15 Funkcje

Materiał omawiany na laboratoriach zostanie tu powtórzony i rozszerzony. Wydaje mi się, że łatwiej Państwu będzie przygotowywać się do egzaminu, gdy informacje będą w jednym miejscu.

15.1 Dlaczego używamy funkcji

Funkcje są podstawowymi budulcami programu. Zamykają w sobie ciąg instrukcji służący pewnemu zadaniu, opatrując go nazwą. Zadanie to może być sparametryzowane, a jego wyniki przekazane na zewnątrz. Zadanie może być wielokrotnie wykonane w jednym programie z różnymi wartościami parametrów.

Funkcje służą do podzielenia programu na mniejsze fragmenty. Ułatwia to jego czytanie i zrozumienie, zwłaszcza, gdy ich nazwy są starannie dobrane, bo pozwala na skupieniu się nad zasadniczym jego celem z pominięciem szczegółów. Podział ten powinien być bezpośrednim odzwierciedleniem algorytmu.

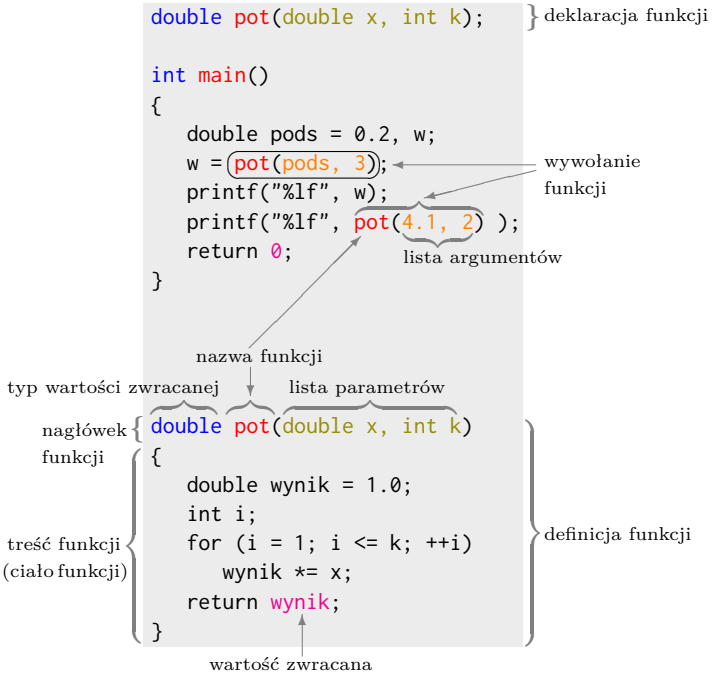
Podział programu na funkcje stosujemy w celu:

1. **zwiększenia czytelności programu** — małe fragmenty łatwiej zrozumieć,
2. **wprowadzenia lepszej organizacji kodu** — funkcje pozwalają na wyodrębnienie w dużym, skomplikowanym programie jego prostych składowych i „zlepienie" z nich całości
3. **umożliwienia pracy zespołowej** — nad różnymi funkcjami mogą pracować różni programiści,
4. **możliwości użycia kodu bez konieczności rozumienia, jak dokładnie działa** — funkcja działa jak czarna skrzynka: dajemy jej dane, ona produkuje wyniki, wiemy jak wyniki zależą od danych, ale nie musimy wiedzieć jak skrzynka je produkuje,
5. **ułatwienia testowania programu** — każdą funkcję testujemy osobno (zwykle nie są to trudne testy), potem testujemy współpracę funkcji,
6. **unikania powtarzania podobnych fragmentów kodu** — ułatwia to jego poprawianie: dokonujemy zmian w jednym miejscu, a nie kilku,
7. **ułatwienia rozbudowy programu** — dostosowujemy do nowych warunków wybrane funkcje, zmiany pojawiają się we wszystkich miejscach,
8. **umożliwienia ponownego użycia kodu** — gotowe i przetestowane fragmenty można użyć w innym programie bez konieczności ich dostosowywania.

15.2 Wskazówki dotyczące pisania funkcji

1. Jeśli w programie podobny kod występuje dwa razy, to jest to powód do napisania funkcji.
2. Jeśli fragment kodu realizuje konkretną, dobrze zdefiniowaną czynność, którą da się nazwać, to jest to powód, by stał się on funkcją, nawet gdy w programie występuje tylko jeden raz.
3. Dobrze napisana funkcja powinna zasadniczo wykonywać jedno zadanie (np. liczyć pole trójkąta, a nie liczyć pole trójkąta i przy okazji znajdować najdłuższy bok).
4. Funkcja powinna być krótka (maksymalnie 30 - 40 linii), jeśli jest dłuższa, to trzeba w niej wyodrębnić fragmenty, które staną się osobnymi funkcjami.

