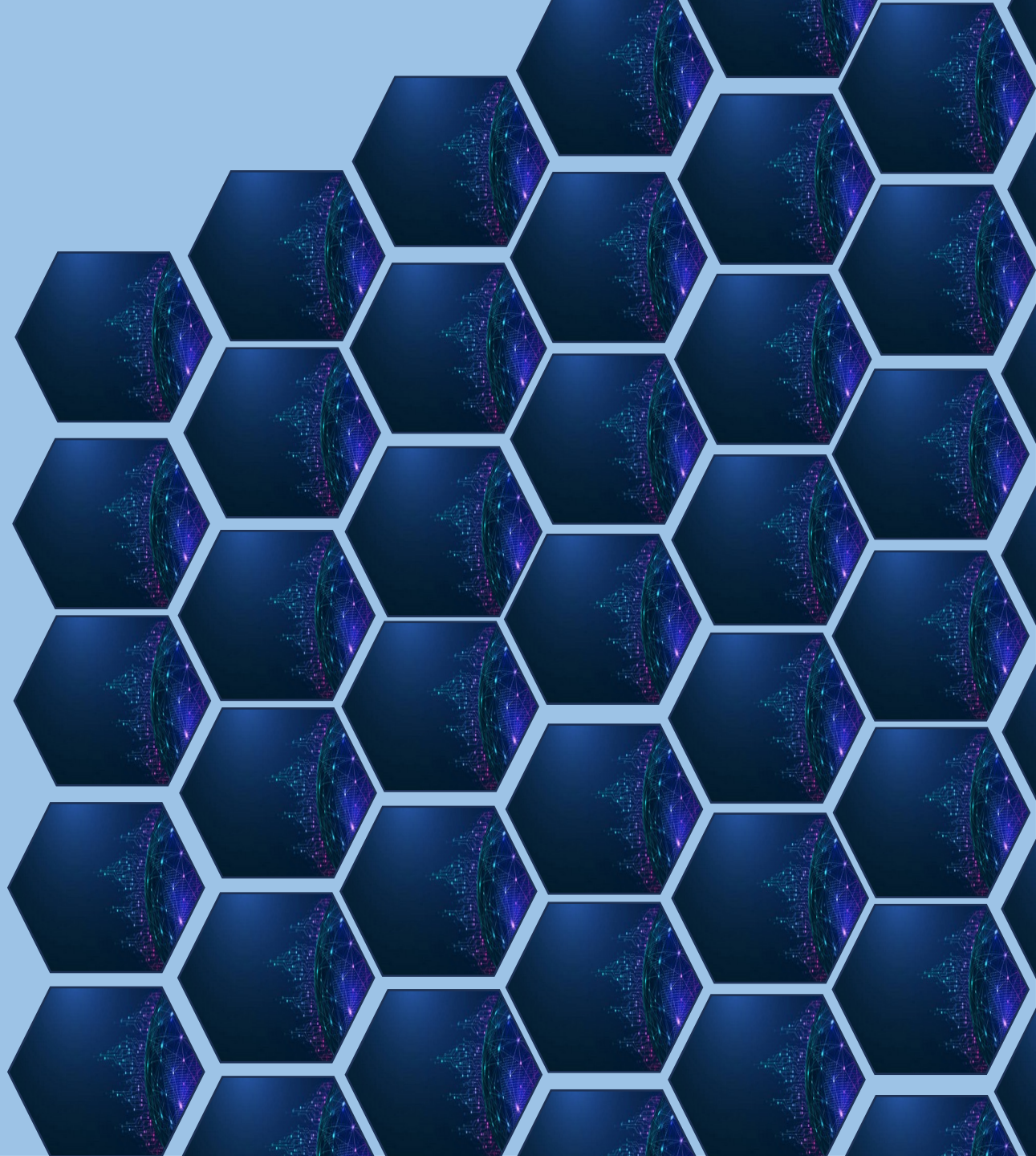
ROBOTICS CORNER

Robotics Corner



Robotics Corner

STL (standered Template library)

01 Containers

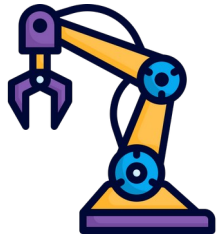
02 Algorithms

03 Iterators

ROBOTICS CORNER

Alexander Stepanov

https://www.sgi.com/tech/stl/drdobbs-interview.html

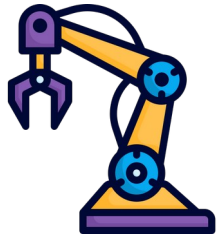


STL – General View

- Containers – Template Class
generalized data structures (you can use them for any type).

- Algorithms – Template Function
generalized algorithms (you can use them for almost any data Structure).

- Iterators – Glue between Containers and Algorithms
specifies a position into a container (generalized pointer)
permits traversal of the container.

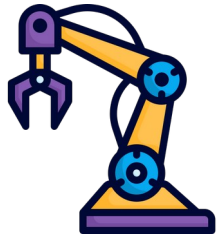


Basic STL Containers

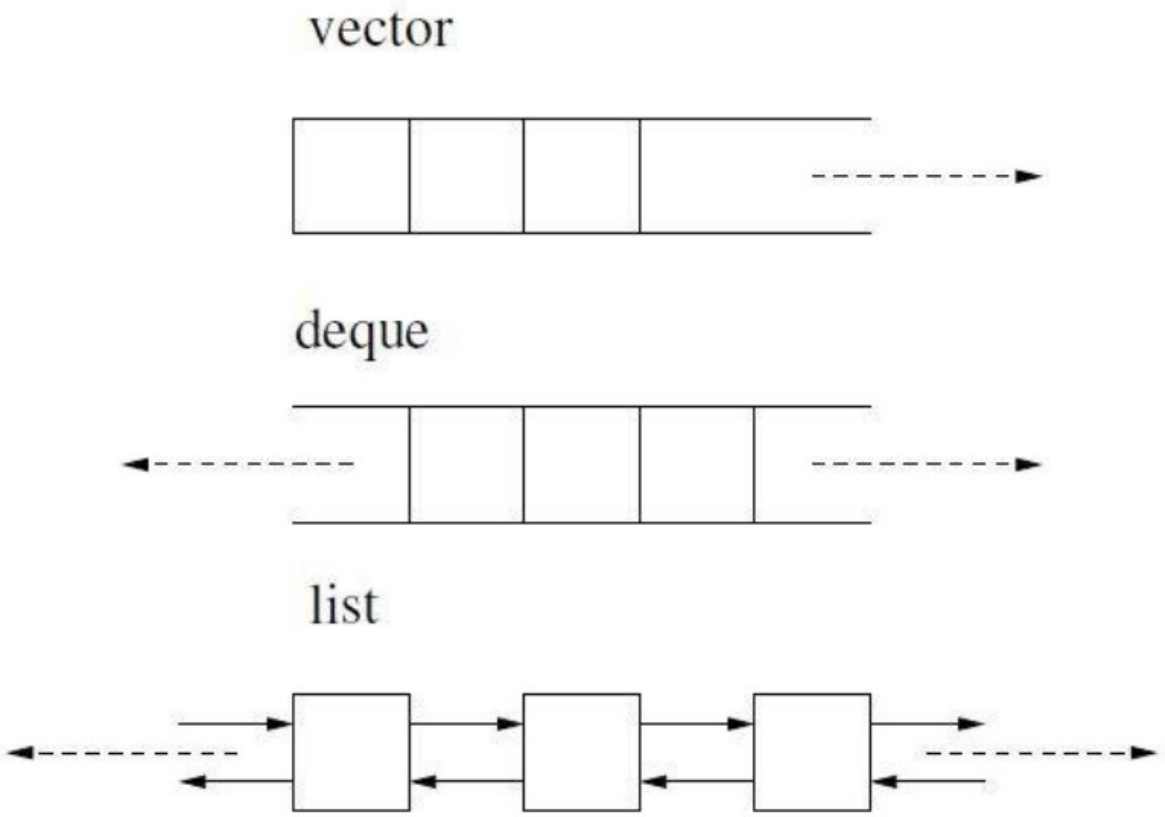
- **Sequence containers**
  - linear arrangement
    - `vector, deque, list` — `<vector> <deque> <list>`
    - `stack, queue, priority_queue` — `<stack> <queue>`

