Wojciech Myszka

# Wskaźniki — próba podsumowania

wer. 9 z drobnymi modyfikacjami!

2017-06-12 23:11:19 +0200

# Spis treści

# 1. Wskaźniki — próba podsumowania

## 1.1. Zmienne

1. Zmienna, to po prostu zmienna: miejsce w pamięci komputera do przechowywania jakiejś jednej wartości.
2. Każda zmienna ma jakiś adres, ale tymi adresami interesujemy się dosyć rzadko.

## 1.2. Tablice

1. Tablica to pojemnik do przechowywania wielu danych tego samego typu.
2. W komputerze — adres początku, typ elementu i numer elementu to wszystko co jest potrzebne aby dobrać się do konkretnego elementu.
3. adres(t[i]) = adres(t) + i $*$ długość elementu
4. Z tego powodu w C tablice indeksowane są od 0 (zera) — bo pierwsza (o numerze zerowym) wartość w tablicy umieszczana jest na samym jej początku).

## 1.3. Adresy

1. Adres — wartość jak każda inna.
2. Do przechowywania adresów są potrzebne specjalne zmienne zwane „wskaźnikami” (*pointer*).
3. Do pobrania adresu obiektu służy operator jednoargumentowy & (*apersand*).
4. W języku C czym innym jest adres zmiennej int, czym innym adres zmiennej char, czym innym adres zmiennej double...

5. Do deklaracji wskaźników używamy specjalnego znaczka „∗” (przed nazwą).
6. Żeby utrudnić wszystkim życie wymyślono też wskaźniki bez podania typu (void).
7. Na adresach (i zmiennych zawierających adresy) można wykonywać proste operacje: dodawanie i odejmowanie stałej oraz odejmowanie adresów tego samego typu.

## 1.4. Użycie wskaźników

1. Pobranie wartości (występuje zazwyczaj po prawej stronie znaku równości): ... = ∗ip.
2. Podstawienie wartości: ∗ip = ... .

### 1.4.1. Adresowanie pośrednie

#### Adresowanie pośrednie

1. Adresowanie pośrednie czyli wpisanie jakieś wartości do miejsca pamięci wskazywanego przez adres (lub zmienną przechowującą adres):
   ∗ip = 7;

## 1.5. Funkcje

1. Funkcje nie mają wiele wspólnego ze wskaźnikami.
   Natomiast wspomieć należy, że każda funkcja musi być zapisana (jej binarny obraz) po kompilacji gdzieś w pamięci. I warto wiedzieć, że nazwa funkcji (podobnie jak nazwa tablicy) to wskaźnik.
2. Gdy argumentem funkcji jest zmienna, do wnętrza funkcji przekazywana jest jej wartość.
3. Gdy argumentem funkcji jest wyrażenie — do funkcji przekazywana jest jego wartość.

4. Co jest drugim argumentem funkcji scanf("%d", &i) (Nie zajmujmy się funkcją scanf, jest zbyt skomplikowana!) Co jest argumentem funkcji moja_funkcja( &i )?
5. Argumentem jest wyrażenie polegające na pobraniu adresu zmiennej i. I adres zmiennej i jest przekazywany do wnętrza funkcji!
6. Deklaracja funkcji powinna wyglądać jakoś tak: moja_funkcja(int *ip);
7. Co funkcja może zrobić z przekazanym jej adresem? Niezbyt wiele, ale zawsze może użyć go w celu adresowania pośredniego, czyli zrobić coś takiego:
   $*ip = 127;$
   albo
   $*ip = (*ip) + 1;$
8. Parametrem funkcji może być element tablicy:
   inna_moja_funkcja(tablica[7])
   do wnętrza funkcji przekazywana jest wartość wyrażenia polegającego na pobraniu z tablicy jej siódmego elementu.
9. Argumentem funkcji może być nazwa tablicy:
   inna_funkcja(tablica)
   Żeby to miało sens, deklaracja funkcji musi wyglądać jakoś tak:
   inna_funkcja(**int** t []);
   Jeżeli dodatkowo funkcja znać będzie jeszcze długość tablicy — będzie mogła wykonywać na niej dowolne operacje...
   Załóżmy, że chcemy napisać funkcję, która będzie zerowała (wypełniała zerami) zadaną tablicę. Do funkcji przekażemy nazwę tablicy (czyli adres jej początku) oraz jej długość.
   Funkcję można zaprogramować na kilka różnych sposbów. W pierwszym przypadku nie użyjemy (bezpośrednio) wskaźników.

```
void zerowanie(int t[], int n)
{
    int i;ą
    for(i = 0; i < n; i++)
        t[i] = 0;
}
```

