Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 18.04.2020

7. Łańcuchy znaków

Łańcuchy znaków – wprowadzenie (1)

- W języku C/C++ brak jest specjalnego typu danych dla reprezentacji napisów. Każdy napis jest traktowany jako ciąg znaków. Przyjętą reprezentacją napisu jest tablica znaków.
- Standardowo, znak jest reprezentowany przez jeden bajt. Takie założenie było przez długie lata wystarczające. Liczba symboli graficznych wymaganych w aplikacjach języku angielskim doskonale mieści się w zakresie od 0-127. Pozostałe znaki były używane do reprezentacji znaków specjalnych (np.: elementów ramek)
- Języki europejskie wymagają dodatkowych znaków, którym przydzielono kody powyżej 127. Niestety, układ symbole graficznych poszczególnych grup języków może ze sobą kolidować. (np.: zachodnioeuropejskich i środkowoeuropejskich).

Łańcuchy znaków – wprowadzenie (2)

- Odwzorowanie wartości bajtów w postać symboli graficznych uzależnione jest od używanej *strony kodowej* .
- Strony kodowe ISO 8859-2 (norma) oraz Windows 1250 (źródło: Wikipedia)

ISO/IEC 8859-2:1999

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*Znaki kontrolne*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*Nieużywane*															
9x																
Ax	NBSP	Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź	SHY	Ž	Ż
Bx	°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
Cx	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
Dx	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
Ex	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
Fx	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

Windows-1250

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	€	*NZ*	‚	*NZ*	„	…	†	‡	*NZ*	‰	Š	‹	Ś	Ť	Ž	Ź
9x	*NZ*	‘	’	“	”	•	–	—	*NZ*	™	š	›	ś	ť	ž	ź
Ax	NBSP	ˇ	˘	Ł	¤	Ą	¦	§	¨	©	Ş	«	¬	SHY	®	Ż
Bx	°	±	˛	ł	´	µ	¶	·	¸	ą	ş	»	Ľ	˝	ľ	ż
Cx	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
Dx	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
Ex	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
Fx	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

Łańcuchy znaków – wprowadzenie (3)

- W językach azjatyckich zawsze posługiwano się większą liczbą znaków. Z tego powodu używa się reprezentacji mieszanej napisów zawierającej zarówno znaki jedno i dwubajtowe..
- Nowszym standardem jest standard UNICODE. Każdy znak jest reprezentowany przez 16-bitową liczbę bez znaku. Standard UNICODE pokrywa symbole graficzne rozmaitych języków i pozwala na ich równoczesne użycie.

Hex	Znak	Unicode	Hex	Znak	Unicode
0xA1	Ą	U+0104	0xB1	ą	U+0105
0xC6	Ć	U+0106	0xE6	ć	U+0107
0xCA	Ę	U+0118	0xEA	ę	U+0119
0xA3	Ł	U+0141	0xB3	ł	U+0142
0xD1	Ń	U+0143	0xF1	ń	U+0144
0xD3	Ó	U+00D3	0xF3	ó	U+00F3
0xA6	Ś	U+015A	0xB6	ś	U+015B
0xAF	Ż	U+017B	0xBF	ż	U+017C
0xAC	Ź	U+0179	0xBC	ź	U+017A

Łańcuchy znaków – wprowadzenie (4)

- W wersji podstawowej, tablice znaków języka C/C++ są ciągami 8-bitowych wartości. Odwzorowanie kodów znaków w symbole graficzne pozostawione jest parametrom sterującym aplikacją (np.: użytemu fontowi, stronie kodowej przyjętej dla systemu operacyjnego).
- Znakiem szczególnym jest znak o zerowej wartości. Nigdy nie ma on reprezentacji graficznej i pełni funkcję znacznika specjalnego (ang. *sentinel*).
- W języku C/C++ napisy reprezentowane są jako ciągi 8-bitowych znaków zakończone dodatkowym znakiem zerowym (znacznikiem końca).

Przykłady

- Stała `"Tekst"` jest reprezentowana jako tablica znaków umieszczona w segmencie danych.

T	e	k	s	t	0

- Deklaracja tablicy znakowej z inicjalizacją

  `char text[]="Tekst";`

  Kompilator automatycznie przydzieli tablicy `text` 6 znaków odpowiednio ustawiając znaki.

  `char text[256]="Tekst";`

  Kompilator przydzieli tablicy `text` 256 znaków. Pierwszych sześć znaków zostanie zainicjowanych, pozostałe będą miały wartość zero. W tablicy można umieszczać teksty zawierające 255 znaków (należy zarezerwować miejsce na ostatni znak **0**).

Funkcje działające na tablicach znakowych

- Funkcje działające na tablicach znakowych zdefiniowane są w pliku nagłówkowym `<string.h>`
- Większość z nich zakłada, że łańcuchy znakowe są zakończone znakiem zerowym.

