ISSP/C++: lista 6

Temat: funkcje i klasy

A. **Funkcje**.

1) Bartek napisał program testujący swoje umiejętności w posługiwaniu się funkcjami, jednak pech chciał, że postawiona na laptopie kawa spowodowała nieodwracalne zniknięcie części funkcji. Aby ułatwić Ci zadanie, Bajtek zapisał swój uszkodzony plik w repozytorium ISSP_CPP (GitHub) jako `zadania/programy/class_and_fun/coffeed_fun.cpp` Zachował się też plik z wynikiem działania programu:

```
> ./a.out
v = [3, 4, -6, 8, 5]
sum(v) = 14
reversed v = [5, 8, -6, 4, 3]
resized v = [5, 8, -6, 4, 3, 0, 0, 0, 0, 0,...]

*** performing iota ***
v = [0, 1, 2, 3, 4]
sum(v) = 10
reversed v = [4, 3, 2, 1, 0]
resized v = [4, 3, 2, 1, 0, 0, 0, 0, 0, 0,...]
good bye!
```

Pomóż Bajtkowi odtworzyć brakujące funkcje:

a) Funkcję **`sum`** wyznaczającą sumę elementów wektorów typu **`std::vector<int>`**
b) Funkcję **`reverse`** odwracającą kolejność elementów wektorów typu **`std::vector<int>`**
c) Funkcję **`iota`**, która kolejnym elementom wektora typu **`std::vector<int>`** przypisze kolejne liczby całkowite poczynając od zera (tj. po wywołaniu **`iota(v)`** powinno zachodzić **`i == v[i]`** dla wszystkich elementów **`v`**).

Uwaga: Najrozsądniej jest najpierw napisać kadłubki brakujących funkcji, tzn. same prototypy funkcji z pustymi lub trywialnymi implementacjami (np. return 0;). Kadłubki dadzą Ci możliwość kompilowania kodu na bardzo wczesnym etapie jego tworzenia, co ułatwia diagnostykę błędów.

B. Ania zauważyła, że implementacja operatora<< w powyższym zadaniu jest błędna, tj. w pewnych warunkach ten operator powoduje segfault. Czy potrafisz bez uruchamiania testów, na podstawie samej lektury kodu tej funkcji, domyślić się, która instrukcja może spowodować segfault? Spróbuj tę funkcję wywołać na pustym wektorze. Popraw implementację tak, by wyeliminować ten błąd.

C. (*) Nieszczęście! Bajtek zalał kawą część swojej klasy **X**, która służyła mu do zliczania liczby operacji porównania (**`operator<`**) i przypisania (**`operator=`**) użytych w funkcji **`std::sort`** służącej do porządkowania obiektów od najmniejszego do największego. Na szczęście zachował się fragment pliku (`zadania/programy/class_and_fun/coffeed_class.cpp`) oraz plik z wynikami programu:

```
> ./a.out
N =         10, op<:          22 = 0.66*N*log2(N);  op=:          46 = 1.38*N*log2(N)
N =        100, op<:         781 = 1.18*N*log2(N);  op=:         525 = 0.79*N*log2(N)
N =       1000, op<:       11820 = 1.19*N*log2(N);  op=:        6772 = 0.68*N*log2(N)
N =      10000, op<:      161724 = 1.22*N*log2(N);  op=:       84268 = 0.63*N*log2(N)
N =     100000, op<:     2016583 = 1.21*N*log2(N);  op=:     1005573 = 0.61*N*log2(N)
N =    1000000, op<:    23978809 = 1.20*N*log2(N);  op=:    11653809 = 0.58*N*log2(N)
N =   10000000, op<:   279796899 = 1.20*N*log2(N);  op=:   132245571 = 0.57*N*log2(N)
N =  100000000, op<:  3218649681 = 1.21*N*log2(N);  op=:  1477183659 = 0.56*N*log2(N)
```

Pomóż Bajtkowi odtworzyć klasę **X**!

a) Wskazówka: obiekty klasy **X** otrzymują podczas swojej konstrukcji losową wartość, więc program Bajtka testuje liczbę operacji **<** oraz **=** dla wektora **N** losowo wybranych liczb całkowitych mieszczących się w zakresie liczb typu **`int`**.

b) Zapis **`N = 10, op<: 22, op=: 46`** oznacza, że dla **N**=10 funkcja sortująca 22 razy wywołała **`operator<`** oraz 46 razy wywołała **`operator=`** w klasie **X**.

c) Za każdym razem program produkuje nieco inne wyniki (dlaczego?)

d) Gotowy program kompiluj w trybie Release, chyba że masz bardzo dużo wolnego czasu.

e) I to tyle wskazówek, dalej pracuj samodzielnie.

