POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

NAZWA PRZEDMIOTU

Programowanie strukturalne

Temat wykładu 6.

Stałe, zmienne lokalne i globalne. Zasięg i zakres zmiennych. Operatory arytmetyczne, relacyjne, logiczne i bitowe.

dr hab. inż. Jerzy Montusiewicz, prof. PL

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

6. Agenda

6.1. Stałe.

6.2. Zmienne lokalne i globalne.

6.3. Zasięg i zakres zmiennych.

6.4. Operatory arytmetyczne.

6.5. Operatory relacyjne, logiczne.

6.6. Operatory bitowe.

6.1. Stałe, p6-1a

Stałe liczbowe:

- liczbowe całkowite dziesiętne (ciągi cyfr ze znakiem lub bez),
- liczbowe całkowite dziesiętne (mogą równie być zapisywane w formacie 16; <podstawa 16> mogą to być cyfry od **0** do **9** oraz litery od **a** do **f** lub od **A** do **F**), poprzedzamy prefiksem ”0x”, np. **0xFF**,
- format ósemkowy stałych całkowitych <podstawa 8>, cyfry od **0** do **7**; prefiks w postaci ”0”, np. **0177**,
- stałe liczbowe zmiennoprzecinkowe, np. **123.45**,
  lub w formacie wykładniczym, np. **54.32e-5** lub **123e+6**,

  Domyślnie stałe zmiennoprzecinkowe są traktowane jako stałe **double**, a konwersja na **float** wykonywane jest domyślnie lub na skutek jawnego rzutowania typu, np.

  (**float**)a/3 , gdzie mamy **int** a.

6.1. Stałe

Stałe znakowe

to pojedyncze znaki ASCII ujmowane standardowo w apostrofy, np.

‘a’ lub **‘\n’** czy **‘7’**

*Pamiętaj rozróżniamy kod ASCII cyfry siedem od liczby **7**.*

Stałe napisowe

to łańcuchy znakowe ograniczone obustronnie cudzysłowem, np.

”Pomagam bo warto”

Te stałe podobnie jak napisy są automatycznie uzupełnianie przez terminator w postaci bajtu zerowego.
Tak więc mamy różnicę między **‘a’** oraz **”a”**.

6.1. Stałe

Stałe znakowe z odwróconym ukośnikiem.

- **'\b'** – cofniecie (ang. backspace),
- **'\f'** – nowa strona (ang. form feed),
- **'\n'** – nowy wiersz (ang. new line),
- **'\r'** – powrót karetki (ang. return),
- **'\t'** – tabulator poziomy,
- **'\"'** – cudzysłów,
- **'\''** – apostrof,
- **'\a'** – alarm,
- **'\0'** – bajt zerowy (nullbajt).

6.1. Stałe, p6-1b

Definiowanie stałych za pomocą polecenia preprocesora oraz zmiennych z użyciem modyfikatora "const".

```
#include<stdio.h>
#define INCREMENT  11
int main()
{  const int one=11, two=99;
   double three=9.99;
   unsigned short sh_int;
/*         two = 99;  */
/*         one = one + INCREMENT; */
           printf ("one = %d; two = %d\n", one, two);
           printf ("three = %lf\n", three);
   sh_int = 65538; // max 65536
           printf ("sh_int = %d\n", sh_int);
   return 0; }
```

```
one   = 11; two   = 99
three  = 9.990000
sh_int  = 2
```

6.2. Zmienne lokalne i globalne

Typy podstawowe i wartości.

Pierwszym znakiem **nazwy** musi być litera lub znak podkreślenia ”_”. Kolejne znaki mogą być literami, cyframi lub podkreśleniem. Długość identyfikatora (nazwy) do 255 znaków (zależy od kompilatora).

Ma znaczenie wielkość liter: **ala** to nie to samo co **ALA**.

Dla platformy Uniksowej

Typ zmienej	Długość (bity)	Wart. min.	Wart. max.
Char	8	-128	127
short int	16	-32768	32767
int	32	-2147483648	2147483647
long int	64	$-9{,}22 \cdot 10^{18}$	$9{,}22 \cdot 10^{18}$
float	32	$1{,}17549435 \cdot 10^{-38}$	$3{,}40282347 \cdot 10^{+38}$
double	64	$2{,}2250738 \cdot 10^{-308}$	$1{,}79769313 \cdot 10^{+308}$
void	0	--	--

6.2. Zmienne lokalne i globalne, p6-2a

Zmienne "short int" ze znakiem (domyślne) i bez znaku (unsigned).

```
#include<stdio.h>
int main()
{  short int sign_num;
   unsigned short int unsign_num;
   unsign_num = 60000;
   sign_num = unsign_num;
   printf ("signed = %d; unsigned = %u\n", sign_num, unsign_num);
         return 0;}
```

```
signed = -5536; unsigned = 60000

--------------------------------
```

Dla zmiennej typu unsignum mamy:

32768 (część dodat.) + **32767** (część ujemna.) + '**0**' = **65536**

W przypadku liczby typu signum interpretowana jest jako liczba 16-bitowa w systemie uzupełnień do 2 stąd mamy liczbę ujemną równą **-5536**

6.2. Zmienne lokalne i globalne, p6-2b

Zastosowanie zmiennej typu char jako zmiennej sterującej w pętli ’for’. Druk / wyświetlanie alfabetu – duże litery.

```
#include<stdio.h>
int main()
{ char litera;
 for (litera='A';litera<='Z';litera++)  printf ("%c ", litera);
 printf ("\nKody ASCII liter od A do J\n");
 for (litera='A';litera<='J';litera++)  printf ("%d   ", litera);
 printf ("\n");
 for (litera='A';litera<='J';litera++)  printf ("%#x ", litera);
 printf ("\n");
 for (litera='A';litera<='J';litera++)  printf ("%x   ", litera);
 printf ("\n");
 return 0; }
```

6.2. Zmienne lokalne i globalne, p6-2b

W pętli **for** zmienną kontrolną jest zmienna ***litera*** typu **char**.
Inicjowana jest kodem ASCII *litery* **A**, ograniczeniem *litera* **Z**.
W sekcji przyrostu instrukcji for zmienna litera jest inkrementowana (zwiększana) jak zwykła zmienna całkowitoliczbowa.
%c – znakowy kod formatujący

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Kody ASCII liter od A do J
65   66   67   68   69   70   71   72   73   74
0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4a
41   42   43   44   45   46   47   48   49   4a


--------------------------------
```

6.2. Zmienne lokalne i globalne

W programowaniu w językach strukturalnych stosujemy zasadę modularyzacji oprogramowania i unikania zmiennych globalnych.

Zmienne globalne są:

- deklarowane na zew. funkcji składających się na tworzony program,
- dostępne dla wszystkich funkcji.

NIBY DOBRZE, ALE NIE KONIECZNIE

Wady:

- możliwa niezamierzona modyfikacja (przesłanianie zmiennej globalnej przez lokalne zmiennej o identycznych nazwach),
- funkcje korzystających z takich zmiennych są niezbyt ogólne,
- Zmienne takie są zmiennymi statycznymi (zajmują miejsce w pamięci przez cały czas działania programu, kłopot przy dużych tablicach).

6.2. Zmienne lokalne i globalne, p6-2c

Zakres widzialności zmiennych lokalnych i globalnych.
Przesłanianie zmiennej globalnej przez zmienną lokalną o tej samej nazwie.

```
#include<stdio.h>
int suma; /* zmienna globalna */
void total(int x);
void lokalna_suma (void);
void display (void);

int main()
{ int oblicz;  /* zmienna lokalna widoczna tylko w "main() */
        suma = 0;    /* inicjalizacja zm. globalnej */
        for (oblicz=0; oblicz<6; oblicz++)
        {       total(oblicz);
                lokalna_suma();
                display();      }
        return 0; }
```

Prototypy funkcji – wskazane aby funkcje nie zwracały domyślnie wartości „**int**".

6.2. Zmienne lokalne i globalne

```
void total(int x)
{  suma = x + suma; /* inkrementacja zm. globalnej "suma" */ }

void lokalna_suma (void)
{  int suma;       /* zmienna lokalna "suma" przeslania zmienną globalną */
   suma = 77;
   printf ("  wew. lokalna_suma(): suma = %d\n", suma);
}

void display (void)
{  int oblicz;  /* zmienna lokalna widoczna tylko w "display" */
   for (oblicz=0; oblicz<10; oblicz++)
            printf (".");
   printf ("aktualna globalna suma = %d\n", suma);
}
```

6.2. Zmienne lokalne i globalne

10 kropek wynika z funkcji
void dispay (void)

77 wynika z funkcji
void lokalna_suma (void)

```
..........aktualna globalna suma = 0
  wew. lokalna_suma(): suma = 77
..........aktualna globalna suma = 1
  wew. lokalna_suma(): suma = 77
..........aktualna globalna suma = 3
  wew. lokalna_suma(): suma = 77
..........aktualna globalna suma = 6
  wew. lokalna_suma(): suma = 77
..........aktualna globalna suma = 10
  wew. lokalna_suma(): suma = 77
..........aktualna globalna suma = 15
```

aktualna suma wynika z funkcji
void total (int x)

6.3. Zasięg i zakres zmiennych, p3-3a1

Zasięg, a z tego wynikająca „żywotność” zmiennej (zmiennych, tablic) może być ograniczona do:

- bloku instrukcji,
- **frazy if** instrukcji **if-else-if** lub **frazy else**,
- instrukcji typu pętla.

Zmienna **suma1** zadeklarowana **we frazie else**.

Próba wydruku zawartości zmiennej suma1 **poza frazą else**.

```
else {   int suma1=-1;  // zmienna we frazie else
         suma=suma1;
         printf("suma = %d, ",suma);
     }
printf("suma1 = %d", suma1);
```

[Error] 'suma1' undeclared (first use in this function)

6.3. Zasięg i zakres zmiennych, p3-3a2

Zasięg i zakres zmiennych. Obliczanie średniej arytmetycznej liczb>5.

```
int main()
{ int al1=0, al2=0;  //  int suma = 0;
  for (al1; al1<11; al1+=2)
  { int suma;
    if (al1>5)  {  suma+=al1;
      printf("suma = %d\n", suma);
      al2++;
      float average=0;
      average=(float)suma/al2;
      printf("wartosc sredniej = %f\n", average);  }
    else {  int suma1=-1;  // zmienna we frazie else
      suma=suma1;
      printf("suma = %d, ",suma);  }
      //   printf("suma1 = %d", suma1);
  }
  return 0;
}
```

```
suma = -1, suma = -1, suma = -1, suma = 5
wartosc sredniej = 5.000000
suma = 13
wartosc sredniej = 6.500000
suma = 23
wartosc sredniej = 7.666667
```

6.4. Operatory arytmetyczne

Zmianę kolejności działań na wszystkich poziomach uzyskujemy przez użycie nawiasów ”**( )**”.
Gdy występują operatory o tym samym priorytecie to działania wykonywane są od lewej do prawej.

Operator	Dzialanie
-	**odejmowanie, także unarny minus**
+	**dodawanie**
*	**mnożenie**
/	**dzielenie**
%	**modulo – reszta z dzielenia całkowitoliczbowego**
--	**dekrementacja (zmniejszenie o 1)**
++	**inkrementacja (zwiększenie o 1)**

6.4. Operatory arytmetyczne, p6-4a

Dzielenie argumentów całkowitoliczbowych oznacza obcięcie części ułamkowej i zaokrąglenie ilorazu do najbliższej mniejszej co do modułu wartości całkowitej, np.: z **5/9** otrzymamy **0,** z **7/2 → 3**.

Inkrementacja i dekrementacja najczęściej stosowana jest do zmiennych **int** , ale można ją stosować również do zmiennych **float**.

Operatory można stawiać jako **prefiksowe** lub **postfiksowe** (sufiksowe).

Przykład: x = 10; (patrzymy na priorytet);

y = ++x; **x** i **y** otrzymają wartości **11**.

```
x = 10;
y = x++;
```

y uzyskuje wartość **10** (wartość przed inkrementacją), zaś **x** wartość **11** .

6.4. Operatory arytmetyczne, p6-4a

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int a,b;
a=10;b=++a;
printf("preinkrementacja,  a= %d , b= %d\n", a, b);
a=10; b=a++;
printf("postinkrementacja, a= %d , b= %d\n", a, b);

printf("  Inkrementacja zmiennych typu float\n");
float c, d=10.0;  // c=a+++++a;
c=(d++)+(++d);
printf("z nawiasami,  d= %5.2f , c= %5.2f\n", d, c);
return 0;   }
```

```
preinkrementacja,  a= 11 , b= 11
postinkrementacja, a= 11 , b= 10
  Inkrementacja zmiennych typu float
z nawiasami,  d= 12.00 , c= 22.00
```

6.5. Operatory relacyjne, logiczne

Wartości logiczne w języku C mają następującą reprezentację: **fałsz/false** przez **0**, **prawda/true** przez **1** (ew. inna wartość różna od zera).

