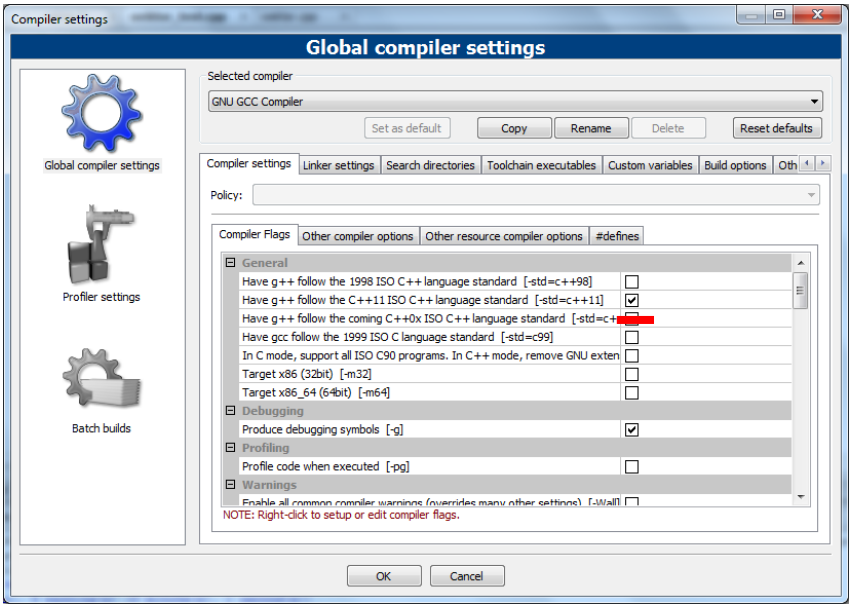
ISSP/C++: tydzień 5

Temat: klasy i przeciążanie operatorów

A. W podkatalogu `zadania\programy\wektor` znajdziesz program składający się z 3 plików: `wektor_test.cpp`, `wektor.cpp` i `wektor.h`.



1. Skompiluj ten program w C++11. Możesz w tym celu utworzyć projekt (w `code::blocks` lub `QtCreator`) i zaznaczyć w nim odpowiednią flagę, przestawiającą kompilator w tryb C++11, lub też z konsoli wydać polecenie
   `> g++ -std=c++11 *.cpp`
2. Uruchom program, zanotuj wyniki trzech testów:

```
======== testing operator+ performance (C++11) ========
> elapsed time (operator+): 2.03
======== testing operator+= performance (C++11) =======
> elapsed time (operator+=): 2.07
======== testing C-style performance ========
> elapsed time (C-style): 1.4
```

   (liczby powyżej uzyskałem na moim komputerze, są to czasy w sekundach)
3. Teraz zmień flagi na C++98, skompiluj program i go uruchom. Zanotuj wyniki. Uwaga: jeśli program kompilujesz za pomocą projektu, będziesz musiał(a) przebudować cały projekt.
4. Teraz dodaj flagę optymalizacyjną `-O2` lub `-O3` i powtórz testy. W przypadku `QtCreatora` i `Code::blocks` powyższą flagę włącza się, zmieniając tryb kompilacji z `Debug` na `Release`. Trzeba pamiętać o konieczności przebudowania całego projektu!
5. Wyniki zapisz w tabelce jak poniżej (wpisane liczby odpowiadają moim wynikom, w sekundach):

Test	*Debug (-g)*		*Release (-O2)*	
	C++11	*C++98*	*C++11*	*C++98*
operator+	2.03	2.8	0.39	0.68
operator+=	2.07	2	0.34	0.31
C-style	1.4	1.47	0.1	0.1

6. Każdy z testów wyznacza wartość tego samego wyrażenia. Zwróć jednak uwagę na to, że czas, w jakim obliczane jest to wyrażenie silnie zależy od trybu kompilacji (`Debug`/`Release`), sposobu implementacji testu, a nawet wersji języka C++.
7. Przedyskutuj z kolegą/koleżanką następujący problem: skoro wybór trybu kompilacji, `Debug` lub `Release`, może zmienić czas wykonania obliczeń o czynnik przekraczający 10, to po co w ogóle stosuje się „wolny” tryb `Debug`? Dlaczego jest to tryb domyślny kompilatora i środowisk programistycznych? Na końcu omówcie ten problem z prowadzącym ćwiczenia.

