Język ANSI C

część 9

wskaźniki - c.d

Jarosław Gramacki
Instytut Informatyki i Elektroniki

Przykłady podstawowych wyrażeń ze wskaźnikami

– jaki będzie wynik działania programu ?

```
void main(void)
{
int Tab[10] = {1, 2, 3, 7, -22, 9, 7, -3, 11, 25}, *p, *q;

                                  p    q
p = Tab;
q = &Tab[9];
while( *p++ != *q-- )     // najpierw przesuń potem pobierz wskazywaną
wartość
;                         // średnik dla czytelności TUTAJ (w nowej linii)
printf("\n%d ",*p);       // -22
printf("\n%d ",*q);       // 9
printf("\n%d ",*(p+1));   // 9
printf("\n%d ",*(q-1));   // -22
printf("\n%d ",*p+1);     // -21
printf("\n%d ",*q-1);     // 8
q++;                      // przesunięcie wskaźnika
printf("\n%d ",*q);       // 7
printf("\n%d ",*(p+1)-1); // 8
printf("\n%d ",*(q-3));   // 7
*q++;                     // przesunięcie wskaźnika z pobraniem wartości.
                             Ew. warning "Code has no effect"
printf("\n%d ",*q);       // -3
printf("\n%d ",*(q-3)+2); // -20
printf("\n%d ",*(q-3)==*(p+2)); // 0
}
```

analogicznie do:

```
int z = 7;
z;  // ???
```

Stałe i zmienne wskaźnikowe

– napisów nie można kopiować operatorem =

```
#include <stdio.h>
void main(void) {
char *str1 = "123";
char *str2 = "45";

printf("Ciag1=%s\n", str1);        // 123
printf("Ciag2=%s\n", str2);        // 45

str1 = str2;               // modyfikacja wskaźników, nie wartości
printf("Ciag1=%s\n", str1);        // 45
printf("Ciag2=%s\n", str2);        // 45

*str1 = *str2;             // kopia tylko pierwszego elementu
printf("Ciag1=%s\n", str1);        // 423
printf("Ciag2=%s\n", str2);        // 45
}
```

Uwaga:
literał tylko do odczytu (stała napisowa). Nie wolno zmieniać

w niektórych kompilatorach udaje się próba modyfikacji stałego literału !!!

Stałe i zmienne wskaźnikowe

– zmieniamy deklaracje tablic znakowych z `char *` na `char []`

```
#include <stdio.h>
void main(void) {
char str1[] = "123";
char str2[] = "45";

printf("Ciag1=%s\n", str1);        // 123
printf("Ciag2=%s\n", str2);        // 45

*str1 = *str2;
printf("Ciag1=%s\n", str1);        // 423
printf("Ciag2=%s\n", str2);        // 45

str1 = str2;                       // Error; L-value required
printf("Ciag1=%s\n", str1);        // ???
printf("Ciag2=%s\n", str2);        // ???
}
```

podobnie nie jest możliwe napisanie na przykład
```
str1++;
str2++;
```

analogicznie do:

```
int z = 7;
7 = 5; // ???
7++    // ???
```

Wskaźniki do znaków

– implementacja funkcji `strlen()` z biblioteki standardowej

```
int _strlen (char *s)
{
  int i;
  for (i=0; *s != '\0'; s++)
  // zwiększanie s całkowicie poprawne.
  // modyfikacja wskaźnika NIE ma wpływu na wartość wskaźnika w miejscu wywołania
  // funkcja strlen() zwiększa jedynie kopię wskaźnika
    i++;
  return i
}
```

co się stanie gdy TU będzie średnik ?

lepiej przenieść do pętli for(). dlaczego ?

```
int _strlen (char *s)
{
  char *p = s;            // lokalna kopia (kopii) wskaźnika
  while ( *p != '\0')
    p++;
  return (p - s)          // arytmetyka na wskaźnikach
}
// UWAGA: wskaźników NIE wolno do siebie dodawać
```

suma dwóch adresów prawie zawsze jest liczbą "bez wartości"

```
int _strlen (char *s)
{
  int i = 0;
  for (; *s++; i++) // dopóki nie koniec napisu ( ... *s++ != '\0' ... )
  ;
  return i - 1 }
```


Wskaźniki do znaków

– implementacja funkcji `strcpy()` z biblioteki standardowej

```
// notacja "tablicowa" (używamy indeksów)
void _strcpy (char *s, char *t)
{
  int i;
  while (( s[i] = t[i] ) != '\0') // kopiuj t do s
    i++
}
```

```
char* _strcpy (char *s, char *t)   // "w duchu" biblioteki standardowej
{
  int i;
  while (( s[i] = t[i] ) != '\0') // kopiuj t do s
    i++;
  return s;
}
```

zwrócenie z funkcji jej "efektów działania" jest wygodne w wywołaniach tej funkcji.
Np: `wynik = strcpy(s1, s2); zamiast`

```
strcpy(s1, s2);
wynik = s1;
```

```
// notacja "wskaźnikowa" 1
// (używamy wskaźników)
void _strcpy (char *s, char *t)
{
  while ( (*s = *t) != '\0')
    {
    s++; t++;
    }
}
```

