Tablicy w argumentach funkcji

- **Tablicy nie są przekazywane przez wartość.**
- **Tablicy są przekazywane przez referencje lub przez wskaźnik.**

Przykład:

```
char str[] = “abcdef”;
.....................
fun(str);      // argument faktyczny – to id tablicy – adres pierwszego bajtu
fun_1(str);  // argument faktyczny – to id tablicy – adres pierwszego bajtu
.....................

void fun(char my_str[])
/*   przekazanie argumentu przez referencje  */
{
    char symb;
    int pos = 2;
   ..............
    symb = my_str[pos];  // symb = ‘c’
   ……….
}
```

```
char str[] = “abcdef”;
.....................
fun(str);       // argument faktyczny – to id tablicy – adres pierwszego bajtu
fun_1(str);   // argument faktyczny – to id tablicy – adres pierwszego bajtu
.....................

void fun_1(char *ptr_str)
/*   przekazanie argumentu przez wskaźnik  */
{
    char symb;
    int pos = 2;
   ..............
    symb = *(ptr_str+pos);  // symb = ‘c’
   ……….
}
```

```
char str[] = “abcdef”;
.....................
fun(&str[3]);
fun_1(str+3);
.....................

void fun(char my_str[])
/*   przekazanie argumentu przez referencje  */
{
   char symb = my_str[0];  // symb = ‘d’, my_str = „def”
   ……….
}

void fun_1(char *ptr_str)
/*   przekazanie argumentu po wskaźniku  */
{
   char symb = *(ptr_str);  // symb = ‘d’, ptr_str = „def”
  ……….
}
```

Przykład W6_1

Tablice znakowe o dwóch indeksach

Definicja:
typ id_tablicy[*liczba_stała_1*][*liczba_stała_2*]

id_tablicy – to jest wskaźnik typu typ.

Przykład:

```
#define MAX_DIM_1 128
#define MAX_DIM_ 2 824
 int main()
{
   char str[5][128];
   double A[MAX_DIM_1][MAX_DIM_2];
  ……..……
 }
```

Pierwsza definicja rezerwuje w pamięci 5x128 słów typu char (5x128x1 b), druga – 128 x 824 słów typu double (128x824x8 b)

Dostęp do elementów tablicy:

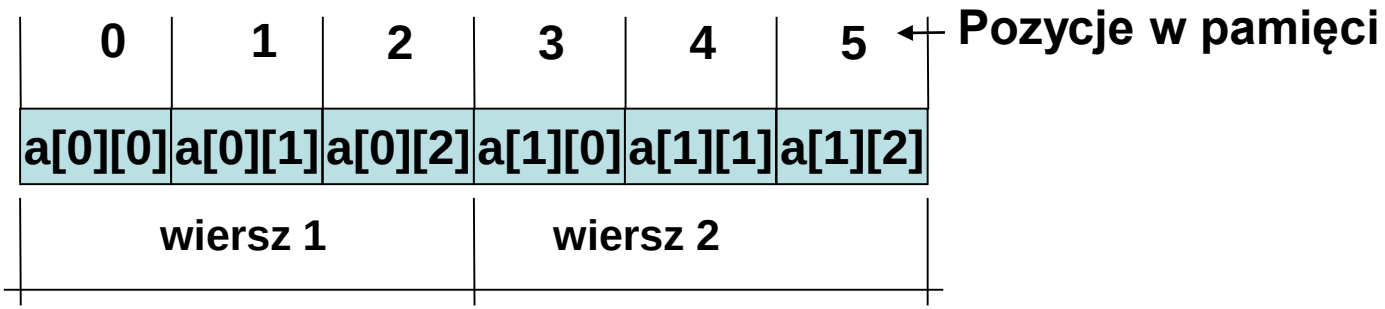
```
char str[3][4];
char symb = 'a';
int i, j;

for(i=0; i<3; i++)
{
      for(j=0; j<4; j++, symb++)
      {
            str[i][j] = symb;
      }
}
```

str[ ][ ]	j = 0	j = 1	j = 2	j = 3
i = 0	a	b	c	d
i = 1	e	f	g	h
i = 2	i	j	k	l

Dostęp do elementów tablicy:

Elementy tablicy dwuwymiarowej są zarezerwowane w pamięci ciągłe, wiersz po wierszu:

int a[2][3];



Kolejność rozmieszenia elementów tablicy dwuwymiarowej w pamięci

0 a[0][0]	1 a[0][1]	2 a[0][2]
3 a[1][0]	4 a[1][1]	5 a[1][2]

Przykład W6_2

Inicjalizacja tablicy dwuwymiarowej

```
int ii[2][3] = {0, 1, 2,
                3, 4, 5};

char str[3][4] = { ‘a’, ‘b’, ‘c’, ‘d’,
                   ‘e’, ‘f’, ‘g’, ‘k’ };

char str[3][4] = { 97,  98,   99, 100,
                   101, 102, 103, 104 };
```

Wykł. W4 – pliki.

