Templates, Function Objects, Namespaces and the C++ Standard Library

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

11/10/2016

Outline

TEMPLE UNIVERSITY

Outline

Generic Programming

C++: strict type system

```
void sort(int* a, int len);
void sort(float* a, int len);
void sort(double* a, int len);
```

- all of these functions are different in C++
- you have to explicitly create a function for each data type
- the data type is fixed at compile time

Generic Programming

C++: strict type system

```
void sort(int* a, int len);
void sort(float* a, int len);
void sort(double* a, int len);
```

- all of these functions are different in C++
- you have to explicitly create a function for each data type
- the data type is fixed at compile time

Python: duck typing

```
def sort(a):
    length = len(a)
    ...
```

- Python doesn't care what types come in as parameters
- It will try to use them
- Type checking is done at run time

If it quacks like a duck, looks like a duck, then it is a duck.

Generic Algorithms

In-Place Insertion Sort (**int** version)

```
void insertion_sort(int * a, int length)
{
    for(int j = 1; j < length; j++) {
        int key = a[j];
        int i = j - 1;
        while(i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

- in-place insertion sort works inside the original array
- does not need additional array, so requires less memory
- here we are sorting integer numbers

Generic Algorithms

In-Place Insertion Sort (**float** version)

```
void insertion_sort( float * a, int length)
{
    for(int j = 1; j < length; j++) {
        float key = a[j];
        int i = j - 1;
        while(i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

- sorting floating-point numbers works the same way
- the only change is the data type in some parts of the code

Generic Algorithms

In-Place Insertion Sort (`float` version)

```
void insertion_sort( float * a, int length)
{
    for(int j = 1; j < length; j++) {
        float key = a[j];
        int i = j - 1;
        while(i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

- sorting floating-point numbers works the same way
- the only change is the data type in some parts of the code
- Wouldn't it be nice if we could write this code only once and easily change the data type?

Generic Algorithms

In-Place Insertion Sort (`float` version)

```
void insertion_sort( float * a, int length)
{
    for(int j = 1; j < length; j++) {
        float key = a[j];
        int i = j - 1;
        while(i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

- sorting floating-point numbers works the same way
- the only change is the data type in some parts of the code
- Wouldn't it be nice if we could write this code only once and easily change the data type?
- $\Rightarrow$ **Template Functions**

Function Templates

In-Place Insertion Sort (template version)

```
template<typename T>
void insertion_sort( T * a, int length)
{
    for(int j = 1; j < length; j++) {
        T key = a[j];
        int i = j - 1;
        while(i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

- templates allow us to write generic algorithms which are parameterized by types
- a template instructs the compiler on how to generate a function for any given type
- when the compiler first sees a usage of the function with a specific type it will generate a function for that combination

Using a function template

```
int *   a = new int[10];
float * b = new float[10];

// compiler will create sort(int*, int)
insertion_sort(a, 10);

// compiler will create sort(float*, int)
insertion_sort(b, 10);
```

Template Specializations

```
// generic implementation
template<typename T>
void print(T value) {
    printf("NaN");
}

// specialization for int
template<>
void print(int value) {
    printf("%d", value);
}

// specialization for float
template<>
void print(float value) {
    printf("%2.2f", value);
}
```

```
print("Hello World"); // NaN
print(42.0f);         // 42.00
print(7);             // 7
```

- specializations allow you to add an implementation of a template function for a specific type combination
- the benefit is you can have a generic implementation for most types, and a tuned one for specific types

Class Templates

`int` Stack

```
class Stack {
     int data[100];
public:
     Stack();
     ~Stack();

     void push(int value);
     int pop();
};
```

`float` Stack

```
class Stack {
     float data[100];
public:
     Stack();
     ~Stack();

     void push(float value);
     float pop();
};
```

Class Templates

Generic Stack

```
template<typename T>
class Stack {
    T data[100];
public:
    Stack();
    ~Stack();

    void push(T value);
    T pop();
};
```

Usage

```
Stack<int> istack;
istack.push(10);
istack.push(20);

