Podstawy programowania.

Wykład 8

Wskaźniki

Adresy zmiennych

- Język C pozwala na operowanie adresami w pamięci
  - stąd, między innymi, kwalifikowanie C jako języka relatywnie niskiego poziomu, bliskiego asemblerowi, językowi maszynowemu
- Operator adresu & zwraca adres początku obszaru pamięci powiązanego ze zmienną
  - adres zmiennej typu T jest typu "wskaźnik do T"
  - adres jest liczbą o reprezentacji zależnej od aktualnego środowiska wykonania programu
    - standard C nie gwarantuje, że adres zmiennej będzie można reprezentować za pomocą zmiennej określonego typu arytmetycznego (np. różnych wariantów liczb całkowitych)
    - adresy w systemach 32 bitowych zazwyczaj można rzutować na zmienne typu unsigned integer (liczba całkowita bez znaku)
    - w systemach 64 bitowych powinno to być long unsigned integer
      - printf("zmienna a: %d, jej adres: %lu\n", a, &a);

Wskaźniki

- Język C pozwala na przechowywanie adresów w zmiennych
  - a także na wykonywanie operacji na zmiennych przechowujących adresy – o tym na zakończenie wykładów
- Deklaracje i definicje zmiennych przechowujących adresy
  - określają typ zmiennej jako wskaźnik
  - wymagają określenia do zmiennej jakiego typu jest to wskaźnik, np.

    ```
    int * wskaznik_do_int;
    double * double_p;
    ```

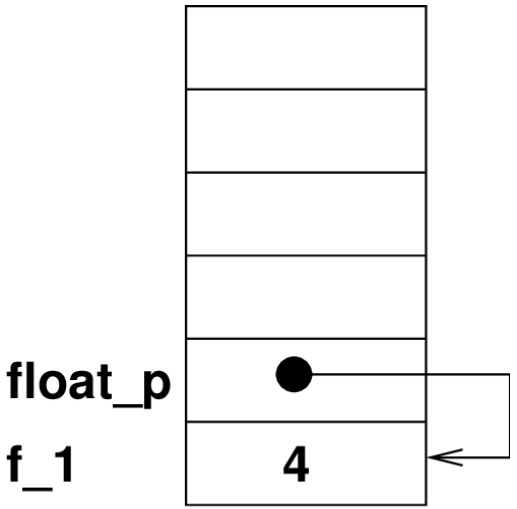
  - wyjątkiem jest wskaźnik typu

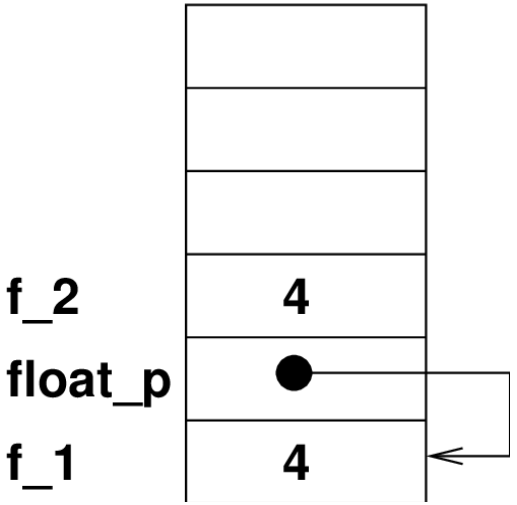
    ```
    void * czysty_wskaznik_do_pamieci;
    ```

    - wykorzystanie zmiennych zadeklarowanych jako void * wymaga najczęściej wcześniejszego rzutowania na wskaźnik do zmiennych konkretnego typu (nowsze standardy często pozwalają na pominięcie rzutowania)

      ```
      printf("%c", * ((char *) czysty_wskaznik_do_pamieci));
      ```

Wskaźniki





- Zmienna typu "wskaźnik do T" pozwala uzyskać dostęp do obszaru pamięci na który wskazuje
  - służy do tego operator wyłuskania (przekierowania, dereferencji)