15.3 Terminologia



15.4 Definicja funkcji

Definicja 35 *Definicja funkcji*[57] jest to dokładny opis akcji, które funkcja wykonuje.

Definicja funkcji składa się z nagłówka oraz treści.

Definicja funkcji zaczyna się od *nagłówka*[58], w którego skład wchodzą poniższe informacje o funkcji.

1. *Nazwa funkcji*[59], czyli poprawny w danym języku identyfikator, który nie jest słowem kluczowym zarezerwowanym. Każda funkcja ma unikatową nazwę (nie wolno w jednym programie stworzyć dwóch funkcji o tej samej nazwie).
2. *Lista parametrów*[60], która składa się z nazw parametrów oraz ich typów. Parametry służą do przekazywania wartości z miejsca wywołania do treści funkcji. Lista ta może być pusta, wtedy zaleca się wpisanie słowa `void`[61].
3. Typ *zwracanej wartości*[62], czyli wartości, którą funkcja może przekazać do miejsca wywołania. Jeśli funkcja nie ma wartości zwracanej, należy wpisać słowo `void`.

[57] ang. function definition
[58] ang. header
[59] ang. function name
[60] ang. parameter list
[61] Parametry funkcji bywają też nazywane *parametrami formalnymi*.
[62] ang. returned value

Przykład (dlaczego **trzeba** pisać `void`, gdy funkcja nic nie zwraca)

Pozostawienie pustego miejsca przed nazwą funkcji oznacza, że zostanie ustalony domyślny tym wartości zwracanej, czyli `int`. Funkcja o nagłówku

```
void f(float x)
```

nie posiada wartości zwracanej, ale nagłówek

```
g(float x)
```

oznacza, że `g` ma zwraca wartość typu `int`. ■

Przykład (dlaczego **zaleca się** pisać `void`, gdy lista parametrów jest pusta)

Pozostawienie pustego miejsca, gdy lista parametrów jest pusta powoduje, że otrzymujemy od kompilatora mniej informacji o błędach. Podczas kompilacji w Visual Studio programu

```
int a() {
   return 1;
}

main() {
   a(1);
}
```

nie są zgłaszane żadne błędy, ani ostrzeżenia. Jeśli kompilujemy program

```
int a(void) {
   return 1;
}

main() {
   a(1);
}
```

to pojawia się ostrzeżenie

```
Warning  C4087 'a': declared with 'void'
parameter list
```

■

Po nagłówku następuje *treść funkcji*[63], czyli blok instrukcji realizujący zadanie przypisane funkcji. W bloku tym wolno deklarować zmienne lokalne funkcji. Jeśli funkcja ma wartość zwracaną, to wewnątrz treści musi wystąpić instrukcja `return`, a za nią wyrażenie, którego wartość jest wartością zwracaną funkcji. Instrukcja `return` może pojawić się w treści wielokrotnie. Jej wykonanie kończy działanie funkcji.

- Funkcja może mieć tylko jedną definicję.
- Nazwa funkcji powinna odzwierciedlać przeznaczenie funkcji.
- Parametrów powinno być dokładnie tyle, co trzeba, a nie więcej. Jeśli nie wiemy, jaką wartość przekazać do parametru w miejscu wywołania, to parametr jest zbędny.
- Jeśli potrzebujemy zmiennych służących tylko do realizacji zadań wykonywanych przez funkcję, to powinny być one zmiennymi lokalnymi funkcji, a nie parametrami.
- Funkcje nie widzą zmiennych lokalnych i parametrów innych funkcji.
- Parametry i zmienne lokalne różnych funkcji mogą się tak samo nazywać. Nie oznacza to, by miały one ze sobą jakikolwiek związek.

15.5 Deklaracja funkcji

Podprogram może mieć, obok definicji, również deklarację. Nie jest ona obowiązkowa, ale jest powszechnie stosowana.

Definicja 36 *Deklaracja funkcji*[64] jest to zapowiedź definicji funkcji, przekazująca informację, że funkcja o pewnej nazwie, będzie lub jest już zdefiniowana.

Deklaracja składa się z nagłówka funkcji zakończonego średnikiem, przy czym dopuszcza się ominięcie nazw parametrów lub ich zmianę, ale muszą pozostać ich typy.

Przykład Deklaracja funkcji z sekcji 15.3 może mieć każdą z poniższych form.

```
double pot(double x, int k);
double pot(double x, int n);
double pot(double a, int b);
double pot(double, int);
```

W praktyce używa się albo pierwszej, albo ostatniej formy. ■

Przed wywołaniem funkcji musi znajdować się definicja lub deklaracja tej funkcji.

Jeżeli pojawi się wywołanie funkcji, której wcześniej nie zdefiniowano lub nie zadeklarowano, to wystąpi błąd kompilacji.