Stack<float> fstack;
fstack.push(10.0f);
fstack.push(20.0f);
```

Caution

- Template classes are generated at compile time
- There is no inheritance-like relationship between templates (e.g., `Stack<int>` is not compatible with `Stack<float>`)
- however, they can have common base classes

```
struct IPrintable {
      virtual void print() = 0;
};

template<typename T>
class Stack : public IPrintable {
public:
      ...
      virtual void print() { ... };
}
```

```
Stack<int> istack;
Stack<float> fstack;

IPrintable & a = istack;
IPrintable & b = fstack;

a.print();
b.print();
```

Outline

Generic Programming
- Function Templates
- Class Templates

Function Objects

Namespaces

C++ Standard Library
- Strings
- Sequential Containers
- Iterators
- Associative Containers
- Algorithms

Function Objects

Objects which acts like a function

- when having such an object, you can call them with parameters

```
less_than op;     // op is an object of class 'less_than'
bool result  = op(4, 3);
bool result2 = op(3, 4);
```

Function Objects

Objects which acts like a function

- when having such an object, you can call them with parameters

```
less_than op;     // op is an object of class 'less_than'
bool result  = op(4, 3);
bool result2 = op(3, 4);
```

Writing a class for a function object

```
struct less_than {
    bool operator()(int a, int b) {
        return a < b;
    }
};
```

```
struct greater_than {
    bool operator()(int a, int b) {
        return a > b;
    }
};
```

- Function object classes implement the **operator**() method

```
// sort in ascending order
sort_list(list, length, less_than());

// sort in descending order
sort_list(list, length, greater_than());
```

Template function which takes a function object as parameter

```
template<typename T>
void sort_list(int * a, int length, T compare)
{
    for(int j = 1; j < length; j++) {
        int key = a[j];
        int i = j - 1;
        while(i >= 0 && compare(key, a[i])) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
```

Live Demo: Using function objects for sorting

Outline

Avoiding name collisions

- a recurring problem in programming is finding good names for things
- every declaration in C/C++ has to have a unique name
- all names live in the global namespace
- once your code base grow you will eventually run out of names
- workaround: add prefixes to names
  - `initialize`
  - `subsystemA_initialize`
  - `subsystemB_initialize`

Namespaces

```
// in global namespace
void initialize();
class Info;

namespace SubsystemA {
    // in SubsystemA namespace
    void initialize();
    class Info;
}

namespace SubsystemB {
    // in SubsystemB namespace
    void initialize();
    class Info;
}

namespace SubsystemA {
    // again in SubsystemA namespace
    void cleanup();
}
```

- in C++ you can create your own namespaces to group functions, classes or global variables
- a namespace is defined using the **`namespace`** keyword followed by a name and a block
- anything inside the block is part of that namespace
- you can always add more to a namespace by just adding another block

Using names in namespaces

- names in a different namespace can be accessed with their fully qualified name `NamespaceName::symbol`

```
SubsystemA::initialize();
SubsystemB::initialize();
```

- you can also import a name into your current scope and use it without any prefix

```
using SubsystemB::initialize;
initialize(); // this is SubsystemB::initialize()
```

- or import all names from a namespace into your current scope

```
using namespace SubsystemA;
initialize(); // this is SubsystemA::initialize()
cleanup(); // this is SubsystemA::cleanup()
```

Nested namespaces

