Podstawy programowania
Materiały dydaktyczne do laboratorium

dr inż. K. M. Ocetkiewicz

3 listopada 2016

Zadania domowe

1 Materiały wstępne

Zapoznaj się z następującymi materiałami:

- fprintf, fscanf, fgets, fputs, fgetc, fputc, fopen, fclose, fread, fwrite, fflush, fseek, ftell
  (http://www.cplusplus.com/reference/cstdio/)
- clock (http://www.cplusplus.com/reference/ctime/)
- cin (http://www.cplusplus.com/reference/iostream/cin/)
- cout (http://www.cplusplus.com/reference/iostream/cout/)
- cerr (http://www.cplusplus.com/reference/iostream/cerr/)
- fstream (http://www.cplusplus.com/reference/fstream/fstream/)
- obsługa konsoli w systemie Windows
  (http://dream.cs.bath.ac.uk/software/sndan/use_console.html,
  http://en.wikipedia.org/wiki/Tab_completion)

2 Pytania kontrolne do materiałów wstępnych

- Jaka jest różnica pomiędzy wartością zwracaną przez printf a scanf?
- Do czego służy napis formatujący "%n"?
- Jak wypisać na ekran szesnastkowo?
- Co (i dlaczego w przypadku 4) zobaczymy wołając printf(FMT, 12.3), gdzie FMT to:

1. “%6.2lf”
2. “%06.2lf”
3. “%-6.2lf”
4. “%6d”

- Co zwraca fread a co fwrite?
- O ile przez sekundę zwiększy się licznik, którego wartość zwraca clock()?
- Jak automatycznie uzupełnić nazwę pliku częściowo wpisaną w konsoli?
- Gdzie znajduje się skompilowany plik wykonywalny Twojego programu, i jak ustawić ten katalog na bieżący w konsoli?

3 Konsola

Najbardziej rozpowszechnioną metodą komunikacji z komputerem jest konsola. Konsola składa się z klawiatury, poprzez którą użytkownik wydaje polecenia oraz ekranu na którym prezentowane są odpowiedzi komputera.

Naciskanie klawiszy klawiatury powoduje wstawienie odpowiednich liczb, najczęściej kodów ASCII wybranych znaków, do bufora klawiatury. Liczby te (pamiętaj, że w języku C liczba jest równoważna znakowi, np. 'A' == 65) oczekują w kolejce, w kolejności wprowadzenia, na odebranie przez program. Są one pamiętane nawet wtedy, gdy program nie sygnalizuje gotowości odczytywania klawiatury.

Zadanie 1. Uruchom następujący program i wpisz
aaa<Backspace><Backspace><Backspace>xyz<Enter>:

```
#include<stdio.h>
int main() {
  int a, b;
  a = getchar();
  printf("a=\'%c\'\n", a);
  b = getchar();
  printf("b=\'%c\'\n", b);
  return 0;
  };
```

Zaobserwowaliśmy tu dwa efekty, oba mające związek z buforowaniem wejścia. Po pierwsze, programy mają domyślnie ustawione buforowanie wierszy. Oznacza to, że wpisane znaki są przekazywane do programu dopiero po naciśnięciu klawisza Enter. Dodatkowo, dopóki nie naciśniemy tego klawisza, możemy „edytować” nasze wejście (używając np. Backspace). Po naciśnięciu klawisza Enter do programu trafia tylko wersja ostateczna. Możemy zmienić to zachowanie wybierając inny sposób buforowania, wiąże się to jednak z wydajnością naszego programu (generalnie im większe buforowanie, tym szybciej działają operacje wejścia/wyjścia).

Drugim efektem jest wspomniane wcześniej kolejkowanie znaków. Program zatrzymał się na pierwszym getchar, a my wpisaliśmy (ostatecznie) cztery znaki: 'x', 'y', 'z', '\n'. Dopiero odczyt znaku '\n' spowodował dalsze wykonanie programu i pierwsze wywołanie getchar odczytało znak 'x'. Drugie wywołanie getchar nie zatrzymało programu, gdyż

w kolejce czekały już kolejne znaki. Funkcja getchar pobrała i zwróciła pierwszy z nich (czyli 'y'). Gdy program zakończył się, pozostałe w kolejce znaki zostały zapomniane.

