Funkcje

- Funkcja jest wydzielonym fragmentem kodu programu, spełniającym określone zadanie.
- Funkcja może zarówno wykonywać czynności (np. wyświetlenie danych na ekranie), jak i zwracać wartości (np. obliczenie długości łańcucha znakowego).
- Funkcje pozwalają one uniknąć powtarzania tych samych fragmentów kodu.

```
/* naglowekl.c */
#include <stdio.h>
#define NAZWA "MEGATHINK, INC."
#define ADRES "10 Megabuck Plaża"
#define MIEJSCOWOŚĆ "Megapolis, CA 94904"
#define GRANICA 65
void gwiazdki(void); // prototyp funkcji
int main(void)
  {
  gwiazdki();        // wywołanie funkcji
  printf("%s\n", NAZWA);
  printf("%s\n", ADRES);
  printf("%s\n", MIEJSCOWOŚĆ);
  gwiazdki();        // wywołanie funkcji
  return 0;
  }

void gwiazdki(void)     // definicja funkcji
 {
 int licznik;
 for (licznik = 1; licznik <= GRANICA;licznik++)
     putchar('*');
 putchar('\n');
 }
```

wynik działania programu:

```
*****************************************************
MEGATHINK, INC.
10 Megabuck Plaza
Megapolis, CA 94904
*****************************************************
```

```c
/* naglowek2.c */
#include <stdio.h>
#include <string.h> // zawiera prototyp strlen()
#define NAZWA "MEGATHINK, INC."
#define ADRES "10 Megabuck Plaza"
#define MIEJSCOWOŚĆ "Megapolis, CA 94904"
#define GRANICA 65
#define ODSTĘP ' '

void n_znak(char ch, int num);

int main(void)
  {
  int odstępy;

  n_znak('*', GRANICA);  // stale jako argumenty
  putchar('\n');

  // program oblicza, ile odstępów należy wyświetlić
  odstępy = (65 - strlen(NAZWA)) / 2;
  n_znak(ODSTĘP, odstępy); //zmienna jako argument
  printf("%s\n", NAZWA);

  odstępy = (65 - strlen(ADRES)) / 2;
  n_znak(ODSTĘP, odstępy); //zmienna jako argument
  printf("%s\n", ADRES);
  n_znak(ODSTĘP, (65 - strlen(MIEJSCOWOŚĆ)) / 2);
                        // wyrażenie jako argument
  printf("%s\n", MIEJSCOWOŚĆ);
  n_znak('*', GRANICA);
  putchar('\n');
  return 0;
  }

/* definicja funkcji n_znak() */
void n_znak(char ch, int num)
  {
  int licznik; // zmienna lokalna
  for (licznik = 1; licznik <= num; licznik++)
                                    putchar(ch);
  }
```

wynik działania programu:

```
**********************************************************
                    MEGATHINK, INC.
                  10 Megabuck Plaza
                  Megapolis, CA 94904
**********************************************************
```

Definiowanie funkcji pobierającej argument: *argumenty formalne*

Definicja funkcji rozpoczyna się poniższą deklaracją:

```
void n_znak(char ch, int num)
```

która informuje kompilator, że funkcja `n_znak()` pobiera dwa argumenty o nazwach `ch` i `num`, że ch należy do typu `char` oraz że `num` należy do typu `int`.

Zmienne `ch` i `num` nazywamy ***argumentami formalnymi***.

Podobnie jak zmienne zadeklarowane wewnątrz funkcji, argumenty formalne są zmiennymi lokalnymi, stanowiącymi prywatną własność funkcji. Oznacza to, że w innych funkcjach mogą istnieć niezależne zmienne o tych samych nazwach.

Zmienne `ch` i `num` otrzymują wartości przy każdym wywołaniu funkcji `n_znak()`.

Składnia C wymaga, aby każda zmienna była poprzedzona nazwą swojego typu (nie wolno stosować list zmiennych należących do tego samego typu).

```
void ping(int x, y, z)          // nieprawidłowy nagłówek funkcji
void pong(int x, int y, int z)  // prawidłowy nagłówek funkcji
```

Prototyp funkcji pobierającej argumenty

Funkcję `n_znak()` zadeklarowaliśmy przy pomocy następującego prototypu :

```
void n_znak(char ch, int num);  // nagłówek z definicji zakończony średnikiem
```

Prototyp określa liczbę i typy argumentów przyjmowanych przez funkcję. Argumenty rozdzielone są przecinkami. Nazwy zmiennych w prototypie mogą zostać pominięte:

```
void n_znak(char, int);
```

Użycie konstrukcji `char ch` i `int num` w prototypie nie powoduje utworzenia żadnych zmiennych.

