Poprzedni wykład [*14.12.2021*]:

- Operacje plikowe; *pliki binarne* i *tekstowe*
- Wejście/wyjście: znakowe, wierszowe, formatowane, i niskopoziomowe (fukcje: `fread` / `fwrite`)
- Dynamiczny przydział pamięci (`malloc` / `calloc`)
- ***Tablice wskaźników***
- Wskaźniki na wskaźniki i tablice wielowymiarowe; związek tablic wielowymiarowych z tablicami wskaźników

Argumenty wiersza poleceń

Systemy operacyjne, w których uruchamiamy skompilowane programy w języku C, umożliwiają przekazywanie parametrów z wiersza poleceń do uruchamianego programu.

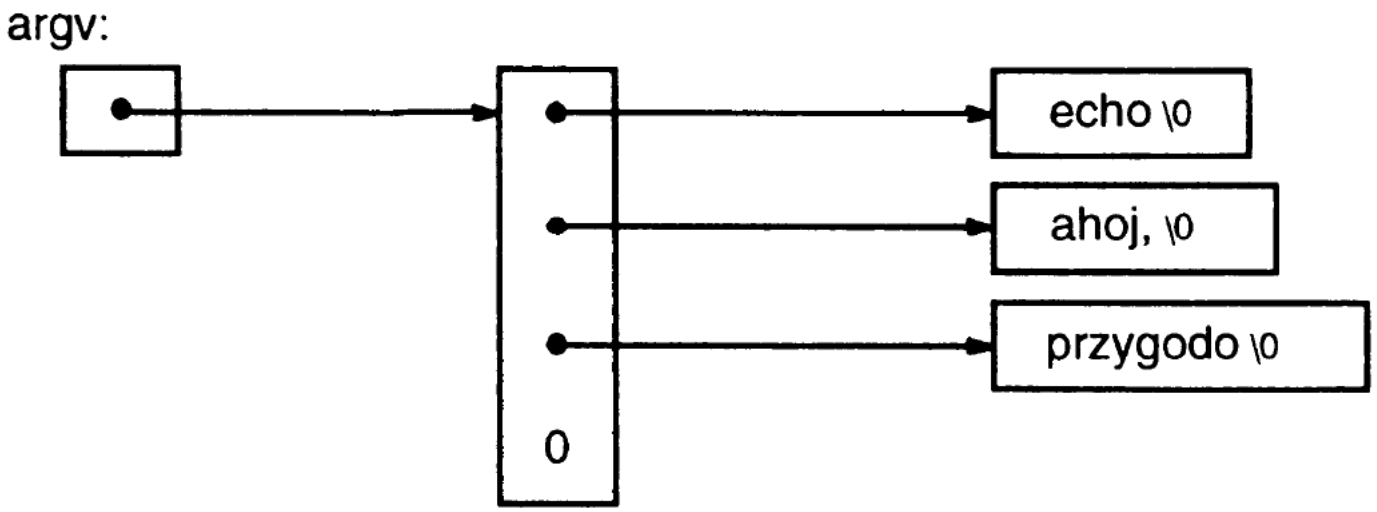
W chwili wywołania, funkcja `main` otrzymuje ***2 argumenty***:

- *pierwszy* (typu int) to liczba parametrów wywołania, zwyczajowo nazywana `narg` (ang. *number of arguments*)
- *drugi* — zwyczajowo `argv` (ang. *argument vector*) to tablica wskaźników znakowych, której kolejne komórki przechowują *adresy napisów* zawierających poszczególne argumenty wpisane w wierszu poleceń [*jak również* ***nazwę programu*** *— na początku i* ***wskaźnik pusty*** *— na końcu*].

Przykładowo, program ”echo”, który po wypisuje argumenty z wiersza poleceń na standardowe wyjście, po wywołaniu:

```
$ echo ahoj, przygodo
```

otrzyma argument `narg` o wartości `3` oraz tablicę z elementami `argv[0]` … `argv[2]` *wskazującymi na następujące napisy*:



[`argv[narg]` zawiera zawsze ***wskaźnik pusty:*** **`NULL`** <=> *synonim 0*.]

