Język ANSI C

część 13

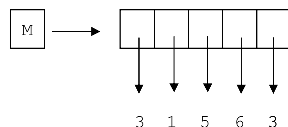
wskaźniki do funkcji
funkcje o zmiennej liczbie parametrów

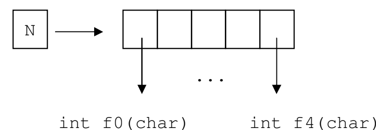
Jarosław Gramacki
Instytut Informatyki i Elektroniki

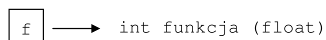
Wskaźniki do funkcji

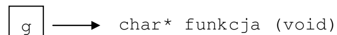
– kilka podstawowych i kilka bardziej złożonych deklaracji

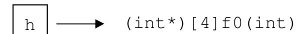
```
int *(*M)[5];     // wskaźnik M do tablicy 5-u wskaźników do int

int (*f)(float);  // wskaźnik do funkcji pobierającej float a zwracającej int
                  // wywołania: x = f(y); LUB x = (*f)(y);
                  bez nawiasów:
                  int *f (float);

char *(*g)(void); // wskaźnik do funkcji bezparametrowej zwracającej char*

int (*(*N)[5])(char);      // wskaźnik tablicy 5-u wskaźników do funkcji o
                           // argumencie char zwracających int

int (*(*h)(int))[4];       // wskaźnik do funkcji o argumencie int zwracających
                           // wskaźnik do tablicy 4-ch elementów typu int
```











Wskaźniki do funkcji

– kilka podstawowych i kilka bardziej złożonych deklaracji

```
// Standardowa funkcja sinus ma prototyp w postaci:  double sin(double).
// Po deklaracji double (*fun)(double) = sin; możliwe
// są równoważne wywołania:

double x, y;

y = sin(x);
y = (*fun)(x);    // wywołanie "zwykłe", 2 x ()()
y = fun(x);       // wywołanie "uproszczone", 1 x ()
```

drugie () to operator wykonania funkcji. Pierwsze () konieczne ze względu na priorytet

```
// Można również zdefiniować tablicę wskaźników do funkcji:

double ( *tw[ ] ) (double) = {sin, cos, tan};  // deklaracja + inicjalizacja

// tw[0] - wskaźnik funkcji sin;
// tw[1] - wskaźnik funkcji cos;
// tw[2] - wskaźnik funkcji tan;

// Wówczas, równoważne są następujące wywołania:

y = ( *tw[0] )(x);         y = tw[0](x);     // j.w. 2 x ()() oraz 1 x ()
y = ( *tw[1] )(x);         y = tw[1](x);
y = ( *tw[2] )(x);         y = tw[2](x);
```


Wskaźniki do funkcji

- Na wskaźnikach funkcji można wykonywać następujące operacje:
  - przekazywać jako argumenty do innych funkcji
  - zwracać jako wynik funkcji
  - porównywać ze wskaźnikiem NULL
  - poddawać operacji wyłuskania
- Uwaga:
  na wskaźnikach funkcji nie wolno wykonywać operacji arytmetycznych.
- W języku C funkcje nie mogą zawierać definicji innych funkcji. Można jednak przekazywać do funkcji wskaźniki innych funkcji.
- Na przykład można zaprojektować funkcję, która korzysta z rodziny funkcji typu `void (*f)(void)`.

```
void pisz( int n, void (*f)  (void) )
{
  f();
  printf("%d", n);
}

Wywołanie funkcji:  pisz(5, funkcja);      // funkcja, NIE funkcja();
```

qsort()

– wskaźniki do funkcji nie są często stosowane, ale są sytuacje, gdy jest to niezbędne

```
void qsort(      // dużo czytelniej !!!
   void *base,
   size_t num,
   size_t width,
   int (*compare )(const void *, const void *)
);
```

```
// qsort - sort a table of data
#include <stdlib.h>

void qsort( void* base, size_t num, size_t width,
int (*compare)(const void *, const void *) );

base                Pointer to the first element of the array where
                    the sorting process has to be performed.
num                 Number of elements in the array pointed by base.
width               Width of each element in the array.
fncompare           Function that compares two elements. This should be provided
                    by the caller and must be properly casted

        int fncompare (const void *elem1, const void *elem2 );

    The function should receive two parameters that are pointers
    to elements, and should return an int value with result of comparing them:

        return value      description
        <0                *elem1 goes before *elem2
        0                 *elem1 == *elem2
        >0                *elem1 goes after *elem2
```