Container adapters →

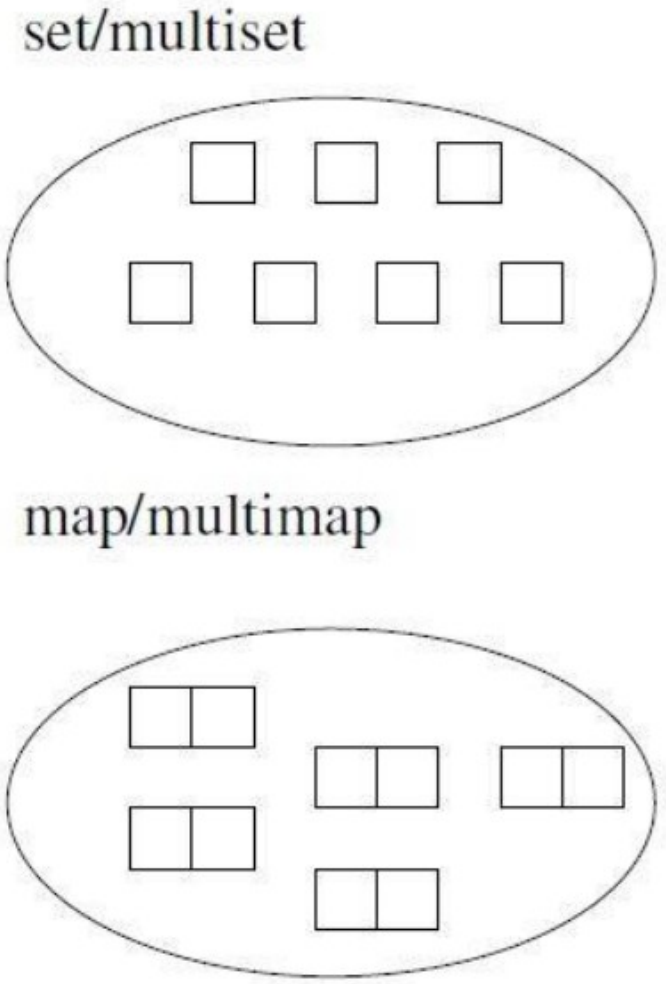
- **Associative containers**
  - provide fast retrieval of data based on keys
    - `set, multiset, map, multimap` — `<set> <map>`

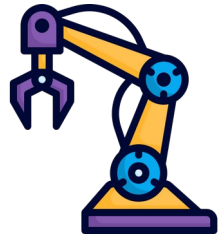


Sequence Containers



Associative Containers



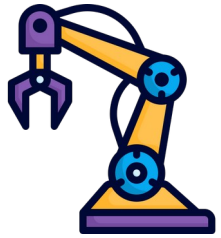


STL Containers C++11

- **Sequence containers**
  - `array` (C-style array)
  - `forward_list` (singly linked list)
- **Associative containers**
  - `unordered_set`, `unordered_multiset` (hash table)
  - `unordered_map`, `unordered_multimap` (hash table)

`<array>`
`<forward_list>`

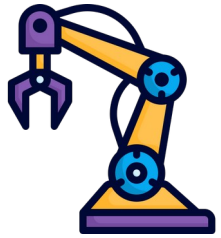
`<unordered_set>`
`<unordered_map>`



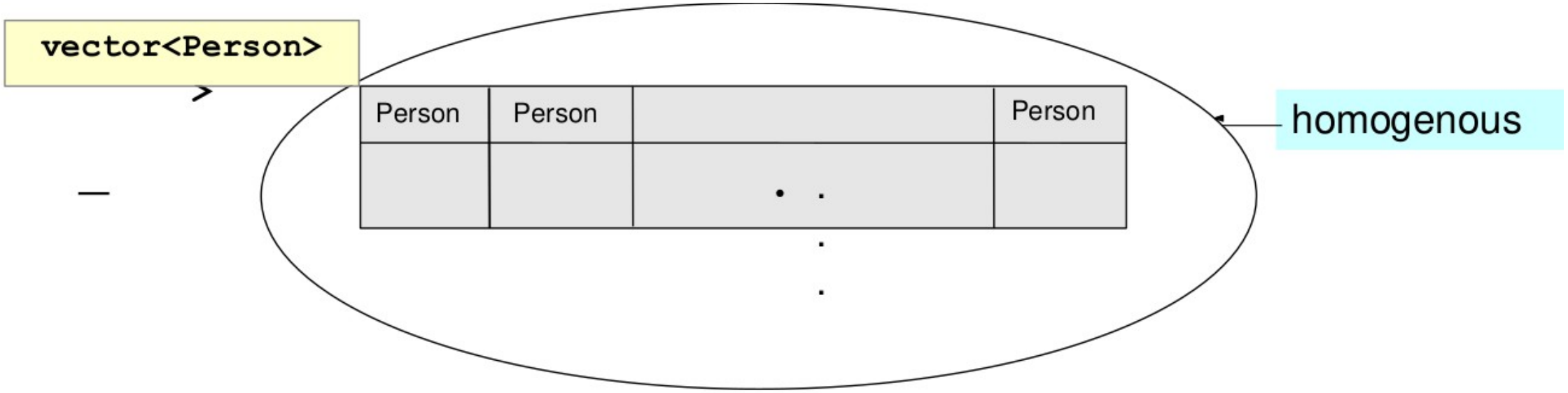
STL Containers

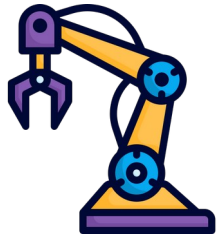
- homogeneous:
  - `vector<Person>, vector<Person*>`
- polymorphism
  - `vector<Person*>`

```
class Person{};
class Employee: public Person{};
class Manager : public Employee{};
```

