

KIV/CPP – Programming in C++

02. STL Containers, RAII, Smart Pointers

Martin Úbl

KIV ZČU

2025/2026

- *iterator*
- design pattern
- tightly bound to a container
- “points” to exactly one element
- each container provides an iterator type, for example

```
std::vector<int>::iterator
```

- optionally a const variant, for example

```
std::vector<int>::const_iterator
```

- sometimes a reverse variant is available, for example

```
std::vector<int>::reverse_iterator
```

- *side note*
- meaning of the syntax

```
std::vector<int>::iterator
```

- there is a class named `iterator` inside the definition of a templated class `vector` (here, typed `int`) inside a namespace called `std`
 - therefore, if there is `iterator` class inside other class, it is a different class

- before C++20 - *named requirements*, now with the *Legacy* prefix
- *Iterator*
 - general principle; dereference, increment
- *InputIterator*, *OutputIterator*
 - iterator for reading / writing
- *ForwardIterator*
 - forward iterator, multipass
- *BidirectionalIterator*
 - bidirectional iterator; additionally decrement
- *RandomAccessIterator*
 - random access; addition, subtraction
- and more...

- after C++20 - *concepts*, iterator library
- `std::input_iterator`, `std::output_iterator`
 - iterator for reading / writing
- `std::forward_iterator`
 - forward iterator, multipass
- `std::bidirectional_iterator`
 - bidirectional iterator; additionally decrement
- `std::random_access_iterator`
 - random access; addition, subtraction
- and more...
- the only named requirement that remains is *ConstexprIterator*

- concept example
- `std::random_access_iterator`

```
template<class I>
concept random_access_iterator =
    std::bidirectional_iterator<I> &&
    std::totally_ordered<I> &&
    std::sized_sentinel_for<I, I> &&
    requires(I i, const I j, const std::iter_difference_t<I> n) {
        { i += n } -> std::same_as<I>;
        { j + n } -> std::same_as<I>;
        { n + j } -> std::same_as<I>;
        { i -= n } -> std::same_as<I>;
        { j - n } -> std::same_as<I>;
        { j[n] } -> std::same_as<std::iter_reference_t<I>>;
    };
```

- we will cover concepts in later lectures
- in short, this is a “listing” of all requirements in a form the compiler understands

- after a change to the underlying container, it may no longer be valid!
 - there is no back reference from container to the iterator
 - <https://en.cppreference.com/w/cpp/container>
- the reason why, for example, deletion (erase) returns a new iterator

```
itr = myList.erase(itr);
```

- obtaining an iterator – begin, rbegin, find, ...

```
auto itr = myList.begin();
```

- typically, for example, traversing a container from the beginning (`begin()`) to the end (`end()`)

```
for (auto itr = myList.begin();  
     itr != myList.end();  
     ++itr) ...
```

- or backward from `rbegin()` to `rend()` (if the container supports it)
- `end()` (`rend()`) is a special iterator
 - marks the end of the container
 - `find()` returns it when an element is not found

- *RandomAccessIterator*, `std::random_access_iterator`
 - `std::array`, `std::vector`

```
// 5th element
```

```
auto itr = myVector.begin() + 4;  
std::cout << *itr;
```

- *ForwardIterator*, `std::forward_iterator`
 - `std::list`

```
// 5th element
```

```
auto itr = myList.begin();  
std::advance(itr, 4);  
std::cout << *itr;
```

Initializer list

- `std::initializer_list`
- in certain contexts it can be written without a type, using `{` and `}`
 - class constructors
 - parameter initialization
 - initialization of container “internals”
- prevents so-called *narrowing* type conversions (a subset of implicit conversions)
 - for example double to float
 - integer types to floating-point numbers and vice versa
- safer
- https://en.cppreference.com/w/cpp/language/list_initialization

```
std::array<int, 3> arr{1,2,3};      // direct
std::array<int, 3> arr = {1,2,3};  // copy
Player plr{"Adam"};
Player* pplr = new Player{"Adam"};
```

```
#include <array>
std::array<int, 3> arr = {1, 2, 3};
```

- generic type (templates), representation of a fixed-length array

insertion	access	deletion	search
NO	arr[i], $O(1)$	NO	NO ¹
	iterator, $O(1)$		

Table: Operations on std::array

¹generic only

```
#include <vector>
std::vector<int> vec = {1, 2, 3};
```

- generic type (templates), representation of a variable-length array

insertion	access	deletion	search
push_back, $O(1)$	vec[i], $O(1)$	erase, $O(n)$	NO^2
insert, $O(n)$	iterator, $O(1)$		

Table: Operations on std::vector

²generic only

```
#include <list>
std::list<int> lst = {1, 2, 3};
```

- generic type (templates), representation of a linked list (bidirectional)
- std::forward_list variant – singly linked

insertion	access	deletion	search
push_back, $O(1)$	iterator, $O(n)$	erase, $O(1)$	NO^3
insert, $O(1)$			