Funkcja strlen

```
size_t strlen( const char *string );
```

Funkcja oblicza długość łańcucha znakowego.

```
unsigned mystrlen( const char *string )
{
        unsigned i;
        for(i=0;string[i];i++);
        return i;
}
```

Funkcja strcpy

```
char *strcpy( char *dest, const char *source );
```

- Funkcja kopiuje zawartość łańcucha znakowego `source` do tablicy `dest`.
- Tablica docelowa musi mieć rozmiar ≥ `strlen(source)+1`.
- Działanie funkcji w przypadku nakładania się tablic `source` oraz `dest` jest nieokreślone.

```
char *mystrcpy(char *dest, const char *source )
{
        int i;
        for(i=0; source[i];i++){
                dest[i]= source[i];
        }
        dest[i]= 0;
        return dest;
}
```

Funkcja strcat

```
char *strcat( char *dest, const char *source );
```

Funkcja dodaje na końcu łańcucha `dest` łańcuch `source` . Tablica docelowa musi mieć rozmiar
≥ `strlen(source)+ strlen(dest)+1`.

```
char *mystrcat(char *dest, const char *source )
{
        char*pdest;
        const char*psrc;
        for(pdest=dest;*pdest; pdest++);
        for(psrc = source; * psrc; psrc ++,pdest++){
                *pdest = *psrc;
        }
        *pdest = 0;
        return dest;
}
```

Funkcja strcmp

```
int strcmp( const char *str1, const char *str2 );
```

Funkcja porównuje zawartość dwóch tablic `str1` oraz `str2`. Zwraca:

- Wartość < 0 - jeżeli `str1` < `str2`
- 0 - jeżeli łańcuchy są identyczne
- Wartość > 0 - jeżeli `str1` > `str2`

```
int mystrcmp( const char *str1, const char *str2 )
{
	for(;*str1 && *str2;str1++,str2++){
		if(*str1<*str2)return -1;
		if(*str1>*str2)return 1;
	}
	if(*str1 && ! *str2) return 1; // aX > a
	if(!*str1 && *str2) return -1; // a < aX
	return 0;
}
```

Funkcja strcoll

```
int strcoll( const char *str1, const char *str2 );
```

Funkcja – podobnie jak `strcmp` - porównuje zawartość dwóch tablic `str1` oraz `str2`, ale interpretuje teksty zgodnie z ustawieniami regionalnymi (locale). Dzięki temu możliwe jest np.: porównanie polskich tekstów.

Funkcja `setlocale` ustala wszystkie (`LC_ALL`) lub wybrane składniki informacji regionalnych (`LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME`)

```
int main(){
    setlocale(LC_ALL,""); // setlocale(LC_ALL,"C")
    char*s1= "kąt";
    char*s2="kbt";
    printf("%s %d %s",s1,strcoll(s1,s2),s2);
    return 0;
}
```

```
setlocale(LC_ALL,"");

kąt -1 kbt
```

```
//setlocale(LC_ALL,"C");

k╣t 1 kbt
```

Wydzielanie symboli (1)

- Bardzo często w programach występuje konieczność analizy łańcuchów znaków i wydzielenia z nich symboli składowych (np.: słów, słów kluczowych, liczb).
- W szczególnym prostym przypadku tekst może być traktowany jako ciąg symboli oddzielonych separatorami.
- Oznaczmy:
  - *A* –zbiór znaków ASCII {1..255}
  - *S* – zbiór separatorów
  - Symbolami będą dowolne podłańcuchy zawierające znaki ze zbioru $A \setminus S$.
- Funkcją pozwalającą na wydzielenie tak zdefiniowanych symboli (ang. *token*) jest funkcja `strtok()` .

Wydzielanie symboli (2)

Funkcja strtok

```
char * strtok ( char* str, const char* delimiters );
```

- Wielokrotne wywołanie funkcji wydziela kolejne symbole z tesktu `str`.
- Podczas pierwszego wywołania do fukcji dostraczny jest wskaźnik `str`. Musi on wskazywać modyfikowalny tekst (funkcja umieszcza zera po kolejnych symbolach): `strtok(buf," \t\n.,")`
- Podczas kolejnych wywołań nie podaje się już adresu bufora (parametrem jest `NULL`): `strtok(NULL," \t\n.,")`
- Zbiory separatorów `delimiters` mogą być różne dla kolejnych wywołań.
- Funkcja zwraca `NULL` (`0`), jeżeli nie ma już więcej symboli do wydzielenia

Wydzielanie symboli (3)

Przykład

```
#include <string.h>
int main()
{
  char buf[255]; // bufor dla funkcji strtok
  const char text[]="To jest\ttekst. Słowa są"
  "oddzielone\nbiałymi znakami";
  char *ptr;
  const char sep1[]=" \n\t.,:!?";
  const char sep2[]=".?!";
  // Słowa
  strcpy(buf,text);  // strtok niszczy bufor!
  for( ptr=strtok(buf,sep1);       // inicjalizacja
       ptr;                        // czy wydzielono symbol?
       ptr=strtok(NULL,sep1))      // nast. symbol
       {
              printf("%s\n",ptr);
       }
```

