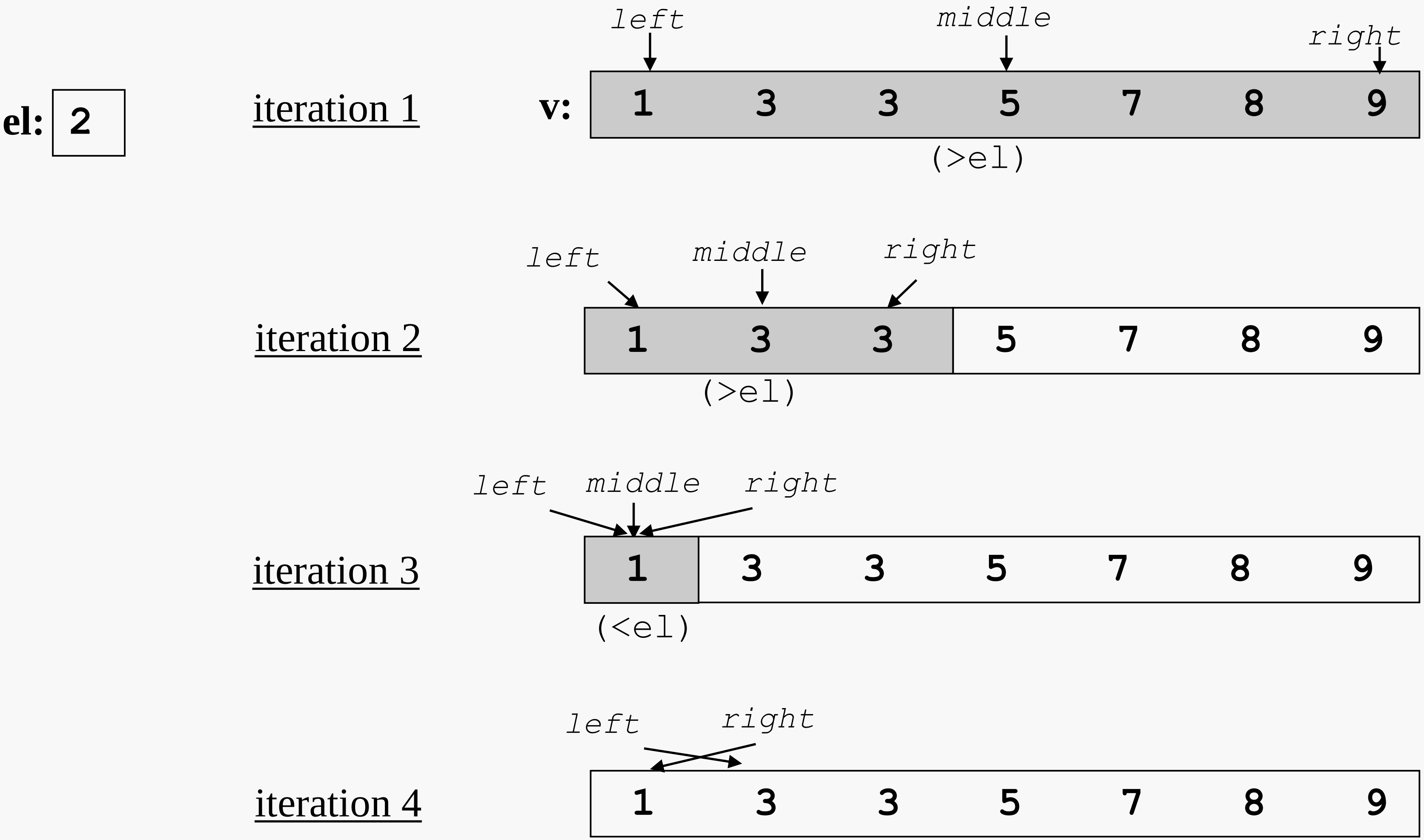


Binary Search

```
/* Search for an element el in an ordered vector v of comparable elements
with the comparison operators. Returns the index of one occurrence of el
in v, if found; otherwise returns -1. */

template <class T>
int BinarySearch(const vector<T> &v, T el)
{
    int left = 0, right = v.size() - 1;
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if (v[middle] < el)
           left = middle + 1;
        else if (el < v[middle])
           right = middle - 1;
        else
           return middle; // found
    }
    return -1; // not found
}
```

