Kto lepszy?

#porządek_leksykograficzny
#komparator
#referencja #&
#operator #<

Kilka razy wspominaliśmy już o sortowaniu tablic czy wektorów, przy czym do tej pory elementy tych struktur danych były raczej prostych typów,* jak liczby czy znaki. Sortowaliśmy też wektor `string`-ów i nie było z tym problemu. Dlaczego? Otóż wszystkie wymienione typy mają z góry określoną relację porządkującą, to znaczy, że dla dowolnej pary wartości danego typu możemy określić, która wartość jest większa, a która jest mniejsza (mogą oczywiście też być równe). Z liczbami czy znakami problemu w ogóle nie ma, natomiast w przypadku ciagów znaków obowiązuje *porządek leksykograficzny*, czyli na przykład **abc** jest „mniejsze niż" **xyz**, zaś **Ala** jest „mniejsza niż" **Alabama** (tak jak w słowniku czy encyklopedii).

Co jednak mamy zrobić, jeśli mamy do portowania wektor złożony z par – na przykład par stringów? Czy zadziała zwykła funkcja `sort()`? Otóż zadziała, ale rezultat może nieco odbiegać od naszych oczekiwań. Prześledźmy to dokładnie, tworząc wektor kilku postaci z „Gry o tron". Przy okazji nauczymy się używać instrukcji `typedef`, która pozwala nadać identyfikator wybranemu przez nas typowi danych:

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;

typedef pair<string, string> GoT;

int main()
{
  vector<GoT> P(5);
  P[0] = make_pair("Jon", "Snow");
  P[1] = make_pair("Sansa", "Stark");
  P[2] = make_pair("Arya", "Stark");
  P[3] = make_pair("Cersei", "Lannister");
  P[4] = make_pair("Tyrion", "Lannister");
  sort(P.begin(), P.end());
  for(int i=0; i<5; i++)
    cout << P[i].first << ' ' << P[i].second << endl;
}
```

*Mówi się na nie również: typy podstawowe.

I tu nas czeka siurpryza, bo na ekranie pojawią się personalia bohaterów GoT w kolejności alfabetycznej, ale według imion:

```
Arya Stark
Cersei Lannister
Jon Snow
Sansa Stark
Tyrion Lannister
```

To znaczy, że funkcja `sort()` porządkuje strukturę danych zawierającą pary względem pierwszej składowej, czyli `first`. A jak spowodować, aby ten wektor został posortowany według nazwisk? Jest na to sposób, bo funkcja `sort()` może przyjąć jeszcze jeden dodatkowy argument: wskazanie funkcji porównującej, czyli *komparatora* (od ang. **compare**, wymawiaj: *komper*, z akcentem na *e*). Taka funkcja powinna mieć dwa argumenty typu `GoT` i zwracać wartość prawdy, jeśli pierwszy z nich jest mniejszy od drugiego, a wartość fałszu – w przeciwnym przypadku. Na przykład mogłaby wyglądać tak (nazwa funkcji jest dowolna, a jej definicja powinna znaleźć się po instrukcji `typedef`, a przed funkcją `main()`):

```
bool compare_GoT(GoT x, GoT y)
{
  return x.second < y.second;
}
```

Instrukcja warunkowa `if` nie jest potrzebna, bo zdanie logiczne po słowie `return` ma właśnie taką wartość, jaką należy zwrócić.

Wywołanie funkcji `sort()` powinno teraz wygladać tak:

```
sort(P.begin(), P.end(), compare_GoT);
```

Proszę zwrócić uwagę, że po nazwie komparatora nie piszemy nawiasów okrągłych, gdyż wtedy byłoby to jego wywołanie, które zresztą spowodowałoby błąd kompilacji (dlaczego?). W ten sposób przekazuje się wskazanie (wskaźnik) do funkcji, która powinna być użyta przy porównywaniu par.

Tym razem dostajemy taką oto kolejność:

```
Cersei Lannister
Tyrion Lannister
Jon Snow
Sansa Stark
Arya Stark
```

Prawie że dobrze, tylko Arya powinna być przed Sansą, bo tak się układa wszelkie listy osobowe. To znaczy, że komparator musi oddzielnie rozważyć przypadek, gdy nazwiska (składowe `second`) są takie same: wówczas będą decydować imiona (składowe `first`):

```
bool compare_GoT(GoT x, GoT y)
{
  if(x.second == y.second)
    return x.first < y.first;
  return x.second < y.second;
}
```

Instrukcja `else` nie jest potrzebna (dlaczego?).

Tym razem dostajemy już poprawną kolejność:

```
Cersei Lannister
Tyrion Lannister
Jon Snow
Arya Stark
Sansa Stark
```

W zasadzie można by na tym poprzestać, ale są jeszcze dwa detale do omówienia, takie wisienki na torcie. Nasze przykłady są bardzo proste i dotyczą przetwarzania niewielkich danych, ale zawsze warto poznać i utrwalić sobie dobre zwyczaje programistyczne.

Po pierwsze powinniśmy poprzedzić argumenty komparatora symbolem *referencji* `&`, który będzie oznaczał, że funkcja uzyska dostęp do oryginalnych zmiennych, a nie otrzyma tylko ich skopiowane wartości. Zaletą takiego podejścia jest przesyłanie mniejszej ilości danych do funkcji, co daje większą efektywność programu. W przypadku zwykłego podania argumentów cała ich zawartość jest kopiowana do funkcji (a może to być całkiem spora porcja danych), a w przypadku referencji kopiowany jest tylko adres zmiennej.

Jest tu pewien haczyk: skoro funkcja ma dostęp do oryginalnych zmiennych, to znaczy, że może zrobić im kuku i zmienić ich wartość. W przypadku komparatora nigdy tak nie powinno być, zatem argumenty funkcji należy opatrzeć słowem kluczowym `const`, które będzie oznaczać, że oryginalne zmienne są bezpieczne (i to jest ta druga wisienka).

Ostatecznie komparator powinien wyglądać tak:

```
bool compare_GoT(const GoT &x, const GoT &y)
{
  if(x.second == y.second)
    return x.first < y.first;
  return x.second < y.second;
}
```

Operator mniejszości <

Warto wspomnieć o jeszcze jednym sposobie wpływania na sposób sortowania struktur danych o złożonych elementach. Można pokusić się o zdefiniowanie specjalnej wersji operatora mniejszości `<` dla naszego typu danych. Pokażemy to na przykładzie sortowania wektora struktur, które reprezentują punkty w przestrzeni. Posortujemy je rosnąco (dokładniej: niemalejąco) względem odległości od początku układu współrzędnych.

Najpierw zdefiniujemy naszą strukturę, którą nazwiemy `point`:

```
struct point
{
  double x, y, z;

  point(double xx, double yy, double zz)
  {
    x = xx;  y = yy;  z = zz;
  }

  point()
  {
    x = y = z = 0.;
  }
};
```

Jak widać, zdefiniowaliśmy dwa konstruktory – dla wygody późniejszego zapisu.

Funkcja `main()` wygląda podobnie, jak w przykładzie o parach:

```
int main()
{
  vector<point> T(5);
  T[0] = point(1., 2., 3.);
  T[1] = point(0., 1., 0.);
  T[2] = point(-1., 0., -2.);
  T[3] = point(2., 3., 4.);
  T[4] = point(1., -1., 1.);
  sort(T.begin(), T.end());
  for(int i=0; i<5; i++)
    cout << T[i].x << ' ' << T[i].y << ' ' << T[i].z << endl;
}
```

Wektor T zawiera przykładowe punkty. *Notabene*, konstruktor bezargumentowy był konieczny, aby móc zadeklarować wektor z podanym rozmiarem (dlaczego?).

Aby jednak to działało, musimy przed funkcję `main()` wstawić definicję operatora mniejszości dla punktów. Przy okazji posłużymy się makrem `sqr()`, które oblicza kwadrat swego argumentu (pojawiło się ono w jednym z poprzednich podrozdziałów). Punkt A uznajemy za mniejszy od punktu B, jeśli jego odległość euklidesowa od początku układu współrzędnych jest mniejsza od tej dla punktu B. Zamiast odległości wygodniej porównywać ich kwadraty, a wynik będzie taki sam:

```
#define sqr(x) ((x)*(x))

bool operator<(const point &A, const point &B)
{
  return sqr(A.x) + sqr(A.y) + sqr(A.z) < sqr(B.x) + sqr(B.y) + sqr(B.z);
}
```

Przypomina to bardzo definicję komparatora, nieprawdaż?

W wyniku działania tego programu dostajemy taki oto wydruk:

```
0 1 0
1 -1 1
-1 0 -2
1 2 3
2 3 4
```

Na oko widać, że kolejność punktów jest prawidłowa.

Pozostaje skomentować, kiedy używać komparatora, a kiedy definiować własny operator mniejszości. Obie metody są dobre, ale komparatorów możemy mieć kilka (dla tego samego typu danych) i wtedy możemy wybierać różne kryteria sortowania.