

# Poprzedni wykład:

- Elementarz formatowanego wejścia/wyjścia (podstawowe *specyfikacje przekształcenia* dla funkcji printf, scanf)
- Sterowanie: instrukcje i bloki; instrukcja warunkowa (*decyzje wielowariantowe*); instrukcja *switch*; pętle (*while, for, do-while*)
- Przykłady: proste algorytmy sortowania tablicy liczbowej (*bubblesort, shellsort*)
- **Operator przecinkowy**
- **Funkcje w języku C** (c.d.n.)

# 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 równoległe kilkoma indeksami.

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

```
#include <string.h>

/* reverse(s) : odwróć napis s „w miejscu” */
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;
    }
}
```

## UWAGI o operatorze przecinkowym:

- Zanim rozpocznie się obliczanie prawego argumentu, wszystkie efekty uboczne obliczania pozostałych są już zakończone.
- **Jeśli przecinek ma znaczenie specjalne** — np. w *liście inicjatorów* lub *liście argumentów funkcji* — wymaganą jednostką („pomiędzy przecinkami”) jest *wyrażenie przypisania*, wówczas **zagnieżdżony przecinek** może wystąpić w wyrażeniu ujętym w nawiasy okrągłe:

$$f(a, (t=3, t+2), c)$$

**W powyższym wywołaniu funkcji** mamy 3 argumenty; drugi z nich ma wartość 5.

# 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*).

( *Funkcje typu **void** — nie zwracające żadnej wartości — zastępują „procedury” znane w innych językach programowania.* )

Program napisany w **dobrym stylu** powinien składać się z jak największej liczby *krótkich* funkcji (których działanie można testować niezależnie); 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.

# Definicja funkcji w C:

```
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!**

[ *ALE deklaracje funkcji — tzw. **prototypy** — można umieszczać wewnątrz bloków: { ... }* ]

Oprócz **nazwy-funkcji**, każdą część definicji można pominąć:

```
minima() {} /* ==> łac. „najmniejsza” */
```

Funkcje takie jak `minima() {}` bywają użyteczne do „zarezerwowania miejsca” w pliku źródłowym, jeśli planujemy dalszą rozbudowę programu.

Jeśli w definicji funkcji pominięto *typ-zwracanej-wartości* — **kompilator przyjmuje automatycznie**, że jest to **int**.

Jeśli nie chcemy, aby funkcja zwracała jakąkolwiek wartość - deklarujemy ją jako `void` [ wówczas instrukcja `return;` *zwróci jedynie sterowanie do miejsca wywołania.* ]

## Czy funkcja `main()` może być typu `void` ?

**NIE** może i nigdy nie mogła, ani w standardzie C ani w C++ !

[ zob. <https://www.geeksforgeeks.org/fine-write-void-main-cc/> ]

**Kompilator gcc** — program zawiera funkcje `minima() {}` oraz `void main() { } /* NIEDOZWOLONE! */`

```
$ gcc --ansi wzorzec.c
wzorzec.c:9:11: warning: control reaches end of non-
void function
      [-Wreturn-type]
minima() {}
      ^
wzorzec.c:12:1: error: 'main' must return ,int'
void main()
^~~~
int
1 warning and 1 error generated.
```

[ Deklaracje: `void minima()` oraz `int main()` eliminują błąd i ostrzeżenie. *Brak **return** w funkcji `main()` nie generuje ostrzeżenia!* ]



**Przykład:** Program wyszukujący wzorzec (ustalony ciąg znaków) we wprowadzonym tekście. [wg Kernighan & Ritchie, 1994]

Chcemy, aby program czytał kolejne wiersze ze standardowego wejścia, a jeśli w wierszu pojawi się zadany wzorzec — wypisał taki wiersz na ekran. Zapis zadania w pseudokodzie:

while (wczytano wiersz)

    if (wiersz zawiera szukany wzorzec)

        wypisz ten wiersz

[ *Przykład posłuży do zilustrowania zasad podziału programu na funkcje i komunikacji pomiędzy funkcjami.* ]