W drugim przypadku, wykorzystamy wskaźniki, w deklaracji parametrów funkcji użyjemy zmiennej **int** $*$ it.

```
void zerowanie(int *it, int n)
{
    int i;
    for(i = 0; i < n; i++)
        *(it + i) = 0;
}
```

Trzeci przypadek jest „mieszany”. Deklarujemy zmienną it jako adres (wskaźnik) ale do elementów tablicy odwołujemy się w sposób „klasyczny” — korzystamy z zapisu z indeksem.

```
void zerowanie(int *it, int n)
{
    int i;
    for(i = 0; i < n; i++)
        it[i] = 0;
}
```

W pewnym sensie wszystkie zapisy są sobie równoważne.

### Tablice po raz drugi

1. Tablice muszą być deklarowane.
2. W deklaracji trzeba podać ich długość.
3. Co prawda taki kod jest poprawny:

   ```
   int n;
   n = 10;
   int tablica[n]
   ```

   ale jego działanie jest ograniczone do niezbyt wielkich n! Mi udało się, zadeklarować tablicę o 2 milionach elementów, ale już nie o 2 100 000 elementów.
4. Znacznie lepsze rozwiązanie jest takie:

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

int main(void)
{
    int n, i;
    printf("Podaj n ");
    scanf("%d", &n);
    int *tablica;
    tablica = malloc(n * sizeof(int));
    if (tablica != NULL) {
        for (i = 0; i < n; i++)
            tablica[i] = i;
/*  Tu reszta programu */
        free(tablica);
        return 0;
    } else {
        printf("Pomocy! Nie dostalem " \
               "pamieci!\n");
        return 1;
    }
}
```

## 1.6. Parę pokręconych przykładów

W dalszej części będzie kilka przykładów z dosyć szczegółowym opisem tego co (i czemu) robią. Osobna kwestia *po co to robią* nie będzie rozpatrywana.

### 1.6.1. Przykład pierwszy

**Przykład pierwszy**

```
int * f1(int N)
{
```

```
        int tab[N];
        return tab;
}
```

```
int main(int argc, char **argv)
{
        int * T;
        int i;
        T = f1(1000);
        printf("%p\n", T);
        for(i=0;i < 1000; i++)
                T[i] = -1;
        return 0;
}
```

Idea jest taka: chcemy stworzyć funkcję, która korzystając ze standardowych deklaracji tablic w języku C stworzy tablicę o zadanym rozmiarze, a jej adres początkowy zwróci do funkcji wywołującej.

Ze względu na lokalność zmiennych automatycznych — nie ma szans to działać. W funkcji f1 zostanie utworzona tablica, natomiast po „wyjściu" z funkcji przestanie ona istnieć. Zatem wartość zwracana przez funkcję wskazuje na **nic**. Pokazuje to kolejny przykład.

### Przykład pierwszy prim

```
void g1(int N)
{
        int T[N];
        printf("g1: %p\n", T);
}

void g2(int M)
{
        float T[M];
        printf("g2: %p\n", T);
}
```

```
int main(int argc, char **argv)
{
```

```
        g1(1000);
        g2(1000);
        return 0;
}
```

```
g1: 0x7ffcdd5f1d90
g2: 0x7ffcdd5f1d90
```

Jak widać, pamięć raz przydzielona w funkcji g1() jest zwalniana i ponownie przydzielana w funkcji g2(). Świadczy o ym ta sama wartość drukowanego adresu tablicy (wskaźnika)[1].

W zależności od wersjoi użytego kompilatora Zachowanie programu może byc różne. Wersja „stara" (obecna w chwli pisania tych słów w laboratorium 604 B1) podczas kompilacji zgłasza ostrzeżenie brzmiące tak:
tab1.c:32:9: warning: function returns address of local variable
ale funkcja zwraca adres wskaźnika.

Wersja kompilatora współcześniejsza[2] zgłasza podobny komunikat, natomiast sama funkcja zwraca wartość (nil) (0 — zero). Skutecznie uniemożliwia to wykorzystanie takiego wskaźnika.

### 1.6.2. Przykład drugi

### Przykład drugi

```
int * f3()
{
        static int tab[1000];
        return tab;
}
```

```
int main(int argc, char **argv)
{
        int * T;
```

[1] Specyfikacja formatu %p służy do drukowania wartości wskaźników. Wyprowadzane są one w formie szesnastkowej.

[2] 6.2.0 20161005 (Ubuntu 6.2.0-5ubuntu12)