```
namespace Simulation {
    namespace Physics {
        ...
    }

    namespace Data {
        ...
    }

    namespace Output {
        void dump_to_file();
    }
}

// usage example
Simulation::Output::dump_to_file();
```

- namespaces can be nested
- this allows you to group your code into logical units

Outline

C++ Standard Library

- written using the C++ core language
- to use it, you only need to include headers
- provides classes, functions and templates to solve general computing
- all of them live inside the `std` namespace problems
- grouped into several categories
  - General utilities library
  - **Strings library**
  - Localization library
  - **Container library**
  - **Iterators library**
  - **Algorithms library**
  - Numerics library
  - **Input/Output library**
  - Regular Expressions library (C++11)
  - Thread support library (C++11)
  - etc.

Strings

```
#include <string>
#include <stdio.h>
using namespace std;

int main() {
    string first = "Richard";
    string last  = "Berger";

    // use + for concatination
    string full = first + " " + last;

    // access individual characters with indexing
    for(int i = 0; i < full.length(); ++i) {
        printf("%c\n", full[i]);
    }

    // access c-string
    printf("%s\n", full.c_str());
    return 0;
}
```

Strings

Benefits of using the string class

- automatic memory management of string
- string can grow and shrink dynamically
- offers familiar access to characters like with C-Strings

Other useful member functions

```
String str = "First, solve the problem. Then, write the code.";
string str2 = str.substr(7, 5); // "solve"

size_t pos = str.find("Then");  // find start position of "Then"
string str3 = str.substr(pos);  // get string starting from "Then"
```

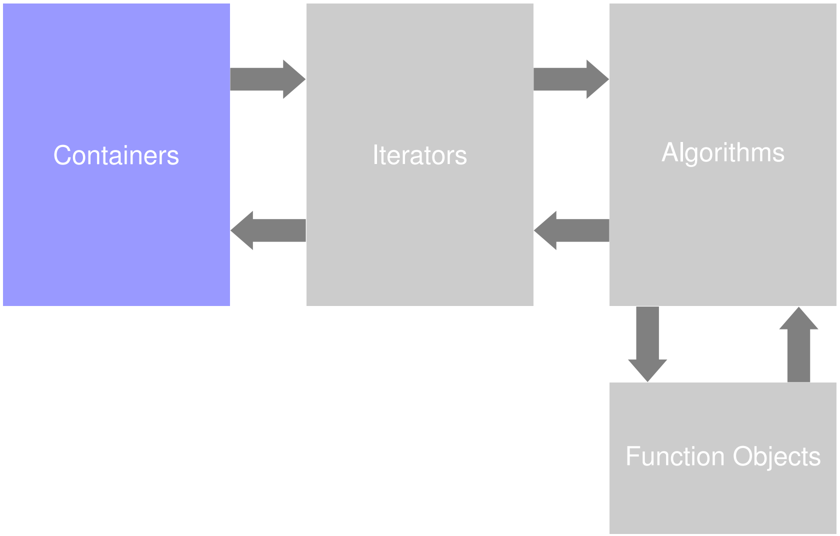
Basic I/O with Streams

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

int main() {
    // write to stdout, endl ends line
    cout << "Hello World!" << endl;

    // write out combination of values and strings
    int a = 42;
    float f = 3.14f;
    cout << "a = " << a << ", f = " << f << endl;

    // write to stderr
    cerr << "Error!" << endl;

    // read value
    int parameter;
    cin >> parameter;
    return 0;
}
```

C++ Standard Template Library (STL)

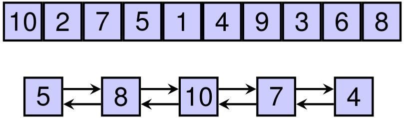
- Common Algorithms and Data Structures
- Optimized for General Computing
- Uses C++ template mechanisms extensively
- no virtual calls

C++ Standard Template Library (STL)

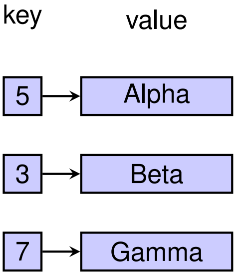


Containers

Sequential Containers



Associative Containers



Containers

Operations

- insertion
- accessing
- searching
- deletion

Variants

- ordered
- unordered

Sequential Containers

- `std::vector` (flexible sequence)
- `std::deque` (double-ended queue)
- `std::list` (double-linked list)
- `std::array` (fixed sequence, C++11)
- `std::forward_list` (single-linked list, C++11)