Funkcja fgets:

```
char *fgets( char *str, int n, FILE *stream );
```

char *str – wskaźnik na ciąg znaków (bufor)
int n – maksymalna ilość symboli w buforze str
FILE *stream – wskaźnik do struktury typu FILE (identyfikuje plik, z którego pobieramy dane.

Zwraca: wskaźnik na bufor str przy zakończeni normalnym;
NULL – przy czytaniu pliku spotkaliśmy EOF lub przy czytaniu wystąpił błąd.

Czyta znaki z bieżącej pozycji pliku do pierwszego spotkania symbolu '\n' (razem z symbolem '\n‘) lub do końca pliku lub dopóki nie przeczyta n-1 znaków (koniec bufora) - co się pierwsze zdarzy. Wynik jest zapisany w str i jest dopisany symbol '\0'. Znak '\n' , jeśli był przeczytany, jest dołączany do wiersza tekstowego.

Przykład:

```
#include <stdio.h>

int main( void )
{
   FILE *stream;
   char line[100];

   if((stream = fopen("crt_fgets.txt", "r" )) != 0 )
   {
      if( fgets( line,  sizeof(line),  stream ) == NULL)
         printf( "fgets error\n" );
      else
         printf( "%s", line);
      fclose( stream );
   }

   return 0;

 }
```

Przykład używania tablicy dwuwymiarowej - W6_3

W przykładzie poprzednim:

char str[NO_LINES][NO_SYMB_IN_LINE];

nie zależnie od rozmiaru problemu alokujemy w pamięci NO_LINES*NO_SYMB_IN_LINE*sizeof(char) bajtów. Przy tym faktycznie używamy dla wierszu o indeksie i – (faktyczny_ilość_znaków_w_wiersze+1)*sizeof(char) bajtów. (funkcja fgets(...) dopisuje symbole specjalne ‘\n’ i ‘\0’ po każdym wywołaniu (W6_3 – debuger).

Uwaga! Statyczna alokacja pamięci często jest bardzo nieskuteczna.

1. Dla każdego wiersza jest zarezerwowano NO_SYMB_IN_LINE symboli typu char, a maksymalna ilość symboli w wierszu dla tego przykładu – 9. Więc pamięć, która jest potrzebna dla alokacji najdłuższego wiersza – (9+2)*sizeof(char) bajtów.

2. Zarezerwowano pamięć dla NO_LINES wierszy, a faktycznie użyto – 3 wierszy.

Problem polega na tym, że przy statycznej alokacji z góry nie wiadomo, ile pamięci będzie potrzebne dla podanego problemu. I dla tego wychodzi, że dla dużych zagadnień zarezerwowanej pamięci będzie za mało, a dla małych – za dużo.

Dynamiczna alokacja pamięci

Pamięć pozostaje wydzieloną i zwolnioną w trakcie wykonania kodu programy. Nadaje możliwość skutecznie używać resursy pamięci.

#include <malloc.h>

void *malloc(size_t *size*);

Alokuje w pamięci blok o rozmiarze size_t *size* bajtów. Zwraca wskaźnik do początku tego bloku lub NULL przy błędnym zakończeniu (nie wystarczyło pamięci).

void *calloc(size_t *num*, size_t *size*);

Alokuje w pamięci tablicą o *num* elementów, każdy z których ma rozmiar *size* bajtów. Wykonuje się inicjowanie elementów tablicy — każdy element będzie inicjowany jako zero. Zwraca wskaźnik do początku tablicy lub NULL przy błędnym zakończeniu (na przykład, zabrakło pamięci).

Dynamiczna alokacja pamięci

void *realloc(void **memblock*, size_t *size*);

Zmienia rozmiar bloku, zaalokowanego w pamięci. Argument *memblock* jest wskaźnikiem do początku bloku. Jeśli *memblock* = *NULL,* realloc alokuje blok pamięci o rozmiarze *size*. Jeśli *memblock* nie NULL, to musi być wskaźnikiem, zwróconym przez poprzednie wywołanie *malloc, calloc, realloc*.

Zwraca wskaźnik typu void na ponownie alokowany blok, przy czym blok może być przesunięty w pamięci po realokacji.

Dynamiczna alokacja pamięci

void *realloc(void **memblock*, size_t *size*);

Gdy pamięci nie wystarcza dla rozszerzenia bloku do potrzebnego rozmiaru, pierwotny blok pozostaje bez zmian, a realloc zwraca NULL.

Gdy *size = 0,* realloc zwalnia pamięć, przy czym wskaźnik *memblock* nie zmienia swojej wartości.

Dynamiczna alokacja pamięci

void free(void *_memblock_);

Zwalnia blok pamięci *memblok*, który był wydzielony *malloc, calloc, realloc*.

