Napredna uporaba C++

Tadej Petrič

8. september 2019

Zgodovina

- Kako se C++ razlikuje od C?

Zgodovina

- Kako se C++ razlikuje od C?
- C with classes že leta 1980
- Preobloženi operatorji že leta 1983
- Templates za generične tipe leta 1990

Zgodovina

- Kako se C++ razlikuje od C?
- C with classes že leta 1980
- Preobloženi operatorji že leta 1983
- Templates za generične tipe leta 1990
- To je bilo že 30 let nazaj!
- C++ je sodoben in aktiven jezik. Kaj ima torej novega?
- C++ je znan kot "bloated" jezik, torej mora imeti še veliko razlik od minimalističnega jezika C

Kazalo

Array (C++11)

V C smo pisali

```
int a[10];
```

V C++ je priporočljivo uporabljati

```
std::array<int, 10> a;
```

iz knjižnice `array`. Uporablja se skoraj enako, le da se ne pretvori v kazalec implicitno.

List

Namesto zamudnega postopka programiranja svojih kazalčnih seznamov lahko uporabimo standardnega.

```
#include <list>

int main() {
    std::list<int> seznam = {1, 2, 3};
    seznam.push_front(0);
    seznam.push_back(4);
}
```

Vector

Verjetno najbolj uporabljena struktura v C++. Array spremenljive velikosti.

```
#include <vector>

int main() {
    std::vector<int> a(10);
    a[1] = 5;
    a.push_back(3);
}
```

Prednosti in slabosti v primerjavi z `list` in `array`?

Vse

`https://en.cppreference.com/w/cpp/container`

- `array`, `vector`, `list`, `forward_list` (C++11), `deque`
- `set`, `map`, `multiset`, `multimap`
- Od C++11 še unordered različice (npr. `unordered_set`)
- `stack`, `queue`, `priority_queue`
- `span` (C++20)

Algorithm

Veliko algoritmov (čez 100), za vse poglejte
https://en.cppreference.com/w/cpp/algorithm ali
https://www.youtube.com/watch?v=2olsGf6JIkU

```cpp
#include <algorithm>
#include <functional> // for std::greater
int main() {
    std::array<int, 5> arr = {2, 8, 3, 3, 6};
    std::sort(arr.begin(), arr.end(), std::greater<int>());

    auto adj = std::adjacent_find(arr.begin(), arr.end());
    arr[std::distance(arr.begin(), adj)]; // this is 3
    // if no adjacent, returns last element
}
```

auto (C++11)

Namesto eksplicitnega podajanja tipov jih lahko prevajalnik izbere sam. Zelo koristno za težje tipe.

```
#include <utility>
auto f() -> std::pair<int, bool> {return {1, true};}
int main() {
    auto first_a = 3;
    int first_e = 3;

    auto second_a = 3ull;
    unsigned long long second_e = 3;

    auto [a, b] = f(); // C++17
}
```

Iteratorji

```
#include <vector>
#include <iterator>
int main() {
    std::vector<int> arr {1, 2, 3, 4};
    // we could've used auto here
    std::vector<int>::iterator it = arr.begin();
    std::advance(it, 2);
    while (it != arr.end()) { *it; ++it;}
}
```

range for-loop (C++11)

Pogosto želimo samo iterirati čez celotno strukturo.

```
#include <vector>
#include <iostream>
int main() {
    std::vector<int> a {1, 2, 3, 4};
    for (int x: a) { std::cout << x << " "; }
    std::cout << "\n";
    for (auto&& x: a)
        ++x;
    for (const auto& x: a)
        std::cout << x << " ";
}
```

Lambda(C++11)

Lambda funkcije so anonimne funkcije. Ponavadi so kratke.

```
int main() {
    auto a = [](){};
    auto b = [](int x){return x+1;};
    b(6); // returns 7
    int num1 = 3; int num2 = 5;
    auto c = [=](auto x) { // auto lambda C++14
        return x + num1 + num2;};
    auto d = [x = [](int y){return y-2;}] (int y) {
        return x(5) + y;
    }; // capture init C++14
    auto e = [num1]() mutable -> int {++num1; return num1;};
}
```

funkcije višjega reda

Funkcija, ki kot argument sprejme funkcijo se imenuje funkcija višjega reda.

```
template <typename Fn>
int apply_3(Fn&& f) {
    return f(3);
}
int main() {
    apply_3([](int x){ return x + 1;});
}
```