- `std::stack`
- `std::queue`

C-Arrays

- simplest sequential data structure
- data is stored in range $[0, nelements)$
- has a fixed size N
- wasteful if we don't use entire storage
- consecutive memory, which allows efficient access

C-Arrays

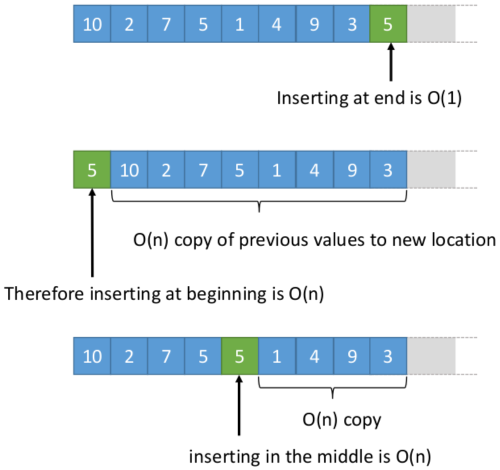
```
int a[10000] { 10, 2, 7, 5, 1, 4, 9, 3};
int nelements = 8;
```

insertion at the end is O(1)

```
a[nelements++] = 5;
```

insertion at the beginning is O(n)

```
for(int i = nelements; i > 0; --i) {
    a[i] = a[i-1];
}
a[0] = 5;
nelements++;
```



std::vector - Creation

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

```
// empty construction
vector<int> a;

// vector of size 10 and default initialized elements
vector<int> a(10);

// vector of size 100 with initial value of -1
vector<int> a(100, -1);

// C++ 11 initializer lists
vector<int> a { 3, 5, 7, 9, 11 };
```

`std::vector` - Inserting and removing elements

```
// insertion at end
a.push_back(3);
a.push_back(5);
a.push_back(7);

// delete at end
a.pop_back();

// insertion at beginning
a.insert(a.begin(), new_value);
```

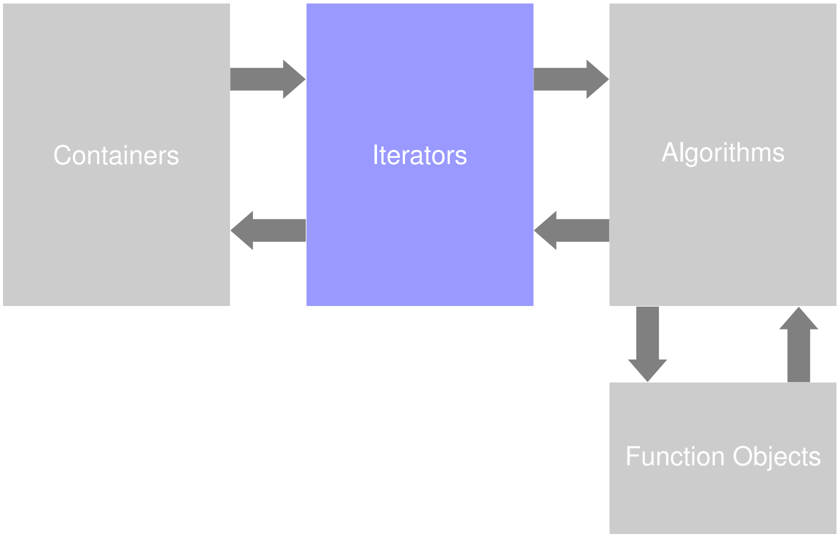
std::vector - Accessing elements

```
// accessing elements just like arrays
for(int i = 0; i < a.size(); i++) {
    printf("%d\n", a[i]);
}
```

Note

But there is another way using iterators. . .

