

# Podstawy Programowania

## Wykład nr 6: Funkcje

dr hab. inż. Dariusz Dereniowski

Katedra Algorytmów i Modelowania Systemów  
Wydział ETI, Politechnika Gdańska

## Dlaczego funkcje są potrzebne?

- Podczas pisania programu często zachodzi potrzeba wielokrotnego wykonywania tych samych obliczeń (tzn. wykonywania tych sekwencji instrukcji) w różnych sytuacjach (zazwyczaj dla różnych danych). Przykładami mogą być: wypisanie komunikatu informującego o stanie programu, znalezienie najmniejszego elementu w tablicy, posortowanie tablicy, wyznaczenie rozwiązań równania kwadratowego.
- W obrębie funkcji można zamknąć operacje wykonywane dla zadanych parametrów wejściowych, np. ciąg liczb, współczynniki równania kwadratowego  $a$ ,  $b$ ,  $c$ .
- W praktyce każdy (większy) program jest zbudowany z wielu funkcji (dobry styl programowania, wiele krótkich i czytelnych funkcji).
  - poprawia to przejrzystość i czytelność programu,
  - zapewnia oszczędność czasu (funkcja raz napisana może być wielokrotnie wykorzystana),
  - ułatwia dokonywanie zmian w programie: jeśli daną, wielokrotnie powtarzaną czynność należy zmodyfikować, to czynimy tą modyfikację w jednym tylko miejscu, gdy jest ona wyodrębniona jako funkcja,
  - pozwala zaoszczędzić pamięć: funkcja jest umieszczona w jednym miejscu w pamięci podczas wykonywania programu.

## Deklaracja a definicja funkcji

- **Deklaracja funkcji:** informuje kompilator o tym, że określona funkcja może (ale nie musi) zostać użyta w programie. Deklaracja zawiera informację o nazwie funkcji, typie zwracanej wartości oraz typach argumentów. Deklaracja nie dostarcza kodu funkcji.

```
int min( int a, int b);
```

- **Definicja funkcji:** informuje o tym co funkcja robi. W odróżnieniu od deklaracji, definicja skutkuje przydzieleniem przez kompilator pamięci na kod funkcji.

```
int min( int a, int b) {  
    if ( a < b ) return a;  
    else return b;  
}
```

## Przykład użycia funkcji – program obliczający współczynnik dwumianu

- Zadanie polega na napisaniu programu obliczającego wartość  $\binom{n}{k}$  dla podanych przez użytkownika wartości  $n$  i  $k$ , gdzie  $n \geq k$ .
- Postanawiamy skorzystać ze wzoru:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

- Obserwujemy, że program będzie wymagał trzykrotnego obliczenia silni dla różnych argumentów.
- Aby nie pisać trzykrotnie identycznego kodu (a jedynie operującego na innych zmiennych), decydujemy się na napisanie funkcji obliczającej silnię z zadanego argumentu.
- Silnia nawet dla niewielkich argumentów przyjmuje znaczące wartości (np.  $20! = 2432902008176640000$ ), więc aby zwiększyć liczbę argumentów, dla których nie nastąpi efekt przepełnienia typu, wartości zwracane przez naszą funkcję będą typu `long int`.

## Przykład użycia funkcji – program obliczający współczynnik dwumianu

```
#include <iostream>
using namespace std;

long long silnia( int );

/* Program obliczający współczynnik dwumianowy z liczb
   n i k podanych przez użytkownika.
   Uwaga: nawet dla małych wartości n może nastąpić przepełnienie. */
int main() {
    int n, k;
    cin >> n;
    cin >> k;
    cout << silnia(n)/( silnia(k)*silnia(n-k) ) << endl;
    return 0;
}

long long silnia( int n ) {
    long long s = 1;

    for (int i=1; i<=n; i++)
        s *= i;
    return s;
}
```

## Przykład użycia funkcji – zadanie domowe

Napisz program obliczający wartość  $e^x$ , gdzie  $e$  jest liczbą Eulera ( $e \approx 2.718$ ) a  $x$  jest liczbą rzeczywistą.