Stosowanie deklaracji jest konieczne, jeśli kompilator musi przetłumaczyć wywołanie funkcji przed przetłumaczeniem jej definicji. Dzieje się tak np. wtedy, gdy dwie funkcje nawzajem się wywołują w swoich treściach.

Przykład Rozpatrzmy następujący szkielet programu w C.

```
int f2();

int f1(){
   ...
   f2();
}

int f2(){
   ...
   f1();
}
```

Bez początkowej deklaracji, kompilator nie przetłumaczyłby treści funkcji `f1`, bo nazwa `f2` byłaby nieznana. ■

[63] ang. function body

[64] ang. function declaration

Typowy układ programu zapisanego w jednym pliku jest następujący.

- Włączenie wszystkich bibliotek.
- Linia przerwy.
- Definicje wszystkich stałych (wraz z komentarzami).
- Linia przerwy.
- Deklaracje wszystkich funkcji (poza `main`) w dowolnej kolejności.
- Linia przerwy.
- Definicje wszystkich funkcji wraz `main` w dowolnej kolejności. Po każdej definicji linia przerwy.

15.6 Wywołanie funkcji

Napisanie definicji funkcji nie powoduje, że zostanie ona uruchomiona. Aby uruchomić funkcję, trzeba ją w odpowiednim miejscu wywołać.

Definicja 37 *Wywołanie funkcji*[65] jest żądaniem wykonania treści dla pewnych szczególnych wartości parametrów. Te szczególne wartości nazywa się *argumentami*[66].

Wywołanie następuje przez napisanie nazwy funkcji, a za nią w nawiasach okrągłych listy argumentów.

Odpowiedniość między parametrem a argumentem (czyli „co za co podstawić") ustalana jest na podstawie pozycji w liście: pierwszy argument odpowiada pierwszemu parametrowi itd. Każdą funkcję można wielokrotnie wywoływać w jednym programie. Podczas wywołania, wartości argumentów są kopiowane do parametrów.

Zgodnie z konwencją języka C, po uruchomieniu programu, funkcja `main` jest wywoływany w sposób automatyczny, jako pierwsza czynność.

Pozostałe funkcje są zawsze wywoływane wewnątrz innych funkcji.

W momencie wywołania następuje

1. zawieszenie działania funkcji wywołującej,
2. alokacja na stosie parametrów funkcji wywoływanej,
3. obliczenie wartości argumentów i skopiowanie ich do parametrów,
4. przekazanie sterowania na początek treści funkcji wywoływanej.

Następnie wykonywana jest treść funkcji. Jeśli są tam definicje zmiennych lokalnych, to następuje ich alokacja na stosie (chyba, że są one zdefiniowane jako statyczne).

Zakończenie działania funkcji następuje albo przez dojście do instrukcji `return`, albo przez dojście do klamry zamykającej treść funkcji. Podczas kończenia działania funkcji

1. jest obliczana wartość zwracana (wartość wyrażenia za `return`), o ile funkcja ją posiada, i zapamiętywana w przeznaczonym do tego miejscu w pamięci,
2. parametry i zmienne lokalne są dealokowane,
3. sterowanie wraca do miejsca wywołania,
4. wartość zwracana jest odczytana z miejsca, w którym jest zapisana i użyta w miejscu wywołania.

W C możliwy jest **każdy typ** wartości zwracanej **oprócz typu tablicowego i funkcyjnego** [67].

Definicja 38 Funkcję nazywa się *aktywną*, jeśli została wywołana i wykonywanie jej treści nie zakończyło się.

W trakcie działania programu tylko jedna funkcja w danym momencie jest wykonywana, czyli aktywna.

Przykład (Zadanie 35)

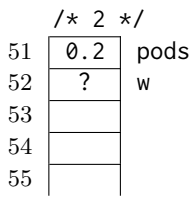
Prześledźmy działanie poniższego programu.

```
double pot(double x, unsigned int k); /*0*/

main()
{                           /*1*/
   double pods = 0.2, w;    /*2*/
   w = pot(pods, 1);        /*3*/
   printf("%lf", w);        /*4*/
   return 0;                /*5*/
}                           /*6*/

double pot(double x, unsigned int k) /*5*/
{                                    /*6*/
   double wynik = 1.0;               /*7*/
   int i;                            /*8*/
   for (i = 1; i <= k; ++i)          /*9*/
      wynik *= x;                   /*10*/
   return wynik;                    /*11*/
}                                    /*12*/
```

Program zaczyna wykonanie od pierwszej linii funkcji `main`. Nie jest to zależne od tego, czy `main` jest pierwszą, czy ostatnią funkcją w pliku.