Operator	Działanie
>	większy
>=	większy lub równy
<	mniejszy
<=	mniejszy lub równy
==	równy
!=	nierówny
&&	iloczyn logiczny
||	suma logiczna

6.5. Operatory relacyjne, logiczne

Wynik działania operatorów relacyjnych i logicznych:

- negacja logiczna (relacja unarna),
- iloczyn logiczny (relacja binarna),
- suma logiczna (relacja binarna),
- suma modulo (relacja binarna).

Arg 1 0=false 1=true -----	Arg 2 0=false 1=true ------	Negacja Arg1 !	iloczyn logiczny Arg1, Arg2 &&	suma logiczna Arg1, Arg2 ||	suma modulo Arg1, Arg2 ^
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

6.5. Operatory relacyjne, logiczne, p6-5a

W C89 wartość logiczna "false" = 0, a "true" !0 (np. =1).

```
#include <stdio.h>
int main() {
int one, two;
printf ("Enter dwie liczby: \n");
scanf ("%d %d", &one, &two);
printf ("%d == %d jest %d\n", one, two, one==two);
printf ("%d != %d jest %d\n", one, two, one!=two);
printf ("%d <= %d jest %d\n", one, two, one<=two);
printf ("%d >= %d jest %d\n", one, two, one>=two);
printf ("%d  < %d jest %d\n", one, two, one<two);
printf ("%d  > %d jest %d\n", one, two, one>two);
return 0; }
```

printf – *drukowanie wartości liczb i wartości **logicznej relacji**.*

*zastosowanie kodu **%d** do wyprowadzenia wartości logicznej (**1** lub **0**)*

```
Enter dwie liczby:
3 4
3 == 4 jest 0
3 != 4 jest 1
3 <= 4 jest 1
3 >= 4 jest 0
3  < 4 jest 1
3  > 4 jest 0
```

6.6. Operatory bitowe

Operacje bitowe są zdefiniowane tylko dla liczb całkowitych. Działają one na poszczególnych bitach przez co mogą być szybsze od innych operacji.

A1 0=false 1=true -----	A2 0=false 1=true -----	Negacja bitowa A1 (NOT) ~	Koniunkcja bitowa A1, A2 (AND) &	Alternatywa bitowa A1, A2 (OR) |	Alternatywa rozłączna A1, A2 (XOR) ^
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Zobacz: **A1 ^ A2 ^ A2 to po prostu A1**. Właściwość ta została wykorzystana w różnych algorytmach szyfrowania oraz funkcjach haszujących. Alternatywę wyłączną stosuje się np. do szyfrowania kodu wirusów polimorficznych.

Kontynuacja – wykład nr 9.

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

NAZWA PRZEDMIOTU

Programowanie strukturalne

Temat wykładu 7.

Zmienne wskaźnikowe, operatory i arytmetyka wskaźnikowa. Tablice wskaźnikowe. Wskaźniki jako argumenty funkcji.

dr hab. inż. Jerzy Montusiewicz, prof. PL

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

7. Agenda

7.1. Zmienne wskaźnikowe.

7.2. Operatory i arytmetyka wskaźnikowa.

7.3. Wskaźniki i tablice.

7.4. Wskaźniki do wskaźników.

7.5. Wskaźniki jako argumenty funkcji.

7.6. Powstające problemy.

7.1. Zmienne wskaźnikowe

Wskaźniki (ang. pointer/pointers) to specjalne zmienne **służące do przechowywania adresów** innych zmiennych (tzw. adresowanie pośrednie).

Adres pamięci to liczba całkowita definiująca położenie obiektu (liczby, znaku) w pamięci komputera.

Wynika z tego, że **zmienne wskaźnikowe** nie przechowują wartości liczbowej tylko **adres** takiej wartości liczbowej.

Elementarną jednostką alokacji pamięci jest bajt. Wskaźnik pamięta jaki jest rozmiar wskazywanej przez niego zmiennej, np. zmienna **float** zajmuje 4 bajty. Więc zmiana wskaźnika pokaże następną zmienną, a nie kolejny bajt bieżącej zmiennej.

Zastosowanie wskaźników:

- modyfikacja argumentów przekazywanych do funkcji (oryginał/kopia),
- mogą zastępować efektywniej tablice,
- obsługują dynamiczną alokację pamięci.

7.1. Zmienne wskaźnikowe

Deklaracja zmiennej wskaźnikowej typu int:

int *ptr_to_int; można i tak **int* ptr_to_int;**

Zmienna posiada typ i nazwę (typy jak dla zmiennych).

Pamiętaj! **Błędy użycia wskaźników są niekiedy trudne do zlokalizowania.**

Operatory dotyczące wskaźników:

- Wpisanie adresu do wskaźnika,
  zmienna i wskaźnik muszą być tego samego typu

  ptr_to_int = &count; (count –zmienna typu **int**)
  (**&** - ampersand czyli „and” Ampera)

- Przekazanie wartości:

  new_var = *ptr_to_int; (**new_var = count**, zapis standardowy)

Operatory **&** i ***** są operatorami unarnymi.

7.1. Zmienne wskaźnikowe, p7-1a

Operatory adresowe i wskaźnikowe mają wyższy priorytet od operatorów arytmetycznych (są na poziomie unarnego minusa).

Prosty przykład zastosowania wskaźników.

```
#include <stdio.h>
int main()
{  int *count_adr, count, value, jm;
   count = 77;
   count_adr = &count;
   value = *count_adr;
   jm = *count_adr;
   printf("count = %d; value = %d; jm = %d\n", count, value, jm);
   printf("wartosc wskaznika = %p; wartosci %d\n", count_adr,
   *count_adr);
}
```

Powiązanie zmiennej ze wskaźnikiem

Operator adresowy '&'.

Operator wartościowy '*'.

```
count = 77; value = 77; jm = 77
wartosc wskaznika = 000000000062FE0C; wartosci 77
```

7.1. Zmienne wskaźnikowe, p7-1b

Zmienna wskaźnikowa **musi być** wiązana tylko ze swoim podstawowym typem zmiennej.

```
 8  #include <stdio.h>
 9
10  int main()
11  {
12      float x = 12.34, y;
13      int *ptr_to_int;
14  // Ostrzezenie kompilatora o niezgodnosci typow
15      ptr_to_int = &x;
16      y = *ptr_to_int;
17  // Przypadkowe wartosci
18
19      printf("y = %6.3f\n", y);
20  }
```

[Warning] assignment from incompatible pointer type [enabled by default]

```
y = 1095069824.000

--------------------
```

```
y = 12.340

------------
```

Tak powinno być

7.1. Zmienne wskaźnikowe

Stała łańcuchowa jest przechowywana w postaci tablicy znaków z dołączonym elementem '\0', którego oczywiście nie piszemy. Używając wskaźnika można to zapisać:

char *string1=”Jurek”;

string1 jest wskaźnikiem, a nie nazwą tablicy i może być zmieniany.

Samą tablicę można zdefiniować poprzez zapis:

char tab1[ ] = ”Janek”;

Rozmiar tablicy zostanie ustalony automatycznie i wynika z liczby znaków. W tym przypadku **tab1** jest stałym wskaźnikiem i nie może być zmieniany.

Istnieje możliwość definiowania wskaźnika do void jako ***void**. Można przekształcać wskaźniki różnych typów do typu **void*** bez utraty informacji. W ten sposób po wykonaniu operacji z powrotem na wskaźnik typu przywraca jest oryginalną wartość wskaźnika.

7.1. Zmienne wskaźnikowe, p7-1c

Użycie wskaźnika typu void *.

```
#include <stdio.h>
int main()
{ int x=23;
  int *p_x=&x;
  printf("Zawartosc wskaznika do int  p_x  =%d\n", *p_x);
  void *p_v;
  char *p_c1,*p_c2;
  p_c1=(char*) p_x;
  p_v=p_c1;
  p_c2=(char *)p_v;
  printf("Zawartosc wskaznika do char p_c2 =%d\n\n", *p_c2);
  system("pause");
  return 0;
}
```

Rzutowanie wskaźnika do **int** na pierwszy wskaźnik do **char**

Podstawienie wskaźnika do **char** na wskaźnik do **void**

Rzutowanie wskaźnika do **void** na nowy wskaźnik do **char**

```
Zawartosc wskaznika do int  p_x  =23
Zawartosc wskaznika do char p_c2 =23
```

7.2. Operatory i arytmetyka wskaźnikowa

Pamiętaj wskaźniki są pewnego rodzaju adresami zmiennych, tak więc modyfikacja adresów da nam możliwość dostępu do sąsiednich zmiennych.

Co można robić:

- wykonać inkrementowanie i dekrementowanie (**++, --**),
- dodawanie do / odejmowanie od wskaźnika liczby całkowitej,
- podstawianie wskaźnika do wskaźnika,
- porównywanie wskaźników,
- określanie różnicy wskaźników
  (określenie liczby zmiennych zdefiniowanego typu).

7.2. Operatory i arytmetyka wskaźnikowa, p7-2a

Podstawienie wskaźnika do wskaźnika.

```
#include <stdio.h>
int main()
{  float x;
   float *ptr1, *ptr2, *ptr3;
   x = 20.50;
   ptr1 = &x;
   ptr2 = ptr1;
   ptr3 = ptr2;
   printf("wskaznik ptr1 = %p pokazuje %f\n", ptr1, x);
   printf("wskaznik ptr2 = %p pokazuje %f\n", ptr2, *ptr2);
   printf("wskaznik ptr3 = %p pokazuje %f\n", ptr3, *ptr3);
   return 0;
}
```

deklaracja zmiennych wskaźnikowych

Przekazanie wartości zmiennej adresowej

Podstawienie wskaźnika do wskaźnika

Drukowanie wskaźnika i wartości

```
wskaznik ptr1 = 000000000062FE04 pokazuje 20.500000
wskaznik ptr2 = 000000000062FE04 pokazuje 20.500000
wskaznik ptr3 = 000000000062FE04 pokazuje 20.500000
```

7.2. Operatory i arytmetyka wskaźnikowa, p7-2b

Przeglądanie zawartości tablicy, zmienna wskaźnikowa.

```
int main() { short swar[] = {10, 20, 30, 40, 50};
short *p_short, noix; printf ("Zawartosc tablicy: \n");
for (noix=0; noix<5; noix++) printf ("%d  ", swar[noix]); printf("\n") ;
        p_short = &swar[0];
printf ("--- wskaznik p_short = %p  wskazuje %d\n", p_short, *p_short); p_short++;
printf ("po 1 inkrem. p_short = %p wskazuje %d\n", p_short, *p_short);
p_short = p_short + 1;
printf ("po 2 inkrem. p_short = %p wskazuje %d\n", p_short, *p_short); ++p_short;
printf ("po 3 inkrem. p_short = %p wskazuje %d\n", p_short, *p_short); p_short--;
printf ("po 1 dekrem. p_short = %p wskazuje %d\n", p_short, *p_short);
return 0;   }
```

Wskaźnik pokazuje początkowy element tablicy.

```
Zawartosc tablicy:
10  20  30  40  50
--- wskaznik p_short = 000000000062FE00 wskazuje 10
po 1 inkrem. p_short = 000000000062FE02 wskazuje 20
po 2 inkrem. p_short = 000000000062FE04 wskazuje 30
po 3 inkrem. p_short = 000000000062FE06 wskazuje 40
po 1 dekrem. p_short = 000000000062FE04 wskazuje 30
```

7.2. Operatory i arytmetyka wskaźnikowa, p7-2b

Zwróć uwagę na adresy.

Adresy różnią się o **2 bajty**.

Mieliśmy liczby typu **short**, które reprezentowane są przez 2 bajty.

Dostęp do zawartości tablic jest szybszy.

```
Zawartosc tablicy:
10  20  30  40  50
--- wskaznik p_short = 000000000062FE00 wskazuje 10
po 1 inkrem. p_short = 000000000062FE02 wskazuje 20
po 2 inkrem. p_short = 000000000062FE04 wskazuje 30
po 3 inkrem. p_short = 000000000062FE06 wskazuje 40
po 1 dekrem. p_short = 000000000062FE04 wskazuje 30
```

Gdy dodajemy do wskaźnika 3 to oznacza, że przesuwamy się 3*(rozmiar zmiennej), a nie o 3 bajty w pamięci komputera.

Adresy zależą od platformy sprzętowo-programowej oraz od typu zmiennej.

7.2. Operatory i arytmetyka wskaźnikowa

Ograniczenia operacji arytmetyki wskaźników.