Pierwsza wersja: traktujemy `argv` jako tablicę wskaźników:

```
#include <stdio.h>

/* echo argumentów wywołania: Wersja 1 */
int main(int narg, char *argv[])
{
   int i;
   for (i = 1; i < narg; i++)
      printf("%s%s", argv[i], (i<narg-1)?" ":"");
   printf("\n");
   return 0;
}
```

Druga wersja (zaawansowana) — `argv` to także synonim *wskaźnika na wskaźnik znakowy*:

```
#include <stdio.h>

/* echo argumentów wywołania: Wersja 2 */
int main(int narg, char **argv)
{
   while (--narg > 0)
      printf("%s%s", *++argv, (narg>1)?" ":"");
   printf("\n");
   return 0;
}
```

Bezpośredniu po uruchomieniu programu (<=> wywołaniu funkcji `main`) wartość `argv` wskazuje na pierwszy element (`argv[0]`) w tablicy wskaźników (powiązany z napisem: `"echo"`).

Operacja *zwiększenia wskaźnika o 1* (`++argv`) przesuwa wskaźnik z elementu `argv[0]` do elementu `argv[1]`.

W każdym kolejnym powtórzeniu pętli `while` przesuwamy zatem wskaźnik `argv` do kolejnego *napisu / argumentu-wiersza-poleceń*, (dereferencja `*argv` zwróci wskaźnik do *pierwszego znaku* wspomnianego napisu).

Jednocześnie — w każdym powtórzeniu pętli — zmniejszamy wartość `narg` o 1, dopóki jest ona (*po zmniejszeniu!*) > 0.

[*Tym sposobem, wypisanych zostanie* `narg-1` *argumentów, kolejno od* `argv[1]` *do* `argv[narg-1]`*; po każdym za wyjątkiem ostatniego dodatkowo pojawi się spacja. Białe znaki są pomijane.*]

Instrukcja wypisująca argumenty i spacje może jeszcze zostać zastąpiona nieco krótszą wersją:

```
printf((narg>1) ? "%s " : "%s", *++argv);
```

gdzie korzystamy z faktu, że pierwszy argument funkcji `printf` (zawierający *specyfikacje przekształcenia*) również może być — ***jak każdy argument każdej funkcji*** — wyrażeniem wymagającym wykonania obliczeń.

W kolejnym przykładzie, rozważymy program będący uproszczoną wersją polecenia `cat` systemu Unix. W szczególności,

```
$ cat a.c b.c
```

wypisuje na standardowe wyjście zawartość plików `a.c` i `b.c`.

[Zatem aby np. *dokleić* `b.c` do `a.c`, piszemy: **`$ cat a.c b.c > a.c`**]

```
#include <stdio.h>
#include <stdlib.h>  /* zob. funkcja exit */

/* Sklejanie plików - wg Kernighan&Ritchie, 1994 */
int main(int narg, char *argv[])
{
   FILE *fp;
   void filecopy(FILE *, FILE *);  /* kopiuje plik */
   char *prog = argv[0];  /* nazwa programu */

   if ( narg == 1 )  /* kopiuj standardowe wejście */
      filecopy(stdin, stdout);
   else {  /* c.d.n. */
```

```
      /* c.d. "else" <=> (narg > 1) */
      while (--narg>0)
         if (NULL==(fp=fopen(*++argv,"rb"))) {
            fprintf(stderr, "%s: cannot open %s\n",
                    prog, *argv);
            exit(1);
         } else {  /* przypadek bez błędów: */
            filecopy(fp, stdout);
            fclose(fp);
         }
   }
   if (ferror(stdout)) {
      fprintf(stderr, "%s: stdout error\n", prog);
      exit(2);
   }
   exit(0);
}
```

Ponownie, użyliśmy *strumienia błędów* (`stderr`) aby komunikaty o błędach trafiały ***zawsze na ekran*** (*również w przypadku, gdy wyjście programu jest przekierowane do pliku*).

Pierwszym nowym elementem jest funkcja standardowa `exit`:

Wewnętrz funkcji `main` instrukcja `exit(`*wyrażenie*`);`

jest równoważna instrukcji: `return` *wyrażenie*;

ma jednak tę przewagę, że pozwala *natychmiast przerwać* działanie programu, ***także w środku wykonywania innej funkcji***.