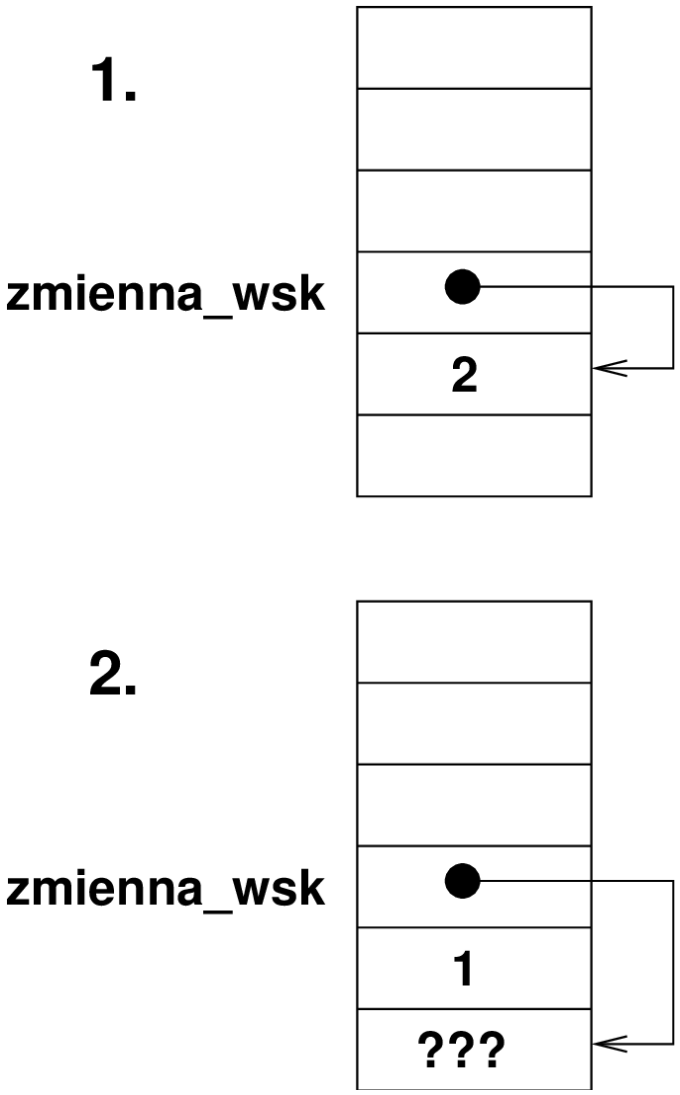
```
float f_1 = 4;
float* float_p = &f_1;
// f_1 musi być zmienną typu float
```

  - `float f_2 = *float_p;`

```
// podstaw do f_2 to na co
// wskazuje float_p
```

  - efekt powyższych operacji jest identyczny jak: `float f_2 = f_1;`

Wskaźniki



- Wynik operatora wyłuskania jest miejscem w pamięci, co oznacza, że wyrażenie postaci (* zmienna_wsk) jest l-wartością (może być argumentem lewostronnym przypisania):

  *zmienna_wsk = 1;

- Uwaga na notację:

  1. (*zmienna_wsk)++; // zmiana wartości
     // w komórce wskazywanej

  2. *zmienna_wsk++ // zmiana wartości
     // wskaźnika jako pierwsza - arytmetyka
     // wskaźników (może być źródłem błędów)

