Standard Template Library

STL Containers

Motivation

Structure p;

…… // store elements to p

How to sort a set of elements stored in a structure p?

Motivation

DataStructure p;

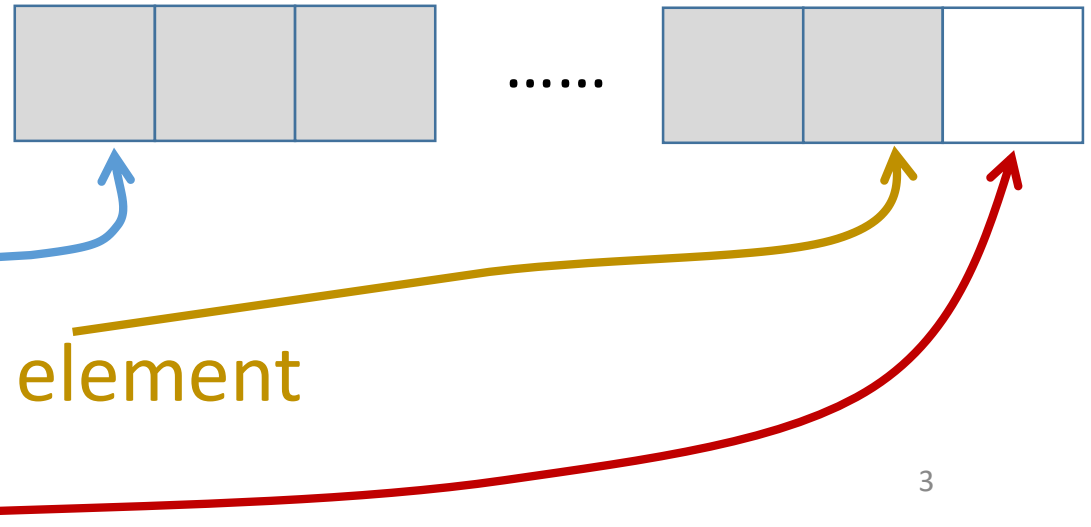
…… // store elements to p

How do we sort a set of elements stored in a data structure p?

sort(p.begin(), p.end(), compare)

;begin( ) : the first element

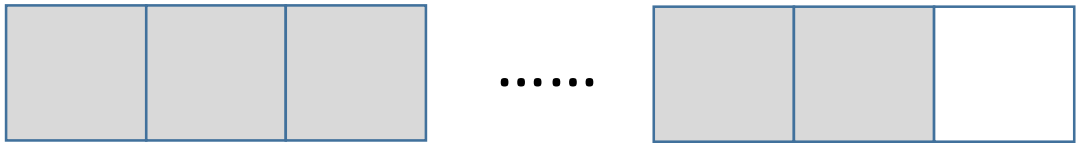
;end( ): the element next to the last element



Motivation

DataStructure p;

// store elements to p



Need to traverse the data structure to get the elements

How to sort a set of elements stored in a data structure p?

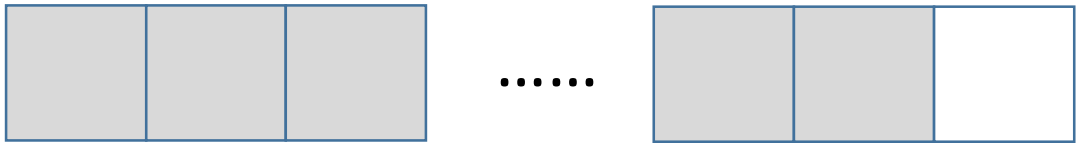
sort(p.begin(), p.end(), compare)

algorithm data Comparison function

Motivation

DataStructure p;

// store elements to p



Need to traverse the data structure to get the elements

How to **find** an element, **key**, in a data structure p?

find(p.begin(), p.end(), compare, **key**)

algorithm data Comparison function

Three components of STL

Container p;

……

find(p.begin(), p.end(), compare, **key**)

Three components of STL

```
Container p;
……
find( p.begin(), p.end(), compare, key)
```

The Standard Template Library (STL) has container classes.

Containers: They stores elements or a collection of data.

Examples: vector, list, map

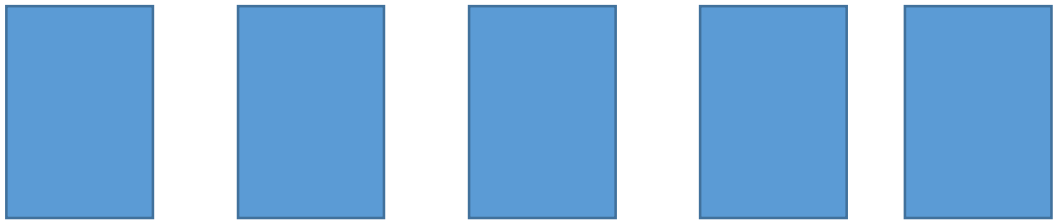
Iterators: They facilitate traversing through the elements in a container. They are useful for accessing and manipulating the elements.

Algorithms: They manipulate data such as sorting, searching, and comparing elements. Most of them use iterators.

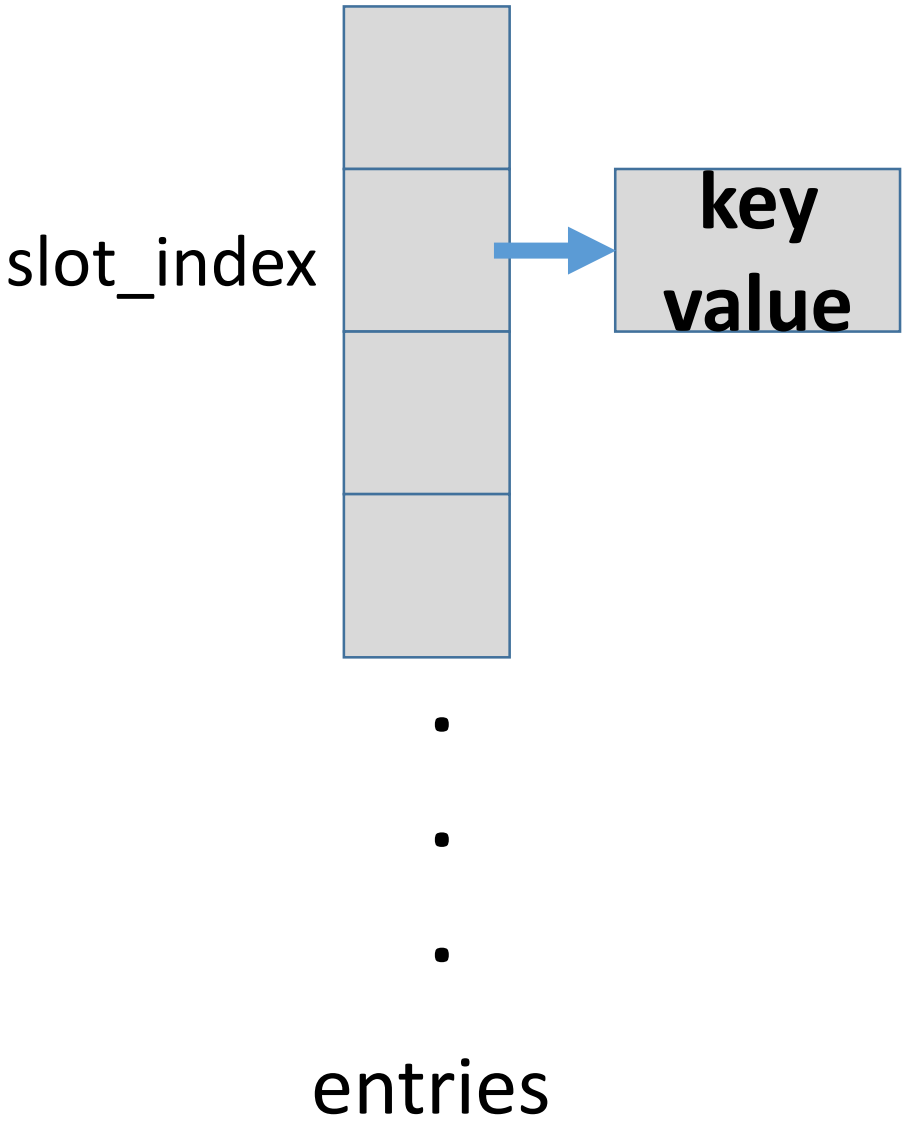
Sequence Containers

The sequence containers (also known as (aka) sequential containers) represent linear data structures.