Wskaźniki do znaków

– implementacja funkcji `strcpy()` z biblioteki standardowej

```
// notacja "wskaźnikowa" 2
void _strcpy (char *s, char *t)
{
  while ( (*s++ = *t++) != '\0')  // jeszcze krócej: != '\0' zbędne
  ;
}
```


```
// notacja "wskaźnikowa" 2
void _strcpy (char *s, char *t)
{
  while ( (*s++ = *t++) != '\0')  // jeszcze krócej: != '\0' zbędne
  ;
}
```


– przykład wywołania

```
int main (void)
{
...
return 0;
}
```


Wskaźniki do znaków

– implementacja funkcji `strchr()` z biblioteki standardowej

```
// szukaj znaku w napisie
char* _strchr (char *s, char ch)
{
  while ( (*s++ != ch) && (*s) )   // wewnętrzne nawiasy zbędne
  ;                                // średnik TU
  return (*s ? --s : (s = NULL));  // zwraca wskaźnik do znalezionego znaku ch
                                   // lub NULL gdy nie znaleziono ch w napisie
}

int main (void)
{
  char *str = "ABCDE";
  char ch = 'C';
  printf("Od znaku %c: %s", ch, _strchr (str, ch) );
  return 0;
}
```

bardzo skondensowane wyrażenia - bardzo typowe dla C

– przykład wywołania

```
int main (void)
{
...
return 0;
}
```

Wskaźniki do znaków

– implementacja funkcji `strcmp()`

```
int strcmp (const char *s1, const char * s2)
{
  while (*s1 == *s2)
  {
   if   (*s1 == '\0') return 0;
   s1++;
   s2++;
  }
  return (*s1 - *s2);
}
```

zakłada się uporządkowanie znaków w tablicy znaków

– przykład wywołania

```
int main (void)
{
...
return 0;
}
```


Wskaźniki do znaków

– odwróć znaki w argumencie, wersja wskaźnikowa
– do tej pory NIE modyfikowaliśmy argumentu funkcji. Teraz to robimy !

```
// odwróć znaki w napisie (w notacji wskaźnikowej)

char* reverse (char *s)
{
  char *pom = s, *kon = s + strlen(s) - 1; // wsk. na pocz. i koniec napisu
  char c;

  while ( pom < kon )                      // dopóki wskaźniki nie mina się
  {                                        // (pom < kon)
  c = *pom;
  *pom = *kon;
  *kon = c;
  ++pom;
  --kon;
  }
return s;
}
```

Jak poprzednio: dla wygody wywoływania funkcji zwracamy wartość (zmodyfikowanego) argumentu

dla deklaracji `char *str = "ABCDE"` będzie błąd. Dlaczego ? Modyfikacja stałego literału

```
int main (void) {
  char str[] = "ABCDE";
  printf("%s\n", reverse(str) );   // oryginalny napis str jest modyfikowany
  printf("%s\n", reverse(str) );   // dwa razy
  return 0;
}
```

Wskaźniki do znaków

– odwróć znaki w argumencie, wersja tablicowa

```
// odwróć znaki w napisie
// w notacji tablicowe

char* reverse ( char s[] )
{
  int c, i, j;

  for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
  // dopóki indeksy się nie miną (i < j)
  {
  c = s[i];
  s[i] = s[j];
  s[j] = c;
  }
return s;
}
```

być może na początek łatwiej analizować tą wersję

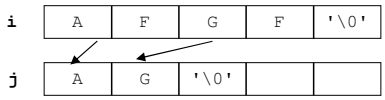
Wskaźniki do znaków

– usuń znak(i) z tekstu

```
int UsunZnak (char *s, char c)  /* usuwa wszystkie znaki c z tekstu s */
{
  int i, j;
  for (i=j=0; s[i] != '\0'; i++)
    if ( s[i] != c )
      s[j++] = s[i];  // gdy napotkamy c, to i przesuwa się (i++) a j nie


  s[j] = '\0';
  return 0;
}
```

Tu można by też postąpić "w duchu" biblioteki standardowej i zwrócić s

Nie zapomnij o tym !




```
int main (void)  {
  char tekst[50];   char znak;

  printf ("\nPodaj tekst: ");
  gets(tekst);
  printf ("\nPodaj znak: ");
  znak = getchar();

  UsunZnak (tekst, znak);
  puts(tekst);
  getchar();  getchar();
  return 0;
}
```

Wskaźniki do znaków

– wytnij podciąg z ciągu znaków

obsługa maksymalnie wielu błędnych sytuacji

```
char *CutStr(char *t, int startp, int count)
{
  int i;
  if ( !t ) return 0;                                  // czy string istnieje
  if ( startp < 0 || startp > strlen(t)) return 0;     // warunki na startp
  if ( count < 0 || count > strlen(t) - strtp) return 0; // warunki na count

  // właściwa pętla kopiująca znaki na nowe pozycje
  // +1 gwarantuje przekopiowanie znaku '\0'

  for (i = startp; i < strlen(t) - count + 1; i++)
    t[ i ] = t[ i + count ];

 return t; }
```

```
int main(void)
{
char Tab[] = "1234567890";

char *tmp;
printf("%s\n", Tab);
if( tmp = CutStr(Tab, 2, 4) ) printf("%s\n", tmp);
 else printf("\nERROR\n");:;
getchar();                                        // czeka na jakiś klawisz
return 0;
}
```