Wskaźniki

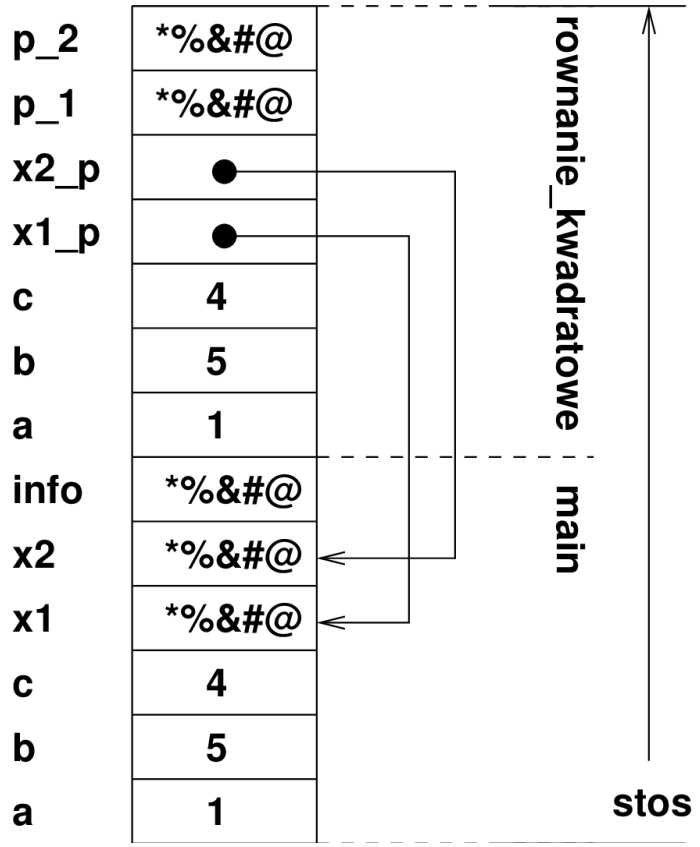
- Możliwość przypisania wskaźnikowi dowolnej wartości oznacza, że możemy sprawić, żeby wskazywał na dowolny obszar pamięci
  - może wskazywać np. na obszar kodu lub obszar stałych, co w przypadku modyfikacji wartości w komórkach, na które wskazuje prowadzi do trudnych do wykrycia błędów kodu
- Dlatego wskaźniki powinny zawsze być inicjowane, tak żeby wskazywały na komórki zawierające dane do modyfikacji
  - w przypadku braku takich lokalizacji w pamięci, w momencie definicji wskaźnika dobrze jest inicjować wartością NULL
    - NULL jest technicznie wartością 0
    - z założenia wskaźnik o wartości NULL nie może wskazywać na żaden obszar pamięci
  - inicjowanie wartością NULL gwarantuje błąd w przypadku każdej próby użycia tak zainicjowanego wskaźnika do wyłuskania

Wskaźniki

- Jedną z najważniejszych ról wskaźników w C jest umożliwienie przekazywania z funkcji większej liczby obliczonych wartości, niż jedna zwracana przez funkcję
- Odbywa się to przez przekazanie do funkcji argumentu będącego adresem
  - adres jest przekazany przez wartość i skopiowany do odpowiedniego argumentu formalnego funkcji
  - posługując się odpowiednim typem wskaźnikowym funkcja może modyfikować wartość komórki pamięci o adresie przekazanym jako argument
- Częstą praktyką jest
  - zwracanie przez funkcję kodu sukcesu lub niepowodzenia (błędu)
  - przekazywanie wyniku przetwarzania danych wejściowych poprzez wskaźniki do zmiennych

Wykorzystanie wskaźników

```
// funkcja oblicza pierwiastki równania kwadratowego
int  rownanie_kwadratowe( // funkcja zwraca kod sukcesu lub następujące kody...
       double a, double b, double c, // współczynniki równania kwadratowego
       double* x1_p, double* x2_p // wskaźniki do miejsc. gdzie mają zostać
     );                                  // umieszczone obliczone pierwiastki równania

void main(void )
{  double a=1, b=5, c=4, x1, x2;
   int info = rownanie_kwadratowe(a, b, c, &x1, &x2);
}

int  rownanie_kwadratowe( double a, double b, double c,
                                  double* x1_p, double* x2_p )
{  double p_1, p_2; // punkt testowania
  ...
 // obliczenie pierwiastków, obsługa sytuacji wyjątkowych
   *x1_p = p_1; *x2_p = p_2;
}
```