Wydzielanie symboli (4)

Kontynuacja…

```
        // Zdania
        //  const char sep2[]=".?!";
        strcpy(buf,text);
        for(   ptr=strtok(buf,sep2);
                        ptr;
                        ptr=strtok(NULL,sep2)){
               printf("%s\n",ptr);
        }
        return 0;
}
```

Inne funkcje

- Istnieją wersje funkcji , które ograniczają porównanie, kopiowanie do określonej liczby znaków:
  - `int strncmp ( const char * str1, const char * str2, size_t num );`
  - `char * strncpy ( char * destination, const char * source, size_t num );`
- Funkcje do porównywania mogą ignorować duże małe litery: `strcasecmp()`. Standardowo, znaki diakrytyczne nie są poddawane translacji, stąd `strcasecmp("trąba","TRĄBA") ≠ 0`. To zachowanie może być zmienione przez ustawienie *locale*, zmiennej odpowiedzialnej za określenie rodzaju języka.
- Istnieje szereg funkcji, które działają na tablicach bajtów, ale nie zakładają, że są one tekstami zakończonymi znakiem `0`. Ich dodatkowym parametrem jest zawsze wielkość tablicy: `memXXX()`:
  - `void * memcpy ( void * destination, const void * source, size_t num );`
  - `int memcmp ( const void * ptr1, const void * ptr2, size_t num );`

Wyszukiwanie znaków

Do wyszukiwania znaków w tekście służy funkcja `strchr()` zadeklarowana jako:

```
const char * strchr ( const char * str, int c );
```

- str – przeszukiwany tekst
- c – szukany znak

Funkcja zwraca wskaźnik do pierwszego wystąpienia znaku w tekście lub wskaźnik zerowy, jeżeli znaku nie odnaleziono.
Istnieje też wersja bez `const` (wykorzystana poniżej) oraz funkcja wyszukująca znaki od końca: `strrchr()`.

```
int main(){
    char filename[]="c:\\Users\\jan kowalski\\Documents\\zadanie.docx";
    for(char*ptr=strchr(filename,'\\');ptr;ptr=strchr(ptr+1,'\\')){
        *ptr='/';
    }
    printf("%s",filename);
}
```

```
c:/Users/jan kowalski/Documents/zadanie.docx
```

Wyszukiwanie znaków

Typowy przykład: sortujemy słowa według liczby samogłosek.

Funkcja `count_vowels()` zwraca liczbę samogłosek.

```
int count_vowels(const char*txt){
    int cnt=0;
    while(*txt){
        if(strchr("aeiouyAEIOUY",*txt))cnt++;
        txt++;
    }
    return cnt;
}
```

Wywołanie qsort 1

```
int cmp_words(const void*a,const void*b){
    return count_vowels(*(const char**)a)-count_vowels(*(const char**)b);
}

int main(){
    const char*words[]={"alpha","queueing","stack","myth","asthma","beta"};
    int n = sizeof(words)/sizeof(words[0]);
    qsort(words,n,sizeof(const char*),cmp_words);
    for(const char**ptr=words;ptr<words+n;ptr++)
        printf("%s ",*ptr);
}
```

```
stack myth alpha asthma beta queueing
```

Sortujemy wskaźniki do stałych łańcuchowych umieszczonych w tablicy.

- Funckja `qsort()` przekazuje do `cmp_words()` dwa wskaźniki typu `void*`, których wartości są adresami elementów tablicy -- a są nimi wskaźniki typu `const char*`.
- Pod wskaźnikami `void*` kryja się więc wskaźniki typu `const char**` i na taki typ rzutujemy, a następnie stosujemy dereferencję, `*(const char**)a` dzięki której otrzymujemy wskaźniki do tekstów `const char*`.

Wywołanie qsort 2

```
int cmp_words_in_2D_array(const void*a,const void*b){
    return count_vowels((const char*)a)-count_vowels((const char*)b);
}

int main(){
#define WORD_LEN 20
    char words[][WORD_LEN]={"alpha","queueing","stack","myth","asthma","beta"};
    int n = sizeof(words)/sizeof(words[0]);
    qsort(words,n,WORD_LEN,cmp_words_in_2D_array);
    for(const char(*ptr)[WORD_LEN]=words;ptr<words+n;ptr++)
        printf("%s ",ptr);
}
```

```
stack myth alpha asthma beta queueing
```

Tym razem sortujemy dwuwymiarową tablicę mającą `WORD_LEN=20` kolumn. Podczas sortowania są zamieniane miejscami 20-elementowe wiersze. Funkcja do porównywania otrzymuje jako argumenty wskaźniki na teksty w dwóch wierszach (czyli wskaźniki `[const] char*`).

- W funkcji `cmp_words_in_2D_array()` rzutujemy na `const char*`.
- Proszę zwrócić uwagę jak wygląda wydruk. Wskaźnik `ptr` „wie” ile tablica ma kolumn i `ptr++` przestawia go na następny wiersz.