Wywoływanie funkcji pobierającej argumenty: *argumenty faktyczne*

Zmienne `ch` i `num` otrzymują swoje wartości dzięki użyciu w wywołaniu funkcji ***argumentów faktycznych*** (argumentów aktualnych).

```
n_znak(ODSTĘP, 25);
```

Argumentami faktycznymi są tu znak odstępu oraz liczba 25. Wartości te zostają przypisane odpowiadającym im argumentom formalnym funkcji `n_znak()`, czyli zmiennym `ch` i `num`.

Argument formalny jest zmienną lokalną w wywoływanej funkcji, a ***argument faktyczny*** konkretną wartością przypisaną tej zmiennej przez funkcję wywołującą.

Argument faktyczny może być stałą, zmienną lub wyrażeniem. W każdym przypadku jest on obliczany, a jego wartość zostaje skopiowana do argumentu formalnego funkcji.

Zwracanie wartości przy pomocy instrukcji *return*

```
/* mniejsze.c – znajduje mniejsze zlo */
#include <stdio.h>
int imin(int,   int);
int main(void)
  {
  int zlol, zlo2;
  printf("Podaj dwie liczby całkowite (q kończy program):\n");
  while (scanf("%d %d", &zlol, &zlo2) == 2)
    {
    printf("Mniejsza liczba spośród %d i %d jest %d.\n", zlol, zlo2, imin(zlol, zlo2)) ;
    printf("Podaj dwie liczby całkowite (q kończy program):\n");
    }
  return 0;
  }

int imin(int n, int m)
  {
  int min;
  if (n < m) min = n;
  else       min = m;
  return min;
  }
```

wynik działania programu:

```
Podaj dwie liczby całkowite (q kończy program):
509  333
Mniejsza liczba spośród 509 i 333 jest 333.
Podaj dwie liczby całkowite (q kończy program):
-9393  6
Mniejsza liczba spośród -9393 i 6 jest -9393.
Podaj dwie liczby całkowite (q kończy program):
q
```

Słowo kluczowe ***return*** sprawia, że wartość następującego po nim wyrażenia staje się wartością zwracaną funkcji.
W tym przypadku funkcja zwraca wartość zmiennej `min`.

Ponieważ zmienna `min` należy do typu `int`, do tego typu należy również funkcja `imin()`.

Zmienna `min` jest wprawdzie prywatna dla funkcji `imin()`, ale jej wartość jest przekazywana do funkcji wywołującej za pośrednictwem słowa ***return***.

Efektem poniższej instrukcji jest więc nadanie zmiennej `mniejsze` wartości zmiennej `min`:

```
mniejsze = imin(n,m);
```

Zwrócona wartość może zostać nie tylko przypisana zmiennej, ale także użyta w wyrażeniu:

```
odpowiedz = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + odpowiedz, GRANICA));
```

Wartość zwracana funkcji może być dostarczona przez dowolne wyrażenie, nie tylko zwykłą zmienną.

```
/* funkcja wartości minimalnej, druga wersja */
imin(int n, int m)
  {
  return (n < m)? n : m;
  }
```

Instrukcja ***return*** powoduje zakończenie funkcji i przejście do kolejnej instrukcji w funkcji wywołującej nawet jeśli *return* nie jest ostatnią instrukcją w funkcji.

```
/* funkcja wartości minimalnej, trzecia wersja */
imin(int n, int m)
  {
  if (n < m)return n;
  else      return m;
  }

/* funkcja wartości minimalnej, czwarta wersja */
imin(int n, int m)
  {
  if (n < m)return n;
  else      return m;
  // poniższa instrukcja nigdy nie będzie wykonana
  printf("Profesor Fleppard to kretyn.\n");
  }
```

Instrukcja:

```
return;
```

powoduje zakończenie funkcji i powrót do funkcji wywołującej.

Ponieważ po słowie ***return*** nie znajduje się żadne wyrażenie, nie ma również wartości zwracanej - forma ta powinna być więc stosowana tylko w funkcji typu `void`.

Typy funkcji

Deklaracja funkcji musi zawierać jej typ.

Funkcja powinna należeć do tego samego typu, co zwracana przez nią wartość. Funkcja, która nie zwraca wartości, powinna należeć do typu `void`.

Jeśli w deklaracji funkcji nie podano typu, język C zakłada, że funkcja należy do typu `int`.

Deklaracja typu jest częścią definicji funkcji i odnosi się do wartości zwracanej, a nie do argumentów.