Wykorzystanie wskaźników

```
// funkcja zwiększa o 1 wartość zmiennej posługując się wskaźnikiem
void funkcja_1(
      float * float_1_p // wskaźnik do zmiennej – konwencja: [in/out]
     );                 // konwencja informuje, że wskaźnik jest użyty do
                        // przekazania efektu działania funkcji
void main(void )
{
   float f = 1.0;
   funkcja_1(&f);
}

// funkcja zwiększa o 1 wartość zmiennej posługując się wskaźnikiem
void funkcja_1(
      float * float_1_p // wskaźnik do zmiennej  – konwencja: [in/out]
){
   (*float_1_p)++;
}
```

Wskaźniki i tablice

- Druga istotna rola wskaźników w C wynika z częściowego utożsamienia tablic i wskaźników
  - nazwa tablicy jest wskaźnikiem do jej pierwszego elementu
    - a == &a[0]

```
int int_tab[10] = {1,2,3,4}; // inicjowanie tablic listą wartości
int* int_wsk = NULL; // inicjowanie wskaźników wartością zero
int* int_wsk = int_tab; //  lub adresem istniejącej tablicy
// nie można inicjować wskaźnika listą wartości
// dostęp do elementów – równoważność notacji indeksowej i wskaźnikowej:
// po podstawieniu int_wsk = int_tab;
int_tab[2] = 5; // równoważne: int_wsk[2] = 5;
*(int_tab+2) = 5;  // równoważne: *(int_wsk+2) = 5;
// dozwolone operacje:
int_wsk++; // operacja int_tab++; - niedozwolona
```

Wskaźniki i tablice

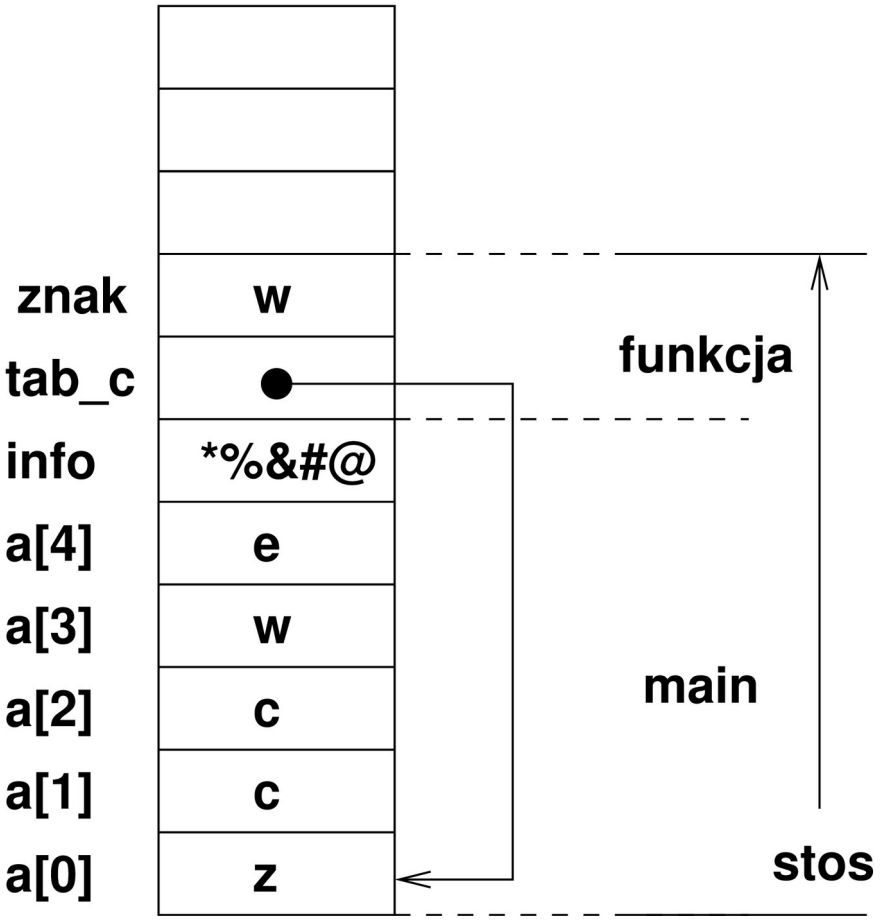
- W C nigdy nie przesyła się całych tablic jako argumentów funkcji
  - zawsze przesyłany jest, przez wartość (!), wskaźnik do tablicy, czyli adres początku tablicy
- Wewnątrz funkcji zmienna przesłana jako argument, niezależnie od tego czy w postaci np. int tab[] czy int* tab zachowuje się tak samo (czyli jak wskaźnik)
  - oznacza to, że wewnątrz wywoływanej funkcji zawsze można dokonywać modyfikacji elementów przesyłanej jako argument tablicy

```
void funkcja( int * a ){ // identyczne z void funkcja( int a[] ){
    a[0] = 5 ; a[1]++ ;     // dla każdego wskaźnika można
                            // stosować notację indeksową
// operacje poniżej są niestandardowe – dla lepszego zrozumienia jak działa C
    a+=3; // w tym miejscu dozwolone – a jest zmienną na stosie
    printf("%d\n", a[-3]);  // co zostanie wydrukowane? (dawne a[0] czyli 5)
}
```