- nie można skonstruować wskaźnika wskazujące miejsce poza zadeklarowaną tablicę,
- nie do tyczy sytuacji, gdy to jest miejsce (obiekt) bezpośrednio za ostatnim (np. elementem tablicy – ang. one past last).
- nie można odejmować od siebie wskaźników wskazujących na obiekty znajdujące się w różnych tablicach.

```
int tab[] = {1,3,5,7,9};
int *ws;
ws = &tab[0] + 8; //niezdefiniowane
ws = &tab[0] - 8; //niezdefiniowane
ws = &tab[0] + 5; //zdefiniowane, one past last
int tab1[] = {1,3,5,7,9} , b2[] = {2,4,6,8};
int *ws1= &tab1[0], *ws2= &b2[0];
int *def = ws1 – ws2;  //niezdefiniowane
```

7.3. Wskaźniki i tablice

Tablica to też rodzaj zmiennej wskaźnikowej. Nazwa tablicy jest wskaźnikiem, który wskazuje na miejsce pamięci, w którym przechowywany jest pierwszy element tablicy – element z indeksem ’0’ (elementy liczymy od 0).
Kolejne elementy tablicy znajdują się w następnych komórkach pamięci (odstępy zależą od wielkości zmiennej, czyli od zadeklarowanego typu zmiennej).

W kolejnych przykładach zaprezentowano różne metody wypisywania danych z tablicy:

- wykorzystanie indeksowania,
- inkrementowanie wskaźnika,
- indeksowaniem wskaźnika.

7.3. Wskaźniki i tablice, p7-3a

Wypisanie elementów tablicy przechowującej napis i zamiana dużych liter na małe.

```
int main() { char napis[80] = {"KATEDRA INFORMATYKI"};
int i_el;
printf ("* WERSJA INDEKSOWANA *\n");
printf ("Zamiana duzych liter na male\n");
printf("%s\n", "KATEDRA INFORMATYKI");
printf ("Napis malymi literami: \n");
printf("%c %d \n", napis[0], napis[0]);
printf("%c %d \n", napis[1], napis[1]);
for (i_el=0; napis[i_el]; i_el++)
{   printf ("%c ", tolower(napis[i_el]));  }
```

Zmienia duże litery na małe.

```
printf("\n");
printf("%c %d \n", tolower(napis[0]), tolower(napis[0]));
printf("%c %d \n", tolower(napis[1]), tolower(napis[1]));
printf("'k'-'K'= %d \n", tolower(napis[0])-napis[0]);
printf("'a'-'A'= %d \n", tolower(napis[1])-napis[1]);
return 0;   }
```

```
* WERSJA INDEKSOWANA *
Zamiana duzych liter na male
KATEDRA INFORMATYKI
Napis malymi literami:
K 75
A 65
k a t e d r a   i n f o r m a t y k i
k 107
a 97
'k'-'K'= 32
'a'-'A'= 32
```

7.3. Wskaźniki i tablice, p7-3b

Wypisanie elementów tablicy przechowującej napis i zamiana dużych liter na małe.

```
int main()
{ char napis[80] = {"KATEDRA INFORMATYKI"};
   char *ptr; int i_el;       ptr = &napis[0];
   printf ("* WERSJA WSKAZNIKOWA *\n");
   printf ("Zamiana duzych liter na male\n");
   printf("%s\n", "KATEDRA INFORMATYKI");
   printf ("Napis malymi literami: \n");
   while (*ptr)
   printf ("%c ", tolower(*(ptr++)));
   printf("\n");    return 0;
}
```

Koniec pętli, gdy wskaźnik "ptr" osiąga null-bajt.

```
* WERSJA WSKAZNIKOWA *
Zamiana duzych liter na male
KATEDRA INFORMATYKI
Napis malymi literami:
k a t e d r a   i n f o r m a t y k i
```

Następuje wykonanie operacji * (wyłuskanie wartości), a dopiero później inkrementacja wskaźnika **ptr++**. Nowa wartość wskaźnika obowiązuje w kolejnym obiegu pętli jeśli warunek jest spełniony.

7.3. Wskaźniki i tablice

Efektywność dostępu wskaźnikowego zamiast przez indeksy jest widoczna przy tablicach wielowymiarowych.

Pamiętaj!

Nazwa tablicy jest wskaźnikiem i może współpracować z operatorem indeksowania.

Tak więc wskaźniki też mogą być indeksowane.

7.3. Wskaźniki i tablice, p7-3c

Indeksowanie wskaźników.

```
int main()
{  char napis[80] = {"KATEDRA INFORMATYKI"};
   char *ptr; int i_el, i_st;
   ptr = &napis[0];
   printf ("* WERSJA Z INDEKSOWANIEM WSKAZNIKA *\n");
   printf ("Zamiana duzych liter na male\n");
   printf("%s\n", "KATEDRA INFORMATYKI");
   printf ("Napis malymi literami: \n");
   i_st=strlen(ptr);
   printf("dlugosc netto napisu = %d\n", i_st);
   for (i_el=0;i_el<i_st;i_el++)
   printf ("%c ", tolower(ptr[i_el]));
   printf("\n");
   return 0;
}
```

Funkcja strlen() oblicza długość netto stałej znakowe.

```
* WERSJA Z INDEKSOWANIEM WSKAZNIKA *
Zamiana duzych liter na male
KATEDRA INFORMATYKI
Napis malymi literami:
dlugosc netto napisu = 19
k a t e d r a   i n f o r m a t y k i
```

7.4. Wskaźniki do wskaźników, p7-4a

Struktura z dodatkowym stopniem pośredniości w adresowaniu.
We wskaźniku do wskaźnika można przechowywać adresy wskaźników (a nie adresy zmiennych).
W składni zapisu stosujemy **dwie gwiazdki wskaźnikowe** przed identyfikatorem zmiennej.

typ **nazwa_wsk1 , **nazwa_wsk2;

```
#include <stdio.h>
int main() {
 int i_war, *ptr_1, **ptr_2;
 i_war = 77;
 ptr_1 = &i_war;
 ptr_2 = &ptr_1;
printf ("Pojedyncza posredniosc: \n");
 printf ("    *ptr_1 = %p wskazuje wartosc %d\n", ptr_1, *ptr_1);
printf ("Podwojna posredniosc: \n");
 printf ("   **ptr_2 = %p wskazuje *ptr_1 %p ", ptr_2, *ptr_2);
 printf (" i wartosc %d\n", **ptr_2); return 0; }
```

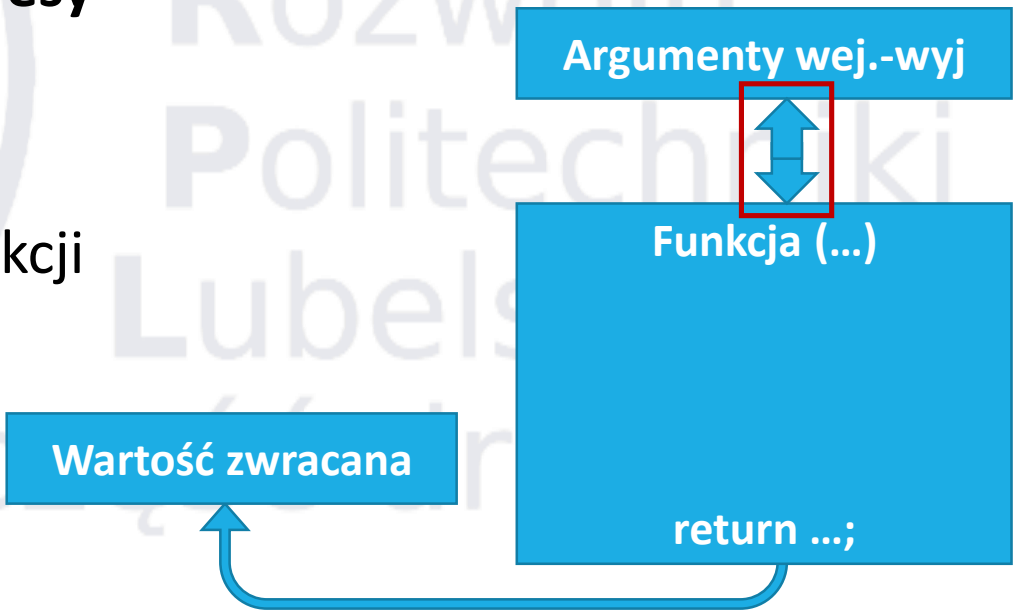
```
Pojedyncza posredniosc:
    *ptr_1 = 000000000062FE14
wskazuje wartosc 77
Podwojna posredniosc:
   **ptr_2 = 000000000062FE08
wskazuje *ptr_1 000000000062FE14
i wartosc 77
```

7.5. Wskaźniki jako argumenty funkcji

Funkcja ma charakter proceduralny, gdy sprawimy, że argumenty funkcji będą pełniły rolę wejścia do funkcji, ale także wyjścia.

Schemat poniżej

Aby zmienić rolę argumentów funkcji należy przekazywać **adresy zmiennych**, a nie ich wartości.
Poprzez dostęp do adresów **zmieniamy oryginały**.
Tak więc jako argumentów funkcji należy używać **wskaźników** do zmiennych.



7.5. Wskaźniki jako argumenty funkcji, p7-5a

Przekazywanie adresów argumentów (oryginałów) w funkcji TrueSwap i kopii w funkcji TrySwap.

```
#include <stdio.h>
void TrueSwap (int*, int*);
void TrySwap  (int , int );

int main()
{  int x, y;
   x = 9; y = 17;
   printf ("Wartosci poczatkowe \t   x=%d y=%d\n", x, y);
   TrueSwap ( &x, &y);
   printf ("main, wartosci po TrueSwap x=%d y=%d\n\n", x, y);
   x = 9; y = 17;
   printf ("Wartosci przywrocone \t   x=%d y=%d\n", x, y);
   TrySwap ( x, y);
   printf ("main, wartosci po  TrySwap x=%d y=%d\n", x, y);
   return 0;
}
```

Deklaracja prototypu.

Zmienne wskaźnikowe.

Wywołanie funkcji z adresami zmiennych.

Wywołanie funkcji ze zmiennymi.

7.5. Wskaźniki jako argumenty funkcji, p7-5a

```
void TrueSwap (int *w1, int *w2)
{  int temp;
   temp = *w1;
   *w1 = *w2;
   *w2 = temp;
   printf ("TrueSwap, wartosci \t   x=%d y=%d\n", *w1, *w2);
}
```

Wyłuskanie wartości

```
void TrySwap (int z1, int z2)
{  int temp;
   temp = z1;
   z1 = z2;
   z2 = temp;
   printf ("wartosc w TrySwap\t   x=%d y=%d\n", z1, z2);
}
```

```
Wartosci poczatkowe          x=9 y=17
TrueSwap, wartosci           x=17 y=9
main, wartosci po TrueSwap   x=17 y=9

Wartosci przywrocone         x=9 y=17
wartosc w TrySwap            x=17 y=9
main, wartosci po  TrySwap   x=9 y=17
```

7.6. Powstające problemy, p7-6a

Częsty błąd – brak inicjowania wskaźnika. Niezanicjowany wskaźnik. Błąd wykonania. Kompilator nie pokazał błędu.

```
#include <stdio.h>
int main()
{  int iwar, *pointer;
   iwar = 77;
   pointer = &iwar;
   *pointer = iwar;
   printf ("   iwartosc = %d; z pointer = %d\n", iwar, *pointer);
   return 0;
}
```

```
   iwartosc = 77; z pointer = 77
```

// pointer = &iwar;

Brak inicjowania wskaźnika

```
--------------------------------
Process exited after 1.735 seconds with return value 3221225477
Press any key to continue . . .
```

7.6. Powstające problemy, p7-6b

Częsty błąd – zamiast wskaźnika wartość zmiennej.

```
#include <stdio.h>
int main()
{  int iwartosc, *pointer;
   iwartosc = 77;
   pointer = iwartosc;
   printf ("   iwartosc = %d; z pointer = %d\n", iwartosc, *pointer);
   return 0;
}
```

Brak & przed zmienna

```
In function 'main':
[Warning] assignment makes pointer from integer without a cast [enabled by default]
--------------------------------
Process exited after 2.757 seconds with return value 3221225477
```

Po poprawkach

pointer = **&**iwartosc;

```
iwartosc = 77; z pointer = 77
```

printf ("   iwartosc = %d; z pointer = **%d**\n", iwartosc, *pointer);

printf ("   iwartosc = %d; z pointer = **%p**\n", iwartosc, pointer);

```
iwartosc = 77; z pointer = 000000000062FE14
```

7.6. Powstające problemy, p7-6c

Częsty błąd – niekontrolowana modyfikacja wskaźnika.
Inicjalizacja wskaźnika. null-bajt na końcu napisu.

```
#include <stdio.h>
#include <string.h>
{  char *ptr = "Witam w PL!";
   char text[40];
   int el, length;
   printf (ptr);
   printf ("\n");
   for (el=0; *ptr!=0; el++)  text[el] = *ptr++;
  // druga wersja
   for (el=0; el<strlen(ptr); el++)  text[el] = *ptr++;

   text[el] = '\0';
   printf ("  %s", text);
   return 0;
}
```

***ptr** – *zmienna wskaźnikowa wraz z nadaniem wartości*

wypisanie łańcucha bez podania formatu

```
Witam w PL!
  Witam w PL!
```

```
Witam w PL!
  Witam
```

W tej postaci pętli wartość "**strlen(ptr)**" jest określona w każdym obiegu i w rezultacie w tablicy "text" znajduje się tylko polowa łańcucha wskazywanego przez "**ptr**" .