Zachowanie ekranu konsoli wywodzi się od drukarek mozaikowych (zanim pojawiły się monitory, komputery prezentowały wyniki drukując je). Ekran pamięta miejsce, które nazywamy pozycją kursora, w którym będą pojawiały się kolejne znaki. Wypisanie znaku powoduje przesunięcie kursora o jeden znak w prawo, zatem wypisanie na ekran kolejno znaków 'A', 'B', 'C' spowoduje, że zobaczymy na ekranie napis 'ABC'. Gdy kursor dojdzie do prawej krawędzi ekranu, wraca on do lewego brzegu, przesuwając się jednocześnie o jeden wiersz w dół. Gdy skończy się miejsce na ekranie, czyli gdy kursor będzie w ostatnim wierszu i będzie musiał przesunąć się do kolejnego wiersza, cała zawartość ekranu zostanie przesunięta o jeden wiersz w górę i kursor trafi na początek ostatniego wiersza ekranu (teraz już pustego). Niektóre znaki wyświetlane na ekranie mają specjalne znaczenie. Najistotniejsze z nich to:
'\n' – LF (Line Feed), nowa linia – powoduje przejście kursora na początek kolejnej linii (wraz z ewentualnym przesunięciem ekranu gdy kursor jest w ostatniej linii ekranu[1],
'\r' – CR (Carriage Return[2]), "powrót karetki" — powoduje przesunięcie kursora na początek bieżącej linii,
'\t' – tabulacja pozioma – przesunięcie kursora do następnej kolumny o numerze będącym wielokrotnością liczby 8 (numerując kolumny od 0); przydatne przy wyświetlaniu tabel,
'\b' – backspace – przesunięcie kursora o jeden znak w lewo, o ile kursor nie znajduje się przy lewym brzegu ekranu; w przypadku drukarek znak ten umożliwiał łączenie znaków (np. aby wypisać znak ą nie mając polskiej czcionki można było wydrukować znak a, cofnąć kursor i wydrukować przecinek), w przypadku ekranu nowe znaki zastępują stare (czyli zamiast ą dostaniemy sam przecinek).

Zadanie 2. Przeanalizuj znaki wysyłane na ekran przez poniższy program i zastanów się, co zobaczysz na ekranie. Uruchom program i zestaw swoje oczekiwania z wynikiem zaobserwowanym na ekranie.

```
#include<stdio.h>
#include<time.h>

// oczekiwanie na reakcję użytkownika
void CzekajNaEnter() {
  printf("nacisnij Enter\n");
  getchar();
  };

// zatrzymanie programu na czas sekund
void Czekaj(double czas) {
  int t;
  t = clock();
  while((clock() - t) < czas * CLOCKS_PER_SEC) {
    /* pass */
    };
```

[1] W przypadku drukarek LF powoduje tylko przesunięcie papieru o jeden wiersz, bez powrotu głowicy drukującej do lewego brzegu. Zatem "ekranowe" LF odpowiada "drukarkowemu" LF + CR.

[2] http://en.wikipedia.org/wiki/Carriage_return

```
  };

int main() {
  printf("0123456789 -- tabulator\n");
  printf("\t*\n");
  printf("0\t*\n");
  printf("01\t*\n");
  printf("012\t*\n");
  printf("0123\t*\n");
  printf("01234\t*\n");
  printf("012345\t*\n");
  printf("0123456\t*\n");
  printf("01234567\t*\n");
  printf("012345678\t*\n");
  printf("0123456789\t*\n");
  printf("0123456789A\t*\n");
  CzekajNaEnter();

  char *imiona[] = { "Ala", "Ela", "Ola", "Ula",
        "Bolek", "Lolek", "Reksio" };
  int wyniki[] = { 10, 40, 20, 30, 20, 0, 40 };
  printf("+-------+-------+-------+\n");
  printf("|imie\t|punkty\t|[%%]\t|\n");
  printf("+-------+-------+-------+\n");
  for(int i = 0; i < sizeof(imiona)/sizeof(imiona[0]); ++i) {
    printf("|%s\t|%d\t|%d\t|\n",
           imiona[i],
           wyniki[i],
           wyniki[i] * 5 / 2);
    };
  printf("+-------+-------+-------+\n");
  CzekajNaEnter();

  printf("stary napis, nowa linia na koncu\n");
  printf("nowy napis\n");
  printf("----------\n");
  printf("stary napis, powrot karetki na koncu\r");
  printf("nowy napis\n");
  printf("----------\n");
  CzekajNaEnter();

  printf("obliczanie: ");
  for(int i = 0; i <= 100; i++) {
    Czekaj(0.1);
    printf("%3d%%\b\b\b\b", i);
    };
  printf("\n");
  CzekajNaEnter();
```

```
for(int i = 0; i <= 100; i++) {
  Czekaj(0.1);
  printf("obliczanie drugie: %3d%%\r", i);
  };
printf("\n");

return 0;
};
```

Odpowiedz na następujące pytania:

1. Dlaczego wynik procentowy obliczany jest w sposób `dane[i].wynik * 5 / 2` zamiast `dane[i].wynik * 2.5`?
2. Co oznacza %3d a co %% w napisie formatującym dla funkcji printf?
3. Dlaczego w obliczaniu pierwszym muszą być dokładnie cztery znaki Backspace? Co się stanie (i dlaczego), gdy zmniejszymy ich liczbę do trzech, a co gdy zwiększymy ją do pięciu?
4. Dlaczego w pętli for w obliczaniu pierwszym wypisujemy tylko liczbę a w obliczaniu drugim cały napis?
5. Dlaczego funkcje z rodziny prinf (zwłaszcza fprintf) i scanf uważane są za niebezpieczne[3]?

