

Poprzedni wykład:

- Co to jest algorytm? (*Przykłady*)
- Przekształcenia typów i rzutowanie
- Operatory zwiększania (++) i zmniejszania (--)
- **Operatory bitowe**
- Operatory przypisania (+= -= *= /= %= <<= >>= ...)
- **Operator „?:”**

Elementarz formatowanego wejścia/wyjścia

Używając funkcji **printf** do wypisywania wartości zmiennych, np.

```
fahr = 32 + (9.0/5) * celc;  
printf("%3.0f %6.1f\n", celc, fahr);
```

podajemy tzw. **specyfikacje przekształcenia**, w tym wypadku:

%3.0f oznacza, że liczba zmiennopozycyjna (**celc**) zostanie wypisana w postaci co najmniej 3 znaków, bez kropki dziesiętnej i części ułamkowej; zaś **%6.1f** oznacza, że kolejna taka liczba (**fahr**) ma zająć co najmniej 6 znaków i zawierać 1 cyfrę po kropce.

Wyniki:

```
0    32.0  
10   50.0  
...
```

Przykładowo, dla następujących **specyfikacji przekształcenia**, argument zostanie wypisany jako:

- `%d` liczba całkowita w zapisie dziesiętnym;
- `%6d` j.w., co najmniej 6 znaków;
- `%o` liczba całkowita w zapisie ósemkowym;
- `%x` liczba całkowita w zapisie szesnastkowym;
- `%f` liczba zmiennopozycyjna;
- `%.2f` liczba zmiennopozycyjna z 2 miejscami po kropce;
- `%6.2f` liczba zmiennopozycyjna z 2 miejscami po kropce, zajmująca co najmniej 6 znaków;
- `%12.6e` liczba zmiennopozycyjna w zapisie wykładniczym, co najmniej 12 znaków, 6 cyfr po kropce.

Ponadto: `%c` — jeden znak, `%s` — napis, zaś `%%` — wypisze znak „%”.

Sterowanie

Instrukcje i bloki. Wyrażenie, np. `i=0` czy `i++` albo `printf(...)` staje się *instrukcją*, jeśli jest zakończone średnikiem:

```
i = 0; i++; printf(...);
```

średnik w C to **ogranicznik instrukcji** (**nie operator!**);

w C istnieje zatem instrukcja pusta:

```
;
```

Nawiasy klamrowe `{` oraz `}` są używane do **grupowania** deklaracji i instrukcji w **bloki**; blok to inaczej instrukcja złożona, składniowo równoważna jednej instrukcji.

Zmienne (*automatyczne!*) można deklarować wewnątrz **dowolnego** bloku.

[**Po nawiasie zamykającym blok „}” nie piszemy średnika!**]

Instrukcja warunkowa (*if-else*)

Instrukcja `if-else` służy **podjęwaniu decyzji**:

```
if (wyrażenie)          /* można też: „if (wyrażenie != 0)” */
    instrukcja1
else
    instrukcja2
```

Część `else` jest opcjonalna, a w razie niejednoznaczności:

```
if (n > 0)
    if (a > b) z = a;
    else z = b;
```

=> „else” jest automatycznie przyporządkowane **najbliższej** instrukcji „if” **niezawierającej** „else” (!) [*Za dużo { } nie przeszkadza ...*]

Decyzje wielowariantowe zapisywane są za pomocą ciągu:

```
if (wyrażenie1)
    instrukcja
else if (wyrażenie2)
    instrukcja
else if (wyrażenie3)
    instrukcja
...
else
    instrukcja
```

Wartości wyrażeń są kolejno obliczane, ***pierwsze wyrażenie prawdziwe (tzn. != 0)*** powoduje wykonanie odpowiedniej instrukcji i zakończenie wykonywania całej konstrukcji.

Instrukcja *switch*

Niestrukuralny substytut konstrukcji *else-if*, który nadaje się do zastosowania w przypadku, gdy wystarczy nam porównywanie pewnego **wyrażenia** z wartościami stałych całkowitych (lub **wyrażeń stałych o wartościach całkowitych**):

```
switch (wyrażenie) {  
    case wyrażenie-stałe: instrukcje /* po if, else-if było ... */  
    case wyrażenie-stałe: instrukcje /* ... „instrukcja” =>JEDNA (!) */  
    default: instrukcje  
}
```

[**Wartości „wyrażeń-stałych” muszą być RÓŻNE.**]

Wykonywanie instrukcji rozpoczyna się od pierwszego przypadku (case) dla którego **wyrażenie == wyrażenie-stałe**; instrukcje można pominąć.