Linia 3 zawiera wywołanie funkcji `pot`. Aby wywołać funkcję, potrzebne są informacje zarówno z miejsca wywołania, jak i **nagłówka definicji** funkcji. Linie 3 i 5 wykonywane są łącznie. Na stosie alokowane są parametry `x` i `k` i inicjalizowane wartościami argumentów z miejsca wywołania, czyli z linii 3. Wartość zmiennej `pods` jest odczytywana i wstawiana do parametru `x`. Zmienna `pods` nie jest widoczna w treści funkcji `pot`.

[66] ang. function call

[66] Czasami argumenty wywołania nazywane są też *parametrami aktualnymi*.

[67] Typ funkcyjny nie był omawiany, jest to temat zaawansowany.

```
   /* 3/5 */
51 | 0.2 | pods
52 |  ?  | w
53 | 0.2 | x
54 |  1  | k
55 |     |
```

Wykonywana jest treść funkcji `pot`. Najpierw na stosie są alokowane zmienne lokalne.

```
   /* 7,8 */
51 | 0.2 | pods
52 |  ?  | w
53 | 0.2 | x
54 |  1  | k
55 |  1  | wynik
56 |  ?  | i
57 |     |
```

Dalej wykonywana jest pętla.

```
   /* 9a,b, 10 */
51 | 0.2 | pods
52 |  ?  | w
53 | 0.2 |  x
54 |  1  |  k
55 | 0.2 | wynik
56 |  1  | i
57 | true |
```

```
   /* 9c,b */
51 | 0.2 | pods
52 |  ?  | w
53 | 0.2 |  x
54 |  1  |  k
55 | 0.2 | wynik
56 |  2  | i
57 | false |
```

Instrukcja `return`

1. oblicza wartość wyrażenia stojącego za nią, czyli odczytuje wartość zmiennej `wynik` i zapisuje w przeznaczonym do tego miejscu w pamięci, które rysujemy obok,
2. zwalnia pamięć po wszystkich zmiennych lokalnych i parametrach,
3. przerywa działanie funkcji i program wraca do miejsca wywołania.

```
   /* 11 */
51 | 0.2 | pods
52 |  ?  | w
53 |  ?  | ?
54 |  ?  | ?
55 |  ?  | ?
56 |  ?  | ?
57 |     |        | 0.2 |
```

Powrót do miejsca wywołania oznacza powrót do linii 3. Odczytujemy wartość zwróconą (przestajemy ją rysować) i używamy. Użycie można sobie wyobrazić tak, jakby zamiast wywołania `pot(pods, 1)` pojawiła się wartość zwrócona, czyli trzeba wykonać przypisanie `w = 0.2`.

```
   /* 3 */
51 | 0.2 | pods
52 | 0.2 | w
53 |     |
54 |     |
```

Na koniec program wypisuje na ekran komunikat i kończąc działanie dealokuje zmienne.

```
   /* 4 */
51 | pbz |
52 |     |
53 | ekran |
54 | 0.2 |
```

```
   /* 5 */
51 |  ?  | ?
52 |  ?  | ?
53 |     |
54 |     |
```

Wartość zwrócona przez funkcję `main` jest przekazywana do systemu operacyjnego. ■

- Parametry i zmienne lokalne funkcji żyją tylko tak długo, jak długo jest wykonywana treść funkcji.
- Każde wywołanie funkcji na nowo alokuje parametry i zmienne lokalne, funkcja kończąc działanie dealokuje je. Wyjątek stanowią zmienne lokalne, których deklaracje są poprzedzone słowem `static`.
- Kolejne wywołanie funkcji nie ma dostępu do wartości parametrów i zmiennych lokalnych z poprzedniego wywołania. Wyjątek stanowią zmienne lokalne, których deklaracje są poprzedzone słowem `static`.
- Jeśli argument wywołania jest zmienną, to jego wartość jest kopiowana do parametru. Zmiany wartości parametru w treści funkcji dotyczą jedynie parametru i nie mają wpływu na wartość argumentu wywołania.

15.7 Parametry

Parametry służą do przekazywania danych pomiędzy miejscem wywołania a treścią funkcji. W C istnieje tylko jeden sposób, który nazywa się „przekazywaniem przez wartość”.

Definicja 39 Mówimy, że parametr jest przekazywany *przez wartość*, jeśli

1. jest on traktowany jak zmienna lokalna funkcji,
2. argument nie musi być zmienną, może być wyrażeniem,
3. wartość argumentu inicjalizuje parametr.

Wartość argumentu jest obliczana przed wywołaniem funkcji. Na początku wywołania przydzielana jest pamięć parametrom na stosie i są one inicjalizowane wartościami argumentów. W momencie zakończenia funkcji pamięć przydzielona parametrom zostaje zwolniona. Zatem czasem życia parametru przekazywanego przez wartość jest okres, w którym funkcja jest aktywna.