```

#include <stdio.h>
#define MAXLINE 1000 /* maks. dlugosc wiersza */

int strindex(char source[], char searchfor[]);
char pattern[] = "nie"; /* szukany wzorzec */

/* Wypisz wiersze zawierajace wzorzec: */
int main()
{
    char line[MAXLINE+1]; /* "+1" - miejsce na '\0' */
    int found = 0;

    while (NULL != fgets(line,MAXLINE+1,stdin))
        if (strindex(line,pattern) >= 0) {
            printf("%s",line);
            found++;
        }
    return found; /* liczba znalezionych wierszy */
}

```

```

/* strindex: zwraca pozycje napisu t[] w s[] */
/*          LUB -1 jesli napisu nie znaleziono */
int strindex(char s[], char t[])
{
    int i,j,k;

    for (i=0; s[i]!='\0'; i++) {
        for (j=i,k=0; t[k]!='\0' && s[j]==t[k]; j++,k++)
            ;
        if ( k>0 && t[k]=='\0' )
            return i;
    }
    return -1;
}

```

**Program w C** to, zasadniczo: zbiór definicji zmiennych i funkcji.

**Komunikacja pomiędzy funkcjami** odbywa się za pośrednictwem:

- argumentów wywołania funkcji
- wartości zwracanych przez funkcje
- zmiennych zewnętrznych

**W pliku źródłowym**, funkcje mogą występować w dowolnej kolejności [ => przed wywołaniem: **definicja** lub **prototyp!** ].

Program można **dzielić pomiędzy pliki źródłowe** pod warunkiem, że żadna z funkcji nie zostanie podzielona.

**W naszym przykładzie**, przed funkcją `main()` mamy:

```
int strindex(char source[], char searchfor[]);  
char pattern[] = "nie";
```

Pierwsza linia to tzw. **prototyp funkcji**; a druga – **definicja zmiennej zewnętrznej** [ => *przydzielana jest pamięć!* ].

W *prototypie funkcji*, nazwy parametrów mogą być ( i są! ) inne niż w **definicji funkcji** [ po `main()` ]:

```
int strindex(char s[], char t[]) { ... }
```

Musi się zgadzać jedynie **liczba i typy** parametrów [ *jak również typ-powrotu dla funkcji!* ].

Nazwy są całkowicie **lokalne**; inne funkcje mogą zatem używać parametrów o tych samych nazwach.

Funkcja `strindex` jest wywołana w funkcji `main()`, w wierszu:

```
if (strindex(line,pattern) >= 0) { ... }
```

Wywołanie przekazuje do funkcji dwa argumenty i sterowanie; po zakończeniu działania funkcja wraca do miejsca wywołania wraz z liczbą — pozycją napisu `pattern[]` w `line[]`.

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

Instrukcja **return** — to narzędzie, dzięki któremu funkcja przekazuje wartość pewnego wyrażenia ( i sterowanie ) do miejsca wywołania.

Po słowie kluczowym `return` można umieścić dowolne wyrażenie:

```
return wyrażenie;
```

Jeśli zajdzie potrzeba — wyrażenie zostanie przekształcone do *typu-powrotu* funkcji. [Często wyrażenie otaczane jest nawiasami okrągłymi. Nie jest to konieczne, ale poprawia czytelność.]

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 } )

Jeśli funkcja *powinna* zwrócić wartość a nie ma, lub nie zostanie wykonana, instrukcja `return`; **zostaną zwrócone „śmieci”**.

[ W takiej sytuacji **porządny kompilator** wygeneruje ostrzeżenie. ]

*Nasz program wyszukiwania wzorca zwraca liczbę wierszy ze wzorcem; z tej wartości może skorzystać otoczenie programu.*

## **PRZYPOMNIENIE: Przekazywanie przez wartość**

W języku C 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**. ]*

W przypadku funkcji `strindex` — `s[]` oraz `t[]` to tablice, w języku C równoważne *wskaźnikom na pierwszy element* — w tym przypadku ewentualne modyfikacje zawartości napisów będą miały skutki dla zmiennych `line` oraz `pattern`.



## Programy wieloplikowe (*przypadek najprostszy*)

Techniki ładowanie i tłumaczenia programów podzielonych na kilka plików źródłowych zależą od *systemu* i *kompilatora*.

Przykładowo, jeśli definicję funkcji `strindex` umieścimy w pliku `strindex.c` zaś resztę — tj. deklaracje i funkcję `main()` — w pliku `worzec.c`, wówczas polecenie:

```
gcc worzec.c strindex.c
```

przetłumaczy wszystko i załaduje kod wynikowy do pliku `a.out`.

Poszczególne pliki można też **skompilować niezależnie**:

```
gcc -c worzec.c  
gcc -c strindex.c
```

Następnie, otrzymane kody pośrednie (ang. *object files*) łączymy:

```
gcc worzec.o strindex.o
```

## Uwagi o programach wieloplikowych:

- Podział programu na kilka plików umożliwia niezależną edycję i kompilację ( `gcc -c` ) fragmentów kodu
- W szczególności, na etapie pośrednim — można napisać jedynie **prototypy** niektórych funkcji i sprawdzać poprawność składniową reszty programu [ **WAŻNE: prototypy funkcji mogą się powtarzać, nawet w tym samym pliku, definicje — NIE!** ]

Przykład (`strindex.c`) jest wyjątkowo prosty, ponieważ funkcja w tym pliku nie wywołuje innych funkcji (ani nie korzysta ze zmiennych zewnętrznych); zwykle konieczne są deklaracje, które umieszcza się w osobnych plikach ( np. `worzec.h` ) włączanych dyrektywą `#include "plik-nagłówkowy"`. (ang. *header-file*)

[ **Ostatnie zagadnienie będzie jeszcze omawiane!** ]

## Uwagi o programach wieloplikowych — c.d.

Chociaż podział programu na kilka plików (w celu niezależnego rozwijania *logicznie odrębnych* fragmentów) jest bardzo wygodny, warto pamiętać o kilku **zasadach / pułapkach**:

- jeśli pomiędzy *definicją* a *wywołaniem* funkcji występuje **niezgodność typów**, kompilator zgłasza błąd, o ile definicja i wywołanie występują w tym samym pliku (!)

Jeśli **wywołanie** i **definicja** są w różnych plikach, działają reguły:

- w przypadku **braku prototypu**, mamy deklarację niejawną (przez kontekst) przy pierwszym wywołaniu. Domyślnie, niezadeklarowana nazwa z nawiasami po prawej stronie uznawana jest za funkcję typu `int`.

[ *Jeśli definicja jest np. typu **double** - wyniki nie będą miały sensu!* ]

- **prototyp bez parametrów:** `int strindex()`; wyłącza kontrolę poprawności wywołania ( **brak założeń o parametrach!** ), nie oznacza koniecznie, że funkcja na nie ma parametrów!

Jeśli chcemy napisać prototyp funkcji bez parametrów, najlepiej napisać jawnie:

*typ-powrotu* **nazwa-funkcji**( `void` );

[ *Pozwoli to uniknąć wielu **złośliwych** błędów.* ]

**UWAGA:** Opisany mechanizm **prototypów domyślnych** w ANSI C, wygodny w przypadku jednorazowego wywołania funkcji z biblioteki zewnętrznej z b.długą listą parametrów, został porzucony w standardzie C99. Nie należy się zatem do niego przyzwyczajać ...

# Zmienne zewnętrzne

Program w C — to zbiór **obiektów zewnętrznych**; mogą być nimi zmienne lub funkcje.

**Zmienne zewnętrzne** definiuje się poza wszystkimi funkcjami, mogą być zatem [ *dalej — omówimy „zasięg nazw”* ] dostępne dla *wszystkich funkcji*.

Funkcje są zawsze zewnętrzne; **prototypy** można jednak deklarować wewnątrz bloków ograniczonych nawiasami { }

## „Zewnętrzna łączność nazwy”:

Przyjmuje się, że wszystkie odwołania do zmiennych zewnętrznych i funkcji za pomocą tej samej nazwy — także z **różnych plików źródłowych** — dotyczą tego samego obiektu.

Zmienne zewnętrzne są **ogólnie dostępne** — a zatem mogą stanowić wygodny substytut argumentów funkcji oraz wartości zwracanych przez funkcje; zwłaszcza w sytuacjach, kiedy

- funkcje operują na dużej liczbie wspólnych danych
- zachodzi potrzeba faktycznej modyfikacji wartości wielu zmiennych (por. **przekazywanie przez wartość**)
- warto skrócić długie listy argumentów funkcji
- zachodzi potrzeba przechowywania wyników pomiędzy wywołaniami różnych funkcji

*Dowolna funkcja może odwołać się do każdej zmiennej zewnętrznej za pomocą jej nazwy, o ile nazwa ta jest zadeklarowana w sposób widoczny [ **patrz dalej — zasięg nazw !** ] dla definicji funkcji.*

# Zasięg nazw

Zasięgiem nazwy (lub *identyfikatora*) jest ta **część programu**, wewnątrz której danej nazwy można używać.

- Dla ***zmiennej automatycznej*** — zasięgiem jest obszar od deklaracji do zamykającego nawiasu klamrowego } }
- Zmienne lokalne o takich samych nazwach, występujące w różnych funkcjach (lub różnych blokach { ... } ) nie są powiązane w żaden sposób
- Powyższe uwagi — dotyczą także ***parametrów funkcji***

Dla ***zmiennych zewnętrznych*** ( *i funkcji!* ) zasięg rozciąga się od deklaracji do końca pliku źródłowego.

## Przykład – *ilustrujący zasięg zmiennych zewnętrznych.*

W kalkulatorze [zob. *Kernighan—Ritchie, rozdział 4.*] używamy tzw. stosu, zdefiniowanego z użyciem tablicy liczb `double` i zmiennej typu `int` przechowującej informacje, *ile liczb położono na stosie.*

**Definicje** pojawiają się w następującej kolejności:

```
int main()    { ... }

double stos[ROZMIAR_STOSU];
int top=0;

void push(double x) { ... }
double pop(void) { ... }
```



Funkcje `push()` i `pop()` mogą odwoływać się do zmiennych `stos` i `top` po prostu **przez nazwy**, żadne *dodatkowe deklaracje nie są potrzebne*.

Zmienne `stos` i `top` są jednak niewidoczne dla funkcji `main()`, *podobnie* zresztą jak funkcje `push()` i `pop()` (!)

Jeśli odwołanie do zmiennej zewnętrznej występuje przed jej definicją (lub *definicja znajduje się w innym pliku*) konieczna jest deklaracja z użyciem słowa kluczowego `extern`. Przykładowo:

```
extern double stos[];
```

```
extern int top;
```

*informuje* resztę pliku, że `stos` jest tablicą typu `double` zaś `top` zmienną typu `int`, nie przydziela jednak pamięci dla tych zmiennych

[ ***Deklaracja: rezerwuje tylko nazwę, definicja — także pamięć.*** ]

# Zmienne statyczne

Słowo kluczowe **static** w deklaracjach *zmiennych zewnętrznych* (*lub funkcji!*) dodatkowo ogranicza ich zasięg — od miejsca wystąpienia, do końca konkretnego pliku źródłowego.

Deklarowanie obiektów zewnętrznych jako **static** jest zatem sposobem na ukrycie ich nazw (*dla innych plików*).

**Także funkcje** można deklarować jako **static**, np:

```
static void push(double x) { ... }
```

spowoduje, że funkcja `push(...)` pozostanie ukryta poza plikiem zawierającym deklarację.

**Zmienne wewnętrzne** zadeklarowane jako **static** — są niewidoczne poza „swoim” blokiem; *jednak przechowują dane przez cały czas działania programu* [ *nie są to zatem zm. automatyczne!* ]

## **Przypomnienie - ( zob. Wykład 2 ! ):**

Zmienna lokalna poprzedzona **static** zachowuje się (*prawie ...*) jak zewnętrzna, tzn. pamięta wartość pomiędzy wywołaniami funkcji, jednak *nie jest widoczna poza swoim blokiem* { ... }

Jeśli zmienna zewnętrzna jest zadeklarowana jako static, będzie widoczna **wyłącznie w "swoim" PLIKU** (ale można ją redefiniować w innym pliku za pomocą extern ).

# Zmienne rejestrowe

Słowo kluczowe **register** pozwala deklarować zmienne, informując jednocześnie kompilator, że odwołania do nich będą częste, a zatem powinny zostać — w miarę możliwości — umieszczone w **pamięci podręcznej** procesora (ang. *CPU cache*).

Deklaracja **register** jest dozwolona dla zmiennych automatycznych i parametrów formalnych funkcji:

```
fun(register unsigned m, register long n)
{
    register int j;
    ...
}
```

W praktyce: *typy*, a nawet *liczba zmiennych rejestrowych* dostępnych jednocześnie podlegają dodatkowym ograniczeniom.

- „Nadliczbowe” użycie słowa **register** nigdy jednak nie jest błędem; zmienna **register** będzie w takiej sytuacji działała jak zwykła zmienna.
- Podobnie, niepoprawne deklaracje „*zewnątrznych zmiennych* **register**” doprowadzą do utworzenia zwykłych zmiennych

***Nigdy nie jest możliwe uzyskanie adresu*** zmiennej poprzedzonej kwalifikatorem **register**, niezależnie od tego, czy faktycznie zmienna *trafiła do rejestru*, czy też nie.

## Uwagi o strukturze blokowej programu:

W C nie mamy „klasycznej struktury blokowej”, tj. funkcje nie mogą być umieszczone wewnątrz innych funkcji („zagnieżdżone”).

[ Pewnym substytutem „zagnieżdżania funkcji” jest możliwość definiowania funkcji **static**, widocznych tylko w jednym pliku. ]

- **Deklaracje** (lub **definicje**) zmiennych mogą być umieszczone wewnątrz dowolnej instrukcji złożonej { ... }
- To samo — dotyczy **prototypów funkcji**.

**Zmienne automatyczne** (a także *parametry funkcji*) zastępują zmienne zewnętrzne o tych samych nazwach.

[ *Zastępowanie pobliskich zmiennych lepiej unikać; niebezpieczeństwo popełnienia **trudnych do wykrycia błędów** jest znaczne.* ]

# Inicjowanie zmiennych

Poniżej — zasady inicjowania zmiennych dla różnych klas pamięci.

[ **Inicjowanie** = nadawanie wartości początkowych ]

Jeśli nie podano *jawnie* wartości początkowych:

- Zmienne **zewnętrzne** i **statyczne** zawsze inicjowane zerami
- Wartości zm. **automatycznych** i **rejestrowych** — przypadkowe

Zmienne **skalarne** (= *nie-tablice*) można inicjować przy definicji:

```
int x = 1;
```

```
char squote = '\''; /* Tak piszemy apostrof */
```

```
long day = 60L * 60L * 24L; /* doba w sek. */
```

Wartością początkową zmiennych **zewnętrznych** i **statycznych** musi być stała lub wyrażenie stałe.

[ Inicjowanie takich zmiennych odbywa się **tylko raz** — *przed rozpoczęciem działania programu.* ]

Zmienne **automatyczne** i **rejestrów** — inicjowane ponownie przy każdym wejściu do funkcji lub bloku.

Wartością początkową zmiennej **automatycznej** lub **rejestrów** nie musi być stała — dopuszczalne są dowolne wyrażenia, w tym **również wywołania funkcji (!)**.

[ **Jawne inicjowanie** — to *de facto* skrócony zapis instrukcji przypisania.

=> **Klasyczne przypisania** bywają bardziej przejrzyste — odległość *definicji od miejsca użycia zmiennej* może być znaczna ... ]



**Tablice** *inicjujemy* podając listę wartości początkowych:

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Jeśli w definicji pominięto rozmiar tablicy — kompilator sam go ustala [ **w tym przypadku 12** ].

Jeśli podany rozmiar byłby **większy** niż długość listy wartości (np. `int days[100] = {...};` ) wówczas *nadliczbowe* pola zostaną wyzerowane. Jeśli podany rozmiar będzie **za mały** — wystąpi błąd.

**Tablice znakowe** ( = *napisy* ) inicjujemy następująco:

```
char wzorzec[] = "nie";
```

Jest to równoważne definicji:

```
char wzorzec[] = {'n', 'i', 'e', '\0'};
```

[ **A zatem rozmiar wynosi 4** = 3 znaki + znak-końca napisu: '\0' ]

# Rekurencja

Funkcje w C mogą być *wywoływane rekurencyjnie* — tzn. funkcja może wywoływać samą siebie (bezpośrednio lub pośrednio).

**Przykład:** `printf(n)` wypisuje `n` w postaci ciągu znaków

```
void printf(int n)
{
    if (n<0) {
        putchar('-');
        n = -n;
    }
    if (n/10) printf(n/10);
    putchar(n % 10 + '0');
}
```

Kiedy funkcja **wywołuje samą siebie**, każde jej wznowienie otrzymuje zupełnie nowy komplet wszystkich zmiennych automatycznych.

**Na przykład**, po wywołaniu `printf(123)` :

- *pierwsze* `printf` otrzymuje argument `n` o wartości 123
- *drugie* `printf` otrzymuje wartość 12
- *trzecie* `printf` otrzymuje 1 [ **mamy zatem:  $n/10==0$  (!)** ]

Wówczas, warunek `if (n/10)` przestaje być prawdziwy, trzeci `printf` wypisuje zatem znak '1' i wraca na drugi poziom.

Dalej, `printf` z drugiego poziomu wypisze '2' i wróci na poziom pierwszy, z którego wypisane zostanie '3' po czym funkcja zakończy działanie.

# Porządkowanie tablicy liczbowej (3): *quicksort*

Algorytm *quicksort* został podany przez C.A. Hoare'a w 1962 r.

- 1) Dla tablicy wybieramy pewien element (*tutaj będzie to element środkowy*); pozostałe elementy dzielimy na **2 podzbiory**:
- 2) elementy mniejsze od wybranego umieszczamy **przed** nim
- 3) elementy większe — **za** wybranym elementem
- 4) Kroki 1)–3) powtarzamy (*rekurencyjnie*) dla dwóch podzbiorów: elementów poprzedzających wybrany w kroku 1), oraz dla elementów umieszczonych po nim.

Rekurencja kończy się, gdy podzbiory mają „**mniej niż dwa**” elementy (a zatem: 0 lub 1) — wówczas tablica jest już uporządkowana.

[ *Wersja poniżej — wg Kernighan & Ritchie, rozdz.4 — nie jest wersją najbardziej efektywną, jest jednak **najprostsza**.* ]

```
void qsort(int v[], int left, int right)
{
    int i, last;    /* zmienne automatyczne */
    void swap(int v[], int i, int j); /* prototyp f.*/

    if (left >= right) return;    /* koniec! */
    swap(v, left, (left + right)/2); /* wybrany el. */
    last = left;    /* ostatni el. < wybrany el. */
    for (i=left+1; i<=right; i++)
        if (v[i] < v[left]) swap(v, ++last, i);
    swap(v, left, last);    /* wybrany elem. wraca */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Operacje zamiany 2 elementów realizuje **osobna funkcja**:

```
/* swap: zamień miejscami v[i] z v[j] */  
void swap(int v[], int i, int j)  
{  
    int temp;  
  
    temp = v[i];  
    v[i] = v[j];  
    v[j] = temp;  
}
```

[ *Biblioteka standardowa <stdlib.h> zawiera implementację funkcji **qsort(...)** która potrafi sortować obiekty dowolnego typu. ]*