Naloga

Naredi funkcije, ki v seznamu števil preštejejo:

- Koliko elementov enakih nekemu številu
- Koliko elementov manjših od nekega števila
- Koliko elementov deljivih z nekim številom

Template programming language

C++ templati so Turing complete. To pomeni, da so dovolj močni, da izvedejo vsak algoritem. Ker templati obstajajo preden izvedemo program (torej obstajajo le za prevajalnik) to pomeni, da lahko vsak algoritem izvedemo med compile-time (ko pa zaženemo program, le uporabljamo kar je prevajalnik izračunal).
Kot primer bomo napisali program, ki izračuna fakulteto
$n! = n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1$

constexpr

Programiranje s templati je grdo.

constexpr

Programiranje s templati je grdo.
Na srečo od C++11 (ter veliko bolj enostavno v C++14, 17 in 20) to ni več potrebno. Prejšni program lahko ponovno napišemo s pomočjo `constexpr` funkcij.

variadic templates

Templati ponavadi dovolijo, da neko funkcijo pokličemo z različnimi tipi argumentov.
Od C++11 lahko s templati predstavimo tudi funkcije z različnim številom argumentov.

```
template <typename Head, typename... Tail>
Head f(Head&& x, Tail&&... y) {
    return x;
}

int main() {
    f(1,2,3,3,4,54,5,6,72); // -> 1
    f(2,1); // -> 2
}
```

Naloga

Napiši funkcijo, ki sprejme poljubno število argumentov ter vsakega izpiše (med njimi pa presledki).

fold (C++17)

Z uporabo fold expressionov lahko neko binarno operacijo izvedemo na vseh argumentih funkcije. Tako se lahko v mnogih primerih izognemo rekurzivnim klicom funkcij.

fold (C++17) cont.

```
//take any amout of arguments
template <typename... Args>
int sum(Args... args) {
    // apply binary operation + to all arguments
    // sum(1, 2, 3, 4, 5)
    // -> ((((1+2) +3) +4) +5)
    return (... + args);
}
```

Naloga: Napiši funkcijo `any`, ki sprejme poljubno število boolov in vrne ali je kateri koli od njih `true`

RAII

RAII pomeni Resource Acquisition Is Initialisation. Pomeni, da spremenljivka obstaja natanko takrat, ko obstajajo podatki, ki si jih lasti. Ko spremenljivka neha obstajati, se podatki avtomatično sprostijo.

```
int main() {
    int a = 3;
    {
        int x = 5;
    }
    // x ni več dosegljiva, za sabo pobriše podatke
}
// a ni dosegljiva, zbriše podatke
```

RAII kdaj ne dela?

```
int main() {
    int* x = new int(5);
    {
        int* a = new int(3);
    } // a memory leaks
} // x memory leaks
```

Še več težav z dinamičnim spominom

```
int read_delete(int* p) {
    // do something with p
    delete p; // clean up
}
int main() {
    int* p = new int(5);
    read_delete(p);
    delete p; // clean up
}
```

Če imamo globje funkcije je to težavo zelo težko odkriti!
Več kot 70% napak, ki jih odkrije Microsoft v Windows je zaradi teh dveh težav. Kako se izogniti temu?

RAII dela če ga uporabljamo (v C++14)

V C++11 so dodali pametne kazalce, ki sami skrbijo za brisanje in dodeljevanje spomina preko principa RAII. V C++14 so jih naredili dovolj uporabne, da `new` in `delete` v modernih programih ni več potrebno videti.

```
#include <memory>
int main() {
    std::unique_ptr<int> a = std::make_unique<int>(3);
    {
        auto p = std::make_unique<int>(5);
    } // p pobriše za sabo
} // a pobriše za sabo
```

Naloga

Z uporabo pametnih kazalcev implementiraj kazalčni seznam. Dodaj nekaj funkcij za uporabo.

- Funkcijo ki sprejme n (ali $n+1$ če je prvi element seznam) argumentov in prvih n mest inicializira s temi vrednostmi
- Funkcijo za izpis seznama
- Funkcijo za izbris nekega elementa iz seznama