- Język C nie posiada operatora potęgowania w swoim asortymencie.
- W celu obliczenia  $e^x$  moglibyśmy użyć odpowiedniej funkcji matematycznej z biblioteki `math.h`. Elementem dobrego stylu programowania jest korzystanie z gotowych funkcji bibliotecznych gdy tylko jest to możliwe. W tym wypadku jednak świadomie z tego zrezygnujemy.
- Nie możemy obliczyć wartości  $e^x$  mnożąc  $e$  przez siebie  $x$  razy, gdyż  $x$  nie musi być liczbą całkowitą. Z pomocą przychodzi nam wzór:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Oczywiście, obliczanie nieskończonej sumy nie jest możliwe, więc ograniczymy się do przybliżenia wartości  $e^x$  poprzez sumowanie kilku początkowych wartości powyższego szeregu, np.

$$e^x \approx \sum_{n=0}^7 \frac{x^n}{n!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!}.$$

## Przykład użycia funkcji – zadanie domowe

Zanim zaczniemy pisać program, warto dokonać szeregu przemyśleń, które ułatwią kodowanie.

- Nie wiemy ile początkowych wyrazów szeregu powinniśmy brać pod uwagę (8 we wcześniejszym wzorze). Nasz plan jest taki, aby sprawdzić to eksperymentalnie. Aby program był łatwy do modyfikacji, spróbujemy liczbę wyrazów określać za pomocą stałej. Na początku programu umieścimy więc dyrektywę

```
#define LICZBA_WYRAZOW 8
```

i dalej w kodzie posługujemy się tym identyfikatorem zamiast stałą liczbową.

W konsekwencji, jeśli postanowimy, że program powinien uwzględnić początkowych 10 wyrazów, zmienimy to zmieniający 8 na 10 tylko w dyrektywie. Powinna to być jedyna zmiana wymagana w tym przypadku!

- Zauważamy, że pewne obliczenia się powtarzają: obliczanie  $n!$  oraz  $x^n$ . Napiszmy więc funkcje, które wykonują te obliczenia. Jakie typy powinny mieć parametry i zwracane wartości?
- Samo obliczenie  $e^x$  też warto wyodrębnić jako osobną funkcję pobierającą na wejściu  $x$  (typu `double`) oraz zwracającą wartość  $e^x$  (również typu `double`).
- Uwaga dla zainteresowanych: program napisany w ten sposób jest podatny na błędy przepiętnie i zaokrągleń. Metody jego udoskonalania wykraczają poza zakres tego wykładu.

## Zasięg zmiennych

- Obiekt zdefiniowany na zewnątrz wszystkich funkcji ma **zasięg globalny**.
  - Zmienne globalne oraz zmienne z modyfikatorem **static** umieszczane są w normalnym obszarze pamięci (static), który przed uruchomieniem programu jest zerowany, a wartości zmiennych są zachowywane pomiędzy kolejnymi uruchomieniami funkcji.
- **Zmienne lokalne** funkcji (oraz **argumenty** przekazywane do funkcji) są zmiennymi **automatycznymi** (auto).
  - W momencie, gdy kończymy blok, w którym zmienne lokalne zostały powołane do życia, automatycznie przestają one istnieć.
  - Obiekty automatyczne komputer przechowuje na stosie.
  - Należy pamiętać, że zmienne automatyczne nie są zerowane w chwili utworzenia.
  - Jeśli nie zainicjowaliśmy ich jakąś wartością, to przechowują one **wartości przypadkowe**.

```
#include <iostream>
using namespace std;

int x;
void f( void ) {
    int i;
}

int main() {
    int a = 1;
    {
        int b = 2;
        cout << x; // 0 (globalne x)
    }
    cout << x; // 0 (globalne x)
    {
        int b = 3;
        int x = 4;
        int a = 5;
        {
            int d = 6;
            int x = 7;
            cout << a; // 5
            cout << b; // 3
            cout << x; // 7 (def. int x=7)
        } // przestania obie poprzednie
        cout << b; // 3
        cout << a; // 5
        cout << x; // 4
        cout << d; // Błąd kompilacji!
    }
    return 0;
}
```