Binary Search



el: 2

iteration 1

left middle right

v: 1 3 3 5 7 8 9

(>el)

iteration 2

left middle right

1 3 3 5 7 8 9

(>el)

iteration 3

left middle right

1 3 3 5 7 8 9

(<el)

iteration 4

left right

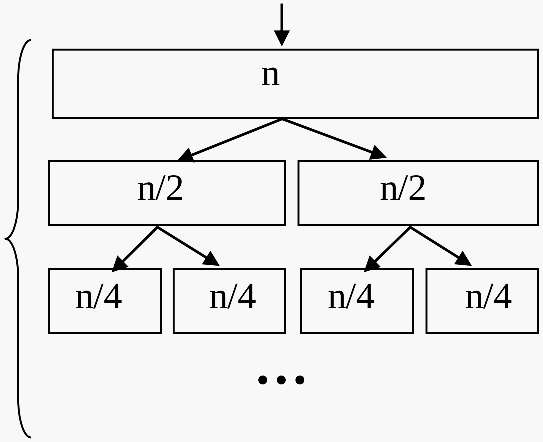
1 3 3 5 7 8 9

sub-array is empty → element 2 does not exist!

Binary Search complexity

- Binary Search time complexity
  - in each iteration, the size of the sub-vector to be analyzed is divided by ≈ 2
  - after **k** iterations, the size of the sub-vector to analyze is approximately **$n/2^k$**
  - if the target element does not exist in the vector, the cycle only ends when

$$n/2^k \approx 1 \;\rightarrow\; n \approx 2^k \;\rightarrow\; k \approx \log_2 n$$



  - so, in the worst case, the number of iterations is $k \approx \log_2 n$, ***T(n) = O(log n)***

- Binary Search space complexity

S(n) = O(1)

The Painter's Partition Problem: using binary search

Problem

- there are paint ***n*** boards of length ***{l*$_1$*, l*$_2$*…l*$_n$*}*** and there are **k** painters available
- each painter takes 1 unit of time to paint 1 unit of the board
- any painter will only paint continuous sections of boards
- the problem is to find the minimum time to get this job done

example: ***k*** = 2, ***board*** = [10, 20, 30, 40]

algorithm:

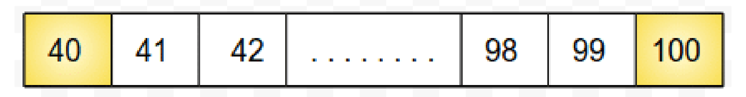
- apply binary search on the search space and
- according to the problem reduce the search space which will finally give the final result

The Painter’s Partition Problem: using binary search

example: ***k*** = 2, ***board*** = [10, 20, 30, 40]

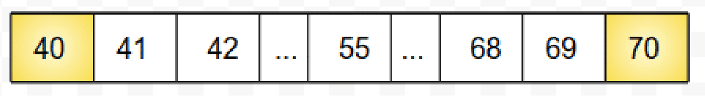
– Search space is the maximum range where the answer contains:
  - the maximum time will be (10+20+30+40) = 100
  - the minimum time will be 40

– the search space will be [40 – 100]



40	41	42	. . . .	98	99	100

– divide the search space, *middle* = 40 + (100 - 40) / 2 = 70



40	41	42	...	55	...	68	69	70

  - assume that no painter will paint more than 70 units of the board
  - how many painters will be required? ***k****=2* is enough? yes, so the search space will be reduced and will change to [40, 70]

– …

The Painter's Partition Problem: using binary search

```
int partition(vector<int> &board, int k)
{
    int n = board.size(), s = 0, m = 0;
    for(int i = 0; i < n; i++)
    {
        m = max(m, board[i]);
        s += board[i];
    }

    int low = m, high = s;
    while (low < high)
    {
        int mid = low + (high - low) / 2;
        int painters = findkp(board, mid);

        if (painters <= k) high = mid;
        else low = mid + 1;
    }
    return low;
}
```

The Painter's Partition Problem: using binary search

```
int findkp(vector<int> &board, int atmost)
{
    int n = board.size();
    int s = 0, painters = 1;

    for (int i = 0; i < n; i++)
    {
        s += board[i];
        if (s > atmost)
        {
            s = board[i];
            painters++;
        }
    }
    return painters;
}
```

