Poprzedni wykład [*22.11.2022*] :

- Funkcje i struktura programu
- Zasady podziału programu na pliki (*nagłówki, pliki źródłowe, kody pośrednie, plik wykonywalny*)
- Polecenie `make` i pliki `Makefile`
- *Preprocesor języka C*

Co dzieje się z gotowym programem?

1. ***Preprocesor:*** dopisuje za nas fragmenty kodu źródłowego
2. ***Kompilator:*** tłumaczy program na język maszynowy (tworzy *kody pośrednie - „object files”*)
3. ***Linker:*** łączy kody pośrednie (i biblioteki) w plik wykonywalny („*executable file”*)
4. ***System operacyjny (+powłoka):*** uruchamia i kontroluje działanie programu; zapewnia komunikację z pamięcią masową, peryferiami, itp.

Preprocesor języka C *(przypomnienie)*

Preprocesor to narzędzie uruchamiane bezpośrednio przed tłumaczeniem programu na kod maszynowy; umożliwia *mechaniczne generowanie* pewnych fragmentów kodu źródłowego.

Przykładowo, dyrektywy:

```
#include <nazwa1>   #include "nazwa2"
```

powodują ***wstawienie zawartości pliku*** o podanej nazwie.

Z kolei dyrektywa:

```
#define nazwa tekst-zastępujący
```

to tzw. ***makrodefinicja*** (lub *makro*): powoduje, że *każde* pojawienie się słowa *nazwa* zostanie zastąpione przez *tekst-zastępujący*.

Makra z parametrami *(przypomnienie)*

Makra takie jak:

```
#define max(A,B) ( (A)>(B) ? (A) : (B) )
```

wywołuje się podobnie jak funkcje: `x=max(p+q, r+s);`

Makra jednak ***nie są funkcjami*** — *argumenty aktualne* nie są obliczane lecz *mechanicznie podstawiane* w miejscach, gdzie w definicji makra (po prawej stronie) występują *parametry formalne*.

[=> *W definicji makra nie* ***może zabraknąć nawiasów!***]

Rozwijanie argumentów „wewnątrz" napisów:

```
#define xprint(A) printf("Wartość " #A " = %g\n", A)
```

Wywołanie: `xprinf(x/y);` jest równoważne instrukcji:

```
printf("Wartość x/y = %g\n",x/y);
```

Wskaźniki i tablice

Wskaźnik = zmienna, która przechowuje *adres innej zmiennej* w pamięci komputera

Ponieważ *tablice* w C są ***niskopoziomowe*** (*tablica* = ciągły zbiór komórek pamięci) związek ze wskaźnikami jest b. silny:

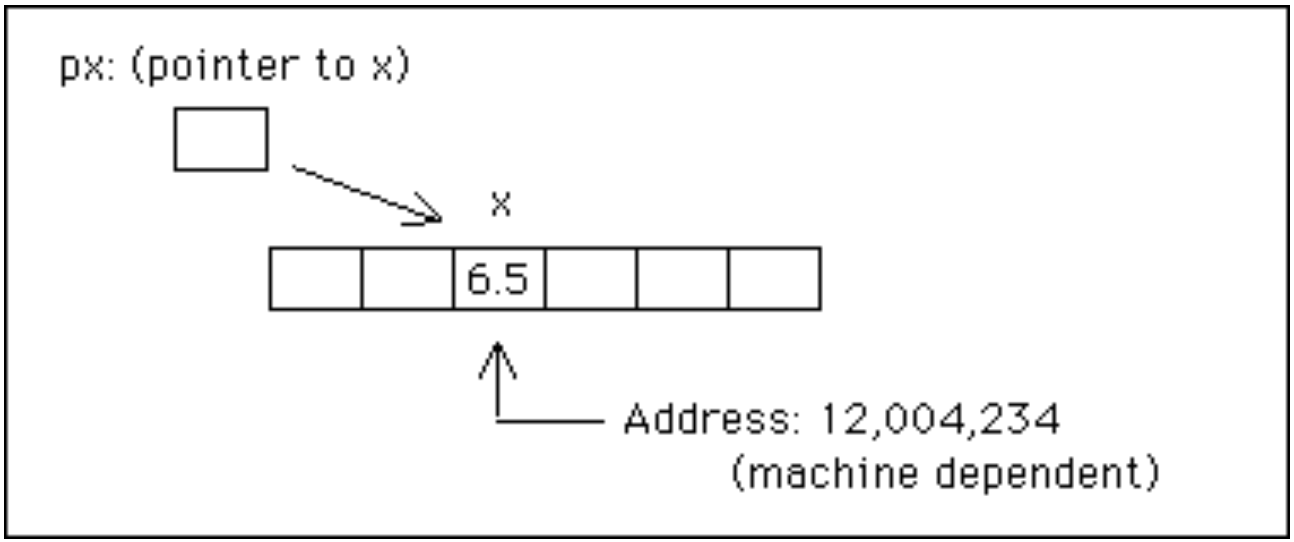
=> ***nazwa tablicy*** jest *adresem* jej *pierwszego elementu*