Jeśli wskaźnik nie jest zainicjowany (!= 0x0000000) albo przetaszczeń adresowa procesu pozostała uszkodzona, albo wskaźnik był zmieniony tak, że nie wskazuje do początku bloku, wywołanie free(...) powoduje page falt.

size_t _msize(void * _memblock_);

Zwraca rozmiar bloku pamięci *memblock* , alokowanego dynamicznie. Wartość wskaźnika *memblock* powinna być dokładnie taką, jaką zwrócili funkcji *malloc*, *calloc*, *realloc*.
Jeśli w funkcje *_msize* podstawić wartość wskaźnika do bufora, alokowanego statycznie, dostaniemy page fault.

Dynamiczna alokacja pamięci

Przykład

```
double  a[100], *b = NULL;
size_t buffsize;

if((b = (double *)malloc(100*sizeof(double))) == NULL)
{
     //błąd przy alokowaniu pamięci
}
……..
buffsize = sizeof(a);   //800 – rozmiar tablicy w B, alokowanej statycznie
buffsize = sizeof(b);   //4 – rozmiar słowa (wskaźnika) aplikacja ia32
                        //8 – rozmiar słowa (wskaźnika) aplikacja x64

buffsize = _msize((void *)a);  // page fault: a nie jest alokowane
                               // dynamicznie
buffsize = _msize((void *)b);  // 800 – rozmiar tablicy w B, alokowanej
                                //dynamicznie
……
```

Dynamiczna alokacja pamięci

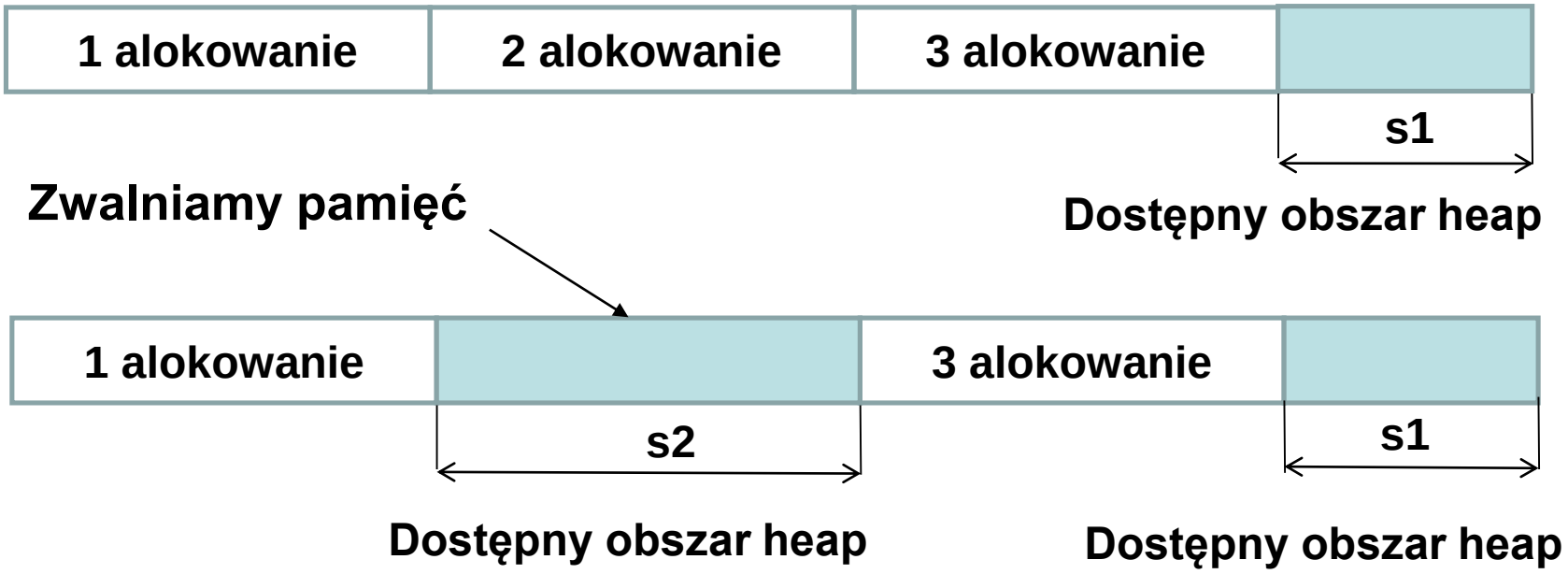
```
#define MAX_SYMB 256

int main( void )
{
   char *string = (char *)malloc(MAX_SYMB*sizeof(char));

   if( string == NULL )
            printf( „brak pamięci\n" );
   else
   {
            printf( “pamięć pozostała wydzielona dla bloku string\n" );
            free( string );
            printf( “pamięć zwolniona\n" );  string = NULL;
    }
 }
```

