Gliwice 18.01.2017

Laboratorium
Programowania Komputerów

14. Maraton leśny

Autor	Jarosław Bartoszuk
Wydział	AEiI
Kierunek	Informayka SSI
Semestr	III
Rok akademicki	2016/2017
Grupa	2
Prowadzący	dr inż. Karolina Nurzyńska
Data oddania	21.01.2017
Termin oddania	25.01.2017

1. Temat

W lesie zorganizowano bieg na dystansie **d**, w którym udział mogą brać ślimaki, stonogi i żółwie. Łączna liczba uczestników maratonu określona jest parametrem **l**, ilość uczestników każdego z rodzaju podlega losowaniu. W jednym momencie bieg może rozpocząć 10 zawodników. W przypadku większej liczby uczestników dzieleni są oni na grupy 10 zawodników, które rozpoczynają bieg w odstępach 1 minuty.

Ślimaki poruszają się z prędkością 5m/min jednak ich prędkość wzrasta w grupie o 10% po przyłączeniu się każdego nowego osobnika. Stonogi są najszybszymi biegaczami i pokonują dystans 7m/min. Jednak ze względu na dużą ilość nóg, które czasami im się plączą, ich prędkość spada o 5% przy uszkodzeniu jednej nogi. Prawdopodobieństwo, że stonoga uszkodzi sobie nogę – sama się nadepnie wynosi 10%. Stonoga, która utraci wszystkie nogi nie może poruszać się dalej. Żółwie są najwolniejszymi biegaczami i poruszają się z prędkością 4m/min. są jednak bardzo zdeterminowane, żeby wygrać i czasami posuwają się do przemocy. Jeżeli co najmniej trzy żółwie spotkają na trasie samotnego ślimaka i nie ma innych świadków, to rozgniotą domek ślimaka, co uniemożliwia mu dalszy ruch. Każdy żółw przebiegając obok stonogi stara się ją przydepnąć i uszkodzić trzy nogi stonogi. Zazwyczaj udaje mu się to z 75% prawdopodobieństwem.

Napisz program symulujący przebieg maratonu. Stan poszczególnych biegaczy rejestrowany jest z dokładnością jednej minuty. W wyniku działania programu powinien on określić kolejność z jaką zwierzęta dobiegną na metę. Podać miejsca w których musieli ukończyć bieg uszkodzone ślimaki i stonogi. W przypadku stonóg, które ukończyły bieg, należy podać z iloma zdrowymi nogami dotarły do mety.

Program uruchamiany jest z linii poleceń z wykorzystaniem następujących przełączników (kolejność jest dowolna):

-d długość trasy w metrach,

-l liczba uczestników,

-o plik wyjściowy ze statystykami.

2. Analiza projektu

Po lekturze tematu w dość jasny sposób określić można wymagania jakie trzeba zrealizować w projekcie. Narzucony został format zarówno danych wejściowych, jak i wejściowych. Drugi akapit w całości poświęcony został warunkom, jakie mają zostać spełnione w wykonywanym programie. Na wstępie zaznaczone zostało także to, że program ma korzystać z obiektowości języka C++, a funkcje globalne i rozbudowana funkcja główna nie mają prawa bytu. Pomimo tych wszystkich informacji w dalszym ciągu pozostaje kilka niedomówień, które jako wykonawca pozwoliłem sobie zrealizować wg własnego uznania. Nieokreślone zostały m.in. kontenery na dane, z jakich powinienem skorzystać, dlatego też wybrałem najprostszą w implementacji (w mojej opinii) formę – wektor. W temacie brakuje informacji na temat tego, co ma się stać w wyniku podania złych parametrów, lub też nieuzyskania dostępu do pliku wynikowego (niemożności stworzenia takowego). Uznałem więc, że w takich przypadkach wyświetlę stosowny komunikat, a program zakończy swe działanie zwracając 1-nkę, która świadczyć będzie o błędzie. Pomimo szerokiego opisu warunków, także w ich przypadku jest kilka drobnostek, które nie zostały napomknięte, jednak wrócę do nich przy okazji opisywaniu realizacji każdego z nich.