Kilka uwag

– krótki, rzeczowy komentarz opisujący działanie funkcji

– można uznać, ze taki komentarz jest bezwzględnie konieczny

– rzadko kiedy programista pracuje samodzielnie !!!

```
// Funkcja CutStr() wycina z ciągu znaków podciąg o długości count
// rozpoczynający się na startp-tym znaku
// (elementy ciągu numeruje się od zera).
// Funkcja zwraca wskaźnik do ciągu znaków, z którego usunięto elementy. W
// przypadku błędu funkcja zwraca NULL.

char *CutStr(char *t, int startp, int count)
{
 ...
}
```

Kilka uwag

- nie stosować wskaźników (notacji wskaźnikowej) "na siłę"
- wydaje się, że notacja tablicowa (indeksowanie) jest czytelniejsza od notacji wskaźnikowej
- przykłady użycia wskaźników "na wyrost"

```
int Tab[3];
int i, j;
int *ptr;

for (i = 0; i < 3; i++) printf("%d ", *(Tab + i));
// czytelniej: Tab[i]

for (ptr = Tab; ptr < Tab + 3; ptr++) printf("%d ", *ptr);
// j.w.

// tablica "od tyłu"
for (ptr = Tab + 2; ptr >= Tab; ptr--) printf("%d ", *ptr);
// czytelniej: for (i = 2; i >= 0; i--) printf("%d ", Tab[i]);

for (i = 1; i < 4; i++) printf("%d ", Tab[i-1]);
// nieuzasadnione indeksowanie od 1
```


Wskaźniki do znaków

- konwersja napisu na liczbę całkowitą

```
#include <stdio.h>
#include <ctype.h>
int Ascii2Int (char s[]);

void main(void) {
  char liczba[] = "       -123";
  printf("Liczba jako string : [%s]\n", liczba);
  printf("Liczba po konwersji: %d", Ascii2Int(liczba));
}

int Ascii2Int (char s[]) {
  int i, n, znak;
  for(i=0; isspace( s[i] ); i++)
    ;                              //pomija biale znaki

  znak = (s[i] == '-') ? -1 : 1;   //bada znak liczby
  if(s[i] == '+' || s[i] == '-')   //przeskocz znak liczby
    i++;

  for(n=0; isdigit (s[i]); i++)    //konwersja
    n = 10 * n + (s[i] - '0');

  return znak * n;
}
```

przeanalizuj sposób konwersji "znaków na liczbę"

Wskaźniki do znaków

– konwersja napisu na liczbę zmiennoprzecinkową

– własne atof(): Ascii2Float()

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
double Ascii2Float (char s[]);

void main(void)
{
  char liczba[] = "       -123.456789e-5";
  printf("Liczba jako string : [%s]\n", liczba);
  printf("Liczba po konwersji: %.10f", Ascii2Float(liczba));
}
```


Wskaźniki do znaków

```
double Ascii2Float (char s[]) {
  double val, power, wykladnik;
  int i, znak_liczby, znak_wykl;
  for(i=0; isspace( s[i] ); i++)
    ;                                   // pomija białe znaki
  znak_liczby = (s[i] == '-') ? -1 : 1;// bada znak liczby
  if(s[i] == '+' || s[i] == '-')        // przeskocz znak liczby
    i++;
  for(val=0.0; isdigit (s[i]); i++)     // konwersja tego co przed kropka
    val = 10.0 * val + (s[i] - '0');
  if(s[i] == '.')                       // gdy jest kropka...
    i++;                                // ...to ja przeskocz
  for(power=1.0; isdigit (s[i]); i++)   // konwersja tego co po kropce
  {
    val = 10.0 * val + (s[i]-'0');
    power *= 10; }
  if(s[i] == 'e' || s[i] == 'E')        // tzn., ze notacja naukowa
    i++;                                // przeskocz ten znak
  else                                  // notacja nie jest naukowa - koniec
    return( znak_liczby * val/power);

  znak_wykl = (s[i] == '-') ? -1 : 1;   // znak wykladnika
  if(s[i] == '+' || s[i] == '-')
    i++;                                // przeskocz znak wykladnika
  wykladnik = (s[i] - '0') * znak_wykl;

  return ((znak_liczby * val/power) * pow(10,wykladnik)); }
```

Wskaźniki do znaków – dynamiczna alokacja pamięci

– wstaw znaki do napisu

– tu w przeciwieństwie do poprzednich przykładów nie obędzie się bez dynamicznej alokacji pamięci gdyż:

– nie można przewidzieć wielkości danych w trakcie pisania programu

```
// wstaw podnapis do napisu

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

char* wstaw (char* s, char* z, int poz);   // wstaw z do s począwszy
                                           // od pozycji poz

int main(void)
{
  char* test = "Ala##kota", *wynik;        // ## oznacza spacje

  wynik = wstaw( test, "ma", 4 );
  printf("1: %s\n", wynik);

  free (wynik);                    // gdyż alokacja pamięci we wstaw()
  printf("2: %s\n", wynik);        // printf() po free() ???
  return 0;
}
```

nienajlepszy pomysł
Dlaczego ?