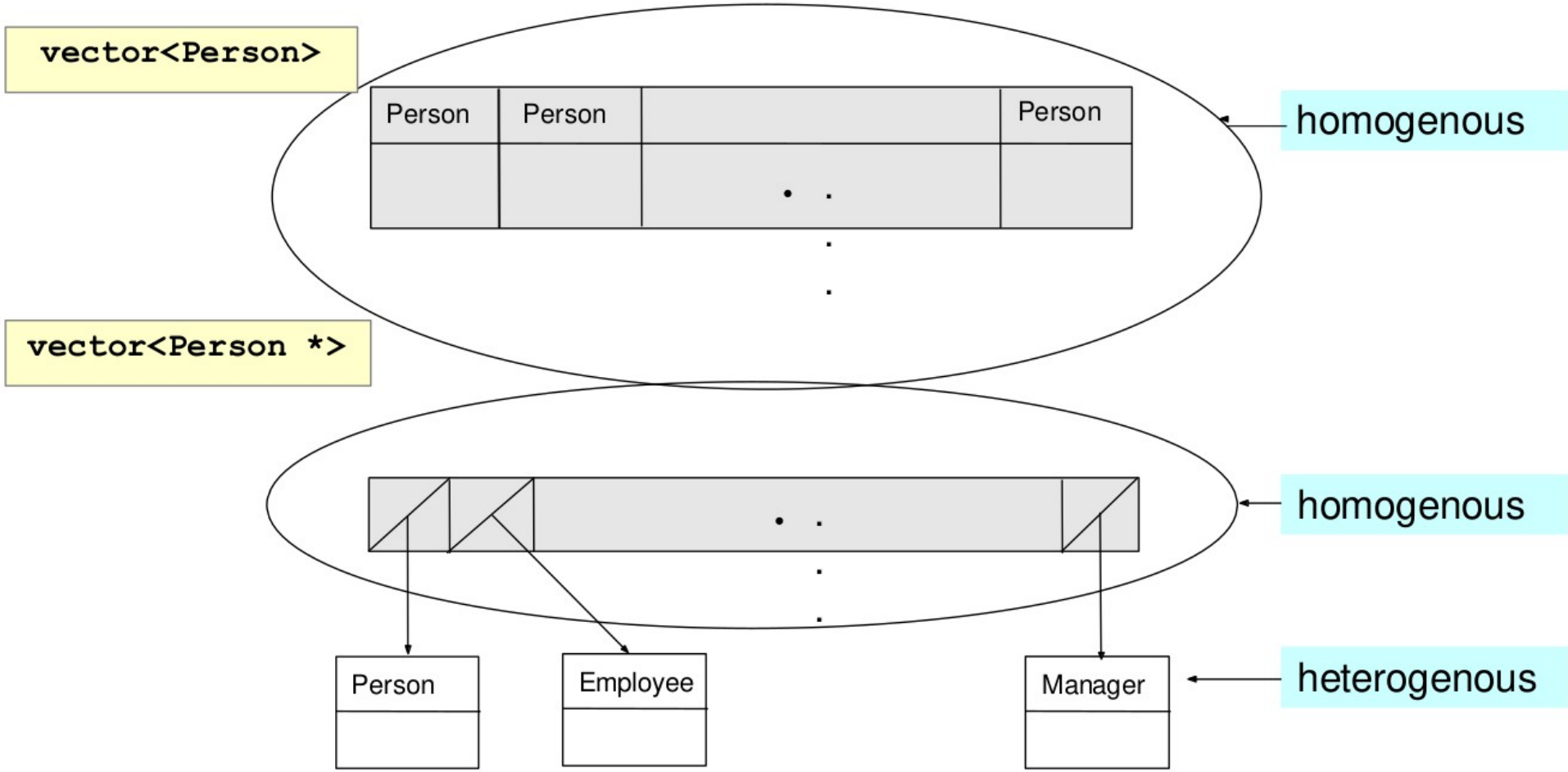


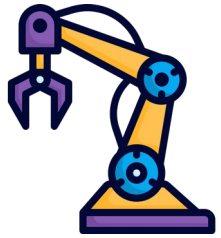
STL Containers





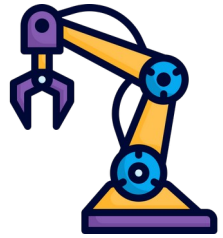
STL Containers





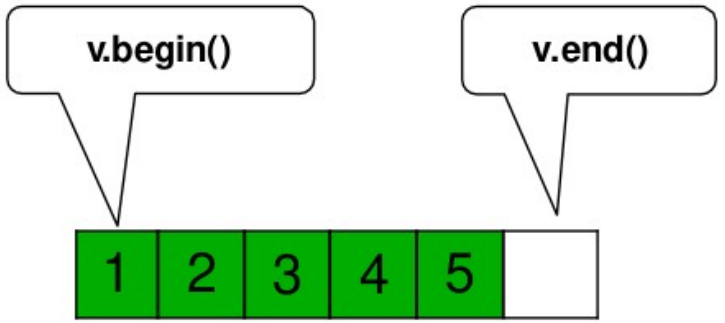
The vector container - constructors

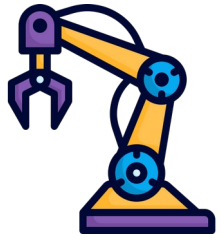
```
vector<T> v; //empty vector

vector<T> v(n, value);//vector with n copies of value

vector<T> v(n);//vector with n copies of default for T
```



The vector container – add new elements

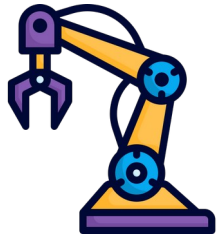
```
vector<int> v;

for( int i=1; i<=5; ++i){
    v.push_back( i );
}
```





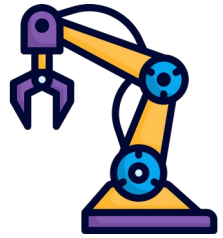
The vector container

```
vector<int> v( 10 );
cout<<v.size()<<endl;//???
for( int i=0; i<v.size(); ++i ){
        cout<<v[ i ]<<endl;
}

for( int i=0; i<10; ++i){
        v.push_back( i );
}
cout<<v.size()<<endl;//???

for( auto& a: v ){
   cout<< a <<" ";
}
```



push_back vs. emplace_back

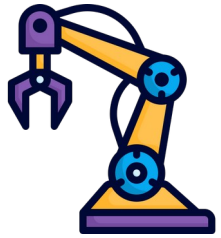
```
vector<Point> v;

for( int i=0; i<10; ++i){



    v.push_back(Point(i,i));
}
```



The vector container: typical errors

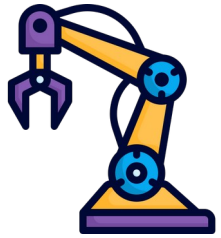
Find the error and correct it!

```
vector<int> v;
cout<<v.size()<<endl;//???
for( int i=0; i<10; ++i ){
    v[ i ] = i;
}

cout<<v.size()<<endl;//???
for( int i=0; i<v.size(); ++i ){
    cout<<v[ i ]<<endl;
}
```



The vector container: capacity and size

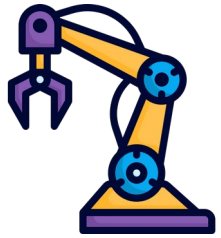
```
vector<int> v;
v.reserve( 10 );

cout << v.size() << endl;//???
cout << v.capacity() << endl;//???
```



The vector container: capacity and size

```
vector<int> v;
v.reserve( 10 );

cout << v.size() << endl;//???
cout << v.capacity() << endl;//???

-------------------------------------------------

vector<int> gy( 256 );
ifstream ifs("szoveg.txt"); int c;
while( (c = ifs.get() ) != -1 ){
          gy[ c ]++;
}
```

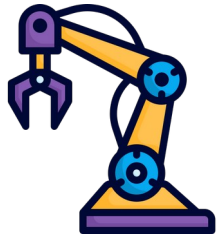
Purpose?



The vector - indexing

```
int Max = 100;
vector<int> v(Max);
//???...
for (int i = 0; i < 2*Max; i++) {
        cout << v[ i ]<<" ";
}
```

```
int Max = 100;
vector<int> v(Max);
for (int i = 0; i < 2*Max; i++) {
        cout << v.at( i )<<" ";
}
```



The vector - indexing

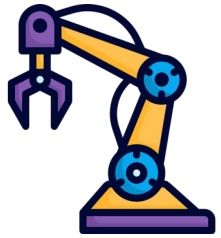
```
int Max = 100;
vector<int> v(Max);
//???...
for (int i = 0; i < 2*Max; i++) {
        cout << v[ i ]<<" ";
}
```

Efficient

```
int Max = 100;
vector<int> v(Max);
for (int i = 0; i < 2*Max; i++) {
        cout << v.at( i )<<" ";
}
```

Safe

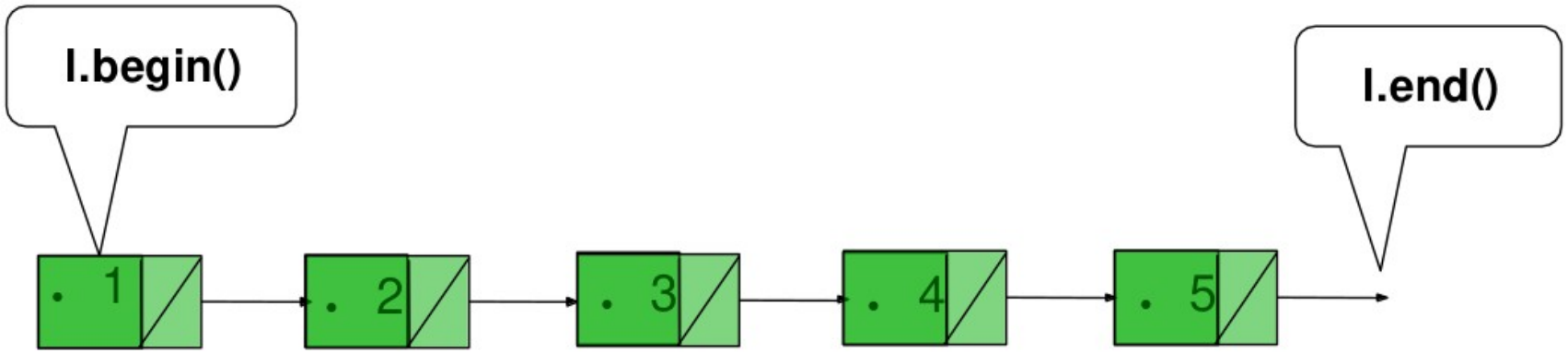
out_of_range exception

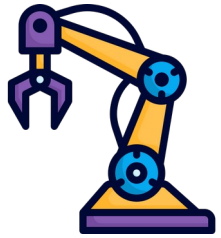


The list container

- doubly linked list

```
list<int> l;
for( int i=1; i<=5; ++i){
    l.push_back( i );
}
```

