7 Operacje wejścia/wyjścia

7.1 Standardowe wejście i wyjście

W systemie operacyjnym z każdym procesem (działającym programem) związany jest wejściowy i wyjściowy strumień znaków oraz strumień komunikatów o błędach. Na rysunku 7.1a przedstawiono pojedynczy proces i strumienie znaków wchodzące i wychodzące z niego.

W systemie Unix/Linux strumień wejściowy nazywa się `stdin`, strumień wyjściowy `stdout` a strumień błędów `stderr`.

Szczególnie ważne są dwa pierwsze strumienie, które służą do wprowadzania danych (`stdin`) i wyprowadzania wyników (`stdout`). Zwykło się je nazywać standardowym wejściem i standardowym wyjściem.

Kiedy program w języku C czyta dane funkcją `scanf()`, to czytane są one ze standardowego wejścia. Z kolei kiedy program drukuje wyniki funkcją `printf()`, to wysyłane one są na standardowe wyjście.

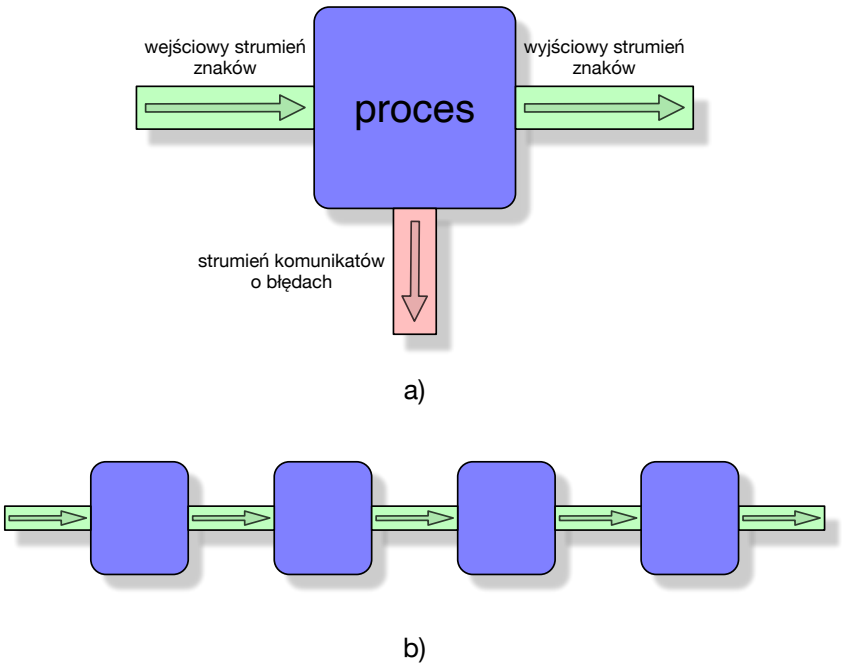
Jak pokażemy na przykładzie przekierowania i potoków, często samo czytanie ze standardowego wejścia i drukowanie do standardowego wyjścia jest wystarczające.

7.1.1 Przekierowania

Gdy uruchamiamy program o nazwie `main` na terminalu, to czytając dane z standardowego wejścia będzie on czytał to co wpiszemy na klawiaturze a drukując wyniki na standardowe wyjście będzie drukował na terminalu:

```
$ ./main
DANE WPISUJEMY NA KLAWIATURZE
WYNIKI POJAWIAJĄ SIĘ W OKNIE TERMINALA
```

Co jednak zrobić gdy dane znajdują się w pliku tekstowym `dane.txt`? Czy musimy zmienić treść programu i skompilować go aby mógł przeczytać dane.



Rysunek 7.1: a) Proces i strumienie znaków; b) Przetwarzanie potokowe.

tego pliku?

Otóż wystarczy użyć do tego przekierowania:

```
$ ./main < dane.txt
WYNIKI POJAWIAJĄ SIĘ W OKNIE TERMINALA
```

Po znaku < podajemy nazwę pliku, którego zawartość ma być skierowana na standardowe wejście programu.

Podobnie jeśli chcemy zachować wyniki w pliku `wyniki.txt`, to możemy odpowiednio przekierować standardowe wyjście do pliku:

```
$ ./main > wyniki.txt
DANE WPISUJEMY NA KLAWIATURZE
```

Po znaku > podajemy nazwę pliku, do którego mają zostać skierowane wyniki ze standardowego wyjścia.