```
char *wstaw(char *source, char *znaki, int poz)
{
  char *work;
  int i;

  work = (char *)malloc( strlen(source) + strlen(znaki) + 1 ); // +1 na '\0'

  for (i=0; i<poz+1; i++)          // kopiuj znaki do pozycji poz
    *(work+i) = *(source+i);       // czytelniej po prostu: work[i] = source[i];
  // *work++ = *source++           // źle, przesuwamy wskaźnik trwale

  for (i=0; i<strlen(znaki); i++)  // doklej podciąg
    work [poz+i] = znaki [i];      // notacja tablicowa czytelniejsza

  for (i=poz; i<strlen(source); i++)               // reszta napisu
    work [i + strlen(znaki)] = source [i];

  work [strlen(source) + strlen(znaki)] = '\0';    // '\0' na końcu

  // free(work);                          // źle, nie będzie co zwrócić !!!
  return work;   }
```

alokacja dynamiczna: pamięć na stercie (ang. heap). Nie mylić ze stosem (ang. stack) !!!

```
source: A l a # # k o t a              // dla wstaw( test, "ma", 4 );
work:   A l a # m a # k o t a
```

source : **"Ala kota"** przez cała sesję
*source: **"A"** przez cała sesję (bo nie modyfikujemy wskaźnika)
*work : **"A"** przez cała sesję (bo nie modyfikujemy wskaźnika)
work : **"A", "Al", "Ala"**, itd.

Stos i sterta

- **Stos** (ang. stack) jest obszarem pamięci, który zostaje automatycznie przydzielony do wykorzystania dla programu.
- Na stosie egzystują wszystkie zmienne zadeklarowane jawnie w kodzie (szczególne te lokalne w funkcjach).
- Wielkość stosu można regulować w ustawieniach linkera.
- Jest on także używany do przekazywania parametrów do funkcji.
- Częsty błąd: błąd przepełnienia stosu (ang. stack overflow). Występuje on zwykle wtedy, gdy nastąpi zbyt wiele wywołań funkcji. Częsty w algorytmach rekurencyjnych.

- **Sterta** (ang. heap) to cała pamięć dostępna dla programu i mogąca być mu przydzielona do wykorzystania.

- **Różnice**:
- Rozmiar stosu jest ustalany raz na zawsze podczas kompilacji programu i nie zmienia się w trakcie jego działania. Wszelkie dane, jakie są na nim przechowywane, muszą więc mieć stały rozmiar - jak na przykład skalarne zmienne, struktury, tablice o ustalonym rozmiarze.
- Kontrolą pamięci sterty zajmuje się programista. Może przyznać swojej aplikacji odpowiednią jej ilość w danej chwili, podczas działania programu (alokując pamięć dynamicznie).

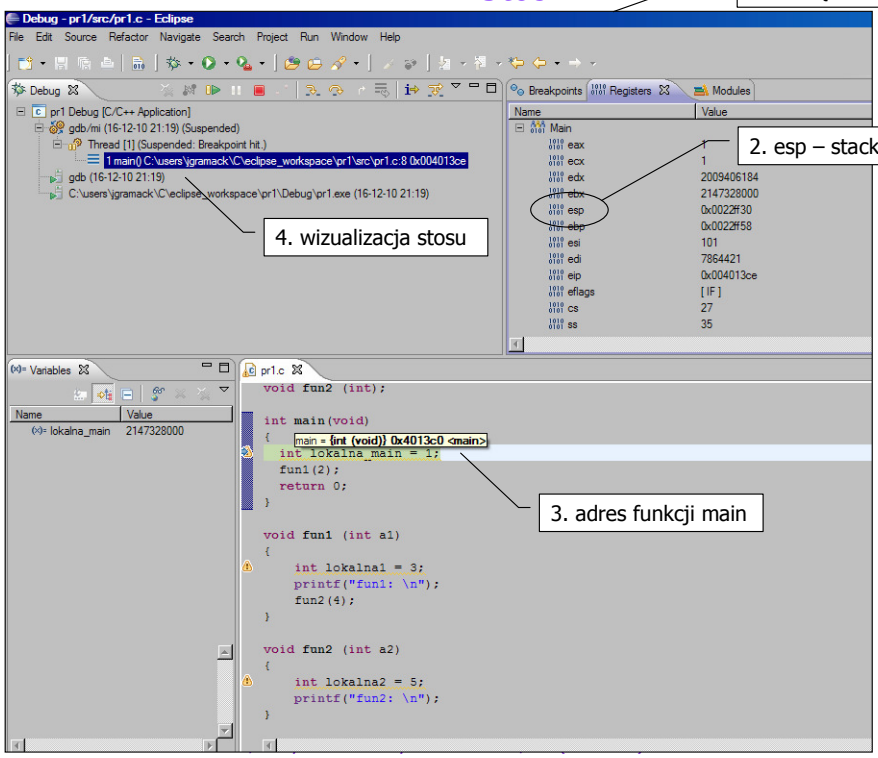
Stos

- Ilustracja **stosu.** Przykład "zwykły"

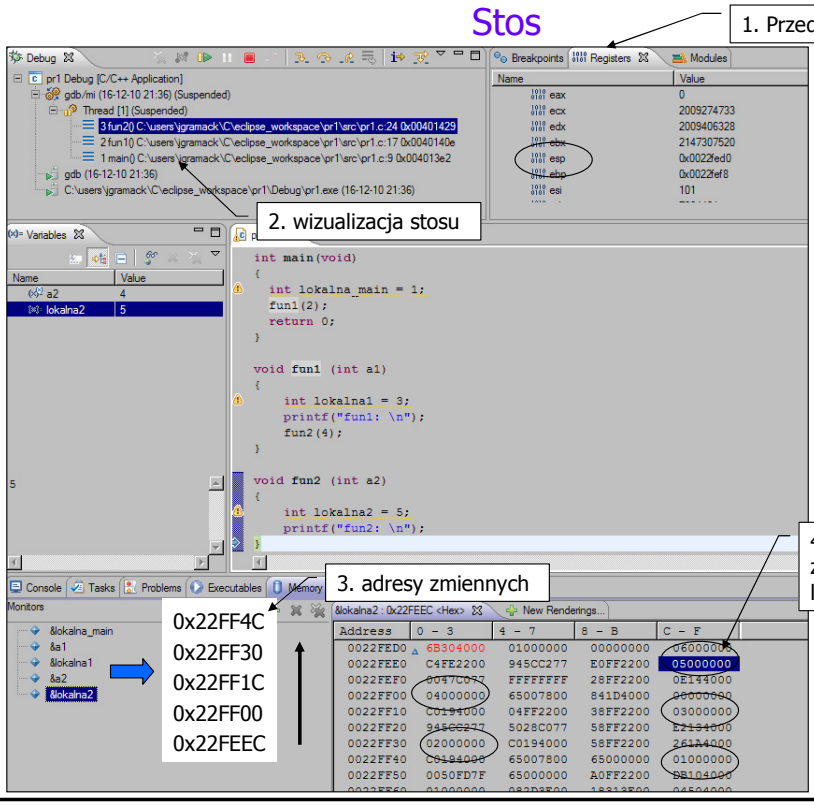
```
#include <stdio.h>