Wskaźniki i tablice

- Schemat pamięci w momencie osiągnięcia punktu testowania

```
int funkcja( char tab_c[] );

void main( void )
{
  char a[5] = {'a','b','c','d','e'};
  int info = funkcja( a );
}

int funkcja( char tab_c[] ){
   char znak = 'w'
   tab_c[0] = 'z',;
   tab_c[1]++ ;
   tab_c[3] = znak;
   // punkt testowania
... }
```



Wskaźniki i tablice

- Specjalne dodatkowe własności dla tablic znaków

```
char tab_c[] = "Hello"; // inicjowanie znakami z napisu
// niedozwolone operacje: tab_c++
// dozwolone operacje (tak jak dla normalnych tablic):
tab_c[0] = ‘M’; tab_c[1]++;

char* wsk_c = "Hello"; // inicjowanie wskaźnika adresem stałego napisu
// niedozwolone operacje: wsk_c[0] = ‘M’; wsk_c[1]++;
// dozwolone operacje:
wsk_c++; // zawsze dozwolona operacja dla zmiennej wskaźnikowej
wsk_c = "Melon"; // podstawienie adresu innego stałego napisu
// nadal niedozwolone: wsk_c[0] = ‘B’; wsk_c[1]++;
wsk_c = tab_c; // podstawienie adresu istniejącej tablicy
// teraz dozwolone operacje:
wsk_c[0] = ‘B’; wsk_c[1]++; // itd.
```