8. Czy z powyższej tabeli wynika, że wersja języka ma wpływ na szybkość działania testu „C-style” oraz „operator+=”? Wskazówka: nie tylko w fizyce wynik pomiaru bez oszacowania błędu tegoż pomiaru może być bardzo mylący. Powyższe testy można powtórzyć kilka razy, by oszacować ten błąd.
9. Zwróć uwagę na to, że tylko jedna implementacja testu, „operator+”, wyraźnie korzysta z przejścia na C++11, przy czym jest to przyspieszenie naprawdę znaczące. Poniżej spróbujemy odpowiedzieć sobie na pytanie, co takiego pojawiło się w C++11, co umożliwia takie przyspieszenie i dlaczego jest to ważne, a dlaczego niespecjalnie ważne.

B. Otwórz plik `wektor.h`.
1. Czy jego lektura umożliwia Ci odpowiedź na pytanie JAK można używać klasy `Wektor`?
2. Czy jego lektura umożliwia Ci odpowiedź na pytanie, z jakich danych składa się każdy `Wektor`?
3. Która część kodu w tym pliku zależy od wersji języka C++? Jakiego elementu języka C/C++ użyto do wyrażenia tej zależności kodu? Czyż to nie piękne, że można mieć jeden kod, który można uruchamiać w różnych środowiskach (w tym przypadku: kod nie zależy od tego, czy dostępny kompilator obsługuje C++11).
4. Czy w pliku nagłówkowym `wektor.h` oprócz *deklaracji* znajdują się też *definicje* funkcji? (wskazówka: definicja funkcji mówi nie tylko o tym, jak jej można użyć, ale zawiera też dokładny opis tego, co dana funkcja robi).
5. Ile konstruktorów zdefiniowano w klasie `Wektor`? Jak można je odróżnić?
6. Ile destruktorów zdefiniowano w klasie `Wektor` i dlaczego odpowiedź brzmi „jeden”, nawet gdyby żadnego destruktora w klasie nie zadeklarowano?
7. Ile operatorów indeksowania (operator[]) zadeklarowano w klasie `Wektor` i skąd kompilator wie, kiedy którego użyć?
   - Ustaw pułapkę w każdym z nich; włącz debuger; zbadaj, w którym miejscu programu głównego aktywowane są te operatory.
   - Czy ma na podstawie samej deklaracji w pliku nagłówkowym można przewidzieć, który z nich zostałby użyty w instrukcji `w[0] = 0;`, gdzie w jest obiektem klasy `Wektor`?
   - Czy na podstawie samej deklaracji można przewidzieć, który z nich zostałby użyty w instrukcji `a = w[0]`, gdzie w jest obiektem klasy Wektor z atrybutem `const`?
8. Jak z samej deklaracji rozpoznać, że funkcja zadeklarowana wyrażeniem
   `Wektor & operator= (Wektor const& wek);`
   może zmieniać stan obiektu, na rzecz którego została wywołana, ale nie może zmienić prawej strony operatora przypisania (np. w instrukcji `w = v;`)?
9. Jak z samej deklaracji rozpoznać, że wywołania operatora zadeklarowanego wyrażeniem
   `Wektor & operator= (Wektor const& wek);`
   można łączyć w ciągi, np. tak: `v = w = u;`?
10. W jaki sposób w pliku `wektor.h` ustawiono tzw. wartownika tego pliku? (por.: https://en.wikipedia.org/wiki/Include_guard)
11. Czy wszystkie operatory zdefiniowano wewnątrz klasy?
12. Dlaczego operator= ma jeden argument, a operator+ ma ich dwa?