void fun1 (int);
void fun2 (int);

int main(void)
{
  int lokalna_main = 1;
  fun1(2);
  return 0;
}

void fun1 (int a1)
{
  int lokalna1 = 3;
  printf("fun1: \n");
  fun2(4);
}

void fun2 (int a2)
{
  int lokalna2 = 5;
  printf("fun2: \n");
}
```

Elementy **pogrubione** będą odkładane na stos w czasie działania programu

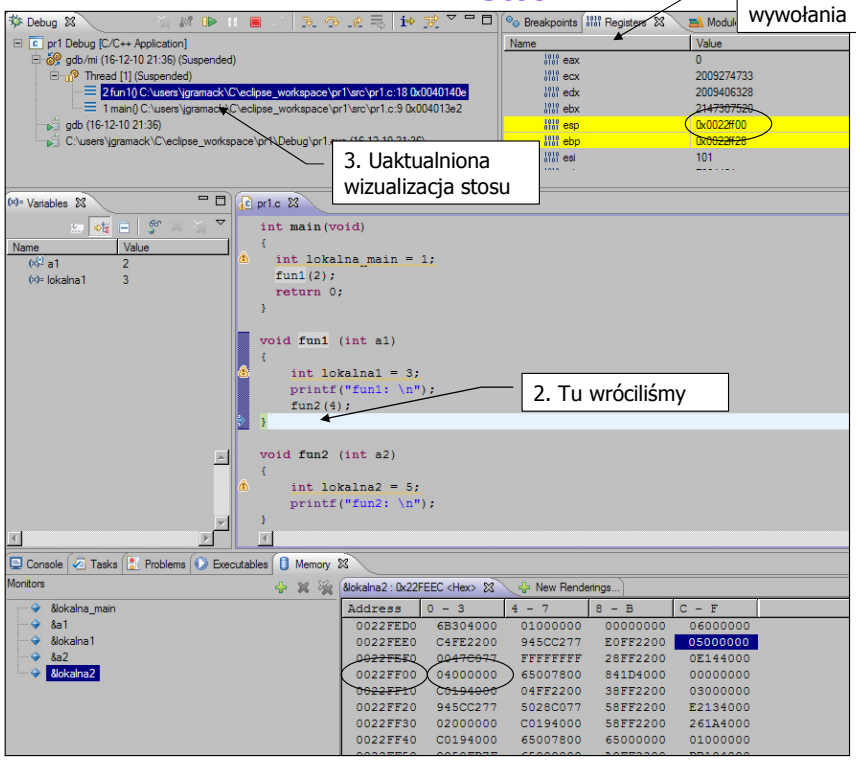
Stos


1. Początek sesji debuggera
2. esp – stack pointer
3. adres funkcji main
4. wizualizacja stosu

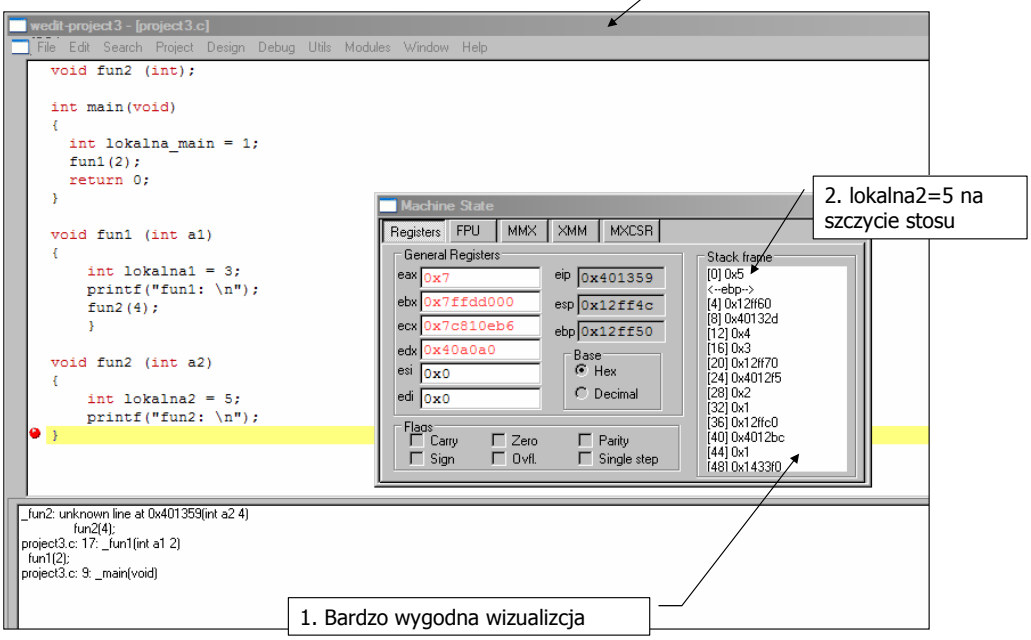
Stos


1. Przed powrotem z fun2()
2. wizualizacja stosu
3. adresy zmiennych