## Do jakich zmiennych ma dostęp funkcja w języku C?

- Z poziomu funkcji uzyskujemy **jawny dostęp** do zmiennych globalnych, zmiennych lokalnych oraz argumentów bieżącego wykonania funkcji (**argumenty są traktowane tak samo, jak zmienne lokalne!**)
- Niejawny dostęp następuje wówczas, gdy funkcja posiada adres pamięci (wskaźnik) innej zmiennej. Dzięki temu, możliwa jest zmiana wartości obszaru pamięci pod tym adresem, a przez to zmiana wartości zmiennej tam umieszczonej.
- Nie istnieje żadna forma ochrony pamięci: korzystając ze wskaźników można odczytać bądź nadpisać dowolny obszar pamięci.
  - działania zamierzone: modyfikacja tablic, modyfikacja zmiennych lokalnych innych funkcji (poprzez wskaźniki do nich)
  - błędne użycie może prowadzić do niezamierzonego działania programu: uszkodzenie stosu wykonania, wkroczenie na „segment kodu” lub odwoływanie się do obszaru pamięci, który nie należy do programu (reakcja na taką sytuację może zależeć od systemu operacyjnego).
- Funkcja nie posiada jawnego dostępu do zmiennych lokalnych innych funkcji.
- Przesłanianie deklaracji zmiennych: w różnych blokach kodu można deklarować zmienne o tej samej nazwie. Odwołanie do zmiennej poprzez jej nazwę odnosi się do zmiennej o tej nazwie zadeklarowanej w najbardziej wewnętrznym bloku — patrz poprzedni przykład.

## Inicjalizacja zmiennych i zmienne static – przykład

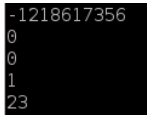
```
#include <iostream>
using namespace std;

int x;

/* Funkcja zwraca liczbę całkowitą, która jest równa liczbie wywołań
funkcji od czasu uruchomienia programu.
Uwaga: licznik nie jest zabezpieczony przed przepiętnieniem. */
int f( void ) {
    static int licz_wywołania; // zmienna ta istnieje i zachowuje swoją
    // wartość pomiędzy kolejnymi wywołaniami funkcji
    return ++licz_wywołania;
}

int main() {
    int a;
    static int b;

    cout << a << endl; // na ekran zostanie wypisana losowa wartość
    cout << b << endl; // wypisuje 0 (zmienne static inicjowane są zerami)
    cout << x << endl; // wypisuje 0 (zmienne globalne inicjowane są zerami)
    cout << f() << endl;
    f();
    a = f() + f()*f(); // Uwaga: tutaj mamy do czynienia z efektem ubocznym:
    // wiadomo, że 3 powyższe wywołania funkcji f() zwrócą wartości 3,4,5,
    // lecz nie wiadomo które wywołanie zwróci którą wartość!!
    cout << a << endl;
    return 0;
}
```



```
-1218617356
0
0
1
23
```

## Przekazywanie parametrów

- W języku C występuje **tylko** tzw. przekazywanie parametrów przez wartość, które polega na tym, że każdy argument funkcji jest kopią przekazywanego parametru.
- W przypadku tablicy identyfikator jest adresem tej tablicy. Jeśli więc argumentem funkcji jest tablica, to funkcja otrzymuje kopię adresu przekazywanej tablicy, a nie kopię całej tablicy. Stąd efekt polegający na tym, że zmiana wartości elementu tablicy wewnątrz funkcji powoduje zmianę wartości tablicy przekazanej do funkcji – jest to ta sama tablica (patrz przykład obok).

```
#include <iostream>
using namespace std;
```

```
void f( int x ) {
    x = 1;
}
```

*/\* Funkcja modyfikuje wartość x[1]. Uwaga: programista musi zadbać, aby przekazana na wejście tablica miała rozmiar co najmniej 2 \*/*

```
void g( int x[] ) {
    x[1] = 1;
}
```

```
int main() {
    int x=0, t[] = { 0, 0, 0, 0, 0, 0, 0, 0 };