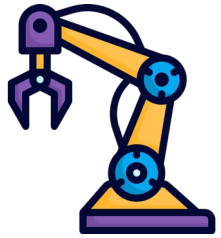




The deque container

- double ended vector

```
deque<int> l;
for( int i=1; i<=5; ++i){
    l.push_front( i );
}
```

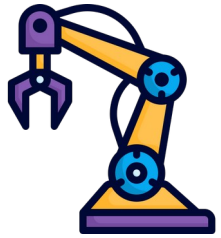


Algorithms - sort

```
template <class RandomAccessIterator>
void sort ( RandomAccessIterator first,RandomAccessIterator last );
```

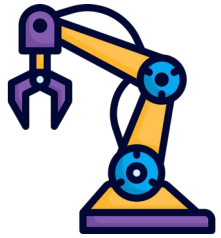
```
template <class RandomAccessIterator, class Compare>
void sort ( RandomAccessIterator first, RandomAccessIterator last,
            Compare comp );
```

- what to sort: **[first, last)**
- how to compare the elements:
  - **<**
  - **comp**



Algorithms - sort

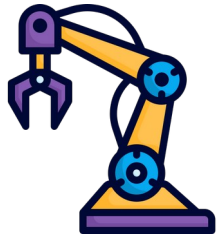
```
struct Rec {
    string name;
    string addr;
};
vector<Rec> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());
sort(vr.begin(), vr.end(), Cmp_by_addr());
```



Algorithms - sort

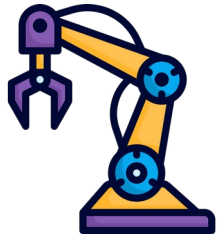
```
struct Cmp_by_name{
   bool operator()(const Rec& a, const Rec& b) const{
       return a.name < b.name;
   }
};


struct Cmp_by_addr{
   bool operator()(const Rec& a, const Rec& b) const{
        return a.addr < b.addr;
   }
};
```

function object



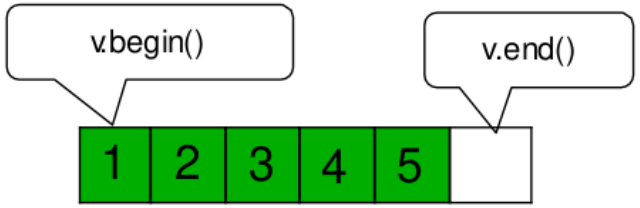
Iterators

- The *container* manages the contained objects but does not know about *algorithms*
- The *algorithm* works on data but does not know the internal structure of *containers*
- *Iterators* fit containers to algorithms

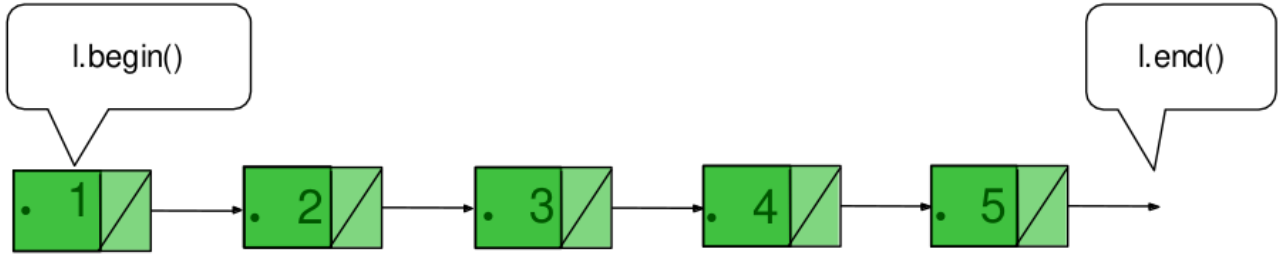


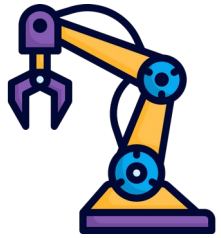
Iterator - *the glue*

```
int x[]={1,2,3,4,5}; vector<int>v(x, x+5);
int sum1 = accumulate(v.begin(), v.end(), 0);
```



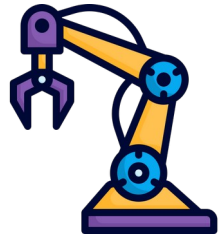

```
list<int> l(x, x+5);
double sum2 = accumulate(l.begin(), l.end(), 0);
```





Iterator - *the glue*

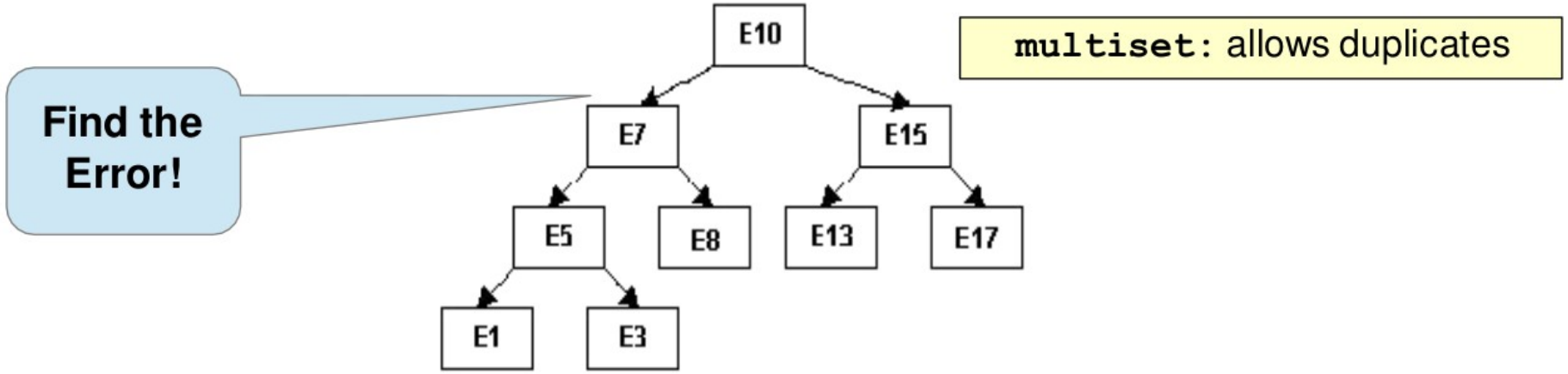
```
template<class InIt, class T>
T accumulate(InIt first, InIt last, T init){
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```



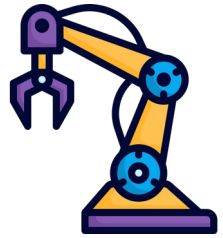
The `set` container

```
set< Key[, Comp = less<Key>]>
```

usually implemented as a balanced binary search tree

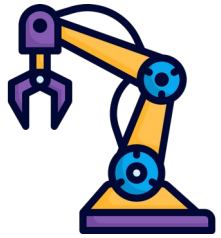


Source:http://www.cpp-tutor.de/cpp/le18/images/set.gif



The `set` container - usage

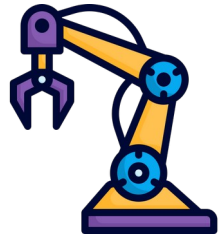
```
#include  <set>
using namespace std;

set<int> intSet;

set<Person> personSet1;


set<Person, PersonComp> personSet2;
```



The `set` container - usage

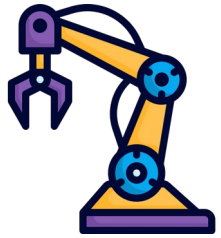
```
#include <set>

set<int> intSet;

set<Person> personSet1;

set<Person, PersonComp> personSet2;
```

. bool operator<(const Person&, const Person&)



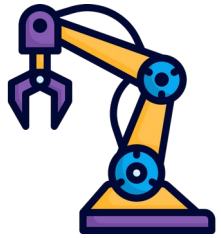
The `set` container - usage