2.1. Analiza obiektowa

Pierwszy podział na klasy który przychodzi do głowy to stworzenie jednej klasy głównej określającej dowolne zwierzę biegnące w maratonie, a następnie klas pochodnych określających odrębności każdego z typu biegnących zwierzaków – ślimaka, stonogi, czy żółwia. Tego typu podział jest maksymalnie prosty i zrozumiały, dlatego musiał mieć miejsce w rozwiązaniu, w przypadku moje projektu są to klasy:

Zawodnik
└ Ślimak
└ Stonoga
└ Żółw

Klasa Zawodnik wg tematu powinna zawierać pola wspólne dla wszystkich zawodników, a więc ***prędkość*** maratończyka. Zastanawiając się jednak troszkę bardziej, możemy dojść do wniosku, że potrzebne będą także pola ***położenie*** i ***status***, które mówić nam będą czy zawodnik nadal biegnie, czy został już wyeliminowany z rywalizacji, a także na którym metrze zakończył swój udział. W dalszej części okazało się, że realizacja sprawdzania niektórych warunków wymaga, bym posiadał informację o tym z jakim zawodnikiem mam do czynienia. Dodana została więc zmienna znakowa reprezentująca zawodnika po pierwszej literze jego pełnej nazwy, zmienną tą nazwałem ***typ***.

Będąc świeżo po lekturze *„Czystego Kodu”,* autorstwa *Roberta Martina,* myślałem, że te 4 klasy będą w zupełności wystarczające. Resztę rzeczy realizowałem w funkcjach globalnych, wykorzystując

m.in. wspominaną na wykładach krotkę do przechowywania parametrów uruchomieniowych. Mają w funkcji głównej *main* proste instrukcje *odczytaj_parametry(); rozpocznij_bieg(); wypisz_wyniki();* byłem z siebie niemal dumny, jednak projekt nie na tym miał polegać, toteż byłem zmuszony dodać nieco więcej klas.

W pierwszej kolejności zrezygnowałem z nieprzyjemnej w zapisie krotki, na rzecz klasy *Parametry*, w której polach znalazły się dotychczasowe pola krotki – *długośćTrasy, ilośćUczestników, nazwaPlikuWyjściowego.* Funkcje globalne z nią związane ustąpiły miejsca metodom. Ten proces zmiany kodu był dość przyjemny.

Niezbędne okazało się także utworzenie klasy nadrzędnej - klasy realizującej założenia całego maratonu, klasy która zabierze wszystkie instrukcje z mojej funkcji *main* i pozwoli przełożyć je na nową kanwę – do swojego konstruktora. Z takich racji dodałem klasę *Bieg.*

2.2 Analiza problemu

O ile analiza zawartości obiektowej wynikała wprost z treści tematu, a wszelkie niedogodności spowodowane były moimi niedopatrzeniami, o tyle analiza warunków zawartych w drugiej części tematu jest o wiele bardziej rozbudowana. Czytałem warunki wielokrotnie, jednakże jedyne co można z nimi zrobić przed przystąpieniem do pisania i szukania rozwiązań w trakcie to określić ich ilość i ewentualne powiązania ze sobą.

Tak więc lista warunków odczytanych przeze mnie przed przystąpieniem do realizacji wyglądała następująco:

1) Bieg może rozpocząć w jednej chwili tylko 10 uczestników,
    a) jeśli jest ich więcej, rozpoczynają falami po 10 zawodników z odstępem czasowym równym 1 minucie.
2) Prędkość ślimaka ulega przyrostowi o 0.1 z każdym uczestnikiem biegu,
    a) jednakże trzeba pamiętać, że liczba uczestników biegu ulega zmianie, bo startują oni falami,
    b) przyjąłem, że przyrost odnosi się do wartości początkowej, nie dotychczasowej.
3) Nogi stonogi mogą zostać uszkodzone podczas biegu, utrata jednej nogi to utrata 5% prędkości,
    a) może uszkodzić je sobie sama, prawdopodobieństwo to 10%,
    b) mogą je uszkodzić żółwie, jeśli spotkają się na tym samym odcinku trasy, żółw z 75% prawdopodobieństwem uszkodzi 3 nogi stonogi.
4) Domek ślimaka może zostać zepsuty przez złe, nikczemne żółwie, stanie się tak, jeśli 3 żółwie spotkają na tym samym odcinku trasy 1 ślimaka i nie będzie żadnego świadka tego zdarzenia.

