Poprzedni wykład [*21.12.2021*]:

- Argumenty wiersza poleceń
- Podstawy obsługi błędów: `stderr, exit, ferror`
- Parametry ***opcjonalne*** wywołania programu
- **Wskaźniki do funkcji:** deklaracje, tablice
- Użycie wskaźników do funkcji jako **argumentów funkcji** (zagadnienie *redundancja kodu źródłowego*)

Parametry funkcji `main()`: *Uzupełnienie*

⇒ Czy funkcja **`main()`** może wywoływać sama siebie?

⇒ Czy inne funkcje mogą wywoływać **`main()`** ?

Według standardu ANSI, w języku C: **TAK**

*(**Zaś w języku C++: NIE!**)*

https://stackoverflow.com/questions/4238179/calling-main-in-main-in-c/

Przykład 1. — *Echo argumentów bez pętli:*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  if (argc>1 && !main(argc-1, argv))
    printf((argv[argc]) ? "%s " : "%s\n",
           argv[argc-1]);

  /* tutaj można dodać obsługę błędu stdout … */

  return 0;
}
```

Jeśli program został wywołany z *jakimiś* argumentami, mamy `argc>1`, a zatem drugi składnik koniunkcji logicznej, tj.:
`!main(argc-1,argv)` — ***będzie obliczany***.

Oznacza to, że funkcja `main` zostanie wywołana ponownie, *z pierwszym argumentem zmniejszonym o 1*.

Powyższe działanie zostanie wykonane (łącznie) `argc-1` razy, po czym warunek (`argc>1`) stanie się fałszywy, *ostatnia* funkcja main zwróci zero i sterowanie wróci poziom wyżej.

Dalej, funkcje `printf(…)` z kolejnych poziomów `main()` będą wykonywane, wypisując `argv[1] argv[2] … argv[argc-1]`

[*Powyżej — odwołujemy się do `argc` z „pierwszego main” !*]

Użyty w 1. argumencie printf, warunek `(argv[argc])` [!= NULL] sprawdza, czy wartość `argc` została zmodyfikowana.

Jak w przypadku każdej funkcji, użycie:

```
return wyrażenie
```

wewnątrz main spowoduje powrót poziom wyżej.

Z kolei, wywołanie: **`exit(wyrażenie)`** spowoduje zakończenie działania programu *(poprzedzone zamknięciem plików i opróżnieniem buforów)*.

Program realizujący ***echo argumetów bez pętli*** przepiszemy teraz w wersji z przesuwaniem wskaźnika `argv`, tak aby wypisywanie argumentu odbywało się *najpierw*, a następnie funkcja `main()` była wywoływana z pierwszym argumentem zmniejszonym o jeden, a drugim przesuniętym z pozycji `&argv[0]` do `&argv[1]`.

Przykład 2. — *Echo argumentów bez pętli* *(wersja z przesuwaniem wskaźnika **argv**):*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  if (--argc) {
    printf((argc>1) ? "%s " : "%s", *++argv);
    return main(argc, argv);
  }
  printf("\n");
  return 0;
}
```

Podobnie jak poprzednio, warunek `(--argc)` jest prawdziwy jeśli *(przed zmniejszeniem!)* mamy `argc>1`

Wówczas, wywołanie `prinft(…)` spowoduje wypisanie `argv[1]`, a jeśli argc>1 *(po zmniejszeniu)* dodatkowo zostanie wypisana spacja.

Następnie, wywołana zostanie funkcja `main(…)` kolejnego poziomu (argumenty już zostały zmodyfikowane tak, aby funkcja „zobaczyła” kolejny argument jako pierwszy).

Ponieważ — *tym razem* — użyliśmy instrukcji `return main(…)` po powrocie, funkcja `main` wyższego poziomu od razu zakończy działanie.

Ostatnia instrukcja [`printf("\n");`] zostanie zatem wykonana jedynie przy ostatnim wywołaniu main, gdy argc==1.

Czy można napisać program bez *main()*?

Jest jasne, że po uruchomieniu skompilowanego programu z poziomu systemu operacyjnego, nasza funkcja `main()` ***nie może być wywoływana jako pierwsza***.