C++ Standard Template Library (STL)



Looping over an array

Indexing

```
int a[100];

for(int i = 0; i < 100; ++i) {
    a[i] = /* computation */
}
```

With pointers

```
int * b = &a[0];
int * e = &a[0] + 100;

for(int * p = b; p != e; ++p){
    *p = /* computation */
}
```

Looping over a `std::vector`

Indexing

```
std::vector<int> a(100);

for(int i = 0; i < a.size(); ++i) {
    a[i] = /* computation */
}
```

- `std::vector` behaves just like an array and allows you to access each element using an index

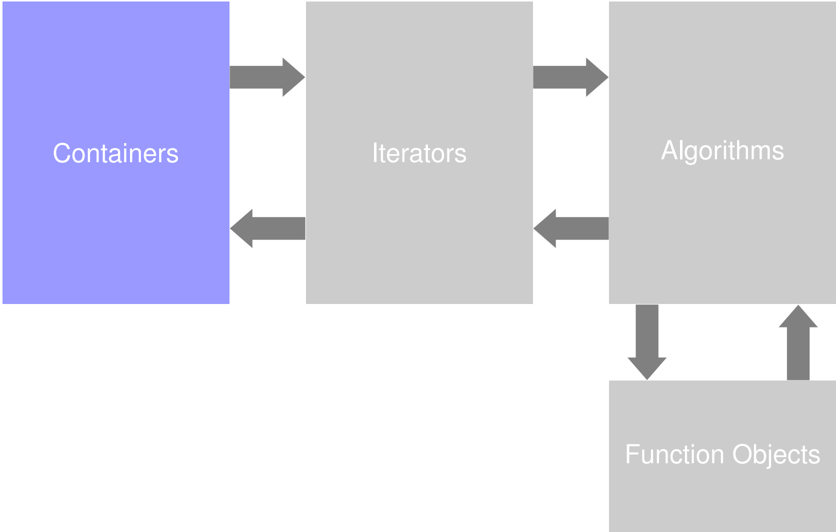
Generic way of looping over any STL container

```
// using iterators (act like pointers)
for(vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    printf("%d\n", *it);
}

// using iterators (C++ 11, auto)
for(auto it = a.begin(); it != a.end(); ++it) {
    printf("%d\n", *it);
}

// C++11 for each
for(auto element : a) {
    printf("%d\n", element);
}
```

Note

- iterators are not just like pointers
- pointers **are iterators** as well
- $\Rightarrow$ anything in the STL that accepts an iterator accepts pointers from arrays

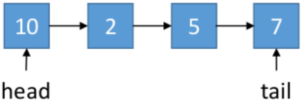
Back to containers. . .



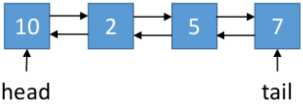
std::list

- double-linked list
- list elements connected through pointers
- first element (head) and last element (tail) are always known
- insertion/deletion at both ends in O(1)
- insertion in the middle is also cheaper
  - finding insertion location is O(n) compared to O(1) with C-Arrays
  - but insertion itself happens in O(1) instead of O(n) copies
- dynamic size
- distributed in memory, less efficient access

Single Linked-List:
only pointer of next element



Double Linked-List:
pointer of previous and next element



std::list - Creation

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

```
// empty construction
list<int> a;

// list of size 10 and default initialized elements
list<int> a(10);

// list of size 100 with initial value of -1
list<int> a(100, -1);

// C++ 11 initializer lists
list<int> a { 3, 5, 7, 9, 11 };
```

std::list - Inserting and removing elements

```
// insertion at beginning
a.push_front(2);
a.push_front(4);
a.push_front(8);

// insertion at end
a.push_back(3);
a.push_back(5);
a.push_back(7);

// delete at beginning
int val = a.pop_front();

// delete at end
int val2 = a.pop_back();
```

```
// insert at element 2
list<int>::iterator it = a.begin();
++it;
a.insert(it, 10);

// access first element
int first = a.front();

// access last element
int last = a.back();
```