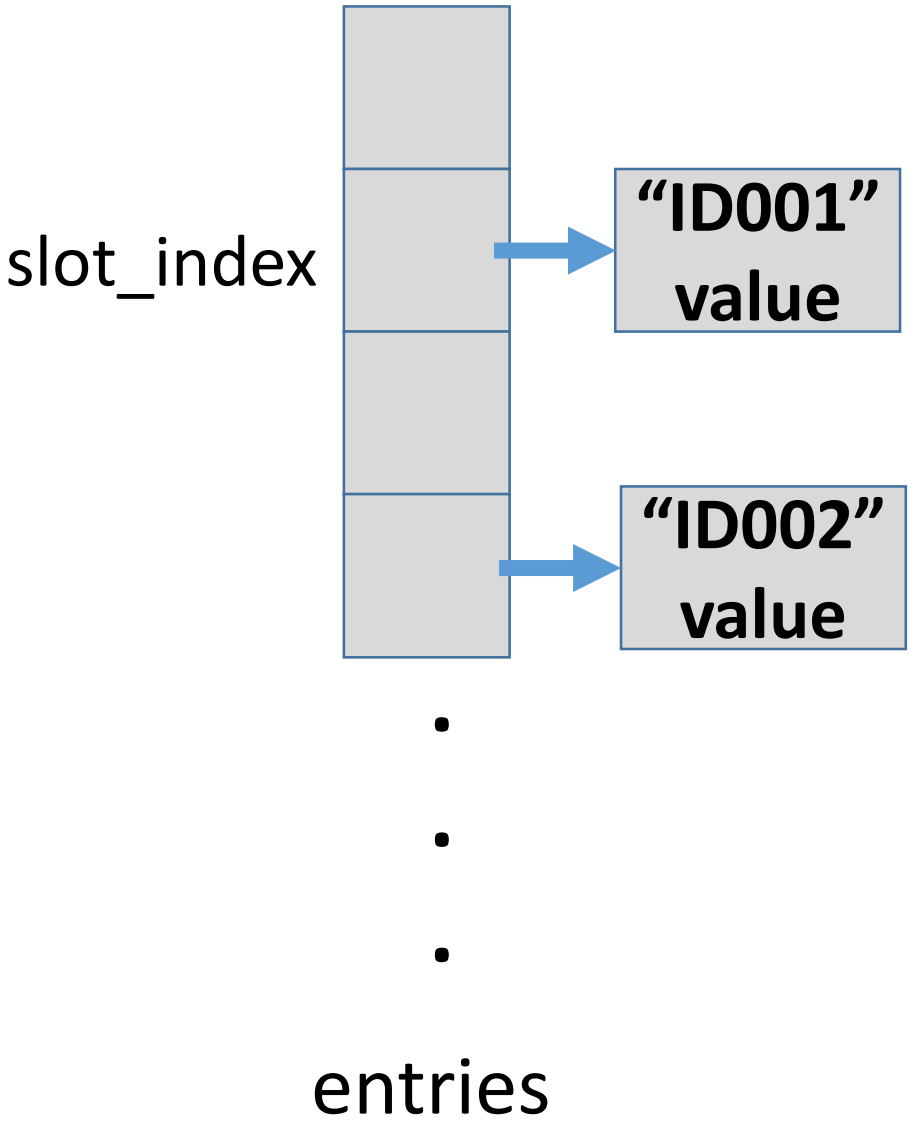
Examples: vector, list, and deque.