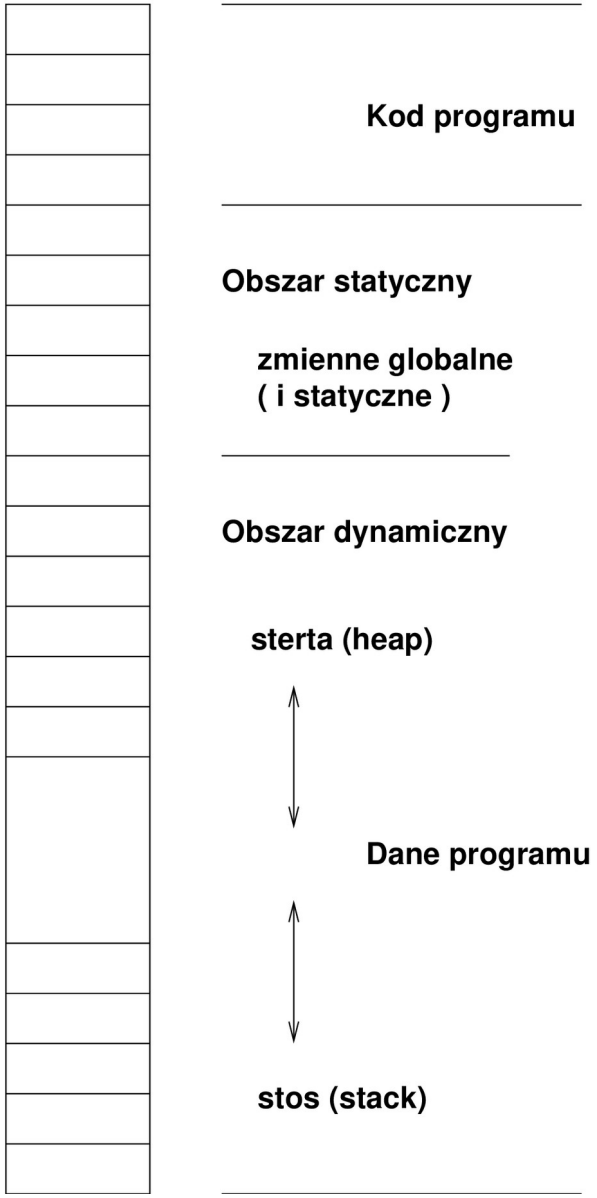
Wykorzystanie wskaźników

```
#define SCALAR int // float, double – algorytm identyczny

void sortowanie_babelkowe(SCALAR tablica[], int rozmiar) {
   int i, j;
   for (i = 0; i<rozmiar-1; i++){
      for (j=0; j<rozmiar-1-i; j++){
         if (tablica[j] > tablica[j+1]) przestaw(&tablica[j], &tablica[j+1]);
}  }  }                                  // alternatywa: przestaw(tablica, j, j+1);

przestaw(SCALAR *scalar_1_p, SCALAR *scalar_2_p);
{
SCALAR temp = *scalar_1_p;
*scalar_1_p = *scalar_2_p;
*scalar_2_p = temp;
}
```

Operator sizeof

- Standard C nie definiuje dokładnie rozmiarów zmiennych wszystkich typów wbudowanych
  - rozmiary mogą być uzależnione od sprzętu (typu procesorów), systemu operacyjnego, środowiska wykonania itp.
- C udostępnia operator sizeof umożliwiający zwrócenie rozmiaru (w bajtach) konkretnej zmiennej lub wszystkich zmiennych konkretnego typu:
  - int a;
  - size_t rozmiar_a = sizeof(a); // (a) jest wyrażeniem
  - size_t rozmiar_int = sizeof(int); // ta forma wymaga nawiasów
- Typ size_t jest zdefiniowany w pliku stddef.h
- Operator sizeof może zostać użyty do obliczenia liczby elementów tablicy:
  - int array_size = sizeof array / sizeof array[0];

Wskaźniki i zmienne dynamiczne



- Trzecia istotna rola wskaźników w C wynika z mechanizmów zarządzania pamięcią
  - zmienne definiowane standardowo mogą mieć dwa sposoby przechowywania
    - w obszarze statycznym – istniejąc przez cały czas wykonania programu
    - na stosie – jako zmienne automatyczne, istniejące przez czas wykonywania określonego fragmentu kodu
  - zmienna będąca wskaźnikiem może wskazywać na dane znajdujące się poza obszarem statycznym i poza stosem
    - taki obszar pamięci nazywany jest obszarem dynamicznym lub stertą
  - przydział obszaru pamięci na stercie jest dokonywany dynamicznie za pomocą specjalnych procedur zarządzania pamięcią