Zadanie 3. Napisz powyższy program korzystając z cin/cout zamiast printf/getchar.

Z ekranem wiąże się także pojęcie echa. Echo powoduje wyświetlanie na ekranie wpisywanych z klawiatury znaków. Należy pamiętać, że echo nie jest częścią wyjścia programu, a jedynie efektem "wizualnym" naciskania klawiszy. Zaobserwujemy to w dalszej części instrukcji.

4 Pliki

Plik jest sekwencją bajtów o określonej długości. Z plikami skojarzone są dodatkowe dane, takie jak nazwa, data utworzenia, prawa dostępu itp. Znaczenie zawartości pliku jest umowne i jest określone przez jego format. Na przykład format plików tekstowych określa, że kolejne bajty odpowiadają kolejnym znakom w linii, zaś para kolejnych bajtów o wartościach 13 i 10 oznaczają zakończenie bieżącej i rozpoczęcie nowej linii. Z kolei format plików PDF precyzuje między innymi, że plik powinien rozpoczynać się od bajtów o wartościach 37, 80, 68, 70, 45 (co odpowiada napisowi %PDF-) po których zakodowana jest wersja wykorzystanego formatu PDF. Nic jednak nie stoi na przeszkodzie, aby otworzyć plik w formacie PDF w notatniku (który jest narzędziem do edycji plików w formacie tekstowym). Zostanie on zinterpretowany jako dokument tekstowy więc nie będzie czytelny z naszego punktu widzenia, jednak dla notatnika będzie to poprawny plik. Otworzenie pliku tekstowego nie powiedzie się w czytniku plików PDF tylko dlatego, że

[3]Czytanka dla zainteresowanych: http://www.cs.cornell.edu/Courses/cs513/2005fa/paper.format-bug-analysis.pdf

analizuje on początkowe bajty pliku i jeżeli nie zgadzają się one z formatem, zgłasza błąd. Należy zapamiętać, że znaczenie wartości bajtów w pliku jest czysto umowne.

Z punktu widzenia języka C/C++ mamy dostęp do zawartości pliku na dwa sposoby: sformatowany i surowy. Sposób sformatowany (m.in. funkcje fprintf, fscanf, fgets) zakładają, że plik zawiera tekst. Wypisując do pliku liczbę 12 funkcją fprintf umieścimy w nim dwa bajty o wartościach 49 i 50 (są to wartości znaków '1' i '2' w ASCII). Z kolei wypisując liczbę 12345678 w pliku znajdzie się 8 bajtów o wartościach od 49 do 56. Podobnie, odczytując liczbę z pliku funkcją fscanf, będzie ona pobierała bajty, dopóki ich wartości będą należały do przedziału '0'-'9'.

Dostęp surowy ignoruje interpretację bajtów. Funkcje takie jak fgetc, fputc, fread, fwrite operują tylko na wartościach bajtów. fgetc odczyta jeden bajt, niezależnie od jego wartości. Wypisując do pliku liczbę całkowitą typu int funkcją fwrite:

```
fwrite(&liczba, 1, sizeof(int), plik);
```

zapiszemy do pliku 4 bajty (o ile rozmiar liczby całkowitej jest równy 4) niezależnie od wartości zmiennej liczba – jeżeli wartością tą było 12, zapiszemy bajty 12, 0, 0, 0, jeżeli wartością było 12345678 zapiszemy bajty 78, 97, 188, 0, gdyż w ten sposób liczby te są reprezentowane w pamięci. Do nas zatem należy obowiązek pamiętania, co i w jakiej postaci zapisujemy i odczytujemy z pliku.