```
. <
#include <set>

. bool operator<(const Person&, const Person&)

set<int> intSet;

set<Person> personSet1;

. struct PersonComp{
.   bool operator() ( const Person&, const Person&
  );
. };

set<Person, PersonComp> personSet2;
```



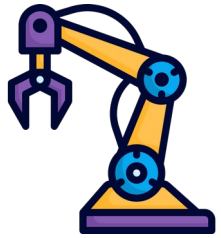
The `set` container - usage

```
#include <set>
set<int> mySet;
while( cin >> nr ){
   mySet.insert( nr );
}

set<int>::iterator iter;
for (iter=mySet.begin(); iter!=mySet.end(); ++iter){
   cout << *iter << endl;
}
```



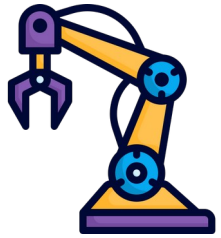
The `set` container - usage

```
set<int>::iterator iter;
for (iter=mySet.begin(); iter!=mySet.end(); ++iter){
   cout << *iter << endl;
}
-----------------------------------------------------------------------


for( auto& i: mySet ){
   cout<<i<<endl;
}
```



Time and Space Complexity

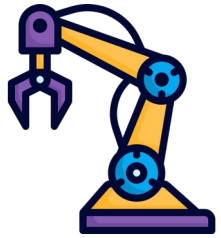
Function	Time Complexity	Space Complexity
M.find(x)	O(log n)	O(1)
M.insert(pair<int, int> (x, y)	O(log n)	O(1)
M.erase(x)	O(log n)	O(1)
M.empty( )	O(1)	O(1)
M.clear( )	Theta(n)	O(1)
M.size( )	O(1)	O(1)



The `multiset` container - usage

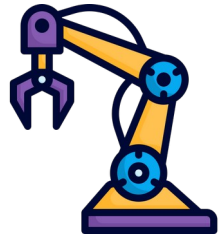
```
multiset<int> mySet;
size_t nrElements = mySet.count(12);


multiset<int>::iterator iter;
iter = mySet.find(10);


if (iter == mySet.end()){
    cout<<"The element does not exist"<<endl;

}
```

?



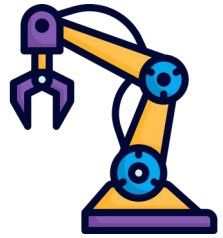
The `multiset` container - usage

```
multiset<int> mySet;
auto a = mySet.find(10);


if (a == mySet.end()){
    cout<<"The element does not exist"<<endl;

}
```



The `set` container - usage

```
class PersonCompare;
class Person {
    friend class PersonCompare;
    string firstName;
    string lastName;
    int yearOfBirth;
public:
    Person(string firstName, string lastName, int yearOfBirth);
    friend ostream& operator<<(ostream& os, const Person& person);
};
```



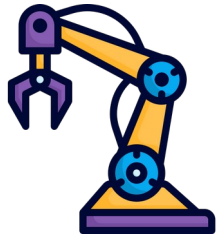
The `set` container - usage

```
class PersonCompare {
 public:
   enum Criterion { NAME, BIRTHYEAR};
 private:
   Criterion criterion;
 public:
   PersonCompare(Criterion criterion) : criterion(criterion) {}
   bool operator()(const Person& p1, const Person& p2) {
       switch (criterion) {
           case  NAME:  //
           case BIRTHYEAR: //
       }
   }
};
```

function object

state

behaviour

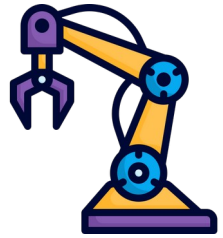


The `set` container - usage

```
set<Person, PersonCompare> s( PersonCompare::NAME );
s.insert(Person("Biro", "Istvan", 1960));
s.insert(Person("Abos", "Gergely", 1986));
s.insert(Person("Gered","Attila", 1986));
-------------------------------------------------------------------
for( auto& p: s){
    cout << p <<endl;
}
```

?

C++ 2011

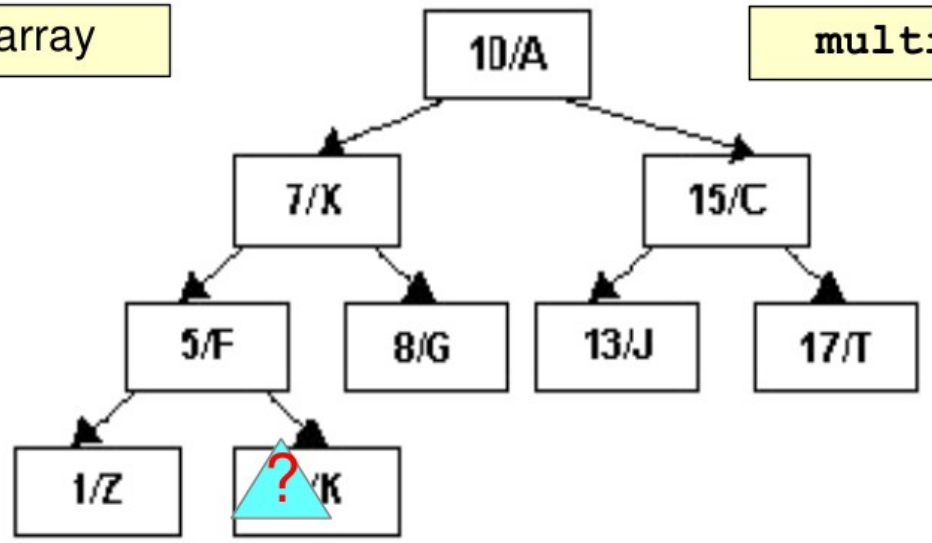


The `map` container

```
map< Key, Value[,Comp = less<Key>]>
```

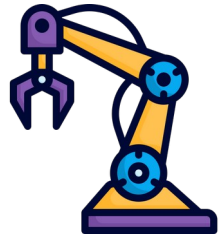
usually implemented as a balanced binary tree

map: associative array



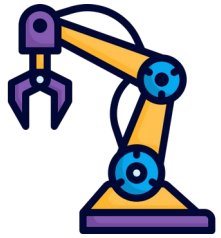
multimap: allows duplicates

Source: http://www.cpp-tutor.de/cpp/le18/images/map.gif



The map container - usage

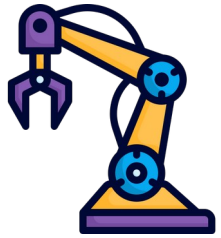
```
#include <map>
map<string,int> products;

products.insert(make_pair("tomato",10));
products.insert({"onion",3});

products["cucumber"] = 6;
cout<<products["tomato"]<<endl;
```



The map container - usage

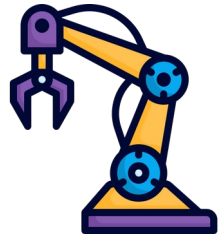
```
#include <map>
map<string,int> products;

products.insert(make_pair("tomato",10));

----------------------------------------

products["cucumber"] = 6;


cout<<products["tomato"]<<endl;
```

Difference between
[ ] and insert!!!



Difference between [ ] and insert

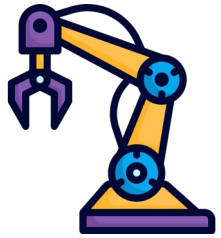
```
map<string, int> products;
products.insert({"tomato", 10});
printProducts(products); //Output?

products.insert({"tomato", 100});
printProducts(products); //Output?

products["tomato"] = 100;
printProducts(products); //Output?
```



The `map` container - usage