Zarządzanie pamięcią

- Standardowa biblioteka C dostarcza kilka procedur zarządzania pamięcią dynamiczną:
  - funkcje alokowania (przydzielania, rezerwowania) pamięci:
    - malloc: void *malloc(size_t size); // argument: rozmiar w bajtach
    - calloc: void *calloc(size_t nmemb, size_t size); // argumenty:
      // liczba obiektów, rozmiar pojedynczego obiektu
    - realloc: void *realloc(void *ptr, size_t size); // argumenty:
      // wskaźnik do istniejącego obiektu, nowy rozmiar pamięci
  - funkcje zwracają wskaźnik do zaalokowanego obszaru pamięci lub wskaźnik NULL w przypadku niepowodzenia
  - calloc inicjuje obszar zerami, realloc pozostawia wartości istniejące
  - zaalokowana pamięć jest dostępna dla programu do momentu jej zwolnienia (dealokacji)

Zarządzanie pamięcią

- Standardowa biblioteka C dostarcza kilka procedur zarządzania pamięcią dynamiczną:
  - funkcja dealokowania (zwalniania) obszaru pamięci:
    - free: void free(void *ptr);
    - funkcja musi otrzymać jako argument wskaźnik uzyskany z funkcji alokowania pamięci (lub wskaźnik NULL - wtedy nie robi nic)
    - zwalnianie niezaalokowanej pamięci lub podwójne zwalnianie pamięci to jedne z najczęstszych błędów wykonania w C
  - alokowanie i niezwalnianie pamięci (wycieki pamięci) jest przyczyną trudnych do wykrycia błędów (zwłaszcza w przypadku programów o długotrwałym działaniu)
    - istnieją specjalne programy, których celem jest wykrywanie wycieków pamięci:
      - poprzez przeglądanie kodu źródłowego
      - lub testowanie kodu binarnego

Zarządzanie pamięcią

- Standardowa biblioteka C dostarcza kilka procedur zarządzania pamięcią dynamiczną:
  - dynamiczne alokowanie pamięci najczęściej stosowane jest dla tablic, których rozmiar znany jest dopiero w trakcie wykonania programu

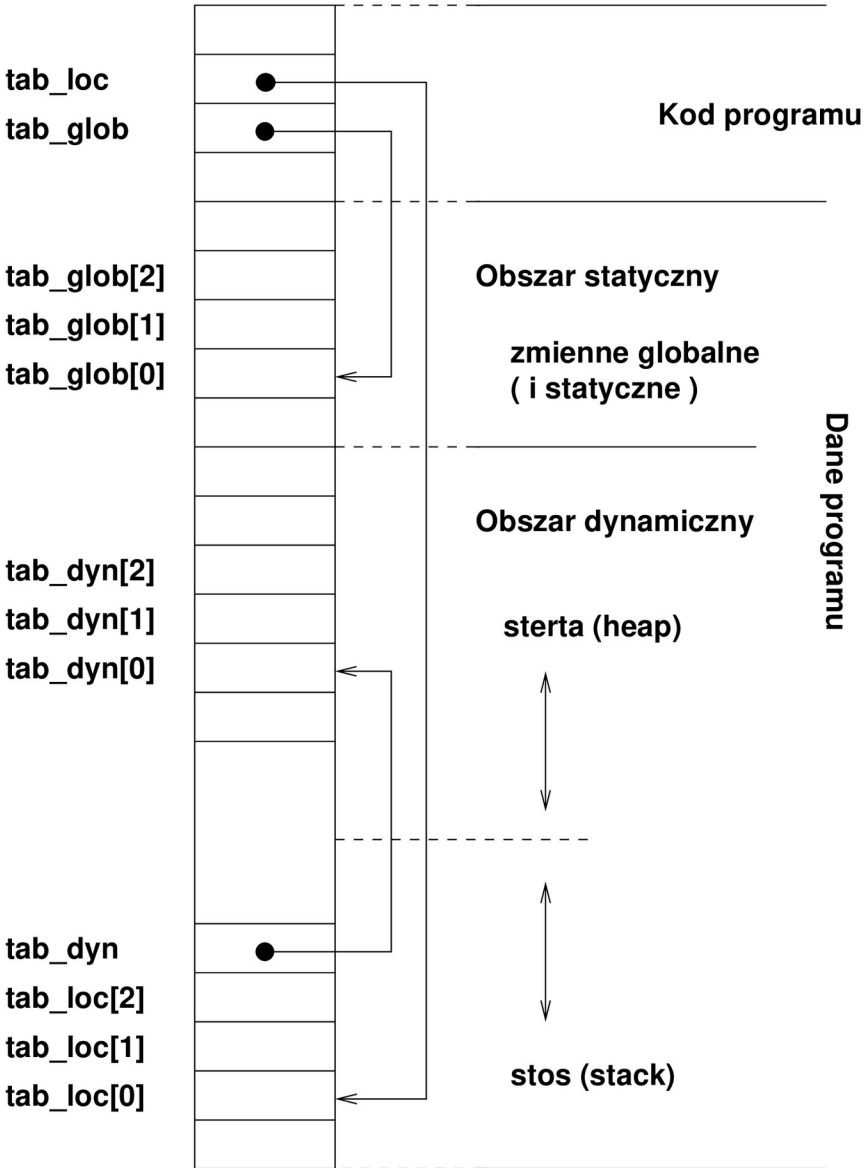
```
int* tab_int = NULL; // tab_int[0]=1; - błąd wykonania
tab_int = malloc(n*sizeof(int)); // domyślna konwersja na (int *)
..... // obszar dostępności (widzialności) tablicy
free( i_p );
..... // tab_int[0]=1; - błąd wykonania (mimo widzialności nazwy)
```

