

# Poprzedni wykład [ **12.11.2019** ] :

- Funkcje w języku C (*definicje i prototypy*)
- Struktura programu: *Zmienne zewnętrzne*, **zasięg nazw**; programy wieloplikowe (*wprowadzenie*)
- Klasy pamięci: ***static, register***
- Inicjowanie zmiennych
- Rekurencja; przykład: Algorytm ***quicksort***

# Prototypy funkcji: *kiedy są konieczne?*

W wielu sytuacjach funkcje wystarczy po prostu zdefiniować; także jeśli wywołują (*rekurencyjnie*) same siebie:

```
double binom(int n, int k)
{
    if (n<2) return 1.0;
    if ( k<1 || k==n ) return 1.0;
    return binom(n-1,k) + binom(n-1,k-1);
}
```

=> Prototyp nie jest potrzebny, jeśli definicja poprzedza wywołanie funkcji w obrębie **jednego pliku** źródłowego.

Jeśli rekurencja jest pośrednia ( i funkcja nie zwraca `int` ) :

```
double binom1(int n, int k)
{
    double binom2(int n, int k); /* KONIECZNE! */
    if (n<2) return 1.0;
    if ( k<1 || k==n ) return 1.0;
    return binom2(n-1,k) + binom2(n-1,k-1);
}
```

```
double binom2(int n, int k)
{
    if (n<2) return 1.0;
    if ( k<1 || k==n ) return 1.0;
    return binom1(n-1,k) + binom1(n-1,k-1);
}
```

[ **binom1(...)** wewn. **binom2(...)** – działa **deklaracja niejawna !** ]

# Programy wieloplikowe (*przypomnienie*)

W przypadku *niezgodności typów* (definicja/wywołanie funkcji) kompilator zgłosi błąd wyłącznie, jeśli definicja i wywołanie znajdują się w tym samym pliku.

[ *W przypadku programu wieloplikowego, mechanizm deklaracji niejawnych może prowadzić do trudnych do wykrycia błędów!* ]

=> Dobrze przemyślane *prototypy funkcji* są kluczowe dla poprawnego działania programu podzielonego na kilka plików.

W szczególności, prototypy ZAWSZE powinny zawierać listę parametrów (lub `void`); pusta lista parametrów — `fukcja()`;  
— *jedynie wyłącza kontrolę poprawności!* [ *a nie oznacza koniecznie „funkcji bez parametrów”* ]

# Pliki nagłówkowe

**Prototypy funkcji** zwykle zbieramy w plikach nagłówkowych (ang. *header files*) a później włączamy *dyrektywą preprocesora*:

```
#include "nazwa_pliku.h"
```

we wszystkich plikach źródłowych, w których są potrzebne.

[ **Nadliczbowe prototypy nie przeszkadzają; brakujące — nie pomagają ...** ]

W plikach nagłówkowych zwykle umieszcza się też *makrodefinicje*; odpowiednie *dyrektywy preprocesora* pozwalają uniknąć powtórzeń (kiedy np. *nagłówek* włącza inny *nagłówek*):

```
#ifndef LINE_MAX
#define LINE_MAX 1000
#endif
```

**Definicje funkcji**, podobnie jak *definicje zmiennych zewnętrznych*, umieszczamy z kolei w plikach źródłowych ( `nazwa.c` ).

*W praktyce, musimy szukać kompromisu pomiędzy dążeniem do tego, aby każda nazwa była widoczna tylko tam, gdzie jest potrzebna, a potrzebą utrzymania porządku w wielu plikach źródłowych przy modyfikacji programu.*

[ => *Pisanie programu nigdy się nie kończy ...* ]

Zagadnienie zilustrujemy teraz na przykładzie programu, który *wczytuje serię liczb ze standardowego wejścia, a następnie sortuje te liczby (w porządku niemalejącym) omawianą poprzednio metodą **quicksort*** [zob. `wyklad05.pdf` ]

```

#include <stdio.h>
#include "sortowanie.h"
#define NMAX 1000

int main()
{
    int j,n;
    long double tab[NMAX];    /* sortujemy long-double! */

    n=0;
    while (n<NMAX && EOF!=scanf("%Lf",&tab[n]))
        n++;
    Lf_qsort(tab,0,n-1);
    printf("# N_SWAP = %d\n", n_swap);
    for (j=0; j<n; j++)    printf("%Lg\n",tab[j]);
    return n;
}