Aby móc poprawnie korzystać z funkcji, program musi znać jej typ, zanim zostanie ona użyta po raz pierwszy.
Można to osiągnąć przez:

- umieszczenie pełnej definicji funkcji przed miejscem jej pierwszego wywołania (takie rozwiązanie nie może być stosowane w przypadku, gdy funkcja jest częścią biblioteki lub znajduje się w osobnym pliku).
- deklarowanie (wstawienie prototypu) przed miejscem jej pierwszego wywołania.

```
#include <stdio.h>
// deklaracja dla wszystkich
// funkcji od tego miejsca
int imin(int, int);
int main(void)
  {
  int zlol, zlo2;
  . . .
```

lub

```
#include <stdio.h>
int main(void)
  {
  // deklaracja tylko dla
  // funkcji main()
  int imin(int, int);
  int zlol, zlo2;
  . . .
```

W standardowej bibliotece C funkcje pogrupowane są w rodziny, z których każda posiada swój ***plik nagłówkowy***.

Plik nagłówkowy zawiera między innymi deklaracje funkcji.

Np. plik `stdio.h` zawiera deklaracje standardowych funkcji wejścia/wyjścia, takich jak `printf()` i `scanf()`,

plik `math.h` - deklaracje funkcji matematycznych, takich jak `sqrt()`.

Dołączenie pliku `math.h` (przy pomocy dyrektywy `#include`) zawierającego deklarację:

```
double sqrt(double);
```

informuje jedynie kompilator, że funkcja `sqrt()` zwraca typ `double` i ma jeden argument typu `double`, ale sam kod funkcji `sqrt()` znajduje się w oddzielnym pliku bibliotecznym.

Rekurencja

Język C pozwala, aby funkcja wywoływała samą siebie. Proces ten nosi nazwę **rekurencji** (ang. *recursion*).

```
/* rsilnia.c – oblicza silnie za pomocą rekurencji */
#include <stdio.h>
long rsilnia(int n);
int main(void)
  {
  int num;
  printf("Podaj liczbę z przedziału 0-15  (q kończy program):\n");
  while  (scanf("%d",  &num)  ==1)
    {
    if  (num < 0) printf("Żadnych liczb ujemnych proszę!\n");
    else if (num > 15) printf("Wartość nie może przekraczać 15.\n");
    else printf("%d silnia = %ld\n", num, rsilnia(num));
    printf("Podaj liczbę z przedziału 0-15 (q kończy program):\n");
    }
  return 0;
  }

long rsilnia(int n)   // funkcja rekurencyjna
  {
  long odp;
  if (n > 0) odp = n * rsilnia(n-1); // n! = n * (n-1)!
  else odp = 1;                      // 0! = 1
  return odp;
  }
```

```c
/* binar.c – wyświetla liczbę całkowita w postaci dwójkowej */
#include <stdio.h>

void do_binar(int n);

int main(void)
  {
  int liczba;
  printf("Podaj liczbę całkowita (q kończy program):\n");
  while (scanf("%d", &liczba) = 1)
    {
    printf("Odpowiednik dwójkowy: "); do_binar(liczba) ; putchar('\n');
    printf("Podaj liczbę całkowita (q kończy program):\n");
    }
  return 0;
  }

void do_binar(int n)  // funkcja rekurencyjna
  {
  int r;
  r = n % 2;
  if (n >= 2) do_binar(n / 2);
  putchar('0' + r); return;
  }
```

przykładowy przebieg działania programu:

Podaj liczbę całkowita (q kończy program):
9
Odpowiednik dwójkowy: 1001
Podaj liczbę całkowita (q kończy program):
255
Odpowiednik dwójkowy: 11111111
Podaj liczbę całkowita (q kończy program):
1024
Odpowiednik dwójkowy: 10000000000
Podaj liczbę całkowita {q kończy program):
q