D. (*) Spróbujmy teraz czegoś ekstra.

a) Przejdź do konsoli Linuxa (użytkownicy Windows mogą zalogować się na komputerze z systemem Linux, np. na pracowni studenckiej; ja w tym celu korzystam z programu putty), po czym skompiluj i uruchom program, który uzupełniłeś w poprzednim zadaniu, w następujący sposób:

```
> g++ coffeed_class.cpp -std=c++11 -O3
> time ./a.out
```

u mnie powoduje to wyświetlenie następujących danych:

```
N =         10, op<:          27 = 0.81*N*log2(N);  op=:          26 = 0.78*N*log2(N)
N =        100, op<:         838 = 1.26*N*log2(N);  op=:         540 = 0.81*N*log2(N)
N =       1000, op<:       12242 = 1.23*N*log2(N);  op=:        6963 = 0.70*N*log2(N)
N =      10000, op<:      156662 = 1.18*N*log2(N);  op=:       85090 = 0.64*N*log2(N)
N =     100000, op<:     2030810 = 1.22*N*log2(N);  op=:     1000170 = 0.60*N*log2(N)
N =    1000000, op<:    24355632 = 1.22*N*log2(N);  op=:    11609326 = 0.58*N*log2(N)
N =   10000000, op<:   285354349 = 1.23*N*log2(N);  op=:   131435454 = 0.57*N*log2(N)
N =  100000000, op<:  3226608519 = 1.21*N*log2(N);  op=:  1475329483 = 0.56*N*log2(N)

real    0m9.271s
user    0m9.239s
sys     0m0.036s
```

Zanotuj czas (real) wykonania programu na Twoim komputerze

b) A teraz obiecane „coś ekstra”. Zmień sposób kompilacji zgodnie z przedstawionym poniżej schematem, ponownie uruchom program i zanotuj czas jego wykonania:

```
> g++ -std=c++11 coffeed_class.cpp -O3 -D_GLIBCXX_PARALLEL -fopenmp
> time ./a.out
N =        10, op<:         40 = 1.20*N*log2(N);  op=:         39 = 1.17*N*log2(N)
N =       100, op<:        803 = 1.21*N*log2(N);  op=:        536 = 0.81*N*log2(N)
N =      1000, op<:       7957 = 0.80*N*log2(N);  op=:       5336 = 0.54*N*log2(N)
N =     10000, op<:      20225 = 0.15*N*log2(N);  op=:      16021 = 0.12*N*log2(N)
N =    100000, op<:     248892 = 0.15*N*log2(N);  op=:     172755 = 0.10*N*log2(N)
N =   1000000, op<:    2901573 = 0.15*N*log2(N);  op=:    1916794 = 0.10*N*log2(N)
N =  10000000, op<:   34453921 = 0.15*N*log2(N);  op=:   21175552 = 0.09*N*log2(N)
N = 100000000, op<:  399708050 = 0.15*N*log2(N);  op=:  230508911 = 0.09*N*log2(N)

real    0m3.721s
user    0m16.367s
sys     0m0.484s
```

Magia! Na moim komputerze (Intel i7) program potrzebował tylko 3.72 zamiast 9.27 sekund! Jednocześnie spędził aż 16.367 sekund w trybie użytkownika i 0.484 sekundy w trybie jądra. Życie jest pełne takich niespodzianek i nie zawsze wiemy, gdzie szukać gotowych odpowiedzi. Jesteś już dość zaawansowanym użytkownikiem komputerów. **Samodzielnie znajdź rozwiązanie powyższego paradoksu.**

c) To teraz jedziemy już całkiem po bandzie i uruchamiamy ten sam program (bez rekompilacji!) w innowacyjny sposób:

```
> OMP_NUM_THREADS=2 time ./a.out
N =        10, op<:         32 = 0.96*N*log2(N);  op=:         29 = 0.87*N*log2(N)
N =       100, op<:        813 = 1.22*N*log2(N);  op=:        572 = 0.86*N*log2(N)
N =      1000, op<:       9589 = 0.96*N*log2(N);  op=:       5484 = 0.55*N*log2(N)
N =     10000, op<:      79302 = 0.60*N*log2(N);  op=:      47510 = 0.36*N*log2(N)
N =    100000, op<:     975000 = 0.59*N*log2(N);  op=:     556607 = 0.34*N*log2(N)
N =   1000000, op<:   12166420 = 0.61*N*log2(N);  op=:    6300349 = 0.32*N*log2(N)
N =  10000000, op<:  140967878 = 0.61*N*log2(N);  op=:   71168916 = 0.31*N*log2(N)
N = 100000000, op<: 1590341642 = 0.60*N*log2(N);  op=:  788347640 = 0.30*N*log2(N)
10.82user 0.06system 0:06.35elapsed 171%CPU (0avgtext+0avgdata 782796maxresident)k
0inputs+0outputs (0major+3865minor)pagefaults 0swaps
```

Jesteś już całkiem nieźle zaawansowanym użytkownikiem komputerów. **Samodzielnie wyjaśnij**, o co chodzi z tym `OMP_NUM_THREADS=2` i ile sekund pracował program. Poeksperymentuj z różnymi wartościami `OMP_NUM_THREADS` (np. 1, 2, 4, 8, 16, 32) i ich związkiem z czasem wykonywania się programu. Notabene: czy potrafisz odpowiedzieć na pytanie, dlaczego polecenie `time` nagle zmieniło format wydruku swojego komunikatu? **Uwaga!** Użyta przez Ciebie implementacja klasy X niemal na pewno będzie fałszować liczbę operacji < i = (wskutek tzw. wyścigu), chyba że użyjesz `OMP_NUM_THREADS=1`. Na razie się tym nie przejmuj. Jeśli potrafisz samodzielnie rozwiązać i ten problem, zapraszam na egzamin w przedterminie.