W instrukcji „switch” **przypadki znaczą tyle co etykiety**, a zatem dalsze instrukcje wykonywane są po kolei, o ile jawnie nie wykonamy akcji przerywającej (zwykle: instrukcja *break* lub *return*). **Przykład:**

```
    /* Zliczamy poszczególne cyfry */
while ((c = getchar()) != EOF) {
    switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        default:
            break;    /* Ten break można pominąć - ALE    */
    }                /* ucierpi przejrzystość programu */
}
}
```


Pętle: *while* oraz *for*

>> *Nie ma programu bez błędów pętli ...* <<

W przypadku pętli:

while (*wyrażenie*)

instrukcja

/ Ponownie: JEDNA instrukcja (!) */*

najpierw obliczane jest *wyrażenie*; jeśli wartość jest **różna od zera** — wykonuje się *instrukcję*, następnie **ponownie** oblicza się *wyrażenie*, itd. [*Cykl powtarza się aż do chwili, gdy wartość wyrażenia stanie się zerowa* — wówczas sterowanie przechodzi do *instrukcji po pętli.*]

UWAGA: Jeśli na początku mamy: *wyrażenie == 0*, instrukcja wewnątrz pętli *while* nie zostanie wykonana **ani razu** (!)

Przykład: podstawiamy wartości 0, 1, 2, ... n-1 do tablicy `int`:

```
i=0;
while (i<n) {
    tab[i] = i;
    i++;
}
```

Inny przykład — pętla nieskończona z instrukcją pustą:

```
while (1)
    ;
```

(*Pętla jak wyżej **nigdy** nie zakończy swojego działania!*)

W praktyce: po `while (1)` zwykle mamy blok `{ ... }` zawierający `break` lub `return`.

Zamiast pętli postaci:

```
wyrażenie1;  
while ( wyrażenie2 ) {  
    instrukcja  
    wyrażenie3;  
}
```

można użyć formy krótszej:

```
for ( wyrażenie1; wyrażenie2; wyrażenie3 )  
    instrukcja
```

[Powyżej zakładamy, że w bloku *instrukcja* nie użyto *continue*;
w przeciwnym przypadku — obie formy różnią się **jednym**
wykonaniem instrukcji *wyrażenie3;*]

Z punktu widzenia **gramatyki** języka C — wszystkie trzy składniki w nawiasie po `for` są dowolnymi wyrażeniami; najczęściej *wyrażenie1* i *wyrażenie3* to **przypisania** (*wyrażenie3* często wykorzystuje operatory `++`, `--`) zaś *wyrażenie2* jest **wyrażeniem warunkowym**.

W przykładzie z inicjowaniem tablicy liczbami 0, 1, ... , n-1:

```
for (i=0; i<n; i++)
    tab[i]=i;
```

[*Ale można też tak:* `for (i=0; i<n; tab[i]=i,i++);`]

Każdy z elementów: *wyrażenie1* ... *wyrażenie3* może zostać pominięty; **musi jednak pozostać średnik!**

Jeśli pominiemy *wyrażenie2* przyjmuje się, że jest ono prawdziwe, a zatem: `for (;;) { ... }` — jest **pętlą nieskończoną**.

ZAGADKA: *Co robi ten program?*

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

```
int main()
```

```
{
```

```
    int i,n=10;
```

```
    for (i=0; i<n; printf("%d\n",i),i++)
```

```
        return 0;
```

```
}
```

Odpowiedź: NIC! — Po nawiasach () pętli for powinna być *instrukcja* (pojedyncza — zakończona średnikiem, lub *blokowa* w nawiasach { }) W tym przypadku, *instrukcja* to: `return 0;`

Program wypisujący liczby 0, 1, ..., 9:

```
int main()
{
    int i,n=10;

    for (i=0; i<n; printf("%d\n",i),i++);
    return 0;
}
```

[Po nawiasie) występuje ***instrukcja pusta!***]

Wybór pętli (*while* czy *for* ?) w dużym stopniu zależy będzie od osobistych upodobań; jeśli nie mamy ani ***części inicjującej*** ani modyfikującej — bardziej naturalne wydaje się użycie *while*, np:

```
while ((c=getchar())==' ' || c=='\n' || c=='\t')
    ; /* pomijamy białe znaki */
```

zamiast

```
for (; (c=getchar())==' ' || c=='\n' || c=='\t'; );
```

Z kolei — jeśli występują proste instrukcje inicjujące i przyrostu, bardziej naturalne będzie użycie *for*, np:

```
for (i=0; i<n; i++) {...}
```

Do ***dobrego stylu programowania*** nie należy „wpychanie” do nawiasów () *for*-a obliczeń niemających nic wspólnego z inicjowaniem ani przyrostem, lepiej umieścić je w bloku {...}

Dalsze przykłady: *Sortowanie tablic*

Pętla *for* — w odróżnieniu od *while* — często umożliwia **skupienie instrukcji sterujących** w jednym miejscu. Jest to szczególnie korzystne w przypadku kilku zagnieżdżonych pętli.

