Modernes C++

...für Programmierer

Unit 05: `array`, `vector`,...,Smart-Pointer

by

Dr. Günter Kolousek

12. September 2019

Überblick

- `array`
- `vector`
- `set`
- `map`
- `tuple`, `pair`
- Smart-Pointer
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`

`array`

- nicht sinnvoll rohe Arrays zu verwenden
- Klasse `array`
  - fast kein Overhead
  - Größe bekannt
  - sicherer Zugriff
  - kopieren und vergleichen möglich

array-2

```cpp
#include <iostream>  // array.cpp
#include <array>
using namespace std;
int main() {
    array<int, 5> arr;
    cout << "size: " << arr.size() << endl;
    for (auto i : arr) {
        cout << i << " ";
    }
}
```

```
size: 5
0 0 0 0 -1625771376
```

array - 3

```cpp
#include <iostream>  // array2.cpp
#include <array>
using namespace std;
int main() {
    array<int, 5> arr{1, 2, 3};
    cout << "size: " << arr.size() << endl;
    for (auto i : arr) {
        cout << i << " ";
    }
}
```

```
size: 5
1 2 3 0 0
```

array-4

```cpp
#include <iostream>  // array3.cpp
#include <array>
using namespace std;
int main() {
    array<int, 5> arr{1, 2, 3};
    // abort or value arbitrary!
    cout << arr[10] << endl;
    cout << arr.at(10) << endl;   // exception!
}
```

```
-1219166208
terminate called after throwing an instance of 'std
  what():  array::at: __n (which is 10) >= _Nm (whi
fish: Job 1, 'go' durch Signal SIGABRT (Abbruch) be
```

array-5

```cpp
#include <iostream>  // array4.cpp
#include <array>
using namespace std;
int main() {
    array<int, 5> arr{1, 2, 3};
    cout << arr[10] << endl;
    try {
        cout << arr.at(10) << endl;
    } catch (const out_of_range& ex) {
        cout<<"out of range: "<<ex.what()<<endl;
    }
}
```

```
0
out of range: array::at: __n (which is 10) >= _Nm (
```

array - 6

```cpp
#include <iostream>  // array5.cpp
#include <array>
using namespace std;
int main() {
    array<int, 5> arr{1, 2, 3};
    array<int, 5> arr2;

    arr2 = arr;
    if (arr == arr2)
        cout << "equal" << endl;
    else
        cout << "not equal" << endl;
}
```

```
equal
```

`vector`

- Sequenz eines Typs
- Größe variabel
  - aktuelle Größe
  - Kapazität: Größe des reservierten Speicherbereiches
- im Zweifelsfall: `vector` verwenden

vector - 2

```cpp
#include <iostream>  // vector.cpp
#include <vector>
using namespace std;
int main() {  vector<int> v{1, 2};
    cout << "size: " << v.size() << " cap: "
         << v.capacity() << endl;
    v.push_back(3); cout << "added 3 → size: "
      << v.size() << " cap: "<<v.capacity()<<endl;
    v.push_back(4); v.push_back(5);
    cout << "added 3, 4 → size: " << v.size()
         << " cap: " << v.capacity() << endl;
}
```

```
size: 2 cap: 2
added 3 → size: 3 cap: 4
added 3, 4 → size: 5 cap: 8
```

vector - 3

```cpp
#include <iostream>  // estd.h
#include <algorithm>
namespace Estd {
    using namespace std;
    template <typename T>
    void print(T& seq) {
        for (const auto& s : seq) {
            cout << s << ' '; }
        cout << endl;
    }
    template <typename T>
    void sort(T& seq) {
        std::sort(begin(seq), end(seq));
    }
}
```