Dostęp do pliku mamy poprzez uchwyt (wartość typu FILE * lub zmienna typu fstream). Jest to obiekt reprezentujący plik. Uchwyt wiążemy z plikiem otwierając go (funkcja fopen lub metoda open obiektu fstream). Jest to jedyne miejsce, w którym podajemy nazwę interesującego nas pliku. Pamiętaj, podając ścieżkę, że odwrócone ukośniki należy podwoić w napisach języka C, czyli "c:\\windows\\system32" zamiast "c:\windows\system32". Jeżeli chcemy napisać program działający zarówno w systemie Linux jak i Windows, lepiej używać ścieżek względnych (nie podając dysku czy katalogu głównego) i rozdzielać elementy ścieżki zwykłym ukośnikiem. Zatem piszemy "../dane/plik.txt" zamiast "/usr/bin/program/dane/plik.txt" czy "c:/Program Files/program/dane/plik.txt".

Warto pamiętać o operacji "spłukiwania". Dostęp do plików jest buforowany (w celu zwiększenia wydajności). Wadą buforowania jest to, że nie mamy pewności, kiedy dane, które zapisaliśmy do pliku, rzeczywiście się w nim znajdą (system operacyjny może zdecydować odłożyć zapis w czasie, aby np. wykonać go razem z następnym zapisem). Powoduje to, że gdy wystąpi awaria (np. wyjątek w programie) dane które oczekują w buforze (funkcja zapisująca się zakończyła, ale system operacyjny nie zapisał ich jeszcze na dysk) zostaną utracone.

Zadanie 4. Spójrz na następujący program.

```
#include<stdio.h>

int main() {
  FILE *pl;
  pl = fopen("plik.txt", "wb");
  fprintf(pl, "napis");
  getchar();
  fclose(pl);
  return 0;
  };
```

Uruchom go i naciśnij klawisz Enter. Obejrzyj zawartość pliku plik.txt. Teraz uruchom program ponownie, zamiast Enter naciskając Ctrl+C, co wymusi natychmiastowe zakończenie programu i zachowanie podobne do tego przy wystąpieniu wyjątku. Obejrzyj zawartość pliku plik.txt.

Możemy wymusić zapisanie zawartości buforu do pliku wołając funkcję fflush lub metodę flush strumienia w C++.

Zadanie 5. Dodaj linię

```
fflush(pl);
```

do powyższego programu przed wywołaniem funkcji getchar() i zaobserwuj jego działanie w obu przypadkach (Enter i Ctrl+C).

Dlaczego dane nie są "spłukiwane" po każdym zapisie? Dostęp do dysku jest bardzo wolny, zatem takie działanie drastycznie obniżyłoby szybkość zapisu na dysku. Dlatego też nie jest to robione automatycznie, lecz, jeżeli tego oczekujemy, możemy po każdym zapisie samodzielnie wywołać fflush. Uwaga:

```
cout << endl;
jest równoznaczne z:
cout << "\n";
cout.flush();
```

zatem endl jest nieco wolniejsze od wypisywania samego znaku nowej linii (ale różnica ta będzie zauważana jedynie w przypadku wypisywania dużej liczby linii).

Wejście programu ("klawiatura", standardowe wejście) oraz wyjście ("ekran", standardowe wyjście) są w rzeczywistości także "plikami". W języku C istnieją obiekty typu FILE * (czyli uchwty plików) o nazwie stdin i stdout, które reprezentują klawiaturę i ekran. Korzysta się z nich tak samo jak ze zwykłych plików (poza otwieraniem — otwarte są od początku działania programu, dodatkowo nie należy "spłukiwać" wejścia — stdin). Czyli np.

```
fprintf(stdout, „napis: %s", napis)
```

jest równoważne

```
printf(„napis: %s", napis)
```

a

```
fscanf(stdin, „%d", &liczba)
```

to to samo co

```
scanf(„%d", &liczba)
```

W każdym programie istnieje także dodatkowy "ekran" – wyjście błędów, reprezentowany przez obiekt stderr w języku C i cerr w języku C++. Domyślnie jest on połączony ze standardowym wyjściem, więc wypisując cokolwiek na ekranie błędów zobaczymy to na zwykłym ekranie, istnieje jednak możliwość ich rozłączenia. Z założenia jest on przeznaczony na komunikaty o błędach (w odróżnieniu od zwykłych komunikatów programu, pisząc program, informacje np. o postępie obliczeń będziemy raczej wyświetlali na standardowym wyjściu, zaś komunikaty związane z debugowaniem na wyjściu błędów).

Zadanie 6. Skompiluj następujący program.

```
#include<iostream>
using namespace std;

int main() {
  char text1[128], text2[128];
  cin >> text1;
  cin >> text2;
  cout << "napis␣1␣na␣standardowym␣wyjsciu:␣" << text1 << endl;
  cerr << "napis␣2␣na␣wyjsciu␣bledow:␣" << text2 << endl;
  return 0;
  };
```

Utwórz plik wejscie.txt z zawartością:

```
napis1
napis2
```

zapisz go w katalogu, w którym znajduje się skompilowany program i uruchom go wydając polecenie:

```
program.exe <wejscie.txt >wyjscie.txt 2>bledy.txt
```

(program.exe jest tu nazwą naszego skompilowanego programu). Obejrzyj zawartość pliku wyjscie.txt i bledy.txt.