`std::list` - Accessing elements

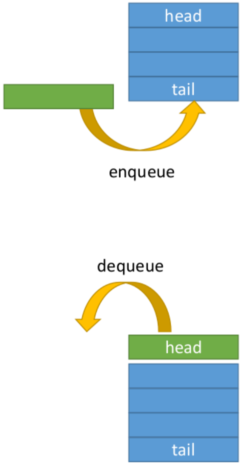
```
// accessing elements using iterators
for(list<int>::iterator it = a.begin(); it != a.end(); ++it) {
    printf("%d\n", *it);
}

// C++ 11 with auto
for(auto it = a.begin(); it != a.end(); ++it) {
    printf("%d\n", *it);
}

// C++ 11 range-based loop
for(auto element : a) {
    printf("%d\n", element);
}
```

Queue

- First-In-First-Out (FIFO) data structure
- adapter which uses other containers for storage
  - `std::list`
  - `std::deque`
- Operations:

  enqueue: put element in queue (insert at tail)

  dequeue: get first element in queue (remove head)



Queue

```cpp
#include <queue>
#include <deque>
#include <list>
```

```cpp
// use default implementation
queue<int> q;

// use list<int> for queue
queue<int, list<int> > q2;

// use deque<int> for queue
queue<int, deque<int> > q3;
```

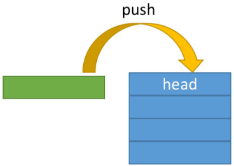
```cpp
// insert element
q.push(3);
q.push(5);
q.push(7);

// remove next element
q.pop(); // 3
q.pop(); // 5
q.pop(); // 7
```

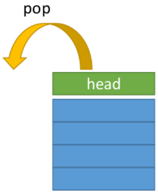
Stack

- Last-In-First-Out (LIFO) data structure
- adapter which uses other containers for storage
  - `std::vector`
  - `std::deque`
  - `std::list`
- Operations:

  push: put element on stack (insert as head)

  pop: get first element on stack (remove head)





Stack

```
#include <stack>
#include <deque>
#include <vector>
```

```
// use default implementation
stack<int> stk;

// use deque<int> for stack
stack<int, deque<int> > stk2;

// use vector<int> for stack
stack<int, vector<int> > stk3;
```

```
// insert element
stk.push(3);
stk.push(5);
stk.push(7);

// remove next element
stk.pop(); // 7
stk.pop(); // 5
stk.pop(); // 3
```

Associative Containers

- map a key to a value
- searching for a specific element in unsorted sequential container takes **linear** time O(n)
- getting a specific element from an associative container using a key can be as fast as **constant** time O(1)

Associative Containers

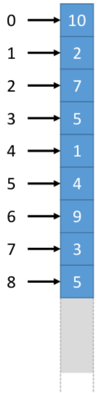
- `map`
- `set`
- `multimap`
- `multiset`

- `unordered_map` (C++11)
- `unordered_set` (C++11)
- `unordered_multimap` (C++11)
- `unordered_multiset` (C++11)

C-Array as Associative Container

- simplest associative data structure
- maps **integer number** to data
  - $0 \rightarrow$ `a[0]`
  - $1 \rightarrow$ `a[1]`
  - $2 \rightarrow$ `a[2]`
  - ...
- efficient access in O(1)
- ineffective storage
- limited to positive integer numbers as keys

```
int a[10000];
```