Korzystanie z plików nagłówkowych

```
/* opłaty.c – program obliczający oplate za pokój */
#include <stdio.h>
#include "hotel.h" // definiuje stale, deklaruje funkcje
int' main (void)
  {
  int noce;
  double hotel;
  int kod;
  while ((kod = menu()) != QUIT)
    {
    switch(kod)
      {
      case 1 : hotel = HOTELI; break;
      case 2 : hotel = H0TEL2; break;
      case 3 : hotel = H0TEL3; break;
      case 4 : hotel = HOTEL4; break;
      default: hotel = 0.0;printf("Ups!\n"); break;
      }
    noce = pobierz_noce();
    pokaz_cene(hotel, noce);
    }
  return 0;
  }
```

```
/* hotel.c – funkcje dla zarządzających hotelami */
#include <stdio.h>
#include "hotel.h"

int menu(void)
  {
  int kod, stan;
  printf("\n%s%s\n", GWIAZDKI, GWIAZDKI};
  printf("Podaj numer hotelu:\n"};
  printf("1) Marek Antoniusz        2) Olimpijski\n");
  printf("3) U Marynarza            4) Savoy\n");
  printf("5) koniec\n"};
  printf("%s%s\n", GWIAZDKI, GWIAZDKI);
  while ((stan = scanf("%d", &kod)) != 1
                          || (kod < 1 || kod > 5))
    {
    if (stan != 1) scanf("%*s");
    printf("Podaj liczbę z przedziału od 1 do 5.\n");
    }
  return kod;
  }
```

```
int pobierz_noce(void)
  {
  int noce;
  printf("Ile nocy będzie potrzebne? ");
  while (scanf("%d", &noce) != 1)
    {
    scanf("%*s");
    printf("Podaj liczbę całkowita, np. 2.\n");
    }
  return noce;
  }

void pokaz_cene(double hotel, int noce)
  {
  int n;
  double suma = 0.0;
  double przelicznik = 1.0;
  for (n = 1; n <= noce; n++, przelicznik *= RABAT)
    suma += hotel * przelicznik;
  printf("Całkowity koszt pobytu wyniesie %0.2f S.\n",
           suma);
  }
```

```
/* hotel.h – stale i deklaracje dla hotel.c */
#define KONIEC 5
#define HOTEL1    50.00
#define HOTEL2    55.00
#define HOTEL3    80.00
#define HOTEL4    100.00
#define RABAT     0.95
#define GWIAZDKI "**********************************"
int menu(void);
int pobierz_noce(void);
void pokaz_cene(double, int);
```

przykładowy przebieg działania programu:

```
**********************************************
Podaj numer hotelu:
1) Marek Antoniusz      2) Olimpijski
3) U Marynarza          4) Savoy
5) koniec
**********************************************
3
Ile nocy będzie potrzebne? 1
Całkowity koszt pobytu wyniesie 80.00 $.
**********************************************
Podaj numer hotelu:
1) Marek Antoniusz      2) Olimpijski
3) U Marynarza          4) Savoy
5) koniec
**********************************************
5
```

Wskaźniki

Wskaźnik (ang. *pointer*) jest zmienną (lub, bardziej ogólnie, obiektem danych), której wartość jest adresem w pamięci.

Uzyskiwanie adresów: *operator &*

Jednoargumentowy operator **&** pozwala uzyskać adres, pod którym przechowywana jest zmienna.
Jeśli `ach` jest nazwą zmiennej, to `&ach` jest jej adresem.

Jeśli zmienna wskaźnikowa nosi nazwę `wsk`, to:

```
wsk = &ach;     // przypisuje zmiennej  wsk  adres zmiennej  ach
```

Mówimy, że `wsk` „wskazuje na" `ach`.

Różnica między `wsk` a `&ach` polega na tym, iż `wsk` jest zmienną, a `&ach` stałą.

Zmienna `wsk` może wskazywać gdzie indziej:

```
wsk = &och;     // Teraz wartością  wsk  jest adres zmiennej  och
```

Operator dereferencji: *

Jeśli `wsk` wskazuje na ach:

```
wsk  =  &ach;
```

Operator dereferencji ***** (ang. dereference) znajduje wartość przechowywaną przez zmienną `ach`.

```
wart  = *wsk;    // znajduje wartość,  na która wskazuje  wsk
```

```
{wsk  =  &ach;   wart  =  *wsk;}   ⇔   wart  =  ach;
```

Deklarowanie wskaźników

```
int *pi;         // pi jest wskaźnikiem do zmiennej całkowitej
char *pc;        // pc jest wskaźnikiem do zmiennej znakowej
float *pf, *pg;  // pf i pg sa wskaźnikami do zmiennych typu float
```

Wykorzystanie wskaźników do komunikacji pomiędzy funkcjami

```
/* zamien3.c – zamiana warosci zmiennych z wykorzystaniem wskaźników */
#include <stdio.h>

void zamiana(int *u,  int *v); // wskaźniki jako argumenty funkcji

int main(void)
  {
  int x = 5, y = 10;

  printf("Początkowo x = %d, a y = %d.\n", x, y);

  zamiana(&x, &y);  // wyslanie adresów do funkcji

  printf("A teraz x = %d, a y = %d.\n", x, y);
  return 0;
  }

void zamiana(int *u, int *v) // wskaźniki jako argumenty funkcji
  {
  int temp;
  temp = *u;   // temp otrzymuje wartość, na która wskazuje u
  *u = *v;
  *v = temp;
  }
```

Tablice

Tablica składa się z ciągu elementów należących do jednego typu.