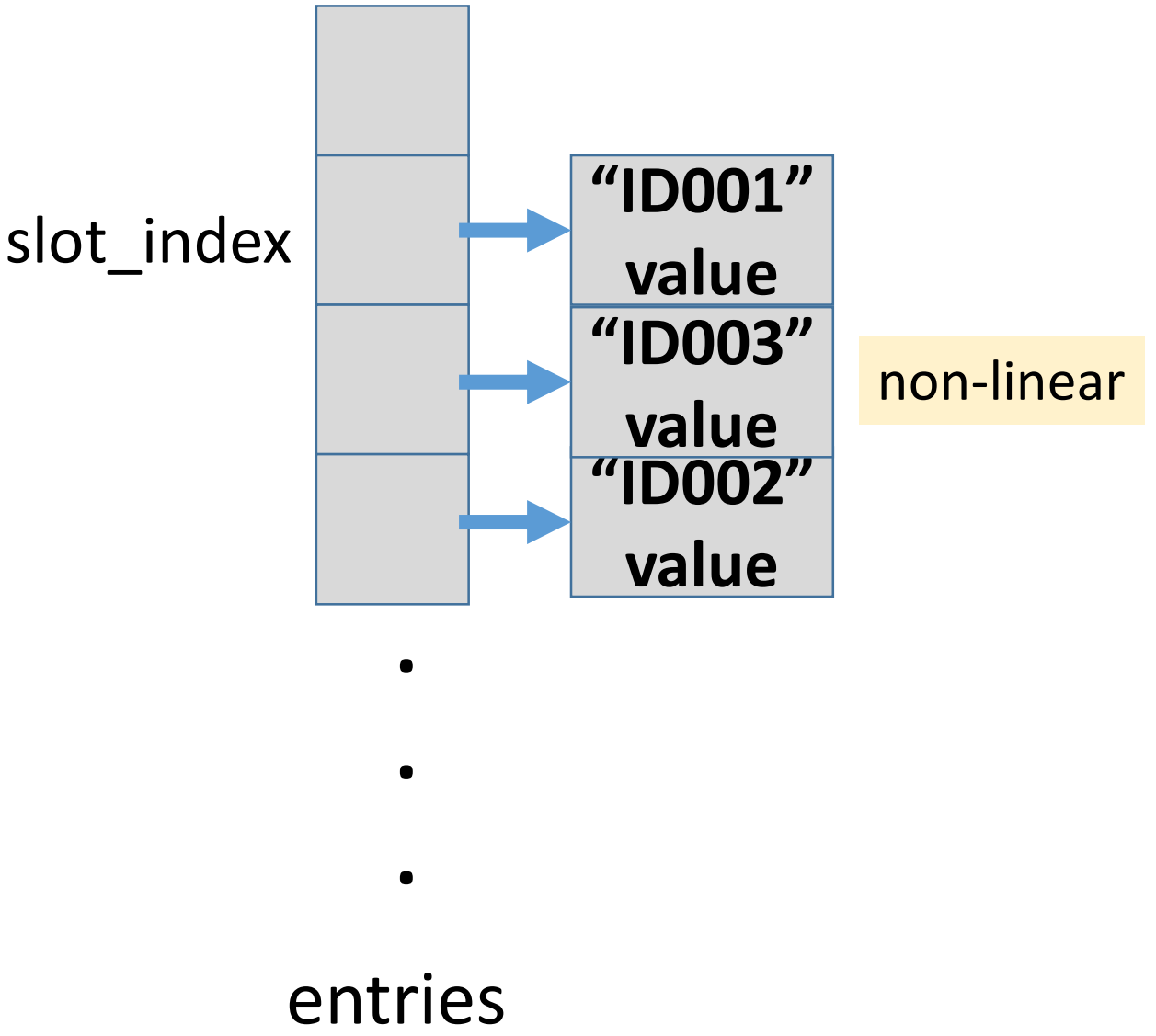
Associative Containers



Associative Containers



Associative Containers

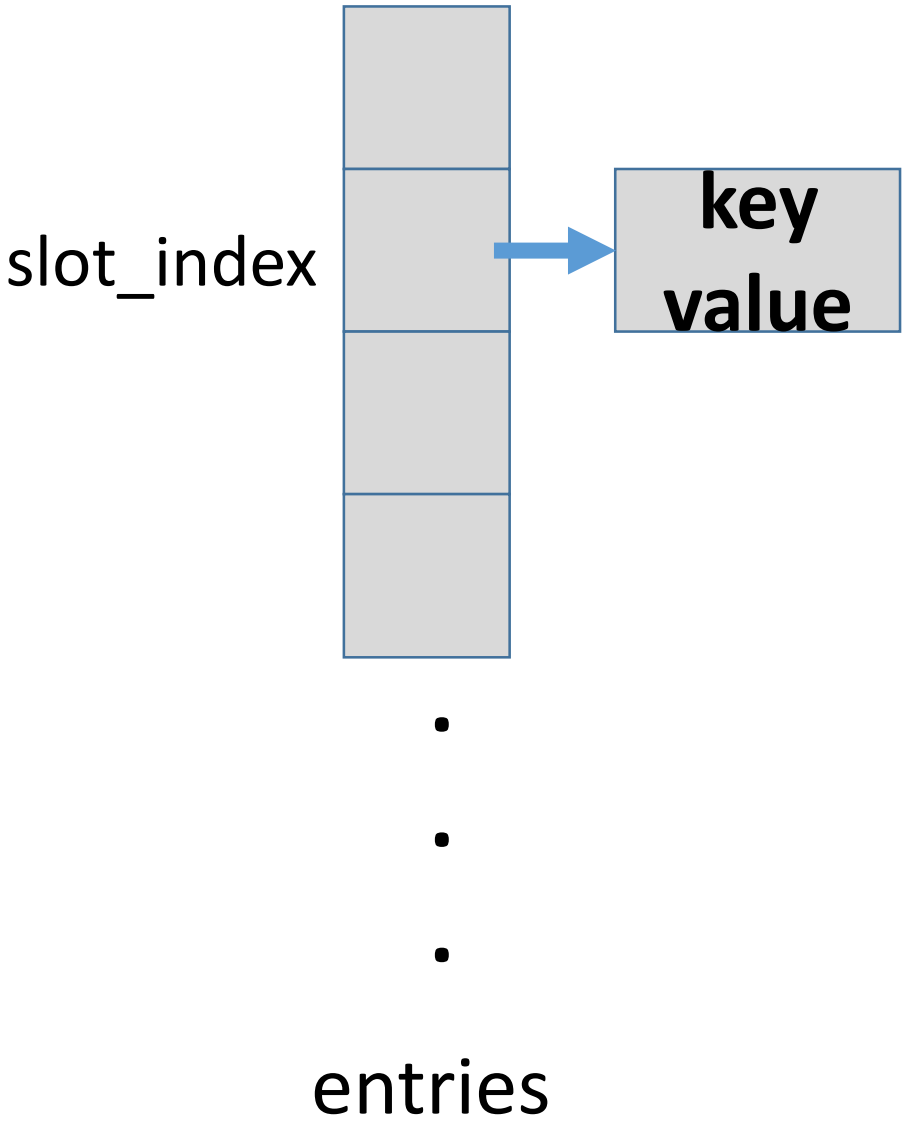


Associative Containers

Associative containers are non-linear containers that can locate elements stored in the container quickly.

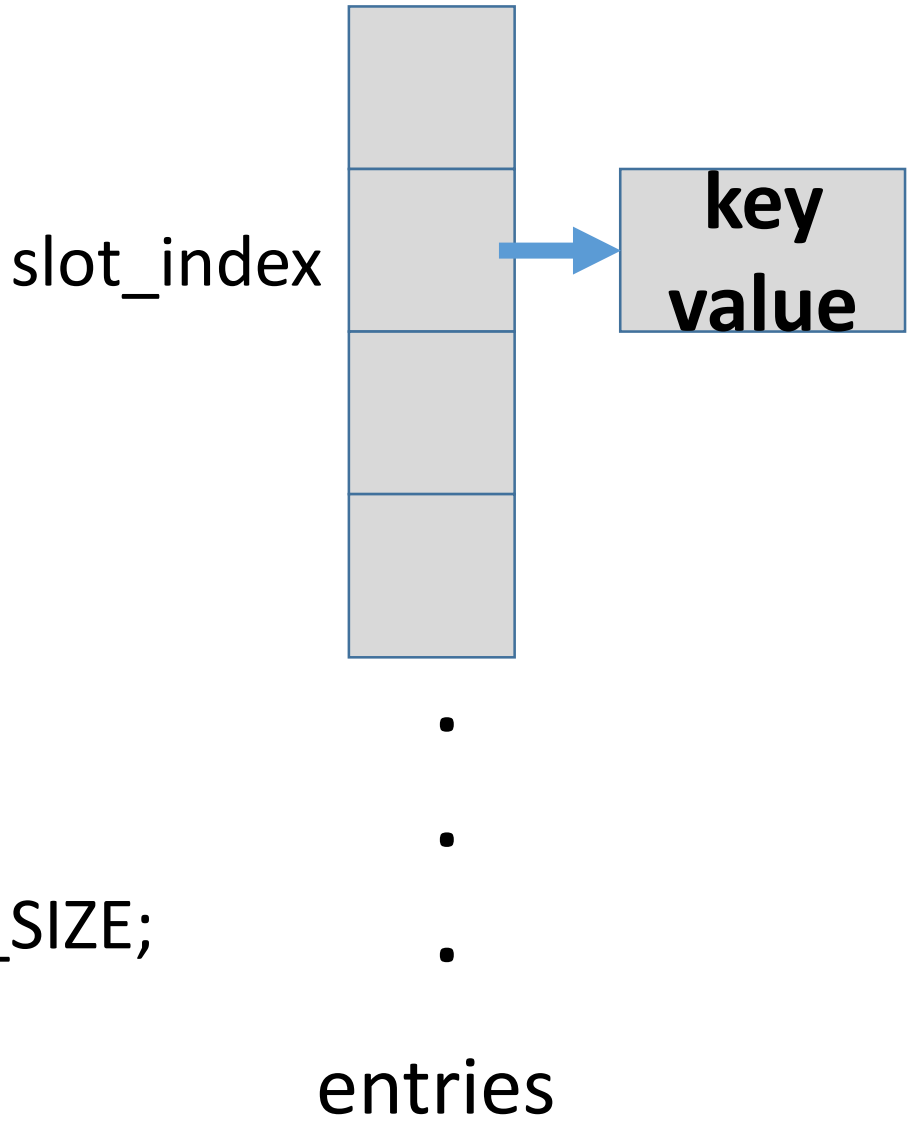
They store sets of values or *key/value* pairs.

Examples:
set, multiset, map, and multimap.



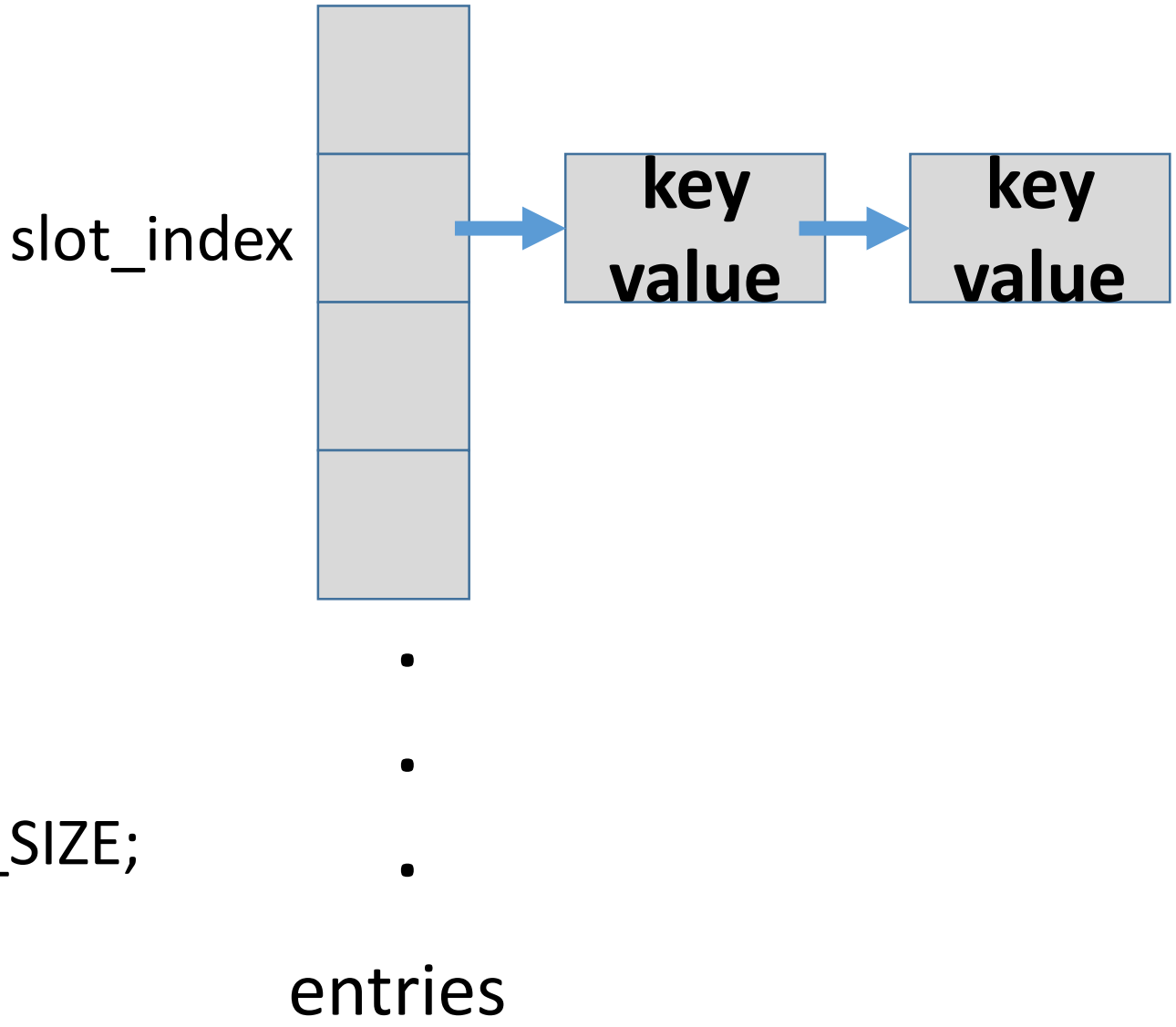
insert(string key, int value)

```
// map key to an integer
unsigned int slot_index;

//prime numbers
for (int i = 0; i < key.size(); ++i) {
        int v = key[i]-'a';
         slot_index += (v*11 + v*97+..)
}
slot_index = slot_index  % MAX_TABLE_SIZE;
```