Przykład 1 użycia qsort()

qsort() pokaże swój prawdziwy potencjał przy sortowaniu dużej ilości danych !

```
// sortowanie tablicy liczb
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };         // table to be sort

int compare (const void *a, const void *b) // function that compares ...
{
  return ( *(int*)a - *(int*)b );           // rzutowanie konieczne
}


int main ()
{

  int n;
  qsort (values, 6, sizeof(int), compare);

  for (n=0; n<6; n++)
    printf ("%d ",values[n]);      // Uwaga, posortowano oryginalna tablicę !

  return 0;
}
Output:
10 20 25 40 90 100
```

w wywołaniu: nazwa funkcji porównującej
Nazwa funkcji to wskaźnik do niej.

(analogicznie jak dla argumentów tablicowych)

Algorytm sortowania qsort (oparty na technice „dziel i rządź". Możliwy do zaimplementowania w wersji rekurencyjnej lub iteracyjnej.
Jest to temat na oddzielny wykład !

Przykład 2 użycia qsort()

```
// sortowanie tablicy napisów

// inaczej niż w przykład 1
int compare (const void *a, const void *b)
{
    char *x = (char *)a;       // rzutowanie do wskaźnika na typ char
    char *y = (char *)b;
    return strcmp(x,y);        // porównanie łańcuchów
}

// tablica 6 łańcuchów o rozmiarze 20
char stab[][20] = { "Wanda", "Ola", "Ala", "Basia", "Asia", "Joasia" };

int main(void) {
      int i, N;
      N = sizeof(stab) / sizeof(stab[0]);   // liczba łańcuchów

      for (i=0; i<N; i++)
        printf ("%s ", stab[i]);

      qsort (stab, N, sizeof(stab[0]), compare);

      for (i=0; i<N; i++)
         printf ("%s ", stab[i]);

return 0;
}
```


Przykład 3 użycia qsort()

qsort() library function allows sorting of nonnumerical objects, so the user must supply his/her own comparison function, which defines whether one object is "smaller" than another;

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

// Compare all of both strings:
static int campare (const void *p1, const void *p2)
{
    /* The arguments to this function are "pointers to
       pointers to char", but strcmp() arguments are "pointers
       to char", hence the following cast plus dereference */

    return strcmp(* (char **) p1, * (char **) p2);
}

int main(int argc, char *argv[]) {
    int j;

    assert(argc > 1);

    // sort program parameters
    qsort(&argv[1], argc - 1, sizeof(char *), compare);

    for (j = 1; j < argc; j++)
        puts( argv[j] );  // Uwaga, posortowano oryginalna tablicę argv !
    exit(EXIT_SUCCESS);  }
```

gdyż sortujemy argumenty wywołania programu

Effekt assert-a:

jarek@mykonos:~$./prog
Assertion failed: argc > 1, file prog.c, line 22

qsort(), do samodzielnego wykonania

- na poprzednim wykładzie były podane przykłady dynamicznej alokacji pamięci dla danych pobieranych z pliku tekstowego
- tu pokazano efektywne podejście do sortowania danych
- napisz więc działający program, który posortuje (z użyciem qsort()) bardzo duży plik tekstowy (na przykład pobrany z internetu) wierszami, słowami (plus ew. informacja o numerze linii, z której to słowo pochodzi)
- zrób to samo ale jakąś prostszą (w sensie algorytmicznym) funkcją sortującą (napisz ją sam). Porównaj czas sortowania w obu przypadkach

Własna funkcja używająca wskaźnika do funkcji

- funkcja znajdująca zero dowolnej ciągłej funkcji rzeczywistej zmiennej rzeczywistej
- testujemy również dla funkcji bibliotecznej `sin()`