Utworzenie tablicy odbywa się za pośrednictwem deklaracji.

Deklaracja tablicy informuje kompilator o liczbie i typie elementów w tablicy.

Elementy tablicy mogą należeć do takich samych typów, jak zwykłe zmienne.

```
/* parę deklaracji tablic */
int main(void)
  {
  float cukierki[365]; // tablica 365 wartości float
  char kod[12];        // tablica 12 znaków
  int stany[50];       // tablica 50 liczb całkowitych
  . . .
  }
```

Nawiasy kwadratowe (**[ ]**) sygnalizują, że cukierki, kod i stany są tablicami; wartości w nawiasach określają liczbę ich elementów.

Aby uzyskać dostęp do poszczególnych elementów tablicy, wskazujemy je korzystając z ***indeksu***.

Numeracja rozpoczyna się od zera, stąd cukierki [0] jest pierwszym elementem tablicy cukierki, a cukierki [364] - elementem 365-tym i ostatnim.

Zmienne i tablice automatyczne

Zmienna lub ***tablica automatyczna*** jest zmienną lub tablicą zadeklarowaną wewnątrz funkcji (obejmuje to także argumenty formalne).

Zmienna zadeklarowana w funkcji jest prywatna dla tej funkcji, nawet jeśli inna funkcja w programie zawiera zmienną o tej samej nazwie.

Zmienna zadeklarowana w funkcji istnieje tylko w czasie działania funkcji. W momencie, gdy funkcja kończy działanie, a program powraca do funkcji wywołującej, pamięć przeznaczona na *zmienną automatyczną* jest zwalniana.

```
int main(void)
  {
  int potęgi[8]  =  {1,2,4,8,16,32,64,128};
  . . .
  }
```

Zmienne i tablice zewnętrzne

Zmienna lub ***tablica zewnętrzna*** jest zmienną lub tablicą zadeklarowaną poza jakąkolwiek funkcją.

```
int raport;                        // zmienna zewnetrzna
int wilki[5] = {12, 10, 8, 9, 6};  // tablica zewnetrzna
int jedzenie(int n);               // prototyp
int main(void)
{
. . .
}
int jedzenie(int n)
{
. . .
}
```

Zmienne i tablice zewnętrzne:

- są znane wszystkim funkcjom, które następują po ich deklaracji w pliku źródłowym.
- istnieją przez cały czas działania programu
- są standardowo inicjalizowane wartością 0 (są „zerowane”).

Zmienne i tablice statyczne

Zmienną lub tablicę statyczną definiujemy wewnątrz funkcji korzystając ze słowa kluczowego **static**:

```
int konto(int n, int m)
{
static int fasola[2] = {343, 332};
. . .
}
```

Powyższa definicja tworzy tablicę lokalną dla funkcji konto (), ale

- zachowującą swoją wartość między wywołaniami funkcji,
- standardowo wyzerowaną.

Więcej o inicjalizacji tablic

```
#define MIESIĄCE 12
int main(void)
{
int dni[MIESIACE] = {31,28,31,30,31,30,31,31,30,31,30,31};
. . .
}
```

```
/* brak_dan.c – niezainicjalizowane tablice */
#include <stdio.h>
#define ROZMIAR 4
int zewntab[ROZMIAR]; // niezainicjalizowana tablica zewnętrzna
int main(void)
{
static int statab[ROZMIAR]; // niezainicjalizowana tablica statyczna
int autab[ROZMIAR];         // niezainicjalizowana tablica automatyczna
int i;
printf("%2s%10s%10s%10s\n", "i",  "zewntab",  "statab",  "autab");
for  (i = 0;  i < ROZMIAR;  i++) printf("%2d%10d%10d%10d\n", i, zewntab[i],statab[i], autab[i]);
return 0;
}
```

wynik działania programu:

i	zewntab	statab	autab
0	0	0	3
1	0	0	6618612
2	0	0	4206088
3	0	0	6618628

```
/* troch_dan.c – częściowo zainicjalizowane tablice */
#include <stdio.h>
#define ROZMIAR 4
int zewntab[ROZMIAR] = {1956,   1966};   // inicjalizacja częściowa
int main(void)
{
static int statab[ROZMIAR] = (-50, -90}; // inicjalizacja częściowa
int autab [ROZMIAR] ='{492, 567};        // inicjalizacja częściowa
. . .
}
```

dane wyjściowe:

i	zewntab	statab	autab
0	1956	-50	492
1	1966	-90	567
2	0	0	0
3	0	0	0

W przypadku inicjalizacji częściowej kompilator przypisuje on pozostałym elementom wartość 0. Zachowanie takie ma miejsce w przypadku tablic wszystkich klas.

```
/* dni_m2.c – liczba elementów jest obliczana przez kompilator */
#include <stdio.h>
int main(void)
{
int dni[] = {31,28,31,30,31,30,31,31,30,31};
int index;
for (index = 0; index < sizeof dni / sizeof(int); index++) // (!!!)
printf("Miesiąc %d ma %d dni.\n", index +1, dni[index]);
return 0;
}
```