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

NAZWA PRZEDMIOTU

Programowanie strukturalne

Temat wykładu 8.

Tablice statyczne jednowymiarowe i wielowymiarowe. Inicjalizacja tablic. Tablice o zmiennych rozmiarach. Tablice wskaźników.

dr hab. inż. Jerzy Montusiewicz, prof. PL

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

8. Agenda

8.1. Tablice statyczne jednowymiarowe.

8.2. Tablice statyczne wielowymiarowe.

8.3. Inicjalizacja tablic.

8.4. Tablice o zmiennych rozmiarach.

8.5. Tablice wskaźników.

8.6. Tablice jako argumenty funkcji.

8.1. Tablice statyczne jednowymiarowe

Tablice pozwalają zapamiętać wiele wielkości tego samego typu pod jedną nazwą zmiennej.

typ nazwa_tablicy[rozmiar];

- typ – nazwa typu zmiennych tworzących tablicę,
- nazwa_tablicy – identyfikator,
- rozmiar – stała liczbowa (istnieją odstępstwa) określająca liczbę elementów składowych tablicy,
- [ ] – sposób podawania rozmiaru, a także do indeksowania tablicy (dostęp do elementów),

Indeksowanie tablicy liczymy od ’0’, stąd **tab[5]** oznacza szósty element tablicy o nazwie **tab**.

PAMIĘTAJ OPERATOR INDEKSOWANIA NIE SPRAWDZA ROZMIARU TABLICY

8.1. Tablice statyczne jednowymiarowe, p8-1a

Brak sprawdzania przekroczenia wymiaru tablicy.

```
#include <stdio.h>
#define   CRASH_LIMIT  20
int main()
{  int tab1[10], t_crash[20];
int it;
printf("Wypelnianie tablicy 'tab1[10]':\n");
for (it=0; it<10; it++)          tab1[it] = it +1;
 printf("Wartosc 5-tego elem. tablicy 'tab1': %d\n", tab1[4]);
/* Naruszenie ochrony pamieci - dla 11-20 elementów */
printf("Wypelnienie z naruszeniem do %d-tego elementu\n", CRASH_LIMIT);
for (it=0; it<CRASH_LIMIT; it++)       t_crash[it] = it +1;
printf("Wartosc 10-tego elem. tablicy 'crash': %d\n", t_crash[9]);
printf("Wartosc 16-tego elem. tablicy 'crash': %d\n", t_crash[15]);
}
```

```
Wypelnianie tablicy 'tab1[10]':
Wartosc 5-tego elem. tablicy 'tab1': 5
Wypelnienie z naruszeniem do 20-tego elementu
Wartosc 10-tego elem. tablicy 'crash': 10
Wartosc 16-tego elem. tablicy 'crash': 16
```

NIEWIELKIE WYKROCZENIE POZA TABLICĘ NIE MUSI WYWOŁYWAĆ POKAZANIA BŁĘDU (np. segmentation fault)

8.1. Tablice statyczne jednowymiarowe, p8-1b

Brak sprawdzania przekroczenia wymiaru tablicy.

```
#include <stdio.h>
#define   CRASH_LIMIT  20
int main() { int tab1[10], t_crash[10];
int it;
printf("Wypelnienie z naruszeniem do %d-tego elementu\n",
   CRASH_LIMIT);
   for (it=0; it<CRASH_LIMIT; it++)          t_crash[it] = it +1;
printf("Wartosc 10-tego elem. tablicy 'crash': %d\n", t_crash[9]);
printf("Wartosc 16-tego elem. tablicy 'crash': %d\n", t_crash[15]);
   printf("Wypelnianie tablicy 'tab1[10]':\n");
   for (it=0; it<10; it++)    tab1[it] = it +1;
   printf("Wartosc 5-tego elem. tablicy 'tab1': %d\n", tab1[4]);
printf("Wartosc 10-tego elem. tablicy 't_crash': %d\n", t_crash[9]);
printf("Wartosc 16-tego elem. tablicy 't_crash': %d\n", t_crash[15]);
return 0;    }
```

Naruszenie ochrony pamięci
- dla elementów > 10.

8.1. Tablice statyczne jednowymiarowe, p8-1b

Brak sprawdzania przekroczenia wymiaru tablicy.

Naruszenie ochrony pamięci
– elementów dla > 10.

t_crash[10];

t_crash[15]);

```
Wypelnienie z naruszeniem do 20-tego elementu
Wartosc 10-tego elem. tablicy 'crash': 10
Wartosc 16-tego elem. tablicy 'crash': 16
Wypelnianie tablicy 'tab1[10]':
Wartosc 5-tego elem. tablicy 'tab1': 5
Wartosc 10-tego elem. tablicy 't_crash': 10
Wartosc 16-tego elem. tablicy 't_crash': 4
```

t_crash[15]

Powtórne drukowanie z tablicy
z przekroczeniem rozmiaru

8.2. Tablice statyczne wielowymiarowe

Tablice wielowymiarowe są funkcjonalnym rozszerzeniem i uzupełnieniem tablic jednowymiarowych.

typ nazwa_tablicy[rozmiar1] [rozmiar2] … [rozmiarj];

rozmiar1, … – stałe liczbowe (istnieją odstępstwa) określające liczbę elementów składowych dla każdego wymiaru tablicy.

Wielokrotne zastosowanie operatora indeksowana.

Indeksy przyjmują wartości od **0** do wartości **rozmiarj-1**.

W tablicach dwuwymiarowych **pierwszy indeks jest wierszem**, **drugi – kolumną**.

(Na poziomie języka maszynowego takie tablice przechowywane są wierszami, w Fortranie zaś kolumnami).

8.2. Tablice statyczne wielowymiarowe, p8-2a

Tablica dwuwymiarowa, przechowywana jest wierszami.

```
#include <stdio.h>
#define  ROWS 3
#define  COLS 5
void ArrayAsOneDimensional (int array[], int size, int no_col);
int main()  { int array [ROWS][COLS];
   int ix, iy;
   printf("Wypelnianie 'array'\n");
   for (ix=0; ix<COLS; ix++)
            for (iy=0; iy<ROWS; iy++)
            array[iy][ix] = (iy*COLS) +ix +1;
printf("Drukowanie tablicy 'array'\n- wiersz po wierszu\n");
   for (ix=0; ix<ROWS; ix++)
   {        printf ("wiersz nr %2d:\n", ix);
            for (iy=0; iy<COLS; iy++)
            printf ("%2d   ", array[ix][iy]);
            printf("\n"); }
   ArrayAsOneDimensional(&array[0][0], ROWS*COLS, COLS);
   return 0; }
```

Tablica jest wskaźnikiem, przekazywanie wartości oryginalnych.

Druk tablicy dwuwymiarowej jako jednowymiarowej.

8.2. Tablice statyczne wielowymiarowe, p8-2a

```
void ArrayAsOneDimensional (int array[], int size, int no_col)
{  int iy;
   printf ("Drukowanie tablicy 'array'\n jako jednowymiarowej\n");
   for (iy=0; iy<size; iy++)
    { printf ("%2d   ", array[iy]);
      if ((iy+1)%no_col == 0) printf ("\n");
    }
}
```

Wypisywanie zawartości tablicy

```
wiersz nr  0:
 1    2    3    4    5
wiersz nr  1:
 6    7    8    9   10
wiersz nr  2:
11   12   13   14   15
Drukowanie tablicy 'array'
 jako jednowymiarowej
 1    2    3    4    5
 6    7    8    9   10
11   12   13   14   15
```

Wydruk w obu przypadkach jest identyczny co potwierdza przyjęte założenie.

8.2. Tablice statyczne wielowymiarowe, p8-2b

Zmiana wartości tablicy w definiowanej funkcji i ich wydruk w main().

```
...ArrayAsOneDimensional(&array[0][0], ROWS*COLS, COLS);
   printf("Drukowanie po zmianie w main()\n");
   for(ix=0; ix<ROWS; ix++)
   {        printf ("wiersz nr %2d:\n", ix);
            for(iy=0; iy<COLS; iy++)
            printf ("%2d   ", array[ix][iy]); printf("\n"); }

void ArrayAsOneDimensional (int array[], int size, int no_col)
{  int iy;
   printf ("Drukowanie tablicy 'array'\n jako jednowymiarowej\n");
   for (iy=0; iy<size; iy++)
   {  printf ("%2d   ", array[iy]); if ((iy+1)%no_col == 0) printf ("\n"); }
   printf ("Zmiana wartosci elementow tablicy\n");
    for (iy=0; iy<size; iy++) array[iy]=2*array[iy];
   printf("Drukowanie po zmianie\n");
    for (iy=0; iy<size; iy++)
    { printf ("%2d   ", array[iy]);
      if ((iy+1)%no_col == 0) printf ("\n"); } }
```

*Funkcja typu **void** nic nie zwraca, ale przeniesienie wartości do funkcji main() przez tablicę, czyli przez wskaźnik, która jest argumentem.*

8.2. Tablice statyczne wielowymiarowe, p8-2b

Zmiana wartości tablicy w definiowanej funkcji i ich wydruk w main().

```
Wypelnianie 'array'
Drukowanie tablicy 'array'
- wiersz po wierszu
wiersz nr  0:
 1    2    3    4    5
wiersz nr  1:
 6    7    8    9   10
wiersz nr  2:
11   12   13   14   15
Drukowanie tablicy 'array'
 jako jednowymiarowej
 1    2    3    4    5
 6    7    8    9   10
11   12   13   14   15
Zmiana wartosci elementow tablicy
Drukowanie po zmianie
 2    4    6    8   10
12   14   16   18   20
22   24   26   28   30
Drukowanie po zmianie w main()
wiersz nr  0:
 2    4    6    8   10
wiersz nr  1:
12   14   16   18   20
wiersz nr  2:
22   24   26   28   30
```

8.2. Tablice statyczne wielowymiarowe, p8-2c

Tablice napisów, przechowywane wierszami.

```
#include <stdio.h>
#define  ROWS 30
#define  COLS 15
int main()
{  char tekst[ROWS][COLS];
   int row, line;
printf("Wypelnianie tabeli 'tekst'\n");
printf("Zakoncz wiersz w miejscu\n pierwszej spacji\n");
   for (row=0; row<ROWS; row++)
   { printf ("%d:  ", row);
     gets(tekst[row]);
     if (tekst[row][0]==' ') break; }
printf("Drukowanie tablicy 'tekst'\n- wiersz po wierszu\n");
for(line=0; line<row; line++) printf ("%s\n", tekst[line]);
   return 0;
}
```

Definiowanie rozmiarów tablicy

Podano jedynie numer wiersza

Pojedynczy indeks - wybieranie wierszy.

```
Wypelnianie tabeli 'tekst'
Zakoncz wiersz w miejscu
 pierwszej spacji
0:  jurek
1:  katedra
2:  politechnika
3:  lubelska
4:
Drukowanie tablicy 'tekst'
- wiersz po wierszu
jurek
katedra
politechnika
lubelska
```

8.3. Inicjalizacja tablic

Inicjalizacja to nadanie pierwotnej wartości elementom tablicy.
Do inicjalizacji tablicy można zastosować następujący szkielet składni.

Typ nazwa_tablicy[rozm1][rozm2]… = {lista_wartości};

- Liczba elementów w **{ }** powinna odpowiadać wymiarom tablicy (nie powinna być większa niż pojemność tablicy);
- Kompilatory czasami informują o nadmiarowej liście elementów;
- Elementy oddzielamy od siebie przecinkami;
- Na końcu nawiasu **}** umieszczamy średnik, a więc tak **};**
- Użyteczne jest inicjowanie tablic globalnych lub statycznych przy pomocy modyfikatora **static** .

UPROSZCZENIA W INICJOWANIU TABLIC

char my_string[] = ”witaj ” *pominięto rozmiar tablicy*
char my_string = ”witaj ” *pominięto nawiasy kwadratowe*

8.3. Inicjalizacja tablic, p8-3a

Inicjalizacja tablic i automatyczne rezerwowanie miejsca.

```
#include <stdio.h>
int main()
{  char tytul[] = "tablica kwadratow\n";
   char wiadomosc[][20] =
   {"tablica kwadratow\n ", "tablica szescianow\n",
   "NOWY tekst\n, jurek\n"};
   int kwadraty[][4] = {1 ,1, 2, 4, 3, 9,
   4, 16, 5, 25, 6, 36, 7, 49, 8, 64 };
   int row, max_rows;
   max_rows = sizeof(kwadraty)/(4*sizeof(int));
   printf("\n   max_rows = %d\n\n", max_rows);
   printf(tytul);
   for(row=0; row<max_rows; row++)
   printf ("%d   %d\n", kwadraty[row][0], kwadraty[row][1]);
            printf (wiadomosc[2]);
            printf (wiadomosc[1]);
            printf (wiadomosc[2]);
return 0; }
```

Stała napisowa. Nie musimy stosować {}

inicjalizacja tablicy **wiadomosc {};**

W tablicach wielowymiarowych pominięto pierwszy wymiar.