C. Otwórz plik plik `wektor.cpp`.

2. Czy jego lektura umożliwia Ci odpowiedź na pytanie JAK można używać klasy `Wektor`?
3. Czy jego lektura umożliwia Ci odpowiedź na pytanie, z jakich danych składa się każdy `Wektor`?
4. Która część kodu w tym pliku zależy od wersji języka C++? Jakiego elementu języka C/C++ użyto do wyrażenia tej zależności kodu? Czyż to nie piękne, że można mieć jeden kod, który można uruchamiać w różnych środowiskach (w tym przypadku: kod nie zależy od tego, czy dostępny kompilator obsługuje C++11)?
5. Czy w pliku nagłówkowym `wektor.h` oprócz *definicji* znajdują się też *deklaracje* funkcji lub zmiennych? (wskazówka: deklaracja funkcji mówi tylko o tym, jak jej można użyć, czyli jaki jest jej interfejs, ale nie zawiera dokładnego opis tego, co dana funkcja robi).
6. Dlaczego niektóre funkcje w tym pliku, np. `operator=`, zawierają w nazwie wyrażenie `Wektor::`? Dlaczego nie było takiej potrzeby w pliku nagłówkowym klasy? Jaki jest związek tego wyrażenia z przestrzeniami nazw i klasami?
7. Czy plik źródłowy (`wektor.cpp`) potrzebuje swojego wartownika? Dlaczego?
8. Czy plik źródłowy włącza swój plik nagłówkowy? W którym miejscu?
9. Czym różni się `#include <...>` od `#include "..."`?. Wskazówka:
   - http://stackoverflow.com/questions/21593/what-is-the-difference-between-include-filename-and-include-filename
   - https://msdn.microsoft.com/en-us/library/36k2cdd4%28v=vs.110%29.aspx
   - https://gcc.gnu.org/onlinedocs/cpp/Include-Syntax.html

D. Otwórz plik programu głównego, `wektor_test.cpp` i spójrz na funkcję `main()`.

1. Jaka jest rola instrukcji `const int N = 50000000;`? W ilu miejscach ona jest ona wykorzystywana i jakie jest jej przeznaczenie? Czy łatwo byłoby zmienić wielkość problemu testowanego w tym programie? Po co więc parametryzuje się programy?
2. O co chodzi w instrukcji
   `const char* mode = (__cplusplus >= 201103L) ? "C++11" : "C++98";`?
   - Skąd się wziął i co oznacza identyfikator `__cplusplus`? (wskazówka: Google).
   - Jak działa operator `?:` ? Ile przyjmuje on argumentów?
   - Czy powyższą definicję można by zapisać przy pomocy instrukcji warunkowej `if`? Ile instrukcji by to zajęło?
   - Co w tej instrukcji oznacza słowo `const`: to, że zmienna `mode` jest stałą czy to, że zmienna ta jest wskaźnikiem na stałą?
3. Skąd kompilator „wie”, że w wyrażeniach `v[3]` i `vc[3]` powinien używać różnych funkcji `operator[]`?
4. Zastanów się nad instrukcją `v = v;` Sprawdź debugerem, że program wchodzi do funkcji
   `Wektor & Wektor::operator= (Wektor const& wek)`
   a w niej wykonuje kod
   ```
   if (this == &wek)
       return *this;
   ```
   - Skąd wiadomo, że dla instrukcji `v = v;` w powyższym kodzie wartość `this` będzie równa `&wek`?
   - I co w ogóle oznacza `&` w wyrażeniu `&wek`?
   - Co oznacza identyfikator `this`?

- Czy potrafisz uzasadnić, że bez powyższej instrukcji `if` implementacja funkcji `operator=` byłaby tak bardzo błędna, że aż groziłaby padem programu? Wskazówki: nie wolno odwoływać się do obszaru pamięci zwolnionego operatorem `delete` (lub `delete []`) ani dwa razy zwalniać tego samego obszaru pamięci operatorem `delete` (lub `delete[]`) – por. destruktor.

5. W dalszej części program testuje funkcje make_wektor i make_simple_wektor. W trybie C++98 mój kompilator produkuje kod, który wyświetla następujące informacje diagniostyczne:

```
> testing make_wektor in a constructor
ctor [n]
ctor [n]
ctor [copy]
> testing make_wektor in operator=
ctor [n]
ctor [n]
ctor [copy]
operator= [copy]
> testing make_simple_wektor in a constructor
ctor [n]
> testing make_simple_wektor in operator=
ctor [n]
operator= [copy]
```