a) w tym miejscu należy się zastanowić, co jeśli świadkiem będzie inny żółw, zapewne biedny ślimak skończyłby z jeszcze bardziej popsutym domkiem, jednak na potrzeby projektu przyjąłem, że jeśli żółwi będzie więcej niż 3, to ich obawa przed wygadaniem tego niemoralnego czynu przez drugiego żółwia będzie zbyt duża i zostawią ślimaczka w spokoju.

5) Ostatni warunek powinien być właściwie pierwszym. Maraton trwa tak długo, dopóki choć jeden zawodnik dalej biegnie.

3. Specyfikacja zewnętrzna

Zgodnie z treścią zadania, komunikacja z programem ogranicza się do podania odpowiednich parametrów uruchomieniowych, w wyniku czego otrzymujemy plik wynikowy o nazwie jaką wybraliśmy.

3.1. Format danych wejściowych

Program oczekuje podania 3 parametrów uruchomieniowych, długości trasy, ilości uczestników i nazwy pliku wyjściowego. Aby uruchomić go z linii poleceń konieczne jest również podanie nazwy programu. Każdy parametr jest obligatoryjny, a jego wystąpienie musi być poprzedzone wystąpieniem odpowiedniego przełącznika (wspominanych w temacie), kolejno: -**d**, -**l**, -**o**, które można kojarzyć jako **d**ługość, **l**iczba i **o**utput. Nazwa programu oczywiście nie wymaga przełącznika. Przykładowe poprawne wywołanie programu w konsoli wygląda następująco:

```
„14. Maraton.exe” -d 1000 -l 30 -o wynik.txt
```

Kolejność z jaką podajemy parametry i ich przełączniki nie ma znaczenia, trzeba jednak zachować schemat, w którym nazwa programu jest na początku, a konkretny parametr jest poprzedzony jego przełącznikiem.

W przypadku podania niewłaściwych parametrów, lub niepodania jednego/więcej z nich, wyświetlony zostanie komunikat i program zwróci wartość 1 świadczącą o błędzie.

W programie nie implementowałem żadnych ograniczeń odnośnie zakresu podawanych wartości. Należy pamiętać, że długość trasy i ilość uczestników przechowywane są w zmiennej typu całkowitego, tak więc podanie wartości powyżej 2 147 483 647 spowoduje (testowane) zapętlone wykonywanie programu w nieskończoność przy wykorzystaniu maksymalnej dostępnej ilości zasobów fizycznych. Równie dobrze zamiast to pisać mógłbym dopisać prostego if’a sprawdzającego przekroczenie zakresu i zmienić typ zmiennych na unsigned long long int, co dałoby zakres do 18 446 744 073 709 551 615. Pozostańmy jednak przy wersji, że jest to program akademicki (dydaktyczny).

3.2. Komunikaty

Chociaż program powinien działać w pełni autonomicznie, istnieją sytuacje, kiedy wymagana jest dodatkowa komunikacja z użytkownikiem. Sytuacje te są awaryjne, zawsze związana z konkretnym błędem. Toteż pierwsza z nich (wspominana w podrozdziale powyżej) odnosi się do sytuacji, w której podany za mało parametrów uruchomieniowych, bądź też podamy je w niewłaściwej formie. Komunikat który ujrzy w takim wypadku użytkownik będzie następujący:

```
Niestety, podane parametry nie spełniają wymagań.
Parametry -d -l -o są OBLIGATORYJNE, wartości muszą następować po nich.
Przykładowe PRAWIDŁOWE wywołanie programu:
"14. Maraton.exe" -d 30 -l 10 -o statystyki.txt
```

Druga sytuacja, w której osoba obsługująca program powinna dowiedzieć się o pewnej niedogodności, to sytuacja, w której nie udało się utworzyć pliku wyjściowego o podanej nazwie (np. użytkownik nie miał odpowiednich uprawnień do tworzenia pliku w określonej lokalizacji). Komunikat wyświetlany w tym przypadku jest zwięzły:

```
Nie udało się uzyskać dostępu do pliku.
```