15.8 Parametry typu wskaźnikowego

Przekazywanie przez wartość przekazuje informacje z miejsca wywołania do treści funkcji. Gdy chcemy przekazać informację z treści funkcji do miejsca wywołania używamy wartości zwracanej. Co jednak mamy zrobić, gdybyśmy z treści funkcji do miejsca wywołani chcieli przekazać nie jedną liczbę, ale kilka i na dodatek różnych typów?

Są dwa rozwiązania tego problemu

1. użycie struktur (struktury nie będą omówione na wykładzie),
2. użycie parametrów typu wskaźnikowego.

Ten drugi sposób „oszukuje” mechanizm przekazywania przez wartość. Przekazując adres pewnej zmiennej, uzyskujemy do niej dostęp przez operację adresowania pośredniego.

Przykład (Zadanie 36)

Prześledźmy działanie poniższego programu.

```
void pot(double x, unsigned k, double *wynik) /*1*/
{
   int i;                          /*2*/
   *wynik = 1.0;                   /*3*/
   for (i = 1; i <= k; ++i)        /*4*/
      *wynik *= x;                 /*5*/
}                                  /*6*/

main()
{                                  /*7*/
   double pods = 0.2, w;           /*8*/
   pot(pods, 1, &w);               /*9*/
   printf("%lf", w);              /*10*/
   return 0;                      /*11*/
}                                 /*12*/
```

Program ten jest podobny do programu z poprzedniego przykładu, ale funkcja `pot` nie zwraca żadnej wartości. Wynik jej działania jest bezpośrednio obliczany w zmiennej `w`, do której dostęp jest zapewniony przez wskaźnik `wynik`.

Od razu chcę zaznaczyć, że wersja z Zadania 35 jest lepsza. Wykorzystuje bowiem naturalne sposoby komunikacji między funkcjami.

Program zaczyna działanie od linii 8 i w linii 9 następuje wywołanie funkcji.

/* 8 */

51	0.2	pods
52	?	w
53		
54		
55		
56		
57		

/* 9/1 */

51	0.2	pods
52	?	w
53	**0.2**	**x**
54	**1**	**k**
55	**52**	**wynik**
56		
57		

Dzięki parametrowi `wynik`, który jest wskaźnikiem, mamy dostęp do zmiennej `w` z funkcji `main` przez operację adresowania pośredniego.

/* 2 */

51	0.2	pods
52	?	w
53	0.2	x
54	1	k
55	52	wynik
56	**?**	**i**
57		

/* 3 */

51	0.2	pods
52	**1**	w
53	0.2	x
54	1	k
55	52	wynik
56	?	i
57		

Funkcja `pot` nie widzi zmiennej `w` z `main`, ale może ją zmieniać dzięki znajomości jej adresu. Każde przypisanie

```
*wynik = ...
```

będzie dotyczyć `w`.

/* 4a,b,5 */

51	0.2	pods
52	**0.2**	w
53	0.2	x
54	1	k
55	52	wynik
56	**1**	i
57	true	

/* 4c,b */

51	0.2	pods
52	0.2	w
53	0.2	x
54	1	k
55	52	wynik
56	**2**	i
57	false	

Funkcja kończy działanie przez dojście do klamry zamykającej. Nie jest obliczana wartość zwracana, bo funkcja jej nie posiada. Następuje jedynie dealokacja zmiennych lokalnych i parametrów. Po powrocie do miejsca wywołania nic się nie dzieje, ale stan pamięci trzeba narysować.

/* 6 */

51	0.2	pods
52	0.2	w
53	**?**	**?**
54	**?**	**?**
55	**?**	**?**
56	**?**	**?**
57		

/* 9 */

51	0.2	pods
52	0.2	w
53		
54		
55		
56		
57		

Na koniec program wypisuje na ekran komunikat i kończąc działanie dealokuje zmienne.

/* 10 */

51	pbz
52	
53	ekran
54	0.2

/* 11 */

51	**?**	**?**
52	**?**	**?**
53		
54		

■

Wadą przekazywania przez wartość jest konieczność rezerwowania pamięci na parametry i poświęcenie czasu na fizyczne kopiowanie wartości. Ma to znaczenie wówczas, gdy rozmiary parametrów są duże.

15.9 Tablice jednowymiarowe

Typ parametrów podlega jednemu ograniczeniu: nie może być typem tablicowym. Jednak w programach widzimy nagłówki postaci

```
void f(float tab[4], int n)
```

Jakiego więc jest typu parametr `tab`? Okazuje się, że `tab` wskaźnikiem o dziedzinie `float` mimo, że wygląda jak tablica.

> Przekazywanie tablicy w C do parametru zawsze polega na przekazaniu adresu jej początku, elementy nigdy nie są kopiowane.

Ograniczenia dotyczące tablic wynikają z tego, że tablice zwykle są duże i ich kopiowanie zajmuje czas i pamięć. Ideą jeżyka C jest, by był szybki.