Podanie jako argumentu uruchomienia programu pliku poprzedzonego znakiem < powoduje ustawienie tego pliku (w naszym przypadku wejscie.txt) jako wejście programu. Jeżeli to zrobimy, program będzie czytał dane ze wskazanego pliku zamiast z klawiatury. Użycie znaku > powoduje przekierowanie standardowego wyjścia do wskazanego pliku (jeżeli użyjemy >> zamiast > nowe dane zostaną dopisane na koniec pliku, zamiast zastąpienia go). Wreszcie 2> powoduje przekierowanie wyjścia błędów do wskazanego pliku.

Zadanie 7. Uruchom powyższy program ze wszystkimi kombinacjami przekierowania (przekierowując wejście, nie przekierowując wejścia, podobnie z wyjściem i błędami). Gdy wejście nie jest przekierowane, a wyjście tak, zaobserwuj gdzie pojawia się echo wprowadzanych znaków.

Zadanie 8. Przepisz powyższy program korzystając z funkcji fscanf i fprintf. Do wypisania na ekran błędów użyj `fprintf(stderr, ...)`.

Proszę pamiętać, że jedynie ekran interpretuje białe znaki, takie jak tabulacja czy Backspace. Przekierowując wyjście do pliku, zostaną w nim zapisane dokładnie te bajty, które wyślemy na wyjście.

Zadanie 9. Uruchom następujący program bez przekierowania wyjścia a następnie przekieruj jego wyjście do pliku. Porównaj zawartość ekranu z zawartością pliku.

```
#include<iostream>
using namespace std;

int main() {
  cout << "abc" << "\b\b\b" << "def\n";
  return 0;
```

```
};
```

Zadanie 10. Napisz program kodujący base64 (https://pl.wikipedia.org/wiki/Base64) zadany plik.

Zadanie 11. Napisz program dekodujący base64 (https://pl.wikipedia.org/wiki/Base64) zadany plik.

Zadanie 12. Napisz program kodujący i dekodujący (w zależności od argumentów linii poleceń) zadany plik szyfrem Gaderypoluki (zob. https://pl.wikipedia.org/wiki/Gaderypoluki). Klucz powinien być jednym z argumentów programu. Np. wywołanie:
`program.exe szyfruj plik.txt GADERYPOLUKI`
przy zawartości pliku plik.txt `Ala ma kota` powinno zmienić tę zawartość na `Gug mg iptg`. Wywołanie:
`program.exe odszyfruj plik.txt GADERYPOLUKI`
powinno odwrócić tę operację.

Zadanie 13. Napisz program pobierający cztery argumenty z linii poleceń: nazwę pliku, liczbę, kolor cyfr, kolor tła. Program powinien tworzyć bitmapę prezentującą daną liczbę. Każda cyfra powinna zajmować 8x16 pikseli. Rozmiar bitmapy powinien zależeć od długości liczby: liczba 1 powinna generować bitmapę rozmiaru 8x16, zaś liczba 54321 bitmapę rozmiaru 40x16. Kolory proszę pobierać jako liczby szesnastkowe. Przy zapisie bitmapy należy skorzystać z 32 bitowego kodowania kolorów (zob. https://en.wikipedia.org/wiki/BMP_file_format). Przykładowe wywołanie:
`program.exe liczba.bmp 54321 ff0000 00ff00`

Zadanie 14*. Napisz program wyświetlający listę plików z archiwum w formacie ZIP (zob. https://en.wikipedia.org/wiki/Zip_%28file_format%29)

5 Czytanie, dopóki są dane

Czasami może pojawić się konieczność odczytania wszystkich danych wejściowych, nie znając ich rozmiaru (np. mamy wczytać wszystkie liczby, nie znając ich liczby). Jesteśmy w stanie to zrobić — gdy dojdziemy do końca danych, dostaniemy informację o braku kolejnych.

Funkcje scanf i fscanf informują nas poprzez wartość zwróconą, ile zmiennych udało im się przeczytać. Zatem jeżeli wiemy, że plik zawiera liczby, możemy je wszystkie przeczytać następującym fragmentem kodu:

```
while(fscanf(plik, „%d", &liczba) > 0) {
  // robimy coś z liczbą
  };
```

Jeżeli dane w pliku się skończą, fscanf zwróci wartość mniejszą od 1 i pętla while się zakończy. W przypadku korzystania z cin możemy użyć następującego fragmentu:

```
while(cin >> liczba) {
  // robimy coś z liczbą
  };
```

który zachowa się w taki sam sposób.