STL algorithms

- Sequential Search in vectors

  iterator find(iterator start, iterator end, const T& val);

  – looks for first occurrence of an element identical to *val* in *[start, end[* (comparison performed by operator ==)

    - success, returns iterator for the found element
    - no success, returns iterator to “the end” of the vector (*v.end()*)

  iterator find_if(iterator start, iterator end, Predicate pred);

  – looks for first occurence for which unary predicate *pred* is true in *[start, end[*

STL algorithms

- Binary Search in vectors

bool binary_search(iterator start, iterator end, const T& val);

– looks for one occurrence of an element identical to *val* in *[start, end[*

– uses operator <

bool binary_search(iterator start, iterator end, const T& val, Compare comp);

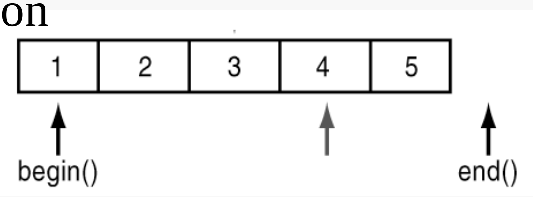
– looks for one occurrence of an element identical to *val* in *[start, end[*

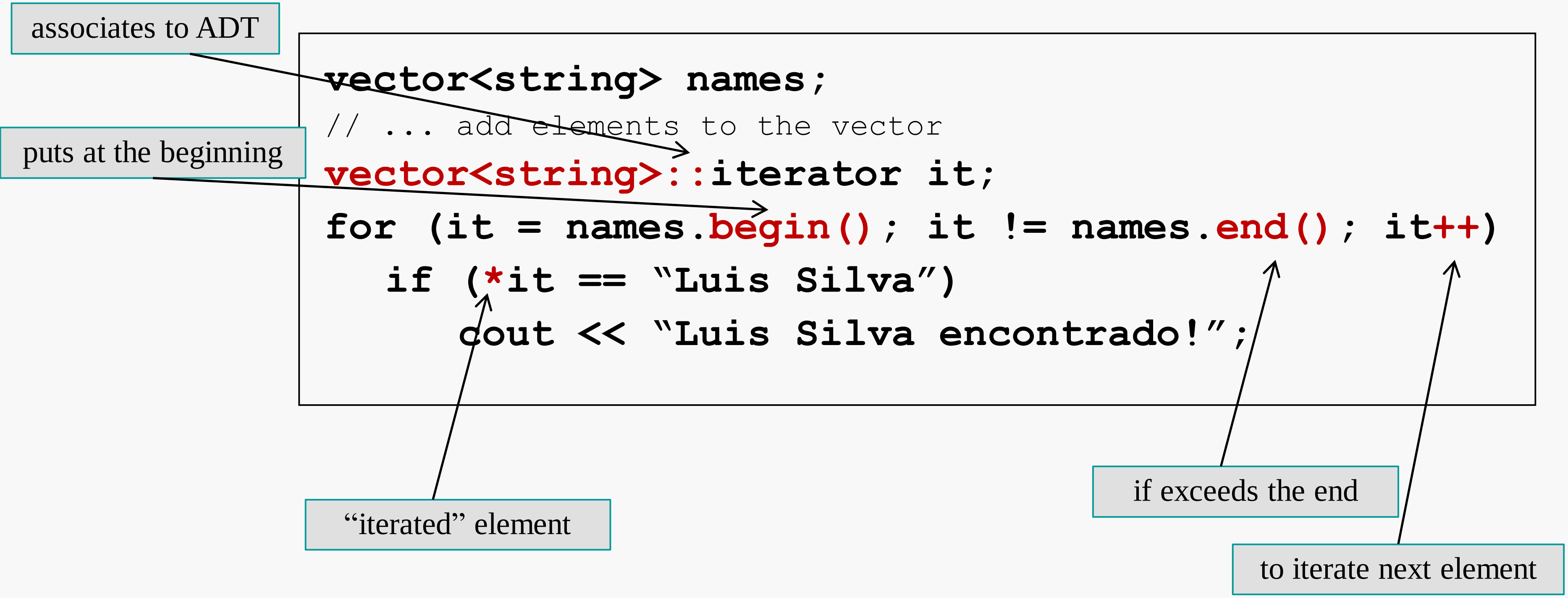
– uses predicate *comp* (*comp* compares two elements)