Wyszukiwanie tekstu

```
int search(const char*text, const char*pattern, int start){
    int n = strlen(pattern);
    int m=strlen(text);
    for(int i=start;i<m-n;i++){
        if(strncmp(text+i,pattern,n)==0)return i;
    }
    return -1;
}


int main(){
    const char*text =  "tak, nie, niebieski, niedrogo, zaniemówił";

    for(int i=search(text,"nie",0);i>=0;i=search(text,"nie",i+1)){
        printf("%d ",i);
    }
}
```

```
5 10 21 33
```

Wyszukiwanie tekstu metodą naiwną.

Funkcja strstr

```
char* strstr (const char* str, const char* pattern);
```

Parametry:

`str` – przeszukiwany tekst

`pattern` – szukany tekst

Funkcja zwraca wskaźnik do znaku wewnątrz `str`, dla którego nastąpiło dopasowanie do `pattern` lub 0 (NULL) jeżeli łańcucha `pattern` nie znaleziono.

```
int main(){
    const char*text =  "tak, nie, niebieski, niedrogo, zaniemówił";

    for(char*ptr=strstr(text,"nie");ptr!=0;ptr=strstr(ptr+1,"nie")){
        printf("%d ",ptr-text);
    }
}
```

```
5 10 21 33
```

Wyszukiwanie tekstu w ciągu znaków

```
void search_and_mark(const char*pattern){
    int cnt=0;
    int size = strlen(pattern);
    char text[size+1];
    for(;;){
        int c = getchar();
        if(c<0)break;
        if(cnt<size){
            text[cnt++]=c;
        }else{
            putchar(text[0]);
            memmove(text,text+1,cnt-1);
            text[cnt-1]=c;
        }
        if(strncmp(text,pattern,size)==0){
            printf("[%s]",pattern);
            cnt=0;
        }
    }
    text[cnt]=0;
    printf("%s",text);
}
```

Użyjemy dwóch funkcji:

- `int getchar()` – czyta znak ze standardowego wejścia, zwraca wartość -1, jeżeli napotkano koniec strumienia
- `int putchar(int c)` – kieruje znak do standardowego wyjścia

Wywołanie:

```
search_and_mark("int");
```

Wejście:

```
int main(){
     int a=5;
     printf("%d\n",a);
}
```

```
[int] main(){
   [int] a=5;
   pr[int]f("%d\n",a);
}
```

Zamiana tekstu w ciągu znaków

```
void search_and_replace(const char*pattern,const char*replace){
    int cnt=0;
    int size = strlen(pattern);
    char text[size+1];
    for(;;){
        int c = getchar();
        if(c<0)break;
        if(cnt<size){
            text[cnt++]=c;
        }else{
            putchar(text[0]);
            memmove(text,text+1,cnt-1);
            text[cnt-1]=c;
        }
        if(strncmp(text,pattern,size)==0){
            printf("%s",replace);
            cnt=0;
        }
    }
    text[cnt]=0;
    printf("%s",text);
}
```

Wywołanie:

```
search_and_replace("int",
"int32");
```

Wejście

```
int main(){
     int a=5;
     printf("%d\n",a);
}
```

Wyjście:

```
int32 main(){
     int32 a=5;
     print32f("%d\n",a);
}
```

Odległość Hamminga

Odległość Hamminga to liczba pozycji, na których dwa teksty różnią się (w zasadzie teksty równej długości). Czyli jeden tekst można przekształcić w drugi przez zamianę znaków.

```
Ala ma kota
Ola ma psa
```

Pomijamy ostanie a – wtedy odległość wynosi 4.

```
int hamming_distance(const char*a,const char*b){
    int d=0;
    while(*a && *b){
        if(*a!=*b)d++;
        a++;
        b++;
    }
    return d;
}

int main(){
    const char*a="Ala ma kota";
    const char*b="Ola ma psa";
    printf("hd=%d\n",hamming_distance(a,b));
}
```

```
hd=4
```

Najdłuższy wspólny podciąg

Najdłuższy wspólny podciąg (ang. longest common subsequence, LCS) to miara odległości ciągów dopuszczająca wstawienia i usunięcia.
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Doskonała miara do sprawdzania plagiatów, oceny podobieństwa kodu przesyłanego do automatycznej oceny, itp..