3.3. Format danych wyjściowych

Plik wyjściowy o podanej przez nas nazwie będzie znajdował się w katalogu w którym umiejscowiony jest program wykonywalny. Format danych wyjściowych jest następujący. Dla wszystkich zawodników, którym udało się ukończyć bieg, ukaże się ich pozycja na mecie, prędkość i typ (a w przypadku stonogi także ilość sprawnych nóg):

```
1. miejsce zajmuje:
Typ: Ślimak
Prędkość: 8

15. miejsce zajmuje:
Typ: Stonoga
Ilość nóg: 75
Prędkość: 2.1515
```

Z kolei jeśli zawodnik nie ukończył maratonu, wyświetlone zostaną informacje takie jak: położenie na jakim uczestnik został wyeliminowany z biegu, prędkość jaką w tej chwili posiadał i oczywiście typ zwierzątka. Tutaj przykład bardzo pechowego ślimaczka:

```
W BOJU POLEGŁ:
Typ: Ślimak
Położenie: 996
Prędkość: 4
```

4. Specyfikacja wewnętrzna

Ponieważ przy okazji analizy obiektowej projektu omówiłem już w jakimś stopniu podział na klasy i pola które w ich budowie wykorzystałem, w obecnym rozdziale nie będę już poświęcał temu aspektowi szczególnej wagi.

4.1. Konstruktory zawodników

Pierwszą rzeczą, którą mogę wyjaśnić są konstruktory, przy pisaniu których skorzystałem z listy inicjalizacyjnej. Konstruktor klasy Zawodnik jest standardowym konstruktorem parametrycznym:

```
Zawodnik::Zawodnik(double prędkość, double położenie, int status, char typ)
{
	this->prędkość = prędkość;
	this->położenie = położenie;
	this->status = status;
	this->typ = typ;
}
```

Żeby wykorzystać ten raz już napisany fragment kodu, w klasach potomnych wykorzystałem następującą formę konstruktora:

```
Ślimak::Ślimak() : Zawodnik::Zawodnik(5,0,0,'ś'), domek(true) {}
```

Forma ta to po prostu wykorzystanie do utworzenia nowego Ślimaka konstruktora klasy Zawodnik, przy nadaniu mu konkretnych, odpowiadających obiektowy klasy Ślimak parametrów początkowych, a więc prędkości i typu. Położenie i status każdego zawodnika w chwili tworzenia są takie same (równe 0) więc właściwie mogłem nadpisać je dodatkowo w konstruktorze klasy Zawodnik, jednak nie ma takiej potrzeby. Dodatkiem w powyższym konstruktorze jest ustawienie prywatnego pola domek ślimaka (przechowującego wartość logiczną) na reprezentację prawdy – iż domek ślimaka istnieje i ma się dobrze.

4.2. Destruktor

Może przy okazji konstruktora od razu poruszę temat destruktora. W wersji programu pokazanej na konsultacjach był problem z destruktorem wirtualnym.

```
virtual ~Zawodnik();
```

Powodem tego było zawarcie w destruktorze klasy bieg odwołania do metody zwalniającej pamięć. Właściwie samym sednem problemu było to, że program nie dochodził do wywoływania destruktorów. Początkowo sytuację tę zauważyłem wpisując instrukcję *cout* do destruktorów klas pochodnych, następnie krok dalej – do destruktora klasy Zawodnik, następnie Bieg, aż zauważyłem, że wywołanie instrukcji zwalniającej pamięć powinno być przeniesione na koniec konstruktora klasy Bieg – gdzie zawarty jest szkielet działania programu. Wywołanie następuje już po zamknięciu pliku:

```
	zwalnianiePamięci();
	zawodnicy.clear();
```

Natomiast samo ciało metody *zwalnianiePamięci()* jest dość szablonowe:

```
void Bieg::zwalnianiePamięci(void)
{
	Zawodnik * wskTemp;
	for (int i = 0; i < this->zawodnicy.size(); i++)
	{
		wskTemp = this->zawodnicy[i];
		delete wskTemp;
	}
}
```

Tworzę dodatkowy wskaźnik na obiekt klasy Zawodnik, który posłuży mi jako kontener do usuwania, do którego w pętli przypisywać będę wskaźnik na Zawodnika obecny w moim wektorze, a następnie usuwał go. Dodatkowo w konstruktorze Bieg zawarłem metodę *clear()*, która wg opisu powinna czyścić wektor. Nie zastąpi ona metody zwalniającej pamięć, bez niej właściwie wszystko działa tak samo, ale skoro ktoś ją stworzył, to warto było spróbować.