Nagłówek funkcji przekazującej do parametru adres początku tablicy może mieć różne postacie:

```
double min(double *tab, int rozm)
double min(double tab[], int rozm)
double min(double tab[10], int rozm)
```

Wszystkie trzy formy są równoważne — **w każdej parametr `tab` jest wskaźnikiem** mimo, że w trzeciej wygląda jak tablica.

Stylistycznie najlepsza jest forma druga:

- nie zawiera zbędnej informacji (liczby 10, która i tak zostanie zlekceważona),
- programiści wiedzą, że parametr `double tab[]` jest wskaźnikiem i dodatkowo widzą, że jego przeznaczeniem jest przechowywanie adresu początku tablicy, a nie adresu jakiejś zmiennej.

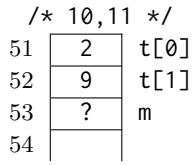
Przykład (Zadanie 34)

Prześledźmy działanie poniższego programu.

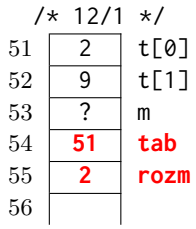
```
double min(double *tab, int rozm) {  /*1*/
   double min_el = tab[0];           /*2*/
   int i;                            /*3*/
   for (i = 1; i < rozm; ++i)        /*4*/
      if (tab[i] < min_el)           /*5*/
         min_el = tab[i];            /*6*/
   return min_el;                    /*7*/
}                                    /*8*/

main(){                              /*9*/
   double t[2] = { 2,9 };            /*10*/
   double m;                         /*11*/
   m = min(t, 2);                    /*12*/
   printf("%lf", m);                 /*13*/
   return 0;                         /*14*/
}                                    /*15*/
```

Program rozpoczyna wykonanie od linii 10, w której jest tworzona tablica `t`.

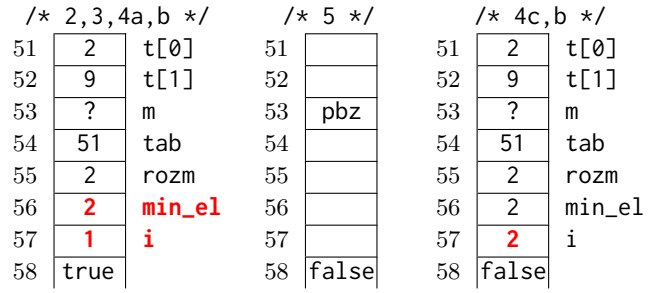


W linii 12 następuje wywołanie funkcji `min`. Zgodnie z konwencją I, nazwa tablicy jest adresem jej zerowego elementu.

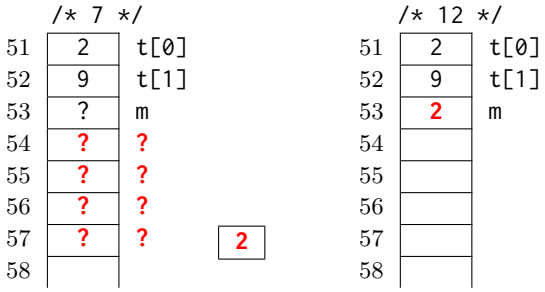


Tablica fizycznie została stworzona w funkcji `main`. Nie będzie istniała żadna jej kopia. Funkcja dostała adres tablicy i znając go, może wykonywać operacje na oryginalnej tablicy z `main`. Oprócz adresu tablicy, konieczne jest przekazanie liczby jej elementów.

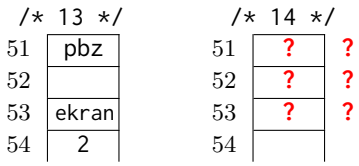
Dalej wykonywana jest treść funkcji `min`. Dzięki konwencji II, parametr `tab` staje się drugą nazwą tablicy `t`.



Instrukcja `return` kończy działanie funkcji i następuje powrót do miejsca wywołania.



Funkcja `main` kończy działanie po wypisaniu komunikatu.



■

15.10 Tablice dwuwymiarowe

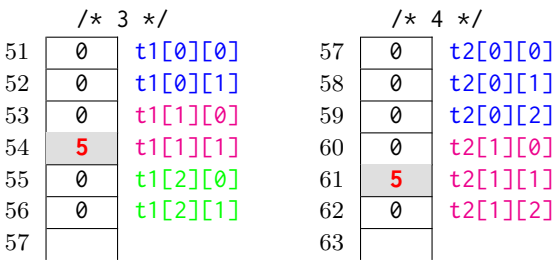
Tak jak w przypadku tablic jednowymiarowych, tablica dwuwymiarowa nie jest kopiowana do parametru, a jedynie przekazujemy do parametru adres początku tablicy. To dzieje się podczas wykonania programu.

Podczas kompilacji potrzebna jest jednak jeszcze jedna informacja o tablicy dwuwymiarowej. Aby zrozumieć dlaczego i jaka to informacja rozważmy następujący przykład.