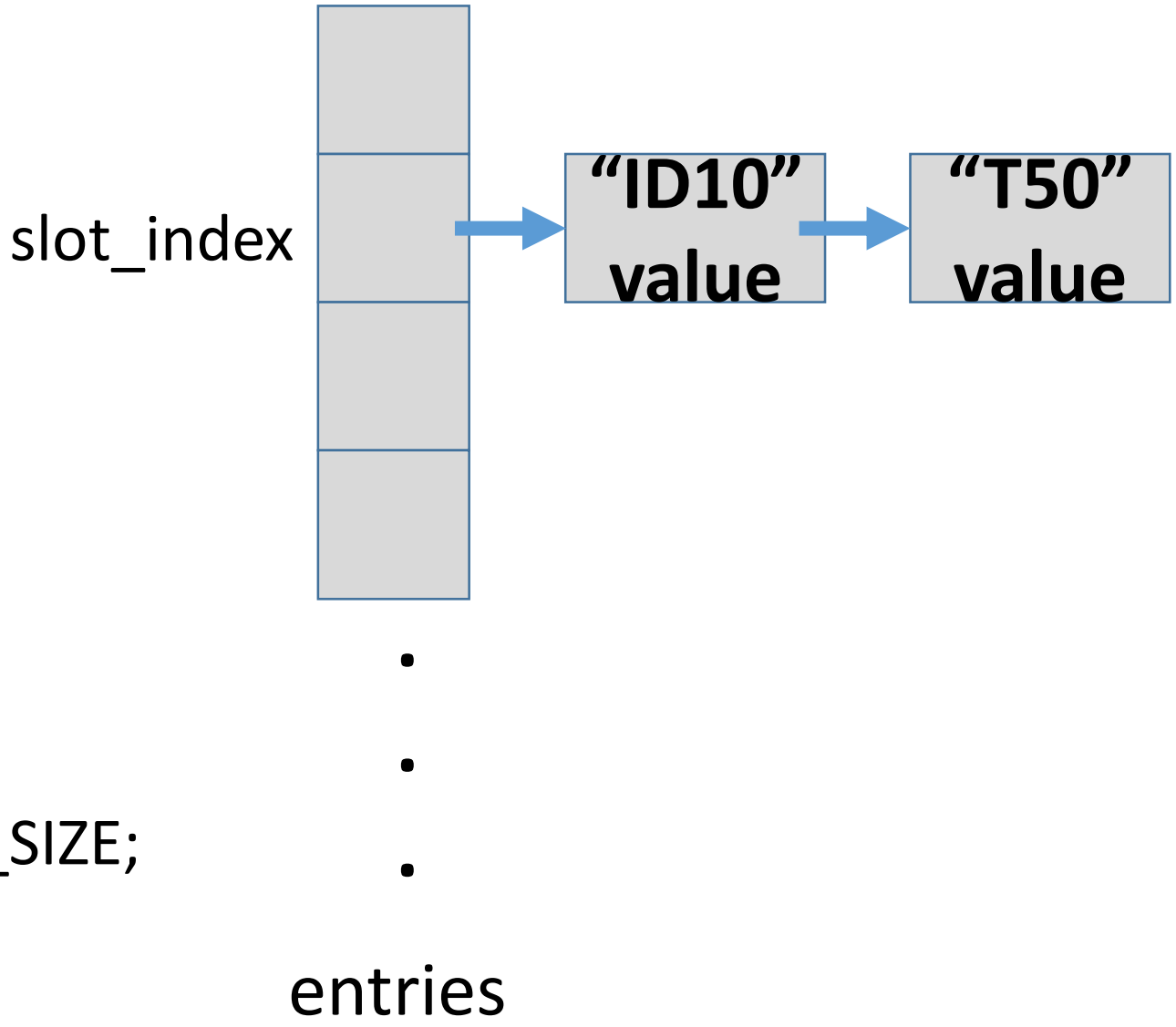
insert(string key, int value)

```
// map key to an integer
unsigned int slot_index;

//prime numbers
for (int i = 0; i < key.size(); ++i) {
        int v = key[i]-'a';
         slot_index += (v*11 + v*97+..)
}
slot_index = slot_index  % MAX_TABLE_SIZE;
```



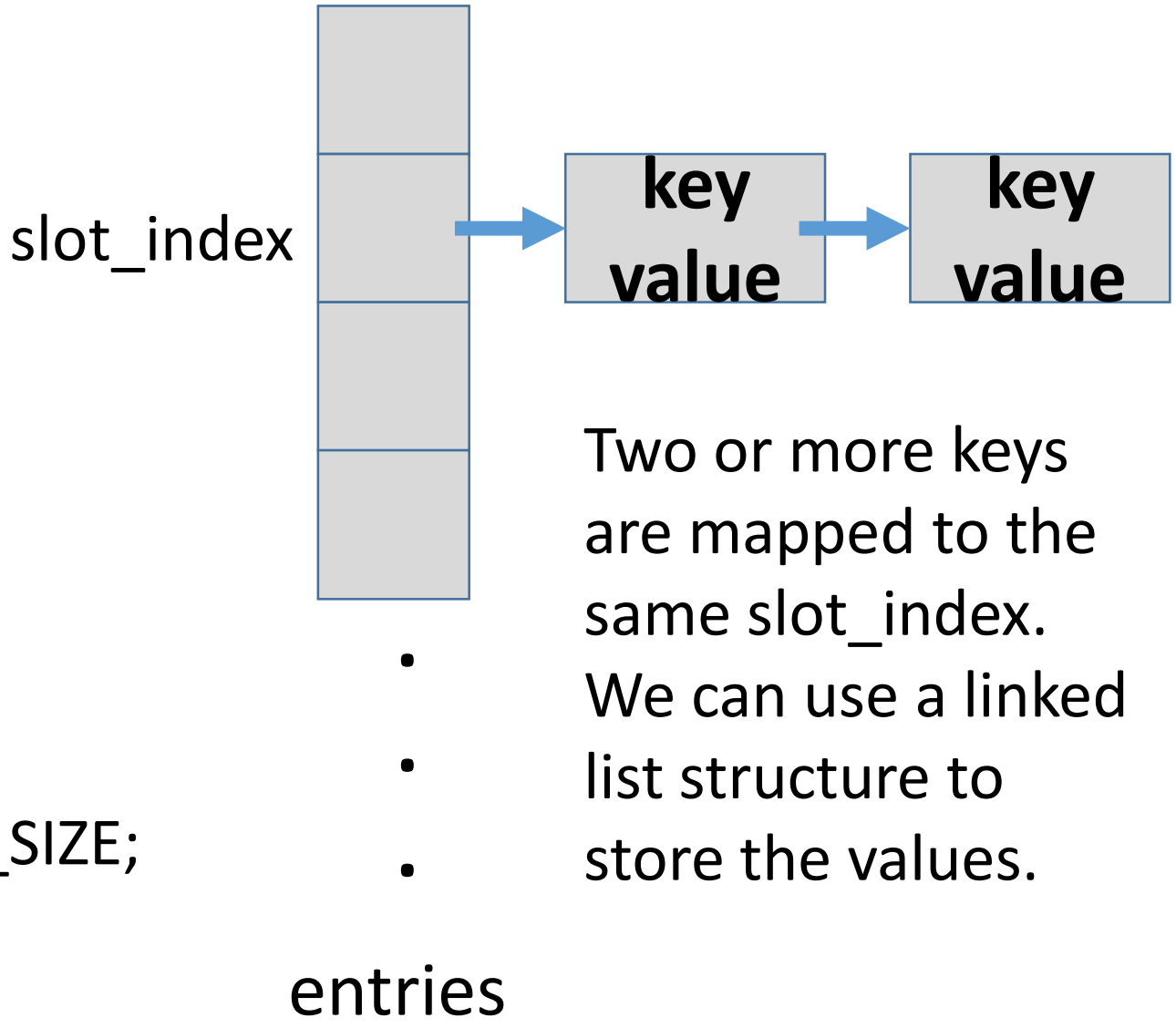
insert(string key, int value)

```
// map key to an integer
unsigned int slot_index;

//prime numbers
for (int i = 0; i < key.size(); ++i) {
        int v = key[i]-'a';
         slot_index += (v*11 + v*97+..)
}
slot_index = slot_index  % MAX_TABLE_SIZE;
```



insert(string key, int value)

```
// map key to an integer
unsigned int slot_index;

//prime numbers
for (int i = 0; i < key.size(); ++i) {
        int v = key[i]-'a';
         slot_index += (v*11 + v*97+..)
}
slot_index = slot_index  % MAX_TABLE_SIZE;
```



Two or more keys are mapped to the same slot_index. We can use a linked list structure to store the values.

Container Adapters

- Container adapters are adapted from sequence containers for handling special cases.