Tablica dwuwymiarowa, stosujemy **{};**

Obliczanie liczby wierszy tablicy

8.3. Inicjalizacja tablic, p8-3a

Z tablicy o 4. kolumnach drukujemy tylko 2. pierwsze kolumny

```
   max_rows = 5

tablica kwadratow
1   1
3   9
5   25
7   49
4   0
NOWY tekst
, jurek
tablica szescianow
NOWY tekst
, jurek
```

```
...
int kwadraty[][4] = {1 ,1, 2, 4, 3, 9,
   4, 16, 5, 25, 6, 36, 7, 49, 8, 64 };
...
for(row=0; row<max_rows; row++)
   printf ("%d   %d\n", kwadraty[row][0], kwadraty[row][1]);
```

8.4. Tablice o zmiennych rozmiarach, C99, p8-4a

Zmienna określająca wymiar tablicy powinna być zadeklarowana i zainicjowana przed użyciem do wymiarowania tablicy. Takie tablice nie są w pełni dynamiczne.

```
#include <stdio.h>
int main()
{ int dim, el;
   printf ("Wprowadz rozmiar tablicy:  \n");
   scanf("%d", &dim);
   int array[dim];
   printf("Wypelnianie tablicy 'array'\n");
   for(el=0; el<dim; el++) array[el] = el+1;
// Sprawdzenie zawartosci tablicy
   printf ("Wyswietlenie zawartosci tablicy:\n");
   for(el=0; el<dim; el++)
   { printf("%4.0d   ", array[el]);
     if((el+1)%5 == 0)  printf("\n"); }
   printf("\n"); }
```

```
Wprowadz rozmiar tablicy:
7
Wypelnianie tablicy 'array'
Wyswietlenie zawartosci tablicy:
   1      2      3      4      5
   6      7
```

Po 5-mej liczbie przejście do nowej linii.

NIE WSZYSTKIE KOMPILATORY OBSŁUGUJĄ C99.

8.5. Tablice wskaźnikowe

Definicja tablicy wskaźników:

typ *tab_nazwa[rozmiar] ;

Np. **int *tab_nazwa[3]** – jest tablicą, której elementami są wskaźniki do **int**.

Takie tablice są przydatne do definiowania tablic grupujących napisy. Gdy tablica dwuwymiarowa ma wiersze o różnej długości (**oszczędzamy miejsce**, nie deklarujemy nadmiarowego rozmiaru równego długości najdłuższego wiersza, np.:

1 – ten napis jest bardzo długi

2 – krótki

3 – ostatni jest średni

8.5. Tablice wskaźnikowe, p8-5a

Tablica wskaźników do przechowywania tekstów o różnych długościach napisów.

```
#include <stdio.h>
int main()  {
   int tab_i[] = {5, 4, 3, 2, 1};
   int *ptr_int[5], el;
   char *napisy[] =
   {"1w_bardzo dlugi napis\n",
    "2w_krotki\n",
    "3w_trzeci wiersz\n", };
   printf ("Zawartosc tablicy int: ");
   for (el=0; el<5; el++) printf ("%d  ", tab_i[el]);
   printf ("\n");
   for (el=0; el<5; el++)
   ptr_int[el] = &tab_i[el];
   printf ("Zawartosc tablicy \n");
   printf ("za posrednictwem tablicy wskaznikow:\n");
   for (el=0; el<5; el++) printf ("%d  ", *ptr_int[el]);
   printf ("\n");
   printf ("Tablica z napisami wskaznikow:\n");
   for (el=0; el<3; el++) printf ("%s", napisy[el]);
   return 0; }
```

Tablica wskaźników do napisów

*Adresy poszczególnych elementów tablicy "**tab_i**" do tablicy wskaźników "**ptr_int**".*

```
Zawartosc tablicy int: 5  4  3  2  1
Zawartosc tablicy
za posrednictwem tablicy wskaznikow:
5  4  3  2  1
Tablica z napisami wskaznikow:
1w_bardzo dlugi napis
2w_krotki
3w_trzeci wiersz
```

8.5. Tablice wskaźnikowe, p8-5a

```
char *napisy[] =
 {"1w_bardzo dlugi napis\n",
  "2w_krotki\n",
  "3w_trzeci wiersz\n", };
```

Charakter dynamiczny ma tylko liczba wierszy

Alternatywa:

```
char text[][20] = {tekst1,tekst2,tekst3};
```

```
int *ptr_int[5], el;
```

Co tu mamy?

```
printf ("%p  ", ptr_int[el]);
```

Druk wskaźników

```
za posrednictwem tablicy wskaznikow:
000000000062FE00  000000000062FE04  000000000062FE08
000000000062FE0C  000000000062FE10
```

8.6. Tablice jako argumenty funkcji

Tablica może być argumentem funkcji. Ponieważ nazwa tablicy jest wskaźnikiem, więc działamy na wartościach będących oryginałem.

Dysponujemy wszystkimi wartościami w funkcji, z której wywoływaliśmy naszą zdefiniowaną funkcję. W definiowanej funkcji można ustalić wartości wszystkich elementów funkcji.

Wprowadzanie wartości tablicy w funkcji dane(). Wykorzystanie tych wartości w funkcji main(). Przekazanie przez instrukcję return wartości określającej liczbę obiegu pętli for.

Oblicz średnie arytmetyczne dla sumy liczb parzystych oraz nieparzystych.

8.6. Tablice jako argumenty funkcji, p8-6a

```
int dane(int war[]);
main()
{ int odd=0, even=0, l_o=0, l_e=0;
int i, mm, war[10];
float o_ave, e_ave;
mm=dane(war);      Zwraca wartość max
for(i=0;i<mm;i++)
 {if(war[i]%2) { odd+=war[i]; l_o+=1;
    printf("%d , %d\n", odd,l_o);}
  else  { even+=war[i];  l_e+=1;
  printf("%d , %d\n", even,l_e);} }
if (l_o) {o_ave=(float)odd/l_o;
printf("average from odd  = %f \n", o_ave); }
if (l_e) { e_ave=(float)even/l_e;
printf("average from even = %f \n", e_ave); }
return 0;
}
```

```
int dane(int war[]) {
 int val, max=5;
for (val=0;val<max;val++)
{ printf("Enter int \t ");
  scanf("%d", &war[val]); }
return max;          }
```

```
Enter int        1
Enter int        3
Enter int        1
Enter int        4
Enter int        2
1 , 1
4 , 2
5 , 3
4 , 1
6 , 2
average from odd  = 1.666667
average from even = 3.000000
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

NAZWA PRZEDMIOTU

Programowanie strukturalne

Temat wykładu 9.

Dynamiczna alokacja pamięci. Dostęp do danych. Klasy pamięci. Operatory bitowe.

dr hab. inż. Jerzy Montusiewicz, prof. PL

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

9. Agenda

9.1. Dynamiczna alokacja pamięci.

9.2. Dostęp do danych.

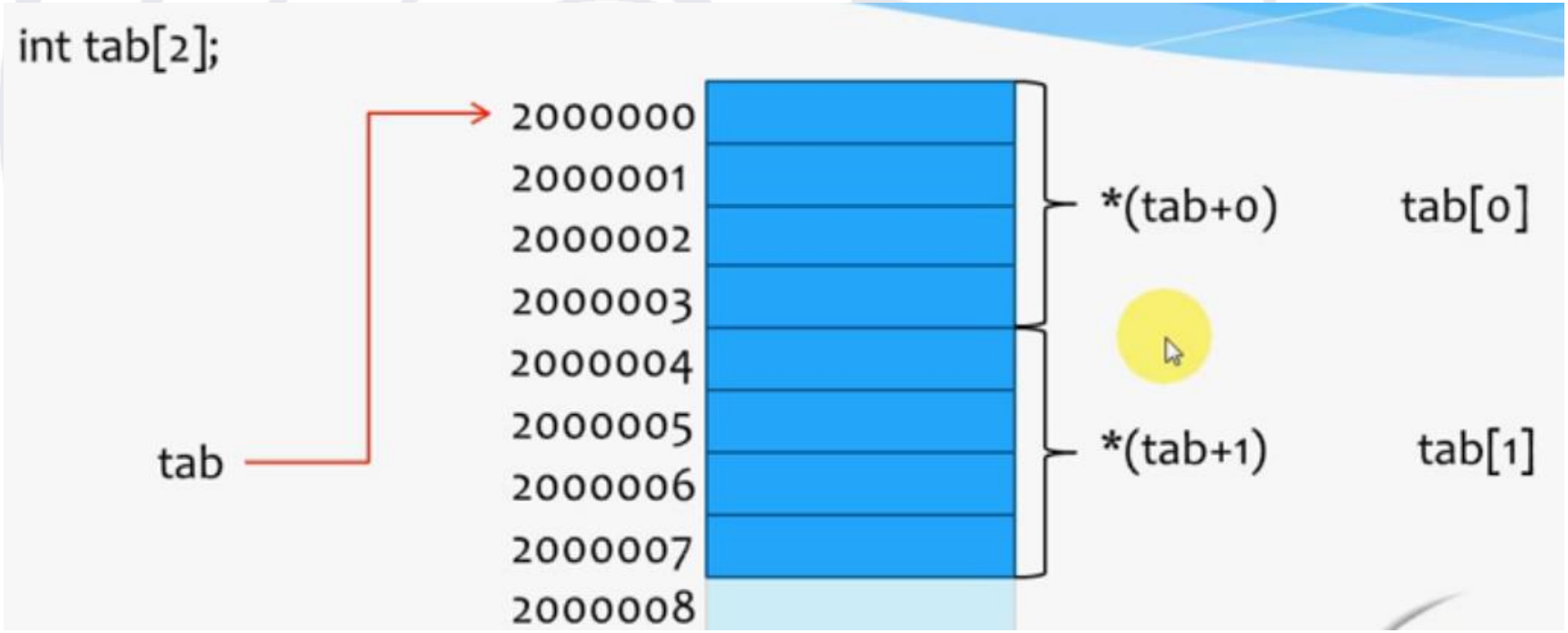
9.3. Klasy pamięci.

9.4. Operatory bitowe.

9.1. Dynamiczna alokacja pamięci

Przypomnienie o statycznej alokacji pamięci. Deklaracja klasyczna tablicy powoduje przydzielenie pamięci na stałe. Tej pamięci nie można zwolnić, pamięć jest zablokowana do końca działania programu.

Należy przy tym pamiętać, że nazwa tablicy jest wskaźnikiem do pierwszego elementu tablicy.



9.1. Dynamiczna alokacja pamięci

Jest przydzielaniem pamięci o rozmiarach określonych dopiero w trakcie wykonywania programu.

Pamięć ta jest przydzielana ze specjalnego obszaru pamięciowego nazywanego stertą (ang. heap).

Dostęp do tej pamięci odbywa się przy użyciu wskaźnika, gdyż funkcja **malloc()** lub **calloc()** alokująca pamięć zwraca wskaźnik do początku zaalokowanego obszaru.

```
wskaznik = (typ *) malloc (rozmiar_w_bajtach);
```

Warto dołączyć bibliotekę **stdlib.h**

Przy określeniu rozmiaru pamięci w bajtach pomocny będzie operator rozmiaru **sizeof**.

9.1. Dynamiczna alokacja pamięci, sizeof, p9-1

Operator fazy kompilacji.

sizeof – określa rozmiar w bajtach typu wbudowanego lub zmiennej.

Można stosować do **zmiennych**, także do **tablic**, **struktur, unii**.
W przypadku zmiennych można je ująć w nawiasy.