Przykład

W programie zadeklarowano poniższe tablice i zmieniono jeden element każdej z nich.

```
double t1[3][2] = {{0}};   // 1
double t2[2][3] = {{0}};   // 2
t1[1][1] = 5;              // 3
t2[1][1] = 5;              // 4
```



Odwołanie do komórki o indeksach `[1][1]` polega na:

- przeskoczeniu jednego wiersza tablicy,
- przeskoczeniu jednego elementu tablicy.

Pierwsza czynność wymaga znajomości długości wiersza! W zależności od długości wiersza, trafiamy w inne miejsce względem początku tablicy.

długość wiersza = liczba kolumn

Gdybyśmy napisali nagłówek funkcji tak

```
/**
 * Funkcja pobiera od uzytkownika liczby zmiennopo-
 * zycyjne i umieszcza je w tablicy dwuwymiarowej.
 * @param   t  adres poczatku tablicy
 * @param   m  uzywana liczba wierszy
 * @param   n  uzywana liczba kolumn
 */
void wczytaj_double_2D(double t[][], int m, int n)
// niepoprawne!
```

i w treści funkcji odwołali się do któregoś elementu tablicy, to nie wiadomo by było jaka jest długość wiersza. Odwołania nie dałoby się wykonać. Dlatego taki program nie skompilowałby się.

Kompilatorowi musimy dostarczyć informację o długości wiersza, czyli nagłówek musi być tę informację zawierać.

```
#define N 10 // liczba kolumn w pamieci
...

/**
 * Funkcja pobiera od uzytkownika liczby zmiennopo-
 * zycyjne i umieszcza je w tablicy dwuwymiarowej.
 * @param   t  adres poczatku tablicy
 * @param   m  uzywana liczba wierszy
 * @param   n  uzywana liczba kolumn
 */
void wczytaj_double_2D(double t[][N], int m, int n)
```

Stała `N` przechowuje liczbę kolumn tabeli — chodzi o liczbę kolumn zajmowaną przez tabelę w pamięci, a nie aktualnie używaną, która może być mniejsza i jest podana w parametrze `n`.

Funkcja o powyższym nagłówku obsługuje tablice, które mogą mieć dowolną liczbę wierszy, ale liczba kolumn, które tablica zajmuje w pamięci, musi wynosić `N`. Nie jest więc ta funkcja w pełni ogólna.

Gdybyśmy chcieli napisać funkcję obsługującą tablice dwuwymiarowe dowolnych kształtów, to najwygodniej byłoby posłużyć się tablicami dynamicznymi ze sterty. Wtedy nagłówek wygląda tak.

```
/**
 * Funkcja pobiera od uzytkownika liczby zmiennopo-
 * zycyjne i umieszcza je w dynamicznej tablicy
 * dwuwymiarowej.
 * @param   t  adres tablicy z adresami wierszy
 * @param   m  uzywana liczba wierszy
 * @param   n  uzywana liczba kolumn
 */
void wczytaj_double_2D(double **t, int m, int n)
```

15.11 Użycie zmiennych globalnych jako niezalecany sposób komunikacji między funkcjami

Funkcje mogą zdobyć i przekazać dane przez zmienne globalne. Zmienne globalne są widoczne w każdym miejscu programu począwszy od ich definicji. Oznacza to, że każda funkcja (której definicja znajduje się poniżej definicji zmiennej globalnej) może ją odczytywać i zmieniać.

Przykład Prześledźmy działanie poniższego programu, który jest modyfikacją Zadania 35 i 36. Jest to najgorsza wśród trzech wersji.

```
double pods = 0.2, w;          /*1*/

void pot(unsigned int k)       /*2*/
{
   int i;                      /*3*/
   w = 1.0;                    /*4*/
   for (i = 1; i <= k; ++i)    /*5*/
      w *= pods;               /*6*/
}                              /*7*/

main()
{                              /*8*/
   pot(1);                     /*9*/
   printf("%lf", w);          /*10*/
   return 0;                  /*11*/
}                              /*12*/
```

Przed wykonaniem programy, zmienne globalne są alokowane w obszarze danych. Program zaczyna wykonanie do linii 9, w której od razu jest wywołanie funkcji `pot`.

obszar danych

		/* PWP */		/* 9/2 */	
	51	0.2	pods	0.2	pods
	52	0	w	0	w
stos	53			1	k
	54				
	55				

Następnie wykonywana jest treść funkcji `pot`

	/* 3,4,5a,b */		/* 6,5c,b */	
51	0.2	pods	0.2	pods
52	1	w	0.2	w
53	1	k	1	k
54	1	i	2	i
55	true		false	

Funkcja kończy działanie przez dojście do klamry zamykającej. Nie jest obliczana wartość zwracana, bo funkcja jej nie posiada. Następuje jedynie dealokacja zmiennych lokalnych i parametrów. Po powrocie do miejsca wywołania nic się nie dzieje.