Table: Operations on std::list

³generic only

```
#include <map>
std::map<int, double> mp
    = {{2, 2.5},{3, 5.5},{5, 12.95}};
```

- generic type (templates), representation of associative storage
- internally often, for example, a *red-black* tree

insertion	access	deletion	search
<code>mp[i], $O(\log n)$</code>	<code>mp[i], $O(\log n)$</code>	<code>erase, $O(\log n)$</code>	<code>find, $O(\log n)$</code>
<code>insert, $O(\log n)$</code>	<code>iterator, $O(\log n)$</code>		

Table: Operations on std::map

```
#include <set>
std::set<int> st = {2, 3, 5};
```

- generic type (templates), representation of a set with unique elements
- internally often, for example, a *red-black* tree

insertion	access	deletion	search
insert, $O(\log n)$	iterator, $O(\log n)$	erase, $O(\log n)$	find, $O(\log n)$

Table: Operations on std::set

- std::string
- specific type std::basic_string<T> for the char type
- variants for wchar_t (std::wstring), char16_t (std::u16string), ...
- internally a vector of characters
- also a container (has an iterator, begin() and end(), ...)

```
#include <string>
```

```
std::string str1("String_1");  
std::string str2{"String_2"};  
std::string str3 = "String_3";  
std::string str4 = {"String_3"};
```

- overloaded == operator (see later seminars) for comparison also with, for example, const char*

```
const char* cstr = "hello";
std::string str("hello");

if (cstr == "hello") {
    // not guaranteed to work
    // sometimes works... why?
}

if (str == "hello") {
    // this is guaranteed to work
}
```

- map initialization and insertion

```
std::map<int, double> mp{{2, 5.1}};  
mp.insert({1, 2.5});
```

- nested initialization (for example, a vector of strings)

```
std::vector<std::string>  
    vec1{"Hello", "World"};  
  
std::vector<std::string>  
    vec2{{"Hello"}, {"World"}};
```

Range-based for

- special for loop syntax for containers with iterators
- often combined with auto and a reference
- sometimes const is desirable
- analogous to foreach from other languages
- iteration from begin() to end() with dereferencing

```
std::list<int> myList = {2, 4, 8, 16, 32};
```

```
for (const auto& element : myList)  
    std::cout << element << std::endl;
```

Range-based for

- for associative storage, always either
 - reference with a const key
 - const reference
 - copy
- the key determines the physical position in memory

```
std::set<int> set = {1, 5, 10};
```

```
for (auto& element : set) {  
    std::cout << element << std::endl;  
    element = 5; // not allowed!  
}
```

- note: auto automatically deduces a type with const

Range-based for

- since C++20 an initialization expression can be added

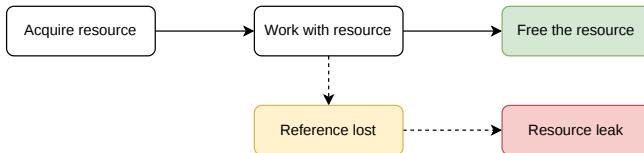
```
for (int i = 5; auto& element : set) {  
    //  
}
```

- more useful for iterating through the content of a temporary object

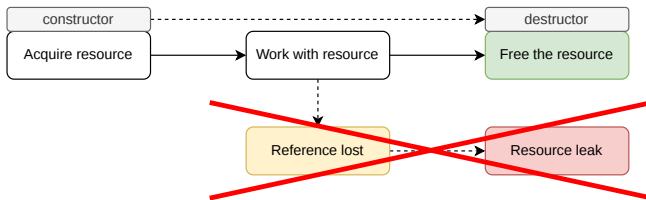
```
for (Obj a = getObj();  
     auto& p : a.items()) {  
    //  
}
```

- In the lab:
 - containers
 - iterators
 - strings

- *Resource Acquisition Is Initialization*
- *resource* = acquired and released entity
 - memory
 - file
 - lock (mutex, ...)
 - thread
 - and more...



- *Resource Acquisition Is Initialization*
- use of language properties
- deterministic object lifetime bounded by a constructor and destructor



- typical applications:
 - working with files (`std::fstream`, ...)
 - smart pointers (`std::unique_ptr`, ...)
 - locks (`std::unique_lock`, ...)
 - STL containers

- *smart pointers*
- several variants
 - `std::unique_ptr`
 - `std::shared_ptr`
 - `std::weak_ptr`
- initializers
 - `std::make_unique`
 - `std::make_shared`

- std::unique_ptr
- RAII structure
- exactly one owner

```
// rectangle
```

```
std::unique_ptr<Rect> rect  
    = std::make_unique<Rect>(2, 3);
```

```
rect->CalcSurface();
```

- cannot be copied

```
auto rect = std::make_unique<Rect>(a, a);  
auto rect2 = rect; // error!
```