`Dr_Jan_Kowalski` `Jan_Maciej_Kowalczyk`	**Długość wynosi 10.** **Pierwszy ciąg można przekształcić w drugi przez:** • **Usunięcie:** `Dr_ s i` • **Wstawienie:** `Maciej_ czy`

Definicja jest rekurencyjna. Przez X_i,$\mathrm{Y_j}$ oznaczymy początkowe podciągi elementów $\mathrm{X_i} = (x_1, \dots, x_i)$, $Y_j = (y_1, \dots, y_j)$

$$lcs(X_i, Y_j) = \begin{cases} 0 \; gdy \; i = j = 0 \\ lcs(X_{i-1,} Y_{j-1}) + 1 \; gdy \; x_i = y_j \\ \max\left(lcs(X_{i,} Y_{j-1}), lcs(X_{i-1,} Y_j)\right) gdy \; x_i \neq y_j \end{cases}$$

Implementacja rekurencyjna

```
int lcs_cnt=0;

int lcs(const char*a,const char*b,int m, int n){
    lcs_cnt++;
    if(m==0||n==0)return 0;
    if(a[m-1]==b[n-1]) return 1+lcs(a,b,m-1,n-1);
    int len1 = lcs(a,b,m,n-1);
    int len2 = lcs(a,b,m-1,n);
    return len1>len2?len1:len2;
}

int main(){
    const char*a="dr Jan Kowalski";
    const char*b="Jan Maciej Kowalczyk";

    clock_t start = clock();
    int l = lcs(a,b,strlen(a),strlen(b));
    clock_t end = clock();
    double t = (double)(end-start)/CLOCKS_PER_SEC;
    printf("time=%fs cnt=%d ",t,lcs_cnt);
    printf("lcs=%d\n",l);
}
```

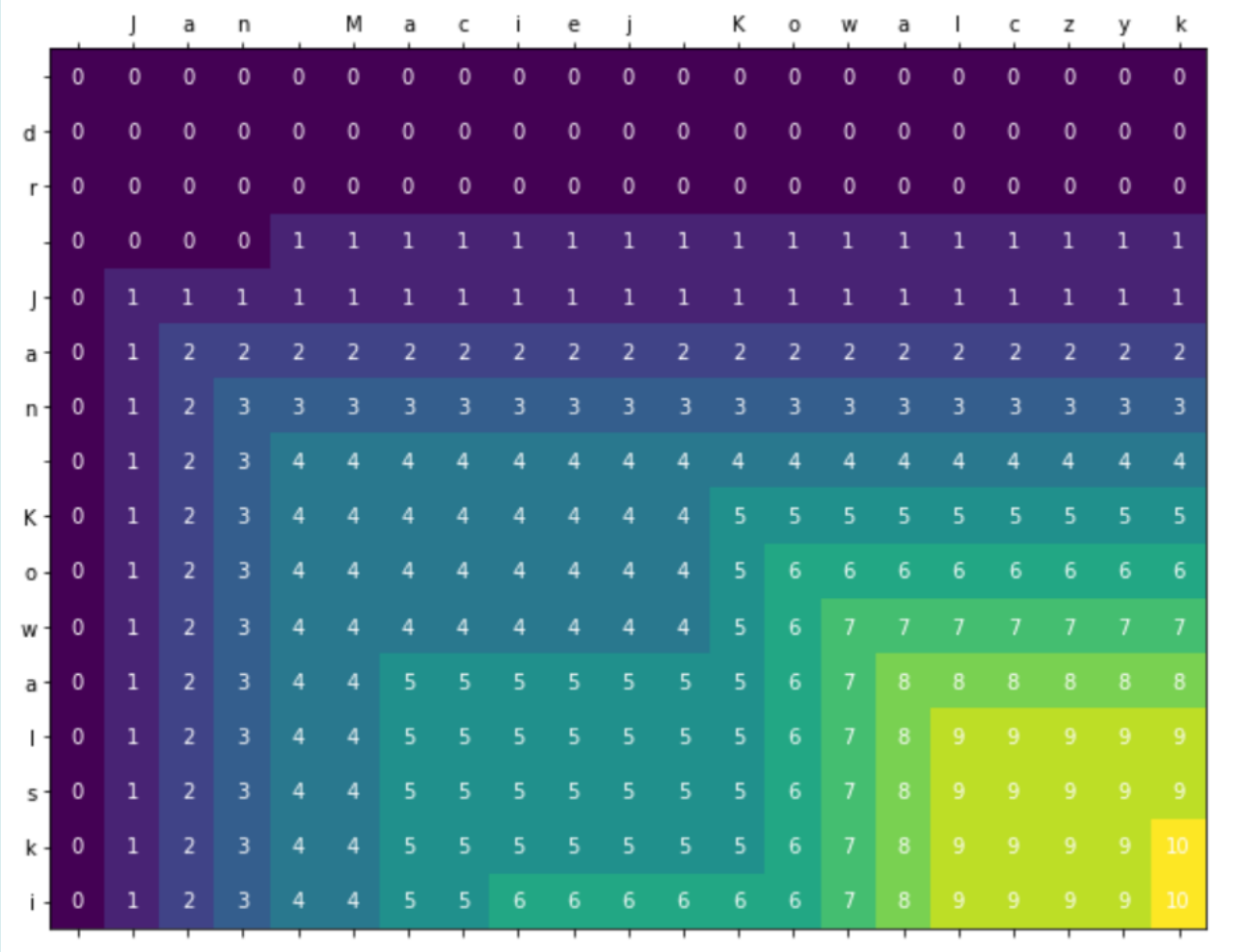
Bardzo niewydajna

```
time=1.390000s
cnt=568918449 lcs=1
```

Funkcja `lcs()` została wywołana 568 mln. razy

Implementacja z macierzą

Odległość $lcs(X_i, Y_j)$ jest obliczana tylko raz i umieszczana w macierzy o wymiarach $m + 1 \times n + 1$. Kolejne wartości obliczane są na podstawie wcześniejszych. Jest to typowy przykład programowania dynamicznego.