0x22FF4C
0x22FF30
0x22FF1C
0x22FF00
0x22FEEC
4. najwyżej na stosie znajduje się zmienna lokalna2

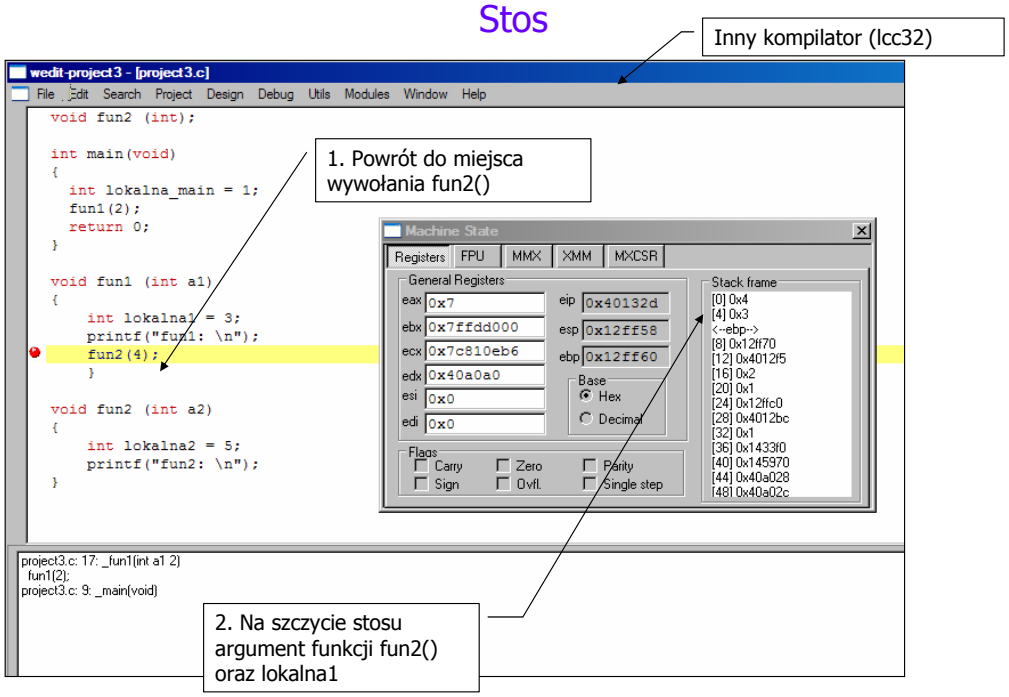
Stos


1. Powrót z fun2() do miejsca wywołania
3. Uaktualniona wizualizacja stosu
2. Tu wróciliśmy
int main(void)
{
int lokalna_main = 1;
fun1(2);
return 0;
}
void fun1 (int a1)
{
int lokalna1 = 3;
printf("fun1: \n");
fun2(4);
}
void fun2 (int a2)
{
int lokalna2 = 5;
printf("fun2: \n");
}

Stos


Inny kompilator (lcc32)
2. lokalna2=5 na szczycie stosu
1. Bardzo wygodna wizualizcja
void fun2 (int);
int main(void)
{