Możemy również jednocześnie przekierować standardowe wejście i wyjście:

```
$ ./main < dane.txt > wyniki.txt
NA TERMINALU POJAWIĄ SIĘ EWENTUALNE KOMUNIKATY O BŁĘDACH
```

To dzięki przekierowaniom polecenia systemu Unix/Linux nie pytają się za każdym razem z jakiego pliku mają czytać dane i do jakiego pliku mają zapisywać wyniki.

7.1.2 Potoki

Systemy operacyjne potrafią jeszcze więcej jeśli chodzi o standardowe wejście i wyjście.

Załóżmy, że mamy trzy programy:

`generowanie` to program, który przygotowuje dane i wyprowadza je na standardowe wyjście,

`przetwarzanie` to program, który czyta dane ze standardowego wejścia, następnie je przetwarza i wyprowadza wyniki na standardowe wyjście,

`prezentowanie` to program, który czyta wyniki ze standardowego wejścia i odpowiednio je prezentuje w czytelny dla użytkownika sposób na standardowym wyjściu.

Stosując przekierowanie można uruchomić je w kolejnych poleceniach:

```
$ ./generowanie > dane.txt
$ ./przetwarzanie < dane.txt > wyniki.txt
$ ./prezentowanie < wyniki.txt
```

Takie uruchomienie tych programów ma następujące wady:

1. Potrzebne jest miejsce na dysku na plik `dane.txt` zawierający przygotowane dane.
2. Przetwarzanie danych (uruchomienie programu `przetwarzanie`) rozpocznie się dopiero wtedy gdy całkowicie zakończy się generowanie danych (zakończy się działanie programu `generowanie`).
3. Potrzebne jest miejsce na dysku na plik `wyniki.txt` zawierające wyniki w postaci nieczytelnej dla użytkownika.
4. Prezentowanie wyników (uruchomienie programu `prezentowanie`) rozpocznie się dopiero wtedy, gdy zakończy się przetwarzanie danych tj. zostaną wygenerowane całe wyniki (zakończy się działanie programu `przetwarzanie`).

Pamiętajmy, że systemy operacyjne są wielozadaniowe, tzn. w tym samym czasie mogą na nich działać liczne programy. Jeśli komputery są wyposażone w wiele procesorów (rdzeni), to programy te mogą wręcz działać jednocześnie.

Można sobie wyobrazić sytuację, w której przetwarzanie danych mogłoby się rozpocząć jeszcze przed wygenerowaniem całych danych. Podobnie prezentowanie wyników mogłoby się rozpocząć przed przetworzeniem całych danych.

W takiej sytuacji wygodnie jest skorzystać z potoków:

```
$ ./generowanie | ./przetwarzanie | ./prezentowanie
```

Oddzielając dwa polecenia znakiem `|` łączy się standardowe wyjście pierwszego polecenia ze standardowym wejściem drugiego polecenia. Powstaje wtedy potok jednocześnie działających poleceń przekazujących sobie dane.

W naszym przykładzie wszystkie trzy programy `generowanie`, `przetwarzanie` i `prezentowanie` zostają uruchomione jednocześnie, przy czym program `przetwarzanie`

rozpocznie przetwarzanie danych natychmiast kiedy na jego wejściu pojawi się pierwsza dana wygenerowana programem `generowanie`. Podobnie program `prezentowanie` rozpocznie prezentowanie wyników natychmiast gdy na jego wejściu pojawi się pierwszy wynik z programu `przetwarzanie`.

Na rysunku 7.1o przedstawiono potok jednocześnie działających procesów połączonych swoimi wyjściami/wejściami.

7.2 Operacje na standardowym wejściu i wyjściu

Aby korzystać ze standardowego wejścia i wyjścia należy dołączyć plik nagłówkowy `stdio.h`.

7.2.1 Czytanie ze standardowego wejścia

W punkcie tym omówione zostaną podstawowe funkcje służące do czytania ze standardowego wejścia `stdin`.

Funkcja getchar

Aby przeczytać pojedynczy znak z wejściowego strumienia znaków należy użyć funkcji `getchar()`. Wartością jej jest kod przeczytanego znaku albo stała `EOF` oznaczająca koniec strumienia.

Domyślnie aby przeczytać znak należy po jego wprowadzeniu nacisnąć klawisz `Enter`.