    - nowe wersje standardu C dopuszczają definiowanie tablic o zmiennym rozmiarze (element opcjonalny):

```
int tab_int[n];
```

  - umieszczając w kodzie funkcję alokowania pamięci powinno się od razu umieścić funkcję zwalniania zaalokowanej pamięci
    - najlepiej jeśli obie operacje są realizowane w tej samej funkcji

Wskaźniki i zmienne dynamiczne




```
int tab_glob[] = {11,12,13};

void main(void )
{
   int tab_loc[3] = {21,22,23};

   int* tab_dyn = malloc (3*sizeof(int));
   tab_dyn[0] = 31; // itd.
...
   free(tab_dyn);
}
```

Wskaźniki

- Bezpieczne użycie wskaźników oznacza:
  - inicjowanie w momencie definicji
  - przypisywanie tylko wartości adresów zmiennych przeznaczonych do modyfikacji poprzez wskaźnik
  - unikanie arytmetyki wskaźników
    - dokonywanie operacji arytmetycznych na wskaźnikach prowadzi w przypadku błędu do trudno wykrywalnych usterek kodu
  - stosowanie wszędzie, gdzie to odpowiednie notacji indeksowej
    - np. operowanie na tablicach przesłanych jako argumenty funkcji za pomocą indeksów, a nie arytmetyki wskaźników
- Ze względu na łatwość popełniania błędów przy stosowaniu wskaźników, wiele języków nie wprowadza pojęcia wskaźnika
  - poprawne i bezpieczne wykorzystanie wskaźników prowadzi do czytelnego i efektywnego kodu

Wskaźniki

- Problemy z wykorzystaniem wskaźników:

```
int* tab_dyn = malloc (3*sizeof(int));
tab_dyn++; // uwaga – arytmetyka wskaźników
free(tab_dyn); // ERROR – błąd wykonania (nie kompilacji!)

int* funkcja_2(int a)
{
    return(&a); // BŁĄD - zwracany adres na stosie ! (to samo dla zmiennych lokalnych)
}                    // błąd nie zgłaszany przez kompilator (ewentualnie ostrzeżenie)
void main(void )
{
    int* int_wsk_1 = funkcja_2(333); // BŁĄD – int_wsk_1 pokazuje na miejsce
                                     // które może być modyfikowane przez inne funkcje
    int a = *int_wsk_1 // nie wiadomo co to za wartość!
                       // int_wsk_1 pokazuje na dawne miejsce na stosie!
```