Cpp

December 18, 2024

1 C++

V C-ju že znate programirati, C++ pa vam ponuja nekaj dodatnih funkcionalnosti, ki vam bodo olajšale življenje. Večinoma preko svoje standardne knjižnice in raznih sintaktičnih bližnjic. Standardna knjižnica vsebuje številne uporabne podatkovne tipe/strukture (*containters*) in algoritme. Osnovana je bila po knjižnici Standard Template Library (STL), ki se še danes pogosto uporablja kar kot sinonim za C++ standardno knjižnico.

Prav vam bo prišla dokumentacija: - cplusplus - cppreference

1.1 Branje in pisanje

V C-ju ste navajeni branja in pisanja podatkov s knjižnico `stdin.h`in funkcijami kot so `scanf`, `printf`, `gets`, ...

C++ pa v knjižnici `iostream` ponuja t.i. tokove (*stream*). Prav nam bosta prišla `cin` (character input) in `cout` (character output). Podpirata operatorje `>>` oz. `<<`, ki poleg branja/pisanja tudi vrneta isti objekt, da lahko operatorje verižimo.

[1]:
```
#include <iostream>
```

[2]:
```
std::cout << "Zivjo!\n";
```

```
Zivjo!
```

Večina funkcionalnosti, ki jih ponuja C++ v svoji standardni knjižnici, se nahaja v imenskem prostoru `std`, do objektov v njem pa dostopamo z `std::ime`. Malo pisanja si lahko prihranimo z deklaracijo uporabe imenskega prostora `std`.

[2]:
```
using namespace std;
```

[28]:
```
int a,b;
cin >> a >> b;
cout << "vsota = " << a+b << endl;
```

```
 9 10
```

```
vsota = 19
```

1.2 Nizi

C++ ponuja podatkovno strukturo `string`, ki nam olajša delo z nizi v primerjavi s C-jem, kjer smo bili obsojeni na delo s tabelami znakov.

[3]:
```
#include <string>
```

[6]:
```
string ime, priimek;
cin >> ime >> priimek;
string oseba = priimek + " " + ime;
cout << "Pozdravljen, " << oseba << "!" << endl;
cout << "Zacetnice: " << ime[0] << priimek[0] << endl;
cout << "Dolzina imena: " << ime.size() << endl;
```

```
 Tomaz Hocevar
```

```
Pozdravljen, Hocevar Tomaz!
Zacetnice: TH
Dolzina imena: 5
```

1.3 Pari

Pari vrednosti so zelo koristni, da nam ni treba za vsako malenkost ustvarjati novih struktur ali razredov. V jezikih, kot je npr. Python, pa je koncept terk (*tuple*) še bolj prisoten. Paru moramo določiti tudi tipe vsebovanih komponent. Sintaksa z "oklepaji" oz. znaki manjše/večje je podobna tisti, ki ste je navajeni iz Jave. Do obeh elementov dostopamo preko atributov `first` in `second`.

[4]:
```
#include <utility>
```

[8]:
```
pair<int,int> xy;
xy = make_pair(10,12);
cout << xy.first << " " << xy.second << endl;
```

```
10 12
```

1.3.1 Inicializacija s seznamom

Nekatere podatkovne tipe lahko inicializiramo tudi s seznami vrednosti (initializer list). Poleg parov bomo videli primere uporabe tudi kasneje pri drugih strukturah.

[9]:
```
pair<int,int> p = {2,3};
cout << p.first << " " << p.second << endl;
```

```
2 3
```

1.3.2 Avtomatska določitev tipa

Ko začnemo uporabljati gnezdene strukture (npr. par osebe in ocene, pri čemer je oseba prav tako par sestavljen iz imena in priimka), postanejo opisi podatkovnih tipov precej dolgi. Ker zna prevajalnik med prevajanjem ugotoviti, da se nek tip ne ujema, ga lahko tudi kar določi, kar označimo s tipom `auto`.

[10]:
```
pair<pair<string,string>, int> ocena = {{"Tomaz", "Hocevar"}, 10};
auto o = ocena;
cout << o.first.first << endl;
```

```
Tomaz
```

1.4 Seznam

V C-ju smo imeli na voljo tabele fiksne velikosti. Če smo želeli hraniti seznam elementov, ki smo ga podaljševali, pa smo naleteli na manjši problem. Tega nam rešuje poatkovnih tip `vector`, ki predstavlja razširljivo tabelo (*resizable array*), podobno kot ArrayList v Javi. V njem lahko shranjujemo samo elemente enakega tipa, ki ga moramo navesti ob deklaraciji (za razliko od npr. Pythona).