   Jak widać, funkcja `make_wektor` generuje wywołanie 3 konstruktorów, w tym jednego kopiującego, podczas gdy funkcja `make_simple_wektor` generuje wywołanie zaledwie jednego konstruktora, w tym żadnego kopiującego. Dlatego `make_simple_wektor` jest DUŻO szybsza. To, że `make_wektor` generuje aż dwa konstruktory "`ctor [n]`" jest jasne: w jej kodzie tworzy się dwa obiekty lokalne, `wynik` i `wynik2` (sprawdź to!). Konstruktor kopiujący wywoływany jest w momencie zwracania wartości przez funkcję, czyli generuje go instrukcja `return wynik;` lub `return wynik2;`. Dlaczego jednak funkcja `return wynik;` nie generuje konstruktora kopiującego w bardzo podobnej funkcji `make_simple_wektor`? Odpowiedź jest nieoczywista: kompilator może zoptymalizować kod tak, by wyeliminować wywołanie konstruktora w funkcji `return` – o ile może z góry przewidzieć, pod jakim adresem znajdować się będzie wynik. Jest to znana sztuczka optymalizacyjna ale też ważna cecha C++: wykonanie programu może zależeć od użytego kompilatora. Dlatego konstruktory nie powinny powodować ***efektów ubocznych***. Lektura:
   - https://pl.wikipedia.org/wiki/Skutek_uboczny_%28informatyka%29
   - https://en.wikipedia.org/wiki/Return_value_optimization

6. Dochodzimy do najważniejszej części programu, czyli testu wydajności trzech implementacji instrukcji dodawania trzech wektorów i zapisywania ich w czwartym: `w = v + v2 + v3;`. Pierwsza metoda to po prostu instrukcja

```
w = v + v2 + v3;
```

   wykorzystująca operator+ i operator=. Zacznijmy od pierwszego z nich:

```
Wektor operator+(Wektor lhs, Wektor const& rhs)
```

   - Który z argumentów powyższego operatora przekazywany jest do funkcji przez wartość? Czy przekazaniu argumentu przez wartość powinien towarzyszyć kostruktor kopiujący?
   - Który z argumentów powyższego operatora przekazywany jest do funkcji przez referencję?

- Czy jest to stała referencja?
- Jaka jest korzyść z korzystania ze stałych referencji, jeśli jest to możliwe? Skonsultuj ten problem z koleżanką/kolegą, a następnie z prowadzącym ćwiczenia.
- Czy korzystanie z referencji wiąże się z wywołaniem konstruktora kopiującego? (1: tak, zawsze; 2: tak, zależnie od możliwości optymalizacyjnych kompilatora; 3: nie, przenigdy!)
- Czyli referencja to taki wskaźnik (adres) „w przebraniu zmiennej”?

7. Wykonanie testu w trybach C++11 i C++98 daje (u mnie) następujące wyniki:

```
======== testing operator+ performance (C++11) ========
ctor [copy]
operator+=
ctor [move]
operator+=
ctor [move]
operator= [move]
> elapsed time (operator+): 1.981
======== testing operator+ performance (c++98) ========
ctor [copy]
operator+=
ctor [copy]
operator+=
ctor [copy]
operator= [copy]
> elapsed time (operator+): 2.792
```

- Jak widać, główna różnica polega na tym, że w C++11 kompilator generuje wywołania tzw. konstruktora przesuwającego,

  `Wektor::Wektor(Wektor && w)`

  natomiast w C++98 jest to konstruktor kopiujący,

  `Wektor::Wektor(Wektor const& w)`

  Różnica w składni polega na tym, że konstruktor kopiujący (ang. *copy constructor*) klasy `X` jako argument przyjmuje stałą referencję do `X`, czyli `X const&` (tu: `Wektor const& w`), natomiast konstruktor przesuwający (ang. *move constructor*) otrzymują jako argument specjalny typ referencji (ang. *rvalue reference*) oznaczany symbolem `&&` , czyli `X&&` (tu: `Wektor && w`). Generalnie ***zwykła referencja*** (***&***) do obiektu oznacza uchwyt (poprzez adres) do obiektu, czyli możliwość pracy na oryginału obiektu bez konieczności używania specjalnej, nieporęcznej składni typowej dla wskaźników. Z kolei wprowadzona w C++11 referencja ***&& oznacza referencję do obiektu, którego ważność właśnie się kończy*** – najczęściej jest to obiekt tymczasowy, który za chwilę przestanie istnieć. Typowy przykład takiego obiektu do wartość funkcji, o ile funkcja jako wartość zwraca obiekt przez wartość. W naszym przypadku właściwość tę ma `operator+`, który zwraca jako swoją wartość obiekt klasy `Wektor` przez wartość. Jeśli kompilator C++11 widzi, że funkcja zwraca obiekt klasy `X` przez wartość, a w klasie tej zdefiniowano konstruktor przenoszący, to w instrukcji `return X;` kończącej działanie tej funkcji kompilator wygeneruje wywołanie konstruktora przenoszącego. Jaką daje to korzyść? Otóż informacja, że argument konstruktora za chwilę straci ważność pozwala wykorzystać przydzielone temu argumentowi