```
int main() {
int izm, u, x, y, z;
float fzm; double tab[5];
   u = sizeof (fzm);
   x = sizeof izm;
   y = sizeof (short); // koniecznie w nawiasach
   z = sizeof tab;
printf("rozmiar zmiennej fzm   = %d\n", u);
printf("rozmiar zmiennej izm   = %d\n", x);
printf("rozmiar typu short     = %d\n", y);
printf("rozmiar tablicy double = %d\n", z);
return 0;  }
```

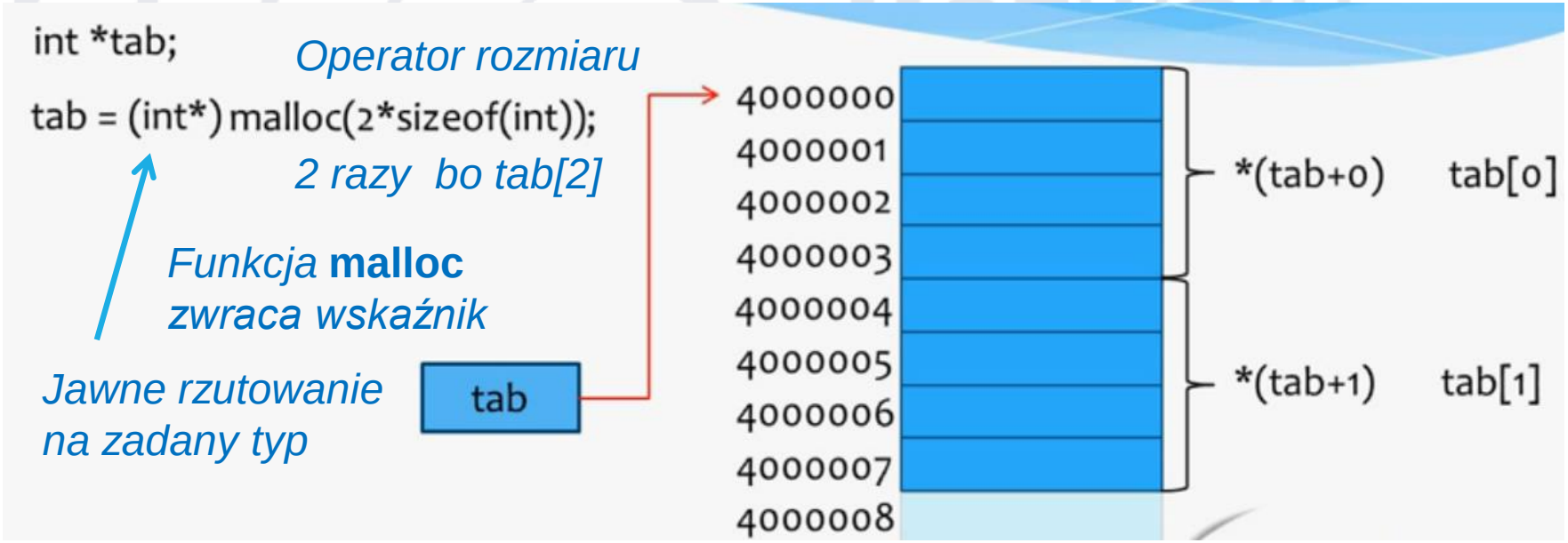
```
rozmiar zmiennej fzm   = 4
rozmiar zmiennej izm   = 4
rozmiar typu short     = 2
rozmiar tablicy double = 40
```

9.1. Dynamiczna alokacja pamięci

Funkcja **malloc()** służy do dynamicznej alokacji pamięci, ale nie wypełnia jej zawartości.

wskaznik = (typ *) malloc (rozmiar_w_bajtach);

wskaźnik typu **int**



9.1. Dynamiczna alokacja pamięci, p9-1b

Dynamiczny przydział funkcją malloc(). Drukowanie adresu i zawartości.

```
int main (void)
{ int *ptr1; int size1, i;
   printf ("Podaj rozmiar 1. obszaru (>=4): ");
   scanf ("%d", &size1);
   if ((ptr1 = (int *)malloc(sizeof(int)*size1)) == NULL)
   { printf ("BLAD ALOKACJI PIERWSZEGO OBSZARU.\n"); goto stop1; }
   printf ("Adres zaalokowanego obszaru: %#x, %p \n", ptr1, ptr1);
   printf ("Zawartosc elementow: \n");
   for (i=0; i<size1; i++) { printf ("%#X ", *(ptr1+i)); } printf ("\n");
   for (i=0; i<size1; i++) { printf ("%d ", *(ptr1+i)); } printf ("\n");
   free (ptr1);
   stop1:
   return 0;
}
```

Zastosowanie kodu **%#x** oraz **%d**

Zastosowanie kodu **%#x** oraz **%p**

```
Podaj rozmiar 1. obszaru (>=4): 6
Adres zaalokowanego obszaru: 0x7114a0, 00000000007114A0
Zawartosc elementow:
0X7170D0 0 0X710150 0 0X3A433D63 0X6E69575C
7434448 0 7405904 0 977485155 1852397404
```

9.1. Dynamiczna alokacja pamięci

Funkcja **calloc()** gwarantuje wyzerowanie przydzielonej pamięci, rozmiar określamy nie w bajtach, lecz jako iloczyn liczby elementów i rozmiar elementu.

wskaznik = (typ *) calloc (liczba_elem, rozmiar_elem_w_bajtach);

Gdyby było za mało pamięci na stercie, funkcje zwracają wartość NULL (‘\0’)

Powinniśmy sprawdzić czy taka sytuacja nie nastąpiła.

Do zwracania zbędnej pamięci do puli systemu operacyjnego – na stertę, używamy w sposób jawny funkcji **free()**.

free (wskaznik);

9.1. Dynamiczna alokacja pamięci, p9-1c

Dynamiczny przydział funkcją calloc(). Drukowanie adresu i zawartości.

```
int main (void)
{  int *ptr1; int size1, i;
   printf ("Podaj rozmiar 1. obszaru (>=4): ");
   scanf ("%d", &size1);
   if ((ptr1 = (int *)calloc(size1, sizeof(int))) == NULL)
   { printf ("BLAD ALOKACJI PIERWSZEGO OBSZARU.\n");
     goto stop1; }
   printf ("Adres zaalokowanego obszaru: %#x\n  %p \n", ptr1, ptr1);
   printf ("Zawartosc elementow: \n");
   for (i=0; i<size1; i++) { printf ("%#X ", *(ptr1+i)); } printf ("\n");
   free (ptr1);
   stop1:
   return 0; }
```

Zastosowanie kodu **%#x** oraz **%p**

```
Podaj rozmiar 1. obszaru (>=4): 6
Adres zaalokowanego obszaru: 0xc414a0
  0000000000C414A0
Zawartosc elementow:
0 0 0 0 0 0
```

9.1. Dynamiczna alokacja pamięci, p9-1d

Do zmiany rozmiaru zaalokowanej pamięci używamy funkcji **realloc()**.

pointer2 = (typ *) realloc(wskaznik_biezacy, nowy_roz_w_bajtach);

pointer2 – nowy wskaźnik wskazujący obszar pamięci

Gdy funkcja przydziela obszar większy od dotychczasowego, to w dotychczasowym obszarze nic nie ulega zmianie.

Gdy obszar jest mniejszy, to część obszaru jest odcinana, a informacja w odciętej części utracona.

Alokacja 2. obszaru po
to aby zająć obszar
pamięci po 1. obszarze.

Zmiana adresu 1. obszaru
alokacji po jego rozszerzeniu.

```
Podaj rozmiar 1. obszaru (>=4): 3
Adres 1. obszaru: 0x9e14a0
Zawartosc elementow:
0 0 0
Adres 2. obszaru: 0x9e14c0

Adres 1. obszaru
      po realloc: 0X9E1500
Zawartosc elementow:
0 0 0 0 0X20202020 0X20202020
```

9.1. Dynamiczna alokacja pamięci, p9-1d

Zmiana rozmiaru zaalokowanej pamięci funkcją realloc().

```
int main (void) {
   int *ptr1; double *ptr2; int size1, i;          Wskaźnik ptr2 ma typ double
   printf ("Podaj rozmiar 1. obszaru (>=4): ");
   scanf ("%d", &size1);
   if ((ptr1 = (int *)calloc(size1, sizeof(int))) == NULL)
   { printf ("BLAD ALOKACJI PIERWSZEGO OBSZARU.\n"); goto stop1; }
   printf ("Adres 1. obszaru: %#x \n", ptr1);
   printf ("Zawartosc elementow: \n");
   for (i=0; i<size1; i++) { printf ("%#X ", *(ptr1+i)); } printf ("\n");
   ptr2 = (double *)calloc(2*size1, sizeof(double));      Alokacja 2. obszaru
   printf ("Adres 2. obszaru: %#x\n  ", ptr2);
   ptr1 = (int *)realloc(ptr1, 2*size1*sizeof(int));      Rozszerzenie 1. obszaru
   printf ("\nAdres 1. obszaru\n     po realloc: %#X\n", ptr1);
   printf ("Zawartosc elementow: \n");
   for (i=0; i<2*size1; i++) { printf ("%#X ", *(ptr1+i)); } printf ("\n");
   free (ptr1); stop1:
   return 0;      }
```

9.1. Dynamiczna alokacja pamięci, p9-1e

Funkcja **memset()** służy do wypełnienia pamięci jednolitą bajtową zawartością:

memset(wskaznik, bajt, liczba_wypelnianych_bajtow);

wskaźnik – wskazuje początek wypełnianego obszaru,

bajt – jest wartością wpisywaną do obszaru,

liczba_wypelnianych_bajtow – to liczba bajtów.

```
memset(ptr1, 0, (size1/2)*sizeof(int));
printf ("Zawartosc elementow po memset():\n");
for (i=0; i<size1; i++)
{ printf ("%#X ", *(ptr1+i)); }
```

Tylko pół obszaru wypełniono '0'.

```
Podaj rozmiar 1. obszaru (>=4): 6
Adres 1. obszaru: 0xb414a0
Zawartosc elementow:
0XB470D0 0 0XB40150 0 0X3A433D63 0X6E69575C
Zawartosc elementow po memset():
0 0 0 0 0X3A433D63 0X6E69575C
```

9.1. Dynamiczna alokacja pamięci, p9-1f

Funkcja **memmove()** kopiuje zawartość pewnego obszaru pamięci w inne miejsce:

memmove(wsk_docelowy, wsk_zrodlowy, liczba bajtow);

```
if ((ptr1 = (int *)calloc(size1, sizeof(int))) == NULL)
{ printf ("BLAD ALOKACJI PIERWSZEGO OBSZARU.\n");
  goto stop1; }
memmove(ptr1+2*size1, ptr1, (size1/2)*sizeof(int));
printf ("Zawartosc elementow po memmove():\n");
for (i=0; i<3*size1; i++)
{ printf ("%#X ", *(ptr1+i)); }
```

```
Podaj rozmiar 1. obszaru (>=4): 4
Adres 1. obszaru: 0x1814a0
Zawartosc elementow:
0 0 0 0
Zawartosc elementow po memmove():
0 0 0 0 0 0 0X23F3B884 0X1000282D 0 0 0 0
```

Zastosowanie kodu **%#X**

9.2. Dostęp do danych

Dostęp do danych może być kontrolowany za pomocą modyfikatora **const**.
Szkielet o postaci:

modyfikator typ lista zmiennych;

Zmienne **const** można **inicjować**, ale nie zmieniać przez podstawienie.

Przykład poprawny:

const double e = 2.718;

Cechy:

- pozwala precyzyjnie kontrolować typ zmiennej,
- łatwo definiować tablice i zapobiegać modyfikacji tablic przekazywanych jako argumenty do funkcji.

Uwaga!

Liczba e (**Eulera**, Nepera), podstawa logarytmu naturalnego. Stała matematyczna, w przybliżeniu 2,718281828459.
Jeden ze wzorów do obliczenia e: $e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$

9.2. Dostęp do danych, p9-2a1, p9-2a2

Cel deklaracji tablic jako const – ochrona przed niepożądaną zmianą tablicy. Tablica jest argumentem funkcji.

```
int ChangeArray (int *numery);
int main (void)
{ int numbers[] = {2, 4, 6, 8, 10, 12};
   ChangeArray(numbers);
   return 0; }
int ChangeArray (int *numery)
{ int ix;s
   for (ix=0; ix<6; ix++)
   numery[ix] = 2*numery[ix];
return 0; }
```

```
tablica przed wywolaniem 'ChangeArray'
2 4 6 8 10 12
tablica w 'ChangeArray'
4  8  12  16  20  24
tablica po powrocie z 'ChangeArray'
4 8 12 16 20 24
```

```
int ChangeArray (const int *numery);
int main (void)
{ int numbers[] = {2, 4, 6, 8, 10, 12};
   ChangeArray(numbers);
   return 0; }
int ChangeArray (const int *numery)
{ int ix;s
for (ix=0; ix<6; ix++)
numery[ix] = 2*numery[ix];
return 0; }
```

Próba zmiany wartości.

```
for (ix=0; ix<6; ix++)
{
numery[ix] = 2*numery[ix];
printf("%d  ",numery[ix]);}
printf("\n");
```

```
In function 'ChangeArray':
[Error] assignment of read-only location '*(numery + (sizetype)((long long unsigned int)ix * 4ull))'
```

9.2. Dostęp do danych, p9-2b

Modyfikator const chroni argumenty funkcji przed przypadkową niepożądaną modyfikacją.

```
void code (const char *str);
int main (void)
{  char *tab = "Politechnika";
   printf("%s\n",tab);
//code ("Politechnika");
   code (tab);
   return 0;
}
void code(const char *str)
{  while (*str)
   printf ("%c", (*str++)+3);
   printf ("\n");
}
```

Stała napisowa.

Tablica jest argumentem.

Tu nie ma modyfikacji tablicy, lecz inkrementacja wskaźnika (kopii)
Kod ASCII znaku z tablicy jest powiększany o 3.