[**Argument funkcji `exit`** zostanie przekazany do procesu, który wywołał program; funkcja `exit` automatycznie wywołuje `fclose` dla wszystkich otwartych plików i wypisuje dane w buforach.]

Drugi nowy element to funkcja `ferror` zadeklarowana jako:

```
int ferror(FILE *fp);
```

Funkcja `ferror` zwraca wartość niezerową, gdy dla strumienia `fp` wystąpi jakiś błąd. [*Podobna funkcja:* `int feof(FILE *)`]

[*Błędy dotyczące standardowego wyjścia* są raczej rzadkie (np. *przepełnienie partycji dysku*), w poważnym programie musimy jednak zadbać o zwracanie wartości opisującej taki stan.]

Ostatni element — ***funkcja kopiująca pliki*** (*zob.* wyklad07.pdf):

```
void filecopy(FILE *ifp, FILE *ofp)
{
   int c;
   while (EOF != (c=getc(ifp)))  putc(c, ofp);
}
```

Wygodna obsługa błędów

W *bibliotece standardowej* C mamy nagłówek `<assert.h>`, w którym zdefiniowano użyteczne makro:

```
void assert(int expression);
```

Jeśli argument `expression == 0` (wskaźnik NULL będzie automatycznie zrzutowany, podobnie znak pusty `'\0'`) funkcja assert wypisuje komunikat o błędzie i przerywa działanie programu. Jeśli `expression != 0` — nic się nie dzieje.

Typowy komunikat o błędzie wygląda tak:

```
prog: some_file.c:16: some_func: Assertion `val == 0'
failed.
```

UWAGA o maskach w systemie Unix:

Polecenia systemowe, jeśli to tylko możliwe — przyjmują jako argumenty tzw. *maski*. Przykładowo,

```
$ cat *.c
```

wyświetli zawartość wszystkich plików (*w bieżącym katalogu*), których nazwy kończą się wzorcem ”.c”, zaś

```
$ cat *.c > all-souce-files.c
```

sklei zawartość tych plików i zapisze w jednym pliku.

Nasz program oczywiście *nie będzie rozwijał masek automatycznie*, możemy jednak napisać:

```
$ ./a.out `ls *.c`
```

Wówczas, system operacyjny *przekaże wynik polecenia* (`ls *.c`) jako listę argumentów do naszego programu.

Backquote (`) to nie apostrof (') !!!

Trzecim przykładem, ilustrującym zastosowanie *argumentów wiersza poleceń*, będzie ulepszona wersja programu wypisującego *wiersze zawierające szukany wzorzec* (*por.* wyklad05.pdf).

Tym razem, podążymy za schematem polecenia `grep` systemu Unix — szukany wzorzec będzie ***pierwszym argumentem*** wywołania programu.

Dla dalszego *usprawnienia działania* programu, użyjemy funkcji biblioteki standardowej: `strstr(s,t)` zadeklarowanej w nagłówku `<string.h>`, która zwraca *wskaźnik do pierwszego wystąpienia* ciągu znaków `t` w napisie `s` lub NULL, jeśli szukany wzorzec (`t`) nie został znaleziony w napisie (`s`).

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

/*   ==> Wypisz wiersze zawierające wzorzec, podany
        jako argument wywołania programu <==    */
int main(int narg, char *argv[])
{
   char line[MAXLINE+1];  /* + miejsce na '\0' */
   int found = 0;

   if (narg != 2)
      fprintf(stderr,"USAGE: %s pattern\n",argv[0]);
   else { /* c.d.n. */
```

```
    /* c.d. "else" <=> (narg == 2) */

    while (NULL != fgets(line, MAXLINE+1, stdin))
      if (NULL != strstr(line, argv[1])) {
        printf("%s", line);
        /* '\n' jest już zawarte w line (!) */
        found++;
      }
  }
  return found;
}
```

Argumenty *opcjonalne* wywołania programu

Polecenia systemu *Unix* najczęściej posiadają parametry (lub *przełączniki*) opcjonalne, podawane po znaku `'-'`.

Przykładowo,