```
        int i;
        T = f3();
        printf("%p\n", T);
        for(i=0;i < 1000; i++)
                T[i] = -1;
        return 0;
}
```

Powyższe rozwiązanie jest poprawne — ale ze względu, że tablica tab jest tablicą statyczną — nie moża zmieniać jej długości. Praktyczne znaczenie powyższego rozwiązania jest wątpliwe.

### 1.6.3. Przykład trzeci

**Przykład trzeci**

```
int * f2(int N)
{
        int * tab = malloc(N * sizeof(int));
        return tab;
}
```

```
int main(int argc, char **argv)
{
        int * T;
        int i;
        T = f2(1000);
        printf("%p\n", T);
        for(i=0;i < 1000; i++)
                T[i] = -1;
        return 0;
}
```

Jedynie te rozwiązanie jest poprawne i przydatne — wykorzystuje funkcję malloc i tworzy tablicę o zadanej długości oraz zwraca jej adres.

### 1.6.4. Przykład czwarty

**Przykład czwarty**

```
char * f4()
{
        char * tekst=
                "Ala ma kota";
        return tekst;
}
```

```
int main(int argc, char **argv)
{
        char * N;
        N = f4();
        printf("%p\n", N);
        printf("%s\n", N);
        N[1]='Z';
        return 0;
}
```

Mimo, że tekst „Ala ma kota” nie jest oznaczony jako **static**, funkcja f4() zadziała poprawnie zwracając adres do napisu. Po wyjściu z tej funkcji wskaźnik poprawnie wskazuje na ten napis. Co więcej napis „Ala ma kota” jest traktowany jako stała[3] i zapisywany w obszarze pamięci chronionej przed zmianami. Tak więc próba wykonania polecenia N[1] = ‘Z’; zakończy się błędem wykonania (*Segmentation fault*).

## 1.7. Tablice znakowe

Tablice znakowe mogą sprawiać różne kłopoty. Warto pamiętać, że sposób w jaki obsługiwane są tablice znakowe jest identyczny, jak sposób obsługi tablic „numerycznych”.

Zapewne bardzo wygodnym byłby zapis:

```
1 int a[10];
2 int b[10];
3 for( i = 0; i < 10; i++)
4         a[i] = i * i;
```

[3] I jeżeli gdziekolwiek w programie użyjemy takiej stałej tekstowej — będzie to **ta sama** stała.

```
5  b = a;
```

Chciałoby się oczekiwać, że linia 5 znaczy tyle co „zawartość tablicy a skopiuj do tablicy b. Niestety tak nie jest. Trzeba napisać sobie funkcję realizującą taką operację:

```
void copy_tab(int N, int * A, int * B)
{
        int i;
        for(i = 0; i < N; i++)
                *(B + i) = *(A + I);
}
```

**Tablice znakowe: problemy**

Bardzo naturalny zapis:

```
char * a = "Ala ma kota";
char * b;
```

Oznacza on tyle, że deklarujemy dwie zmienne wskaźnikowe: a oraz b. Pierwszej z nich przypisujemy adres stałej znakowej zawierającej napis ‚Ala ma kota". Czy można napisać

```
b = a;
```

Tak. Zapis oznacza tyle, że adres (wskaźnik) zawarty w zmiennej a zostaje przepisany do zmiennej b. Oba wskaźniki będą wskazywały na te same miejsce.

Czy można napisać tak:

```
char c = a[5];
```

Tak. Piąty znak z napisu „Ala ma kota" („a" z wyrazu „ma") zostanie przypisany do zmiennej c.

A czy można napisać tak:

```
a[5] = 'm';
```

żeby wyraz „ma” zamienić na „mm”?

Nie. Co prawda a to „tablica”, ale jej zawartość jest stałą. A stałych nie można zmieniać!

**Tablice znakowe: problemy (cd)**

Inny naturalny zapis:

```
char a[] = "Ala␣ma␣kota";
char b[100];
```

Oznacza on tyle, że deklarujemy dwie zmienne wskaźnikowe: a oraz b. Pierwszej z nich przypisujemy adres stałej znakowej zawierającej napis ‚Ala ma kota”. Czy można napisać

```
b = a;
```

Nie. Zapis oznacza tyle, że adres (wskaźnik) związany z nazwą a chcemy przypisać do b. Ale a oraz b to stałe typu **char** $*$.

Czy można napisać tak:

```
char c = a[5];
```

Tak. Piąty znak z napisu „Ala ma kota” („a” z wyrazu „ma”) zostanie przypisany do zmiennej c.

A czy można napisać tak:

```
a[5] = 'm';
```

żeby wyraz „ma” zamienić na „mm”?

Oczywiście. ’m’ nadpisze zawartą w tablicy literę ’a’.

Poza tym oba zapisy są właściwie równoważne. Ale, zwracam uwagę, że problemy zazwyczaj związane są z subtelnościami.

## 1.8. Problemy z sizeof

Na jednym z zajęć projektowych/laboratoryjnych zaproponowałem następujący trik pozwalający łatwo wyznaczyć rozmiar tablicy:

```
double x[] = {
    0., 1., 2., 3., 4., 5.,
};
int n = sizeof ( x ) / sizeof ( double );
```

Pozwala to łatwo zwiększyć/zmniejszyć rozmiar tablicy :

```
double x[] = {
    0., 1., 2., // 3., 4., 5.,
};
int n = sizeof ( x ) / sizeof ( double );
```

Dodany komentarz „ukrywa" część wartości i tablica ulega automatycznemu skróceniu. Łatwo dopisywać kolejne wartości. Bardzo przydatne zwłaszcza podczas testowania programów.

**Dziwne zastosowania**

Wiele osób uznało to za „uniwersalną" metodę wyznaczania rozmiaru tablic. I stosuje ją w bardzo różnych sytuacjach.

1. Do wyznaczania długości tekstów:

   ```
   char a[] = "Ala ma kota";
   int n = sizeof( a ) / sizeof( char );
   ```

   a czasami nawet sprytniej:

   ```
   int n = sizeof( a ) / sizeof( char ) - 1;
   ```

   Lepiej użyć funkcji strlen!
   A taki prosty programik robi to samo:

   ```
   n = 0;
   while (a[n]) n++;
   ```

2. Wewnątrz funkcji. Czyli jakoś tak:
   W funkcji main mamy:

   ```
   double a[] = {1., 2., 3., 4.,};
   double avg = srednia( a );
   ```

A w funkcji:

```
double srednia (double x[])
{
        double srednia = 0.;
        int n = sizeof( x ) / sizeof( double );
        int i;
        for(i = 0; i < n; i++)
                srednia += x[i];
        return srednia / n;
}
```

I cały problem polega na tym, że mierzymy nie to o czym myślimy. Porządny kompilator zgłosi taki komunikat:

```
warning: ‘sizeof’ on array function parameter ‘x’
will return size of ‘double *’ [-Wsizeof-array-argument]
  int n = sizeof( x ) / sizeof( double );
                  ^
```

Gdyż to, co przed chwilą, było tablicą — w funkcji jest tylko i wyłącznie **adresem** początku tablicy. Resztę załatwia magia wskaźników.
Jako dobrą zasadę możemy przyjąć, że jeżeli tworzymy funkcję operującą na tablicach — jednym z parametrów **musi być** rozmiar tablicy.

## 1.9. Zmiana przydziału pamięci

Nie jest to może najlepsze określenie, ale na razie nie znajduję nic lepszego. Chodzi o to, że w miarę potrzeby pamięcią przydzieloną z wykorzystaniem funkcji malloc można „gospodarować" — zmniejszając lub zwiększając jej ilość. W dalszej części podam prosty dosyć przykład takiego zastosowania funkcji realloc.

Zadanie jest takie:

— Wczytać mamy dane nieznanej długości.
  — Może to być linia tekstu.

— Może to być strumień danych (nieznanej objętości), który należy wczytać w całości zanim będzie mógł być przetworzony.

W instrukcjach laboratoryjnych rozważane są różne pomysły jak problem rozwiązać, tu opiszę dokładniej jedno z rozwiązań.

**Funkcja czytanie**

```
1  /*
2   * napis.c
3   * Copyright 2016 wojciech myszka <myszka@norka.eu.org>
4   */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  char * czytaj(void)
9  {
```

dN (linia 10) zawiera informację o „kwancie" przydzielanych bajtów. Może to być praktycznie dowolna wartość: gdy pamięć zostanie zapełniona, przydzielony zostanie następny kwant. Gdy odczyt danych się zakończy, niezapełniona danymi pamięć będzie zwolniona.