vector - 4

```
#include ”estd.h”
#include <vector>
using namespace Estd;  // vector2.cpp
int main() {  vector<string> v1{”apple”, ”orange”};
    vector<string> v2{”plum”, ”apricot”};
    v1.insert(v1.begin()+1,”banana”);//before pos 1
    v1.insert(v1.begin()+2, v2.begin(), v2.end());
    print(v1);
    cout << v1.size() << ' ' <<v1.capacity()<<endl;
    v1.shrink_to_fit();
    cout << v1.size() << ' ' <<v1.capacity()<<endl;
}
```

```
apple banana plum apricot orange
5 6
5 5
```

vector - 5

```
#include ”estd.h”
#include <vector>
using namespace Estd;  // vector3.cpp
int main() {
    vector<string> v1{”apple”,”banana”,”plum”,
        ”apricot”,”orange”};
    sort(begin(v1), end(v1));   print(v1);
    cout << v1.front() << ' ' << v1.back() << endl;
    v1.erase(begin(v1));  v1.pop_back();
    cout << v1.front() << ' ' << v1.back() << endl;
    v1.clear();  cout << v1.empty() << endl;
}
```

```
apple apricot banana orange plum
apple plum
apricot orange
1
```

set

```
#include ”estd.h”
#include <set>
using namespace Estd;  // set.cpp
int main() {
    set<int> s1{2, 1, 2, 1, 3, 4};
    print(s1);
    auto search = s1.find(2);  // iterator...
    if (search != s1.end())
        cout << ”Found ” << (*search) << endl;
    else
        cout << ”Not found”  << endl;
    // insert, erase, clear, empty, size, begin&end
}
```

```
1 2 3 4
Found 2
```

map

```
template<typename M>
void print_map(M& m) {
    cout << '{'; auto i=begin(m);
    for (auto j=--end(m); i != j; i++)
         cout << i->first << ':'
              << i->second << ',';
    cout << i->first << ':' << i->second << '}'
         << endl;
}
```

map - 2

```
#include ”estd.h”
#include <map>
using namespace Estd;  // map.cpp
#include ”print_map.h”
int main() {
    map<string, int> pb{{”maxi”,123},{”mini”,999}};
    pb[”otto”]=475; pb[”maxi”]=112; print_map(pb);
    try {
        cout << pb.at(”xxx”) << endl;
    } catch (...) { cout << ”not found! ”; }
    cout << pb[”xxx”] << ' ';
    cout << pb.at(”xxx”) << endl;  // found!
}
```

```
{maxi:112,mini:999,otto:475}
not found! 0 0
```

pair

```cpp
#include <iostream>  // pair.cpp
#include <utility>
using namespace std;
using namespace std::literals;  // can be ommitted (mayb
int main() {
    auto key{make_pair(1234, ”Maxi Muster”s)};
    get<0>(key) = 4711;  // look it will be modified
    cout << '(' << key.first << ”, ”
                << key.second << ')' << endl;
    // error if multiple identical types:
    cout << '(' << get<0>(key) << ”, ”
                << get<string>(key) << ')';
    int id;
    string name;
    tie(id, name) = key;
    cout << endl << id << ”, ” << name;
}
```

tuple

```cpp
#include <iostream>  // tuple.cpp
#include <tuple>
using namespace std;
using namespace std::literals;  // can be ommitted (mayb
int main() {
  auto key{make_tuple(1234,”Maxi Muster”s,'A')};
  get<0>(key) = 4711;  // look it will be modified
  // error if multiple identical types:
  cout << '(' << get<int>(key) << ”, ”
       << get<string>(key) << ”, ”
       << get<char>(key) << ')';
  int id;   string name;  char type;
  tie(id, name, type) = key; // tuple unpacking...
}
```

```
(4711, Maxi Muster, A)
```

Smart-Pointer

- simuliert *raw pointer* (”roher Zeiger”)
- zusätzlich ”Garbage Collection”
- in C++ mittels *reference counting*