4.3. Funkcja główna main()

Zgodnie z zaleceniami funkcja główna programu miała być zminimalizowana, tak więc w moim programie zawarte w niej są tylko 3 instrukcje:

```
int main(int argc, char **argv)
{
	srand(time(NULL));
	Bieg maratonLeśny(argc, argv);
	return 0;
}
```

Od razu widać, że nie jest to postać minimalna – punkt startowy rand'a mógł się znaleźć równie dobrze w pliku Bieg.cpp, gdyż po zmianie formy rozwiązania na taką bez funkcji globalnych instrukcji z funkcji losujących korzystam już tylko w tym pliku. Właściwie nawet wartość zwracana nie jest konieczna:

```
void main(int argc, char **argv)
{
	Bieg maratonLeśny(argc, argv);
}
```

Powyższa implementacja została przetestowana i program w dalszym ciągu działa, jednak myślę że tak skrajne upraszczanie nie jest konieczne.

4.3. Właściwy szkielet programu – konstruktor klasy Bieg

Skoro funkcja *main()* został ograbiona z zawartości, gdzieś musiała się ona pojawić. Miejsce te, to ciało konstruktora klasy Bieg:

```
Bieg::Bieg(int argc, char **argv)
{
	Parametry uruchomieniowe(argc, argv);
	dodajZawodników(uruchomieniowe);
	biegTrwa = uruchomieniowe.getLiczbaUczestników();
```

```
	ofstream fout(uruchomieniowe.getNazwaPlikuWyjściowego(), ios::out);
	if (fout.good() == true)
	{
		while (biegTrwa)
		{
			wykonajKrok(uruchomieniowe, fout);
		}
		fout.close();
	}
	zwalnianiePamięci();
}
```

Jest to oczywiście uproszczona wersja na potrzeby wytłumaczenia zasady działania. Tak więc w pierwszym kroku odczytywane są parametry uruchomieniowe przekazane do konstruktora jeszcze z funkcji *main().* Odczyt następuje poprzez odpowiednie metody w klasie Parametry – a więc w konstruktorze tworzymy tylko obiekt tej klasy i przekazujemy mu to, co otrzymaliśmy. Kolejna instrukcja to wywołanie metody dodającej zawodników do wektora:

```
for (int i = 0; i < uruchomieniowe.getLiczbaUczestników(); i++)
	{
		zawodnicy.push_back(losujZawodnika());
	}
```

Jak widać przynajmniej niekiedy dbałem o to, by metody były proste. Do wektora dodawany jest losowy zawodnik (realizacja losowania na podstawie instrukcji *switch-case*), to właśnie w metodzie *losujZawodnika()* przydzielana jest pamięć – tworzeni są nowi zawodnicy:

```
	case 1:
		return new Stonoga();
		break;
```

Jednak wracając do konstruktora. Kolejna linia to dość niejasna instrukcja przypisania. Zmienna *biegTrwa* to moja zmienna przechowująca warunek wykonywania pętli kroku minutowego, przypisuję do niej na początku ilość zawodników biorących udział w biegu, a następnie odejmuję każdego, który skończył lub został wyeliminowany. Stąd wiem kiedy przerwać wykonywanie kroku. Sam krok następuje tuż po instrukcji przypisania, jest on oczywiście zagnieżdżony. Musi być zagnieżdżony w pętli, której warunek to oczywiści wspomniany *biegTrwa*, a dodatkowo zagnieżdżony w instrukcji wyboru sprawdzającej, czy udało się otworzyć plik wynikowy o nazwie podanej przez użytkownika. Tak, aby w sytuacji, gdy nie mamy gdzie wypisać wyniku pracy, w ogóle jej nie wykonywać. Jeśli udało się nam plik otworzyć, to po wykonaniu wszystkich kroków i wypisaniu odpowiednich danych wyjściowych należy plik zamknąć. Jest tu także wywołanie instrukcji zwalniającej pamięć opisane szerzej w 4.2..