[*Pewne operacje z zakresu tzw.* ***arytmetyki wskaźników*** *są jednak niedozwolone dla tablic.*]

Jednoargumentowy ***operator adresu*** (*referencji*) pozwala uzyskać adres zmiennej, która jest jego argumentem.

Jeśli np. `x==6.5` jest typu double, zaś `px` to *wskaźnik na double*, wówczas instrukcja: `px = &x;` — przypisuje `px` adres `x`.

Odwrotnie działa ***operator dereferencji*** (*wyłuskania*): `(*px)` będzie liczbą typu double *„wydobytą”* spod adresu `px`.



[***Źródło***: https://www.physics.drexel.edu/~valliere/General/C_basics/c_tutorial.html]

Przykładowo, po zestawie deklaracji:

```
int x=1, y=2, z[10];   /* 2 zmienne + tablica */
int *ip;   /* ip to >>wskaźnik do int<< */
```

poprawne będą *instrukcje*:

```
ip = &x;      /* teraz ip >wskazuje< x*/
y = *ip;      /* y ma wartość 1 */
*ip = 0;      /* x ma wartość 0 */
ip = &z[0];   /* ip wskazuje z[0] */
```

Ogólnie, *deklaracja wskaźnika*, np. `double *px;`
ma formę tzw. ***mnemonika*** — informuje, że *wynikiem dereferencji* (`*px`) będzie liczba typu double. [*Inna forma:* **`double* px;`**]

Podobnie, *prototyp funkcji:* `double atof(char*);`
informuje, że argument to *wskaźnik na char*.

Zasadniczo, *wskaźnik* zawsze *wskazuje* określony rodzaj obiektu; jest zatem powiązany z konkretnym *typem danych*.

Niekiedy wygodnie będzie użyć — np. jako parametru funkcji — „***wskaźnika do*** `void`” który pozwala przechować dowolny rodzaj wskaźnika (*dereferencja* jest wówczas ***niemożliwa!***).

Jeśli (*przykładowo!*) `ip` jest wskaźnikiem na `x` typu `int` wyrażenie `*ip` może wystąpić *wszędzie tam*, gdzie samo `x`:

```
*ip = *ip + 10;  /* zwiększa *ip o 10 */
y = *ip + 1;     /* pobiera *ip, dodaje 1, … */
*ip += 1;        /* zwiększa *ip o 1 */
```

Zwiększenie *wartości wskazywanej* o 1 następuje także w wyrażeniach: `++*p` oraz `(*p)++` [*UWAGA na nawiasy!*]

Dla odmiany, wyrażenie: `*p++` spowoduje *zwiększenie wskaźnika*, zamiast wartości wskazywanej (*po takiej operacji,* `p` *będzie wskazywać na* ***następną*** *komórkę pamięci komputera*).

[=> Operatory jednoargumentowe `*` i ++ *wiążą od prawej do lewej*.]