Implementacja z macierzą

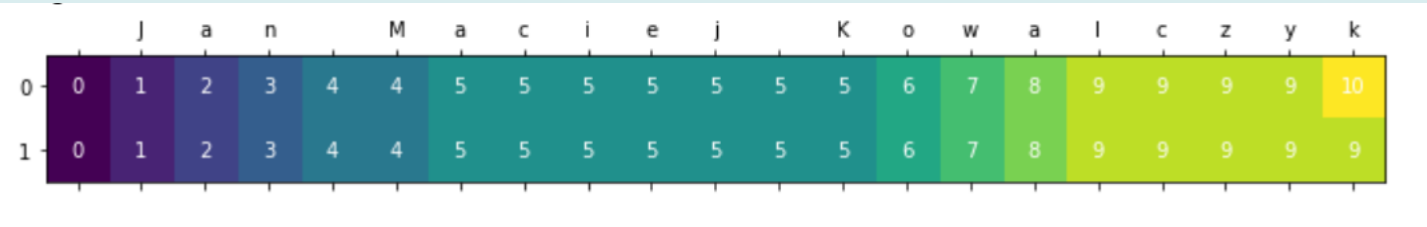
```
int lcs2(const char*a,const char*b){
    int m = strlen(a);
    int n = strlen(b);
    // Alokacja tablicy d[m+1][n+1]
    int (*d)[n+1]=malloc((n+1)*(m+1)*sizeof(int));
    for(int i=0;i<m+1;i++){
        for(int j=0;j<n+1;j++){
            if(i==0 || j==0){
                d[i][j]=0;
                continue;
            }
            if(a[i-1] == b[j-1]){
                d[i][j]=d[i-1][j-1]+1;
            }else{   // max
                d[i][j]=d[i-1][j] > d[i][j-1] ? d[i-1][j] : d[i][j-1];
            }
        }
    }
    int max = d[m][n];
    free(d);
    return max;
}
```

Wypełnianie zerami pierwszego wiersza i kolumny

Obliczanie długości na podstawie wcześniejszych podciągów

Macierz o dwóch wierszach

Nie jest konieczne przechowywanie całej macierzy o rozmiarach $m + 1 \times n + 1$. Wystarczy $2 \times n + 1$



Na przemian wypełniane będą wiersze o indeksach 0 i 1.

Wartości w bieżącym wierszu `idx` będą wypełniane na podstawie zawartości wiersza o indeksie `1-idx`.

Macierz o dwóch wierszach

```
int lcs3(const char*a,const char*b){
    int m = strlen(a);
    int n = strlen(b);
    // Alokacja tablicy d[2][n+1]
    int (*d)[n+1]=malloc((n+1)*(2)*sizeof(int));
    int idx;
    for(int i=0;i<m+1;i++){
        idx=i&1; // indeks bieżącego wiersza...
        for(int j=0;j<n+1;j++){
            if(i==0 || j==0){d[idx][j]=0; continue;}
            if(a[i-1] == b[j-1]){
                d[idx][j]=d[1-idx][j-1]+1;
            }else{
                d[idx][j] = d[1-idx][j] > d[idx][j-1] ?
                        d[1-idx][j] : d[idx][j-1]; // max
            }
        }
    }
    int max = d[idx][n]; // zwracana wartość w bieżącym wierszu...
    free(d);
    return max;
}
```