Dynamiczna alokacja pamięci

```
// This program uses calloc to allocate space for
// 40 long integers. It initializes each element to zero.

#include <stdio.h>
#include <malloc.h>

int main( void )
{
   long *buffer;
   buffer = (long *)calloc( 40, sizeof( long ) );

   if( buffer != NULL )
      printf( "Allocated 40 long integers\n" );
   else
      printf( "Can't allocate memory\n" );
   ..........................................................................
   free( buffer );     buffer = NULL;
}
```

Dynamiczna alokacja pamięci

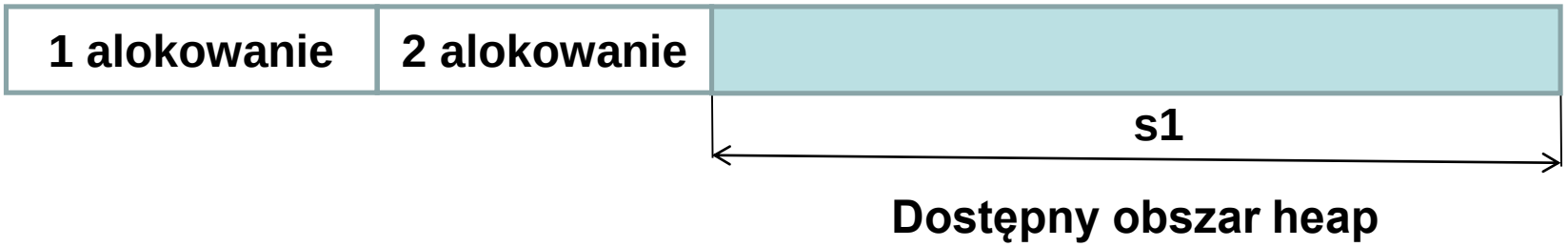
Fragmentowanie heap



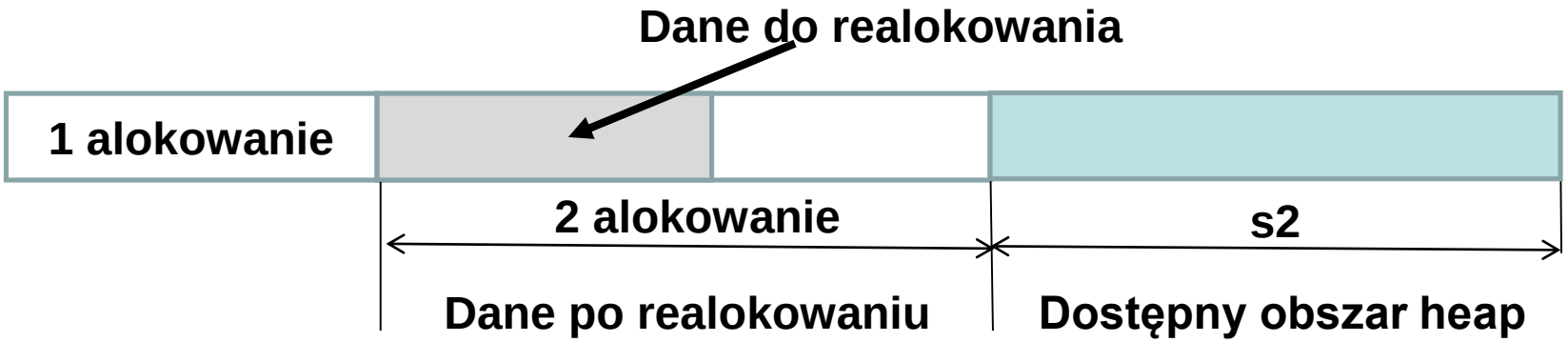
Rozmiar największego bufora, który można zaalokować dynamicznie, wynosi max{s1, s2}, a nie s1 + s2.