```

Zawartość *pliku nagłówkowego*: "sortowanie.h"

```
extern int n_swap;  
void Lf_swap(long double v[], int i, int j);  
void Lf_qsort(long double v[],  
              int left, int right);
```

[ => *TUTAJ zebrano po prototypy **wszystkich** funkcji, które są zdefiniowane w kolejnych plikach źródłowych; wraz z deklaracją zmiennej zewnętrznej: n\_swap ]*



```
#include <stdio.h>                                /* Zawartość pliku: "swap.c" */
#include "sortowanie.h"

int n_swap=0;   /* licznik operacji "swap" */

/* swap: zamień miejscami v[i] z v[j] */
void Lf_swap(long double v[], int i, int j)
{
    long double temp;

    if (i != j) {
        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
        n_swap++;
    }
}
```

```

#include <stdio.h>                                /* Zawartość pliku: "qsort.c" */
#include "sortowanie.h"

void Lf_qsort(long double v[], int left, int right)
{
    int i, last;    /* zmienne automatyczne */

    if (left >= right) return;    /* koniec sortowania */
    Lf_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])
            Lf_swap(v, ++last, i);
    Lf_swap(v, left, last); /* wybrany elem. wraca */
    Lf_qsort(v, left, last-1);
    Lf_qsort(v, last+1, right);
}

```

Program możemy teraz **skompilować** sekwencją poleceń:

```
gcc -c main.c
```

```
gcc -c swap.c
```

```
gcc -c qsort.c
```

```
gcc main.o swap.o qsort.o -o sortowanie.out
```

Kolejność **pierwszych trzech** poleceń jest najzupełniej dowolna, a jeśli w katalogu nie ma innych plików źródłowych — możemy je zastąpić jednym poleceniem: `gcc -c *.c`

[ *Polecenie: "gcc \*.c" będzie miało nieco inny skutek — wówczas przetłumaczone zostanie wszystko i wygenerowany plik wykonywalny: "a.out" ]*

Jeśli zmodyfikujemy *tylko jeden* z plików źródłowych ( `main.c` , `swap.c` lub `qsort.c` ) wystarczy przetłumaczyć tylko ten plik, a następnie ponownie połączyć pliki obiektowe ( `*.o` )

[ *Jeśli jednak modyfikujemy nagłówek "sortowanie.h" — na ogół konieczna będzie jest **rekompilacja całego programu.*** ]

Projektując program założyliśmy, że **obiekty globalne** (zmienna zewnętrzna `n_swap` oraz funkcje `Lf_swap` i `Lf_qsort` ) mają być widoczne dla wszystkich plików źródłowych.

Zawartość nagłówka "sortowanie.h" faktycznie włączona jest **trzykrotnie**; w niczym to nie przeszkadza — są tam wyłącznie *deklaracje i prototypy funkcji*, które mogą się powtarzać.

[ *Można też napisać: **cat \*.c >all.c** a następnie: **gcc all.c*** ]

**Przykładowy wynik** działania programu ( `./sortowanie.out` ):

```
1e+1000
1
-1
1e-100
# N_SWAP = 5
-1
1e-100
1
1e+1000
```

# Polecenie `make` i pliki `Makefile`

Polecenie `make` w systemie UNIX jest wygodnym narzędziem umożliwiającym kontrolę procesu kompilacji w przypadku skomplikowanych zależności pomiędzy plikami źródłowymi.

[ *Polecenie `make` nie jest ściśle związane z kompilatorem `gcc` czy z językiem C — jest to zupełnie ogólne narzędzie kontroli procesu, w którym pliki **wynikowe** są tworzone na podstawie **źródłowych**.* ]

Kontrolę działania polecenia `make` umożliwia plik `Makefile`, który zawiera sekcje napisane według schematu:

```
target:    dependencies
           commands
           ...
```

Jeśli chcemy wywołać konkretną sekcję, tzw. *cel* (ang. *target*), piszemy np.: `make target`

Polecenie `make` bez podania *celu* spowoduje wywołanie wszystkich sekcji `Makefile-a`, *zasadniczo* w kolejności, w jakiej występują — pierwszeństwo będą miały jednak *zależności* (ang. *dependencies*) potrzebne do realizacji *celów* występujących wcześniej.