Jeśli chcemy pobrać znak bez czekania na naciśnięcie klawisza `Enter`, to wymaga to trochę kombinowania.

W systemie Unix/Linux dostępne jest polecenie `stty`, które służy do sterowania działaniem terminala. Terminal domyślnie działa w trybie „**ugotowanym**” (ang. *cooked*) czyli wymaga naciśnięcia klawisza `Enter` po każdej wprowadzonej porcji danych (znaków). Poleceniem `stty` można przełączyć go w tryb „**surowy**” (ang. *raw*), w którym nie jest wymagane naciśnięcie klawisza `Enter` po wprowadzeniu znaku.

Aby w programie zmienić stan terminala poleceniem `stty` należy skorzystać z funkcji `system()`, która uruchamia dowolne polecenie w systemie (wymaga dołączenia pliku nagłówkowego `stdlib.h`).

Poniższy przykład pokazuje jak zatrzymać działanie programu do chwili gdy użytkownik naciśnie dowolny klawisz.

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

int main(void)
{
  char ch;

  printf("wciśnij dowolny klawisz\n");
  system("stty raw");
  ch = getchar();
  system("stty cooked");
  printf("kontynuuję pracę\n");
```

```
  return 0;
}
```

Funkcja scanf

Stosując funkcję `scanf()` możemy czytać dane różnych typów. To jak zostaną zinterpretowane przeczytane znaki zależy od formatu jaki podamy w pierwszym argumencie wywołania funkcji.

Funkcja `scanf()` zwraca jako swój wynik liczbę poprawnie przeczytanych danych.

Funkcja rozpoczyna analizowanie znaków pominąwszy wcześniej znaki białe (odstęp, tabulacja, nowa linia). Natomiast kończy czytanie znaków po napotkaniu pierwszego białego znaku za znakami zawierającymi daną.

Zatem, jeśli czytamy napis formatem `%s`, to należy pamiętać, że zostanie przeczytane tylko pierwsze słowo. Wpisując na wejście programu napis `Ala ma Asa.` zostanie wczytane tylko słowo `Ala`. Kolejne słowa należy czytać kolejnymi wywołaniami funkcji `scanf()`.

7.2.2 Pisanie na standardowe wyjście

W punkcie tym omówione zostaną podstawowe funkcje służące do pisania na standardowym wyjściu `stdout`.

Funkcja putchar

Aby wydrukować pojedynczy znak możemy skorzystać z funkcji `putchar()`, której argumentem wywołania jest kod drukowanego znaku.

Na listingu 7.1 przedstawiono program, który kopiuje wszystkie znaki ze standardowego wejścia na standardowe wyjście.

Listing 7.1: Kopiowanie standardowego wejścia na standardowe wyjście

```
// kopiowanie.c
//
// Program kopiuje standardowe wejście na standardowe wyjście.

#include <stdio.h>

int main(void)
{
  char ch;
  while((ch = getchar()) != EOF) putchar(ch);
  return 0;
}
```

Wyjaśnienia wymaga warunek pętli **while** w powyższym programie. Wykonuje on trzy rzeczy:

1. Czyta kolejny znak ze standardowego wejścia za pomocą funkcji `getchar()`.
2. Podstawia pod zmienną `ch` kod zwrócony przez funkcję `getchar()`.

3. Wartością wyrażenia `(ch = getchar())` jest wartość podstawiona pod zmienną `ch` i wartość ta jest porównywana ze stałą `EOF`.

Działanie pętli **while** kończy się w chwili gdy osiągnięto koniec strumienia wejściowego i nie udało się przeczytać kolejnego znaku (funkcja `getchar()` zwróciła wartość `EOF`).

Pętlę `while` z listingu 7.1 możemy zapisać w sposób następująco:

```
char ch = getchar();
while(ch != EOF)
{
  putchar(ch);
  ch = getchar();
}
```

Okazuje się, że na każdym z poziomów optymalizacji (nawet na niskim poziomie `-O1`) przekłady na język maszynowy tych dwóch pętli niczym się nie różnią. Dzięki optymalizacji kodu nie jest konieczny taki styl programowania, w którym starano się w jednym wyrażeniu zawrzeć jednoczesne liczenie wartości wyrażenia i zmienianie stanu obliczeń (podstawienia).

Funkcja printf

Dzięki funkcji `printf()` mamy możliwość prezentowania danych w postaci sformatowanego ciągu znaków czytelnych dla człowieka. Możemy ustalić między innymi ile znaków przeznaczamy na prezentację danej (tzw. szerokość pola) oraz przy typach rzeczywistych zadać liczbę drukowanych cyfr po przecinku).

7.3 Operacje na plikach

Omówimy teraz podstawowe operacje na plikach tekstowych.

Aby program mógł operować na pliku musi znać informacje o pliku takie jak np. jaka jest bieżąca pozycja czytania (w którym miejscu pliku jesteśmy z jego czytaniem). Informacje te przechowywane są w tzw. deskryptorze pliku.

Wywołując funkcje operujące na pliku należy dostarczyć im wskazanie na jego deskryptor. Deskryptor pliku jest typu `FILE`, który zdefiniowano w pliku nagłówkowym `stdio.h`.

Wskazanie na deskryptor pliku deklarujemy następująco:

```
FILE *fp;
```

W pliku nagłówkowym `stdio.h` zdefiniowano między innymi następujące trzy wskazania na deskryptory:

`stdin` wskazanie na deskryptor standardowego wejścia,

`stdout` wskazanie na deskryptor standardowego wyjścia,

`stderr` wskazanie na deskryptor komunikatów o błędach.

7.3.1 Otwieranie pliku

Aby powiązać deskryptor pliku z konkretnym plikiem na dysku należy użyć funkcji `fopen()`. Jej pierwszym argumentem jest nazwa pliku (może zawierać ścieżkę do pliku) a drugim tryb otwarcia pliku. Zwracaną wartością jest wskazanie na deskryptor otwartego pliku.

Możliwe tryby otwarcia, to

`r` otwarcie dla czytania (plik musi istnieć)

`w` otwarcie dla pisania (plik nie musi istnieć)

`a` otwarcie dla dopisywania na końcu (plik nie musi istnieć)

Możliwe są również tryby dla jednoczesnego czytania i pisania, które oznacza się dopisując znak `+` na końcu odpowiedniego trybu.

Jeśli nie uda się otworzyć pliku (bo np. chcemy z niego czytać a plik nie istnieje), to funkcja zwraca wartość `NULL`.

Poniższy przykład pokazuje jak otworzyć plik do czytania:

```
FILE *fp;
fp = fopen("katalog/plik.txt", "r");
```


7.3.2 Czytanie z pliku

Odpowiednikiem funkcji `getchar()` jest funkcja `fgetc()`. Jej argumentem jest wskazanie na deskryptor czytanego pliku a wartością kod przeczytanego znaku albo `EOF` jeśli plik skończył się.

Poniżej przedstawiono jak na trzy sposoby można przeczytać znak ze standardowego wejścia:

```
char ch1 = getchar();
char ch2 = fgetc(stdin);
FILE *fp = stdin;
char ch3 = fgetc(fp);
```

Odpowiednikiem funkcji `scanf()` jest funkcja `fscanf()`. Jej dodatkowym pierwszym argumentem jest wskazanie na deskryptor czytanego pliku. Również zwraca ono liczbę przeczytanych danych.

Poniższe trzy instrukcje czytania są sobie równoważne:

```
scanf("%d %f", &n, &x);
fscanf(stdin, "%d %f", &n, &x);
FILE *fp = stdin;
fscanf(fp, "%d %f", &n, &x);
```

7.3.3 Pisanie do pliku

Odpowiednikiem funkcji `putchar()` jest funkcja `fputc()`. Jej dodatkowym pierwszym argumentem jest wskazanie na deskryptor pisanego pliku.

Poniżej przedstawiono jak na trzy sposoby można wypisać znak na standardowe wyjście:

```
putchar('A');
fputc(stdout, 'A');
FILE *fp = stdout;
fputc(fp, 'A');
```

Odpowiednikiem funkcji `printf()` jest funkcja `fprintf()`. Jej dodatkowym pierwszym argumentem jest wskazanie na deskryptor pisanego pliku.

Poniższe trzy instrukcje pisania są sobie równoważne:

```
printf("n=%d x=%f", n, x);
fprintf(stdout, "n=%d x=%f", n, x);
FILE *fp = stdout;
fprintf(fp, "n=%d x=%f", n, x);
```

7.3.4 Zamykanie pliku

Kiedy kończymy operować na pliku musimy go zamknąć funkcją `fclose()`:

```
int fclose(FILE *fp);
```

Jako wynik zwraca ona kod powodzenia. Wartość zero oznacza, że plik został poprawnie zamknięty.

Trzeba koniecznie pamiętać o zamykaniu pliku. Warto to zrobić natychmiast po tym gdy program skończy korzystać z niego.

Pamiętajmy, że operacje odczytywania i zapisywania na dysku są dużo wolniejsze niż zapisywanie i odczytywanie z pamięci operacyjnej. Z tego powodu system operacyjny zamiast pisać co chwila do pliku zapisuje drukowane dane w buforze przechowywanym w pamięci operacyjnej komputera. Dopiero gdy bufor się wypełni dane zostaną przeniesione z bufora na dysk.

Oznacza to, że wypisane do pliku dane wcale nie muszą od razu znaleźć się na dysku. Jeśli program zakończy pracę bez zamknięcia pliku (np. w wyniku utraty zasilania), to dane nie zostaną przeniesione z bufora na dysk i bezpowrotnie je utracimy.

Co jeśli program działa miesiącami i co jakiś czas wypisuje do pliku żmudnie wyliczane dane? Wówczas najbezpieczniej jest przed każdym wypisaniem kolejnej danej otworzyć plik do dopisywania (tryb `a`) i po wypisaniu natychmiast go zamknąć.

7.3.5 Przykład

Zanim przedstawimy przykład programu operującego na plikach, pokażemy w jaki sposób program w języku C może dostać się do argumentów jego wywołania.

Na listingu 7.2 przedstawiono program `argumenty.c`.

Listing 7.2: Pobieranie argumentów wywołania

```
// argumenty.c
//
// Program drukuje argumenty uruchomienia programu.

#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
  printf("Liczba argumentów (argc) = %d\n", argc);
  printf("Kolejne argumenty o indeksach od 0 do %d:\n", argc -
      1);
  for(int i = 0; i < argc; i++)
    printf("\targv[%d] = \"%s\"\n", i, argv[i]);
  return 0;
}
```

Zawarta w nim funkcja `main()` ma dwa parametry:

`argc` liczba argumentów powiększona o jeden,

`argv` wektor wskazań na napisy będą kolejnymi argumentami wywołania (zerowym argumentem jest sama nazwa polecenia).

Oto kilka wywołań programu `argumenty`:

```
$ ./argumenty
Liczba argumentów (argc) = 1
Kolejne argumenty o indeksach od 0 do 0:
argv[0] = "./argumenty"
$ ./argumenty pierwszy drugi trzeci
Liczba argumentów (argc) = 4
Kolejne argumenty o indeksach od 0 do 3:
argv[0] = "./argumenty"
argv[1] = "pierwszy"
argv[2] = "drugi"
argv[3] = "trzeci"
$ ./argumenty "ten jest pierwszy" 'a ten jest drugi'
Liczba argumentów (argc) = 3
Kolejne argumenty o indeksach od 0 do 2:
argv[0] = "./argumenty"
argv[1] = "ten jest pierwszy"
argv[2] = "a ten jest drugi"
```


Ciekawą własnością systemu Unix/Linux jest możliwość traktowania napisów wysyłanych na standardowe wyjście jednego polecenia jako argumenty drugiego polecenia:

```
$ ls
argumenty kopiowanie liczby.txt maze.c sumator.c
argumenty.c kopiowanie.c maze sumator
$ ./argumenty 'ls'
Liczba argumentów (argc) = 2
Kolejne argumenty o indeksach od 0 do 1:
argv[0] = "./argumenty"
argv[1] = "ls"
$ ./argumenty `ls`
Liczba argumentów (argc) = 10
```

```
Kolejne argumenty o indeksach od 0 do 9:
argv[0] = "./argumenty"
argv[1] = "argumenty"
argv[2] = "argumenty.c"
argv[3] = "kopiowanie"
argv[4] = "kopiowanie.c"
argv[5] = "liczby.txt"
argv[6] = "maze"
argv[7] = "maze.c"
argv[8] = "sumator"
argv[9] = "sumator.c"
```

Aby polecenie zostało zastąpione napisami przezeń wydrukowanymi musi być ujęte w odwrócone apostrofy.

Rozpatrzmy przykład programu, który czyta kolejne liczby rzeczywiste i liczy ich sumę (listing 7.3).

Listing 7.3: Sumowanie liczb


```
// sumator.c
//
// Program czyta liczby rzeczywiste i oblicza ich sumę.
// Jeśli nie podano parametru wywołania, to czyta ze
    standardowego wyjścia.
// W przeciwnym przypadku czyta z pliku, którego nazwę podano.
// Program kończy czytanie gdy skończy się strumień znaków lub
    nie może
// zinterpretować wejścia jako liczba rzeczywista.