Wskaźniki to także zmienne; jeśli `iq` oraz `ip` są wskaźnikami na int, instrukcja przypisania:

```
iq = ip;
```

skopiuje adres zapisany w `ip` do zmiennej `iq`.

Wskaźniki jako argumenty funkcji

Przypomienie: W języku C argumenty są *przekazywane przez wartość* — funkcja *de facto* otrzymuje ich kopie.

Próbujemy napisać funkcję swap zamieniającą miejscami dwie liczby całkowite:

```
void swap(int x, int y)   /* ŹLE !!! */
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Poprawiona wersja — operuje na ***wskaźnikach*** do x i y:

```
void swap(int *px, int *py)   /* DOBRZE! */
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Dostęp do zmiennych jest realizowany ***pośrednio*** (poprzez adresy), dlatego zamiana wyłuskanych wartości *dokona się naprawdę*.

Wywołanie funkcji dla a i b — zmiennych całkowitych:

```
swap(&a, &b);  /* podajemy >>adresy<< ! */
```

Podobny mechanizm stosuje się w licznych sytuacjach, gdy funkcja musi zmodyfikować wartość jakiejś zmiennej, a proste zwrócenie wartości (instrukcją return) nie jest możliwe.

Przykładowo, funkcja biblioteczna scanf zwraca liczbę wczytanych elementów (lub kod błędu), zaś samo *wczytanie elementu* dokonuje się za pośrednictwem adresu.

Przykład — **Wczytywanie liczb *„do końca pliku”***:

```
int n, tab[NMAX];
for (n=0; n<NMAX && EOF!=scanf("%d",&tab[n]); n++)
   ;
```

Wczytana *liczba* może być *„równa EOF”* (to pewna stała całkowita), a zatem nie może zostać zwrócona przez funkcje wczytującą.

Związek wskaźników z tablicami

W języku C **wskaźniki** są silnie powiązane z **tablicami** — *każda operacja na tablicy może być wykonana z użyciem wskaźnika.*

Instrukcja:

```
int a[10];
```

definiuje tablicę 10 liczb typu *int*; elementy tablicy mają *nazwy*:

```
a[0], a[1], …, a[9]
```

i są umieszczone w pamięci ***kolejno jeden po drugim***.

Ogólnie: `a[i]` oznacza `i`-ty element tablicy; jeśli tablica zawiera `n` elementów — poprawne są wartości `i` to: `0,1, …, n-1`.

UWAGA: Standard języka C gwarantuje, że odwołanie do elementu bezpośrednio ***za tablicą*** (*tutaj:* `a[10]`) *formalnie nie jest błędem* (**!**)

W języku C nazwa tablicy jest *synonimem* ***adresu pierwszego elementu***; jeśli zadeklarujemy:

```
int *pa;
```

poprawne (***i równoważne!***) będą przypisania:

```
pa = &a[0];
```
oraz
```
pa = a;
```

a później — odwołania do wartości i-tego elementu to:

```
a[i]   *(pa+i)   *(a+i)   pa[i]
```

[*Różne* ***„typy wskaźników”*** *— potrzebne aby wykonać przesunięcie!*]

Różnica pomiędzy tablicą a wskaźnikiem: **Wskaźnik to zmienna**, poprawne zatem będą wyrażenia: `pa=a` `pa++`

[Natomiast: **`a=pa`** oraz **`a++`** **są niepoprawne!**]

Kiedy nazwa tablicy jest ***przekazywana do funkcji*** — funkcja *de facto* otrzymuje adres pierwszego elementu.

Wewnątrz funkcji — wspomniany adres kopiowany jest do *zmiennej lokalnej*, która działa już jak ***zwyczajny wskaźnik.***

Dopuszczalne są zatem operacje *zwiększania* i *zmniejszania*, które nie mają żadnych konsekwencji *na zewnątrz* funkcji.