Specjalny status ma sekcja `all` — jeśli występuje, polecenie `make` wykona wyłącznie te sekcje (*rekursywnie!*) od których `all` zależy, pozostałe (np. często spotykaną sekcję `clean` ) trzeba wywołać jawnie (pisząc: `make clean` ).

[ *Więcej informacji:* <https://www.tutorialspoint.com/makefile/> ]

Plik **Makefile** dla programu sortującego liczby:

```
HEADER = sortowanie.h
EXEC = sortowanie.out
all:$(EXEC)

$(EXEC):main.o swap.o qsort.o
    gcc main.o swap.o qsort.o -o $(EXEC)
main.o:main.c $(HEADER)
    gcc -c main.c
swap.o:swap.c $(HEADER)
    gcc -c swap.c
qsort.o:qsort.c $(HEADER)
    gcc -c qsort.c

clean:
    rm *.o
```



Wówczas, polecenie `make` skompiluje wszystkie pliki, dla których czas ostatniej modyfikacji jest *późniejszy* niż czas modyfikacji powiązanego pliku z kodem pośrednim, i połączy całość.

[ Np., przypadku modyfikacji pliku `sortowanie.h` — całość programu będzie rekompilowana. ]

Jeśli pliki pośrednie ( `*.o` ) nie istnieją, polecenie `make` uruchomi po prostu całą procedurę tłumaczenia i łączenia programu:

```
gcc -c main.c
```

```
gcc -c swap.c
```

```
gcc -c qsort.c
```

```
gcc main.o swap.o qsort.o -o sortowanie.out
```

Jeśli nie trzeba już niczego kompilować, otrzymamy komunikat:

```
make: Nothing to be done for `all'.
```

Polecenie: `touch swap.c` wymusi ponowną kompilację *jednego* pliku źródłowego ( `swap.c` ) po którym konieczne będzie ponowne *łączenie*; `make` wywoła zatem komendy:

```
gcc -c swap.c
```

```
gcc main.o swap.o qsort.o -o sortowanie.out
```

Z kolei polecenie: `make clean` kasuje wszystkie *object files* (por. *ostatnia sekcja* w pliku `Makefile` ).

Ponowne wywołanie `make` rekompiluje wówczas cały program.

Podobny skutek będzie miała komenda:

```
touch sortowanie.h
```

— wymusi ona rekompilację całości programu ( *ponieważ wszystkie kody pośrednie zależą od nagłówka !* ).

# Preprocesor języka C

Dyrektywy preprocesora ( np. `#include` oraz `#define` ) stanowią wygodne rozszerzenie języka C.

[ Formalnie, *preprocesor* wykonuje oddzielny, ***pierwszy krok*** tłumaczenia programu, poprzedzający *właściwą kompilację*. ]

Oprócz ***wstawiania zawartości plików*** ( `#include` ) oraz ***definicji*** ( `#define` — zastępowanie symbolu dowolnym ciągiem znaków) preprocesor umożliwia również kontrolę ***kompilacji warunkowej*** (przydatne w przypadku wielokrotnego wstawiania tych samych plików nagłówkowych) jak również definiowanie ***makrodefinicji z argumentami***.

# Preprocesor cz. 1: *Wstawianie plików*

Wstawianie plików (`#include`) umożliwia — w szczególności — łatwe dołączenie wspólnych deklaracji `extern`, definicji `#define` oraz *prototypów funkcji* dla każdego pliku źródłowego.

Stosowanie dyrektywy: `#include` jest ***zalecanym*** sposobem sporządzania deklaracji dla *dużego programu*.

[ *Pliki źródłowe zaopatrzone w te same deklaracje zmiennych i prototypy funkcji automatycznie eliminują szczególnie złośliwy rodzaj błędów.* ]

**Działanie dyrektywy #include.** Każdy z wierszy postaci:

```
#include "plik1"    lub    #include <plik2>
```

zostanie *mechanicznie zastąpiony* zawartością wskazanego pliku.

Użycie pierwszej formy ( "plik1" ) powoduje, że poszukiwanie *pliku wstawianego* rozpoczyna się w katalogu, w którym znajduje się konkretny plik źródłowy.

Jeśli plik wstawiany nie zostanie *odnaleziony*, lub użyjemy drugiej formy ( <plik2> ) — efekt określają zasady przyjęte w danej implementacji.