Inne funkcje także informują nas o końcu pliku. fgetc zwraca specjalną wartość EOF, gdy dotarliśmy do końca pliku, zaś fgets zwraca w takiej sytuacji NULL.

Wprowadzając dane z klawiatury także możemy zasygnalizować ich zakończenie. Służy do tego Ctrl+Z w systemach Windows i Ctrl+D w systemach z rodziny Linux. Należy jednak pamiętać o buforowaniu wierszowym (może wystąpić konieczność naciśnięcia jeszcze klawisza Enter, aby dane z bufora trafiły do programu). Co więcej, o ile w pliku koniec danych jest ostateczny i nieodwołalny, to nic nie zabrania użytkownikowi wpisywania danych z klawiatury po naciśnięciu Ctrl+Z / Ctrl+D, a nasz program będzie w stanie je odczytać. Nie powinniśmy jednak oczekiwać na dane po zaobserwowaniu końca pliku, gdyż takie skonstruowanie programu uniemożliwi wprowadzanie danych przez przekierowanie wejścia.

Zadanie 15. Napisz program, który wczytuje (dopóki są dostepne) parametry trójmianu kwadratowego ($ax^2 + bx + c = 0$) i dla każdej trójki wyświetla jego rozwiązanie.

Zadanie 16. Napisz program, który wczytuje z klawiatury kolejne linie i wyświetla na ekran tylko te, które zawierają napis podany w linii poleceń. Np. dla `program.exe kot` i wpisaniu:

```
Ala ma kota.
Kot ma Ale.
Kot ma kota.
Ala ma psa.
```

na ekran powinno zostać wypisane ("Kot" to nie to samo co "kot" — rozróżniamy wielkość liter):

```
Ala ma kota.
Kot ma kota.
```

6 Znaki nowej linii

Obsługując dane tekstowe należy zwrócić uwagę na sposób rozdzielania linii. W systemach rodziny Windows służy do tego para znaków CR, LF. Z kolei systemy z rodziny Linux korzystają tylko ze znaku LF. Powoduje to, że plik tekstowy zapisany w notatniku i otworzony pod Linuxem ma „dziwne" znaki na końcach linii (są to znaki CR), zaś plik zapisany pod Linuxem i otworzony w notatniku ma postać jednej linii (dla notatnika sam znak LF nie rozdziela linii). My, pisząc programy, powinniśmy pisać je tak, aby poradziły sobie z obydwoma sposobami rozdzielania linii, zwłaszcza, że nie jest to trudne. Funkcje scanf oraz strumień cin przeskakują wszystkie białe znaki, więc z ich punktu widzenia nie ma znaczenia sposób rozdzielania linii. Używając fgets wystarczy sprawdzić, czy ostatnim lub przedostatnim znakiem odczytanego napisu jest CR i jeżeli tak, usunąć go, zaś cin.getline usuwa znaki kończące linie, więc nie musimy już nic robić.

Dopiero gdy próbujemy wczytywać wejście znak po znaku, musimy zwrócić uwagę na sposób rozdzielania linii. Najczęściej jednak wszystko, co musimy zrobić to traktować znaki LF jako rozdzielające linie, a znaki CR świadomie ignorować, czyli np.:

```
int c;
while((c = fgetc(plik)) != EOF) {
  if(c == '\r') continue; // ignorowanie CR
  else if(c == '\n') {
```

```
    // nowa linia
  } else {
    // kolejny znak w tej samej linii
    };
  };
```

Zadanie 17. Zaimplementuj Unixowe polecenie wc (https://en.wikipedia.org/wiki/Wc_%28Unix%29). Nie trzeba implementować liczenia znaków. Program powinien działać zarówno na plikach (np. `wc -l plik.txt`) jak i w trybie interaktywnym (gdy nie podamy nazwy pliku dane są wczytywane z klawiatury (np. `wc -l`).

7 Uwagi dotyczące stylu

Jeżeli tylko masz możliwość, nie używaj kodów ASCII w miejsce znaków. Po pierwsze, jest to mniej czytelne, spójrz na poniższy kod:

```
if(znak == 110) Zatwierdz();
else if(znak == 121) Anuluj();
```

Czy jest on poprawny? Porównaj go z kodem:

```
if(znak == 'n') Zatwierdz();
else if(znak == 'y') Anuluj();
```

Tu od razu widać błąd – znak 'n' powinien anulować zaś 'y' zatwierdzać. Po drugie, nie powinniśmy oczekiwać od czytelnika kodu znajomości tabeli ASCII. Dodatkowo, jeżeli okazałoby się, że trzeba przenieść kod na maszynę korzystającą z innego kodowania znaków (jest to jednak ekstremalnie mało prawdopodobne), kod korzystający ze znaków (czyli drugi z powyższych) pozostanie poprawny, zaś ten używający ich kodów poprawny nie będzie (np. w kodowaniu EBCDIC znak 'A' ma wartość 193).