	/* 7 */		/* 9 */	
51	0.2	pods	0.2	pods
52	0.2	w	0.2	w
53	?	?		
54	?	?		
55				

Na koniec program wypisuje na ekran komunikat i kończy działanie.

```
/* 10 */
51 | pbz
52 |
53 | ekran
54 | 0.2
```

```
/* 11 */
51 | 0,2  | pods
52 | 0.2  | w
53 |
54 |
```

■

Używanie zmiennych globalnych jest uważane za zły styl programowania.

Przekazywanie danych przez parametry jest bardziej uniwersalne — łatwo użyć gotową funkcję w innym programie, gdyż nie trzeba pamiętać o przekopiowaniu zmiennych globalnych. Podczas kopiowania zmiennych globalnych może dojść do konfliktu nazw. Używanie zmiennych globalnych utrudnia też czytanie programu i sprzyja powstawaniu trudno wykrywalnych błędów.

15.12 Funkcje i aliasy

Parametry funkcji mogą być aliasami innych zmiennych. Dzieje się tak, np. gdy parametr przekazywany przez wartość jest wskaźnikiem zawierającym adres zmiennej globalnej.

Przykład (Zadanie 38)

Prześledźmy działanie poniższego programu.

```
int x = 4, *s = &x;        /*1*/

void p(int *y, int *z) {   /*2*/
   static int a;           /*3*/
   *y += x;                /*4*/
   z[0] += 3;              /*5*/
   *s += 7;                /*6*/
}                          /*7*/

main()
{                          /*8*/
   p(&x, s);               /*9*/
   printf("%d", x);        /*10*/
}                          /*11*/
```

obszar danych

```
          /* PWP */          /* 9/2 */          /* 4 */
51 |  4 | x      51 |  4 | x      51 |  8 | x
52 | 51 | s      52 | 51 | s      52 | 51 | s
53 |  0 | a      53 |  0 | a      53 |  0 | a
54 |    |        54 | 51 | y      54 | 51 | y
55 |    |        55 | 51 | z      55 | 51 | z
56 |    |        56 |    |        56 |    |
```

stos

```
          /* 5 */            /* 6 */            /* 7 */
51 | 11 | x      51 | 18 | x      51 | 18 | x
52 | 51 | s      52 | 51 | s      52 | 51 | s
53 |  0 | a      53 |  0 | a      53 |  0 | a
54 | 51 | y      54 | 51 | y      54 |  ? | ?
55 | 51 | z      55 | 51 | z      55 |  ? | ?
56 |    |        56 |    |        56 |    |
```

```
/* 9 */
51 | 18 | x
52 | 51 | s
53 |  0 | a
54 |    |
```

```
/* 10 */
51 | pbz
52 |
53 | ekran
54 | 18
```

```
/* 11 */
51 | 18 | x
52 | 51 | s
53 |  0 | a
54 |    |
```

W treści funkcji p nazwy x, *y, y[0], *z, z[0], *s oraz s[0] są aliasami. ■

15.13 Komentarze do funkcji

Funkcje powinny być opatrzone komentarzami. Część programistów uważa, że komentarze powinny być przy deklaracjach, część, że przy definicjach, ale nigdy nie należy dublować komentarza. Niektórzy przy deklaracji umieszczają komentarz dla osób, które miałyby używać tą funkcję, a przy definicji inny komentarz, informujący o rozwiązaniach technicznych (dla osób, które miałyby poprawiać/rozbudowywać funkcję).

Komentarz powinien być na tyle dokładny, że osoba, która nie pisała funkcji, czytając go, powinna umieć poprawnie ją wywołać (bez zaglądania do definicji funkcji). Koniecznie muszą być zawarte informacje:

1. jakie zadanie wykonuje funkcja,
2. do czego służą parametry funkcji,
3. co oznacza wartość zwracana.

Istnieje bardzo dużo różnych stylów pisania komentarzy do funkcji. Poniży współpracuje z programem Doxygen automatycznie generującym dokumentację.

```
/** Pobiera od uzytkownika liczbe calkowita i
  * zwraca ja. Przeprowadza kontrole poprawnosci
  * danych. Pozwala porawic bledne dane.
  * @return pobrana liczba
  */
int wczytaj_int();

/** Pobiera od uzytkownika ustalona ilosc liczb
  * calkowitych i umieszcza je w tablicy.
  * Przeprowadza kontrole poprawnosci danych.
  * Pozwala porawic bledne dane.
  * @param tab    adres poczatku tablicy
  * @param n      ilosc liczb
  */
void wczytaj_int_1W(int tab[], int n);
```

Zachęcam do wykonania zadań 41 – 46