Containers, Iterators and Raw memory

Sequence Containers

- Values as a sequence

Name	
vector	Array that can be extended at the end
deque	Array that can be extended at the beginning and end
list	Doubly-linked list
forward_list	Singly-linked list
array	Fixed size array

Associative Containers

- Sorted data structure (normally a tree), search is O(log n)

Name	
set	A set of keys, sorted by keys
map	Collection of key-value pairs, sorted by keys, keys are unique
multiset/multimap	Not unique

Unorded Associative Containers

- Hash tables, search is O(1)
- unordered_set, unordered_map, unordered_multiset_set, unordered_multiset_map,

Example, Simple container

```
class FloatCont {
    static const int size = 100;
    float array[size];
public:
    FloatCont() :array() {};
    float* begin() { return array; }
    float* end() { return array + size; }
};

float TestFloatCont(FloatCont& v) {
    int i = 1;
    for (float* it = v.begin(); it != v.end(); ++it, ++i)
        *it = 1.0f / i;
    float sum = 0;
    for (float* it = v.begin(); it != v.end(); ++it)
        sum += *it;
    return sum;
}
```

Generalized Sum

```
template<class CONT, typename T>
T Sum(CONT c) {
    T sum = T();
    for (auto it = c.begin(); it != c.end(); ++it)
        sum += *it;
    return sum;
}
```

- typename or class in template – no difference

- `auto x=7.5;` Compiler set type of x to same as the initialization (double).

- `auto y=’x’;` (char)

Sum med iterators

```
template<class Iterator, typename T>
T SumIT(Iterator first, Iterator last) {
    T sum = T();
    for (auto it = first; it != last; ++it)
        sum += *it;
    return sum;
}
```

- STL-library uses iterators everywhere, one of the advantages is that it also handles cases when we only want to treat a part of the container.

- E.g. sum the first five elements

```
        Sum(v.begin(), v.begin()+5);
```

- If the iterators is not random access (can handle +), write instead:

```
        Sum(v.begin(), ++ ++ ++ ++ ++v.begin());
```

Iterators

- We have here used pointers as iterators, all containers has iterators:

```
std::vector<float> v(5);
    int i = 1;
    for (auto it = v.begin(); it != v.end(); ++it,
++i)
        *it = 1.0f / i;
    y = Sum<std::vector<float>, float>(v);
    z = SumIT<std::vector<float>::iterator,
float>(v.begin(), v.end()); //STL stuk
```

- Types for iterators tend to be long, next lecture we take up how to handle it without writing the types explicit

Iterators as objects

- In many cases a simple pointer is not enough, we here show how to make the iterator to a real object.

```
class FloatContIt{
    float* ptr;
public:
    FloatContIt(float *p) :ptr(p) {}
    bool operator==(const FloatContIt& rhs) { return this->ptr ==
rhs.ptr; }
    bool operator!=(const FloatContIt& rhs) { return this->ptr !=
rhs.ptr; }
    float& operator*() { return *ptr; }
    //    float operator->() { return *ptr; }
    FloatContIt& operator++() {
        ++ptr;
        return *this;
    }
};
```

- Computing the sum

```
float z = SumIT<FloatContIt, float>(fc.begin(), fc.end()); //STL stuk
```

Allocation without construction

- When we allocate more space for a std::vector we do not want to construct to new elements:
    - Inefficient, they will probably get a new value when we increase the size of the vector (push_back)
    - Missing default constructor for T

- Solution: Allocate raw memory without construction.

Raw allocation and placement new

```
#pragma once
#include <cstdlib>
#include <new>

struct C {
    float value;
};

class RawCont {
    size_t _capacity, _size;
    C * _data;
public:
    RawCont(size_t cap) : _capacity(cap), _size() {
        _data = static_cast<C*>(malloc(_capacity*sizeof(C)));
        if (_data == nullptr)
            throw std::bad_alloc();
    }
    void push_back(const C& v) {
        new(_data + _size) C(v);
        ++_size;
    }
    ~RawCont() {
        for (size_t i = 0; i < _size; ++i)
            _data[i].~C();
        free _data;
    }

};
```

std::allocator
is the modern version of malloc

```
#include <new>
#include <memory>

template < class Allocator = std::allocator<C>>
class RawContAllocator {
    size_t _capacity, _size;
    C * _data;
    Allocator a = Allocator();
public:
    RawContAllocator(size_t cap) : _capacity(cap), _size() {
        _data = a.allocate(_capacity);
        if (_data == nullptr)
            throw std::bad_alloc();
    }
    void push_back(const C& v) {
        new(_data + _size) C(v);
        ++_size;
    }
    ~RawContAllocator() {
        for (size_t i = 0; i < _size; ++i)
            _data[i].~C();
        a.deallocate(_data,_capacity);
    }
};
```

Typedefs in containers (some of)

Name	type	notes
value_type	T	Eraseable
reference	T&	
const_reference	const T&	
iterator	iterator pointing to T	ForwardIterator
const_iterator	const iterator pointing to T	ForwardIterator
difference_type	signed integer	must be the same as iterator_traits::difference_type for iterator and const_iterator
size_type	unsigned integer	large enough to represent all positive values of difference_type

Några av operationerna

expression	return type	semantics	conditions	complexity
C()	C	Creates an empty container	C().empty() == true	Constant
C(a)	C	Create a copy of a	T must be CopyInsertable Post: a == C(a)	Linear
a = b	C&	All elements of a are destroyed or move assigned to elements of b	Post: a == b	Linear
(&a)->~C()	Void	Destroy all elements and free all memory		Linear
a.begin()	(const_)iterator	Iterator to the first element		Constant
a.end()	(const_)iterator	Iterator to one past the last element		Constant
a.cbegin()	const_iterator	const_cast<const C&>(a).begin()		Constant
a.cend()	const_iterator	const_cast<const C&>(a).end()		Constant
a.size()	size_type	distance(a.begin(),a.end())		Constant
a.max_size()	size_type	b.size() where b is the largest possible container		Constant
a.empty()	convertible to bool	a.begin() == a.end()		Constant

Några fler av operationerna

expression	return type	semantics	conditions	complexity
a == b	convertible to bool		T must be [EqualityComparable]()	linear
a != b	convertible to bool	!(a==b)		Linear
a.swap(b)	void	exchanges the values of a and b		Constant
swap(a,b)	void	a.swap(b)		Constant