Kod korzystający z liczb zamiast znaków nie jest bardziej „hakerski". Jedynymi osobami, na których może zrobić to wrażenie, to osoby zupełnie nie potrafiące programować. Jednak nawet dla programisty z niewielkim doświadczeniem użycie liczb zamiast znaków jest sygnałem, że autor kodu jest neofitą próbującym sprawiać wrażenie profesjonalisty.

Proszę pamiętać, że powyższe uwagi dotyczą języków C i C++. W innych językach może nie być równoważności pomiędzy liczbą a znakiem (może nawet nie być w nich typu znakowego) i użycie liczb może być uzasadnione gdy np. język gwarantuje nam korzystanie z jednego kodowania znaków a skorzystanie z liczb jest nieco szybsze niż z napisów. W takich sytuacjach najlepszym rozwiązaniem może być skorzystanie ze stałych:

```
var ZNAK_Y = „y".charCodeAt(0); // kod w języku JavaScript
var ZNAK_N = „n".charCodeAt(0);
...
if(znak == ZNAK_Y) Zatwierdz();
else if(znak == ZNAK_N) Anuluj();
```

Zwróć uwagę na komentarz /* pass */ w funkcji Czekaj. Równie dobrze pętla mogłaby mieć postać:

```
while((clock() - t) < czas * CLOCKS_PER_SEC);
```

Gdzie leży różnica? W postaci z komentarzem mówimy wyraźnie, że zawartością while jest pusta akcja (oczywiście można dać inny komentarz, np. „nic nie robie", czy „do nothing"). Drugi przypadek (średnik bezpośrednio po while) powinien wyglądać podejrzanie – sami nie powinniśmy umieszczać takiej konstrukcji w kodzie, a gdy ją zobaczymy powinniśmy wyjątkowo uważać. Porównaj kody:

```
while(Warunek());
{
   Obliczenia();
}
```

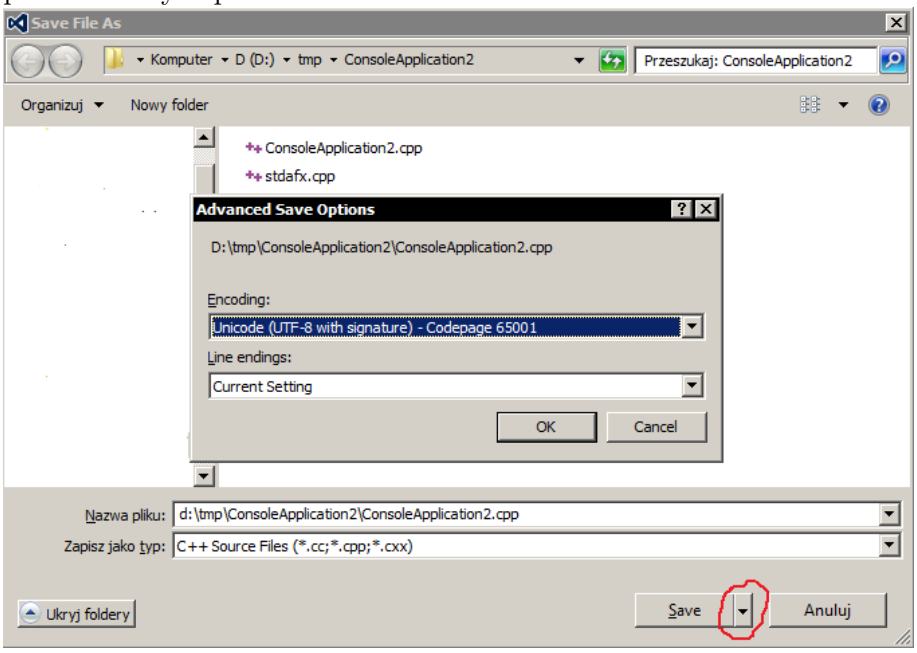
```
while(Warunek())
{
   Obliczenia();
}
```

Oba są poprawne składniowo i kompilują się bez błędów jednak działanie ich jest zupełnie różne. Dodatkowo średnik jest niewielkim znakiem i stosunkowo trudno go dostrzec. W takich sytuacjach konsekwentne trzymanie się stylu ułatwi nam znajdowanie błędów – łatwiej dostrzeżemy średnik gdy jest on w miejscu gdzie nie powinno go być (gdy nigdy nie stawiamy średnika za nawiasem kończącym while czy for) niż gdy jest on w miejscu gdzie może się znaleźć (kiedy nie stosujemy tej zasady).