Przypisywanie wartości do tablic

Przypisanie wartości elementom tablicy odbywa się za pośrednictwem indeksu element po elemencie.

```
/* przypisywanie wartości do tablicy */
#include <stdio.h>
#define ROZMIAR 50
int main(void)
  {
  int licznik, parzyste[ROZMIAR];
  for (licznik = 0; licznik < ROZMIAR; licznik++) parzyste[licznik] = 2 * licznik;
  . . .
  }
```

C nie pozwala na przypisywanie tablic w całości.
Nie wolno również korzystać z listy wartości, stosowanej przy inicjalizacji.

```
/* nieprawidłowe przypisania */
#define ROZMIAR 5
int main(void)
  {
  int byki[ROZMIAR] = {5,3,2,8};     // w porządku
  int jaki[ROZMIAR];
  jaki = byki;                       // niedozwolone
  jaki[ROZMIAR] = byki[ROZMIAR];     // nieprawidłowe
  jaki[ROZMIAR] = {5,3,2,8};         // niedopuszczalne
  . . .
  }
```

Wskaźniki do tablic

Nazwa tablicy jest równocześnie adresem jej pierwszego elementu.

```
domek == &domek[0]        *domek == domek[0]     // jeśli domek jest tablicą
```

własności:

```
domek + 1 == &domek[1]    *(domek + 1) == domek[1]

domek + i == &domek[i]    *(domek + i) == domek[i]  // dla i typu int
```

```
float tab[10];  // deklaracja tablicy (może być innego, dowolnego typu)
float *wsk;     // deklaracja wskaźnika (tego samego typu co wyzej)
wsk = &tab[3];  // przypisanie wskaźnikowi adresu 4-tego elementu tablicy
// . . . od tego miejsca prawdziwe są wyrażenia:
wsk[0] == tab[3]
wsk[1] == tab[4]
wsk[2] == tab[5]
// itd.
```

Funkcje, tablice i wskaźniki

```
/* suma_tl.c – sumuje elementy tablicy */
#include <stdio.h>
#define ROZMIAR 10

long sumuj(int tab[], int n);
// long sumuj(int *tab, int n);  // wersja alternatywna

int main(void)
  {
  int kulki[ROZMIAR] = {20,10,5,39,4,16,19,26,31,20};
  long wynik;
  wynik = sumuj(kulki, ROZMIAR);
  printf("Całkowita liczba kulek wynosi %ld.\n", wynik);
  return 0;
  }

long sumuj(int tab[], int n)
// long sumuj(int *tab, int n)  // wersja alternatywna
  {
  int i;
  long suma = 0;
  for(i = 0;  i < n; i++) suma += tab[i];
  return suma;
  }
```

```c
/* suma_t2.c – sumuje elementy tablicy - wersja wskaznikowa */
#include <stdio.h>
#define ROZMIAR 10

long sumujw(int *początek, int *koniec);

int main(void)
  {
  int kulki[ROZMIAR] = {20,10,5,39,4,16,19,26,31,20};
  long wynik;
  wynik = sumujw(kulki, kulki + ROZMIAR);
  printf("Całkowita liczba kulek wynosi %ld.\n", wynik);
  return 0;
  }

long sumujw(int *początek, int *koniec) // wersja wskaznikowa
  {
  long suma = 0;
  while (początek < koniec)
    {
    suma += *poczatek;   // dodaje wartość do sumy
    poczatek++;          // przenosi wskaźnik do następnego elementu
    }
  return suma;
  }
```

Działania na wskaźnikach

Język **C** udostępnia pięć podstawowych operacji, jakie można wykonywać na wskaźnikach:

- ***Przypisanie***. Wskaźnikowi można przypisać adres. Zwykle odbywa się to przez podanie nazwy tablicy lub za pomocą operatora adresu (**&**).
- ***Pobranie wartości (dereferencja).*** Wartość przechowywaną we wskazywanym miejscu pamięci otrzymujemy za pomocą operatora *****.
- ***Uzyskanie adresu wskaźnika***. Tak jak wszystkie zmienne, zmienne wskaźnikowe mają adres i wartość. O miejscu przechowywania wskaźnika informuje operator **&**.
- ***Zwiększenie wskaźnika***.[1] Można je uzyskać przez zwykłe dodawanie lub za pomocą operatora inkrementacji. Zwiększenie wskaźnika do elementu tablicy sprawia iż wskazuje on kolejny element.
- ***Odejmowanie***.[1] Możliwe jest znalezienie różnicy między dwoma wskaźnikami. Działanie to jest wykonywane zwykle na wskaźnikach do elementów tej samej tablicy w celu określenia, jak daleko od siebie się znajdują. Wynik jest wyrażony w jednostce o rozmiarze typu. Odejmowanie jest prawidłową operacją, o ile oba wskaźniki wskazują na elementy tej samej tablicy.

[1] Istnieje kilka rzeczy, o których należy pamiętać przy zwiększaniu lub zmniejszaniu wskaźnika:

- komputer nie sprawdza, czy po wykonaniu jednej z tych operacji wskaźnik nadal wskazuje na element tablicy.
- operatory inkrementacji i dekrementacji można stosować do zmiennych wskaźnikowych, ale nie do stałych adresowych, takich jak nazwy tablic.

Ochrona zawartości tablicy - zastosowanie słowa kluczowego *const* w parametrach formalnych

Pisząc funkcję przetwarzającą jeden z podstawowych typów danych, takich jak int, mamy wybór między przekazaniem wskaźnika do zmiennej lub jej wartości.

Powszechnie stosowana zasada każe przekazywać wartość zmiennej, chyba że występuje konieczność zmiany tej wartości - w takiej sytuacji przekazujemy wskaźnik.

W przypadku tablic nie mamy wyboru: musimy przekazać wskaźnik (kwestia wydajności).

Aby uniemożliwić zmianę zawartości tablicy przez funkcję, w deklaracji argumentu formalnego wystarczy użyć słowa kluczowego **const**.

```
long sumuj(const int tab[], int n);  // prototyp

long sumuj(const int tab[], int n)   // definicja
{
int i;
long suma = 0;
for( i = 0; i < n; i++) suma += tab[i]++; // <- tu kompilator wykryje błąd!
return suma;
}
```

Podobnie, w deklaracji tablic lokalnych:

```
#define MIESIĄCE 12
const int dni[MIESIĄCE] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Każda próba zmiany wartości elementu takiej tablicy pociągnie za sobą błąd kompilacji.

Możliwe jest również tworzenie wskaźników do wartości stałych:

```
double opłaty[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double *wd = opłaty;        // wd wskazuje na początek tablicy
```

Drugi wiersz stwierdza, że `wd` wskazuje na wartość typu `const double`. Oznacza to, że `wd` nie może posłużyć do zmiany wskazywanych wartości:

```
*wd = 29.89;          // niedozwolone
wd[2] = 222.22;       // niedozwolone
opłaty[0] = 99.99;    // dozwolone, ponieważ tablica opłaty nie jest stałą
```

… ale można zmienić wartość wskaźnika `wd` tak, aby wskazywał on gdzie indziej – sam wskaźnik nie jest stałą.

```
wd ++; // sprawia,  że wd wskazuje na opłaty[1] - dozwolone!
```

Wskaźnik do stałej jest zwykle wykorzystywany jako parametr funkcji, aby zaznaczyć, że funkcja nie będzie zmieniać danych w funkcji wywołującej.

Wskaźnik do stałej może otrzymać adres zarówno danych stałych, jak i zmiennych, natomiast zwykły wskaźnik może przechowywać tylko adres danych zmiennych:

```
double opłaty[5] = {88.99, 100.12, 59.45, 183.11,340.5};
const double zablok[4] = {0.08, 0.075, 0.0725, 0.07};
const double *cwd = opłaty;  // dozwolone
double *wd = opłaty;         // dozwolone
cwd = zablok;                // dozwolone
cwd = &oplaty[3];            // dozwolone
wd = zablok;                 // niedozwolone
wd = &oplaty[3];             // dozwolone
```

Można na przykład zadeklarować i zainicjalizować wskaźnik tak, aby nie mógł on wskazywać na żadne inne miejsce w pamięci. Wskaźnik taki może służyć do zmiany wartości, nie można jednak przypisać mu żadnego adresu innego niż początkowy.