- passing using std::move

```
auto rect = std::make_unique<Rect>(2, 3);  
// ...  
auto rect2 = std::move(rect);  
rect2->CalcSurface();  
rect->CalcSurface(); // error!
```

- passing using std::move
- function return value

```
std::unique_ptr<Rect> CreateSquare(int a) {  
    auto rect = std::make_unique<Rect>(a, a);  
    // ...  
    return std::move(rect);  
}
```

- may not be necessary; C++ allows applying so-called *copy elision* when returning local objects (NRVO)

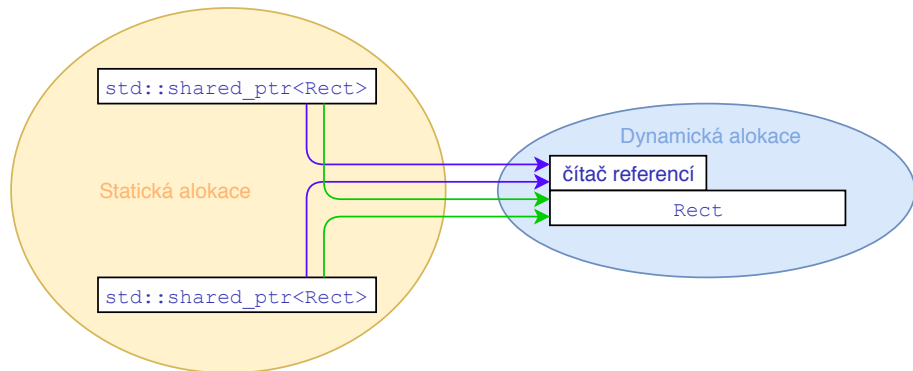
- std::shared_ptr
- RAI structure
- at least one owner
- internal reference counter
 - constructor increments it
 - destructor decrements it

```
// rectangle
std::shared_ptr<Rect> rect
    = std::make_shared<Rect>(2, 3);

rect->CalcSurface();
```

- can be copied

```
auto rect = std::make_shared<Rect>(2, 3);  
auto rect2 = rect;
```



- return value is not a problem

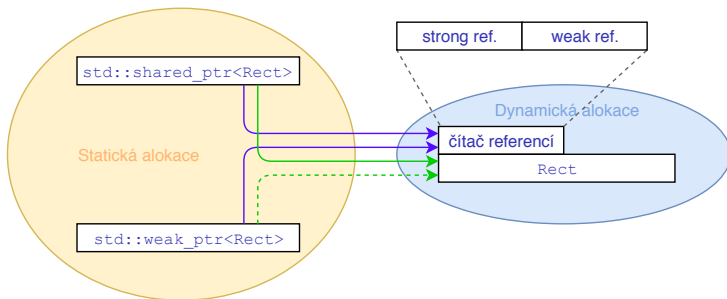
```
std::shared_ptr<Rect> CreateSquare(int a) {  
    auto rect = std::make_shared<Rect>(a, a);  
    // ...  
    return rect;  
}
```

- both can be invalidated
 - by assigning `nullptr`
 - using the `reset()` method

```
ptr = nullptr;  
ptr.reset();  
ptr.reset(nullptr); // unique_ptr only
```

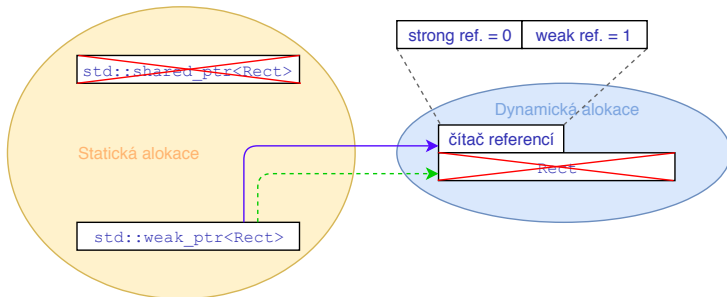
- std::weak_ptr
- weak reference, always bound to std::shared_ptr

```
auto rect = std::make_shared<Rect>(2, 3);  
std::weak_ptr<Rect> rect_weak = rect;
```



- never work with it directly; always obtain a std::shared_ptr instance

```
std::weak_ptr<int> rect_weak = rect;  
if (auto shared = rect_weak.lock()) {  
    // ...  
}
```



- useful, for example, for working with circular lists

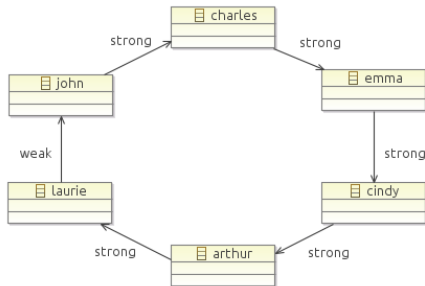


Figure: Cyclic dependency; borrowed image⁴

⁴<https://visualstudiomagazine.com/articles/2012/10/19/circular-references.aspx>