#include <stdio.h>
#include <stdbool.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  if(argc < 2)
    fp = stdin;
  else
    {
      fp = fopen(argv[1], "r");
      if(fp == NULL)
        {
          printf("Plik %s nie istnieje!\n", argv[1]);
          return 1;
        }
    }

  double suma = 0.0;

  while(true)
    {
      double wartosc;
      int wynik = fscanf(fp, "%lf", &wartosc);
      if(wynik < 1)
```

```
            break;
        suma = suma + wartosc;
    }
  fclose(fp);

  printf("suma = %f\n", suma);
  return 0;
}
```

Program sprawdza czy podano argument jego wywołania. Jeśli tak, to liczby czyta z pliku o nazwie podanej w tym argumencie. W przeciwnym przypadku czyta liczby ze standardowego wejścia.

Załóżmy, że w pliku `liczby.txt` znajdują się następujące liczby:

```
1.0
3.0
5.0
7.0
9.0
```

Oto przykładowe dwa uruchomienia programu:

```
$ ./sumator liczby.txt
suma = 25.000000
MacBook-Pro-Przemyslaw:wyk_7 przemko$ ./sumator
1
2
3
[ctrl-d]
suma = 6.000000
```


7.4 Ciekawostka

Kilkuset stronicowa książka pod frapującym tytułem

10 PRINT CHR$(205.5+RND(1)); : GOTO 10

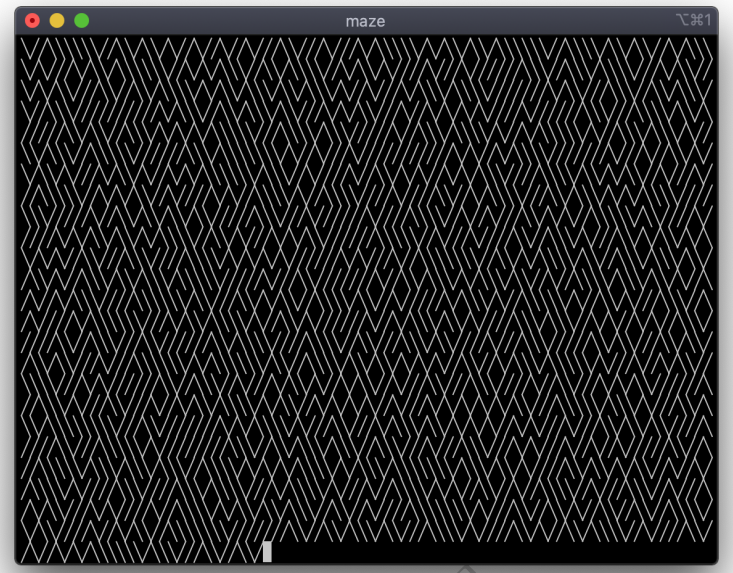
poświęcona jest jednolinijkowemu programowi w języku BASIC. Przedstawiono w niej dogłębną analizę wyniku działania tego programu.

Książka dostępna jest w formacie PDF na stronie: `http://10print.org`

Na listingu 7.4 przedstawiono analogiczny kod w języku C.

Listing 7.4: Drukowanie losowego labiryntu

```
// maze.c
//
// Generowanie nieskończonego losowego "labiryntu".
//
// Na podstawie książki pt. "10 PRINT CHR$(205.5+RND(1)); :
    GOTO 10"
// http://10print.org
```



Rysunek 7.2: Efekt działania programu `maze.c`.

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

int main(void)
{
  srand(time(NULL));
  while(1)
  {
    printf(rand()&1?"\u2571":"\u2572");
    fflush(stdout);
  }
}
```

Wykorzystano w nim znaki ukośnych linii zapisane w kodowaniu UTF-16 (prefiks \u a za nim kod szesnastkowy).

Aby drukowane znaki pojawiały się natychmiast na ekranie (nie były przechowywane w buforze) wywoływana jest funkcja `fflush()`, która zrzuca zawartość bufora do pliku na dysk.

Na rysunku 7.2 przedstawiono efekt działania powyższego programu.