int lokalna_main = 1;
fun1(2);
return 0;
}
void fun1 (int a1)
{
int lokalna1 = 3;
printf("fun1: \n");
fun2(4);
}
void fun2 (int a2)
{
int lokalna2 = 5;
printf("fun2: \n");
}
Machine State
Registers
FPU
MMX
XMM
MXCSR
General Registers
Stack frame
_fun2: unknown line at 0x401359(int a2 4)
fun2(4);
project3.c: 17: _fun1(int a1 2)
fun1(2);
project3.c: 9: _main(void)

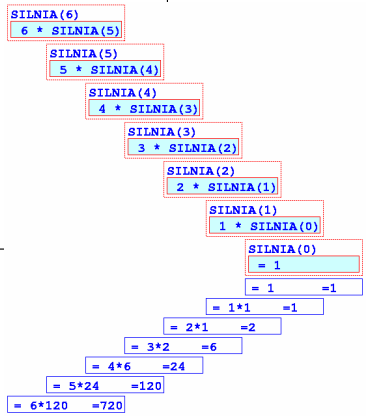
Stos



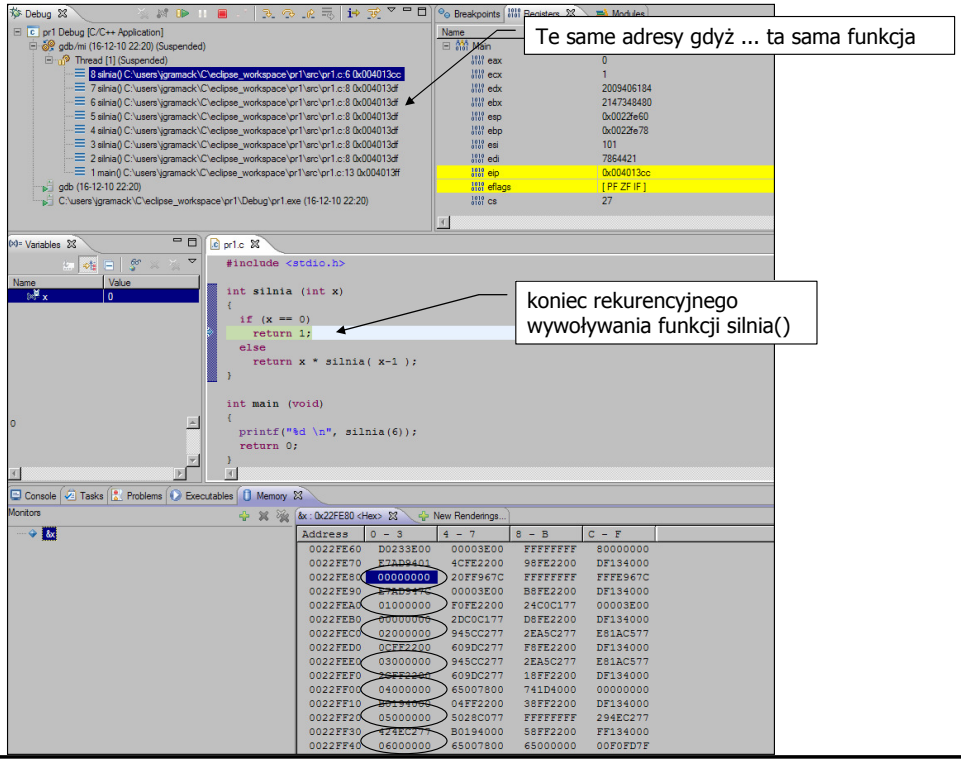
Stos i rekurencja

– Ilustracja **stosu.** Przykład rekurencyjny

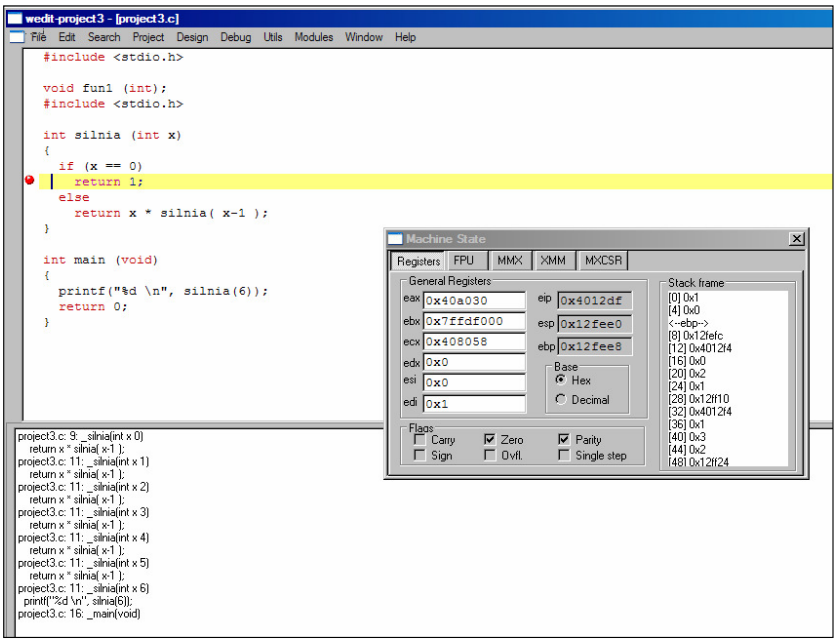
```
#include <stdio.h>