```
$ ls *.c
```

wypisze tylko nazwy plików pasujących do *maski* `"*.c"`, zaś

```
$ ls -l *.c
```

wypisze szczegółowe informacje o tych plikach:

```
-rw-r--r--+ 1 adamr  staff  145 Nov 14 19:42 co-robi.c
-rw-r--r--+ 1 adamr  staff  202 Oct 22 00:40 silnia.c
-rw-r--r--  1 adamr  staff  820 Dec 16 00:53 sklej.c
-rw-r--r--+ 1 adamr  staff  822 Nov 14 13:11 wzorzec.c
```

[***W maskach***: `*` = dowolny ciąg (*nie-białych*) znaków; `?` = 1 znak.]

Pokażemy teraz, jak zmodyfikować *program wyszukujący wzorzec* tak, aby posiadał dwa opcjonalne przełączniki:

- `-x` (ang. *eXcept*) powoduje, że program wypisuje wszystkie wiersze ***oprócz*** tych zawierających wzorzec.
- `-n` (ang. *numbering*) podaje numery wypisywanych wierszy

Chcemy, aby przełączniki można było podawać ***w dowolnej kolejności***; użyteczną możliwością jest także ***grupowanie opcji***:

```
$ ./a.out -nx wzorzec
```

=> powinny zostać wypisane wiersze *niezawierające wzorca*, poprzedzone numerami.

[*Naturalnie, program wywołany bez opcji powinien działać jak poprzednio, tj. wypisywać wiersze zawierające wzorzec, bez numeracji.*]

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

/*   ==> Wypisz wiersze zawierające wzorzec, podany
        jako argument wywołania programu <==    */
int main(int narg, char *argv[])
{
   char *prog = argv[0];  /* nazwa programu */
   char line[MAXLINE+1];
   long lineno = 0;  /* numer wiersza */
   int c, exc = 0, num = 0, found = 0;

/* c.d.n. */
```

```
while (--narg > 0 && (*++argv)[0] == '-')
   while ((c = *++argv[0]))  /* <=> c!='\0' */
      switch (c) {
      case 'x':
         exc = 1; break;
      case 'n':
         num = 1; break;
      default:
         fprintf(stderr, "%s: illegal option "
                 "-%c\n", prog, c);
         narg = 0;
         found = -1;
         break;
      }
```

```
   if (narg != 1)
      fprintf(stderr, "USAGE: %s -x -n pattern\n”,
              prog);
   else
      while (NULL != fgets(line,MAXLINE+1,stdin)) {
         lineno++;
         if ( exc != (NULL!=strstr(line,*argv)) ) {
            if (num) printf("%8ld: ", lineno);
            printf("%s", line);
            found++;
         }
      }
   return found;
}
```

Z każdym powtórzeniem pierwszej pętli `while`, wartość `narg` zostaje *zmiejszona o 1*, a wartość `argv` *zwiększona o 1*.

Jeśli nie wystąpiły błędy, na końcu trafiamy na argument, którego pierwszy znak (`*argv[0]` lub `**argv`) nie jest już znakiem `'-'`
A zatem — jest to argument „obowiązkowy” (nasz *wzorzec*), reprezentowany przez `argv[0]` (lub `*argv`).

Warto podkreślić, że nawiasy w wyrażeniu `(*++argv)[0]` są konieczne, ponieważ `[]` wiąże mocniej niż ++, a zatem bez nawiasów, de facto dostalibyśmy: `*++(argv[0])`

Korzystamy z tego w drugiej pętli `while`, która przegląda kolejne znaki *napisu* `argv[0]` — zwiększamy zatem wskaźnik `argv[0]`, natomiast `argv` pozostaje niezmieniony (!)

Przykładowo, wywołanie programu:

```
$ ./a.out -n int\ < wzorzec2.c
```

wyszuka wzorzec `"int "` (spację wprowadzamy pisząc `'\ '`)
w pliku źródłowym tego programu (`wzorzec2.c`).

Wynikiem będzie:

```
  8: int main(int narg, char *argv[])
 12:   int c, exc = 0, num = 0, found = 0;
```

Otrzymaliśmy zatem linie kodu źródłowego, zawierające deklaracje obiektów typu `int`.