[4]:
```
#include <vector>
```

[4]:
```
vector<int> v;
for (int x=1; x<=1024; x*=2) v.push_back(x);
for (int i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
```

```
1 2 4 8 16 32 64 128 256 512 1024
```

1.4.1 Iteratorji

Koncept iteratorjev vam je že poznan iz Jave. V C++ so iteratorji kazalci na elemente v strukturah, s katerimi se lahko premikamo po elementih v tej strukturi. Vsaka struktura ima svoj tip iteratorja (npr. ‘`vector<int>::iterator`) `in ponuja iteratorja na svoj začetek in konec` (`.begin()`in`.end()`‘).

[6]:
```
for (vector<int>::iterator it=v.begin(); it!=v.end(); it++) {
    cout << *it << " ";
}
cout << endl;
```

```
1 2 4 8 16 32 64 128 256 512 1024
```

1.4.2 For each

Iteracija čez vse elemente strukture je zelo pogosta operacija, zato je v številnih programskih jezikih na voljo tudi temu prilagojena sintaksa. V C++ je to `for (tip element : struktura)`.

[17]:
```
vector<pair<int,int>> koordinate = {{2,6},{1,4},{-2,6}};
for (pair<int,int> xy : koordinate) cout << "[" << xy.first << ", " << xy.
 ↪second << "]" << endl;
```

```
[2, 6]
[1, 4]
[-2, 6]
```

1.4.3 Razpakiranje

Dostop do posameznih elementov kompleksnejšega tipa (npr. seznama parov) je lahko kar dolg. To si lahko skrajšamo z uporabo razpakiranja (*structured binding declaration*). Sintaksa je `auto [a, b, ...] = x`.

[18]:
```
for (auto [x, y] : koordinate) cout << "[" << x << ", " << y << "]" << endl;
```

```
[2, 6]
[1, 4]
[-2, 6]
```

1.5 Vrsta, sklad, slovar, množica, povezani seznam, ...

C++ pozna še cel kup drugih podatkovnih tipov, kot so `queue`, `stack`, `map`, `set`, `list` ... Več o njih pa takrat, ko bomo obravnavali abstraktne podatkovne tipe.

[6]:
```
#include <map>
#include <set>
```

[7]:
```
map<string,int> vpisna = {{"Ana", 123}, {"Miha", 456}, {"Tine", 789}};
set<string> prisotni = {"Ana", "Tine"};
for (auto ime : prisotni) cout << vpisna[ime] << endl;
```

```
123
789
```

1.6 Reference

Referenca je drugo ime za isto spremenljivko. Označimo jo s znakom `&`. V spodnjem primeru je `y` referenca na spremenljivko tipa `int`, kar zapišemo kot `int &y`. Referenca ne more biti prazna, inicializirati jo moramo z drugo spremenljivko ob deklaraciji. Deklaracija reference brez inicializacije (`int &y;`) ali inicializacija s konstantno vrednostjo (`int &y = 9;`) nista možni.

[5]:
```
int x = 10;
int &y = x;
cout << x << " " << y << endl;
```

```
10 10
```

Če sedaj spremenimo vrednost spremenljivke `x`, se ta sprememba odraža tudi v spremenljivki `y`, in obratno.

[6]:
```
x = 11;
cout << x << " " << y << endl;
y = 12;
cout << x << " " << y << endl;
```

```
11 11
12 12
```

V C++ se argumenti funkcij *prenašajo po vrednosti*. Funkcija torej prejme kopijo podanega argumenta. V Pythonu ali Javi bi z dodajanjem elementov seznamu, ki ga sprejme funkcija, spremenili tudi zunanji seznam. V C++ temu ni tako.