zasoby, ZNACZNIE przyspieszając konstrukcję nowego obiektu. Można wręcz powiedzieć, że konstruktor przesuwający „kradnie” zasoby kontrolowane przez swój argument. Jest to dopuszczalne, skoro ten argument za chwilę zniknie. Porównajmy konstruktor przesuwający:

```
Wektor::Wektor(Wektor && w)
    : _rozmiar(w._rozmiar),
      _ptab (w._ptab)
{
    w._rozmiar = 0;
    w._ptab = nullptr;
}
```

Z konstruktorem kopiującym:

```
Wektor::Wektor(Wektor const& w)
    :  _rozmiar (w._rozmiar),
       _ptab(0)
{
    _ptab = new int [_rozmiar];
    for (size_t i = 0; i < _rozmiar; i++)
        _ptab[i] = w._ptab[i];
}
```

Konstruktor przesuwający może „ukraść” cały bufor pamięci (tu: `_ptab`) wraz z przechowywanymi w nim danymi! Całość kodu tego konstruktora mieści się w czterech prostych operacjach przypisania. Tymczasem konstruktor kopiujący musi wywołać bardzo kosztowny operator `new`, który przydziela (po negocjacjach z systemem zarządzającym pamięcią komputera) programowi nowy bufor danych, po czym musi skopiować do tego bufora wszystkie dane. Uwaga: po przejęciu zasobów ze swojego argumentu, konstruktor przenoszący musi ustawić ten argument w poprawnym stanie, wskazującym na wyczyszczenie zawartości obiektu tak, aby destruktor tego obiektu, który za chwilę zostanie wywołany, nie popełnił jakichś głupot. Dlatego w tym konstruktorze zerujemy składowe obiektu `w`! Możliwość użycia konstruktora przesuwającego w implementacji funkcji operator+ jest głównym powodem przyspieszenia testu w C++11 względem C++98.

- Klasa `Wektor` obok konstruktora przesuwającego posiada też przesuwający `operator=`. Sprawdź (debugerem), że on też jest wykorzystywany w implementacji C++11 instrukcji

  ```
  w = v + v2 + v3;
  ```

  w funkcji `main` (widać to także w komunikatach diagnostycznych). Sprawdź, że implementacja przesuwającego operatora przypisania jest znacznie prostsza i bardziej efektywna niż zwykłego operatora przypisania.

8. Przyjrzyj się działaniu drugiego testu, które tę samą operację wykonuje w 3 krokach:

   ```
   w = v;
   w += v2;
   w += v3;
   ```

   - Czy implementacja funkcji `operator+` posługuje się konstruktorem kopiującym?
   - Dlaczego w C++98 ten kod jest szybszy niż `w = v + v2 + v3 ;`?

9. Trzeci test polega na wywołaniu funkcji `add3`.
   - Czy funkcja `add3` jest jakoś szczególnie skomplikowana?
   - Czy sposób użycia funkcji `add3` jest bardziej niż mniej czytelny niż notacja operatorowa?
   - Czy funkcja `add3` posługuje się obiektami tymczasowymi?
10. Niech `N` oznacza liczbę elementów obiektów klasy `Wektor`. Wyznacz liczbę odczytów i zapisów pamięci w każdym z testów, tj.

    a) `w = v + v2 + v3 ;`
    b) `w = v;  w += v2;  w += v3;`
    c) `add3(w, v, v2, v3);`

    - Czy dostrzegasz związek między liczbą odczytów i zapisów z/do pamięci a wydajnością implementacji algorytmu?