Wskaźniki do funkcji

Funkcje w języku C *nie są zmiennymi*, można jednak definiować ***wskaźniki do funkcji*** (*taki wskaźnik przechowuje adres fragmentu kodu wykonywalnego, który jest realizacją danej funkcji*).

Wskaźniki do funkcji już *są zmiennymi*, mogą być zatem ***przypisywane***, ***grupowane w tablicach***, ***przekazywane do funkcji***, jak również ***zwracane przez funkcje***.
[*Nie istnieje jednak żadna „arytmetyka wskaźników do funkcji”, nie można również przydzielić/zwolnić pamięci za ich pośrednictwem.*]

Deklaracja wskaźnika do funkcji (*jako zmiennej!*) — wg szablonu:

```
typ-powrotu (*nazwa-wskaźnika)(typy-argumentów);
```

dodatkowe nawiasy otaczające *nazwę* są konieczne, bez nich zadeklarowalibyśby po prostu *funkcję zwracającą wskaźnik*.

Przykładowo, jeśli gdzieś w programie zdefiniujemy funkcję:

```
void fun(int a)
{
   printf("Moja ulubiona liczba to %d\n", a);
}
```

a następnie w funkcji main pojawi się deklaracja:

```
void (*fun_ptr)(int) = &fun;
```

wówczas, naszą funkcje możemy wywołać na dwa (*równoważne!*) sposoby: po pierwsze, ***wprost***:

```
fun(7);
```

lub też ***przez wskaźnik***:

```
(*fun_ptr)(7);
```

Podobnie (*trochę podobnie …*) jak w przypadku tablic, nazwa funkcji jest także ***synonimem wskaźnika*** do funkcji.

[*Konwencja jest tutaj* ***trochę mniej przejrzysta****: W przypadku tablicy, nazwa była synonimem wskaźnika do pierwszego elementu, a nie do całej tablicy; TUTAJ brak takiego rozróżnienia …*]

Dla naszej funkcji (`fun`) poprawna będzie także deklaracja wskaźnika do funkcji:

```
void (*fun_ptr)(int) = fun;  /* usunięto & (!)*/
```

i wywołanie *przez wskaźnik* w postaci:

```
fun_ptr(10);  /* usunięto operator * (!) */
```

Jak widać, użyte poprzednio operatory `&` i `*` nie są konieczne; wybór konwencji — jak zawsze — jest kwestią indywidualnych upodobań.

Tablice wskaźników funkcji

W analogii do zwykłych wskaźników, wskaźniki na funkcje także mogą być ***grupowane w tablice***.

W przykładzie poniżej, wybieramy jedno z 3 działań arytmetycznych na liczbach a i b (*dodawanie, odejmowanie,* lub *mnożenia*).
[zob. https://www.geeksforgeeks.org/function-pointer-in-c/]

```
#include <stdio.h>
void add(int a, int b)
{ printf("Addition is %d\n", a+b); }

void subtract(int a, int b)
{ printf("Subtraction is %d\n", a-b); }

void multiply(int a, int b)
{ printf("Multiplication is %d\n", a*b); }
```

```c
int main()
{
    /* Tablica wskaźników funkcji: fun_ptr_arr */
    void (*fun_ptr_arr[])(int, int) =
                        {add, subtract, multiply};
    int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, "
           "1 for subtract and 2 for multiply\n");
    scanf("%d", &ch);

    if (ch < 0 || ch > 2) return 1;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

Wskaźniki do funkcji jako argumenty funkcji

Podobnie jak *zwykłe wskaźniki*, także *wskaźniki do funkcji* mogą być ***przekazywane jako argumenty*** lub ***zwracane*** przez funkcje.

Taki mechanizm w wielu sytuacjach pozwala uniknąć niekorzystnego zjawiska tzw. ***redundancji kodu***.

Redundancja kodu polega na tym, że *skomplikowana* procedura (np. minimalizacja funkcji, poszukiwanie miejsce zerowych, albo — *znany z poprzednich wykładów* — algorytm `quicksort`) musi zostać przepisana w *kilku, niemal identycznych wersjach*, wyłącznie po to aby np. *obliczać wartości różnych funkcji* do minimalizacji, albo wywoływać *funkcje porównujące* obiekty do posortowania.

W takich sytuacjach, zdecydowanie lepiej jest napisać jedną funkcję, wykonującą naszą procedurę, której parametrem jest *wskaźnik na funkcję* (np. porównującą 2 obiekty).

[*Taką funkcję możemy poźniej* ***wywołać kilkakrotnie****, przekazując jej np. wskaźniki do różnych funkcji porównujących.*]

Potrzeba unikania redundancji kodu wynika wprost z zasady:

==> ***„Pisanie programu nigdy się nie kończy.”***

Kiedy zajdzie potrzeba *ulepszenia* implementacji kluczowego algorytmu — zmiany będziemy wprowadzać w jednej wersji, a nie w kilku podobnych.

Elementarny przykład: Funkcja `wrapper()` operuje na argumencie `void fun()` — wskaźniku powiązanym kolejno z dwiema różnymi funkcjami zewnętrznymi.
[*zob.* https://www.geeksforgeeks.org/function-pointer-in-c/]