```
#include <stdio.h>
#include <math.h>// konieczne dla sin() oraz M_PI

int zero (double (*)(double), double, double, double*, double);
double MyFun (double);

int main(void)
{
    double z;
    // radiany ?
    printf("Kod wykonania: %d, ", zero (sin, -1.0, 1.0, &z, 1.0e-2));
    printf("Zero wynosi: %le\n", z);

    printf("Kod wykonania: %d, ", zero (sin, M_PI/2, 1.5 * M_PI, &z, 1.0e-2));
    printf("Zero wynosi: %le\n", z);

    printf("Kod wykonania: %d, ", zero (MyFun, 0.0, 10.0, &z, 1.0e-2));
    printf("Zero wynosi: %le\n", z);

    return 1;
}
```

```
Kod wykonania: 1, Zero wynosi: 0.000000e+000
Kod wykonania: 1, Zero wynosi: 3.147729e+000       // Pi radianów
Kod wykonania: 1, Zero wynosi: 5.146484e+000       // Rys. dalej
```

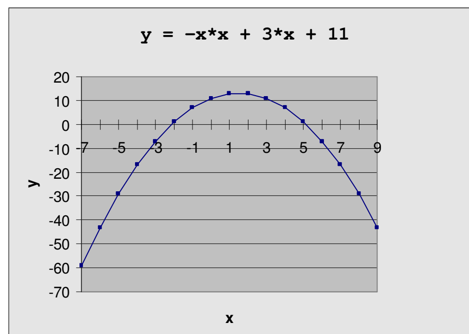
Własna funkcja używająca wskaźnika do funkcji

– uwagi o funkcji `sin()` z `math.h`

```
NAME
       sin, sinf, sinl - sine function

SYNOPSIS
       #include <math.h>
       double sin ( double x );
       float sinf ( float x );
       long double sinl( long double x );
       Link with -lm.

DESCRIPTION
       The  sin() function returns the sine of x, where x is given in radians.

RETURN VALUE
       The sin() function returns a value between -1 and 1.
```


Własna funkcja używająca wskaźnika do funkcji

```
int zero (double (*fun)(double), double a, double b, double *zero, double eps)
{
  double s,            // środek bieżącego przedziału
         fs,           // wartość f(s)
         fa, fb;       // wartości F(a) i f(b)
  fa = (*fun)(a);
  fb = (*fun)(b);
  if (fa * fb >= 0 || a >= b) return (-1); // zły przedział

  while (b - a > eps)
   {
     s = (b + a) / 2.0;
     fs = (*fun)(s);
     if (fs == 0) break;    // znaleziono zero
     if (fa * fs < 0)
       { b=s; fb = fs; }
     else
       { a=s; fa = fs; }
   }
   *zero = s;
   return 1;
}

// dowolna (ciągła i rzeczywista)
// funkcja własna
double MyFun (double x) {
  return -x*x + 3*x +11; }
```



Tablice wskaźników do funkcji

– Projekt 2-poziomowego menu (jakiegoś programu)

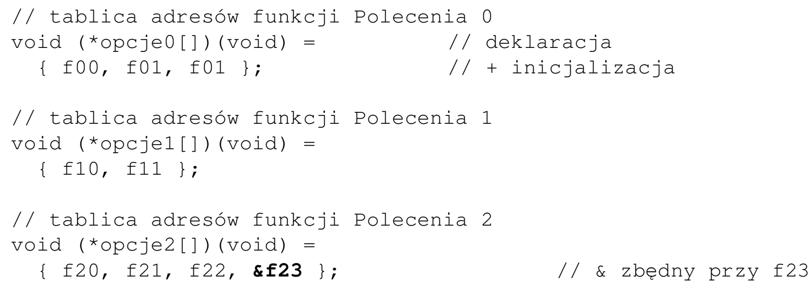
```
Polecenie 0
        Opcja00
        Opcja01
        ...
Polecenie 1
        Opcja10
        Opcja11
        ...
...
```

```
// prototypy dla opcji polecenia 0
void f00 (void);
void f01 (void);
void f02 (void);

// prototypy dla opcji polecenia 1
void f10 (void);
void f11 (void);

// prototypy dla opcji polecenia 2
void f20 (void);
void f21 (void);
void f22 (void);
void f23 (void);
```

```
void f21 (void) {  printf("\n Wywołanie f21"); }
```


Tablice wskaźników do funkcji

– Dla każdego polecenia utworzymy tablicę adresów funkcji odpowiadających jego opcjom



```
// tablica adresów funkcji Polecenia 0
void (*opcje0[])(void) =          // deklaracja
  { f00, f01, f01 };              // + inicjalizacja

// tablica adresów funkcji Polecenia 1
void (*opcje1[])(void) =
  { f10, f11 };

// tablica adresów funkcji Polecenia 2
void (*opcje2[])(void) =
  { f20, f21, f22, &f23 };                // & zbędny przy f23
```

Tablice wskaźników do funkcji

– Tworzymy tablice główną, która pozwoli (poprzez indeksowanie) "poruszać się" po menu