```
#include <map>
using namespace std;
int main ()
{
    map < string , int > m;
    cout << m. size () << endl; // 0
    if( m["c++"] != 0 ){
        cout << "not 0" << endl;
    }
    cout << m. size () << endl ; // 1
}
```

[ ] side effect



The `map` container - usage

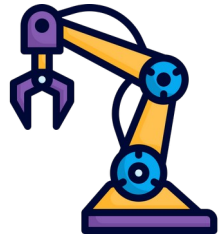
```
typedef map<string,int>::iterator MapIt;
for(MapIt it= products.begin(); it != products.end(); ++it){
   cout<<(it->first)<<" : "<<(it->second)<<endl;
}
```

```
for( auto& i: products ){
    cout<<(i.first)<<" : "<<(i.second)<<endl;
}
```

C++ 2011

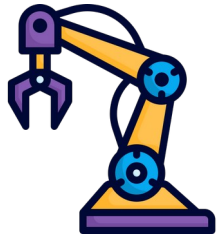
```
for( auto& [key, value]: products ){
    cout<< key <<" : "<< value<<endl;
}
```

C++ 2017



The `multimap` container - usage

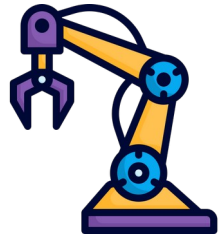
```
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest"));
cities.insert(make_pair("HU", "Szeged"));
cities.insert(make_pair("RO", "Seklerburg"));
cities.insert(make_pair("RO", "Neumarkt"));
cities.insert(make_pair("RO", "Hermannstadt"));

typedef multimap<string, string>::iterator MIT;
pair<MIT, MIT> ret = cities.equal_range("HU");
for (MIT it = ret.first; it != ret.second; ++it)     {
    cout << (*it).first <<"\t"<<(*it).second<<endl;
}
```



The `multimap` container - usage

```
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest"));
cities.insert(make_pair("HU", "Szeged"));
cities.insert(make_pair("RO", "Seklerburg"));
cities.insert(make_pair("RO", "Neumarkt"));
cities.insert(make_pair("RO", "Hermannstadt"));


auto ret = cities.equal_range("HU");
for (auto& [country, city]: cities){
    cout << country <<"\t"<< city <<endl;
}
```

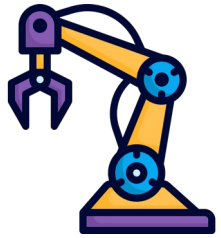
ROBOTICS CORNER



The `multimap` container - usage

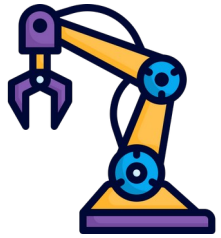
multimaps do not provide **operator[]**
Why???

```
multimap<string, string> cities;
cities.insert(make_pair("HU", "Budapest"));
cities.insert(make_pair("HU", "Szeged"));
cities.insert(make_pair("RO", "Seklerburg"));
cities.insert(make_pair("RO", "Neumarkt"));
cities.insert(make_pair("RO", "Hermannstadt"));


auto ret = cities.equal_range("HU");
for (auto& [country, city]: cities){
    cout << country <<"\t"<< city <<endl;
}
```



The `set/map` container - removal

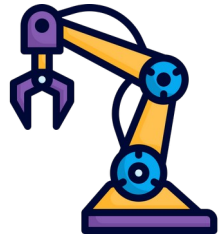
```
void erase ( iterator position );

size_type erase ( const key_type& x );

void erase ( iterator first, iterator last );
```



The `set` – pointer key type

Output??

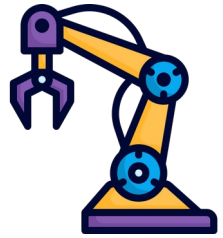
```
set<string *> animals;
animals.insert(new string("monkey"));
animals.insert(new string("lion"));
animals.insert(new string("dog"));
animals.insert(new string("frog"));

for( auto& i: animals ){
   cout<<*i<<endl;
}
```



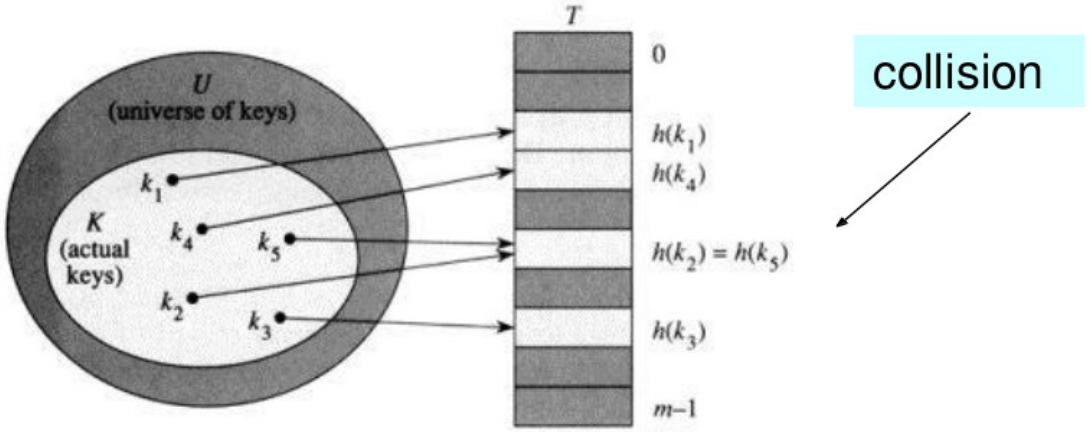
The `set` – pointer key type

Corrected

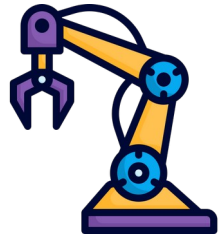
```
  struct StringComp{
     bool operator()(const string* s1,
                     const string * s2){
        return *s1 < *s2;
     }
  };
  set<string*, StringComp> animals;
  animals.insert(new string("monkey"));
  animals.insert(new string("lion"));
  animals.insert(new string("dog"));
  animals.insert(new string("frog"));
-------------------------------------------------------------------------------