- Container adapters have constraints that must be satisfied.

Examples:

stack (LIFO), queue (FIFO), and priority_queue.

vector a; a.back(); a.pop_back();

Container Adapters

- Container adapters are A1 from sequence containers for handling special cases.

- Container adapters have A2 that must be A3

Examples:

stack (LIFO), queue (FIFO), and priority_queue.

vector a; a.back(); a.A4

STL Containers

STL Containers	Header File	
vector	<vector>	
deque	<deque>	
list	<list>	
set	<set>	
multiset	<set>	
map	<map>	
multimap	<map>	
stack	<stack>	
queue	<queue>	
priority_qeue	<queue>	

Common Functions to All Containers

Functions	Description
Non-arg constructor	
Constructors	
Copy constructor	
Destructor	
empty( )	
size( )	
Operator=, <, <=, ……	

Declaration of Objects of Containers

A simple example that demonstrates how to create

```
Vector:              vector <int> x;
List:                list <int> x;
Deque:               deque <int> x;
Set:                 set <int> x;
Multiset:            multiset <int> x;
Stack:               stack <int> x;
Queue:               queue <int> x;
Priority Queue:      priority_queue <int> x;
```

Iterators

Several functions (e.g., begin() and end()) in the first-class containers are related to iterators.

Sequence Containers

```
vector<int> intVector;
intVector.push_back(10); intVector.push_back(40); …..
vector<int>::iterator p1;
cout << "Traverse the vector: ";
for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
{
   cout << *p1 << " ";
}
```

Remark: x.end() points to past-the-end element

Iterators

Several functions (e.g., begin() and end()) in the first-class containers are related to iterators.

Sequence Containers

```
vector<int> intVector;
intVector.push_back(10); intVector.push_back(40); …..
vector<int>::iterator p1;
cout << "Traverse the vector: ";
for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
{
   cout << *p1 << " ";
}
```

*p1: The object associated with the iterator, p1.

Remark: x.end() points to past-the-end element

Iterator

```
STL_Type<parameter> x;
STL_Type<parameter>::iterator p;
for (p = x.begin(); p != x.end(); p++) {
    ……
}
Example:
map<int, string> x;
map<int, string>::iterator p;

for (p = x.begin(); p != x.end(); p++) {
    cout << p->first << " " << p->second << endl;
}
```

Iterator

```
map<int, string>::iterator p;
map<int, string> map1; ……
for (p = map1.begin(); p != map1.end(); p++) {
    cout << p->first << " " << p->second << endl;
}
```

```
map1[102] = "Jane Smith";
map1[103] = "Peter Reed";
```

Iterator

```
map<int, string>::iterator p;
map<int, string> map1; ……
for (p = map1.begin(); p != map1.end(); p++) {
    cout << p->first << " " << p->second << endl;
}
```

```
map1[102] = "Jane Smith";
map1[103] = "Peter Reed";
```

```
Map:
p = map1.find(key);
if (p == map1.end())
   cout << " Key " << key << " is not found in map1";
else
   cout << " " << p->first << " " << p->second << endl;
```

Type of Iterators

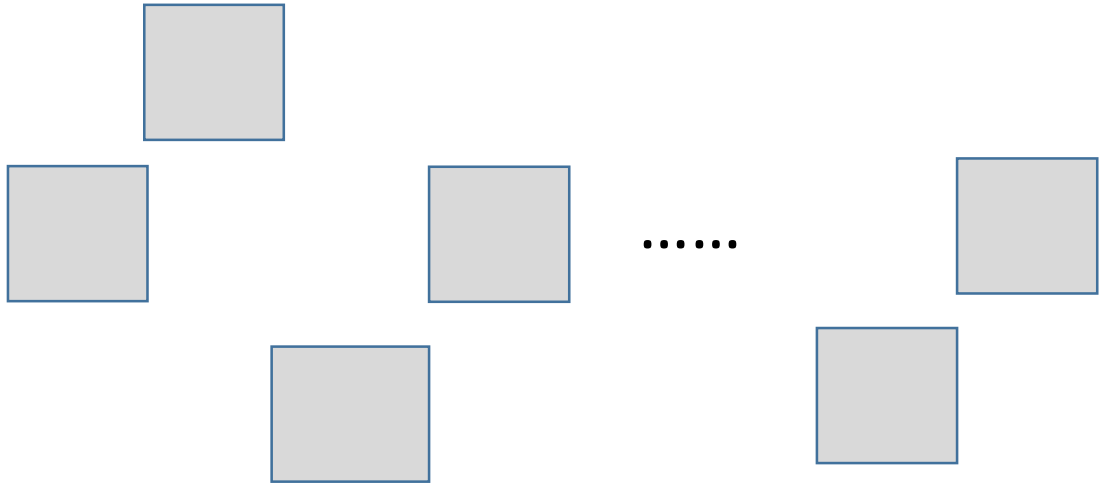
Five categories:

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random access iterators

Type of Iterators

Five categories:

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random access iterators

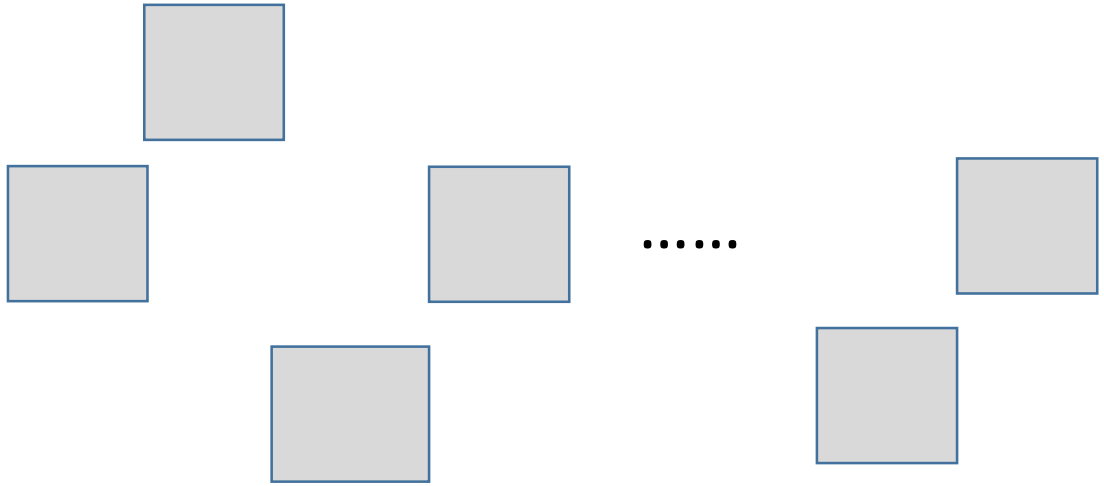


The objects may be organized non-linearly.

Type of Iterators

Five categories:

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random access iterators



The objects may be organized non-linearly.

Example: // copy source to target
copy(src.begin( ), src.end( ), target.begin( ));

Input iterators

- Data can be read from the pointed-to element
- They are used in sequential input operations

```
vector<int> v;
vector<int>::iterator iter;

v.push_back(1);
v.push_back(2);
v.push_back(3);

for (iter = v.begin(); iter != v.end(); iter++)
        cout << (*iter) << endl;      // *iter : extract the element pointed by iter

//*iter = 4; not allowed. We can read only
```

Input iterators: Example

```
void printf_map(const map<int, string> &in_map)
{

        map<int, string>::const_iterator p;
        for (p = in_map.begin(); p != in_map.end(); p++)
                cout << p->first << " " << p->second << endl;
}
```

Input iterators: Example

```
void printf_map(const map<int, string> &in_map)
{
        // map<int, string>::iterator p does not work
        map<int, string>::const_iterator p;
        for (p = in_map.begin(); p != in_map.end(); p++)
                cout << p->first << " " << p->second << endl;
}
```

const_iterator

```
vector<int> intVector; intVector.push_back(11);
vector<int>::iterator p1 = intVector.begin();
vector<int>::const_iterator p2 = intVector.begin();
*p1 = 123; // OK
*p2 = 123; // Not allowed
cout << *p1 << endl; cout << *p2 << endl;
```

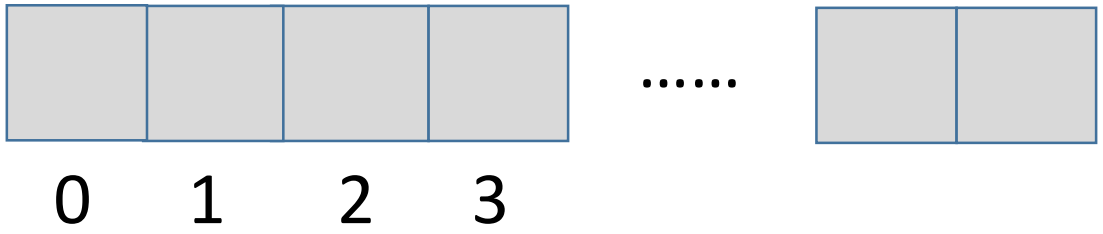
Output iterators

- Output iterators are only for storing.
- ++iter and iter++ to increment it, i.e., advance the pointer to the next element
- *iter = ... to store data in the location pointed to