8 (*) Unicode - polskie litery i inne znaki spoza ASCII

Aby skorzystać z kodowania Unicode w przypadku systemu Windows i Visual Studio należy:

- ustawić skalowalny krój pisma (np. Lucida Console, Consolas) zamiast rastrowego (Terminal) w oknie konsoli,
- plik źródłowy zapisać z kodowaniem UTF-8



- korzystać z szerokich napisów (wchar_t text[] = L"napis"),
- wywołać na początku programu

```
_setmode(_fileno(stdout), _O_WTEXT);
_setmode(_fileno(stdin), _O_WTEXT);
```

- błędy w implementacji biblioteki standardowej w VS powodują, że powyższe wywołania uniemożliwiają skorzystanie ze strumieni C++ (cin/cout/wcin/wcout) – użycie strumienia skutkuje wystąpieniem błędu wykonania
- korzystać z szerokich wersji funkcji we/wy: wprintf zamiast printf, wscanf zamiast scanf; format string dla szerokich napisów to %ls

Np. (zob. plik L7_uwin.cpp).

```
#include<fcntl.h>
#include<io.h>
#include<stdio.h>
using namespace std;

int main() {
  _setmode(_fileno(stdin), _O_WTEXT);
  _setmode(_fileno(stdout), _O_WTEXT);
  wchar_t text[] = L"<ą|ę|ł|ß|\u0413>";
  wchar_t txt[128];
  wscanf(L"%ls", txt);
  wprintf(L"%ls:%ls\n", text, txt);
  return 0;
  };
```

Alternatywą jest stkorzystanie z WinAPI[4]. Proszę pamiętać o kodowaniu UTF-8. Podejście to powinno działać na każdym kompilatorze umożliwiającym pisanie programów dla Windows. Np. (zob. plik L7_uwin2.cpp):

```
#include<wchar.h>
#include<windows.h>

int main() {
  HANDLE out, in;
  DWORD nwr, nrd;
  wchar_t text[] = L"<ą|ę|ł|ß|\u0413>";
  wchar_t txt[128];
  in = GetStdHandle(STD_INPUT_HANDLE);
  out = GetStdHandle(STD_OUTPUT_HANDLE);
  ReadConsoleW(in, txt, sizeof(txt) / sizeof(txt[0]),
               &nrd, NULL);
  txt[nrd] = 0;
  WriteConsoleW(out, text, wcslen(text), &nwr, NULL);
  WriteConsoleW(out, L"\n", 1, &nwr, NULL);
  WriteConsoleW(out, txt, nrd, &nwr, NULL);
  return 0;
  };
```

W przypadku systemów rodziny Linux i kompilatora GCC:

- wymagane jest ustawienie systemowego locale zamiast C (wołając setlocale(LC_ALL, "")),

[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682073%28v=vs.85%29.aspx

- należy korzystać z szerokich napisów (wchar_t text[] = L"napis"),
- należy korzystać z szerokich wersji funkcji we/wy: wprintf zamiast printf, wscanf zamiast scanf; format string dla szerokich napisów to %ls
- plik źródłowy powinien być zapisany w UTF-8,
- należy dodać opcję kompilacji -std=c11

Np. (zob. plik L7_gcc.c):

```
#include<stdio.h>
#include<wchar.h>
#include<locale.h>

int main() {
  setlocale(LC_ALL, "");
  wchar_t text[] = L"<ą|ę|ł|ß|\u0413>";
  wchar_t txt[128];
  wscanf(L"%ls", txt);
  wprintf(L"%ls:%ls\n", text, txt);
  return 0;
  };
```

W przypadku systemów rodziny Linux i kompilatora G++:

- wymagane jest ustawienie systemowego locale zamiast C (wołając setlocale(LC_ALL, "")),
- należy korzystać z szerokich napisów (wchar_t text[] = L"napis"),
- należy korzystać z szerokich wersji strumieni wejścia/wyjścia: wcout zamiast cout, wcin zamiast cim,
- plik źródłowy powinien być zapisany w UTF-8,
- należy dodać opcję kompilacji -std=c11

Np. (zob. plik L7_gpp.cpp):

```
#include<iostream>
using namespace std;

int main() {
  wchar_t text[] = L"<ą|ę|ł|ß|\u0413>";
  setlocale(LC_ALL, "");
  wcout << text << endl;
  for(int i = 0; text[i]; i++) wcout << (int)text[i] << ' ';
  wcout << endl;
  return 0;
  };
```