```
double oplaty[5] = {88.99, 100.12, 59.45, 183.11, 340.5];
double *const wd = opłaty;
wd = &oplaty[2];    // niedozwolone
*wd = 92.99;        // w porządku - zmienia wartość opłaty[0]
```

Słowa const można użyć dwukrotnie w jednej deklaracji, aby utworzyć wskaźnik, który nie pozwala zmienić ani wskazywanego miejsca, ani przechowywanej w nim wartości:

```
double oplaty[5] = {88.99, 100.12, 59.45, 183.11, 340.5];
const double *const wd = opłaty;
wd = &oplaty[2];    // niedozwolone
*wd = 92.99;        // verboten!
```

Tablice wielowymiarowe

W języku C tablice wielowymiarowe są tablicami tablic, np.

```
float deszcz[5][12];   // tablica 5 tablic 12 liczb typu float
int woda[3][4][5];     // tablica 3 tablic 4 tablic 5 liczb typu int
```

Inicjalizacja tablicy dwuwymiarowej

```
float deszcz[LATA][MIESIĄCE] = {{10.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.3, 6.1, 7.4},
                                {9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 5.2},
                                {6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 4.2},
                                {4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
                                {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.2}};
```

Wskaźniki a tablice wielowymiarowe

```
int zippo[4][2];     /* tablica tablic typu int */
```

- `zippo` jest adresem pierwszego elementu tablicy `=> zippo == &zippo[0]`
- `zippo[0]` jest tablicą składającą się z dwóch liczb całkowitych `=> zippo[0] == &zippo[0][0]`
- Dereferencja wskaźnika lub adresu (zastosowanie operatora * lub nawiasów kwadratowych z indeksem) daje w wyniku wartość wskazywanego obiektu.

```
zippo[0] == &zippo[0][0]  ➔  *(zippo [0]) == zippo[0][0]
zippo == &zippo[0] == &zippo[0][0] ➔  *zippo == &zippo[0][0]
**zippo  == *(*zippo) ==  *&zippo[0][0] == zippo[0][0]
```

Każdy element tablicy `zippo` jest tablicą, a tym samym adresem pierwszego elementu:

```
zippo[0] ==  &zippo[0][0] == *zippo
zippo[1] ==  &zippo[1][0] == *(zippo + 1)
zippo[2] ==  &zippo[2][0] == *(zippo + 2)
zippo[3] ==  &zippo[3][0] == *(zippo + 3)
```

Zastosowanie operatora * do wszystkich wyrażeń daje następujący wynik:

```
*zippo[0] ==  zippo[0][0] == **zippo
*zippo[1] ==  zippo[1][0] == *(*(zippo + 1))
*zippo[2] ==  zippo[2][0] == *(*(zippo + 2))
*zippo[3] ==  zippo[3][0] == *(*(zippo + 3))
```

Ogólnie, notacja tablicowa przekłada się na zapis wskaźnikowy zgodnie z poniższym wzorem:

```
zippo[m][n]  == *(*(zippo + m) + n)
```

Uwaga na deklaracje !

```
int (*wz)[2]; // deklaracja wskaźnika do 2-elementowej tablicy typu int
int *pax[2];  // deklaracja 2-elementowej tablicy wskaźników do typu int
```

Funkcje a tablice wielowymiarowe

```
/* podwtabl.c – podwaja elementy tablicy */
#include <stdio.h>
void podw(int tab[], int rozmiar); // prototyp

int main(void)
  {
  int śmieci[3][4] = {{2,4,5,8},{3,5,6,9},{12,10,8,6}};
  int i,  j;
  for (i = 0; i < 3; i++) podw(śmieci[i], 4); // przekazanie jednego wiersza tablicy
  for (i = 0; i < 3; i++)
    {
    for (j = 0; j < 4; j++) printf("%5d", śmieci[i][j]); putchar('\n');
    }
  return 0;
  }

void podw(int tab[], int rozmiar)  // lub: int *tab, …
  {
  int i;
  for (i = 0; i < rozmiar; i++) tab[i] *= 2;
  }
```

```c
/* podwtab3.c – podwaja elementy tablicy */
#include <stdio.h>
void podw2 (int tab[][4], int rozmiar); // prototyp

int main(void)
  {
  static int śmieci[3][4] = {{2,4,5,8}, {3,5,6,9}, {12,10,8,6}};
  int i, j;
  podw2(śmieci, 3);
  for  (i = 0;  i < 3;  i++)
    {
    for (j = 0; j < 4; j
    printf("%5d", śmieci[i][j]); putchar('\n');
    }
  return 0;
  }

void podw2(int tab[][4], int rozmiar)  // lub int (*tab)[4]
  {
  int i, j;
  for (i = 0; i < rozmiar; i++) for (j = 0; j < 4; j++) tab[i][j] *= 2;
  }
```

□