```
#include <stdio.h>
void fun1() { printf("I am fun1\n"); }
void fun2() { printf("I am fun2\n"); }

/* Funkcja wywołująca inne f-cje przez wskaźnik */
void wrapper(void (*fun)()) { fun(); }

int main()
{   wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

Inny (***równoważny!***) zapis funkcji `wrapper()` oraz `main()` [*niekorzystający z synonimów wskaźników do funkcji*]:

```
void wrapper(void (*fun)())
{
  (*fun)();
}
```

```
int main()
{
   wrapper( &fun1 );
   wrapper( &fun2 );
   return 0;
}
```

Jeśli zachodzi potrzeba, aby *funkcja przekazywana przez wskaźnik* do innej funkcji także miała swoje argumenty, często możemy ją tak zaprojektować, aby argumentami były ***wskaźniki ogólne*** (`void*`).

Mechanizm ten wykorzystuje m.in. funkcja `qsort` z biblioteki standardowej, zadeklarowana w nagłówku `<stdlib.h>` w następujący sposób:

```
void
qsort(void *base, size_t nel, size_t width,
      int (*compar)(const void *, const void *));
```

Argumentami są: *tablica do posortowania* (`base`), *liczba elementów* (`nel`), *rozmiar 1 elementu* (`width`), oraz wskaźnik do funkcji *wykonującej porównanie* 2 elementów (`compar`).

Przykład: sortujemy tablicę liczb typu `double`

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

int comp(const void *a, const void *b)
{
  double x = *(double *)a, y = *(double *)b;

  if ( x>y ) return 1;
  else if (x<y) return -1;
  else return 0;
}
```

```
int main()
{
  double arr[] = {10.0, 10.0/2, 10.0/3, 3*4, 9};
  int i, n = sizeof(arr)/sizeof(arr[0]);

  qsort(arr, n, sizeof(double), comp);

  for (i=0; i<n; i++)
    printf((i < n-1) ? "%g " : "%g\n", arr[i]);
  return 0;
}
```

Struktury

Ważnym *typem danych* w języku C są struktury: ***struktura*** to *jedna lub kilka zmiennych*, które *mogą być różnych typów*, połączonych w tak, aby możliwe były odwołania za pomocą ***wspólnej nazwy***.

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

```
struct rekord {
   char *fnam, *snam, *addr;
   int phone;
} r = {"Jan", "Kowalski", "ul. Osobliwa 1", 5568};
```

Po takiej deklaracji, fraza: `struct rekord` może być dalej używana jak *nazwa typu*, możemy np. deklarować zmienne:

```
struct rekord a, b, c;
```

dokonywać przypisań: `a = r;` jak również *przekazywać struktury do funkcji* i *definiować funkcje zwracające struktury;* struktury można także *zagnieżdżać*.

[*Jeśli jednocześnie z deklaracją struktury zadeklarujemy wszystkie zmienne, identyfikator struktury (tutaj:* **`rekord`** *) można pominąć.*]

Odwołania do składowych (*pól*) struktury mają postać:

nazwa-struktury`.`*nazwa-pola*

Przykład: `printf("%s %d\n", a.snam, a.phone);`