Ordered maps and sets

map

- maps **arbitrary keys** (objects, basic types) to **arbitrary values** (objects, basic types)
- Basic idea: if keys are sortable, we can store nodes in a data structure sorted by its keys.
- Such sorted data structures can be searched more quickly, e.g. with binary search
- Elements are ordered by key, not by insertion order
- Worst case lookup time is O(log(n))

set

- only stores unique values
- values are sorted
- any duplicate insertions are ignored

std::map

```cpp
#include <map>
#include <string>
using namespace std;
```

```cpp
map<string, string> capitals;

// setting value for key
capitals["Austria"] = "Vienna";
capitals["France"] = "Paris";
capitals["Italy"] = "Rome";

// getting value from key
cout << "Capital of Austria: " << capitals["Austria"] << endl;
string & capital_of_france = capitals["France"];

cout << "Capital of France: " << capitals << endl;
```

```
// check if key is set
if (capitals.find("Spain") != capitals.end()) {
    cout << "Capital of Spain is " << capitals["Spain"]
else {
    cout << "Capital of Spain not found!" << endl;
}
```

```cpp
// iterate over all elements
for (map<string, string>::iterator it = capitals.begin();
      it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto it = capitals.begin(); it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto & kv : capitals) {
    string & key = kv.first;
    string & value = kv.second;
    cout << "The capitol of " << key << " is " << value << endl;
}
```

std::set

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

```
set<int> s;

s.insert(10);
s.insert(20);
s.insert(30);
s.insert(10);

if (s.find(10) != s.end()) {
    // value is set
}

s.size(); // = 3
```

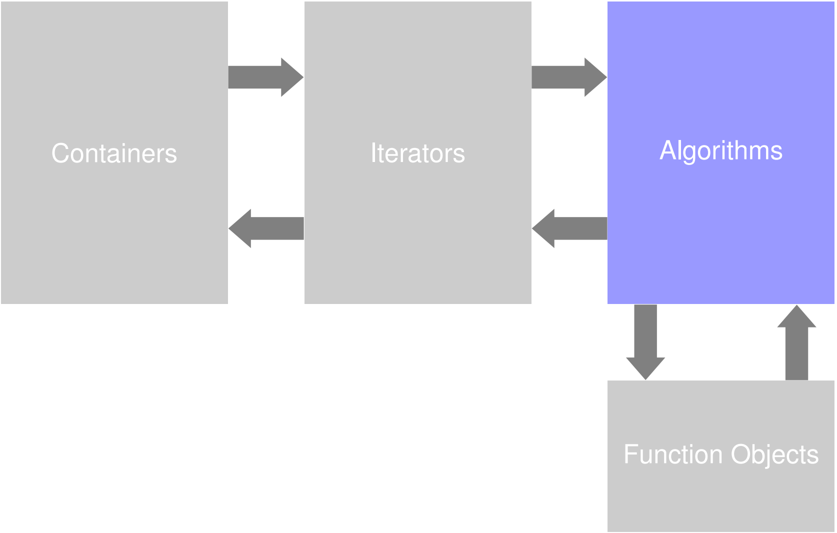
Unordered maps / Hash maps (C++11)

- also maps **arbitrary keys** (objects, basic types) to **arbitrary values** (objects, basic types)
- **on average, accessing a hash map through keys takes O(1)**
- unordered sequence
- similar to Python dictionaries

How hash maps work:

- for each key a number is generated using a **hash function** in O(1)
- this number is called a **hash code**
- each hash code can be mapped to a location called a **bin**
- a bin stores nodes with keys which map to the same hash code
- Lookup therefore consists of:
  - determining the hash code of the key O(1)
  - selecting the correct node inside the bin (worst case O(n))

C++ Standard Template Library (STL)



Algorithms

```
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
    vector<int> v {8, 2, 1, 3, 4, 6, 0, 9};

    // sort in ascending order
    sort(v.begin(), v.end());

    // equivalent
    sort(v.begin(), v.end(), less_than());

    // sort in descending order
    sort(v.begin(), v.end(), less_than());
    return 0;
}
```

`std::sort`

sort sequence from beginning to end in ascending order or with custom comparison function object

Algorithms

`std::sort(first, last, comparator)`

sort sequence from beginning to end in ascending order or with custom comparison function object

`std::stable_sort(first, last, comparator)`

like sort, but ensures that order of objects which are equal is not changed after sorting.