Please read https://www.tutorialandexample.com/cpp-output-iterators

reverse_iterator

```
vector<int> intVector;
intVector.push_back(2);
intVector.push_back(5);
intVector.push_back(11);
vector<int>::reverse_iterator p1 = intVector.rbegin();
for (; p1 != intVector.rend(); p1++) {
    cout << *p1 << " ";
}
```



Iterator Types Supported by Containers

vector	random access iterators
deque	random access iterators
list	bidirectional iterators
set	bidirectional iterators
multiset	bidirectional iterators
map	bidirectional iterators
multimap	bidirectional iterators
stack	no iterator support
queue	no iterator support
priority_queue	no iterator support

The vector class

- Vectors are sequence containers representing arrays that can change in size.
- Vectors use contiguous storage locations for their elements.

```
std::vector<datatype> x;
std::vector<datatype> x(numOfElement);
std::vector<datatype> x(numOfElement, initializedValue);

std::vector<int> x(100);
std::vector<A> x(100, A(0));
for (A n:x) cout << n << endl;          // For each element n in x
```

deque (deck)

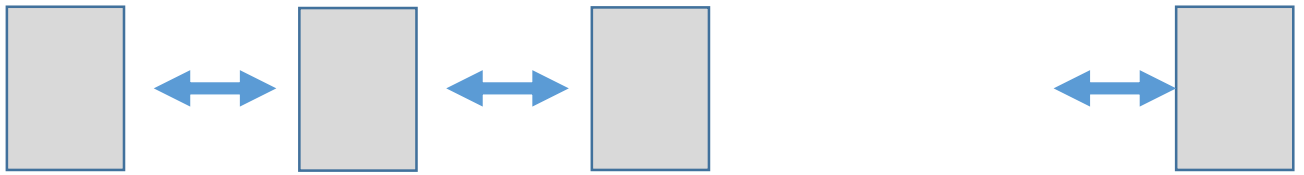
An irregular acronyms of double-ended queue.

Double-ended queues are sequence containers with dynamic sizes.


```
std::deque<int> d = {1, 3, 2, 7};
d.push_front(13);
d.push_back(8);
for (int n:d) cout << n << endl;
```

list

- Lists are sequence containers that allow constant time insert and erase operations.
- Iteration is allowed in both directions.
- List containers are implemented as doubly-linked lists.



set

- Sets are containers that store unique elements following a specific order.
- The value of an element identifies it (the value is itself the key).
- The value of an element must be A1
- The A2 of the elements in a set A3 once it is in the container.
- Elements can be inserted or removed from the container.

set

- Sets are containers that store unique elements following a specific order.
- The value of an element identifies it (the value is itself the key).
- The value of an element must be **unique**.
- The **value** of the elements in a set **cannot** be **modified** once it is in the container.
- Elements can be inserted or removed from the container.

multiset

- Multisets are containers that store elements following **a specific order.**
- **Multiple** A1 can have A2.
- The value of an element also identifies it (the value is itself the key, of type T).
- The value of the elements in a multiset cannot be modified.
- The elements can be inserted or removed from the container.

multiset

- Multisets are containers that store elements following **a specific order.**
- **Multiple elements** can have **equivalent values**.
- The value of an element also identifies it (the value is itself the key, of type T).
- The value of the elements in a multiset cannot be modified.
- The elements can be inserted or removed from the container.

map

- Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.
- The **key values** are generally used to sort and **uniquely** identify the elements.
- The mapped values store the content associated to this key.
- The types of key and mapped value may differ.

```
typedef pair<string , int> value_type;
map<value_type> mapX;
value_type x;
x.first = “1323”
x.second = 10;
mapX[“1323”] = 10;
```

<student_name, student_ID>

```
pair< string , int > x;
x.first = “1323”
x.second = 10;
```

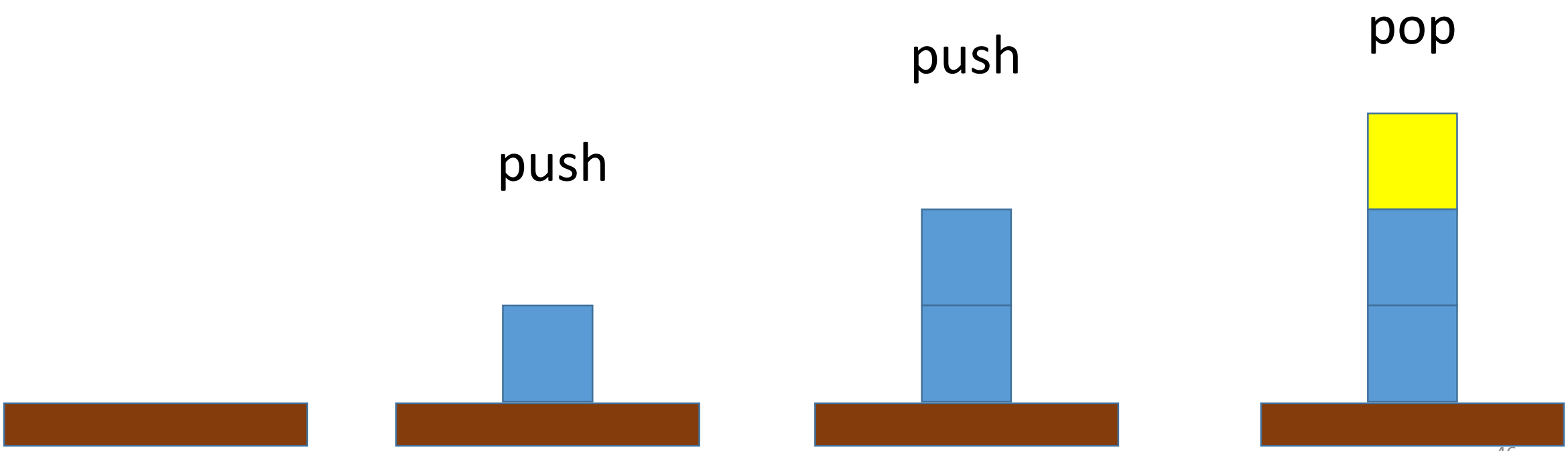
multimap

- Multimaps are associative containers that store elements formed by a combination of a key value and a mapped value
- Properties: a specific order; **multiple elements** can have **equivalent keys**.
- The key values are used to sort and uniquely identify the elements.
- The mapped values store the content associated to this key.
- The types of key and mapped value may differ.

typedef pair<const Key, T> value_type;

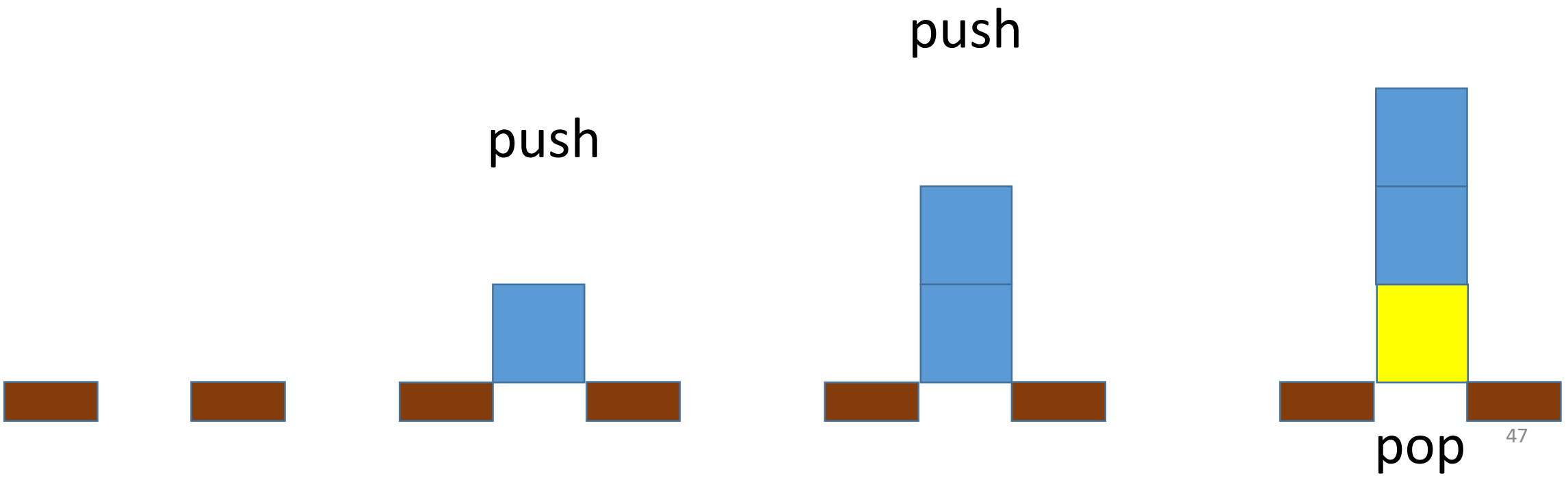
Adaptor: stack

- Stacks operate elements in a LIFO context (last-in first-out).
- The elements are inserted and extracted only from one end of the container.