```
Politechnika
Srolwhfkqlnd

--------------
```

9.3. Klasy pamięci, auto, extern

Dostępne klasy pamięci:
- **auto**,
- **extern**,
- **static**,
- **register**.

Klasa **auto** jest automatyczna i dlatego zapis jest pomijany.
Modyfikator auto stwierdza, że dana zmienna znika po wyjściu z zakresu życia (przydzielona jej pamięć jest zwolniona), np. zmienne lokalne.
Modyfikator **extern** sygnalizuje, że dana zmienna globalna jest zadeklarowana i ewentualnie zainicjowana w innym pliku z postacią źródłową programu.
Stosujemy w programach o postaci źródłowej podzielonej na wiele plików, np.:

extern double a, b, c; **extern int inum;**

9.3. Klasy pamięci, static

Modyfikator klasy pamięci **static** pełni dwie funkcje:

- ograniczenie widzialności zmiennej globalnej wyłącznie do pliku, w którym został zadeklarowany,

```
static double a=700., b=77.;
```

- do zmiennych lokalnych, zmienne takie mają **status trwały** (nie znikają po wyjściu z funkcji, w której są zadeklarowane).

Wartości są zapamiętywane między kolejnymi wywołaniami funkcji.

W ten sposób można uniknąć stosowania zmiennych globalnych definiowanych na zewnątrz funkcji.

9.3. Klasy pamięci, static, p9-3a1, p9-3a2

Oddziaływanie modyfikatora static na zmienne lokalne.

```
void nowawar (int nie);
void dummy (int nie);
int main (void)
{  int obl;
   obl = 1;    Wywołania funkcji.
   nowawar (obl);
   obl = 2          ;
   dummy (obl);
   obl = 3;
   nowawar (obl);
   return 0; }
```

```
nie = 1; inner_val = 222
    5
nie = 3; inner_val = 222
```

*Zastosowano modyfikator **static**.*

```
void nowawar(int nie)
{ static int inner_val;
   if (nie == 1)
   inner_val = 111;
   printf ("nie = %d; inner_val =
   %d\n", nie, inner_val); }
void dummy (int nie)
{  int sum, one = 1, two = 2;
   sum = one + two + nie;
   printf("%5d\n", sum); }
```

*Gdy nie zastosowano modyfikatora **static**.*

```
nie = 1; inner_val = 222
    5
nie = 3; inner_val = 1
```

9.3. Klasy pamięci, register

Modyfikator klasy pamięci **register** jest zleceniem umieszczenia zmiennych w rejestrach procesora (jeśli to możliwe):

- szybszy dostęp,
- nie można stosować do zmiennych globalnych (te zawsze rezydują w pamięci).

Można wykonać program, który będzie mierzył czas działania programu, funkcja biblioteczna **time()** obsługiwana przez plik nagłówkowy **time.h**.
Funkcja ta zwraca liczbę sekund, ale wynik ma zbyt małą rozdzielczość dla prostego programu i szybkiego komputera.

9.4. Operatory bitowe

Wykonują operacje na poszczególnych bitach zmiennych będących ich argumentami.

Argumentami tych zmiennych mogą być tylko zmienne typu całkowitego od **char** do **long**.

Najwyższy priorytet

Operator	Działanie
~	Inwersja (negacja) wartości bitów
<< >>	Przesuniecie w lewo lub w prawo
&	Iloczyn logiczny poszczególnych bitów
^	Suma modulo 2 (EXOR) poszczególnych bitów
|	Suma logiczna poszczególnych bitów

Operatory bitowe wykonują operacje logiczne na odpowiadających sobie bitach argumentów.

9.4. Operatory bitowe

Iloczyn bitowy:	arg. 1	0101 0011
	arg. 2	1100 1001
	iloczyn bitowy ”**&**”	0100 0001
Suma bitowa:	arg. 1	0101 0011
	arg. 2	1100 1001
	suma bitowa ”**|**”	1101 1011

Operator przesunięcia, podajemy o ile bitów ma być przemieszczona w lewo lub w prawo zawartości argumentu.

ival << 3

0101 0011 po **1001 1000**

Pamiętaj:

- bity przesunięte na zewnątrz znikają,
- miejsce opróżnione zapełniane są przez zera.

Uwaga! Język C nie ma operatora przesunięcia cyklicznego.

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

NAZWA PRZEDMIOTU

Programowanie strukturalne

Temat wykładu 10.

Łańcuchy znakowe, tablice, działania na łańcuchach, wczytywanie i wypisywanie łańcuchów.

dr hab. inż. Jerzy Montusiewicz, prof. PL

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

10. Agenda

10.1. Łańcuchy znakowe.

10.2. Tablice znakowe.

10.3. Wypisywanie łańcuchów.

10.4. Wczytywanie łańcuchów.

10.5. Działania na łańcuchach.

10.1. Łańcuchy znakowe

Łańcuchy znakowe to napisy, które są ciągiem następujących po sobie znaków (typ char). Ciąg zakończony jest znakiem „null”, inaczej „null-bajtem”, „\0”.

Łańcuchy znakowe (stałe łańcuchowe) można zapisać między dwoma cudzysłowami, np. ”Katedra Informatyki”. Łańcuch może zawierać znaki „spacji”.

W pamięci łańcuch jest zapisywany jako tablica typu char:

char napis2[]= ”Katedra Informatyki”;

lub wskaźnik do ciągu znaków:

char *napis1 = ”Katedra Informatyki”;

przypisuje adres i umieszcza w obszarze danych.

```
#define NAPIS "Polibuda_Lublin_PL"
```

definicja przez makro.

10.1. Łańcuchy znakowe, p10-1a

Zapis wskaźnikowy deklaruje wskaźnik na stały obszar pamięci. Próba modyfikacji łańcucha, np. napis[0] = ’k’ (mała litera) może spowodować nieprzewidywalne skutki.

Długi napis można zapisać w kilku linijkach umieszczając znak kontynuacji ’\’. Wypisanie takiego łańcucha nastąpi w jednej linii jeśli nie zastosujemy znaku tworzącego nową linię ’\n’.

Do poszczególnych elementów łańcucha znakowego mamy dostęp przez indeksację tabeli lub inkrementację / dekrementację wskaźnika.

Wyznaczanie długości makra.

```
#include <stdio.h>
#define NAPIS "Polibuda_Lublin_PL"
int main() { int max_r;
max_r = sizeof(NAPIS);
printf("\n   dlugosc napisu = %d\n\n", max_r);
max_r = strlen(NAPIS);
printf("\n   dlugosc napisu = %d\n\n", max_r);
return 0;  }
```

```
   dlugosc napisu = 19
   dlugosc napisu = 18
```

10.1. Łańcuchy znakowe

Zawartość standardowej biblioteki **string.h**, najważniejsze funkcje:

strchr() – określa położenie znaku wewnątrz łańcuchu.

strstr() – określa położenie jednego łańcucha znaków wewnątrz drugiego.

strcmp() – porównuje dwa łańcuchy znaków.

strlen() – określa długość łańcucha.

strcpy() – kopiuje jeden łańcuch do drugiego.

strcat() – łączy dwa łańcuchy znaków.

strspn() – określa liczbę znaków pasujących.

```
printf("Ten dlugi tekst pomimo, ze zajmuje dwie linie zostanie \
wypisany w jednej linii.");
```

10.2. Tablice znakowe

Inicjalizacja tabel znakowych pozwala na wprowadzenie napisów w linii, w której deklarowany jest typ, nazwa i rozmiar tablicy.

1. W takiej sytuacji możemy stosować znak przypisania '**=**'.
2. Nie musimy definiować rozmiaru tablicy, wystarczy zapis '**[]**'.
3. Gdy mamy tablicę jednowymiarową nie musimy stosować nawiasów **{}**.
4. W przypadku wielu napisów, nie musimy podawać rozmiaru odnoszącego się do liczby wierszy, należy podać rozmiar odnoszący się do liczby kolumn, np. '**[][20]**'.
5. W takim przypadku łańcuchy znakowe muszą być oddzielone przecinkami '**,**' oraz zgrupowane w blok ograniczony nawiasami '**{}**'.
6. Po nawiasie klamrowym **}** piszemy znak ; → **};**.

10.2. Tablice znakowe, p10-2a

Inicjalizacja tabel znakowych.

```
#include <stdio.h>
int main()
{  char tytul[] = "tablica danych";
   char info[][20] = {"tablica nr 1\n", "tablica nr 2\n", "tablica nr 3\n"};
   int row, max_r;
   max_r = sizeof(info)/20;
   printf("\n   liczba wierszy = %d\n\n", max_r);
   printf(tytul);
   for(row=0; row<max_r; row++)
   printf (info[row]);
   return 0;
}
```

Stała napisowa, nie musimy stosować {}

Pominięto lewy skrajny wymiar []

Obliczanie liczby wierszy.

Wydruk, indeksowanie tablicy.

```
   liczba wierszy = 3

tablica danych
tablica nr 1
tablica nr 2
tablica nr 3
```

10.3. Wypisywanie łańcuchów, 10-3a

Funkcja biblioteczna printf(). Do wypisywania wartości typu **int**, **float**, ale również **znaków** i **łańcuchów znakowych**.

printf("%c", zn); – gdy **zn** jest typu **char** (wydrukujemy znak),

printf("%s", zl); – gdy **zl** jest napisem (typu **char***).

Funkcja biblioteczna puts(str1).

- wysyła łańcuch na standardowe wyjście, napis **str1**,
- a następnie znak nowej linii **(\n)**,
- w przypadku błędu zwraca wartość **EOF**, zaś sukcesu liczbę nieujemną.

```
#include <stdio.h>
int main() {
puts ("Hello");
puts ("W\n orld");
return 0;   }
```

```
Hello
W
 orld
```

10.3. Wypisywanie łańcuchów, 10-3b

Funkcja fputs().

fputs("str1\n", stream);

fputs("str1\n", stdout);

- wpisuje łańcuch **str1** do pliku wskazanego przez zmienną **stream**,
- **stdout** – standardowe wyjście (monitor), w wersji bardziej ogólnej można podać inny strumień wyjściowy (nazwę pliku),
- nie dodaje znaku nowej linii '**\n**' (jak funkcja **puts()**.
- zwraca nieujemny wynik w przypadku powodzenia i **EOF** w przypadku wystąpienia błędu.

```
#include <stdio.h>
int main() {
fputs("Hello world!\n", stdout);
puts("Hello");
puts("Hello");
return 0;   }
```

```
Hello world!
Hello
Hello
```

10.4. Wczytywanie łańcuchów

Funkcja biblioteczna gets().

- brak sprawdzenia czy napis mieści się wewnątrz tablicy,
- pobiera znaki aż do napotkania znaku nowej linii '**\n**',
- dodaje na końcu łańcucha znak zerowy '**\0**',
- pobiera dane tylko ze standardowego wejścia (**stdin**).

Funkcja biblioteczna fgets ().

- wczytuje informacje z dowolnego wejścia oraz **kontroluje** liczbę wprowadzonych elementów (np. pliki),
- gdy odczyta znak nowej linii to pozostawia go w łańcuchu (inaczej niż **gets()**)

fgets(bufor, liczba_znakow, strumien_wej);

- **bufor** – tablica znakowa do której wczytujemy dane,
- **liczba_znakow** – liczba wczytywanych znaków,
- **strumien_wej** – nazwa wejścia (**stdin** – gdy z klawiatury),

Pamiętaj! Standardowe wyjście to **stdout**, a niebuforowane wyjście komunikatów o błędach – **stderr**.

10.4. Wczytywanie łańcuchów, p10-4a

Tablice napisów, przechowywane wierszami.

```
Wypelnianie tabeli 'tekst'
Zakoncz wiersz w miejscu pierwszej spacji
0:  Katedra
1:  Informatyki
2:  2019/2020
3:
Drukowanie tablicy 'tekst'- wiersz po wierszu
Katedra
Informatyki
2019/2020
```

```
#include <stdio.h>
#define  ROWS  100
#define  COLS  80

int main()
{ char tekst[ROWS][COLS];
int row, line;
printf("Wypelnianie tabeli 'tekst'\n");
printf("Zakonczenie, spacja w nowym wierszu\n");
for (row= 0; row<ROWS; row++) {
   printf ("%d:  ", row); gets (tekst[row]);
   if (tekst[row][0]==' ')  break;    }
printf("Drukowanie tablicy 'tekst'- wiersz po wierszu\n");
for (line=0; line<row; line++) printf ("%s\n", tekst[line]);
return 0;
}
```

Czytanie łańcucha.

Druk tablicy 'tekst' - wiersz po wierszu.

Pojedynczy indeks - wybieranie wierszy.

10.4. Wczytywanie łańcuchów, scanf()

Funkcja biblioteczna **scanf()**.

- funkcja umożliwia pobranie jednego słowa,
- gdy użyjemy specyfikatora **%s** łańcuch obejmuje wszystkie znaki (bez znaku niedrukowanego)

```
scanf("%s", tablica);
```

Takie działanie może doprowadzić do przepełnienia bufora

- gdy np. **%19s** wczytuje 19 znaków, lub do znaku niedrukowalnego, rozmiar tablicy [20], ostatnie miejsce na znak końca łańcucha

```
scanf("%19s", tablica);
```

Podajemy liczbę o 1 mniejszą niż deklaracja tablicy.

10.4. Wczytywanie łańcuchów, scanf(), p10-4b

Funkcja **scanf()** zwraca liczbę poprawnie wczytanych zmiennych lub EOF jeżeli nie ma już danych w strumieniu lub nastąpił błąd.

Sprawdzając czy wartość zwrócona nie jest równa ’0’ można zbudować kod obsługujący błąd danych wprowadzonych przez użytkownika.

Obsługa błędu wpisania niepoprawnej danej.