[ Formy <plik> zwykle używamy do włączania nagłówek bibliotek standardowych, jak <stdio.h> itp., **własne** nagłówki umieszczamy w tym samym katalogu co program i używamy formy "plik". ]

*Pliki wstawiane* także mogą zawierać dyrektywy **#include (!)**

# Preprocesor cz. 2: *Makrorozwinięcia*

W najprostszym przypadku, **makrodefinicja** ma postać:

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

Każde wystąpienie identyfikatora *nazwa* w pliku źródłowym będzie zastąpione ciągiem znaków tworzących *zastępujący-tekst*.

[ *Zasięg nazwy rozciąga się od miejsca definicji do końca pliku.* ]

Długie makra możemy definiować z użyciem **operatora przedłużenia wiersza**: znak `' \ '` na końcu linii spowoduje, że kompilator *doklei* następny wiersz na końcu aktualnego.

( Operator `' \ '` nie jest ściśle związany z makrami, można go używać np. pisząc długie wyrażenia. )

Makrodefinicje mogą korzystać z innych makrodefinicji, które zostały zdefiniowane wcześniej.

Makrorozwinięć dokonuje się wyłącznie dla **całych leksemów** (inaczej: *jednostek leksykalnych*): jeśli np. zdefiniujemy makro: **KU** a później pojawi się nazwa: **KUKURYDZA** — *nic się nie stanie*.

Podobnie, makrorozwinięcia *nie działają wewnątrz stałych napisowych*: instrukcja `printf("KUKURYDZA");` również nie będzie *zaburzona* wcześniejszym makrem ( **KU** ).

Zastosowania nawet najprostszych makr (**bez argumentów**) nie ograniczają się jedynie do definiowania stałych, gdyż występujący po nazwie makra: *zastępujący-tekst* może być zupełnie dowolny.

Na przykład, makro:

```
#define forever for (;;) /* pętla nieskończona */
```

definiuje słowo `forever` oznaczające *pętlę nieskończoną*.

# Makra z argumentami

W przypadku *makr z argumentami*, zastępujący tekst jest *generowany* niezależnie dla różnych wywołań; np. dla makra

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

jego wywołanie:

```
x=max(p+q, r+s);
```

wygląda na pierwszy rzut oka zupełnie jak wywołanie funkcji.

Wywołanie makra ***nie jest*** jednak ***wywołaniem funkcji*** — wyrażenia nie są obliczane, lecz wstawiane mechanicznie do *tekstu-zastępującego*.

Instrukcja — ***jak wyżej*** — zostanie zastąpiona przez:

```
x= ((p+q)>(r+s)?(p+q):(r+s));
```



Jak widać, we wstawianych wyrażeniach mogą występować **operatory o różnych priorytetach** — **bardzo ważne jest, aby w definicji makra *nie zabrakło nawiasów* !**

Ważna zaleta makr z argumentami polega na tym, że — *inaczej niż funkcje* — co do zasady mogą być stosowane dla dowolnych **typów** argumentów.

Jak każde narzędzie o dużej uniwersalności, makra z parametrami kryją **kilka pułapek**. W makrze `max(A, B)` większe z wyrażień jest obliczane *dwukrotnie*; problemem mogą być zatem *efekty uboczne*. Na przykład, w wyrażeniu:

```
max(i++, j++)
```

większa z wartości *zostanie zwiększona dwukrotnie*.

Inną zaletą *makra z argumentami* jest możliwość uniknięcia narzutów związanych z częstymi wywołaniami funkcji; np. `getchar` i `putchar` są zdefiniowane jako makra w pliku `<stdio.h>`.

Jeśli chcemy zdefiniować swoją wersję `getchar` — definicję makra możemy anulować poleceniem:

```
#undef getchar
```

## Makra zawierające stałe napisowe

Szczególny przypadek stanowią makra z parametrami, w których chcemy użyć *stałych napisowych*. W makrze:

```
#define makro(A) printf("A lovely morning\n")
```

parametr `A` wewnątrz `" "` będzie **nieaktywny** (tzn. `printf` wypisze zawsze *ten sam napis*) — ***a kompilator nie zgłosi błędu !!***

Jeśli jednak nazwa parametru w *zastępującym-tekście* zostanie poprzedzona znakiem `#` wówczas cała kombinacja (np. `#A` ) zostanie zastąpiona argumentem wywołania otoczonym " " .