Dynamiczna alokacja pamięci

Fragmentowanie heap



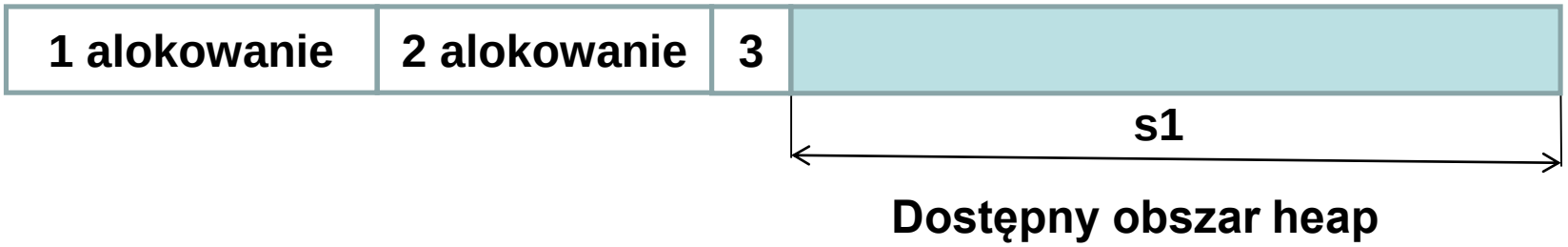
Realokujemy bufor 2 o większy rozmiar



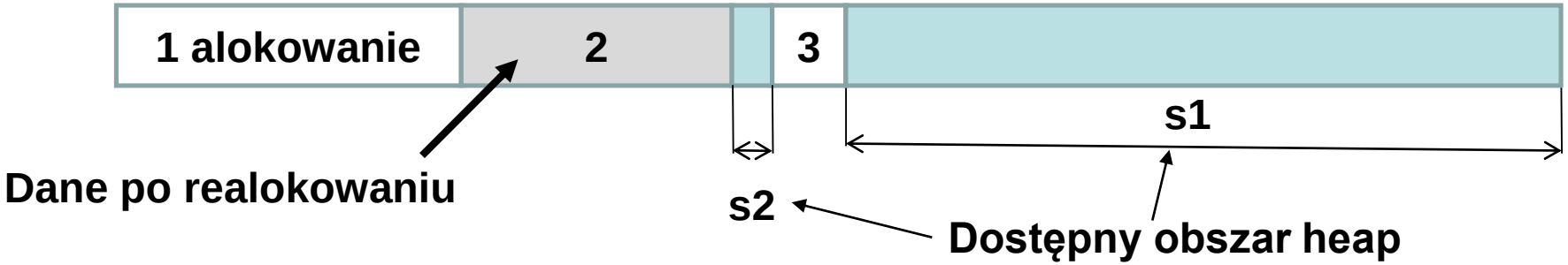
OK, pamięć pozostała nie fragmentowana

Dynamiczna alokacja pamięci

Fragmentowanie heap



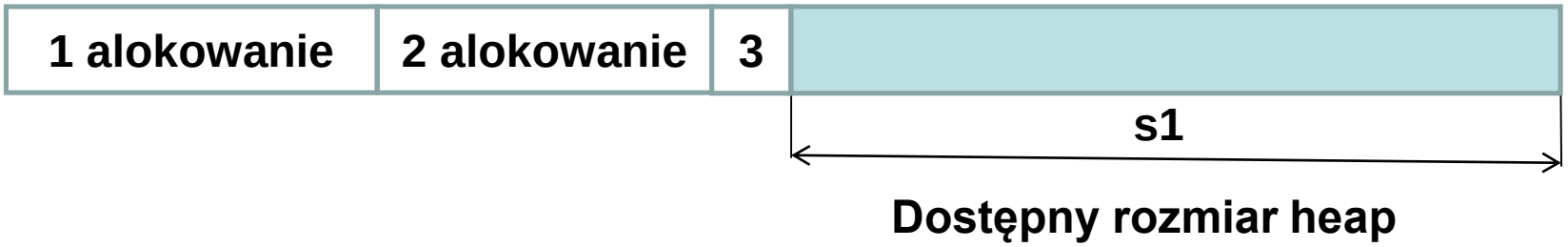
Realokujemy bufor 2 o mniejszy rozmiar



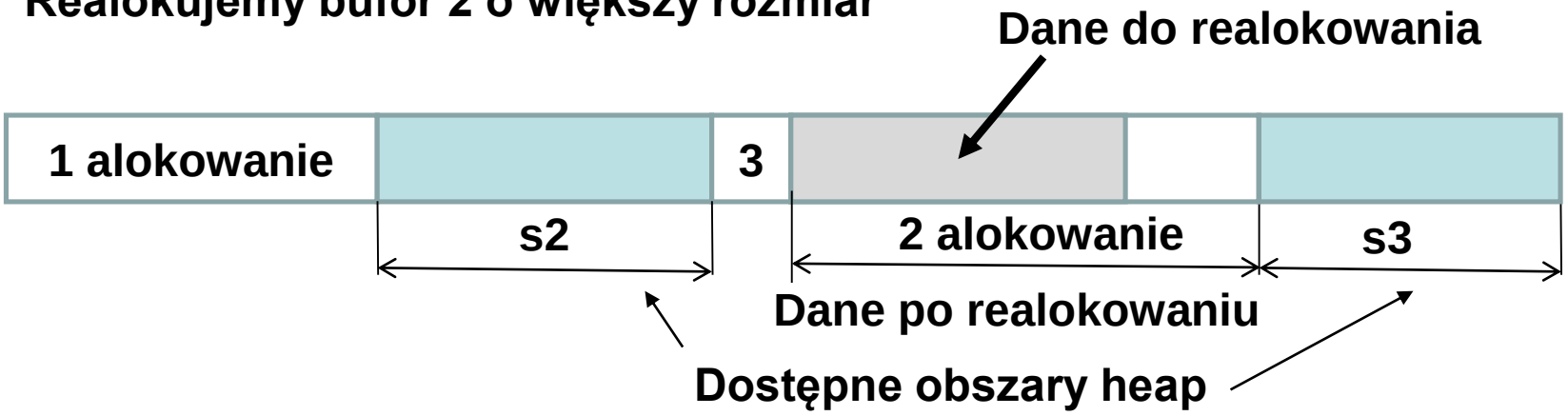
Pamięć fragmentowana, jednak rozmiar dostępnej części heap pozostał taki samy