int silnia (int x)
{
  if (x == 0)
    return 1;
  else
    return x * silnia( x-1 );
}

int main (void)
{
  printf("%d \n", silnia(6));
  return 0;
}
```



Stos i rekurencja



Stos i sterta, rekurencja



Stos i rekurencja

```
Recursion

   1. A recursive function is a function that calls itself.
   2. The speed of a recursive program is slower because of stack overheads.
   3. In recursive function we need to specify recursive conditions,
      terminating conditions, and recursive expressions.

#include <stdio.h>

int add(int k, int m);

int main(void)
{
    int i;

    i = add(2, 7);
    printf("The value of addition is %d\n", i);
}

int add(int pk, int pm)
{
    if(pm == 0)
       return(pk);
    else
       return(1 + add(pk, pm-1));
}
```

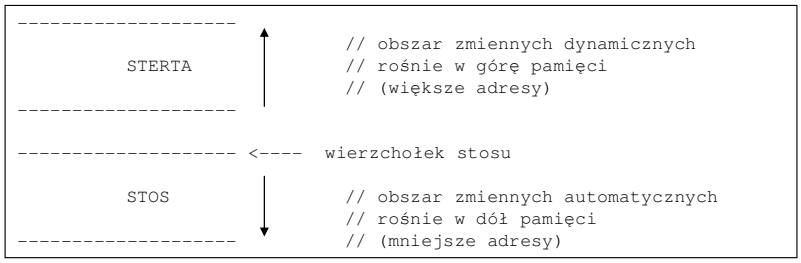
```
żeby zrozumieć rekurencje, trzeba najpierw zrozumieć rekurencję.
```


Rekurencja

Stos i sterta

- Stałe, zmienne statyczne i zewnętrzne mają zarezerwowane miejsce w kodzie wykonywalnym programu (są umieszczane w obszarze danych programu).
- Zmienne lokalne (automatyczne) funkcji są umieszczane na **stosie** w momencie, gdy sterowanie wejdzie do bloku, w którym zostały zdefiniowane. Zmienne tej klasy znikają po wyjściu sterowania z bloku.
- Zmienne dynamiczne są umieszczane na **stercie**.

Dzięki stosowi możliwy jest mechanizm wywołania rekurencyjnego funkcji !



- Zmienna dynamiczna może być utworzona wewnątrz funkcji. Przydzielony obszar pamięci nie znika po zakończeniu funkcji. Należy jednak zadbać o to, aby funkcja przekazała na zewnątrz wskaźnik do tak utworzonego obszaru.

Analiza "mapy pamięci"

```
#define N 4
#include <stdio.h>
#include <stdlib.h>                // free(), malloc()

int* fun1(int [], int []);
int main(void) {
  int i, tab1[N], tab2[N];
  int *tabwynikowa1 = NULL;

  for(i=0; i<N; i++) {
    tab1[i] = i;  tab2[i] = i + 10; }
  tabwynikowa1 = fun1(tab1, tab2);
  free(tabwynikowa1); // zwolnienie zaalokowanej w funkcji fun1() pamięci
  return 0;
}

int* fun1(int t1[], int t2[])
{
  int i;
  int *wynik = NULL;
  wynik = (int*) malloc (sizeof(int) * N);
  // free() będzie trzeba zrobić w main()
  for(i=0; i<N; i++)
    wynik[i] = t1[i] + t2[i];

  return wynik;
}
```

tab1:	0	1	2	3
tab2:	10	11	12	13
tabwynikowa:	10	12	14	16
	a	c	e	10

Gdyby próbować "zwrócić" tablicę lokalną `int wynik[N];`:
[Warning] address of local variable 'wynik' returned

Analiza "mapy pamięci"

przykładowa sesja z gdb

```
--> wewnątrz funkcji main()
--> przed instrukcją: tabwynikowa1 = fun1(tab1, tab2);
(gdb) p tab1                    // p - skrót debugera dla polecenia "print"
$1 = {0, 1, 2, 3}

(gdb) p &tab1
$2 = (int (*)[4]) 0xbffffb1c  // int (*)[4] gdyż zadeklarowano "stałą wskaźnikową" tab1

(gdb) p &tab1[0]
$2 = (int *) 0xbffffb1c       // teraz (int *)

(gdb) p &tab1 + 1
$2 = (int *) 0xbffffb2c       // przeskok o 4 elementy typu int (2c - 1c = f)

(gdb) p *tab1
$3 = 0

(gdb) p *(tab1+1)          20 - 1c
$4 = 1

(gdb) p tab1+1
$5 = (int *) 0xbffffb20                 // następny element tablicy: 20 - 1c = 4

(gdb) p tab1@4                          // porównaj typ zmiennej tab1 w $2
$6 = {{0, 1, 2, 3}, {-1073743000, 1073942863, 1, -1073742956}, {-1073742948,
    134514752, 0, -1073743000}, {1073942817, 1073819884, 1, 134513536}}

(gdb) p tabwynikowa1
$7 = (int *) 0x0                        // patrz inicjalizacja NULL-em

(gdb) p &tabwynikowa1                   // "adres adresu"
$8 = (int **) 0xbffffb08
```


Analiza "mapy pamięci"

przykładowa sesja z gdb

```
--> wewnątrz funkcji fun1()
--> int* fun1(int t1[], int t2[])
(gdb) s                                 // s - "step", weszliśmy do funkcji
fun2 (t1=0xbffffb1c, t2=0xbffffb0c) at tab-fun.c:34

(gdb) p t1
$9 = (int *) 0xbffffb1c                 // "kopia" wskaźnika do przekazywanego argumentu
                                        // Zauważ, że t1 jest typu int* a tab1 jest
                                        // typu int (*)[4]

(gdb) p &t1
$10 = (int **) 0xbffffad4               // tu w pamięci jest ta "kopia"

(gdb) p t1+2
$11 = (int *) 0xbffffb24

(gdb) p t1@4                            // porównaj wynik z $6
$12 = {0xbffffb1c, 0xbffffb0c, 0xbffffb08, 0x80498dc}

(gdb) p *t1
$13 = 0

(gdb) p *&*&*&*t1                       // ależ tu „pośredników” ! Niby poprawne, ale
...
$14 = 0

(gdb) p *t1@4
$15 = {0, 1, 2, 3}

(gdb) p wynik                           // po alokacji pamięci
$16 = (int *) 0x8049a10
```

Analiza "mapy pamięci"

```
--> z powrotem w funkcji main()
--> tabwynikowa1 = fun1(tab1, tab2);

(gdb) p tabwynikowa1
$17 = (int *) 0x8049a10            // porównaj wartość z $16
```