Funkcja `shellsort` porządkuje tablicę liczb całkowitych metodą podaną przez D.L. Shell'a w 1959 roku. [=> **Aby zostać zapamiętanym, warto wymyślać algorytmy dopasowane do nazwiska ...**]

W tej metodzie, w fazach początkowych porównuje się elementy oddalone od siebie, a nie sąsiadujące (jak np. w prostej **metodzie bąbelkowej**). Celem jest szybkie wyeliminowanie „dużego nieporządku”, aby w późniejszych fazach było mniej do zrobienia.

Dla porównania — zaczniemy od prostego algorytmu sortowania „bąbelkowego”:


```

/* A function implementing the bubble-sort */
void bubblesort(int arr[], int n)
{
    int i, j, temp;

    for (i = 0; i < n-1; i++)
        /* Last i elements are already in place */
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
/* https://www.geeksforgeeks.org/bubble-sort/ & AR */

```

W funkcji `bubblesort` występują dwie zagnieżdżone pętle:

W pierwszej z nich, indeks `i` biegnie przez wszystkie elementy tablicy `arr`, zaś pętla **druga** porównuje każdą parę najbliższych sąsiadów w zbiorze `n-i` pierwszych elementów tablicy, i — w razie potrzeby — dokonuje ich zamiany.

Przykładowo, po pierwszym przebiegu pętli wewnętrznej, ***największy element tablicy*** na pewno znajdzie się na miejscu `n-1`.

W drugim przebiegu, największy ***spośród pozostałych*** (`n-1`) — elementów zostanie umieszczony na miejscu (`n-2`) - gim, itd., aż do pełnego uporządkowania elementów tablicy rosnąco (a ściślej: ***niemalejąco***).

```
/* shellsort:  porządkuje v[0] ... v[n-1] rosnąco */  
void shellsort(int v[], int n)  
{  
    int gap, i, j, temp;  
  
    for (gap=n/2; gap>0; gap/=2)  
        for (i=gap; i<n; i++)  
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {  
                temp = v[j];  
                v[j] = v[j+gap];  
                v[j+gap] = temp;  
            }  
}
```

Operator przecinkowy

Ciekawym operatorem języka C, często spotykanym w pętlach for, jest przecinek `,` (*inaczej: **operator przecinkowy***). W ciągu:

wyr1, wyr2, wyr3, ..., wyrN

wyrażenia obliczane są od lewej do prawej; jako typ i wartość wyniku przyjmowane są typ i wartość *wyrN*.

*(Jednak przecinki oddzielające argumenty funkcji, lub np. nazwy zmiennych w deklaracji **nie są operatorami**; w takich przypadkach nie ma gwarancji, że obliczenia będą wykonywane od lewej do prawej.)*

W pętli for możemy użyć operatorów przecinkowych do sterowania równolegle kilkoma indeksami.

Dalej — funkcja `reverse(s)` **odwraca kolejność znaków** w napisie `s`:

```
#include <string.h>

/* reverse(s) : odwróć napis s */
void reverse(char s[])
{
    int c, i, j;    /* to nie operatory! */

    for (i=0, j=strlen(s)-1; i<j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Pętla *do-while*

W przypadku pętli `while` oraz `for`, *warunek zatrzymania* jest sprawdzany na początku — a zatem instrukcja wewnątrz pętli może nie zostać wykonana ani razu.

W przypadku pętli ***do-while*** jest inaczej; pętla zawsze wykona się co najmniej jeden raz. Składnia pętli jest następująca:

`do`

`instrukcja`

`while (wyrażenie);`

[Najpierw wykonuje się ***instrukcję***, a potem oblicza ***wyrażenie***.]

Funkcje w C: Wprowadzenie

W języku C funkcje pozwalają dzielić duże programy na mniejsze podprogramy, jak również korzystać z tego, co zostało już zrobione przez innych (np. *funkcji bibliotecznych*).

Co ciekawe: w C — formalnie rzecz ujmując — nie ma „procedur”, zastępują je funkcje typu `void` — nie zwracające żadnej wartości.

Program napisany w **dobrym stylu** powinien składać się z jak największej liczby *krótkich* funkcji, których działanie łatwo zrozumieć i przetestować niezależnie; później — funkcji można używać jak „czarnych skrzynek”, nie dbając o to, jak zostały zrealizowane.

Dobrze zaprojektowane funkcje pozwalają zignorować to, jak zadanie zostało wykonane, wystarczy jedynie wiedzieć, co będzie zrobione.

Mechanizm definiowania funkcji w C ilustruje przykład podnoszenia liczby *zmiennopozycyjnej* do potęgi *całkowitej*:

```
#include <stdio.h>

double power(double x, int n);

int main()
{
    int j;
    for (j=-10; j<=10; j++)
        printf("%d  %g\n", j, power(2.0,j));
    return 0;
}    /* c.d.n. */
```



```
/* power:  podnieś x do potęgi n */
double power(double x, int n)
{
    double p=1.0;
    int j;

    if (n<0)
        p = 1.0/power(x,-n);  /* rekurencja! */
    else
        for (j=1; j<=n; j++)
            p *= x;
    return p;    /* zwracanie wartości */
}
```

Ogólnie, **definicja funkcji** w C ma postać:

```
typ-zwracanej-wartości nazwa-funkcji(  
    deklaracje parametrów, jeśli występują)  
  
{  
    deklaracje  
    instrukcje  
}
```

Definicje mogą pojawiać się w dowolnej kolejności (także w różnych plikach źródłowych); funkcji **nie można zagnieżdżać**.

[*Typ-zwracanej-wartości void oznacza, że funkcja nie zwraca żadnej wartości użytecznej: return; zwróci sterowanie do miejsca wywołania.*]

Deklaracja występująca przed funkcją **main()**:

```
double power(double x, int n);
```

to tzw. **prototyp funkcji**; nazwy parametrów można pominąć, pisząc po prostu:

```
double power(double, int);
```

Nazwy parametrów mogą być też inne niż w definicji funkcji, musi się jednak zgadzać ich **liczba i typy** (podobnie jak **typ funkcji**).

Nazwy te są przy tym całkowicie **lokalne**; inne funkcje mogą zatem bez problemów używać parametrów o tych samych nazwach.

[*Dla zmiennych zewnętrznych wymyślamy nazwy **oryginane**; dla zmiennych lokalnych — już niekoniecznie.*]

Chociaż nazwy parametrów funkcji w jej prototypie można pominąć, często ułatwią one zrozumienie, co funkcja robi, bez czytania definicji.

Funkcja `power` **jest wywołana** w wierszu funkcji `main` :

```
printf("%d %g\n", j, power(2.0,j));
```

Wywołanie przekazuje do funkcji *dwa argumenty* i sterowanie; po zakończeniu działania funkcja wraca do miejsca wywołania wraz z liczbą (*wartością zwracaną*) przeznaczoną do sformatowania i wypisania na standardowe wyjście.

[**Nazewnictwo:** *parametr* — zmienna w nawiasach okrągłych w definicji funkcji; *argument* — wartość używana przy wywołaniu funkcji.]

Obliczona wartość (przechowywana w *zmiennej automatycznej p*) jest przekazywana do `main` instrukcją: `return p;`

Ogólnie — po słowie kluczowym `return` można umieścić dowolne wyrażenie:

```
return wyrażenie;
```

Jeśli pominąć *wyrażenie* — samo `return;` zwróci do miejsca wywołania funkcji jedynie sterowanie, bez przekazywania wartości. (Podobnie dzieje się przy „*przekroczeniu końca*” funkcji, tj. kiedy sterowanie dotrze do zamykającego nawiasu `})`)

Argumenty funkcji: *Przekazywanie przez wartość*

Specyficzną cechą języka C jest to, że wszystkie argumenty funkcji są przekazywane „*przez wartość*”: oznacza to, że funkcja zamiast swoich argumentów otrzymuje tak naprawdę ich *kopie*, które istnieją jedynie w czasie działania funkcji, jako wartości ***zmiennych tymczasowych***.

[*Jeśli chcemy, aby funkcja faktycznie zmodyfikowała wartość jakiejś zmiennej — musimy **przekazać jej adres**, technicznie: **wskaźnik do tej zmiennej**.*]

Zaletą *przekazywania przez wartość* polega na tym, że zyskujemy — bez jawnej deklaracji — **dodatkowe zmienne tymczasowe** zainicjowane wartościami wywołania. Często pozwala to pisać funkcje w sposób bardziej zwarty.

Przykładowo, w naszej funkcji `power` możemy pominąć deklarację `int j`; pętla wykonująca mnożenia (dla `n >= 0`) będzie wówczas wyglądać następująco:

```
for (p=1.0; n>0; n--)  
    p *= x;
```

[Cokolwiek zrobimy ze zmienną `n` wewnątrz funkcji `power` — nie ma to **żadnego wpływu** na wartość zmiennej `j` w funkcji `main`.]