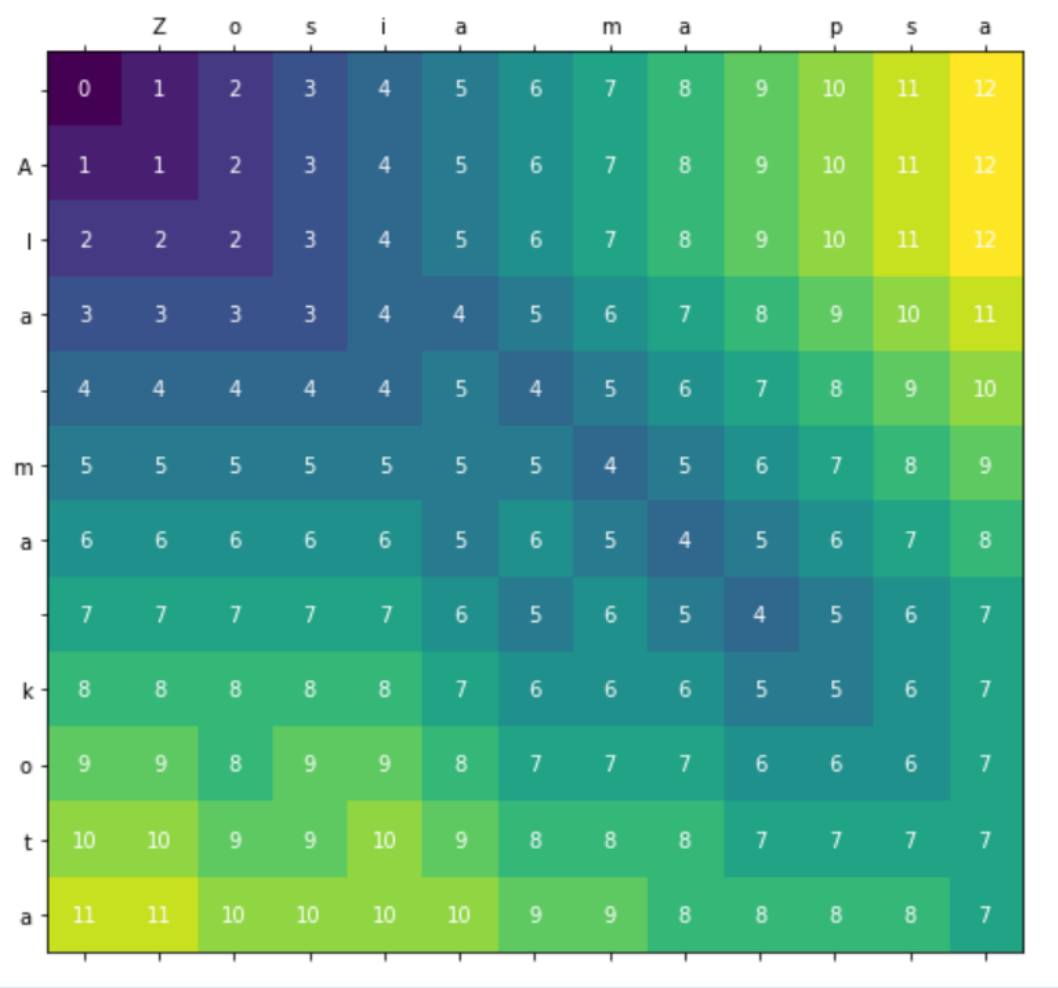
Macierz o dwóch wierszach

```
int main(){
    const char*a="Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i "
                 "ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. "
                 "Dz.U. z 2006 r. Nr 90, poz. 631 z pó´zn. zm.): „Kto przywłaszcza sobie autorstwo "
                 "albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu "
                 "albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolnści "
                 "albo pozbawienia wolności do lat 3.";

    const char*b="Uprzedzony o odpowiedzialności karnej na podstawie "
                 "ustawy o prawie autorskim i prawach pokrewnych „Kto przywłaszcza sobie autorstwo "
                 "albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu "
                 "albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolnści "
                 "albo pozbawienia wolności do lat 3.";



    clock_t start = clock();
    int l = lcs3(a,b);
    clock_t end = clock();
    double t = (double)(end-start)/CLOCKS_PER_SEC;
    printf("time=%fs ",t);
    printf("lcs=%d\n",l);

}
```

```
time=0.000000s lcs=326
```

Odległość Levenshteina

Odległość Levenshteina zdefiniowana jest jako minimalna liczba zamian, wstawień i usunięć pozwalająca przeprowadzić jeden ciąg w drugi.



		Z	o	s	i	a		m	a		p	s	a
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	1	1	2	3	4	5	6	7	8	9	10	11	12
l	2	2	2	3	4	5	6	7	8	9	10	11	12
a	3	3	3	3	4	4	5	6	7	8	9	10	11
	4	4	4	4	4	5	4	5	6	7	8	9	10
m	5	5	5	5	5	5	5	4	5	6	7	8	9
a	6	6	6	6	6	5	6	5	4	5	6	7	8
	7	7	7	7	7	6	5	6	5	4	5	6	7
k	8	8	8	8	8	7	6	6	6	5	5	6	7
o	9	9	8	9	9	8	7	7	7	6	6	6	7
t	10	10	9	9	10	9	8	8	8	7	7	7	7
a	11	11	10	10	10	10	9	9	8	8	8	8	7

Podobnie, jak dla LCS można zrealizować algorytm z pełną lub dwuwierszową macierzą.

W tym przypadku obliczana jest odległość, a nie podobieństwo.