Dynamiczna alokacja pamięci

Fragmentowanie heap



Realokujemy bufor 2 o większy rozmiar



Pamięć pozostała fragmentowana. Rozmiar największego bufora, który można zaalokować dynamicznie, wynosi max{s2, s3}, a nie s2 + s3.

Defragmentowanie heap

heap

Zapisujemy bufory do pliku i zwalniamy pamięć heap

plik na dysku

(przykład W4_2)

Ilość rekordów deskryptor rekordów

Alokujemy pamięć dla każdego poszczególnego rekordu i odczytujemy dane z pliku

heap: brak fragmentowania

W C istnieje dwie możliwości reprezentacji tablic dwuwymiarowych:
pierwsza – to przez definicje tablicy dwuwymiarowej:

```
int a[10][34];  //alokacja statyczna
```

druga – to przez tablice wskaźników:

```
int *ptr_a[10]; //statyczne wydzielenie pamięci dla tablicy wskaźników
for(i=0; i<10; i++)
{
    //alokacja pamięci dla każdego wiersza – każdy wiersz może mieć swoją
    //dlugość

    ptr_a[i] = (int *)malloc(dlugosc_w_bajtach[i]);
    if(!ptr_a[i])
    {
       ..........................................................
     }
}
```

Zwolnienie pamięci:

```
for(i=0; i<10; i++)
{
    //Uwaga! Zwolnienie może być wykonane tylko jeden raz!
    if(ptr_a[i] != NULL)
    {
            free(ptr_a[i]);
            ptr_a[i] = NULL;
    }
}
```

Ta metoda ma fiksowaną ilość wierszy (tablicy wskaźników), przecież długość każdego wiersza może być dowolną.

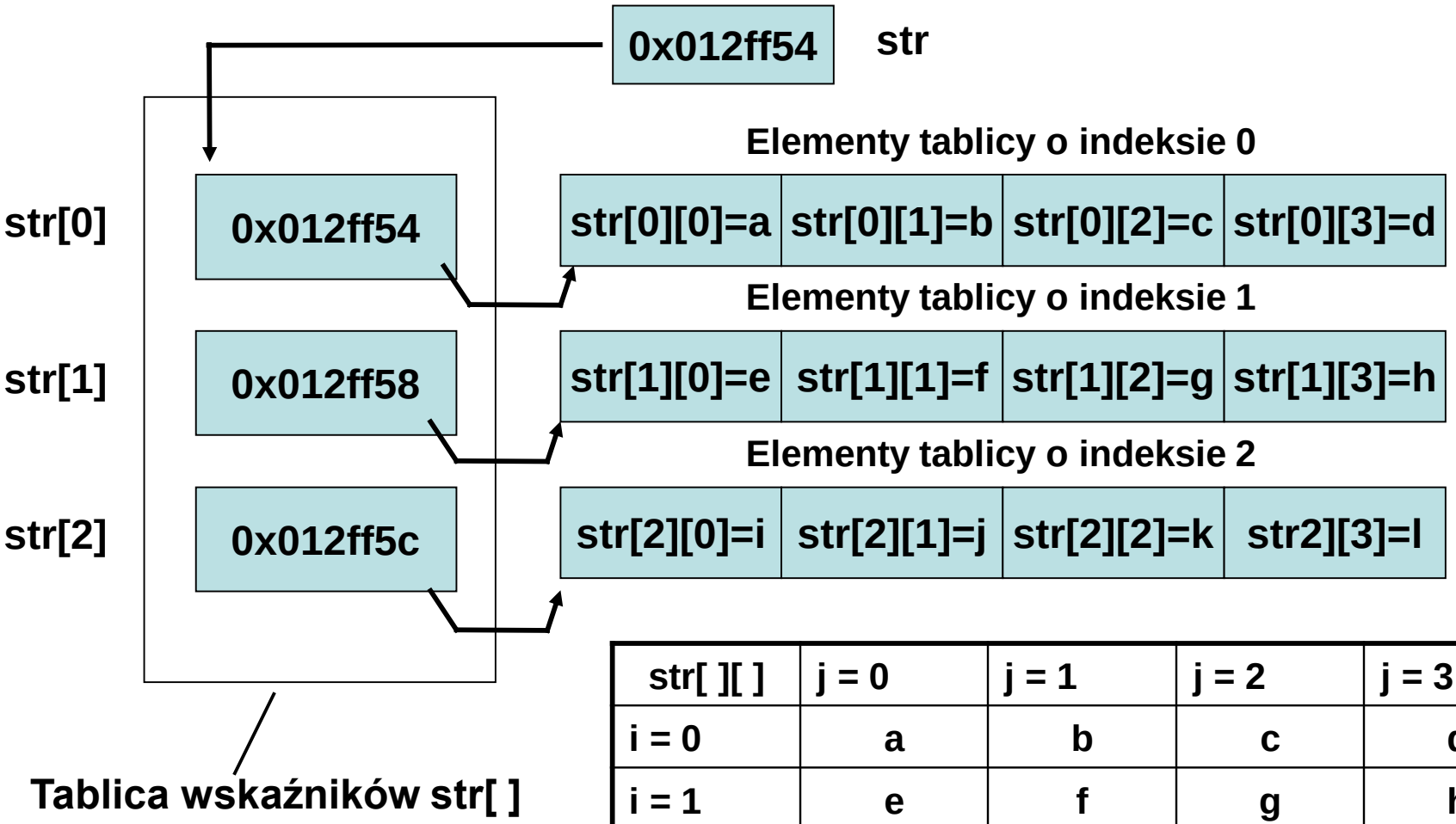
Wskaźnik do tablicy wskaźników (wskaźnik do wskaźnika) – może być dowolna ilość wierszy i dowolna długość każdego wiersza.

```
char **str = NULL;
//alokujemy pamięć dla tablicy wskaźników:
str = (char **)malloc(ilość_wierszy*sizeof(char *));
if(!str)
{.....//obsługa błędnej sytuacji
}

memset((void *)str, 0, ilość_wierszy*sizeof(char *));

//alokujemy pamięć dla elementów każdego wiersza
for(i=0; i<ilość_wierszy; i++)
{
        str[i] = (char *)malloc(długość_wiersza_i*sizeof(char));
        if(!str[i])
        {
         .....//obsługa błędnej sytuacji
         }
}
```