* Iterators: some notes

Iterator

- associates to an Abstract Data Type or its implementation
- example of using vector iterators
  - consider the vector ***names*** (vector of strings)
  - search for the name "Luis Silva" in the vector ***names***





```
vector<string> names;
// ... add elements to the vector
vector<string>::iterator it;
for (it = names.begin(); it != names.end(); it++)
   if (*it == "Luis Silva")
      cout << "Luis Silva encontrado!";
```

associates to ADT

puts at the beginning

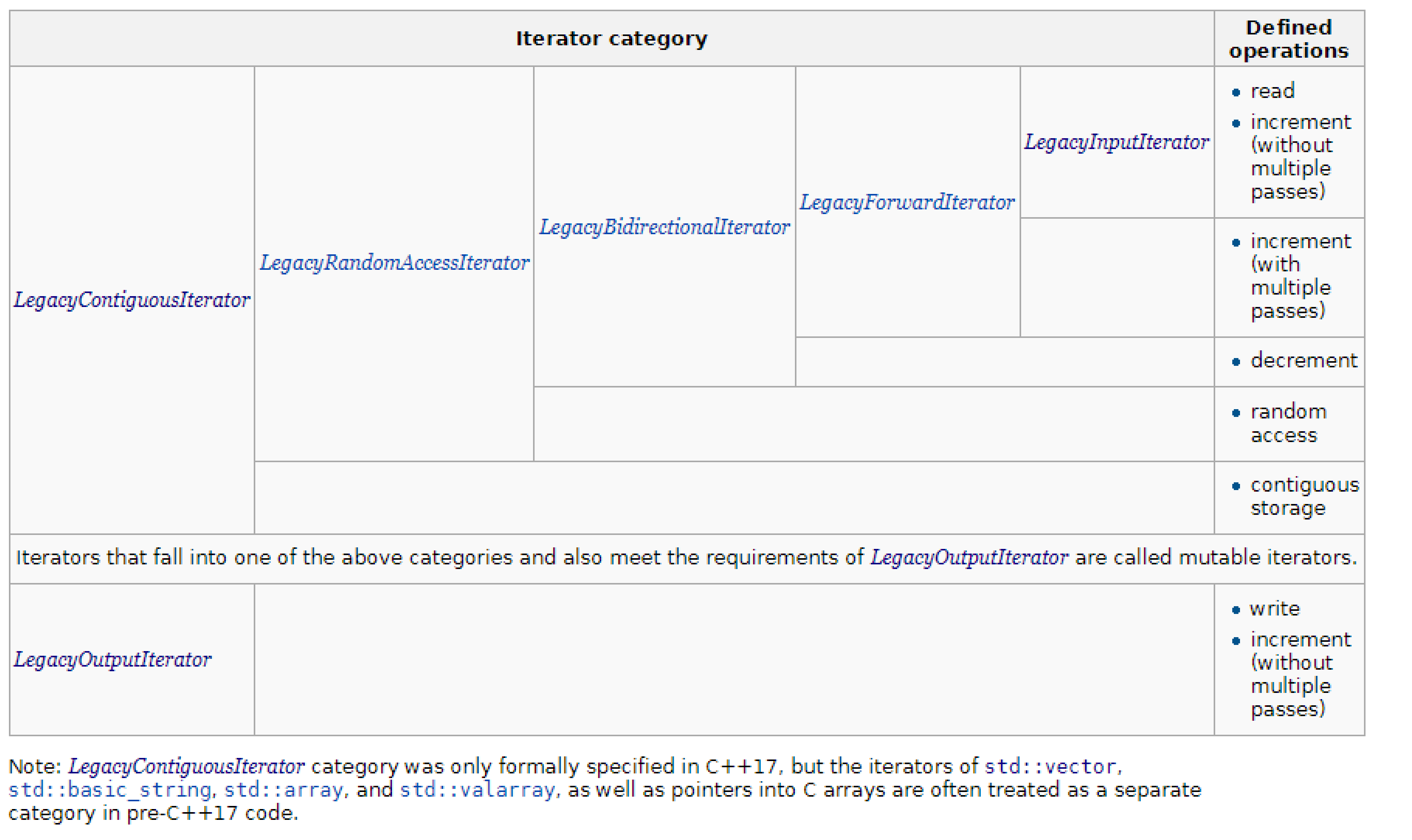
"iterated" element

if exceeds the end

to iterate next element

* Iterators: some notes

- more information about iterators in C++ STL
  - https://en.cppreference.com/w/cpp/iterator