Wydruk tablicy

```
void print_mat(int n, int (*d)[n], int m,
                                const char*rows,const char*cols){
    printf("        ");
    for(int j=0;j<n;j++){
        printf("%3c ",cols[j]);
    }
    printf("\n");
    for(int i=0;i<m;i++){
        if(i>0)printf("%3c ",rows[i-1]);
        else printf("    ");
        for(int j=0;j<n;j++){
            printf("%3d ",d[i][j]);
        }
        printf("\n");
    }
}
```

n – liczba kolumn: przed deklaracją wskaźnika, który „zna” liczbę kolumn

m – liczba wierszy

rows,cols – teksty z nagłówkami wierszy i kolumn

Algorytm

```
int levenshtain_distance(const char*a,const char*b){
    int m = strlen(a);
    int n = strlen(b);
    // Alokacja tablicy d[m+1][n+1]
    int (*d)[n+1]=malloc((n+1)*(m+1)*sizeof(int));

    for(int i=0;i<m+1;i++)d[i][0]=i;
    for(int j=0;j<n+1;j++)d[0][j]=j;

    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            int subst_cost = 1;

            if(a[i-1] == b[j-1])subst_cost = 0;
            d[i][j]=min(d[i-1][j-1]+subst_cost,d[i-1][j]+1,d[i][j-1]+1);
        }
    }
    int max = d[m][n];
    print_mat(n+1,d,m+1,a,b);
    free(d);
    return max;
}
```

```
int min(int a, int b, int c){
    if(a<b && a<c)return a;
    if(b<a && b<c)return b;
    if(c<a && c<b)return c;
}
```

Wywołanie

```
int main(){
    const char*a="Ala ma kota";
    const char*b="Zosia ma psa";
    printf("\nldist=%d\n",levenshtain_distance(a,b));
}
```

```
          Z   o   s   i   a       m   a       p   s   a
      0   1   2   3   4   5   6   7   8   9  10  11  12
  A   1   1   2   3   4   5   6   7   8   9  10  11  12
  l   2   3   2   3   4   5   6   7   8   9  10  11  12
  a   3   3   3   3   4   4   5   6   7   8   9  10  11
      4   5   6   7   4   5   4   5   6   7   8   9  10
  m   5   5   6   7   5   5   5   4   5   6   7   8   9
  a   6   7   6   7   6   5   6   5   4   5   6   7   8
      7   7   7   7   7   6   5   6   5   4   5   6   7
  k   8   9  10  11  12   7   6   6   6   5   5   6   7
  o   9   9   9  10  11   8   7   8   9   6   7   6   7
  t  10  11  12  10  11   9   8   8   9   7   7   7   7
  a  11  11  12  11  11  10   9  10   8   8   9  10   7

ldist=7
```

Szerokie znaki (1)

- Programy w języku C mogą posługiwać się rozszerzoną reprezentacją znaków (w zasadzie zgodną ze standardem Unicode).
- W pliku nagłówkowym `<wchar.h>` zdefiniowano:
  - nowy typ znakowy `wchar_t` – odpowiadający `short`
  - oraz kilkadziesiąt funkcji obejmujących:
    - Formatowane wejście wyjście (`wprintf, putwchar, swscanf`)
    - Funkcje konwersji: `wcstod, wcstof, wcstol`
    - Funkcje manipulujące łańcuchami znaków (utrzymano zasadę, że zerowy szeroki znak kończy łańcuch).

Szerokie znaki (2)

- Przykład

```
int main(){
    wchar_t tab[100]=L"Ala ma kota";
    setlocale(LC_ALL,"");
    printf("Tablica zajmuje %d bajtów\n",sizeof(tab));
    wprintf(L"Zawarto\u015b\u0107 tablicy: %ls\n",tab);
    printf("Długość tablicy: %d\n",wcslen(tab));

    wcscat(tab, L" i  psa");
    wprintf(L"Zawarto\u015b\u0107 tablicy\
            po konkatenacji: %ls\n",tab);
    return 0;
}
```

```
Tablica zajmuje 200 bajtów
Zawartość tablicy: Ala ma kota
Długość tablicy: 11
Zawartość tablicy po konkatenacji: Ala ma kota i  psa
```

Typowe błędy

- Najczęściej spotykane błędy przy posługiwaniu się funkcjami działającymi na tablicach znakowych to użycie wskaźnika, który zawiera adres nieokreślony lub przekroczenie rozmiarów tablicy.
- Język C/C++ nie ma żadnych wbudowanych mechanizmów ochrony przed tego typu błędami. Rozmiary tablicy nie są sprawdzane w trakcie wykonania.

```
char *table1;
strcpy(table1,”Tekst”);

char *table2;
strcmp(table2, ”Tekst”);

char table3[]=”Ala ma ” ;
strcat(table3,”kota”);

char *table4 =”x=%d”;
strcpy(table4,”Txt”); /* błąd, modyfikujemy pamięć, do
której nie mamy dostępu do zapisu */
```

Typowe błędy

- Inny częsty błąd to zapominanie o znaku 0 na końcu – przy implementacji algorytmów przetwarzających tekst lub przy alokacji pamięci.

```
int main(){
    const char*txt = "Ala ma kota";
    char *ptr = malloc(strlen(txt));// strlen(txt)+1 !!!
    for(int i=0;i<strlen(txt);i++){
        ptr[i]=txt[i];
    }
    printf("%s\n",ptr);
    free(ptr);

}
```

Ten program przypadkowo zadziała ale:

- Brakuje miejsca na 0 na końcu
- Nie dodano 0 na końcu
- Jeżeli przypadkowo byłaby tam inna wartość, wydrukowane zostaną dziwne znaki…