```
10 #define dN 10
11     int N = dN; // początkowy przydział
12     int i;
13 /*
14  * Przydzielamy funkcją malloc pamięć na dN znaków
15  */
16     char * bufor;
17     bufor = (char *) malloc(N);
```

Po każdym wykonaniu funkcji malloc/calloc/realloc należy sprawdzić czy zostałą ona wykonana poprawnie. Jakikolwiek błąd powoduje, że funkcja zwraca wartość 0. W języku C zdefiniowana jest stałą NULL typu wskaźnikowego o wartości zero. Przyjęło się, że obszar pamięci ozaczynający się od adresu 0 jest zarezerwowany i niedostępny do programów.

Gdy wystąpi błąd (co raczej jest mało prawdopodobne) — funkcja napis zwraca wartość NULL co jest informacją dla programu wywołującego, że coś poszło źle.

```
18      if ( bufor == NULL )
19          return bufor;
```

Odczyt znaków wykonujemy w nieskończonej pętli (linia 21). Pętla się kończy gdy strumień danych wejściowych się skończy (warunek EOF w linii 24) lub napotkamy znak końca wiersza.

Rozpoznawanie końca wiersza ma sens gdy czytamy informację tekstową, poszczególne „rekordy" oddzielane są znakami \\n (naciśnięcia klawisza Enter). Program będzie działał gdy odrzucimy drugą część warunku w linii 24.

```
20      i = 0; // Liczba przeczytanych znaków
21      while ( 1 )
22      {
23          bufor[i] = getchar();
24          if ( bufor[i] == EOF || bufor[i] == '\n' )
```

Gdy uznajemy, że nie będzie już żadnych dodatkowych informacji — program najpierw „kończy" tekst znakiem o kodzie ASCII 0 (linia 26), a później zwalnia niewykorzystną pamięć używając funkcji realloc. Argumentem te funkcji jest liczba wykorzystanych bajtów pamięci (i przeczytanych i dodane zero, zatem i + 1).

```
25          {
26              bufor[i] = 0; // koniec tekstu
27              bufor = realloc(bufor, i + 1); // zwalniamy
28                                             // nadmiarową pamięć
29              return bufor;
30          }
31          i++;
```

Przeczytany znak został zapisany w pamięci, zatem zwiększamy i (linia 31) oczekując na kolejny znak. Gdy okaże się że grozi przekroczenie pamięci (linia 32) — będziemy musieli przydzielić kolejny kwant pamięci.

Zmienna N (linia 36) gromadzi informacje o sumarycznym rozmiarze bufora i jest odpowiednio ukatualniana.

```
32          if ( i >= N ) // Czy skonczyla sie przydzielona
33                                                  // pamiec?
34          {
35              bufor = realloc(bufor, i + dN);
36              N += dN;
37              printf("%p %d\n", bufor, N);
38          }
39      }
40  }
```

Wydruk (polecenie printf w linii 37) ma charakter diagnostyczny. Aby uprościć kod nie sprawdzam, czy polecenie realloc wykonało się poprawnie (i funkcja zwróciła wartość różną od zera), ale w prawdziwych programach powinno to się robić.

```
41  int main(int argc, char **argv)
42  {
43      char * tekst;
44      char bufor[10];
45      tekst = czytaj();
46      if ( tekst != NULL ) // Sprawdzamy czy coś przeczytane
47          printf("przeczytalem: %d znakow\n \"%s\"\n",
48                  (int) strlen(tekst), tekst);
49      free(tekst); // Zwalniamy przydzieloną pamięć
50      return 0;
51  }
```

## 1.10. Wskaźniki do funkcji

### Metoda połowienia

1. Zadanie jest proste. Mamy funkcję $f(x)$ ciągłą i taką, że na końcach pewnego przedziału $[A, B]$ $f(A)f(B) < 0$. Zatem, funkcja ta zmienia

znak w przedziale $[A, B]$ (co najmniej raz) ma zatem (co najmniej jedno) miejsce zerowe w tym przedziale.
2. Przedział $[A, B]$ dzielimy na pół (wyznaczając odpowiednio punkt $C$).
3. Odrzucamy ten z przedziałów $[A, C]$, $[C, B]$ w którym funkcja nie zmienia znaku (to znaczy ma ten sam znak na końcach przedziału).
4. Postępowanie prowadzimy tak długo, aż długość przedziału $[A, B]$ będzie mniejsza od zadanej liczby $\varepsilon$.

Uwaga: Obliczenia najprościej wykonać dla funkcji sin wybierając $0 < A < 3$ i $3{,}5 < B < 6$.

Powyższe zadanie można również zaprogramować korzystając z rekurencji!

### Realizacja