```
// tablica wskaźników do wskaźników do funkcji

void ( **menu[])(void) =
  { &opcje0[0],           // jak dla tablic: opcje0 było by równie dobre
    &opcje1[0],
    &opcje2[0]
  };
```

– zakładamy, że elementy tablicy zajmują kolejne miejsca w pamięci (i tak też jest w rzeczywistości)

– na przykład w tablicy `opcje0` po adresie funkcji `f00()` występuje adres funkcji `f01()`, potem `f02()`

```
(**menu[1])();    // wywołanie funkcji f10()
```

– element `menu[1]` jest adresem pierwszego elementu w tablicy `opcje1` czyli adresem adresu funkcji `f10()`.

```
(** ( menu[idx_menu] + idx_opcji ))();     // postać sparametryzowana
                                           // wywołania
```


Tablice wskaźników do funkcji

– uproszczenia zapisu skomplikowanych deklaracji

```
typedef void (**MENU)(void);
```

```
// tablica menu inaczej

MENU menu[] =
  { &opcje0[0],
    &opcje1[0],
    &opcje2[0]
  };
```

– poprzednia wersja

```
// tablica wskaźników do wskaźników do funkcji

void ( **menu[])(void) =
  { &opcje0[0],
    &opcje1[0],
    &opcje2[0]
  };
```

Tablice wskaźników do funkcji

– powyżej pokazano jedynie ogólny zarys metody
– w rzeczywistości całość byłaby dużo bardziej skomplikowana (u nas brakuje na przykład: nazw pozycji menu, skrótów klawiszowych, itp.)
– nasz przykład pokazuje (elegancki) sposób wydzielenia w kodzie poleceń i ich opcji. Łatwo więc rozbudowywać menu bez konieczności modyfikowania kodu obsługującego to menu
– kilka ulepszeń:

```
// kompilator sam określi wielkości tablic

#define ROZMIAR_OPCJE0 sizeof(opcje0) / sizeof(opcje0[0])

#define ROZMIAR_MENU sizeof(menu) / sizeof(menu[0])
```

```
// pusty wskaźnik w strukturze do określenia wielkości menu

typedef struct
{
  void (**opcje)(void);
  int rozmiar;
} MENU;

MENU menu[] = {
  { &opcje0[0], ROZMIAR_OPCJE0 },
  { &opcje1[0], ROZMIAR_OPCJE1 },
  { &opcje2[0], ROZMIAR_OPCJE2 }
  };
```

```
Wywołanie funkcji:
(**(menu[idx_menu].opcje +
idx_opcji))();
```


Funkcje o zmiennej liczbie parametrów

Funkcje o zmiennej liczbie parametrów

- stosunkowo rzadko używane, ale czasami jest to bardzo wygodny mechanizm
- przykład: funkcja standardowa `printf()` ma zmienną liczbę parametrów, gdyż zależy ona od liczby znaków formatujących zawartych w łańcuchu formatującym.
  `int printf( char *fmt, ...);`
- informację o zmiennej liczbie parametrów niosą trzy kropki (symbol niedokończenia) występujące na końcu listy parametrów funkcji

```
// fragment z stdio.h

_CRTIMP int __cdecl fprintf (FILE*, const char*, ...);
_CRTIMP int __cdecl printf (const char*, ...);
_CRTIMP int __cdecl sprintf (char*, const char*, ...);
```


Funkcje o zmiennej liczbie parametrów

```
// deklaracja funkcji obliczającej wartość wielomianu
// o zadanym stopniu w danym punkcie