`std::swap(a,b)`

swap values of objects

`std::min(a,b)`

return smallest element

`std::max(a,b)`

return larger element

Algorithms

`std::find_if(first, last, predicate)`
look from `first` to `last` iterators and return an iterator to the first element where the predicate is **`true`**. Otherwise return `last`.

`std::count_if(first, last, predicate)`
look from `first` to `last` iterators return the number of elements for which predicate is **`true`**.

`std::partition(first, last, predicate)`
Rearrange all elements from `first` to `last` in such a way that all elements for which `predicate` is **`true`** come before elements for which it is **`false`**. It returns an iterator which points to the beginning of the second group.

`std::stable_partition(first, last, predicate)`
Same as `partition` but maintains original element order for each group.

Algorithms

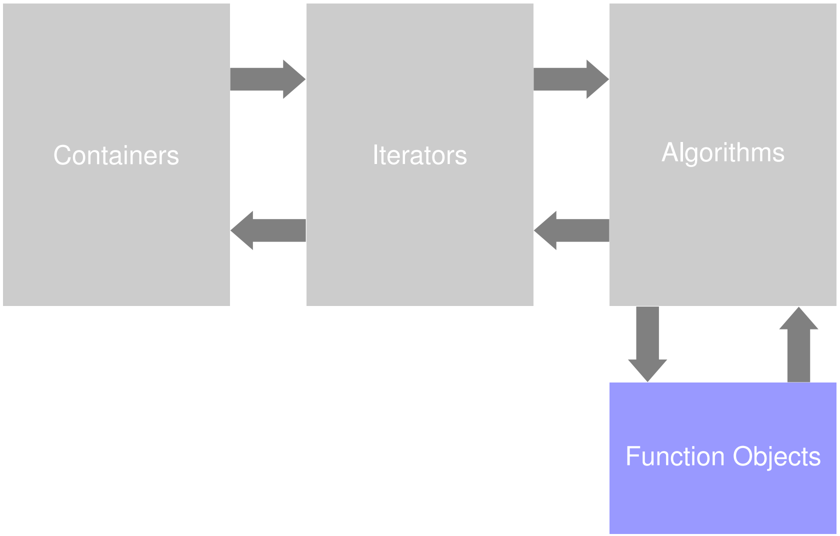
`std::partition(first, last, predicate)`

Rearrange all elements from `first` to `last` in such a way that all elements for which `predicate` is **`true`** come before elements for which it is **`false`**. It returns an iterator which points to the beginning of the second group.

`std::stable_partition(first, last, predicate)`

Same as `partition` but maintains original element order for each group.

C++ Standard Template Library (STL)



Function objects

Some function object classes defined in `<functional>` header

Arithmetic operations

- `plus`
- `minus`
- `multiplies`
- `divides`
- `modulus`
- `negate`

Comparison operations

- `equal_to`
- `not_equal_to`
- `multiplies`
- `greater`
- `less`
- `greater_equal`
- `less_equal`

General

`<algorithm>`

Provides many container algorithms

`<functional>`

Defines function objects which are designed to be used with standard algorithms

`<iterator>`

Classes and templates for working with iterators

`<chrono>` (C++11)

Provides classes for measuring time

I/O and Streams

`<fstream>`

Objects for file-based input and output

`<iostream>`

Objects for basic input and output (like stdout, stdin, stderr)

Strings

`<string>`

C++ string class

`<regex>` (C++11)

Utilities for pattern matching string using regular expressions

Numeric library

`<complex>`

Defines a class template for complex numbers and functions to manipulate them

`<random>`

Utilities for generating (pseudo-)random numbers

`<valarray>`

Defines class templates und functions to manipulate arrays of values