Zwolnienie pamięci (wskaźnik do wskaźnika):

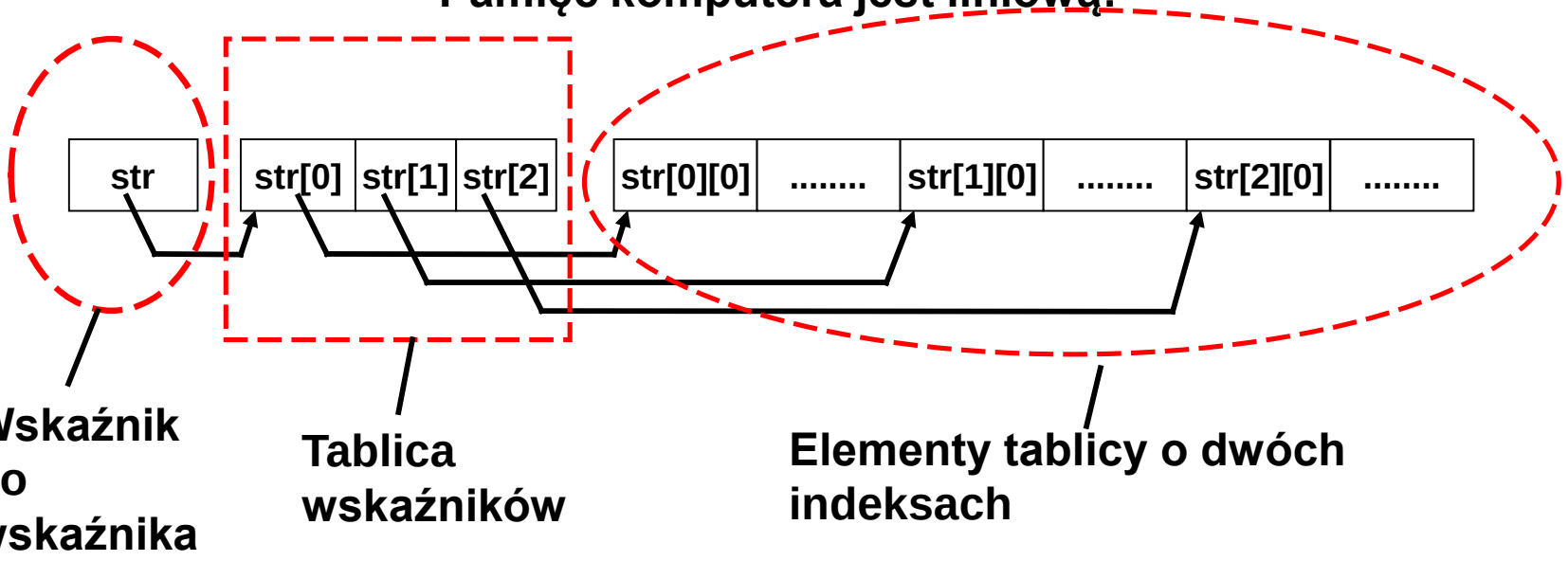
```
//sprawdzamy, czy str jeszcze nie jest zwolnione
if(str)
{
    for(i=0; i<ilość_wierszy; i++)
    {
            // sprawdzamy, czy str[i]  jeszcze nie zwolnione
            if(str[i])
            {
                free(str[i]);     //zwalniamy str[i]
                str[i] = NULL; //zapobiegamy zwolnieniu wielokrotnemu
            }
     }

     free(str);      //zwalniamy str – wskaźnik na tablice wskaźników
     str = NULL; // zapobiegamy zwolnieniu wielokrotnemu
}
```