Smart-Pointer – 2

- `unique_ptr`
  - übernimmt Verantwortung
    - d.h. ”besitzt” Speicherobjekt
    - Löschung des Speicherobjektes wenn `unique_ptr` gelöscht wird
  - kann nicht kopiert werden
  - Einsatz: wenn keine mehrfachen Verweise auf ein Speicherobjekt
- `shared_ptr`
  - teilt sich Verantwortung
    - d.h. ”besitzt” Speicherobjekt nur zum Teil/gar nicht
    - Löschung des Speicherobjektes wenn *letzter* `shared_ptr` gelöscht wird
  - kann kopiert werden
  - Einsatz: wenn mehrfache Verweise auf ein Objekt

Smart-Pointer – 3

- `weak_ptr`
  - übernimmt keine Verantwortung
    - d.h. ”besitzt” Speicherobjekt überhaupt nicht
    - keine Löschung
    - Zugriff nur über `shared_ptr` mittels `lock()`
  - kann kopiert werden
  - Einsatz: zum Aufbrechen von Zyklen

unique_ptr

```
#include <iostream>  // uniqueptr.cpp
#include <memory>
using namespace std;
void use_ptr(int* pi) {
    cout << *pi << endl;
    // delete here?
}
int main() {
    {
        int* pi{new int{1}};
        use_ptr(pi);
        // delete here?
    }
    // no delete → memory leak!
}
```

```
1
```

unique_ptr - 2

```cpp
#include <iostream>  // uniqueptr2.cpp
#include <memory>
using namespace std;
int main() {
    {
        unique_ptr<int> upi{new int{1}};
        cout << *upi << endl;  // like a raw ptr
    }
    // delete done!
}
```

```
1
```

unique_ptr - 3

```cpp
#include <iostream>  // uniqueptr3.cpp
#include <memory>
using namespace std;
// void use_ptr(unique_ptr<int> upi) {  // error
void use_ptr(unique_ptr<int>& upi) {
    cout << *upi << endl;  // want to free here?
}
int main() {
    {
        unique_ptr<int> upi{new int{1}};
        use_ptr(upi);
    }  // deleted → no memory leak
}
```

```
1
```

unique_ptr - 4

```cpp
#include <iostream>  // uniqueptr4.cpp
#include <memory>
using namespace std;
void use_ptr(unique_ptr<int> upi) {
    cout << *upi << ' ';
    // deleted→no memory leak
}
int main() {
    unique_ptr<int> upi{new int{1}};
    use_ptr(move(upi));
    cout << ((upi.get()==nullptr) ? 0:*upi)<< endl;
}
```

```
1 0
```

unique_ptr - 5

```cpp
#include <iostream>  // uniqueptr5.cpp
#include <memory>
using namespace std;
class Game {
    // assumptions: constructor
    //    - gets one argument (see below)
    //    - may throw exception!
};
void use_ptr(unique_ptr<Game> a,
             unique_ptr<Game> b) {};

int main() {
    // memory leak possible (until C++14)
    use_ptr(unique_ptr<Game>{new Game{1}},
            unique_ptr<Game>{new Game{2}});
}
```

unique_ptr – 6

- Auswertung der Ausdrücke in beliebiger Reihenfolge!
- Daher folgende Auswertung möglich:
  1. Speicher für erstes Game anfordern
  2. Konstruktor für erstes Game ausführen
  3. Speicher für zweites Game anfordern
  4. Konstruktor für zweites Game ausführen
  5. `unique_ptr<Game>` für erstes Game anlegen
  6. `unique_ptr<Game>` für zweites Game anlegen
  7. `use_ptr` aufrufen
- Problem
  - Speicher für zweites Game nicht vorhanden
  - Konstruktor für zweites Game wirft Exception

unique_ptr – 7

- Zwei Lösungen:
  - Zerlegung in mehrere Anweisungen

```
unique_ptr<Game> game1{1};
unique_ptr<Game> game2{2};
use_ptr(move(game1), move(game2));
```

  - Ab C++14 → (fast) **immer** verwenden!

```
use_ptr(make_unique<Game>(1),
        make_unique<Game>(2));
```