Przykład — *F-cja obliczająca długość napisu:* [*Kernighan&Ritchie*]

```
int strlen(char s[])  /* <=> int strlen(char *s) */
{
   int n;
   for (n=0; *s != '\0'; s++)  n++;
   return n;
}
```

W funkcji `strlen` deklaracje parametru `(char s[])` oraz `(char *s)` są absolutnie równoważne; w obu wypadkach funkcja otrzymuje ***wskaźnik*** zainicjowany kopią adresu `&s[0]` i operacja zwiększania (`s++`) jest dozwolona.

Wszystkie poniższe wywołania funkcji ***są poprawne***:

```
strlen("Witaj!");  /* dla stałej napisowej  */
strlen(array);    /* np. dla:  char array[100]; */
strlen(ptr);     /* dla: char *ptr; */
```

Poprawne (***i bardzo pożyteczne!***) są także odwołania do części tablicy, np: `strlen(&a[2]);` — wówczas 2 pierwsze elementy tablicy zostaną pominięte. [*Wewnątrz f-cji poprawne będzie:* `a[-7]`]

Komentarz: Zasada przekazywania *de facto* kopii adresu w przypadku, gdy parametrem jest tablica, pozwala zrozumieć działanie funkcji swap w wersji z *poprzednich wykładów*:

```
void swap(int v[], int i, int j)
{
   int temp;
   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

=> *Zmienna lokalna* `v` działa jak wskaźnik, a zatem przypisania, (np. `v[i]=...`) mają realne konsekwencje dla zawartości elementów *zewnętrznej tablicy* podanej jako *pierwszy argument* wywołania.

[*Inaczej będzie np. w przypadku* ***struktury*** *zawierającej 2 elementy!*]

Arytmetyka adresów

Wyrażenia takie jak:

`p++` (*zwiększenie wskaźnika t., aby wskazywał następny element*),

`p--` (*zmiejszenie wskaźnika t., aby wskazywał poprzedni element*)

stanowią przykłady tzw. ***arytmetyki wskaźników***.

Wskaźniki** w języku C* to jednak ***nie liczby całkowite — a zatem nie wszystkie operacje arytmetyczne są dozwolone.

Do wskaźnika można ***dodać*** (lub od niego ***odjąć***) *liczbę całkowitą*: wyrażenie `p+n` (`p-n`) oznacza adres `n`-tego elementu *za* (*przed*) elementem wskazywanym przez p. (*Zależnie od typu wskaźnika, liczba typu **int** zostanie odpowiednio przeskalowana.*)

[***Nie można** jednak dodać dwóch wskaźników!*]

Jeśli `p` i `q` wskazują na ***elementy tej samej tablicy***, poprawne będą wyrażenia *porównania*:

`p == q` `p != q` `p < q` `p >= q` itp.

W sytuacji *j.w.*, poprawne jest także ***odejmowanie wskaźników***:

Jeśli `p<q` (oraz `p` i `q` nadal wskazują na *elementy tej samej tablicy*) wyrażenie: `q-p+1` jest ***liczbą elementów*** od *wskazywanego przez* `p` do *wskazywanego przez* `q` (włącznie).

Przykładowo, po przypisaniach: `p=&a[i];  q=&a[j];`
element pośrodku możemy znaleźć jako wartość wyrażenia :

```
*(p+(q-p)/2)    /* inaczej: a[(i+j)/2] */
```

[*„prostsza" operacja* `*((p+q)/2)` *byłaby* ***niepoprawna!***]

Zaawansowana wersja funkcji strlen może wyglądać tak:

```
int strlen(char *s)
{
   char *p=s;     /*  <=>  char *p;  p=s; */
   while (*p != '\0')
      p++;
   return p-s;
}
```

Po definicji, `p` wskazuje na pierwszy znak napisu `s`.

W każdym obiegu pętli, `p` jest przesuwane do następnego znaku i sprawdza się, czy tym znakiem nie jest `'\0'`.

Wartość `p-s` to długość napisu (*nie licząc znaku* `'\0'`).

[***Inna*** *(poprawna) forma definicji wskaźnika* `p`: **`char* p=s;`**]

Operacje arytmetyki adresów: *Podsumowanie*

Lista poprawnych operacji na wskaźnikach

- przypisanie wskaźników ***tego samego typu*** (*lub* `void*`)
- dodawanie/odejmowanie wskaźnika i ***liczby całkowitej***
- odejmowanie i porównywanie wskaźników wyłącznie w ***obrębie tej samej tablicy***
- przypisanie *zera* i porównania z nim (*zwykle zastępujemy:* NULL)

Inne operacje są niedozwolone! *W szczególności, wskaźników nie możemy dodawać, dzielić ani mnożyć; jak również dodawać do nich (lub przypisywać im…) liczb zmiennopozycyjnych.*

Funkcje operujące na *wskaźnikach znakowych*

Każda ***stała napisowa***, np. `"Jestem napisem!"`

jest reprezentowana w pamięci komputera jako *tablica znakowa*, uzupełniona znakiem `'\0'` na końcu. [*Ten dodatek pozwala różnym funkcjom użytkowym łatwo znajdować* ***koniec napisu****.*]

Pierwszy element takiej tablicy *ma swój adres*, który może być przekazany do funkcji (lub: *przypisany zmiennej typu* `char *`).

Poprawne będzie zatem wywołanie funkcji:

```
strlen("Ja też jestem napisem")
```

jak również deklaracja:

```
char *ptext = "Lorem ipsum dolor sit amet";
```

Istnieje jednak ***pewna różnica*** pomiędzy definicjami:

```
char atext[] = "Lorem ipsum"; /* tablica */
char *ptext = "dolor sit amet"; /* wskaźnik */
```

W pierwszym przypadku, tworzona jest ***tablica znakowa***,
o długości dopasowanej do inicjującej *stałej napisowej (11 znaków)*;
wszystkie znaki w tablicy `atext` mogą być później zmieniane.

W drugim przypadku, `ptext` jest ***wskaźnikiem*** wskazującym na *stałą napisową;* o ile sam wskaźnik może być modyfikowany, standard języka nie określa co się stanie, jeśli spróbujemy modyfikować znaki tworzące napis.

[*W jęz. C nie istnieją żadne operatory przetwarzające* ***całe napisy.***]

Przykład: Funkcja kopiująca `t` do `s`; *wersja tablicowa*

[*wg Kernighan & Ritchie*]

```
void strcpy(char *s, char *t)
{
   int i;

   i = 0;
   while ((s[i] = t[i]) != '\0')
      i++;
}
```

[*Poprawnie zainicjowana tablica znakowa zawiera przynajmniej* `'\0'`!]

UWAGA: W tej wersji, po osiągnięciu końca napisu (`t[i] == '\0'` *) indeks* `i` *nie zostanie już zwiększony.*

Taka sama funkcja; *wersja wskaźnikowa*

```
void strcpy(char *s, char *t)
{
   while ((*s = *t) != '\0') {
      s++;
      t++;
   }
}
```

=> Argumenty `t` i `s` są przekazywane przez wartość, a zatem operacje: **`s++`** i **`t++`** dotyczą w istocie ***ich kopii*** (!)

Kopiowanie napisów: *Wersja wskaźnikowa—zaawansowana*

```
void strcpy(char *s, char *t)
{
   while ((*s++ = *t++) != '\0')
      ;
}
```

Objaśnienie: Operatory zwiększania (++) w wersji ***przyrostkowej*** działają **po** zakończeniu wszystkich obliczeń, zatem przypisanie i porównanie zostaną wykonane *przed* modyfikacją wskaźników.

[**Uwaga**: Tym razem, po wystąpieniu `'\0'` oba wskaźniki zostaną *jeszcze raz* zwiększone, zatem pokażą na *znak poza napisem*.]

Jeszcze krótsza wersja funkcji kopiującej napisy:

```
void strcpy(char *s, char *t)
     { while (*s++ = *t++); }
```

=> Tutaj korzystamy *z idiomu*: znak pusty `'\0'` to zawsze zero, zatem jawne porównanie nie jest potrzebne.

[*Podobnie, `NULL` to zawsze zero, ale już np. `EOF` to zwykle –1.*]

Funkcja `strcpy` jest zdefiniowana w ***bibliotece standardowej*** (nagłówek: `<string.h>`); jej *prototyp* wygląda tak:

```
char *
strcpy(char * dst, const char * src);
```

Funkcja zwraca wskaźnik to tekstu docelowego (`dst`), deklaracja drugiego parametru jako `const char *` daje pewność, że napis źródłowy (`src`) nie będzie zmieniony.

Inna (*prosta)* funkcja pozwala na ***porównanie napisów***:

```
int strcmp(char *s, char *t)  /* z tablicami */
{
   int i;
   for (i = 0; s[i] == t[i]; i++)
      if (s[i] == '\0')
         return 0;
   return s[i] - t[i];
}
```

Funkcja zwraca ***zero***, jeśli napisy wskazywane przez `s` i `t` są identyczne, ***wartość dodatnią***, jeśli `s` jest *leksykalnie większe* od `t`, lub ***wartość ujemną*** — w przeciwnym przypadku.

Wersja wskaźnikowa funkcji porównującej napisy:

```
int strcmp(char *s, char *t)  /* bez tablic */
{
   for ( ; *s == *t; s++, t++)
      if (*s == '\0')
         return 0;
   return *s - *t;
}
```

[*Warunek w pętli można jeszcze zastąpić przez:* `if (!*s)`
— będzie to jednak raczej mało czytelne …]

Wskaźnik `FILE *` i funkcja `fopen`

Operacje na plikach (innych niż *standardowe wejście* lub *wyjście*) realizujemy za pomocą wskaźników plikowych. Potrzebne deklaracje i prototypy funkcji zawiera nagłówek `<stdio.h>`.

Przykładowo, po deklaracji: `FILE *fp;`

instrukcja: `fp = fopen("mojplik.txt","w");`

otworzy plik (`"mojplik.txt"`) w ***trybie zapisu*** (`"w"`) i powiąże go z `fp`. W takim przypadku, *plik nieistniejący* zostanie utworzony, *istniejący* — zamazany. [Oba paramery **`fopen`** są typu: **`char *`**]

Inne ***tryby otwarcia*** to: `"r"` — *odczyt* oraz `"a"` — *dopisywanie*.

[*Istnieją są także tryby:* **`"r+"`** **`"w+"`** oraz **`"a+"`** pozwalające na czytanie i pisanie po *tym samym pliku*.]

Większość systemów rozróżnia ***pliki tekstowe*** i ***binarne***; domyślne są tekstowe, otwarcie pliku binarnego wymaga dodania litery `b` w napisie oznaczającym tryb otwarcia (`"wb"` `"rb"` `"r+b"` itp).

W każdym przypadku, aby zmiany zostały zachowane, konieczne jest zamknięcie pliku, np.:

```
fclose(fp);
```

— wywołanie `fclose` zarazem ***zwalnia wskaźnik*** (`fp`), który można następnie *powiązać* z innym plikiem wywołując `fopen`.

Inne funkcje, użyteczne (zwłaszcza w trybach *„z plusem”*), to np.

`rewind(fp)` — powraca ***do początku pliku***

`fflush(fp)` — ***opróżnia bufor*** (dla strumienia *wyjściowego*)

Ponadto, instrukcja: `fp = tmpfile();` tworzy *plik tymczasowy* (w trybie `"wb+"`), który będzie usunięty po zamknięciu.

W przypadku ***plików tekstowych***, często używamy funkcji: formatowanego wejścia/wyjścia (z nagłówka `<stdio.h>`):