```
1  double polowienie(double A, double B)
2  {
3     double C;
4  pocz:
5     C = ( A + B ) / 2.;
6     if ( f(A) * f(C) < 0 )
7        B = C;
8     else if ( f(A) * f(C) > 0 )
9        A = C;
10    else
11       return C;
12    if ( fabs(A - B) > 0.001 )
13       goto pocz;
14    return C;
15 }
```

Tu jest użyta instrukcja **goto** ale można inaczej.

Aby pozbyć się polecenia **goto** wyrzucamy etykietę w linii 4 i wstawiamy tam **do** { oraz linie 12 i 13 zastępujemy zamknięciem polecenie **do**: **while** ( fabs(A − B) > 0.00000001 );, otrzymując:

### Inna realizacja

```
1  double polowienie(double A, double B)
```

```
2  {
3      double C;
4      do
5      {
6          C = ( A + B ) / 2.;
7          if ( f(A) * f(C) < 0 )
8              B = C;
9          else if ( f(A) * f(C) > 0 )
10             A = C;
11         else
12             return C;
13     }
14     while ( fabs(A - B) > 0.00000001 );
15     return C;
16 }
```

Z pętlą **do**.

„Zapętlenie" można zrealizować również wywołując ponownie funkcję polowienie i realizując rekurencję. Ponownie wyrzucamy linie 4 i 5 oraz zastępując linie 14 (**while**) i 15 ponownie instrukcją **if**:

```
    if ( fabs(A - B) >  0.00000001 )
        return polowienie(A, B);
    else
        return C;
```

oytrzymując:

**Jeszcze inna realizacja**

Zastąpimy pętlę **do** rekurencją.

```
1  double polowienie(double A, double B)
2  {
3      double C;
4          C = ( A + B ) / 2.;
5          if ( f(A) * f(C) < 0 )
6              B = C;
7          else if ( f(A) * f(C) > 0 )
8              A = C;
9          else
10             return C;
11     if ( fabs(A - B) >  0.00000001 )
```

```
12        return polowienie(A, B);
13    else
14        return C;
15 }
```

**Jak z tego skorzystać?**

Aby z tego programu skorzystać, trzeba jego kod dołączyć do naszego oraz napisać funkcję pomocniczą f

```
double f(double x)
{
        return sin(x);
}
```

oraz wywołać:

```
u = polowienie(2., 4.);
```

Niestety, w kodzie programu na stałę jest zapisana nazwa funkcji której miejsca zerowego szukamy. Nie jest to zbyt wygodne. Najwygodniej byłoby uczynić z funkcji jeszcze jedną zmienną. Do tego mogą przydać się wskaźniki.

**Wskaźnik do funkcji**

1. Deklarazja zwykłej zmiennej wygląda tak:

   ```
   typ nazwa;
   ```

2. A wskaźnik deklaruje się tak:

   ```
   typ * inna_nazwa;
   ```

3. Funkcję deklarujemy tak (mam na myśli prototyp):

   ```
   typ_funkcji nazwa_funkcji( typ_argumentu );
   ```

4. A wskaźnik? Przez analogię:

   ```
   typ_funkcji * nazwa_funkcji( typ_argumentu );
   ```

### Połowienie jeszcze inaczej

```
1  double polowienieR( double A, double B, double f(double) )
2  {
3      double C;
4      C = ( A + B ) / 2.;
5      if ( f(A) * f(C) < 0 )
6          B = C;
7      else if ( f(A) * f(C) > 0 )
8          A = C;
9      else
10         return C;
11     if ( fabs(A - B) > 0.00000001 )
12         return polowienieR(A, B, f);
13     else
14         return C;
15 }
```

### Połowienie jeszcze inaczej(cd)

```
double polowienieR( double A, double B, double f(double) )
```

Zwracam uwagę na trzeci argument w definicji funkcji polowienieR — jest to funkcja. Zatem z polowienieR można korzystać w sposób następujący:

```
double f(double x)
{
    return sin(x);
}
...
    double ( *g )(double);
    g = f;
...
    y = polowienieR(A, B, g);
```

albo

```
y = polowienieR(A, B, f);
```

albo

```
y = polowienieR(A, B, sin);
```

Dodatkowo po wykonaniu podstawienia g = f (albo g = sin) symbol (**zmienna**) g staje się „aliasem” (przezwiskiem) podstawionej funkcji.