double wielomian ( double x, int st, ... );
```

- funkcja `wielomian()` pobiera parametr `x`, określający punkt, w którym zostanie obliczona wartość wielomianu oraz `st` – stopień wielomianu. Parametry te nazywamy ustalonymi
- w zależności od stopnia wielomianu różna będzie liczba jego współczynników, nie sposób więc wstępnie określić liczby parametrów określających te współczynniki
- z tego powodu zamiast deklaracji kolejnych parametrów umieszczono informację (w postaci trzech kropek), że liczba pozostałych parametrów jest nie określona. Parametry te nazywamy nieustalonymi

Funkcje o zmiennej liczbie parametrów

- w celu dostępu do parametrów nieustalonych posługujemy się makrodefinicjami `va_start, va_arg, va_end` oraz typem `va_list`, zdefiniowanym w zbiorze nagłówkowym `<stdarg.h>`
- schemat wykorzystania parametrów nieustalonych: `patrz przykład dalej`

1. deklarujemy zmienną `ap` typu `va_list`

2. przed rozpoczęciem dostępu do parametrów nieustalonych wywołujemy makro `va_start` z parametrami: `ap` i identyfikatorem ostatniego parametru ustalonego funkcji.

3. w celu dostępu do parametrów nieustalonych wywołujemy makrodefinicję `va_arg` z parametrami `ap` i nazwą typu pobieranego parametru, np. `va_arg(ap, int)` spowoduje pobranie nieustalonego argumentu i potraktowanie go jako liczby całkowitej. Kolejne wywołania makrodefinicji `va_arg` będą powodowały pobieranie kolejnych parametrów funkcji.

4. po wyczerpaniu listy parametrów nieustalonych w celu ponownego do niej dostępu należy wywołać makro `va_start`. Dalsze postępowanie jest wówczas takie, jak omówiono wyżej

---

5. na zakończenie operacji na parametrach nieustalonych należy wywołać makro `va_end(ap)`. Pozwoli to na normalne zakończenie wykonywania funkcji o zmiennej liczbie parametrów.
(makro `va_end` powinno być wywoływane wyłącznie po przeczytaniu wszystkich argumentów nieustalonych za pomocą `va_arg`. Niespełnienie tego warunku może spowodować nieokreślone zachowanie programu.)

Przykład 0, zasada działania

```
// The first argument to the function is the count of remaining arguments,
// which are added up and the result is returned.
#include <stdarg.h>
#include <stdio.h>

int addInt (int count,...)
{
  va_list ap;
  int i, sum;

  va_start (ap, count);          // Initialize the argument list

  sum = 0;
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);     // Get the next argument value

  va_end (ap);                   // Clean up
  return sum;
}
int main (void) {
  // This call prints 16
  printf ("%d\n", addInt (3,    5, 5, 6) );

  // This call prints 55
  printf ("%d\n", addInt (10,  1, 2, 3, 4, 5, 6, 7, 8, 9, 10) );
  return 0; }
```

obejrzyj omawiane tu makra !
Są bardzo ciekawe

typ kolejnego parametru !!!

Przykład 1, wartość wielomianu w punkcie

```
#include <stdarg.h>       // deklaracje związane ze zmienną listą arg.
#include <math.h>

double wielomian(double x, int st, ...)
{
 double wartoscWiel = 0;

 va_list ap;
 va_start ( ap, st );

 for ( ; st; --st )
   wartoscWiel += va_arg( ap, double ) * pow( x, st );
   // np. 1.6 * x^3 w 1-szym kroku
 wartoscWiel += va_arg ( ap, double );        // wyraz wolny dla st == 0

 va_end ( ap );

 return wartoscWiel;
}

int main(void)  {
  // wartość wielomianu f(x)  stopnia 3 w punkcie 2.2
  // f(x) = 1.6*x^3  +  2.0*x^2  -  3.4*x  +  5.0

  printf("\n%lf", wielomian(2.2, 3,     1.6, 2.0, -3.4, 5.0) );
  return 0; }
```

parametr nieustalony typu double

parametry ustalone (x, st)

typ kolejnego parametru !!!

Przykład 2, "nakładka" na printf()

```
// The function takes a string of format chars and prints out the argument
// associated with each format character based on the type
#include <stdio.h>   #include <stdarg.h>

void foo(char *fmt, ...) {
  va_list ap;
  int d;     char c, *p, *s;

  va_start(ap, fmt);
  while (*fmt)
       switch( *fmt++ ) {                          // null pointer ends list
       case 's':                                   // string
                s = va_arg ( ap, char * );
                printf("string %s\n", s);
                break;
       case 'd':                                   // int
               d = va_arg ( ap, int );
               printf("int %d\n", d);
               break;
       case 'c':                                   // char
               // need a cast here since va_arg only
               // takes fully promoted types
               c = (char) va_arg ( ap, int );
               printf("char %c\n", c);
               break;
       }
 va_end(ap); }
```

dlaczego ?

Przykład 3, sklejanie dowolnej ilości napisów

```
// concatenates an arbitrary number of strings onto the end of an
// existing string

// (we assume that the existing string is stored in an
// object large enough to hold the resulting string