```

```
    f( x );
    cout << x << endl; // wypisujemy 0

```

```
    g( t );
    // ponizej wypisujemy 0,1,0,...
    for ( int i=0; i < (sizeof t)/(sizeof t[0]); i++ )
        cout << t[i] << ' ';
    cout << endl;
    return 0;
}
```

```
0
0,1,0,0,0,0,0,0,
```

## Zwracanie wartości

- Słowo kluczowe **return** z parametrem, odpowiadającym typowi zwracanemu określonego dla funkcji, powoduje zakończenie wykonania funkcji i zwrócenie wartości do funkcji wołającej.
- Typ wyrażenie będącego parametrem instrukcji **return** musi umożliwiać rzutowanie do typu zwracanego przez funkcję.
- Wartość zwracana jest zawsze kopiowana (podobnie, jak przy przekazywaniu argumentów).
- Operacja return powinna się wykonać dla każdej możliwej ścieżki wykonania funkcji w przypadku funkcji zwracającej typ inny niż **void**.
- W funkcji o typie zwracanym **void**, instrukcja **return** jest opcjonalna. W takich przypadkach funkcja kończy się wraz z zakończeniem realizacji jej bloku instrukcji.

## Biblioteka standardowa – ważniejsze pliki nagłówkowe

- **assert.h** – diagnozowanie programów
- **ctype.h** – klasyfikacja znaków
- **errno.h** – zmienne przechowujące informacje o błędach
- **math.h** – podstawowe funkcje matematyczne
- **signal.h** – obsługa zdarzeń wyjątkowych
- **stdio.h** – obsługa wejścia/wyjścia
- **stdlib.h** – funkcje ogólnego przeznaczenia, włączając alokację pamięci, konwersje, i inne
- **string.h** – funkcje manipulacji napisami
- **time.h** – obsługa daty i czasu

## Przykład 1: wyszukanie zadanego elementu w tablicy

Załóżmy, że chcemy napisać funkcję, która stwierdza czy w podanej tablicy znajduje się zadany element. Podejmujemy szereg decyzji na etapie projektowania takiego fragmentu kodu:

- Funkcja taka mogłaby zwracać wartości **0** oraz **1** wskazujące na to czy element znajduje się w tablicy czy nie. Aby nasza funkcja była bardziej ogólna i mogła być przydatna w szerszej gamie sytuacji, postanawiamy, że będzie ona zwracać numer indeksu, pod którym znajduje się szukany element.
- Musimy mieć możliwość zakomunikowania w sytuacji braku szukanego elementu – wówczas nasza funkcja zwróci **-1**.
- Wśród argumentów funkcji będzie oczywiście tablica (nazwijmy ten argument **tablica**) i poszukiwany element (który nazwiemy **element**). Programista piszący funkcję musi jednak znać rozmiar tablicy. Mamy tutaj dwie główne opcje do wyboru:
  - rozmiar jest zadeklarowany na etapie pisania programu, tzn. deklaracja funkcji byłaby następująca: `int szukaj( int tablica[ROZMIAR], int element );`, gdzie **ROZMIAR** jest etykietą zdefiniowaną przez dyrektywę `#define`. Wada: można używać funkcji tylko do tablic takiego rozmiaru (w programie możemy mieć tablice różnych długości).
  - rozmiar jest również parametrem funkcji. Wówczas deklaracja mogłaby wyglądać tak: `int szukaj( int tablica[], int rozmiar, int element );`  
To rozwiązanie nie ma poprzedniej wady, więc je wybieramy.
- Funkcję opatrujemy stosownym komentarzem, który informuje co funkcja robi i jakie jest znaczenie poszczególnych parametrów.