```
int fprintf(FILE * stream, const char * format, …);
int fscanf(FILE * stream, const char * format, …);
```

Są one bardzo podobne do `printf` i `scanf` z tym, że pierwszym argumentem jest zawsze *wskaźnik plikowy*.

Dla ***plików binarnych***, elementarz stanowią ~~funkcje~~ (*makra!*):

`int getc(FILE *fp)` oraz `int putc(int c, FILE *fp)`

które wywołane tak: `getc(stdin)` lub `putc(c,stdout)` zadziałają identycznie jak `getchar()` i `putchar(c)`.

`stdin` i **`stdout`** to także wskaźniki typu `FILE *`, ***są to jednak stałe*** — trwale powiązane ze *standardowym wejściem* i *wyjściem*.

Przykład — funkcja kopiująca plik `ifp` do `ofp`:
[*wg Kernighan & Ritchie*]

```
void filecopy(FILE *ifp, FILE *ofp)
{
   int c;

   while (EOF != (c = getc(ifp)))
      putc(c, ofp);
}
```

[*Inaczej niż przy kopiowaniu napisów - tym razem* ***nie przepisujemy znacznika końca pliku*** — `EOF` *zostanie wstawiony przez system operacyjny podczas zamykania pliku lub po prawidłowym zakończeniu działania programu.*]

W poniższym programie, funkcja `filecopy` jest użyta do ***kopiowania danych*** ze *standardowego wejścia* na *wyjście*:

```
#include <stdio.h>
int main()
{
   void filecopy(FILE *, FILE *);
   filecopy(stdin, stdout);
}
```

=> *Ćwiczenie: Łatwo przekonać się, że mechanizm potoków unixowych pozwala na kopiowanie **dowolnych** plików, także binarnych:*

```
./a.out < plik1 > plik2
```

Możemy też wykorzystać `filecopy` do ***wyświetlenia dowolnego pliku*** na *standardowe wyjście*:

```
#include <stdio.h>
#define MAXNAM 100
int main()
{
   char fnam[MAXNAM];
   FILE *fp;
   void filecopy(FILE *, FILE *);
   printf("==> Podaj nazwę pliku: ");
   scanf("%s",fnam);
   if (NULL == (fp=fopen(fnam,"rb")) ) {
      printf("Nie znaleziono pliku: %s\n",fnam);
      return -1;
   }
   filecopy(fp, stdout);
   fclose(fp);
}
```

Funkcje o zmiennej liczbie argumentów

Język C umożliwia definiowanie funkcji, które mogą być wywołane z większą liczbą ***argumentów***, niż wynosi liczba ***parametrów*** podanych w definicji funkcji.

Jest to możliwe, jeśli lista parametrów kończy się wielokropkiem (**...**); nadliczbowe argumenty obsługiwane są za pomocą makra `va_arg` zdefiniowanego w nagłówku standardowym `<stdarg.h>`

Mechanizm działania makra `va_arg` zilustrujemy teraz na przykładzie prostej funkcji obliczającej średnią arytmetyczną swoich argumentów.

```c
#include <stdio.h>      /* Zob.: tutorialspoint.com */
#include <stdarg.h>

double average(int narg, ...)
{
  va_list list;
  double sum = 0.0;
  int i;

  va_start(list, narg);  /* tworzy listę argumentów */

  for (i = 0; i < narg; i++)
    sum += va_arg(list, double); /* zdejmuje argument */

  va_end(list);  /* kasuje listę argumentów */

  return sum/narg;
}
```

/* ==> **c.d.n** */

```
int main()
{
   printf("Average of 2, 3, 4, 5 = %lf \n",
      average( 4,  2.0, 3.0, 4.0, 5.0 ));

   printf("Average of 5, 10, 15 = %lf \n",
      average( 3,  5.0, 10.0, 15.0 ));

   return 0;
}
```

Uwaga: Aby mechanizm działał, argumenty obowiązkowe muszą zawierać informacje o liczbie pozostałych. Tutaj — mamy `narg`; w przypadku `printf` / `scanf` — mamy do policzenia znaki `'%'` w napisie będącym pierwszym argumentem; zaś dla `fprintf` / `fscanf` — informacja ta zawarta jest w ***drugim*** argumencie.