COŚ musi zebrać argumenty wywołania i przekazać je do main().

W systemach, które obsługują standard **ELF** (ang. *Executable and Linkable Format*) tym „czymś” jest funkcja `_start()` której standardowa wersja jest ładowana na etapie łączenia programu.

Możemy zatem napisać własną funkcję `_start()`, która wywoła *coś innego* niż `main()`, a następnie zmusić kompilator, aby nie ładował wersji standardowej.

https://www.geeksforgeeks.org/executing-main-in-c-behind-the-scene/

Opisane ćwiczenie zostało wykonane pod systemem `Scientific Linux release 6.9 (Carbon)` z kompilatorem gcc `4.4.7` i narzędziem `objdump`.

Na początek, napiszemy *b.mały* program:

```
main() {}  /* Koniec! */
```

i skompilujemy go poleceniem: `gcc -ansi`

(*Powstały plik `a.out` ma rozmiar 6263 bajty*)

Każdy **plik ELF** ma pole **`e_entry`** zawierające adres w pamięci, od którego zaczyna się wykonanie programu.

Sprawdzimy teraz, na co pokazuje nasze pole **e_entry**:

Polecenie: **`objdump -f a.out`** generuje wynik:

```
a.out:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400390
```

Dalej, aby sprawdzić, co dokładnie znajduje się pod adresem **`0x0000000000400390`** piszemy:

```
objdump --disassemble a.out
```

Tym razem wynik jest bardzo długi, można jednak znaleźć sekcję:

```
0000000000400390 <_start>:
  400390:  31 ed                    xor    %ebp,%ebp
  400392:  49 89 d1                 mov    %rdx,%r9
  400395:  5e                       pop    %rsi
  400396:  48 89 e2                 mov    %rsp,%rdx
  400399:  48 83 e4 f0              and    $0xfffffffffffffff0,%rsp
  40039d:  50                       push   %rax
  40039e:  54                       push   %rsp
  40039f:  49 c7 c0 80 04 40 00     mov    $0x400480,%r8
  4003a6:  48 c7 c1 90 04 40 00     mov    $0x400490,%rcx
  4003ad:  48 c7 c7 74 04 40 00     mov    $0x400474,%rdi
  4003b4:  e8 c7 ff ff ff           callq  400380 <__libc_start_main@plt>
  4003b9:  f4                       hlt
  4003ba:  90                       nop
  4003bb:  90                       nop
```

Widzimy zatem, że adres podany wcześniej przez **`objdump -f`** faktycznie wskazuje na funkcje **`_start()`**

Rolą funkcji **`_start()`** jest przygotowanie argumentów dla innej funkcji, **`_libc_start_main()`**, której prototyp wygląda tak:

```
int __libc_start_main(int (*main)(int, char **, char **),
                      /* main is a pointer to the main function */
int argc, /* number of command line args*/
char ** ubp_av, /* command line arg array*/
void (*init)(void), /* pointer to init*/
void (*fini)(void), /* pointer to fini*/
void (*rtld_fini)(void), /* pointer to dynamic linker fini */
void (*stack_end) /* end of the stack address*/
);
```

Funkcja ta przygotowuje zmienne środowiskowe dla programu, wywołuje funkcje **`_init()`**, która przygotowuje argumenty dla **`main()`**, oraz (*po kilku innych krokach ...*) wywołuje **`main()`**.

Prosty program ***bez funkcji main*** może wyglądać tak:

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

int my_fun()  /* nasza funkcja "main" */
{
    printf("Znam ELF-y i nie mam main-a!\n");
    return 0;
}
void _start()
{
    int x = my_fun(); /* wywołanie naszej "main" */
    exit(x);
}
```

Kompilator **`gcc`** udostępnia opcję **`-nostartfiles`** która sprawia, że program nie będzie korzystał ze standardowej funkcji **`_start()`** lecz załaduje naszą.

Pisząc zatem:

```
gcc -nostartfiles nomain.c -o nomain
```

otrzymamy program wynikowy (**`nomain`**) który po uruchomieniu wypisze na standardowe wyjście:

```
Znam ELF-y i nie mam main-a!
```

Struktury

W języku C ***struktura*** to *jedna lub kilka zmiennych*, które *mogą być różnych typów*, połączonych w tak, aby możliwe były odwołania za pomocą ***wspólnego identyfikatora (=nazwy)***.

Naturalnym zastosowaniem struktur są wszelkie bazy danych; pojedynczy ***rekord*** może być np. strukturą zawierającą *imię, nazwisko, adres,* i *numer telefonu* danej osoby.

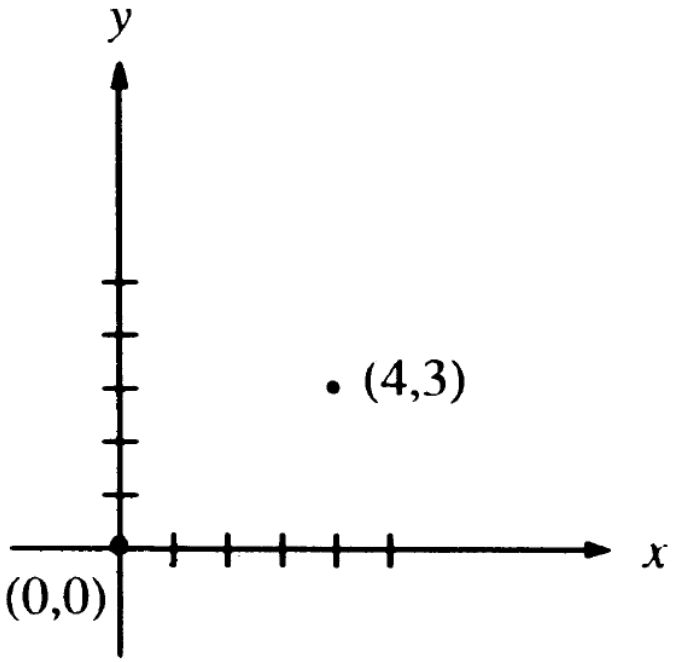
Inaczej niż w przypadku tablic, struktury mogą być ***przypisywane i kopiowane*** (*w całości!*), ***przekazywane do funkcji***, jak również ***zwracane przez funkcje***.

Strukturom (*podobnie jak tablicom …*) można również ***nadawać wartości początkowe*** (=> *inicjowanie struktur*).

Rozważymy teraz przykładowe struktury, często pojawiające się w programach z *grafiką dwuwymiarową*.

Pierwszą strukturą będzie ***punkt***, rozumiany jako para współrzędnych całkowitych *x* i *y*:

```
struct point {
   int x;
   int y;
};
```



Deklarację rozpoczyna kluczowe słowo kluczowe `struct`, po którym pojawia się (*opcjonalna*) etykieta struktury (tutaj: `point`).

[*Fraza:* `struct point` *może być używana zupełnie jak nazwa typu.*]

Zmienne występujące wewnątrz nawiasów `{ … }` to tzw. *składowe* (lub: *pola*) *struktury*.

Po klamrze zamykającej definicję struktury może pojawić się lista zmiennych:

```
struct { … } x, y, z;
```

co `>>składniowo<<` pozostaje w analogii do deklaracji:

```
int x, y, z;
```

Jeśli po ***deklaracji struktury*** nie pojawi lista zmiennych, sama deklaracja nie rezerwuje żadnej pamięci; może posłużyć jedynie do zdefiniowania etykiety, której poźniej użyjemy definiując konkretne wcielenia struktury, jak np.:

```
struct point pt;
```

Struktury można ***inicjować*** podając *listę wartości początkowych poszczególnych składowych*. W takim przypadku, poszczególne wartości początkowe muszą być *stałymi* lub wyrażeniami stałymi:

```
struct point maxpt = { 640, 480 };
```

Innym sposobem, dostępnym dla ***zmiennych automatycznych*** (*będących strukturami*) jest przypisanie innej struktury, lub wywołanie funkcji, która zwraca strukturę właściwego typu.

Dostęp do pól struktury w dowolnym wyrażeniu umożliwia konstrukcja:

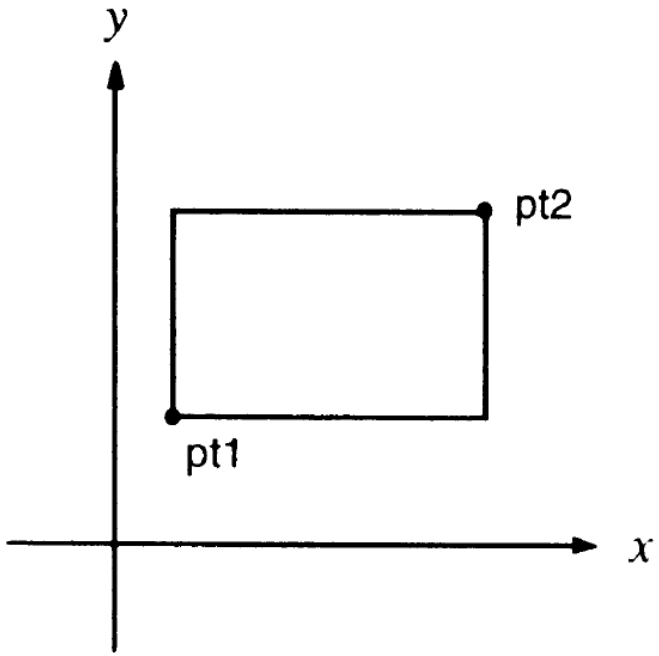
nazwa-struktury.nazwa-składowej

Przykładowo, współrzędne punktu możemy wypisać instrukcją:

```
printf("%d, %d\n", pt.x, pt.y);
```

Struktury można zagnieżdżać. Przykładowo, prostokąt (*o bokach równoległych do osi*) na płaszczyźnie często reprezentuje para przeciwległych wierzchołków:

```
struct rect {
   struct point pt1;
   struct point pt2;
};
```



Jeśli zadeklarujemy zmienną:

```
struct rect screen;
```

to wyrażenie `screen.pt1.x` odwołuje się do współrzędnej `x` punktu `pt1`, który jest składową zmiennej `screen`.

Struktury i funkcje

Dozwolone operacje dla struktury w języku C:

- przypisanie innej struktury (*w całości*)
- skopiowanie do innej struktury (*j.w.*)
- pobranie adresu operatorem `&`
- przekazanie struktury jako argumentu (*przez wartość!*) do funkcji
- zwrócenie struktury jako wartości przez funkcje

Podobnie jak w przypadku tablic, ***niedozwolone*** są operacje *porównania* dla całych struktur.

Funkcje operujące na strukturach, w zależności od sytuacji, piszemy zwykle na jeden z trzech sposobów:

- przekazując ***składowe*** oddzielnie (*pole po polu*)
- przekazując ***całą strukturę***
- przekazując ***wskaźnik do struktury***

Funkcja makepoint tworzy punkt korzystając z podanych x i y:

```
struct point makepoint(int x, int y) {
   struct point tmp;
   tmp.x = x;
   tmp.y = y;
   return tmp;
}
```

Każde wywołanie funkcji makepoint ***przydzieli dynamicznie*** pamięć dla nowej zmiennej typu `struct point` (por. zmienna *automatyczna* `tmp`) może zatem służyć do inicjowania dowolnych zmiennych tego typu, w tym także *zagnieżdżonych w innych strukturach:*

```
struct rect screen;

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX );
```

[*Powyższy przykład obrazuje istotną przewagę struktury nad tablicą w sytuacji, gdy liczba składowych jest* ***niewielka i ustalona****.*]

Kolejna funkcja — ***dodaje współrzędne*** dwóch punktów:

```
struct point
addpoint(struct point p1, struct point p2)
{
   p1.x += p2.x;
   p1.y += p2.y;
   return p1;
}
```

Korzystamy tutaj z faktu, że funkcja w języku C ***operuje zawsze na kopiach*** swoich argumentów, a zatem np. instrukcja:

```
p3 = addpoint(p1,p2);  /* ==> nie zmienia pól p1 */
```

Jeśli zachodzi potrzeba sprowadzenie prostokąta do ***postaci kanonicznej*** (tj. takiej, że współrzędne `p1` są *nie większe* niż współrzędne `p2`), możemy zdefiniować funkcję:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

struct rect canonrect(struct rect r)
{
   struct rect tmp;
   tmp.pt1.x = min(r.pt1.x, r.pt2.x);
   tmp.pt1.y = min(r.pt1.y, r.pt2.y);
   tmp.pt2.x = max(r.pt1.x, r.pt2.x);
   tmp.pt2.y = max(r.pt1.y, r.pt2.y);
   return tmp;
}
```

A następnie wywołać ją podając tę samą strukturę jako argument funkcji i lewy argument operatora przypisania, np.

```
screen = cannonrect(screen);
```

Jeśli chcemy przekazać do funkcji ***dużą strukturę***, zwykle bardziej efektywne od jej kopiowania w całości będzie przekazanie wskaźnika.

Przekazywanie wskaźników może być szczególnie użyteczne w sytuacji gdy chcemy, aby funkcja zmodyfikowała tylko wybrane pola struktury, podczas gdy *„klasyczna”* konstrukcja postaci:

nazwa-struktury = funkcja(nazwa-struktury);

wykonuje ***dwukrotne kopiowanie*** *całej struktury*.

Z drugiej strony, mechanizm *kopiowania struktur* pozwala stosunkowo łatwo obejść ograniczenie języka C polegające na tym, że funkcje nie mogą zwracać tablic (a jedynie wskaźniki).

Jeśli zakładamy, że nasz program będzie wykonywał operacje np. na bardzo wielu ***macierzach 3 x 3*** (a nie chcemy kłopotać się przydziałem pamięci za każdym razem, kiedy zachodzi taka potrzeba) możemy zdefiniować strukturę:

```
struct tab3x3 { double tab[3][3]; };
```

która — ***w odróżnieniu od samej tablicy dwuwymiarowej!*** — będzie mogła zostać zwrócona przez funkcje i przypisana innej zmiennej typu `struct tab3x3`.

W uproszczeniu, *9 elementów:* `tab[0][0]` … `tab[2][2]` zachowuje się zupełnie jak osobne pola struktury.

Przykładowo, funkcja poniżej ***tworzy macierz jednostkową***:

```
struct tab3x3 makeItab3x3()
{
  int i,j;
  struct tab3x3 tmp;
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      tmp.tab[i][j] = (i==j) ? 1.0 : 0.0;
  return tmp;
}
```

Funkcja ***main*** — kopiuje macierz jednostkową 3 x 3 do zmiennej `t` i wypisuje jej zawartość na wyjście:

```
int main()
{
  struct tab3x3 t;
  int i,j;

  t = makeItab3x3();
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      printf((j<2)?"%lf ":"%lf\n", t.tab[i][j]);
  return 0;
}
```

Wskaźniki do struktur deklarujemy podobnie jak dla zwykłych (<=> *skalarnych*) zmiennych:

```
struct point *ppt;
```

oznacza zatem, że `ppt` jest wskaźnikiem do zmiennej typu `struct point`.

Wyrażenie `(*ppt)` może wystąpić wszędzie tam, gdzie nazwa zmiennej typu `struct point`. Przykładowo, możemy napisać:

```
struct point origin, *ppt;

ppt = &origin;
printf("origin == (%d,%d)\n", (*ppt).x, (*ppt).y);
```

Warto podkreślić, że nawiasy w wyrażeniu `(*ppt).x` są niezbędne, ponieważ operator składowej struktury `.` wiąże silniej niż operator dereferencji `*`.

Wyrażenie `*ppt.x` znaczyłoby zatem tyle samo co `*(ppt.x)` — *wyłuskanie wartości ze wskaźnika* `ppt.x` — co w rozważanym przypadku byłoby **błędne**, ponieważ pole x nie jest wskaźnikiem, (*a dodatkowo ppt nie jest strukturą …*).

Wskaźniki do struktur używane są na tyle często, że w języku C występuje ***dodatkowy operator*** umożliwiający skrócony zapis:

nazwa-wskaźnika–>nazwa-składowej

co oznacza odwołanie do konkretnego pola wskazywanej struktury.