## Przykład 1: wyszukanie zadanego elementu w tablicy

```
#include <iostream>
using namespace std;

/* Funkcja szuka liczby o wartości element w tablicy t.
   Parametry:
   1) tablica
   2) n – liczba elementów tablicy t
   3) szukany element
   Zwracana wartość: liczba i,  $0 \leq i < n$ , taka że  $t[i] == \text{element}$ 
   lub -1 jeśli takie i nie istnieje (tzn. gdy liczby element
   nie ma w tablicy t */
int szukaj( const int t[], int n, int element ) {
    for ( int i=0; i < n; i++ ) {
        if ( t[i] == element )
            return i;
    }
    return -1;
}

// W celu przetestowania powyższej funkcji piszemy prosty program testowy:
int main() {
    int t[] = { 0, 0, 0, 4, 0, 9, 0, 4, 5, 3, 4, 5, 6 }, podany;

    cout << "Podaj element, który chcesz znaleźć: ";
    cin >> podany;
    cout << szukaj( t, (sizeof t)/(sizeof t[0]), podany ) << endl;
    return 0;
}
```

## Przykład 2: porządkowanie liczb w tablicy

Założmy, że chcemy napisać funkcję, która porządkuje (niemalejąco) elementy podanej na wejściu tablicy. Przed napisaniem kodu dokonujemy szeregu przemyśleń:

- Rozpoczniemy od zastanowienia się w jaki sposób takie zadanie należy wykonać (ta kwestia była dość naturalna w poprzednim przypadku, więc jej wprost nie poruszyliśmy): jeśli tablica ma  $n$  elementów, to jeśli potrafilibyśmy znaleźć największy z nich i zamienić ten największy z ostatnim (czyli  $n$ -tym) elementem w tablicy, to ten największy element będzie już na swoim właściwym miejscu. Wówczas, wśród pozostałych  $n-1$  elementów znajdziemy największy i umieścimy go na przedostatnim ( $(n-1)$ -szym miejscu) itd.
- Z powyższego wnioskujemy, że potrzebne są nam następujące fragmenty kodu:
  - Funkcja wyszukująca największy element (będzie ona zwracać indeks największego elementu, a na wejściu pobierać tablicę oraz liczbę jej elementów). Przyjmijmy następującą jej deklarację: `int najwiekszy( int t[], int rozmiar );`
  - Funkcja zamieniająca elementy tablicy o wskazanych indeksach. Funkcja nie musi niczego zwracać. Na wejście funkcja dostanie tablicę oraz indeksy elementów do zamiany. Zauważmy, że rozmiar tablicy nie jest w tej funkcji do niczego potrzebny. Deklaracja: `void zamien(int t[], int indeks1, int indeks2 );`
  - Funkcja główna używająca powyższych do porządkowania tablicy. Funkcja nie zwraca wartości, a na wejściu otrzyma tablicę oraz jej rozmiar. Mamy więc deklarację: `void porzadkuj_tablice( int t[], int n );`



## Przykład 2: porządkowanie liczb w tablicy

```
#include <iostream>
using namespace std;

int najwiekszy( int t[], int rozmiar ) {    // Funkcja zwraca indeks
    int indeks = 0, i;                     // w tablicy t, pod którym
    for ( i=1; i < rozmiar; i++ )          // znajduje się jej
        if ( t[i] > t[indeks] )             // największy element.
            indeks = i;
    return indeks;
}

void zamien(int t[], int indeks1, int indeks2 ) {
    int pomocnicza = t[indeks1];           // Funkcja dokonuje zamiany elementów
    t[indeks1] = t[indeks2];               // pod indeksami indeks1 i indeks2
    t[indeks2] = pomocnicza;               // w tablicy t
}

void porzadkuj_tablice( int t[], int n ) { // Funkcja porządkuje
    for ( int i=n-1; i >0; i-- )           // niemalejąco liczby
        zamien( t, i, najwiekszy( t, i+1 ) ); // w talicy t o rozmiarze n
}

int main() { // Dołączamy prosty program testowy.
    int t[] = { 0, 41, 0, 4, 0, 9, 0, 4, 5, 3, 4, 5 , 6, 0, -3, -3, -1, 20 };

    porzadkuj_tablice( t, (sizeof t)/(sizeof t[0]) );
    for ( int i=0; i < (sizeof t)/(sizeof t[0]); i++ )
        cout << t[i] << endl;
    return 0;
}
```

D.Dereniowski (KAiMS,PG)