  for( auto& animal: animals ){
      cout<< *animal <<endl;
  }
```



Hash Tables

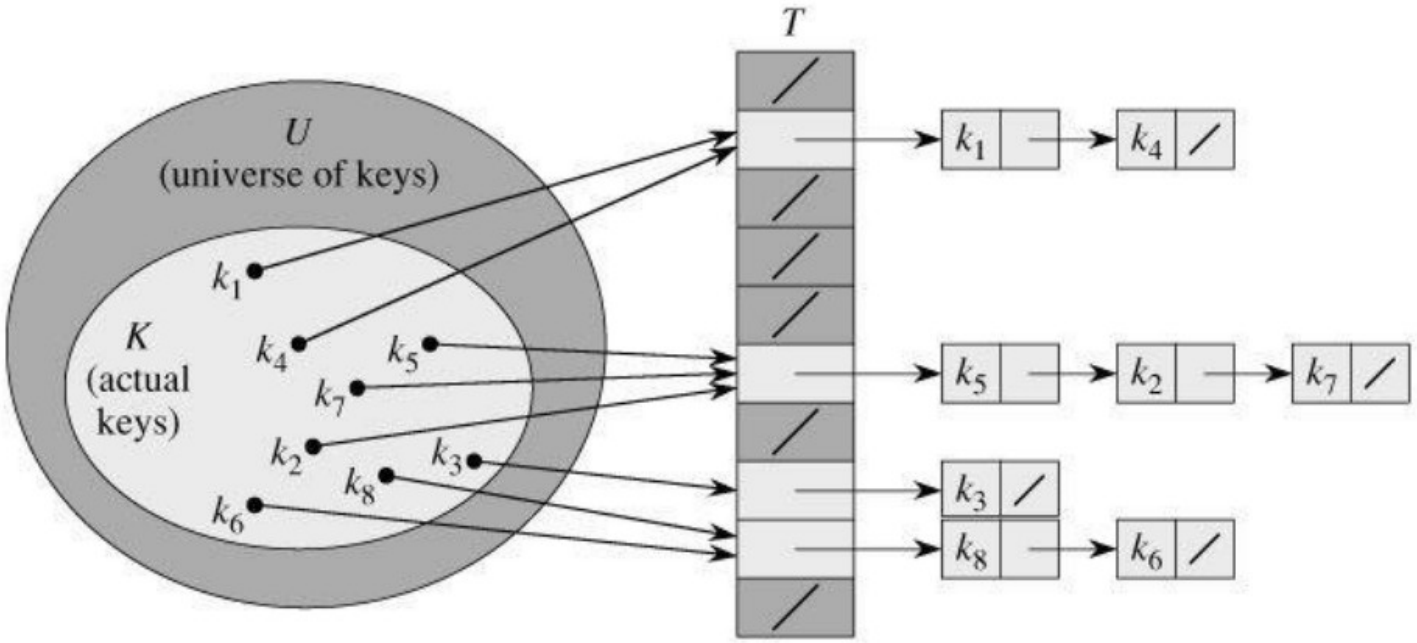


http://web.eecs.utk.edu/~huangj/CS302S04/notes/extendibleHashing.htm

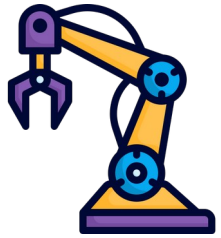


Hash Tables

Collision resolution by chaining

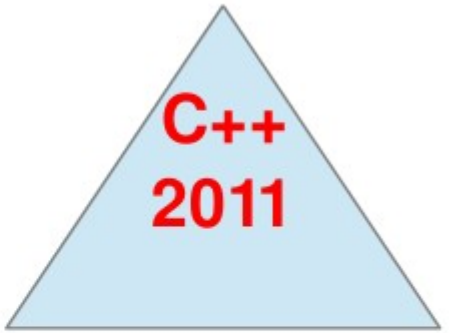


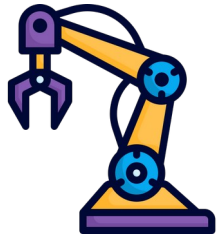
Source: http://integrator-crimea.com/ddu0065.html



Unordered Associative Containers - Hash Tables

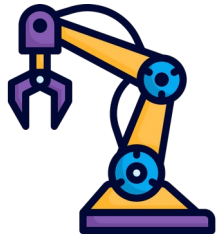
- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap





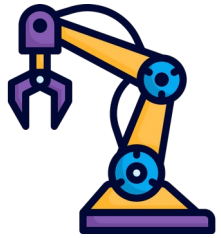
Unordered Associative Containers

- The STL standard does not specify which collision handling algorithm is required
  - most of the current implementations use linear chaining
  - a lookup of a `key` involves:
    - a hash function call `h(key)` – calculates the index in the hash table
    - compares `key` with other keys in the linked list



Hash Function

- *perfect hash*: no collisions
- lookup time: `O(1)` - constant
- there is a default hash function for each STL hash container

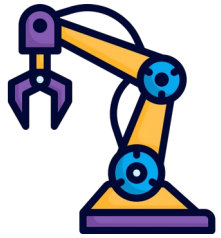


The unordered_map container

```
template <class Key, class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc= std::allocator<pair<const Key, T>>>
class unordered_map;
```

Template parameters:

- Key – key type
- T – value type
- Hash – hash function type
- Pred – equality type

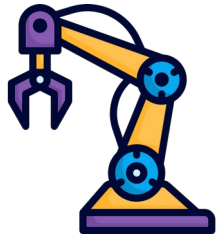


The unordered_set container

```
template <class Key,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc= std::allocator<pair<const Key, T>>>
class unordered_set;
```

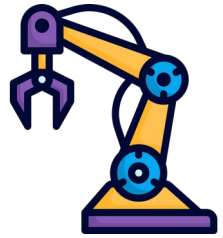
Template parameters:

- Key - key type
- Hash - hash function type
- Pred - equality type



Which container to use?

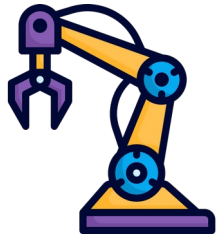
- implement a PhoneBook, which:
  - stores names associated with their phone numbers;
  - names are unique;
  - one name can have multiple phone numbers associated;
  - provides `O(1)` time search;



Which container to use?

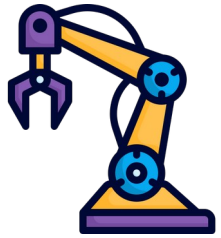
- Usage:

```
PhoneBook pbook;

pbook.addItem("kata","123456");
pbook.addItem("timi","444456");
pbook.addItem("kata","555456");
pbook.addItem("kata","333456");
pbook.addItem("timi","999456");
pbook.addItem("elod","543456");

cout<<pbook<<endl;
```



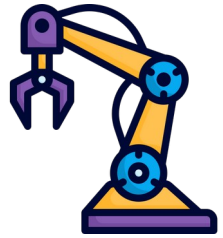
unordered_map: example

```
class PhoneBook {
    unordered_map<string, vector<string>> book;
public:
    void addItem(const string& name, const string& phone);

    bool removeItem(const string& name, const string& phone);

    vector<string> findItem(const string& name);


    friend ostream& operator<<(ostream& os, const PhoneBook& book);

};
```



unordered_map: example

```
void PhoneBook::addItem(const string &name, const string &phone){
    this->book[name].push_back(phone);
}

bool PhoneBook::removeItem(const string &name, const string &phone){
      // Locate the name → use  map.at(key) + try - catch
      // If the name does not exist
      //          → return false
      // Else
      //     locate the given phone in the vector associated to the
      //     name and delete it
      //      In case of empty phone list delete the map entry too
      //              → return true
}
```

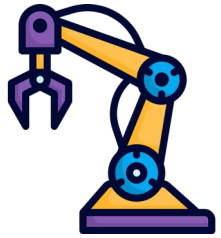


C++/Java

	C++	Java
Objects	`X x;` `X * px = new X();`	`X x = new X();`

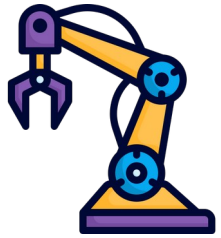
Parameter passing	`void f( X x );`
	`void f( X * px);`
	`void f( X& rx);`
	`void f( const X&rx);`
		`void f( X x );`
		`//pass through reference`

run-time binding	only for `virtual` functions	for each function (except static functions)
memory management	explicit (*2011 - smart pointers!*)	implicit (garbage collection)
multiple inheritance	yes	no
interface	no (*abstract class with pure virtual functions!*)	yes



Algorithms

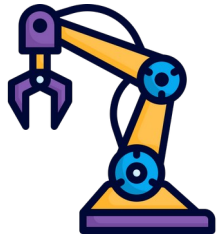
- OOP **encapsulates** *data* and *functionality*
  - data + functionality = object
- The STL separates the *data* (**containers**) from the *functionality* (**algorithms**)
  - only partial separation



Algorithms – why separation?

STL principles:

- algorithms and containers are independent
- (almost) any algorithm works with (almost) any container
- iterators mediate between algorithms and containers
  - provides a standard interface to traverse the elements of a container in sequence

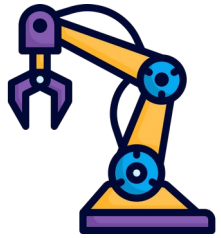


Algorithms

Which one should be used?

```
set<int> s;
set<int>::iterator it = find(s.begin(), s.end(), 7);
if( it == s.end() ){
   //Unsuccessful
}else{
  //Successful
}
```

```
set<int> s;
set<int>::iterator it = s.find(7);
if( it == s.end() ){
   //Unsuccessful
}else{
  //Successful
}
```



Algorithms

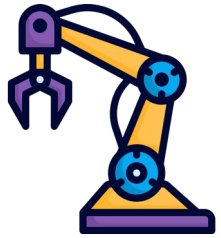
Which one should be used?