Adaptor: queue

- **queue**s operate elements in a FIFO context (first-in first-out).
- The elements are inserted into one end of the container and extracted from the other.



Adaptor: priority_queue

- Priority: Its first element is the greatest of the elements.
- The elements are ordered based on a *strict weak ordering* criterion.

Operators Supported by Iterators

++p	p1>=p2			
p++	p(i)			
p--	*p			
--p	p1!=p2			
p1==p2				

Sequence Containers

- vector, list, and deque.
- The vector and deque containers are implemented using arrays.
- The list container is implemented using a linked list.

Common functions in sequence containers

push_back(element)	
pop_back( )	
front( )	
back( )	

Associative Containers

- Set, multiset, map, and multimap.
- Advantages: fast storage and quick access to retrieve elements using keys, called search keys.
- Elements are sorted according to some sorting criterion.

Common functions in associative containers

find(key)	
count(key)	
lower_bound(key)	
upper-bound (key)	

Multisets

- Store elements following a specific order.
- Multiple elements can have equivalent values.

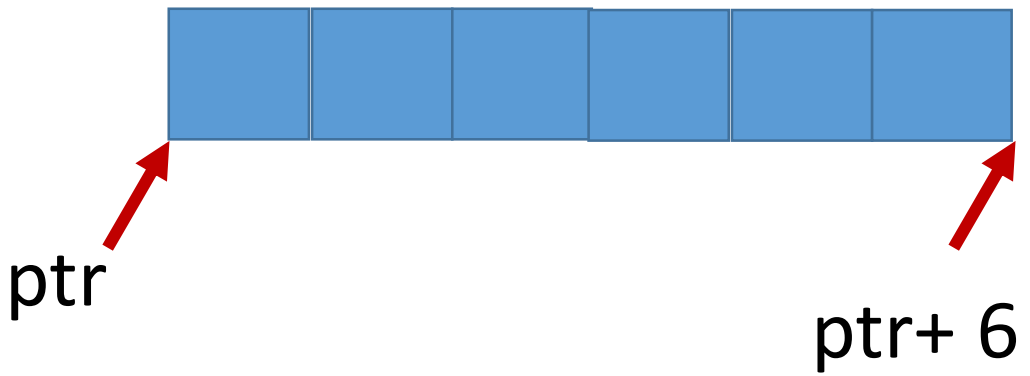
Associative Containers

- set, multiset, map, and multimap.
- They provide fast storage and quick access to retrieve elements using keys, called search keys.
- Elements are sorted according to some sorting criterion.
- The elements are sorted using the < operator by default.

Associative Containers: set and multiset

```
int ptr[] = {3, 8, 2, 7, 2, 3};
multiset<int> set1( ptr, ptr + 6);

cout << “Contents in set1: ";

for (int e: set1) cout << e << " ";
```

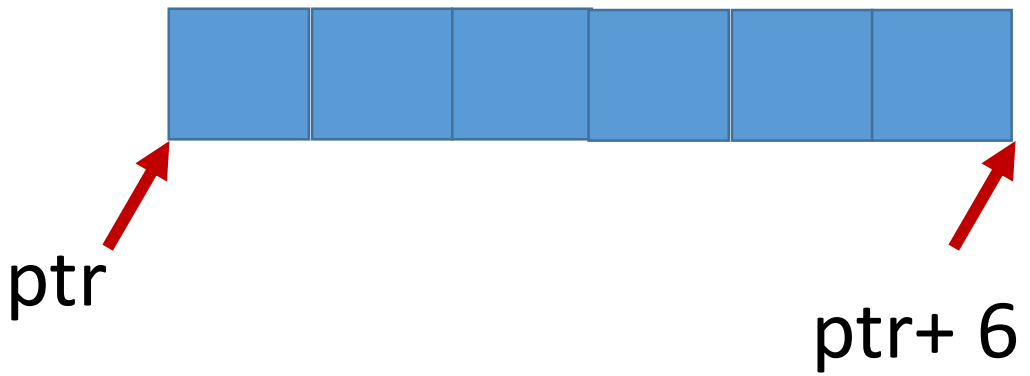
ptr + 6 (this is an address) points to the end of the elements



Associative Containers: set and multiset

```
int ptr[] = {3, 8, 2, 7, 2, 3};
multiset<int> set1( ptr, ptr + 6);

cout << “Contents in set1: ";

for (int e: set1) cout << e << " ";
```

ptr + 6 (this is an address) points to the end of the elements

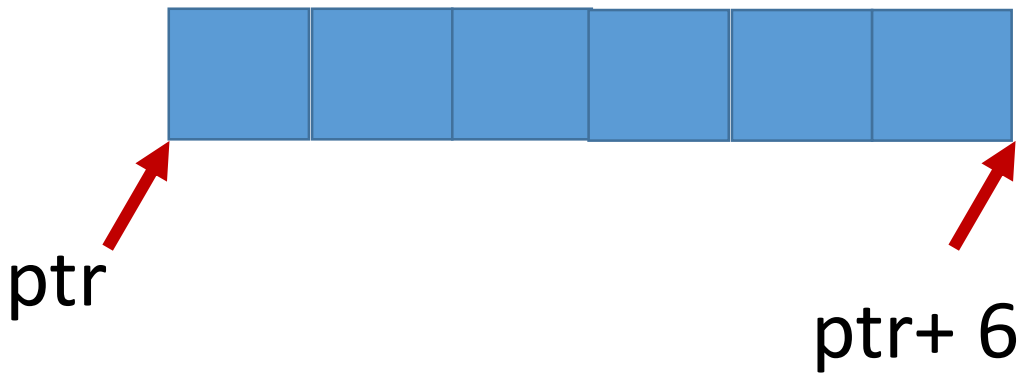


`Remark: x.end()` points to the past-the-end element

Associative Containers: set and multiset

ptr+ 6 (this is an address)
points to the end of the elements

```
int ptr[] = {3, 8, 2, 7, 2, 3};
multiset<int> set1( ptr, ptr+ sizeof( ptr) / sizeof(int));

cout << “Contents in set1: ";

for (int e: set1) cout << e << " ";
```



`Remark: x.end()` points to the past-the-end element

Traversing elements of a multiset structure

```
multiset<int>::iterator e;

//for (int e: set1)
for (e = set1.begin(); e != set1.end(); ++e) {
  cout << *e << " ";
}
```

printf maps

```
void printf_map(const map<string,int> &in_map) {
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

printf maps

```
void printf_map(const map<string,int> &in_map) {
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Change the function to a template.

printf maps

```
void printf_map(const map<string,int> &in_map) {
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Change the function to a template.

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

printf maps

```
void printf_map(const map<string,int> &in_map) {
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Change the function to a template.

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

How to use the template?

How to use the template?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map                    (map1);
```

How to use the template?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map<                    >(map1);
```

How to use the template?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map<map<                    >>(map1);
```

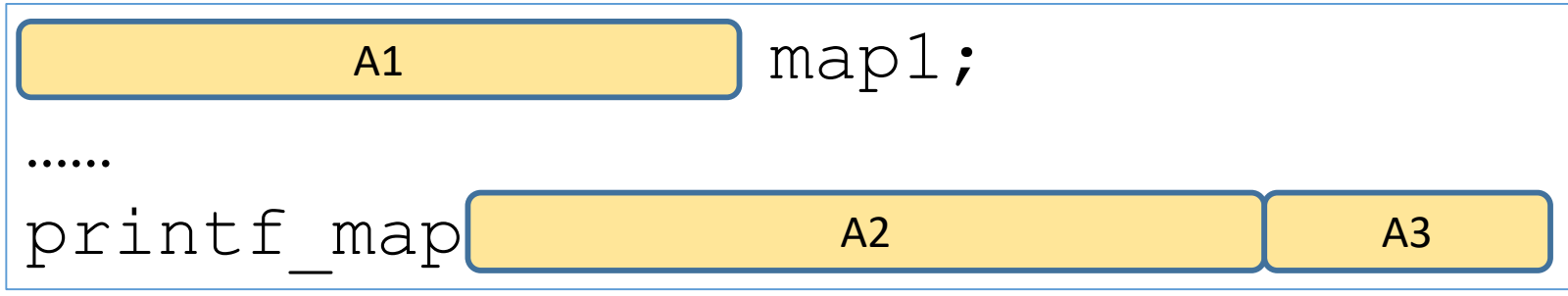
How to use the template?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map<map<string, int>>(map1);
```

How to use the template?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```



vector<int> arr; vector<vector<vector<vector>>> arr;

What does the compiler do?