Dla naszego wskaźnika `ppt` do struktury typu `struct point` możemy zatem napisać:

```
printf("origin == (%d,%d)\n", ppt->x, ppt->y);
```

[*Wyrażenia `ppt->x` oraz `ppt->y` są w pełni* ***równoważne*** *wcześniej użytym formom `(*ppt).x` oraz `(*ppt).y`*]

Operatory `.` oraz `->` są ***łączne lewostronnie***, zatem po definicji:

```
struct rect r, *rp = &r;
```

następujące cztery (*równoważne!*) wyrażenia oznaczać będą odwołanie do współrzędnej `x` punktu `pt1`:

```
r.pt1.x   rp->pt1.x   (r.pt1).x   (rp->pt1).x
```

Operatory działające na strukturach: `.` oraz `->` znajdują się na szczycie ***hierarchii operatorów*** języka C (tzn. **najsilniej wiążą swoje argumenty**), *ex aequo* z nawiasami okrągłymi `( )` wywołania funkcji oraz kwadratowymi `[ ]` indeksowania tablicy.

Przykładowo, po deklaracji:

```
struct { int len; char *str; } *p;
```

wyrażenie

```
++p->len
```

jest równoważne `++(p->len)` a zatem zwiększy zmienną `len` a nie wskaźnik `p` (**!**)

Jeśli chcemy zwiększyć `p` ***przed*** odwołaniem do `len` musimy napisać: `(++p)->len`

Dla odmiany, jeśli chcemy zwiększyć `p` ***po*** odwołaniu do `len` można napisać po prostu: `p++->len` (*łączność lewostronna!*).
[*Chociaż zapewne zapis* `(p++)->len` *jest bardziej zrozumiały …*]

Podobnie, wyrażenie `*p->str` jest równoważne `*(p->str)` a zatem udostępni *znak* wskazywany przez `str`.

Dalej, wyrażenie `*p->str++` zwiększy `str` po *wcześniejszym* udostępnieniu wskazywanego znaku (*znana konstrukcja:* `*s++`).

Wyrażenie `(*p->str)++` powiększy *znak wskazywany* [`(*s)++`].

Wyrażenie `*p++->str` [*jest równoważne* `*((p++)->str)`] — zwiększy `p` po wcześniejszym udostępnieniu znaku wskazywanego przez `str`.

Tablice struktur

Aby zilustrować użyteczność ***grupowania struktur w tablice*** (i stosowania *arytmetyki wskaźników* dla wskaźników na struktury) przedstawimy teraz *program zliczający słowa kluczowe* języka C.

Jedną z możliwości jest stworzenie dwóch równoległych tablic, z których pierwsza przechowuje same *słowa* a druga *liczniki ich wystąpień*:

```
char *keyword[NKEYS];  /* słowa kluczowe */
int keycount[NKEYS];  /* liczniki */
```

Widać jednak, że każdy z liczników jest wyraźnie ***powiązany logicznie*** z odpowiadającym mu słowem, ale za to nie ma większego związku z innymi licznikami (**!**).

[*W szczególności, kiedy zajdzie potrzeba* ***przeczytania następnego słowa*** *z tablicy* `keyword` *— a zdążyliśmi już polubić arytmetykę wskaźników — trzeba będzie każdorazowo zwiększać jednocześnie* `keyword` *oraz* `keycount` *aby się nie zgubić ...*]

Z tych powodów, wygodne będzie zadeklarowanie struktury:

```
struct key {
   char *word;
   int count;
};
```

oraz tablicy takich struktur, np.: `struct key keytab[NKEYS];`

Początek programu będzie wyglądał tak:

```
/*  Zlicza slowa kluczowe C - wersja WSKAŹNIKOWA     */
/*                  wg Kernighan-Ritchie + zmiany AR  */

#include <stdio.h>
#include <ctype.h>
#include <string.h>

struct key {
   char *word;
   int count;
} keytab[] = {
   "auto", 0,  "break", 0,  "case", 0,  "char", 0,
   /*  …  */
   "void", 0, "volatile", 0, "while", 0
};  /* c.d.n … */
```

Zasadniczo, ***inicjując tablicę*** `keytab` powinniśmy pisać:

```
{ {"auto", 0}, {"break", 0}, {"case", 0}, … }
```

tzn. każda para słowo-licznik (umowny *„wiersz tablicy”*) powinna mieć swoją parę nawiasów klamrowych.