Iterator category					Defined operations
LegacyContiguousIterator	*LegacyRandomAccessIterator*	*LegacyBidirectionalIterator*	*LegacyForwardIterator*	*LegacyInputIterator*	read increment (without multiple passes)
					increment (with multiple passes)
					decrement
					random access
					contiguous storage
Iterators that fall into one of the above categories and also meet the requirements of *LegacyOutputIterator* are called mutable iterators.					
LegacyOutputIterator					write increment (without multiple passes)

Note: *LegacyContiguousIterator* category was only formally specified in C++17, but the iterators of `std::vector`, `std::basic_string`, `std::array`, and `std::valarray`, as well as pointers into C arrays are often treated as a separate category in pre-C++17 code.

Example

```
class Person {
   string cc;
   string name;
   int age;
public:
   Person (string c, string nm="", int a=0);
   string getCC() const;
   string getName() const;
   int getAge () const;
   bool operator < (const Person & p2) const;
   bool operator == (const Person & p2) const;
};

Person::Person(string c, string nm, int a):
       cc(c),name(nm), age(a) {}

string Person::getCC() const { return cc; }
string Person ::getName() const { return name; }
int Person ::getAge() const { return age; }
```

Example

```
bool Person::operator < (const Person & p2) const
{
   return name < p2.name;
}

bool Person::operator == (const Person & p2) const
{
   return cc == p2.cc;
}


ostream & operator << (ostream &os, const Person & p)
{
   os << "cc: " << p.getCC() << ", name: " << p.getName()
                 << ", age: " << p.getAge() ;
   return os;
}
```

Example

```
template <class T> void writeVector(vector<T> &v)
{
   for (val:v)
      cout << val << endl;
   cout << endl;
}


bool isTeenager(const Person &p1)
{
   return p1.getAge() <= 20;
}


bool younger(const Person &p1, const Person &p2)
{
   if (p1.getAge() < p2.getAge()) return true;
   else return false;
}
```

Example

```
int main()
{
  vector<Person> vp;
  vp.push_back(Person("6666666","Rui Silva",34));
  vp.push_back(Person("7777777","Antonio Matos",24));
  vp.push_back(Person("1234567","Maria Barros",20));
  vp.push_back(Person ("7654321","Carlos Sousa",18));
  vp.push_back(Person("3333333","Fernando Cardoso",33));

  cout << “initial vector:" << endl;
  writeVector(vp);
```

```
initial vector:
cc: 6666666, name: Rui Silva, age: 34
cc: 7777777, name: Antonio Matos, age: 24
cc: 1234567, name: Maria Barros, age: 20
cc: 7654321, name: Carlos Sousa, age: 18
cc: 3333333, name: Fernando Cardoso, age: 33
```

Example

```
Pessoa px("7654321");
vector<Person>::iterator it = find(vp.begin(),vp.end(),px);
if (it == vp.end())
   cout << px << " does not exist in vector" << endl;
else
   cout<< px << " exists in vector as:" << *it <<endl;
```

```
cc: 7654321, name: , age: 0 exists in vector as
      cc: 7654321, name: Carlos Sousa, age: 18
```

```
it = find_if(vp.begin(),vp.end(),isTeeenager);
if (it == vp.end())
  cout << "there is no teenager in the vector" << endl;
else
  cout << “teenager found " << *it << endl;
```

```
teenager found cc: 1234567, name: Maria Barros, age: 20
```

Example

```
// note that vector vp2 is sorted by age
vector<Person> vp2;
vp2.push_back(Person("7654321","Carlos Sousa",18));
vp2.push_back(Person("1234567","Maria Barros",20));
vp2.push_back(Person("7777777","Antonio Matos",24));
vp2.push_back(Person("3333333","Fernando Cardoso",33));
vp2.push_back(Person("6666666","Rui Silva",34));

Person py("xx","xx",24);
bool exist = binary_search(vp2.begin(),vp2.end(),py,younger);
if (exist == true)
   cout << “there is a person aged " << py.getIdade() << endl;
else
   cout << “there is no person aged ” << py.getIdade()<<  endl;
```

```
there is a person aged 24
```