```
set<int> s;
set<int>::iterator it = find(s.begin(), s.end(), 7);
if( it == s.end() ){
   //Unsuccessful
}else{
  //Successful
}
```

O(n)

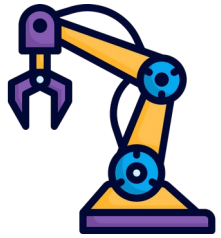
```
set<int> s;
set<int>::iterator it = s.find(7);
if( it == s.end() ){
   //Unsuccessful
}else{
  //Successful
}
```

O(log n)



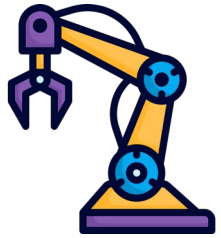
Algorithm categories

- Utility algorithms
- Non-modifying algorithms
  - Search algorithms
  - Numerical Processing algorithms
  - Comparison algorithms
  - Operational algorithms
- Modifying algorithms
  - Sorting algorithms
  - Set algorithms



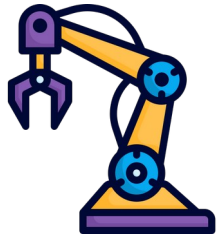
Utility Algorithms

- `min_element()`
- `max_element()`
- `minmax_element()` **C++11**
- `swap()`



Utility Algorithms

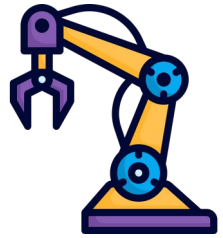
```
vector<int>v = {10, 9, 7, 0, -5, 100, 56, 200, -24};

auto result = minmax_element(v.begin(), v.end() );

cout<<"min: "<<*result.first<<endl;
cout<<"min position: "<<(result.first-v.begin())<<endl;

cout<<"max: "<<*result.second<<endl;
cout<<"max position: "<<(result.second-v.begin())<<endl;
```



Non-modifying algorithms

Search algorithms

- `find()`, `find_if()`, `find_if_not()`, `find_first_of()`
- `binary_search()`
- `lower_bound()`, `upper_bound()`, `equal_range()`
- `all_of()`, `any_of()`, `none_of()`
- ...

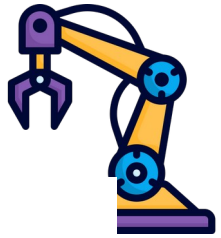


Non-modifying algorithms

Search algorithms - Example

```
bool isEven (int i) { return ((i%2)==0); }

typedef vector<int>::iterator VIT;

int main () {
  vector<int> myvector={1,2,3,4,5};
  VIT it= find_if (myvector.begin(), myvector.end(), isEven);
  cout << "The first even value is " << *it << '\n';
  return 0;
}
```

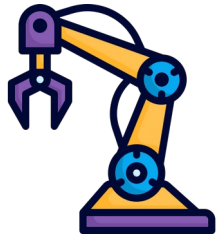
auto



Non-modifying algorithms

Numerical Processing algorithms

- `count()`, `count_if()`
- `accumulate()`
- ...

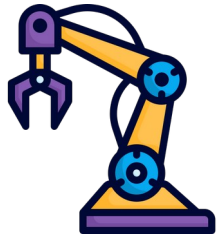


Non-modifying algorithms

Numerical Processing algorithms - Example

```
bool isEven (int i) { return ((i%2)==0); }


int main () {
  vector<int> myvector={1,2,3,4,5};
  int n = count_if (myvector.begin(), myvector.end(), isEven);
  cout << "myvector contains " << n  << " even values.\n";
  return 0;
}
```

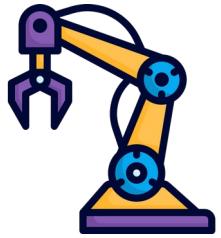
[ ] (int i){ return i %2 == 0; }



Non-modifying algorithms

Comparison algorithms

- `equal()`
- `mismatch()`
- `lexicographical_compare()`



Non-modifying algorithms

Problem

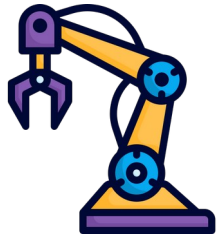
It is given **strange alphabet** – the order of characters are unusual.

Example for a strange alphabet: `{b, c, a}.`
Meaning: `'b'->1, c->'2', 'a' ->3`

In this alphabet: `“abc” >”bca”`

Questions:

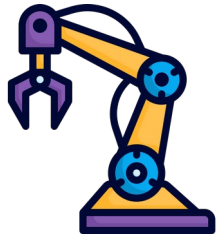
- How to represent the alphabet (which container and why)?
- Write a function for string comparison using the strange alphabet.



Non-modifying algorithms

Comparison algorithms - Example

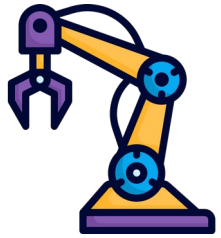
```
// strange alphabet: 'a' ->3, 'b'->1, c->'2'
map<char, int> order;


// Compares two characters based on the strange order
bool compChar( char c1, char c2 ){
    return order[c1]<order[c2];
}
// Compares two strings based on the strange order
bool compString(const string& s1, const string& s2){
    return lexicographical_compare(
        s1.begin(), s1.end(), s2.begin(), s2.end(), compChar);
}
```



Non-modifying algorithms

Comparison algorithms - Example

```
// strange alphabet: 'a' ->3, 'b'->1, c->'2'
map<char, int> order;

// Compares two strings based on the strange order
struct CompStr{
    bool operator()(const string& s1, const string& s2){
        return lexicographical_compare(
            s1.begin(), s1.end(), s2.begin(), s2.end(),
            [](char c1, char c2){return order[c1]<order[c2];} );
    }
}

set<string, CompStr> strangeSet;
```

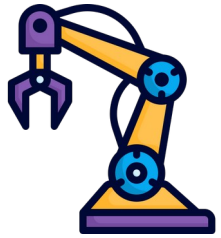


Non-modifying algorithms

Operational algorithms

- for_each()

```
void doubleValue( int& x){
    x *= 2;
}

vector<int> v ={1,2,3};
for_each(v.begin(), v.end(), doubleValue);
```



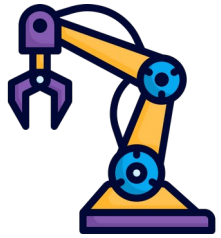
Non-modifying algorithms

Operational algorithms

- for_each()

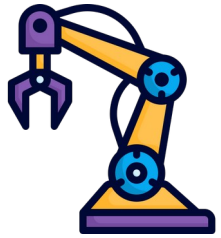
```
void doubleValue( int& x){
    x *= 2;
}

vector<int> v ={1,2,3};
for_each(v.begin(), v.end(), doubleValue);
```

```
for_each(v.begin(), v.end(), []( int& v){ v *=2;});
```



Modifying algorithms

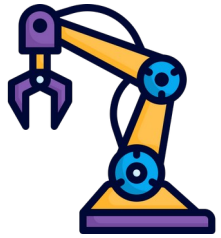
- `copy()`, `copy_backward()`
- `move()`, `move_backward()` **C++11**
- `fill()`, `generate()`
- `unique()`, `unique_copy()`
- `rotate()`, `rotate_copy()`
- **`next_permutation()`, `prev_permutation()`**
- **`nth_element()`** -nth smallest element



Modifying algorithms

Permutations

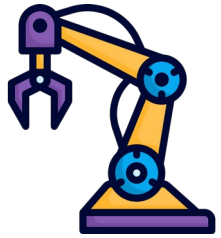
```
void print( const vector<int>& v){
    for(auto& x: v){
        cout<<x<<"\t";
    }
    cout << endl;
}
int main(){
    vector<int> v ={1,2,3};
    print( v );
    while( next_permutation(v.begin(), v.end())){
        print( v );
    }
    return 0;
}
```



Modifying algorithms

nth_element

```
double median(vector<double>& v) {
    int n = v.size();
    if( n==0 ) throw domain_error("empty vector");
    int mid = n / 2;
    // size is an odd number
    if( n % 2 == 1 ){
        nth_element(v.begin(), v.begin()+mid, v.end());
        return v[mid];
    } else{
        nth_element(v.begin(), v.begin()+mid-1, v.end());
        double val1 = v[ mid -1 ];
        nth_element(v.begin(), v.begin()+mid, v.end());
        double val2 = v[ mid ];
        return (val1+val2)/2;
    }
}
```



Thank You

Do you have any questions?

ROBOTICS CORNER