Jednakże, wartościami początkowymi są ***stałe*** i ***napisy stałe***, jak również ***podajemy wszystkie wartości*** — a w takim przypadku wewnętrzne nawiasy można pominąć.

Nie podaliśmy także rozmiaru (NKEYS) tablicy `keytab` — kompilator sam go oblicza, a dostęp do wyniku możemy łatwo osiągnąć za pomocą operatora `sizeof`, co najmniej na ***dwa sposoby*** *[liczenie „ręczne” — pomijamy …]*:

- *Sposób 1*: `sizeof keytab / sizeof(struct key)`
- *Sposób 2*: `sizeof keytab / sizeof keytab[0]`

Drugi ze sposobów jest się ***nieco lepszy***, gdyż odwołuje się wyłącznie do jednego identyfikatora (*nazwy tablicy*), a zatem nie będzie wymagał zmiany jeśli — modyfikując nasz program — decydujemy, że **`keytab`** ma być jakąś inną tablicą.

Warto podkreślić, że *dodanie rozmiarów* poszczególnych elementów struktury byłoby ***nieprawidłowe.***

[***Rozmiar struktury na ogół nie jest równy sumie rozmiarów jej elementów!*** *Nawet struktura złożona ze znaku i liczby całkowitej może, zamiast pięciu, zajmować np. osiem bajtów.*]

Dalsza część ***bloku deklaracji*** programu wygląda tak:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
#define MAXWORDLEN 100

int getword(char * word, int lim);

struct key *
binsearch(char * word, struct key * tab, int n);
```

Funkcja `getword` wczytuje kolejne słowo [*słowa kluczowe często występują w sąsiedztwie nawiasów:* () {}, *lub* *, `scanf` *nie zadziała …*] zaś funkcja `binsearch` sprawdza, czy word występuje w tablicy.

```
int main()
{
   char word[MAXWORDLEN+1];
   struct key *p;

   while (EOF != getword(word, MAXWORDLEN+1))
     if (isalpha(word[0]))
       if (NULL != (p = binsearch(word,keytab,NKEYS)))
         p->count++;
   for (p = keytab; p < keytab + NKEYS; p++)
     if (p->count)
       printf("%s: %d\n", p->word, p->count);
   return 0;
}
```

```
struct key *
binsearch(char * word, struct key * tab, int n)
{
  int comp;
  struct key *low = &tab[0];
  struct key *high = &tab[n];  /* elem. poza tab! */
  struct key *mid;

  while (low < high) {
    mid = low + (high-low)/2;
    if ((comp = strcmp(word, mid->word)) < 0)
      high = mid;  /* ==> word 'mniejsze' od mid */
    else if (comp > 0) low = mid + 1;
    else return mid;
  }
  return NULL;
}
```

Funkcja `binsearch` zwraca wskaźnik do struktury w tablicy, zawierającej szukane słowo (*zamiast numeru tej struktury!*).

Jeśli słowa word nie ma w tablicy — funkcja zwraca `NULL`.

Ponieważ do elementów tablicy — konsekwentnie — odwołujemy się ***za pomocą wskaźników***, warto zwrócić uwagę na kilka spraw:

- reguły arytmetyki wskaźników pozwalają *odejmować* wskaźniki odwołujące się do elementów tej samej tablicy, ale ***wskaźników nie można dodawać***. Dlatego, wyrażenie:
  ```
  mid = (low+high) / 2;  /* ŹLE! */
  ```
  należy zastąpić: `mid = low + (high-low)/2;`
- Algorytm został zaprojektowany tak, aby nie generował odwołań poza tablicę. (`&tab[-1]` *jest zawsze błędne;* `&tab[n]` *może się pojawić, ale nie wolno go użyć do odwołań pośrednich*)

W funkcji main pojawia się pętla:

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Ponieważ `p` jest *wskaźnikiem do struktury*, odpowiednie przesunięcie wskaźnika (`p++` oraz `keytab+NKEYS`) zostanie wykonane w taki sposób, aby adres *trafił* w kolejny element.
[*Kompilator zna* ***prawdziwy*** *rozmiar struktury …*]

Pozostaje do omówienia funkcja `getword`, którą należy napisać specjalnie dla naszego programu z uwagi na możliwość *sklejania* słów kluczowych z niektórymi znakami.