Pora na małe podsumowanie:

1) Głównym obszarem zastosowań języka C++ są złożone programy, od których wymaga się pewności (=bezbłędności) działania, skalowalności, przewidywalności, szybkości i rozsądnego kosztu wytworzenia, testowania i utrzymania. Nowe referencje, `&&`, umożliwiają przyspieszenie kodu poprzez eliminację części zbędnych obiektów tymczasowych, jednak komplikują język.
2) O ile zastosowanie nowych „referencji przenoszących" umożliwiło nam redukcję czasu wykonania instrukcji `w = v + v2 + v3;` do czasu wykonania ciągu instrukcji `w = v; w += v2; w += v3;`, to i tak wydajność obu metod pozostaje daleko w tyle za wydajnością dedykowanej funkcji napisanej bez przeciążania operatorów.
3) Język C++ posiada nieomówione tu mechanizmy umożliwiające znaczące przyspieszenie wykonywania instrukcji w zapisie operatorowym, `w = v + v2 + v3`, jednak są one dostatecznie złożone i pracochłonne, byśmy je w tym kursie pominęli.
4) Skoro zwykłe funkcje pisze się łatwiej i szybciej, a do tego osiąga praktycznie 100% teoretycznej wydajności obliczeniowej, to wniosek jest prosty: **NIE PRZECIĄŻAJ OPERATORÓW ARYTMETYCZNYCH**, jeżeli zależy ci na wydajności programu, chyba że masz dużo wolnego czasu na cyzelowanie swojego programu.
5) Oto operatory, których przeciążanie ma sens: <<, >>, `( )`, `[]`, `new`, `new[]`, `delete`, `delete[]`. Dwa pierwsze przeciąża się do operacji wejścia/wyjścia (tzw. serializacja), `operator()` pozwala używać obiektów jak funkcji, `operator[]` stosuje się do implementacji tablic, wektorów i tym podobnych struktur, natomiast przeciążanie operatorów `new` i `delete` to poziom ekspercki.
6) Łatwiej napisać *efektywną* implementację operatorów przypisania, czyli =, +=, *=, etc. niż operatorów arytmetycznych, np. +, -, *, /, gdyż operatory przypisania zwykle nie wymagają wprowadzania do implementacji obiektów tymczasowych.
7) Skoro referencja do obiektu tymczasowego, `&&`, nie umożliwiła klasie `Wektor` osiągnięcia wydajności operatorów arytmetycznych porównywalnej z wydajnością zwyczajnych funkcji, to **NIE UŻYWAJ w swoim kodzie referencji &&**, chyba że osiągniesz poziom ekspercki i/lub ktoś ci za to zapłaci.
8) W takim wypadku – po co zajmujemy się tak długo przeciążaniem operatorów i referencjami przenoszącymi?
   - Przeciążanie operatorów <<, >>, `( )` i `[]` jest tak powszechne, także w bibliotece standardowej, że musisz mieć świadomość, jak ten mechanizm funkcjonuje (pamiętasz, że `qDebug()` do diagnostyki obiektów Qt używa właśnie bardzo wygodnej funkcji `operator<<`?);

- Referencje do obiektów tymczasowych, konstruktor przenoszący i przenoszący `operator=` weszły do biblioteki standardowej C++ i wkrótce upowszechnią się w innych bibliotekach; musisz więc rozumieć ten mechanizm choćby po to, by rozumieć komunikaty kompilatora.

E. Na koniec trzy (!) zadania do samodzielnego wykonania:
   1) Zoptymalizuj implementację funkcji
      `Wektor & Wektor::operator= (Wektor const& wek)`
      tak, by wyeliminować wykonywanie operacji `delete[]` i `new[]`, jeśli nie jest to konieczne.
   2) Napisz uproszczoną wersję klasy `Complex` implementującą klasę liczb zespolonych. Implementacja powinna być podzielona na plik nagłówkowy, źródłowy i plik z testem klasy. Klasa powinna zawierać:
      a. Konstruktor bezargumentowy, jednoargumentowy i dwuargumentowy
      b. Składowe służące do zapisu i odczytu części rzeczywistej i urojonej.
      c. `operator+`, `operator=`, `operator+=` i `operator<<`
   3) Dla wytrwałych. Skompiluj program omawiany na niniejszej liście z flagą `-pg`. Uruchom go. Po zakończeniu działania programu na Twoim dysku pojawi się plik `gmon.out`. Teraz w konsoli linuksowej uruchom komendę
      `gprof executable-filename gmon.out > output.txt`
      gdzie `executable-filename` oznacza nazwę Twojego pliku wykonywalnego. W pliku `output.txt` ujrzysz analizę wydajności poszczególnych funkcji programu (które zajęły najwięcej czasu, ile razy były wywoływane, jakie funkcje wywoływały jakie funkcje). Ciekawe jest porównanie wyników dla programu kompilowanego w trybie Debug i Release. Prosty, ale bardzo użyteczny program użytkowy.