4.4. Warunki

Omówiona została już funkcja zawierająca cały szkielet rozwiązania, a ani słowem nie zostały wspomniane najtrudniejsze aspekty projektu – rozwiązania warunków. Domyślać się więc można, że

najbardziej rozbudowaną metodą całego projektu będzie metoda stanowiąca kolejny krok zagłębienia – *wykonajKrok()*.

Przyznaję, że metoda ta jest w pewnym sensie moją porażką. 60 linijek kodu, w którym jedyną rzeczą mogącą coś sugerować są komentarze. Mimo wszystko naprawdę udało mi się w dobrym stylu rozwiązać choć kilka problemów związanych z tymi warunkami, toteż poświęcę im trochę więcej uwagi.

4.4.1. Warunek 1

Patrząc na analizę projektu, warunek realizować miał start uczestników falami.

```
for (int i = 0; i < zawodnicy.size() && grupyZawodników; i++, grupyZawodników+=10)
	{ . . . }
```

Całość operacji kroku wykonuję jeśli nie przeszedłem przez wszystkich zawodników w tym kroku i nie wyczerpałem limitu ilości grupy, który powiększa się co 10, tak więc w 1-wszym kroku będzie to 10, w drugim 20, itd.

4.4.2. Warunek 2

Warunek drugi wiązał się z przyrostem prędkości ślimaków proporcjonalnym do rozmiaru grupy.

```
if (zawodnicy[i]->getTyp() == 'ś')
zawodnicy[i]->setPredkość(5 + 0.1 * min(biegTrwa , grupyZawodników));
```

W każdym kroku przypisuję ślimakowi nową wartość prędkości, jest nią mniejsza z dwóch – ilości uczestników biegu którzy nie zostali wyeliminowani, lub ilości uczestników, którzy rozpoczęli już bieg. Początkowo zamiast biegTrwa była tutaj ilość zawodników z parametrów uruchomieniowych, jednak w takiej realizacji jeśli pod koniec biegu byłoby mniej zawodników ze względu na ich odpadanie z biegu – ślimak nadal miałby przyrost związany z niebiegnącymi już uczestnikami.

4.4.3. Warunek 3

Trójka, czyli warunek uszkodzenia nóg stonogi. To pierwsza implementacja o kłótliwej prostocie w moim wykonaniu. Konieczne jest podzielenie rozwiązania na 2 części – związanej z samodzielnym uszkodzeniem jednej nogi, a także z drugiej – związanej z uszkodzeniem 3 nóg przez nikczemnego żółwia. Warunki te mają odpowiadające im funkcje, stworzyłem nawet funkcje uszkadzającą podaną ilość nóg, bo proces ten powielał się w obu przypadkach. Mimo wszystko realizacja tych warunków jest rozbudowana. Dlatego też bez kopiowania kodu wytłumaczę tylko w prostych słowach jak działa.

- Sprawdzam, czy mam do czynienia ze stonogą.

- Jeśli tak, to sprawdzam, czy uszkodzi sobie sama nogę (osobna metoda).
- Jeśli tak, to odejmuję nogę od puli nóg (osobna metoda).
- W tej samej metodzie sprawdzam, czy uszkodzone są już wszystkie nogi tej stonogi.
- Jeśli tak – ustawiam status na -1, dekrementuję *biegTrwa*, inkrementuję *ilośćPoległych*.
- Jeśli nie – po prostu zmieniam prędkość na mniejszą – uwzględniającą uszkodzone nogi.

Następnie ponawiam całą tą operację po raz drugi, zmieniając metodą do sprawdzania, czy stonoga sama uszkodzi sobie nogę, na metodą sprawdzającą, czy wyręczy ją w tym nikczemny żółw.