```
/* getword: wprowadza nastepne slowo */
int getword(char * word, int lim)
{
  int c, getch(void);
  void ungetch(int);
  char *w = word;

  while (isspace(c = getch()))
    ;  /* pomijamy białe znaki */
  if (c != EOF) { /* EOF nie jest "char"! */
    if (c == '\"')
      while (EOF != (c=getchar()) && c != '\"')
        ;  /* pomijamy napisy "w cudzysłowiach" */
    else
      *w++ = c;  /*  <=> *(w++) = c; */
  }
```

```
  if ( !isalpha(c) ) {
    *w = '\0';
    return c;  /* Jesli c to nie litera ==> koniec! */
  }
  /* Kończymy, gdy wczytamy o jeden znak za duzo ... */
  for ( ; --lim > 0; w++)
    if ( !isalnum(*w = getch()) ) {
      /* oddajemy znak, otrzyma go kolejny getch() */
      ungetch(*w);
      break;
    }

  *w = '\0';  /* koniec napisu! */
  return word[0];
}
```

W funkcji `getword` przyjeliśmy założenie, że czytane słowa mogą — *poza pierwszym znakiem* — zawierać także cyfry (wywołanie `isalnum` zamiast `isalpha`).

Chociaż wśród słów kluczowych nie mamy takiego przypadku, takie rozszerzenie niczego nie psuje, a może ułatwi dalszą rozbudowę programu aby np. zliczał użyte w programie *identyfikatory*
[=> ***Ćwiczenie do drzew binarnych!***]

Dalej, w funkcji `getword`, mamy do czynienia z częstą sytuacją przy wczytywaniu znaków ze *strumienia* (tutaj stdin):

> *aby dowiedzieć się, że czytanie należy zakończyć, musimy wczytać o jeden znak za dużo …*

W takiej sytuacji, przydatna jest możliwość ***oddawania*** nieporządanego znaku do `bufora/stosu` (`ungetch`) w taki sposób, aby następne wywołania funkcji wczytującej (`getch`) zebrało znaki z bufora jako pierwsze.

Implementacje takich funkcji zawiera np. populana biblioteka `<curses.h>` dla Linux/UNIX:

http://man7.org/linux/man-pages/man3/ncurses.3x.html

https://www.gnu.org/software/ncurses/ncurses.html

TUTAJ (*korzystając wyłącznie z biblioteki standardowej*) *stos* do buforowania znaków oraz parę obsługujących go funkcji `getch` i `ungetch` zaimplementowano następująco:

```
#define BUFFSIZE 200  /* pojemność bufora */
char buff[BUFFSIZE];
int bufftop = 0;

/* getch: pobiera znak; >>najpierw<< oddany */
int getch(void)
{ return (bufftop > 0) ? buff[--bufftop] : getchar(); }

/* ungetch: oddaje znak >>z powrotem na wejscie<< */
void ungetch(int c)
{
  if (bufftop >= BUFFSIZE)
    fprintf(stderr, "ungetch: too many characters\n");
  else
    buff[bufftop++] = c;
}
```

Przykładowo, zliczanie słów kluczowych w programie do porządkowanie napisów (por. wyklad08, plik: napisy2.c)

```
./a.out < napisy2.c
```

daje następujące wyniki:

```
char: 13
else: 1
for: 1
if: 4
int: 18
return: 5
sizeof: 1
void: 6
while: 2
```

[*UWAGA: Zliczane będą także słowa kluczowe w komentarzach — w przykładach były jednak zawsze ujmowane w cudzysłowy.*]