- Ab C++17
  - Auswertungsreihenfolge noch immer nicht spezifiziert, aber
    - jeder Parameter ist voll ausgewertet bevor ein anderer Parameter wird ausgewertet
  - `f(x).g(y).h(z)` … jetzt: x vor y vor z
  - operator overlading: Auswertung gemäß der Reihenfolge des eingebauten Operators: `cout << f() << g() << h();` … jetzt: `f()` vor `g()` vor `h()`

shared_ptr

```cpp
#include <iostream>  // sharedptr.cpp
#include <memory>
using namespace std;

int main() {
    shared_ptr<int> spi{new int{1}};
    cout << spi.use_count() << ' ';
    {
        shared_ptr<int> spi2{spi};
        cout << spi2.use_count() << ' ';
    }
    cout << spi.use_count() << endl;
}
```

```
1 2 1
```

shared_ptr - 2

```
#include <iostream>  // sharedptr2.cpp
#include <memory>
using namespace std;
struct Person {
    shared_ptr<Person> spouse;
    ~Person() { cout << ”destructed!” << ' '; }
};
int main() { // guideline: use make_shared !!!
  shared_ptr<Person> p1{make_shared<Person>()};
  shared_ptr<Person> p2{make_shared<Person>()};
  cout<<p1.use_count()<<' '<<p2.use_count()<<endl;
}
```

```
1 1
destructed! destructed!
```

shared_ptr - 3

```
#include <iostream>  // sharedptr3.cpp
#include <memory>
using namespace std;
struct Person {
    shared_ptr<Person> spouse;
    ~Person() { cout << ”destructed!” << ' '; }
};
int main() {
  shared_ptr<Person> p1{make_shared<Person>()};
  shared_ptr<Person> p2{make_shared<Person>()};
  p1->spouse = p2;
  p2->spouse = p1;
  cout<<p1.use_count()<<' '<<p2.use_count()<<endl;
}

2 2
```

shared_ptr - 4

```
#include <iostream>  // sharedptr4.cpp
#include <memory>
using namespace std;
struct Person {
    weak_ptr<Person> spouse;
    ~Person() { cout << ”destructed!” << ' '; }
};
int main() {
  shared_ptr<Person> p1{make_shared<Person>()};
  shared_ptr<Person> p2{make_shared<Person>()};
  p1->spouse = p2;   p2->spouse = p1;
  cout<<p1.use_count()<<' '<<p2.use_count()<<endl;
}
```

```
1 1
destructed! destructed!
```

weak_ptr

```cpp
#include <memory>     // weakptr1.cpp
#include <iostream>
using namespace std;
int main() {
    weak_ptr<int> wpi;
    {
        auto spi{make_shared<int>(1)};
        wpi = spi;
        cout << *wpi.lock() << ' ';  // shared_ptr!
    }
    // cout << *wpi << endl;  // error!
    // * undefined on nullptr:
    // cout << *wpi.lock() << endl;  // segfault!
    cout << wpi.lock().get() << endl;  // nullptr!
}
```

```
1 0
```

weak_ptr - 2

```cpp
#include <memory>     // weakptr2.cpp
#include <iostream>
using namespace std;
void use_ptr(weak_ptr<int> wpi) {
    cout << *wpi.lock() << ' ';
}
int main() {
    weak_ptr<int> wpi;
    auto spi{make_shared<int>(1)};
    wpi = spi;
    use_ptr(wpi);  // can be copied
    use_ptr(move(wpi));  // but also moved
    // cout << *wpi.lock() << endl;  // segfault!
}
```

```
1 1
```

weak_ptr - 3

```cpp
#include <memory>     // weakptr3.cpp
#include <iostream>
using namespace std;

int main() {
    shared_ptr<int> spi{new int{42}};
    weak_ptr<int> wpi{spi};  // init possible
    spi.reset();  // does not own anymore
    try {  // the other way around:
        shared_ptr<int> spi2{wpi};
    } catch(const std::bad_weak_ptr& e) {
        cout << e.what() << '\n';
    }
}
```

```
bad_weak_ptr
```