4.4.4. Warunek 4

Ostatni warunek (tak, ostatni, bo warunek 5 rozwiązany został już w konstruktorze Bieg) dotyczy Ślimaka, a właściwie jego domku… i Żółwia – ponownie w roli oprawcy. W tej metodzie pomijając ten sam schemat związany z eliminacją zawodnika znany z warunku 3, ciekawsze jest rozwiązanie problemu spotkania się Żółwi i Ślimaka w jednym miejscu. Solucja jest następująca: jeśli wykonuję krok jako ślimak, to przekazuję jego położenie do metody spotkanieBezŚwiadków() i iterując się po tablicy sprawdzam, czy takie same położenie mają tylko 4 zwierzaki, w tym 3 żółwie (bo 1 to mój ślimak):

```
if(ilośćZawodników==4 && ilośćŻółwi==3)
	return true;
```

5. Testowanie

5.1. Dane testowe – uzasadnienie

Program był wielokrotnie testowany dla niewielkich wartości do 2 000, co prezentowane było również na konsultacjach. Sprawdzone zostały wszystkie możliwe kombinacje podawania parametrów wejściowych, ich braku lub też braku uprawnień do zapisywania w podanej lokacji. Na potrzeby tego punktu wykonałem test także dla większych wartości –d 10 000, -l 100, a z powodów opisywanych poniżej również dla wartości o połowę mniejszych.

5.2. Wyniki

Wszystkie możliwości warunkowe wydają się działać prawidłowo, nie zauważyłem jeszcze żadnych nieprawidłowości z nimi związanych. Komunikaty wyświetlają się prawidłowo, różne kombinacje parametrów wejściowych dają te same – poprawne skutki.

Odnośnie testów dla większych próbek – pomijając już test powyżej zakresu int, także dla 10 000 – 100 nie otrzymałem wyników. Dla wartości o połowę mniejszych wynik pojawił się w pliku po niespełna 2s, natomiast dla pierwszej próbki po minucie wyników w dalszym ciągu nie było. Powodów

takiego stanu rzeczy doszukuję się w warunkach sprzętowych – Celeron G550 nie pierwszy raz ogranicza mój projekt.

6. Wnioski

Program zaakceptowany został na konsultacjach dnia 18.01.2017 (druga osoba tego dnia – drugi projekt Maratonu). Dla prawidłowych danych wejściowych generował on prawidłowe dane na wyjściu, stąd też konkluzja, że program działa.

W trakcie konsultacji uwaga zwrócona została na 2 główne problemy – nierozsądne wypisywanie danych do pliku (przypadek stonogi z nogami w ilości -2), a także wycieki pamięci.

Tak więc po kolei: wypisywanie do pliku zostało zmodernizowane wg wytycznych z tematu. Ni zastosowałem wspominanej instrukcji warunkowej sprawdzającej czy stonoga utraciła wszystkie nogi. Skorzystałem z faktu, iż ilość nóg miała być wypisywana tylko w okolicznościach, gdy stonodze uda się dotrzeć do mety. Takim sposobem chociaż program w dalszym ciągu wyklucza z biegu stonogi mające mniej niż 1 sprawną nogę, to nie pojawi się już niemądry wpis o stonodze z ujemną ilością nóg.

Wycieki pamięci to sprawa już nieco bardziej dramatyczna. Na konsultacjach dopisany został konstruktor wirtualny, którego oczywiście brakowało w pierwotnym rozwiązaniu, ukróciło to istnienie większości wycieków, pozostawiając tylko 2 o niezidentyfikowanym położeniu i dość zagadkowym rozmiarze:

```
{400} normal block at 0x011C04A0, 168 bytes long.
 Data: <0ę   ę   ç  (é  > 30 EA 1B 01 88 EA 1B 01 18 E7 1B 01 28 E9 1B 01
{339} normal block at 0x011BD9D0, 8 bytes long.
 Data: <Đůł     > D0 F9 B3 00 00 00 00 00
```

Nie rozwodząc się zbyt mocno nad metodami których próbowałem by zlokalizować powyższe, powiem tylko że zamienienie testowej linijki:

```
_CrtDumpMemoryLeaks();
```

na poniższą wersję:

```
_CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
```

Dało w pozytywny efekt.
Nigdy nie sądziłem, że będę szukał wycieków pamięci w kodzie do sprawdzania wycieków pamięci.