#include <stdio.h>
#include <stdarg.h>

void va_cat(char *s, ...)                    // va_cat
    {
    char *t;
    va_list ap;

    va_start( ap, s );
    while (t = va_arg( ap, char *) )         // null pointer ends list
        {
        s += strlen ( s );                   // skip to end
        strcpy ( s, t );                     // and copy a string
        }
    va_end ( ap );
    }
```


Przykład 4, Suma ciągu liczb

```
#include <stdio.h>   #include <stdarg.h>
double sum_series(int num, ...);

int main(void)   {
    double d;
    d = sum_series(5, 0.5, 0.25, 0.125, 0.0625, 0.03125);
    printf("Sum of series is %f.\n", d);
    return 0;
  }

 double sum_series(int num, ...)  {
    double sum=0.0, t;
    va_list argptr;

    /* initialize argptr */
    va_start(argptr, num);

    /* sum the series */
    for( ; num; num--) {
      t = va_arg(argptr, double); /* get next argument */
      sum += t;
    }

    va_end(argptr);
    return sum;
  }
```

Opis pewnego zadania projektowego

- VSM (Vector space model)

file_1

```
The indexing process constructs new and updates existing term-document
matrices from documents, in the form of MATLAB sparse arrays. It may include
various steps, such as removal of common words, such as articles and
conjunctions, removal of very short or very long terms, removal of very
frequent or infrequent terms, and the application of stemming.
```

file_2

```
Vector space model (or term vector model) is an algebraic model for
representing text documents (and any objects, in general) as vectors of
identifiers, such as, for example, index terms. It is used in information
filtering, information retrieval, indexing and relevancy rankings. Its first
use was in the SMART Information Retrieval System.
```

```
>> whos
  Name                          Size             Bytes  Class     Attributes

  A                            61x2               828  double    sparse
  dictionary                   61x12             1464  char
  files                         2x1               392  cell
  global_weights               61x1               488  double
  normalization_factors         2x1                16  double
  titles                        2x1              1808  cell
  update_struct                 1x1              7322  struct
  words_per_doc                 2x1                16  double
```

```
>> words_per_doc
words_per_doc =
    56
    52
```

no stemming
no stoplist

- VSM

```
>> A
A =

   (2,1)          3
   (4,1)          1
   (5,1)          1
   (6,1)          1
   ...
   (1,2)          1
   (2,2)          2
   (3,2)          1
  (11,2)          1
  (12,2)          1
  ...
  (57,2)          2
  (58,2)          1
  (60,2)          1
```

TF

```
>> dictionary(1:5,:)
ans =

algebraic
and
any
application
arrays

>> dictionary(55:60,:)
ans =
used
various
vector
vectors
very
was
```

```
>> dense=full(A);
>> size(dense)
ans =
    61      2
>> dense
dense =
     0      1
     3      2
     0      1
     1      0
     1      0
     1      0
     1      0
     1      0
     1      0
     1      0
     1      1
     0      1
     1      0
     0      1
     0      1
     0      2
     1      0
     1      0
     1      0
     0      1
     0      1
     ...
     0      1
     1      0
```


- VSM

```
>> dictionary
dictionary =
algebraic
and
any
application
arrays
articles
common
conjunctions
constructs
document
documents
example
existing
filtering
first
for
form
frequent
from
general
...
vectors
very
was
words
```

```
>> dense
dense =
     0      1
     3      2
     0      1
     1      0
     1      0
     1      0
     1      0
     1      0
     1      0
     1      0
     1      1
     0      1
     1      0
     0      1
     0      1
     0      2
     1      0
     1      0
     1      0
     0      1
     ...
     0      1
     3      0
     0      1
     1      0
```

- stoplist and stemming

```
o'neill criticizes europe on grants
treasury secretary paul o'neill expressed
irritation wednesday that european
countries have refused to go along with a
US proposal to boost the amount of direct
grants rich nations offer poor countries
the bush administration is pushing a plan
to increase the amount of direct grants the
world bank provides the poorest nations to
50 percent of assistance reducing use of
loans to these nations
```

```
o'neill criticizes europe grants treasury
secretary paul o'neill expressed irritation
european countries refused US proposal
boost direct grants rich nations poor
countries
bush administration pushing plan increase
amount direct grants world bank poorest
nations assistance loans nations
```

```
information -> inform
presidency -> presid
presiding       -> presid
happiness -> happi
happily -> happi
discouragement -> discourag
battles -> battl
```