Ezoteryczne Kartki
Pierwsza krucjata
„C z ++”

Jakub Bachurski

wersja 1.2.0

Niektóre nagłówki są odnośnikami do dokumentacji.

1 Język

1.1 C++(98)

1.1.1 Makra

> With great power comes great responsibility.
>
> — Władca Makr

Makrami można osiągnąć bardzo dużo.

```
// Skracamy powtarzalny zapis
#define for4bit(__var) for(int __var = 0; __var < 16; __var++)
for4bit(a) for4bit(c) for4bit(c) for4bit(d)
    cout << a << ", " << b << ", " << c << ", " << d << endl;

// Makra są czasem przydatne do debugu.
#define note(__var) #__var << " = " << __var
int x = 3;
cout << note(x) << endl; // "x = 3"
```

1.1.2 Referencje

```
// mniejsza z przekazanych zmiennych jest inkrementowana
int increment_minimum(int& x, int& y)
{
    if(x < y) x++;
    else      y++;
}
```

Normalnie, gdy przekazujemy parametr do funkcji jest on zawsze kopiowany, co zajmuje cenny czas. Przekazując referencję unikamy tego.

Notacja `const T&` to const-referencja. Jest to bardzo przydatny rodzaj referencji, która nie może być modyfikowana, ale na pewno wartość nie będzie kopiowana. Przekazując typy fundamentalne do funkcji, najlepiej nie przekazywać ich `const T&`, bo nie będzie to szybsze od zwykłego skopiowania. W ogólności `const` obiecuje kompilatorowi, że nie będziemy modyfikować danej wartości.

1.1.3 Conditional operator

```
// wartość równa 42 jeżeli `condition`, wpp. 24
(condition ? 42 : 24)
```

1.1.4 Aggregate initialization, wygodna inicjalizacja

Działa tylko dla bardzo prostych typów (które napisaliśmy). W typach z STL czasem będzie podobna notacja, ale implementuje się ją trochę inaczej (listy inicjalizacyjne – `std::initializer_list`).

```
struct wektor { int x, y; };
wektor u = {3, 5}, v = {1, 2};
// u.x = 3, u.y = 5, v.x = 1, v.y = 2 (agregat)

// inicjalizacja z listy inicjalizacyjnej
vector<int> vec = {1, 3, 3, 7};
```

1.1.5 Operator overloading

```
struct wektor { int x, y; }

wektor operator+ (wektor p, wektor q)
{
    return {p.x + q.x, p.y + q.y};
}

// funktory mają zdefiniowany operator() (wywołania).
// Komparator to funktor który zwraca,
//  czy dany element jest *mniejszy* od innego.
struct Cmp // taki komparator można wykorzystać np. w std::set
{
    bool operator() (int x, int y) { return x > y; }
} cmp; // `cmp` to instancja typu `Cmp`
```

1.2 C++11

1.2.1 auto

```
set<pair<int, string>> S = {{1, "a"}, {3, "c"}, {2, "b"}};
auto it = S.lower_bound({1, "b"});
```

1.3 Funkcje anonimowe lambda

```
stable_partition(
    vec.begin(), vec.end(),
    [](int x) {
        return x < 4;
    }
);
// [] nie łapie nic, [&] złapie wszystko jako referencję,
// [=] kopiuje wszystko. istnieje notacja od pojedynczego capture
```

1.4 Range for

```
vector<int> vec = {4, 2, 0, 1, 3, 3, 7};
for(auto x : vec)
    cout << x << " ";
cout << endl;
/* 4 2 0 1 3 3 7 */

for(int c : {0, 1})
{
    for(auto x : vec)
        cout << x << " ";
    cout << endl;
    reverse(vec.begin(), vec.end());
}
/* 4 2 0 1 3 3 7
   7 3 3 1 0 2 4 */
```

1.5 C++17

1.5.1 Structured bindings

```
// Więcej o pair i tuple poniżej
pair<int, string> student = {2, "Fedora"};
auto [i, name] = student;
tuple<int, char, int> data = {1, '2', 3};
auto& [u, v, w] = data;
w++;
// data == {1, '2', 4}
```

2 Biblioteka standardowa

2.1 Często używane algorytmy

[Algorytmy] pozwalają nam nie pisać powtarzalnego kodu, który trzeba przeanalizować podczas debugowania. Zamiast tego możemy skorzystać z prostej funkcji.

- `partition, stable_partition, merge, nth_element, reverse, iota`
- `includes, set_union, set_intersection` (rodzina `set_*`)
- `unique` – usuwanie duplikatów
- `min, max, min_element, max_element` – warto znać `min({a, b, c})`.
- `gcd, lcm` – wprowadzone w C++17 funkcje od liczenia NWW i NWD.

2.2 Funktory

W `<functional>` mamy zbiorek wielu prostych funktorów, których możemy użyć wykorzystując `<algorithm>`. Na przykład `std::plus<T>`, `std::less<T>` i `std::greater<T>`.

2.3 Kontenery

[Kontenery]

- `std::vector<T>` – pozycja obowiązkowa, ładna dynamiczna tablica. Warto zwrócić uwagę na `reserve` (poprawia wydajność) i pozostałe funkcje. Konstruowany wektor zawsze jest wyzerowany.
- `std::set<T>, std::map<T>` (i wersje `multi`) – struktury utrzymujące posortowane zbiory lub mapowania. Operacje w $O(\log n)$.
- `std::unordered_map<T>, std::unordered_map<T>` – nieposortowane zbiory i mapy, które czasem są szybsze (warto zwrócić uwagę na `reserve` – najlepiej zarezerwować 3-4 razy więcej miejsca). Średni czas „$O(1)$”.
- `std::array<T, N>` – wrapper tablicy, który możemy łatwo przekazywać między funkcjami jako parametr i wartość zwracana. Warto go użyć zamiast vectora, jeżeli wymagany rozmiar jest bardzo mały, a będziemy chcieli często się iterować lub kopiować – może to być znacznie szybsze.
- `std::deque<T>` – kolejka obustronna, która pozwala na indeksowanie w czasie stałym. Jest trochę wolniejsza niż `vector`.
- `std::stack<T>, std::queue<T>` – klasyczna kolejka i stos.
- `std::list<T>` – lista dwukierunkowa
- [`std::valarray<T>`] – rzadko używany, lecz czasem przydatny
- [`std::bitset<N>`] – (Operacje bitowe!)