[7]:
```
void izpisi(vector<int> v) {
    for (int x : v) cout << x << " ";
    cout << endl;
}
```

[8]:
```
auto podvoji(vector<int> v) {
    int n=v.size();
    for (int i=0; i<n; i++) {
        v.push_back(v[i]);
    }
    return v;
}
```

[9]:
```
vector<int> a = {1,2,3,4,5};
vector<int> b = podvoji(a);
izpisi(a);
izpisi(b);
```

```
1 2 3 4 5
1 2 3 4 5 1 2 3 4 5
```

Če želimo, lahko to funkcionalnost dosežemo s *prenosom argumentov po referenci*. Funkcija `podvoji_ref` sprejme argument po referenci, vrednost te spremenljivke spremeni in ne vrne ničesar.

[10]:
```
void podvoji_ref(vector<int> &v) {
    int n=v.size();
    for (int i=0; i<n; i++) {
        v.push_back(v[i]);
    }
}
```

[11]:
```
podvoji_ref(a);
izpisi(a);
```

```
1 2 3 4 5 1 2 3 4 5
```

Prenos po referenci se uporablja za podajanje velikih spremenljivk, da se izognemo ustvarjanju kopije. Druga pogosta uporaba je vračanje več vrednosti iz funkcije preko nastavljanja argumentov, ki so podani po referenci. Slednje smo v C-ju lahko dosegli tako, da smo funkciji podali kazalec na spremenljivko in jo nato spreminjali preko tega kazalca.

V spodnjem primeru funkcija `stat` sprejme seznam celih števil `v` po referenci, da se ne ustvari kopija po nepotrebnem, ker funkcija seznama ne spreminja. Prešteje pozitivna in negativna števila ter rezultate vpiše v argumenta `poz` in `neg`, ki sta prav tako podana po referenci.

[12]:
```
void stat(vector<int> &v, int &poz, int &neg) {
    poz=0;
    neg=0;
    for (int x : v) {
        if (x>0) poz++;
        if (x<0) neg++;
    }
}
```

[13]:
```
vector<int> st = {5, -7, 8, 9, -3, -2, -1, -4};
int poz, neg;
stat(st, poz, neg);
cout << "poz/neg: " << poz << " " << neg << endl;
```

```
poz/neg: 3 5
```

1.7 Algoritmi

Knjižnica `algorithm` ponuje cel kup uporabnih funkcij, kot so `min`, `max`, `min_element`, `swap`, `count`, ...

[13]:
```
#include <algorithm>
```

[14]:
```
cout << min(5,2) << endl;
cout << min({5,3,9}) << endl;

vector<int> v={4,7,1,8};
cout << *min_element(v.begin(), v.end()) << endl;
```

```
2
3
1
```

Verjetno najpogosteje uporabljena funkcija pa je `sort`. Funkcija `sort` sprejme iteratorja na začetek in konec seznama vrednosti, ki jih bo uredila. Tako lahko uporabljamo funkcijo `sort` na različnih strukturah, ki implementirajo pravo vrsto iteratorjev.

[19]:
```
vector<int> sez = {8,41,11,7,2};
sort(sez.begin(), sez.end());
for (int x : sez) cout << x << " ";
cout << endl;
```

```
2 7 8 11 41
```

1.7.1 Anonimne funkcije

Pogosto pišemo kratke funkcije za enkratno uporabo. Zato večina modernih programskih jezikov (Java, Python, C++, ...) pozna anonimne oz. lambda funkcije. Sintaksa v C++ je `[zunanje spremenljivke](argumenti funkcije) { vsebina }`. Če so zunanje spremenljivke prazne, to

pomeni, da lahko vsebina funkcije dostopa samo do svojih argumentov in do globalnih spremenljivk. Vse spremenljivke in argumente lahko funkcija sprejme po vrednosti ali po referenci.

V spodnjem primeru bomo ponovno uredili seznam števil, vendar jih bomo tokrat primerjali po abecedi namesto po vrednosti.

[22]:
```
sort(sez.begin(), sez.end(), [](int a, int b) {
    return to_string(a) < to_string(b);
});
for (int x : sez) cout << x << " ";
cout << endl;
```

```
11 2 41 7 8
```

1.8 Ostalo

C++ seveda ponuja še veliko več. Do tu smo predstavili samo nekaj osnov, ki nam bodo koristile pri reševanju algoritmičnih problemov v nadaljevanju.

Kogar zanima več, si lahko na spletu prebere o temah, kot so: razredi (*classes*), predloge (*templates*), pametni kazalci (*smart pointers*), niti (*threads*), izjeme (*exceptions*), preobremenjevanje operatorjev (*operator overloading*), ...