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map<map<string, int>>(map1);
```

The compiler creates the function for us **on demand**

```
template<typename T>
void printf_map(const T &in_map) {
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

```
map<string, int> map1;
……
printf_map<map<string, int>>(map1);      // text substitution
```

```
void printf_map(const map<string,int> &in_map) {
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Container Adapters

- Container adapters: **stack**, **queue**, and **priority_queue**.

- They are adapted from the sequence containers for handling special cases.

- The STL enables the programmer to choose an appropriate sequence container for a container adapter. For example, we can create a stack with a data structure such as vector, deque, or list.

```
queue<int, list<int> > queue_01;    // ok. vector algorithm is ok
queue<int, vector<int> > queue_02; // error. no pop_front
```

Container Adapter: priority_queue

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}

int main() {
  priority_queue<int, deque<int>, greater<int>> queue2;
  queue2.push(4); queue2.push(1); queue2.push(7);
  printQueue(queue2);
  return 0;
} // what are the output?
```

Container Adapter: priority_queue

```
template<typename T> void printStructure(T& p) {
  while (!p.empty()) {
        cout << p.top() << " "; p.pop();
  }
} // generic
```

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}
```

Container Adapter: priority_queue

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}

int main() {
  priority_queue<int, deque<int>, greater<int>> queue2;
  queue2.push(4); queue2.push(1); queue2.push(7);
  printQueue(queue2);
  return 0;
} // what are the output? ???
```

Container Adapter: priority_queue

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}

int main() {
  priority_queue<int, deque<int>, greater<int>> queue2;
  queue2.push(4); queue2.push(1); queue2.push(7);
  printQueue(queue2);
  return 0;
} // what are the output? 1 4 7          //ascending
```

Container Adapter: priority_queue

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}

int main() {
  priority_queue<int, deque<int>, less<int>> queue2;
  queue2.push(4); queue2.push(1); queue2.push(7);
  printQueue(queue2);
  return 0;
} // what are the output? ???
```

Container Adapter: priority_queue

```
template<typename T> void printQueue(T& pQueue) {
  while (!pQueue.empty()) {
        cout << pQueue.top() << " "; pQueue.pop();
  }
}

int main() {
  priority_queue<int, deque<int>, less<int>> queue2;
  queue2.push(4); queue2.push(1); queue2.push(7);
  printQueue(queue2);
  return 0;
} // what are the output? 7 4 1          //descending
```

Supplemental Materials

Input iterators

```
for (iter = v.begin(); iter != v.end(); iter++)
        cout << (*iter) << endl;
```

- ++iter and iter++ to increment it, i.e., advance the pointer to the next element
- *iter to dereference it, i.e., get the element pointed to
- == and != to compare it another iterator (typically the "end" iterator)

- *iter = 4; **//cannot perform assignment**.

Bidirectional iterators

- all ForwardIterator operations

- --iter and iter-- to decrement it, i.e., advance the pointer to the previous element

Random access iterators

- All BidirectionalIterator operations

- Standard pointer arithmetic, i.e., iter + n, iter - n, iter += n, iter -= n, and iter1 - iter2 (but **not** iter1 + iter2)

- All comparisons, i.e., iter1 > iter2, iter1 < iter2, iter1 >= iter2, and iter1 <= iter2

Adaptor: priority_queue

- Priority: Its first element is the greatest of the elements.
- The elements are ordered based on a *strict weak ordering* criterion.

strict weak ordering:

A **strict weak ordering** is a binary relation < on a set *S* that is a strict partial order (a transitive relation that is irreflexive, or equivalently,[6] that is asymmetric) in which the relation "**neither *a* < *b* nor *b* < *a***" is transitive.

("**neither *a* < *b* nor *b* < *a***" ???)

https://en.wikipedia.org/wiki/Weak_ordering

Adaptor: priority_queue

Its first element is always the greatest of the elements it contains, according to some *strict weak ordering* criterion.

strict weak ordering:

A **strict weak ordering** is a binary relation < on a set *S* that is a strict partial order (a transitive relation that is irreflexive, or equivalently, that is asymmetric) in which the relation "**neither *a* < *b* nor *b* < *a***" is transitive.

"**neither *a* < *b* nor *b* < *a***" : not comparable; incomparability

If “a not comparable with b” and “b not comparable with c”,

then “a not comparable with c”.

printf maps

```
void printf_map(const map<string,int> &in_map)
{
    map<string,int>::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Change it into a template?

printf maps

```
template<typename T>
void printf_map(const T &in_map)
{
    T::const_iterator p;
    for (p = in_map.begin(); p != in_map.end(); p++)
        cout << p->first << " " << p->second << endl;
}
```

Predefined Iterators

We can use the typedef keyword to predefine iterators.

Example:
typedef map<int, string>::const_iterator mapInputIterator;

map<int, string>::const_iterator p1;
mapInputIterator p2;

==
Define a data type:
typedef int integer;

integer value = 40;

Insert iterators

- Insert iterators "point" to some location in a container and insert elements.

- For example,

*iter = value;

// This *inserts* the value in the place pointed to by the iterator.

Forward iterators

- ForwardIterator combines InputIterator and OutputIterator.

- use them to read and write to a container.

Design for Container-Based Code

Two rules for making container-based code general and efficient:

1. Never pass containers into a function.
2. Pass iterators instead.

3. Never return containers.
4. Return or pass iterators instead.

```
template <class Container>
double product( const Container & container )
{
  Container::iterator i = container.begin();
  double prod = 1;
  while ( i != container.end() ) prod *= *i++;
  return prod;
}
```

{4, 5, 9, 6}
prod = 4*5*9*6

```
vector<double> nums;
...
return product( nums );
```

```
double nums[] = { 3.2, 3.5, 7.6, 4.9 };


return product( nums ); //error.
// no .begin() and .end()
```

```
template <class Container>
double product( const Container & container )
{
  Container::iterator i = container.begin();
  double prod = 1;
  while ( i != container.end() ) prod *= *i++;
  return prod;
}
```

{4, 5, 9, 6}
prod = 4*5*9*6

```
vector<double> nums;
...
return product( nums );
```

```
double nums[] = { 3.2, 3.5, 7.6, 4.9 };

return product( nums ); //error.
// no .begin() and .end()
```

```
template <class Iter>
double product( Iter start, Iter stop )
{
  double prod = 1;
  while ( start != stop ) prod *= *start++;
  return prod;
}
```

{4, 5, 9, 6}
prod = 4*5*9*6

```
vector<double> n;
...
return product(
   n.begin(), n.end());
```

```
double nums[] = { 3.2, 3.5, 7.6, 4.9 };

return product( nums, nums+4 ); //
```

Map Example

```
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;

cout << "Initial contents in map1:\n";
map<string, int>::iterator p;
for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " " << p->second << endl;

cout << "Enter a string to serach for the key: “<< endl;
string str;
cin >> str;
p = map1.find(str);
```

Map Example

```
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;

cout << "Initial contents in map1:\n";
map<string, int>::iterator p;
for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " " << p->second << endl;

cout << "Enter a string to serach for the key: “<< endl;
string str;
cin >> str;
p = map1.find(str);
```

Enter a string to search for the key:
John Smith

What are the output?

Map Example

```
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;

cout << "Initial contents in map1:\n";
map<string, int>::iterator p;
for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " " << p->second << endl;

cout << "Enter a string to serach for the key: “<< endl;
string str;
cin >> str;
p = map1.find(str);
```

```
Initial contents in map1:
Jane Smith 102
Jeff Reed 103
John Smith 100
Tom King 101
Enter a string to search for the key:
```

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;;
string str;
cin >> str;
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

Enter a string to search for the key:
John Smith

What are the output?

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;;
string str;
cin >> str;
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

Enter a string to search for the key:
John Smith

not found in map1

What is the mistake?

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;;
string str;
cin >> str; // it returns one word John
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

Enter a string to search for the key:
John Smith

not found in map1

What is the mistake?

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;;
string str;
getline(cin, str, “\n”); // Does this work?
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

Enter a string to search for the key:
John Smith

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;;
string str;
getline(cin, str, “\n”); // Does this work?
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

```
Enter a string to search for the key:
John Smith
```

“\n” is a string.

```
#include <istream>
map<string, int> map1;
map1.insert(map<string, int>::value_type("John Smith", 100));
map1.insert(map<string, int>::value_type(“Tom King", 101));
map1["Jane Smith"] = 102;
map1["Jeff Reed"] = 103;
……
cout
  << "Enter a string to search for the key: “
  << endl;
string str;
getline(cin, str, ‘\n’); // Good.      ‘\n’ is a character.
p = map1.find(str);
if (p == map1.end())
   cout << "  String " << str << " not found in map1";
 else
   cout << "  " << p->first << " " << p->second << endl;
```

```
Enter a string to search for the key:
John Smith
  John Smith 100
```