str – to jest „wskaźnik do wskaźnika” – wskazuje do tablicy wskaźników

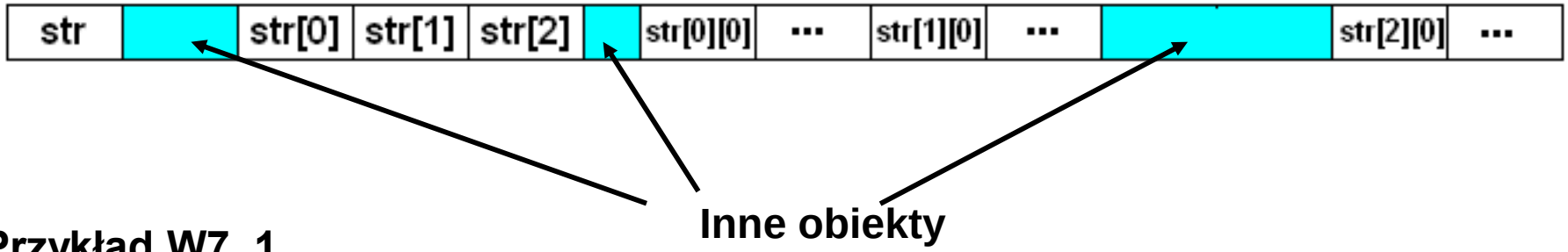


str[ ][ ]	j = 0	j = 1	j = 2	j = 3
i = 0	a	b	c	d
i = 1	e	f	g	h
i = 2	i	j	k	l

Pamięć komputera jest liniowa:



Przy dynamicznej alokacji pamięć pomiędzy wierszami może być nie ciągłą



Przykład W7_1
Przykład W7_2 – czytanie z pliku.
Przykład Lab_5 – Wiersze tablicy. (st. zaoczne – samodzielnie)

Funkcje obsługi tekstów: (#include <stdlib.h>)

char *_itoa(int value, char *str, int radix);

Przetwarza cyfry liczby całkowitej *value* w ciąg znaków (do 33 symbolów), zakończony '\0', i zapisuje w *str* ; *radix* = [2,36] – podstawa potęgi (*radix* = 10 – liczbę dziesiętne, *radix* = 2 – liczbę binarne). Zwraca wskaźnik na *str*. Nie identyfikuje zakończenia błędnego.

int atoi(const char *str);

Przetwarza ciąg znaków *str* (zakończony '\0'), które mogą być potraktowane jako cyfry, w liczbę typu integer. Czytanie *str* będzie zakończone przy pierwszym spotkaniu znaku, który nie może być potraktowany jako cyfra. Zwraca liczbę typu int – wynik przetwarzania.

Analogicznie funkcje (st zaocne – samodzielnie)

char *_ecvt(double value, int count, int *dec, int *sign);
char *_fcvt(double value, int count, int *dec, int *sign);

Przetwarzają liczbę zmiennoprzecinkowe *value* w ciąg znaków.

```
char *_ecvt( double value, int count, int *dec,
int *sign );
```

Parameters

value	Number to be converted.
count	Number of digits stored (in mantissa).
dec	Stored decimal-point position.
sign	Sign of the converted number.

Return Value

_ecvt returns a pointer to the string of digits; NULL if an error occurred.

Przykład:

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

int main( void )
{
   int     decimal,   sign;
   char    *buffer;
   int     precision = 10;
   double  source = 3.1415926535;

   buffer = _ecvt( source, precision, &decimal, &sign ); // C4996
   printf( "source: %2.10f   buffer: '%s'  decimal: %d  sign: %d\n",
           source, buffer, decimal, sign );
}
```

Output:

source: 3.1415926535

buffer: '3141592654'

decimal: 1 sign: 0

Przetwarza ciąg znaków w liczbę typu double. Szczegóły są podane w MSDN.

double atof(const char *str);

Przykład W7_2