[ *Resztę problemu rozwiązuje **dodawanie stałych napisowych!*** ]

Przykładowo, makro:

```
#define exprint(expr) printf(#expr " = %g\n", expr)
```

wywołane w instrukcji:

```
exprint(x/y);
```

zostanie zastąpione (tj. *preprocesor zastąpi*) przez:

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

Następnie, kompilator wykona **dodawanie** (=sklejanie) napisów, i funkcja `printf` otrzyma (*pierwszy*) argument: `"x/y = %g\n"`

Dodatkowo, jeśli w *argumencie wywołania* makra (w którego definicji użyto znaku *#* przed odpowiednim parametrem) wystąpi znak cudzysłowiu: *"* zostanie zastąpiony kombinacją: *\*"

Podobnie, znaki: *\* będą zastępowane przez: *\\*

— tak, aby wyniki były **poprawnymi** stałymi napisowymi.

## Operator preprocesora: **##**

Sklejanie argumentów aktualnych makra, *niebędących stałymi napisowymi*, jest także możliwe w języku C.

Jeśli w *zastępującym-tekście* parametr sąsiaduje z operatorem **##**, parametr zostanie zastąpiony argumentem wywołania, następnie operator **##** będzie usunięty wraz z otaczającymi go *białymi znakami*, po czym wynik zostanie przetworzony ponownie.

Przykładowo, makro:

```
#define attach(A,B) A ## B /* skleja A z B */
```

Wywołanie `attach(fun,123)` utworzy słowo: `fun123`

**Zasady zagnieżdżania operatora `##`** są dość zaskakujące:

wywołanie `attach(attach(1,2),3)` będzie **niepoprawne**, ponieważ obecność operatora `##` zablokuje rozwinięcie wewnętrznego argumentu, w efekcie czego dostaniemy:

```
attach(1,2)3
```

Problem rozwiązuje wprowadzenie **makra drugiego poziomu**:

```
#define xattach(A,B) attach(A,B)
```

wywołanie: `xattach(xattach(1,2),3)` generuje pożądany napis: `123 [ ponieważ xattach nie zawiera jawnie ## ]`

# Preprocesor cz. 3: *Kompilacja warunkowa*

Dyrektywy *kompilacji warunkowej* pozwalają **sterować procesem tłumaczenia** programu na język maszynowy:

W zależności od pewnych **warunków** — obliczanych podczas kompilacji — określone fragmenty kodu źródłowego są włączane do programu lub pomijane.

[ *Jeśli np. pliku nagłówkowym mamy definicje zmiennych zewnętrznych, które nie mogą się powtarzać — można spowodować, aby zawartość takiego pliku de facto została włączona tylko raz.* ]

W wierszu zaczynającym się od **#if** obliczana jest wartość stałego wyrażenia całkowitego (takie wyrażenie **nie może** zawierać **sizeof**, rzutowania, ani stałych **enum**). Jeśli **wartość != 0** — włączane są *kolejne wiersze* aż do: **#endif** **#elif** lub **#else**

Po instrukcji `#if` często występuje `defined(nazwa)` — jest ono równe `1`, jeśli nazwa została wcześniej zdefiniowana za pomocą `#define`, zaś `0` — w przeciwnym przypadku.

Jeśli chcemy, aby zawartość pliku `myheader.h` została *de facto* wstawiona do programu **tylko raz** (pomimo *wielokrotnego* użycia `#include "myheader.h" !`) zawartość pliku należy otoczyć następującymi dyrektywami preprocesora:

```
#if ! defined(MYHEADER)
```

```
#define MYHEADER
```

```
/* tutaj wpisujemy zawartość "myheader.h" */
```

```
#endif
```

Zestawy instrukcji podobne do powyższego są tak *często spotykane*, że wprowadzono specjalne (skrótowe) instrukcje: `#ifdef` oraz `#ifndef`. Możemy zatem napisać:

```
#ifndef MYHEADER  
#define MYHEADER
```

```
/* tutaj wpisujemy zawartość "myheader.h" */
```

```
#endif
```

— a efekt będzie **identyczny**.



Niekiedy *kontrola kompilacji* wymaga **decyzji wielowariantowych**; możemy np. wstawić jedną z kilku wersji nagłówka:

```
#if WHICH_HEADER == 1
    #define HEADER "myheader1.h"
#elif WHICH_HEADER == 2
    #define HEADER "myheader2.h"
#else
    #define HEADER "myheader_default.h"
#endif
#include HEADER
```