2.4 pair, tuple, tie

Para to prosty typ z konstruktorami, zdefiniowanym porównywaniem (leksykograficznym) i działającym structured binding.

```
pair<int, char> p = {3, 'D'}; // p.first == 3, p.second == 'D'
```

Krotka to rozszerzenie par, która może mieć więcej niż dwa typy. Interfejs sprawdzania poszczególnych elementów jest trochę niewygodny, ale mamy wbudowane porównywanie.

```
tuple<int, char, string> t = {5, 'D', "fenwick"};
// get<0>(t) == 5, get<1>(t) == 'D', get<2>(t) == "fenwick"
```

`make_pair(a, b)` i `make_tuple(a, b, c, ...)` to wygodne funkcje tworzące pary i krotki bez potrzeby pisania typu ręcznie. `tie` to specjalny konstruktor krotki, który tworzy specjalne referencje „lewostronne", co pozwala nam odpakować krotkę (jest to przydatne przede wszystkim przed C++17, kiedy wprowadzono podobne structured bindings).

```
int a; char b; string c;
tie(a, b, c) = t;
```

3 Szablony

Krótki wstęp: `https://www.tutorialspoint.com/cplusplus/cpp_templates.htm`

```
template<typename T>
T hasz(T a, T b, T c) { return a + 3*b + 9*c; }

// hasz to szablon
// hasz<int>, hasz<long>, hasz<char> to różne funkcje
// Gdy piszemy hasz(1, 2, 3) kompilator zgaduje, że chodzi o
// hasz<int>.

template<typename T>
using min_priority_queue = priority_queue<T, vector<T>, greater<T>>
```

4 Różności

```
// w bits/stdc++.h są praktycznie wszystkie nagłówki z STLa,
// więc nie musimy o nich sami pamiętać, ani szukać w dokumentacji
// (lecz jest to raczej zła praktyka w Profesjonalnym Programowaniu)
//  -> może to wydłużyć czas kompilacji
#include <bits/stdc++.h>
```

```cpp
using namespace std;

using graph_t = vector<vector<pair<size_t, int>>>;

template<typename T>
using min_priority_queue = priority_queue<T, vector<T>, greater<T>>;

const int64_t oo = (1ll << 60); // 2^60, 1ll to 1 typu long long

// Ekstrawagancka Dijkstra
vector<int64_t> sssp(const graph_t& graph, size_t source)
{
    const size_t n = graph.size();

    vector<int64_t> dist(n, +oo);
    vector<bool> lock(n);
    min_priority_queue<pair<int, size_t>> Q;

    auto maybe = [&](size_t u, int64_t d) {
        if(d < dist[u])
            Q.emplace(dist[u] = d, u);
    };
    maybe(source, 0);
    while(not Q.empty())
    {
        auto [_, u] = Q.top(); Q.pop();
        if(lock[u]) continue;
        lock[u] = true;
        for(auto [v, w] : graph[u])
            maybe(v, dist[u] + w);
    }
    return dist;
}

// cout wykorzystuje operator overloading
template<typename T>
ostream& operator<< (ostream& out, const vector<T>& vec)
{
    out << '{';
    for(auto it = vec.begin(); it != vec.end(); ++it)
    {
        if(it != vec.begin()) out << ", ";
        out << *it;
    }
    return out << '}';
}
```

```cpp
// ...cin również
// (W obu przypadkach musimy zwrócić strumień - możemy wtedy
//  wykorzystać to, że jest on zawsze zwracany przy wypisywaniu)
template<typename U, typename V>
istream& operator>> (istream& in, pair<U, V>& p)
{
    return in >> p.first >> p.second;
}

int main()
{
    double a = 3, b = 2;
    complex<double> z = a + b*1i; // liczby zespolone!

    // sumy prefiksowe
    vector<int> A = {1, 2, 3, 4, 5};
    vector<int> S = {0};
    partial_sum(A.begin(), A.end(), back_inserter(S));
    // S = {0, 1, 3, 6, 10, 15}

    // Wartości niezinicjalizowane są niezdefiniowane
    // -> dotyczy to m.in. typów fundamentalnych, tablic, std::array
    //    i prostych structów (czyli głównie agregatów)
    int trash_v;
    int trash_c_arr[4];
    array<int, 4> trash_arr;

    // Należy zaznaczyć, że chcemy zainicjalizować typ.
    int v = 0;
    int c_arr[4] = {};
    array<int, 4> arr = {};
}
```

5 Koszty abstrakcji

Dekompilator online: `https://godbolt.org/`.

Kompilator jest głupiutki i czasem małe zmiany mogą zrobić dużą różnicę. Oznaczenie odpowiednych zmiennych jako `const` pozwala mu na bardziej kreatywne optymalizacje. Warto uważać na tworzenie małych struktur (np. vectorów) – lepiej robić małe tablice. Szablony bywają wolniejsze, ale głównie gdy kod jest bardzo długi i kompilator uznaje optymalizacje za niewarte zachodu. Z pewnością sytuacje, gdy przez błąd kompilatora (bo coś źle zoptymalizuje) kod jest 2 razy wolniejszy, są bardzo rzadkie.