```
#include <stdio.h>
int main(void)
{  int wynik, a;
  printf("Wprowadz liczbe int ");
  do  {    wynik = scanf("%d", &a);
  printf("Wartosc zwrocona przez scanf() = \
          %d\n",  wynik);
  if (wynik) printf("Wartosc a*a = %d\n", a*a);
  else  wynik = scanf("%s");
       }  while (wynik!=EOF);
  return 0;
}
```

```
Wprowadz liczbe int 3
Wartosc zwrocona przez scanf() = 1
Wartosc a*a = 9
7
Wartosc zwrocona przez scanf() = 1
Wartosc a*a = 49
e
Wartosc zwrocona przez scanf() = 0
U
Wartosc zwrocona przez scanf() = 0
11
Wartosc zwrocona przez scanf() = 1
Wartosc a*a = 121
```

10.4. Wczytywanie łańcuchów, p10-4c

Tablice znakowe. Wczytywanie z użyciem funkcji gets() i fgets(). Łańcuchy znakowe z terminatorem w postaci null-bajtu).

```
#include <stdio.h>
int main()          Deklaracja tablicy.
{  char str[15];
   printf("Wprowadz lancuch, \
uzyj funkcji gets(): \n");
gets(str);          Wczytanie łańcucha.
printf ("%s\n", str);
printf("  lancuch bez sformatowania\n");
printf (str);       Bezpieczne wczytanie,
printf ("\n");      bez protestów kompilatora
   printf("Wprowadzanie drugiego lancucha,\n \
   uzyto funkcji fgets(): \n");
   fgets(str, 15, stdin);
   printf ("%s\n", str);
return 0;                  Wczytanie łańcucha, tylko
}                          15 znaków.
```

```
Wprowadz lancuch, uzyj funkcji gets():
Politechnika Lubelska
Politechnika Lubelska
 lancuch bez sformatowania
Politechnika Lubelska
Wprowadzanie drugiego lancucha,
        uzyto funkcji fgets():
Politechnika Lubelska
Politechnika L
```

10.5. Działania na łańcuchach, podstawienie, łączenie

Podstawienie stałych napisowych do napisów lub napisów do napisów.
Np. stałej napisowej: **”Napis nowy”**. Ciąg znaków w cudzysłowach.
Operacja podstawienia (kopiowania**)**

strcpy (dokad, skad);

- **dokad** – to tablica znakowa lub napis,
- **skad** – to napis lub stała napisowa,

Brak sprawdzenia czy napis się zmieści.

Operacja łączenia (konkatenacja**)**

strcat (napis1, napis2);

- dołącza do dotychczasowej zawartości **napis1** zawartość **napis2**,
- pierwszy argument zmienia się, drugi pozostaje bez zmian,

napis1 nie może być stałą napisową.

Pamiętaj! Musimy dołączy ć bibliotekę: string.h

10.5. Działania na łańcuchach, podstawienie, łączenie, p10-5a

Napisy kopiowanie/podstawienie. Drukowanie tablic znakowych.

```
#include <stdio.h>
#include <string.h>
int main()
{  char str0[80], str1[80], str2[40];
printf("Kopiuj stala napisowa do lancucha\n");
strcpy(str1, "Witam ");
   printf ("%s\n", str1);
strcpy(str2, "tutaj");
   printf("%s\n", str2);
printf ("Konkatenacja (laczenie) \
lancucha nr 1 i nr 2;\n ");
printf ("Wynik lancucha nr 1:\n");
strcat(str1, str2);          Łączenie
   printf ("%s\n", str1);    napisów
   printf("Kopiowanie lancucha do lancucha:\n");
strcpy(str0, str1);          Kopiowanie łańcuchów
   printf ("%s\n", str0);
}
```

```
Kopiuj stala napisowa do lancucha
Witam
tutaj
Konkatenacja (laczenie) lancucha nr 1 i nr 2;
 Wynik lancucha nr 1:
Witam tutaj
Kopiowanie lancucha do lancucha:
Witam tutaj
```

10.5. Działania na łańcuchach, porównywanie, p10-5b

Porównywanie napisów.

```
strcmp (napis1, napis2);
```

- funkcja zwraca wartość **0** gdy napisy są równe (słownikowo, inaczej mówiąc leksykograficznie),
- wartość **dodatnią** gdy **napis1** jest słownikowo większy (dalej w słowniku),
- wartość **ujemną** gdy **napis1** jest mniejszy (bliżej początku słownika),
- w porównaniu korzystamy z kodu ASCII (cyfry **<** duże litery **<** małe litery).

10.5. Działania na łańcuchach, porównywanie, p10-5b

Napisy: porównywanie leksykograficzne.

```
int main()   { char str1[80], str2[40];
   printf("Zamkniecie aplikacji \n\
przez wpisanie 'quit'\n");
   for (; ;)
   { printf ("Wprowadz pierwszy lancuch:\n");
// Funkcja fgets() czyta i zwraca newline.
fgets(str1, 80,stdin);
if (!strcmp("quit\n", str1))   break;
printf ("Wprowadz drugi lancuch:\n");
fgets (str2, 80, stdin);
if (strcmp(str1, str2)>0)
printf ("\n%s wiekszy slow. niz %s\n", str1, str2);
   else if (strcmp(str1, str2)==0)
printf ("\n%s rowny %s\n", str1, str2);
   else
printf ("\n%s mniejszy slow. niz %s\n", str1, str2);
   } return 0; }
```

```
Zamkniecie aplikacji
        przez wpisanie 'quit'
Wprowadz pierwszy lancuch:
katedra
Wprowadz drugi lancuch:
informatyki

katedra
 wiekszy slow. niz informatyki

Wprowadz pierwszy lancuch:
12
Wprowadz drugi lancuch:
aga

12
 mniejszy slow. niz aga

Wprowadz pierwszy lancuch:
quit
```

10.5. Działania na łańcuchach, długość łańcucha, p10-5c

Generowanie napisu wypisywanego w odwrotnym kierunku (na wspak). Wyznaczanie długości łańcucha znakowego.

```
int main()
{  char str[80];   int i;
printf("Zamkniecie aplikacji \n\
przez wpisanie 'quit'\n");
for (; ;)
 {printf ("Wprowadz 1. lancuch:\n");
   // Funkcja gets() czyta i zwraca \
      napis bez terminatora newline.
   gets (str);
   if (!strcmp("quit", str)) break;
   for (i=strlen(str)-1; i>=0; i--)
            printf ("%c ", str[i]);
            printf ("\n");
  } return 0;
}
```

```
Zamkniecie aplikacji
 przez wpisanie 'quit'
Wprowadz 1. lancuch:
Politechnika
a k i n h c e t i l o P
Wprowadz 1. lancuch:
niski strop
p o r t s   i k s i n
Wprowadz 1. lancuch:
quit
```

Przeglądanie napisu wstecz od indeksu minus 1 czyli indeksu ostatniego znaku w napisie.

10.5. Działania na łańcuchach, terminator napisu, p10-5d

Napis jest konwertowany na wersaliki. Null-bajt jako zakończenie pętli przetwarzania napisu.

*Funkcja **gets()** czyta i zwraca napis bez terminatora newline (\n).*

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>  /* obsluga funkcji toupper */
int main() { char str[80];        int i;
printf("Zamkniecie aplikacji\n\
 przez wpisanie 'quit'\n");
for (; ;) {
   printf ("Wprowadz lancuch:\n");
   gets(str);
   if (!strcmp("quit", str))        break;
   for (i=0; str[i]; i++)
            str[i] = toupper(str[i]);
            printf ("%s\n", str); }
return 0;   }
```

*Sprawdzamy czy kod ASCII znaku **str[i]** jest różny 0.*

```
Zamkniecie aplikacji
 przez wpisanie 'quit'
Wprowadz lancuch:
katedra informatyki 2019
KATEDRA INFORMATYKI 2019
Wprowadz lancuch:
Agnieszka
AGNIESZKA
Wprowadz lancuch:
quit
```

10.5. Działania na łańcuchach, terminator napisu, p10-5e

Napis jest konwertowany na małe litery. Null-bajt jako zakończenie pętli przetwarzania napisu.

*Funkcja **gets()** czyta i zwraca napis bez terminatora newline (\n).*

```
#include <stdio.h>
#include <ctype.h>
int main()
{  char str[80], *ptr;
   int ielem;
printf ("WERSJA WSKAZNIKOWA\n");
printf (”Napisz duzymi literami: \n");
          gets (str);
   printf ("Napis malymi literami: \n");
   ptr = str;
   while (*ptr)
   printf ("%c", tolower(*ptr++));
   printf("\n"); return 0;
}
```

*Inicjalizacja wskaźnika **ptr** adresem początkowego elementu tablicy **str***

ptr = &str[0] *, nazwa tablicy jest wskaźnikiem*

*Koniec pętli **while**, gdy wskaźnik "ptr" osiąga ’\0’.*

```
WERSJA WSKAZNIKOWA
Napisz duzymi literami:
KATEDRA INFORMATYKI 2019
Napis malymi literami:
katedra informatyki 2019

------------------------
```

*Funkcja bib. **tolower** przetwarza duże litery na małe*

10.5. Działania na łańcuchach, znaki pasujące, p10-5f

Określanie znaków pasujących w dwóch łańcuchach.
Funkcja **strspn()** zlicza od początku liczbę znaków w łańcuchu **s1**, które należą do łańcucha **s2** i zatrzymuje się na pierwszym niepasującym, którego nie liczy. Zwracana jest liczba zliczonych znaków.

size_t strspn(const char *s1, const char *s2);

```
#include <stdio.h>
#include <string.h>
int main()
{  int pasuj;
pasuj= strspn("Janeczek", "Janek");
printf("Janeczek i Janek,  wynik = %d\n",pasuj);
pasuj= strspn("manierka", "maniak");
printf("manierka i maniak, wynik = %d\n",pasuj);
   return 0;
}
```

```
Janeczek i Janek,  wynik = 4
manierka i maniak, wynik = 4
```

Porównywane łańcuchy w postaci stałych łańcuchowych.

10.5. Działania na łańcuchach, łańcuch pasujący, p10-5g

Określanie łańcuchu występującego w innym łańcuchu.
Funkcja **strstr()** odszukuje zdefiniowany ciąg znaków w innym łańcuchu oraz podaje jego adres w pamięci. Gdy w łańcuchu nie ma szukanego ciągu znaków, to funkcja zwraca wartość 0. W innym przypadku zwraca adres komórki pamięci, w której znajduje się pierwsza litera szukanego ciągu znaków.

```
char *strstr(const char *s1, const char *s2);
```

10.5. Działania na łańcuchach, łańcuch pasujący, p10-5g

Określanie łańcuchu występującego w innym łańcuchu.
Funkcja **strstr()** odszukuje zdefiniowany ciąg znaków w innym łańcuchu oraz podaje jego adres w pamięci. Gdy w łańcuchu nie ma szukanego ciągu znaków, to funkcja zwraca wartość 0. W innym przypadku zwraca adres komórki pamięci, w której znajduje się pierwsza litera szukanego ciągu znaków.

char *strstr(const char *s1, const char *s2);

***s1 –** przeszukiwany łańcuch,

***s2 –** poszukiwany łańcuch,

wartość zwracana – do pierwszego znaku **s1** dopasowania łańcucha **s2**,
– zerowy, jeśli łańcuch **s2** nie zawiera się w **s1**,
– **s1** jeśli **s2** ma zerową długość.

10.5. Działania na łańcuchach, łańcuch pasujący, p10-5g

Poszukiwanie łańcucha znaków w innym łańcuchu.

```
#include <stdio.h>
#include <string.h>
int main()
{ char array[] = "wydzial.informatyki.i.matematyki";
char wait_str[]="matyki";
if (strstr(array,wait_str))
{ printf("Szukany napis to: '%s' \n \
  adres pamieci: %#x \n",\
   wait_str,strstr(array,array));
   printf("napis od miejsca: %d,\
   ma %d liter",strstr(array,wait_str)-array, \
   strlen(wait_str)); }
else { printf(" Szukany napis to: '%s'\n",wait_str);
printf(" Brak wspolnego ciagu znakow.");}
return 0;
}
```

```
Szukany napis to: 'matyki'
   adres pamieci: 0x62fde0
napis od miejsca: 13, ma 6 liter
--------------------------------
```

```
Szukany napis to: 'xyz'
Brak wspolnego ciagu znakow.
--------------------------------
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny