Kurs C

Radosław Łukasik

Wykład 1

O przedmiocie

Egzamin ustny z zagadnień z wykładu.
Możliwość zwolnienia z egzaminu - projekt (tematy projektów podane będą na laboratoriach).

Wykłady będą sukcesywnie pojawiały się na Teams.
Terminy wykładów: co dwa tygodnie jak na stronie z planami zajęć.

Literatura

Główna:

- D. Kernighan, D. Ritchie "Język ANSI C", WNT, 2002
- K. N. King, Język C. Nowoczesne programowanie. Wydanie II, Helion, Gliwice 2011.

Uzupełniająca:

- "Programming languages - C", Standard ISO C99, ISO/IEC 9899:1999
- "Information technology - Programming languages - C", Standard ISO C11, ISO/IEC 9899:2011
- K. Jassem, A. Ziemkiewicz, Sztuka dobrego programowania, PWN, Warszawa 2016.
- P. Koprowski "0x80 zadań z C i C++", Exit, 2009
- http://cpp0x.pl
- http://www.cplusplus.com
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Trochę historii

- 1972 - pierwsza wersja języka C (Dennis Ritchie)
- 1983 - pierwsza wersja języka C++ (Bjarne Stroustrup) będącego obiektowym rozszerzeniem C
- 1989 - ANSI C (inaczej C89), chwilę później to samo jako norma ISO/IEC 9899:1990 (C90)
  (nowa definicja argumentów funkcji, **double** jako podstawowy typ zmiennoprzecinkowy)
- 1999 - norma ISO/IEC 9899:1999 (C99)
  (nowe typy danych, tablice o zmiennej długości, mocno rozbudowana biblioteka standardowa)
- 2011 - norma ISO/IEC 9899:2011 (C11)
  (wątki, makra generyczne - te rzeczy nie są zgodne z C++ 11)
- 2017 - norma ISO/IEC 9899:2018 (C17 lub C18)
  tylko poprawki istniejących rzeczy, bez dodawania nowych

Języki programowania

Rozróżniamy języki programowania niskiego (np. asembler) i wysokiego poziomu (np. C++). Rożnica polega na tym, że języki niskiego poziomu wymagają znajomości działania procesorów i mogą być pisane pod konkretny sprzęt (co pozwala na ich lepszą wydajność). Języki wysokiego poziomu mają za to składnie ułatwiającą zrozumienie przez człowieka. Kod źródłowy takiego języka jest tłumaczony przez kompilator lub interpreter na język niższego poziomu i dopiero wtedy możemy wykonać program.

Kompilator - tworzy plik wykonywalny (pod konkretny system operacyjny) tłumacząc kod programu na język maszynowy (przeważnie stosując optymalizację, dzięki czemu nie ma dużej różnicy w szybkości w porównaniu do języków niskiego poziomu).

Interpreter - analizuje kod źródłowy i wykonuje przeanalizowane polecenia. Łatwiej pisze się taki kod pod różne systemy operacyjne, ale jest on wolniejszy i potrzebuje więcej pamięci do działania.

Ogólna budowa programu w C

- Dyrektywy preprocesora - wyrażenia zaczynające się od znaku # pozwalające na wykonanie pewnych działań na początku kompilacji.

Przykłady dyrektyw:

- #include <nazwa_biblioteki> - dołączanie biblioteki znanej kompilatorowi;
- #include "nazwa_biblioteki" - dołączanie biblioteki użytkownika, wraz z ścieżka względną lub bezwzględną;
- #define nazwa_stałej wartość - definiowanie stałej (kompilator zamienia w kodzie podaną nazwę na wartość jej przypisaną);
- #define nazwa_makra(arg1,arg2,...,argN) makro - definiowanie makra, np.

```
#define MNOZ(a,b) (a)*(b) // nawiasy dla dobrej kolejności działań
```

- #undef, #ifdef #ifndef #else #endif, #if #elif #else #endif, #error, #warning, ...

Ogólna budowa programu w C

- Dyrektywy preprocesora - wyrażenia zaczynające się od znaku # pozwalające na wykonanie pewnych działań na początku kompilacji.
- Zmienne globalne i funkcje użytkownika

Ogólna budowa programu w C

- Dyrektywy preprocesora - wyrażenia zaczynające się od znaku # pozwalające na wykonanie pewnych działań na początku kompilacji.
- Zmienne globalne i funkcje użytkownika
- Główna funkcja programu

Główna funkcja programu jest miejscem, gdzie program rozpoczyna swoje działanie. Stąd każdy program w C musi mieć tę funkcję. Poniżej przykład działającego programu:

```
#include <stdio.h>

int main(){
    printf("Jestem Programem\n");
    return 0;// program zwraca do systemu pewną wartość
}
```

Funkcja **main** może mieć jakieś parametry, ale o tym będzie później.

Uwagi do pisania kodu w C

- Komentarze są ważne!
  W C komentarze możemy wstawiać za pomocą // (komentarz liniowy od C99, wywodzący się od C++) lub zaczynając od /∗ i kończąc na ∗/ (komentarz blokowy obejmujący kilka wierszy).

```
int x;// komentarz do końca tej linii
int y;
/*
Komentarz wielolinijkowy
*/
```

- Każda instrukcja powinna być zakończona średnikiem. Na instrukcję może składać się wiele wyrażeń, których nie oddzielamy średnikami.
- Dla poprawy czytelności kodu każdą instrukcję umieszczaj w nowej linii.
- Stosuj wcięcia dla instrukcji będących w bloku (czyli między nawiasami { ... }).
- Używaj pustej linii do oddzielenia pewnych fragmentów kodu.

Słowa kluczowe w C

Słowa kluczowe są to słowa których nie możemy użyć do definiowania własnych nazw zmiennych, funkcji itp.

auto	extern	short	while
break	float	signed	_Alignas (C11)
case	for	sizeof	_Alignof (C11)
char	goto	static	_Atomic (C11)
const	if	struct	_Bool (C99)
continue	inline (C99)	switch	_Complex (C99)
default	int	typedef	_Generic (C11)
do	long	union	_Imaginary (C99)
double	register	unsigned	_Noreturn (C11)
else	restrict (C99)	void	_Static_assert (C11)
enum	return	volatile	_Thread_local (C11)

Słowom kluczowym z podkreślnikiem na początku odpowiadają również pewne słowa już bez podkreślnika i z małej litery i mogą być dostępne po dołączeniu odpowiedniej biblioteki.

Zmienne i stałe

- Zmienna to pewien obszar w pamięci komputera, w którym przechowywane sa dane. Wielkość tego obszaru i sposób reprezentacji danych zależą do typu zmiennej.
- Kompilator znając typ zmiennej wie ile pamięci trzeba zarezerwować dla niej, jak należy interpretować bity tej zmiennej oraz w jaki sposób będą wykonywane operacje na zmiennej.
- Deklarując użycie zmiennej w programie należy podać jej typ i nazwę (ewentualnie ją zainicjować).
- Każda nazwa musi być zadeklarowana zanim zostanie użyta oraz może być użyta tylko raz*.
- Każda zmienna powinna być zainicjowana przed pierwszym użyciem (pamięć nie jest nigdy pusta!).
- Stałe możemy albo tworzyć za pomocą #define (stała symboliczna) albo dodając przed typem zmiennej słowo kluczowe **const**. Wówczas jest ona tworzona w pamięci komputera ale kompilator nie pozwala nam jej modyfikować w bezpośredni sposób (poza inicjowaniem).

Podstawowe typy danych

typ	rozmiar	opis
char	1B	znak, l. całkowita od -128 do +127
wchar_t (wchar.h)	2B	szeroki typ znakowy do obsługi Unicode
short	2B	l. całkowita od -32768 do +32767
int	4B	l. całkowita od -2^{31} do $2^{31}-1$
long	4B lub 8B	l. całkowita jak **int** lub **long long int**
long long	8B	l. całkowita od -2^{63} do $2^{63}-1$
float	4B	l. zmiennoprzecinkowa pojedynczej precyzji
double	8B	l. zmiennoprzecinkowa podwójnej precyzji
long double	8B–16B	l. zmiennoprzecinkowa
void	-	typ nieokreślony
bool (stdbool.h)	1B	wartości logiczne **true**=1 i **false**=0

Wszystkie liczby całkowite występują też w wersji **unsigned** oznaczającą liczbę bez znaku o takim samym rozmiarze (zakres od 0 do $2^{\text{ilość bitów}}$-1). Zamiast **unsigned int** można napisać krócej **unsigned**.
Aby sprawdzić rozmiar danego typu można użyć polecenia **sizeof**(typ) lub **sizeof**(zmienna).

Liczby zmiennoprzecinkowe są przechowywane (począwszy od najwyższego bitu) w postaci znaku (1b), wykładnika (8b dla **float**, 11b dla **double**) oraz mantysy (23b dla **float**, 52b dla **double**). Wynikowa niezerowa wartość przechowywanej liczby wyraża się wzorem: $x = (-1)^{znak}(1.\text{mantysa}) \cdot 2^{\text{wykładnik}-\text{przesunięcie}}$, gdzie przesunięcie jest równe 127 dla **float** lub 1023 dla **double**. Zero występuje w wersji $+$ lub $-$ (mantysa i wykładnik są zerowe). Oprócz tego są wyróżnione specjalne wartości nie będące liczbami przeważnie powstałe w wyniku niedozwolonej operacji (dla wykładnika 255 dla **float**, 2047 dla **double**) (qNaN, sNaN, $\pm\infty$, np. pierwiastkowanie liczby ujemnej, dzielenie przez 0) lub powstał niedomiar (wykładnik równy 0) i wynik jest liczbą nieznormalizowaną postaci $x = (-1)^{znak}(0.\text{mantysa}) \cdot 2^{1-\text{przesunięcie}}$.

liczba (dwójkowo)	**int**	**unsigned**	**float**
10000000 00000000 00000000 00000000	-2147483648	2147483648	-0.0
10111111 10000000 00000000 00000000	-1065353216	3212836864	-1.0
11111111 11111111 11111111 11111111	-1	4294967295	qNaN
00000000 00000000 00000000 00000001	1	1	2^{-149}
01111111 10000000 00000000 00000000	2139095040	2139095040	$+\infty$

Zmienne tekstowe można inicjować następująco:

- dla pojedynczego znaku

```
char c='a'; // używamy apostrofów przed i za jednym znakiem
```

- dla łańcucha znaków

```
char lancuch[10]="znaki"; // używamy cudzysłowu przed i za znakami
```

  Łańcuch znaków zostaje przy inicjacji domyślnie zakończony bajtem zerowym.

Uwaga "a" to dwa bajty: 'a' i '\0'.

Dzięki bibliotece **complex.h** możemy również tworzyć i operować na liczbach zespolonych. Dostępne są tam typy całkowite i zmiennoprzecinkowe, które trzeba poprzedzić słowem **complex**, np. **complex float**. Zajmują one dwa razy tyle miejsca co odpowiadające im typy całkowite czy rzeczywiste. Działają na nich te same operatory jakie występują dla liczb rzeczywistych. Stała **I** oznacza jednostkę urojoną. Dla tych liczb są dostępne pewne funkcje matematyczne podobne do tych w bibliotece **math.h** (te same nazwy poprzedzone literą 'c').

ASCII

Zmienne typu **char** przechowują znaki, ale mogą być również traktowane jako liczby. Wartości przypisane znakom są związane z kodem ASCII.
ASCII (czytaj aski) – siedmiobitowy system kodowania znaków. Zawiera on 95 znaków drukowalnych (m.in. litery, cyfry) oraz polecenia sterujące. Każdy znak ma przypisaną pewną wartość.
Nie wszystkie znaki sterujące działają (są zależne od sprzętu). Oto niektóre z nich:

SYGNAŁ DŹWIĘKOWY	7
BACKSPACE	8
TABULATOR	9
NOWA LINIA	10
POCZĄTEK WIERSZA	13

ASCII

Poniżej znajduje się lista znaków drukowalnych i ich wartości.

spacja	32
!	33
"	34
#	35
$	36
%	37
&	38
’	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63

@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79

P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[	91
\	92
]	93
^	94
_	95

‘	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111

p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122
{	123
|	124
}	125
~	126

Nazwy zmiennych

Nazwa zmiennej to napis składający się z liter alfabetu angielskiego, cyfr, oraz podkreślnika, przy czym nie może on zaczynać się od cyfry.
Jeżeli nazwa składa się z wielu słów, to dla poprawy czytelności możemy je łączyć podkreślnikiem lub stosować dużą literę na początku każdego ze słów.
Również w celu poprawy czytelności kodu można stosować notację węgierską polegającą na poprzedzeniu nazwy zmiennej mała literą lub literami określającą typ zmiennej. Poniżej mamy przykład takich przedrostków.

s	string (łańcuch znaków)
sz	łańcuch znaków zakończony bajtem zerowym
c, n, i, l, ll	odpowiednio **char**, **short**, **int**, **long**, **long long**
by, w, dw, qw	wersje **unsigned** odpowiednio **char**, **short**, **int**, **long long**
x, y	**int** (przy zmiennych określających współrzędne)
cx, cy	**int** (przy zmiennych określających rozmiar, długość)
b	**bool**
f	flaga (bitowa)
h	uchwyt
p	wskaźnik

Deklaracje i inicjowanie zmiennych, zasięg

Deklaracja zmiennej polega na podaniu jej typu oraz nazwy.
Definicja (inicjowanie) zmiennej polega na przypisaniu jej wartości początkowej.

```
int x;// deklaracja
x=5;// inicjowanie
int y=4;// jednoczesna deklaracja i inicjalizacja
```

Miejsce definicji zmiennej decyduje o jej zasięgu (widoczności dla kompilatora) i przechowywaniu. Zmienne globalne są tworzone na starcie, dostępne wszędzie i przechowywane do zakończenia programu. Zmienne lokalne występujące w funkcjach, pętlach (w pętli **for** tworzenie zmiennych w instrukcjach inicjujących od C99), instrukcjach warunkowych są tworzone i dostępne tylko w tych miejscach, a gdy program opuści to miejsce, to są one usuwane.
Dodając modyfikator **static** przed typem zmiennej możemy utworzyć taką zmienną lokalną, która zostaje usunięta dopiero na koniec programu (jak zmienna globalna) ale jej zasięg pozostaje dalej lokalny. modyfikator **auto** jest domyślnym, oznacza tworzenie lokalnej zmiennej i nie trzeba go pisać przy zmiennych. Zdarza się również widzieć operator **register** oznaczający rejestr procesora, ale lepiej zamiast niego włączyć optymalizację kompilatora.

Rzutowanie typów

Aby zamienić jeden typ zmiennej na drugi stosujemy tzw. rzutowanie typu. Polega ono na podaniu nowego typu zmiennej w nawiasach okrągłych przed przypisaniem do nowej zmiennej lub operacją na tej zmiennej. Czasami podanie nowego typu nie jest konieczne i mówimy wtedy o konwersji domyślnej (np. gdy zwiększamy rozmiar zmiennej).

```
char u;
int n=400,m;
float x,y=5.7;
x=n;// x=400.0 =256+144
x=(float) n;
m=y;// m=5
m=(int) y;
u=n;// u=-112 konwersja stratna, 144+112=256
u=(char) n;
```

Sufiksy dla stałych liczbowych

Może się zdarzyć sytuacja, gdzie zmiennej przypisujemy stałą wartość ale z określeniem jakiego typu ona ma być, np. liczby całkowite są domyślnie typu **int** (dla mniejszych zakresów są przycinane, więc nic złego się nie dzieje, ale dla większych mogą występować różne nieprzewidziane zachowania).

```
long long x=1024*1024*4096; // źle, x=0
long long x=1024LL*1024*4096; // poprawne, LL oznacza long long
long y=-3;
printf("%d\n", y < 12U); // 0 bo (unsigned)-3 = 2^32 -3 >12
```

Stosuje się sufiksy: U, L, UL, LL, ULL dla liczb całkowitych oraz F, L dla liczb zmiennoprzecinkowych.

Operatory

$+$	dodawanie liczb
$-$	odejmowanie liczb
$*$	mnożenie liczb
$/$	dzielenie całkowite lub zwykłe
%	reszta z dzielenia liczb całkowitych
$++$	inkrementacja (zwiększanie o 1)
$--$	dekrementacja (zmniejszanie o 1)

Wynik działań na typach mieszanych jest zawsze ogólniejszym typem z nich, czyli np. działanie na dwóch **int** jest też **int**, działanie na **int** i **float** daje **float**.
Inkrementacja i dekrementacja występują w formie przedrostkowej i przyrostkowej.

```
int x=2,y;
y=++x;//y=3, x=3
// kod równoważny linii powyżej
x++;
y=x;
```

```
int x=2,y;
y=x++;//y=2, x=3
// kod równoważny linii powyżej
y=x;
x++;
```

Operatory binarne

|	bitowe or
&	bitowe and
^	bitowe xor
~	bitowe not
<<	przesunięcie bitowe w lewo
>>	przesunięcie bitowe w prawo

Przesunięcia w lewo/prawo możemy traktować jako mnożenie/dzielenie przez 2 do podanej potęgi. Należy zwrócić tutaj uwagę, że przesunięcie w prawo dla liczb ze znakiem i bez znaku daje różne wyniki (dla liczb ze znakiem do najwyższego bitu wpisywane jest 1 lub 0 w zależności od znaku, dla liczb bez znaku zawsze 0).

```
char n=-6,m=3,k;// n=1111 1010, m=0000 0011
unsigned char u=131,w;// u= 1000 0011
k=n|m;// k=1111 1011
k=n&m;// k=0000 0010
k=n^m;// k=1111 1001
k=~n;// k=0000 0101
k=n<<2;// k=1110 1000
k=n>>2;// k=1111 1110
w=u>>1;// w=0100 0001
```

Operatory przypisania

Oprócz operatora przypisania $=$ są to operatory postaci **działanie=**, gdzie działanie jest jednym z operatorów arytmetycznych lub bitowych. Są to:

$$=,\ +=,\ -=,\ *=,\ /=,\ \%=,\ \hat{}=,\ |=,\ \&=,\ <<=,\ >>=.$$

Każdy z operatorów postaci **działanie=** można rozpisać w następujący sposób:

a **działanie=** b;
a = a **działanie** b;

Mimo, że oba te zapisy robią dokładnie to samo, to jest różnica w ich wykonaniu przez procesor (**działanie=** zajmują mniej miejsca w kodzie i są szybsze).
Po lewej stronie przypisania musi być zawsze zmienna, po prawej mogą występować stałe. Możliwa jest również konwersja niejawna typów, gdy typy zmiennych po obu stronach są różne.

Operatory logiczne i relacyjne

Są to operatory, które zwracają wartości logiczne **true** lub **false**.

wartość1 == wartość2	sprawdza czy wartości są równe
wartość1 < wartość2	sprawdza nierówność $<$ pomiędzy wartościami
wartość1 <= wartość2	sprawdza nierówność $\leqslant$ pomiędzy wartościami
wartość1 > wartość2	sprawdza nierówność $>$ pomiędzy wartościami
wartość1 >= wartość2	sprawdza nierówność $\geqslant$ pomiędzy wartościami
wartość1 != wartość2	sprawdza czy wartości są różne
warunek1 || warunek2	alternatywa dwóch warunków
warunek1 && warunek2	koniunkcja dwóch warunków
!warunek	negacja warunku
warunek ? kodt : kodf;	jeżeli warunek jest prawdziwy, to wykona się kodt, w przeciwnym wypadku wykona się kodf

Operator warunkowy ? : ; służy też do warunkowego przypisywania liczbie całkowitej jednej z dwóch wartości:

zmienna_całkowita = warunek ? wartość_true : wartość_false;

Operatory mają swoją kolejność wykonywania. Jeśli nie jesteśmy pewni kolejności lub chcemy ją zmienić, to użyjmy zwykłych nawiasów. Poniższa tabela pokazuje hierarchię operatorów (im niżej tym mniejszy priorytet) i typ ich łączności.

operatory	typ łączności
jednoargumentowe przyrostkowe, tzn. [], ., ->, wywołanie funkcji, postinkrementacja, postdekrementacja	lewostronna
jednoargumentowe przedrostkowe, tzn. !, ~, −, *, &, sizeof, rzutowanie, preinkrementacja, predekrementacja	prawostronna
*, /, %	lewostronna
+, −	lewostronna
<<, >>	lewostronna
<, <=, >, >=	lewostronna
==, !=	lewostronna
&	lewostronna
^	lewostronna
|	lewostronna
&&	lewostronna
||	lewostronna
? :;	prawostronna
=	prawostronna
,	lewostronna

Operacje wejścia i wyjścia

W C możemy pobrać i wyświetlać dane poprzez funkcje **scanf** i **printf** z biblioteki **stdio.h**.

```
#include <stdio.h>
int x,y;
⋮
scanf(" %d %d",&x,&y);
printf("Suma liczb: %d\n",x+y);
```

Funkcja **scanf** wczytuje znaki z wejścia do napotkania pierwszego białego znaku*. Wstawiając w **scanf** spacje jako pierwszy znak w cudzysłowie usuwamy wszystkie białe znaki występujące na początku wczytywanej zmiennej. W przypadku funkcji **scanf** nazwa zmiennej musi być poprzedzona znakiem & (o ile nie jest to tablica). Funkcja **printf** zwraca ilość poprawnie wypisanych znaków. Funkcja **scanf** zwraca ilość poprawnie wypełnionych zmiennych (jeśli zwraca 0, to wystąpił jakiś błąd wejścia). Niestety jeśli wywołamy więcej razy **scanf** niż mamy danych, to będziemy musieli je dopisać. Z tego względu nie jest łatwo określić czy dotarliśmy do końca danych. Musimy to sprawdzać inaczej wiedząc, że ostatni znak, to '$\backslash n$'.

Znaki specjalne

Znaki specjalne:

\a	sygnał dźwiękowy (systemowy)
\b	cofnięcie o jeden znak
\n	nowa linia
\r	powrót do początku wiersza
\t	tabulator
\\	backslash
\'	apostrof
\"	cudzysłów
\0	wartość 0, koniec łańcucha znaków

Białe znaki - znaki, których nie widać na ekranie (spacja, tabulator, nowa linia).

Znaki formatujące

W poleceniach **scanf** i **printf** pierwsza zmienna zawiera w cudzysłowach tekst oraz pewne znaki formatujące występujące po znaku %, które mają postać:
(dla **printf**) **%[flagi][szerokość][.precyzja][długość]specyfikacja**
(dla **scanf**) **%[szerokość][długość]specyfikacja**

Specyfikacja	Opis
d i	Liczba całkowita
u	Liczba naturalna
o	Zapis ósemkowy
x X	Zapis szesnastkowy (małe/duże litery, #x wstawia 0x przed liczbę)
f F	Zapis dziesiętny liczby zmiennoprzecinkowej
e E	Zapis naukowy (mantysa/wykładnik), (małymi lub dużymi literami)
g G	Krótsza z form reprezentacji: %e lub %f (%E lub %F)
a A	Zapis szesnastkowy z przecinkiem (mantysa/wykładnik, małe/duże litery)
c	Znak
s	Łańcuch znaków (**scanf** - do pierwszego białego znaku, **printf** - do '\0')
[znaki]	**scanf** - wczytywanie znaków aż do napotkania znaku spoza podanych
[^znaki]	**scanf** - wczytywanie znaków aż do napotkania jednego z podanych
p	Adres w pamięci
n	Zapis ilości wyświetlonych/wczytanych znaków (wskaźnik)
%	znak %

Flagi	Opis
-	Wyrównanie tekstu do lewej gdy podana jest szerokość tekstu
+	Liczby dodatnie są wyświetlane ze znakiem +
(spacja)	Przed liczbami dodatnimi wstawiana jest spacja
#	Używane z o, x lub X powoduje wyświetlenie przed liczbą 0, 0x lub 0X. Używane z a, A, e, E, f, F, g lub G wymusza pojawienie się zawsze kropki w zapisie.
0	Zamiast spacji przed liczbą wyrównaną do prawej są zera

Szerokość	Opis
(liczba)	Dla **printf** - Minimalna liczba znaków do wyświetlenia (domyślnie do prawej, ze spacjami z lewej). Dla **scanf** - maksymalna liczba znaków do odczytania.
*	Dla **printf** - szerokość jest podana jako argument poprzedzający wyświetlaną zmienną. Dla **scanf** - dane są odczytane z wejścia ale ignorowane.

Precyzja dla **printf** oznacza dla liczb całkowitych długość minimalną wyświetlanej liczby z dopisanymi zerami z przodu. Dla liczby zmiennoprzecinkowej oznacza ilość liczb po przecinku do wyświetlenia. W przypadku łańcuchów znaków oznacza maksymalną ilość znaków do wyświetlenia. Precyzja może być też znakiem * i oznacza to, że jest ona podana jako argument poprzedzający wyświetlaną zmienną.

Długość dla **printf**:

dł.	d i	u o x	f e g a	c	s	p	n
-	int	unsigned int	double	int	char*	void*	int*
hh	char	unsigned char	-	-	-	-	char*
h	short	unsigned short	-	-	-	-	short*
l	long	unsigned long	-	w_char	w_char*	-	long*
ll	long long	unsigned long long	-	-	-	-	long long*
L	-	-	long double	-	-	-	-

Długość dla **scanf**:

dł.	d i	u o x	f e g a	c s	p	n
-	int*	unsigned int*	float*	char*	void**	int*
hh	char*	unsigned char*	-	-	-	char*
h	short*	unsigned short*	-	-	-	short*
l	long*	unsigned long*	double*	w_char*	-	long*
ll	long long*	unsigned long long*	-	-	-	long long*
L	-	-	long double*	-	-	-

Tabele łatwo zapamiętać: hh - 1/4 rozmiaru, h - 1/2 rozmiaru, l - wersja **long** danej zmiennej, ll - wersja **long long**, L - tylko dla **long double**.
Funkcja **scanf** zwraca ilość pomyślnie wypełnionych zmiennych.

Przykład

```
int x=1024;
long long z=123456789123;
float y=3.141592;
double t=1.41421;
char tekst[15]={"Tu mamy napis."};
printf("Liczba x = %.8d\n",x); //Liczba x = 00001024
printf("%8d\n",x);             //     1024
printf("%8.3f\n",y);           //    3.142
printf("%8.0f\n",y);           //        3
printf("%#X\n",x);             //0X400
printf("%lld\n",z);            //123456789123
printf("%f\n",t);              //1.41421
printf("%.4s\n",tekst);        //Tu m
printf("%18s\n",tekst);        //    Tu mamy napis.

scanf("%d %lld",&x,&z);// wczytaj do x i z
scanf("%14s",tekst);// wczytaj max 14 znaków do tekst
scanf(" %14s",tekst);// jak wyżej, ale bez białych znaków na początku
scanf("%[^\n]s",tekst);// białe znaki poza \n są zapisane

complex double u=1+2*I;
printf("%u\n",sizeof(u));
printf("%f + i%f\n", creal(u), cimag(u));
printf("%f + i%f\n", u);// też działa
```

Do dyspozycji mamy również inne funkcje wypisujące lub wczytujące znaki:

```
int putchar(int character);
```

wypisuje jeden znak, zwraca ten sam znak lub **EOF** w przypadku błędu.

```
int puts(const char* lancuch);
```

wypisuje łańcuch znaków, zwraca liczbę nieujemną w przypadku powodzenia lub **EOF** w przypadku błędu.

```
int getchar();
```

pobiera jeden znak z klawiatury i zwraca go. Jeżeli zwracana wartość wynosi **EOF**, to znaczy, że wystąpił błąd odczytu znaku.
Przed C11 była dostępna również funkcja **gets**, ale ze względu na to, że mogła powodować przepełnienie bufora, została usunięta. Zamiast niej należy użyć:

```
char* fgets(char* lancuch, int ilosc, stdin);
```

która jest związana z plikami, zapisuje do podanego łańcucha znaków nie więcej niż **ilosc-1** znaków z wejścia lub do napotkania znaku końca linii (włącznie z tym znakiem!). Zwraca ona zero w przypadku niepowodzenia.
Powyższe funkcje mogą pozostawiać jakieś znaki na wejściu. Żeby je usunąć musimy pobierać wszystkie pozostałe znaki (czyli do '\n') np. używając:

```
void clean_stdin(void){
    for(char c=getchar();c!='\n' && c!=EOF;c=getchar());
}
```

Przykład

```c
#include <stdio.h>
int main (){
    char znak;
    puts ("Wpisz zdanie:\n");
    do{
        znak=getchar();
        putchar(znak);
    }while(znak!='\n');
    return 0;
}
```

Przykład

```c
#include <stdio.h>
int main (){
    char napis[16];
    puts ("Wpisz zdanie:\n");
    fgets(napis,16,stdin);
    printf("%s\n",napis);
    fflush(stdin);
    fgets(napis,16,stdin);
    printf("%s\n",napis);
    return 0;
}
```

Pętle

Pętle umożliwiają powtarzanie pewnego bloku instrukcji aż do napotkania warunku kończącego tę pętle. Możliwe jest też wymuszenie wewnątrz pętli jej zakończenie za pomocą polecenia **break** lub przejście do sprawdzenia warunku w pętli za pomocą **continue** (w pętli **for** wykonane zostają wcześniej instrukcje modyfikujące).
Rozróżniamy trzy rodzaje pętli (podano ich typowe zastosowania):

- **while** - nie znamy ilości powtórzeń i warunek sprawdzający jest wykonywany na początku;
- **do**...**while** - nie znamy ilości powtórzeń i warunek sprawdzający jest sprawdzany na końcu (pętla ma się wykonać co najmniej raz);
- **for** - znamy ilość powtórzeń, warunek sprawdzający jest wykonywany na początku.

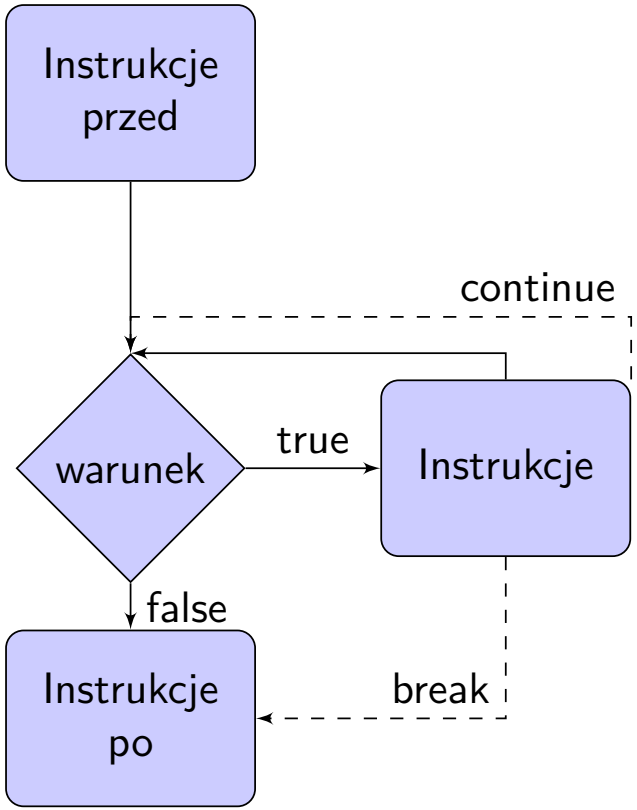
Każdą z powyższych pętli można symulować pozostałymi.

Instrukcja **while**

```
instrukcjaPrzed;
while(warunek){
    instrukcje;
    :
    :
}
instrukcjaPo;
```

gdy tylko jedna instrukcja w pętli:

```
instrukcjaPrzed;
while(warunek)
    instrukcja;
instrukcjaPo;
```



Przykład - Algorytm Euklidesa

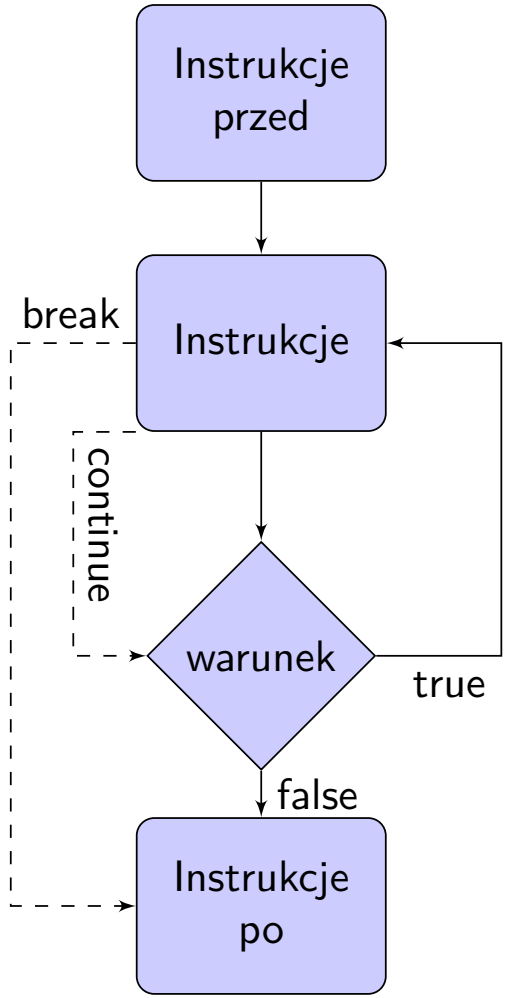
```
unsigned a,b;
printf("Podaj dwie liczby całkowite nieujemne\n");
scanf(" %u %u",&a,&b);
while(b!=0){
    unsigned r=a%b; //Obliczanie reszty
    a=b;
    b=r;
}
printf("NWD= %u\n",a);
```

Instrukcja **do...while**

```
instrukcjaPrzed;
do{
    instrukcje;
    :
    :
}while(warunek);
instrukcjaPo;
```

gdy tylko jedna instrukcja w pętli:

```
instrukcjaPrzed;
do
    instrukcja;
while(warunek);
instrukcjaPo;
```



Przykład - menu programu

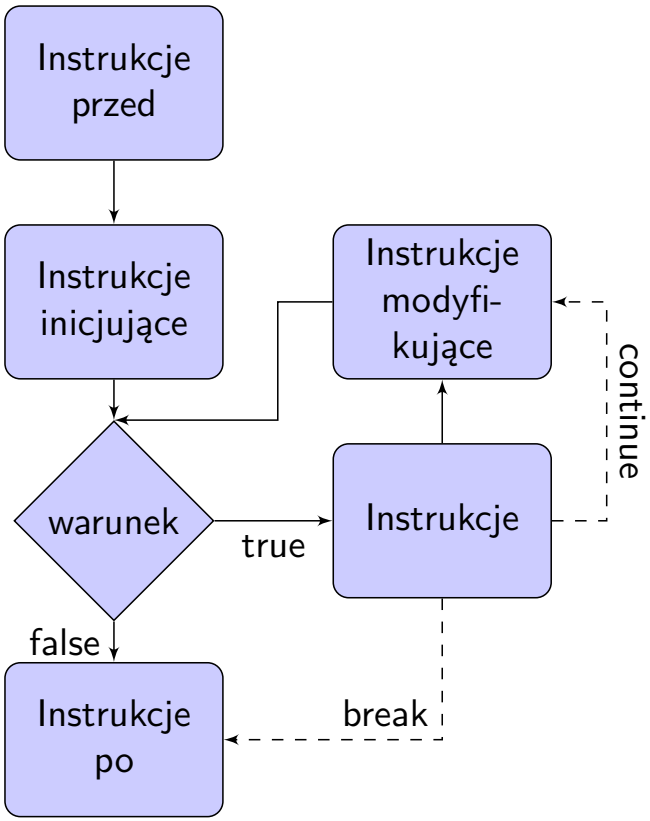
```
int main(){
    char znak;
    do{
        /*
            instrukcje w menu
        */
        printf("Zakonczyc program? (T/N): ");
        scanf(" %c",&znak);
    }while((znak!='t') && (znak!='T'));
    return 0;
}
printf("Koniec programu.\n");
```

Instrukcja **for**

```
instrukcjaPrzed;
for(InstrInicj;warunek;InstrModyf){
    instrukcje;
    ⋮
}
instrukcjaPo;
```

gdy tylko jedna instrukcja w pętli:

```
instrukcjaPrzed;
for(InstrInicj;warunek;InstrModyf)
    instrukcja;
instrukcjaPo;
```



- Można umieścić więcej niż jedną instrukcję inicjującą lub modyfikującą wewnątrz for. Należy oddzielać je przecinkami i czasem dać w okrągłych nawiasach.
- Instrukcje inicjujące, modyfikujące lub warunek pod **for** może być pusty.
- Od C99 w instrukcji inicjującej można definiować zmienne, które są dostępne w pętli. Zmienne występujące w instrukcjach inicjujących lub modyfikujących nazywa się zmiennymi sterującymi.

Przykład - obliczanie silni

```
unsigned n,silnia=1;
printf("Podaj liczbe naturalna\n");
scanf(" %u",&n);
for(int i=2;i<=n;i++)// int i od C99
    silnia*=i;
printf("%u!= %u\n",n,silnia);
```

Przykład - podwójna pętla i jej zapis równoważny za pomocą jednej

```
int i,j;
for(j=0;j<4;j++)
    for(i=0;i<4;i++)
        printf("%d,%d\n",i,j);
```

```
int i,j;
for((i=0, j=i);j<4;(j+=(++i/4),i%=4))
    printf("%d,%d\n",i,j);
```

Przykład - wypisanie liczb podzielnych przez 3 ale nie przez 4

```
unsigned n;
printf("Podaj liczbe naturalna\n");
scanf(" %u",&n);
for(int i=2;i<=n;i++){
    if(i%3!=0) continue;
    if(i%4!=0)
        printf("%u, ",n);
}
```

Kurs C

Radosław Łukasik

Wykład 2

Etykiety i skoki bezwarunkowe

Etykietą nazywamy napis składający się z liter alfabetu angielskiego, cyfr, oraz podkreślnika (nie może on zaczynać się od cyfry), zakończony dwukropkiem, wskazujący miejsce w kodzie. Polecenie

```
goto etykieta;
```

jest instrukcją skoku bezwarunkowego do miejsca wskazanego przez etykietę.

Przykład

```
    int x=5;
poczatek:
    printf("Jestesmy na poczatku\n");
    x++;
srodek:
    printf("Jestesmy w srodku\n");
    x++;
    if(x<8)
        goto poczatek;
    if(x<12)
        goto srodek;
    printf("Koniec\n");
```

Chociaż skokami bezwarunkowymi oraz **if**...**else** można symulować każdą pętle, to należy ich ostrożnie używać (niektórzy nawołują do całkowitego zrezygnowania z nich) ponieważ źle napisany kod z ich użyciem może zawiesić program lub spowodować błąd w jego działaniu. Przeważnie wszystkie zastosowania tej instrukcji można zastąpić bardziej czytelnymi poleceniami **if**...**else**, instrukcją wyboru **switch** oraz pętlami.

```
for(int i=0;i<20;i++){
    if(i%5 != 0)
        printf("%d\n",i);
}
```

```
int i=0;
poczatek:
    if(i>=20)
        goto koniec;
    if(i%5 != 0)
        printf("%d\n",i);
    i++;
    goto poczatek;
koniec:
```

Stos, sterta i inne segmenty danych

Stos jest to liniowa struktura danych w pamięci przydzielona do konkretnego programu, w którym dane są dokładane na jej wierzchołku i później z niego zdejmowane (bufor typu LIFO). Na stosie przechowywane są wszystkie niestatyczne zmienne lokalne oraz parametry funkcji, wartości zwracane przez funkcje i adresy instrukcji występujących po wywołaniu funkcji. Stos ma z góry ustalony rozmiar zależny od kompilatora (można go zmienić) i jest rzędu 1MB. Nadmierna rekurencja funkcji lub tworzenie dużej ilości zmiennych lokalnych może powodować do jego przepełnienia i zakończenia działania programu.
Sterta jest to obszar pamięci dostępny dla wszystkich programów, w którym można tworzyć dynamicznie zmienne.
Należy tutaj pamiętać, że zmienne globalne lub lokalne statyczne, które są zainicjowane są przechowywane w segmencie danych a te niezainicjowane w segmencie BSS. Może się też zdarzyć, że dane są ukryte w segmencie kodu (wartości przy operacjach arytmetycznych, czy też przyrównaniach, a także pewne łańcuchy znaków), który to jest tylko do odczytu. Przykładowo poniższy kod powoduje błąd naruszenia pamięci:

```
char *tekst="Tekst w segmencie kodu";
tekst[0]='a';// to powduje błąd
```

Tablice (statyczne)

Tablica jest ciągiem zmiennych tego samego typu. Nazwa tablicy jest zarazem wskaźnikiem na jej początek (czyli pierwszy element numerowany indeksem 0).
Tablice mogą być jedno lub wielowymiarowe. W przypadku tablic wielowymiarowych o wymiarach n_1 aż do n_k pozycja $[i_1][i_2]...[i_k]$ jest oddalona od początku tablicy o (w rozmiarze typu tablicy)

$$i_k + n_{k-1} * (i_{k-1} + n_{k-2} * (i_{k-3} + ... + n_1 * i_1)...).$$

Łańcuchem znaków nazywamy tablicę jednowymiarową typu **char** (ewentualnie **wchar_t**). Znak '\0' jest zarezerwowany do oznaczania końca łańcucha (który nie musi być końcem tablicy).
Deklarując tablicę (w sposób statyczny, na stosie) należy podać jej wszystkie wymiary (jako stałe liczby). W przypadku gdy tablica jest zainicjowana pewnymi wartościami możemy pominąć podawanie pierwszego wymiaru. Lista wartości początkowych tablicy musi zgadzać się z rozmiarem tablicy.
Od C99 można tworzyć tablicę o długości zdefiniowanej poprzez zmienną, przy czym takiej tablicy nie można zainicjować.
Również od C99 można inicjować tylko wybrane pola tablicy, reszta zostanie uzupełniona zerami.

```
typ nazwa[rozmiar] = { [wspolrzedna] = wartosc };
```

Rozmiar tablicy można uzyskać za pomocą **sizeof**, a ilość elementów poprzez podzielenie rozmiaru tablicy przez rozmiar typu tablicy (lub przez rozmiar pierwszego elementu).

Przykład

```
int tablica1[40];
double liczby[4]={ 1.0, 2.0, 3.0, 4.0 };
int power[]={ 1, 2, 4, 8, 16 };
int tablica2[3][2]={{1,2},{2,3},{3,4}};
    // można też tak   ={1,2,2,3,3,4};
    // gdy są podane wszystkie wymiary
char znaki[][10]={"zero","jeden","dwa"};
int tablica3[5]={[2]=16,[4]=11};// ={0,0,16,0,11}
```

Przykład

```
int tablica1[40];
for(int i=0;i<40;i++)
    tablica1[i]=2*i;

int tablica2[3][3];
for(int i=0;i<3;i++)
    for(int j=0;j<3;j++)
        tablica2[i][j]=i+j;
```

Wskaźniki

Wskaźnikiem nazywamy zmienną przechowującą adres pamięci, pod którym znajduje się zmienna pewnego typu (zmienna wskazywana). Deklarujemy go w następujący sposób:

```
typ_wskazywanej_zmiennej *nazwa_wskaznika;
```

Do wskaźnika możemy przypisać adres zmiennej używając operatora przypisania & lub przypisać do niego inny wskaźnik tego samego typu lub przypisać wskaźnik pusty (wartość 0 lub **NULL**). Do zmiennej możemy przypisać wartość wskazywaną przez wskaźnik (tego samego typu co zmienna) używając operatora wyłuskania *. Należy dbać o to by wskaźnik nie miał losowej wartości, bo wtedy jego użycie może spowodować błąd programu. Używanie pustego wskaźnika nie powoduje błędów, więc gdy już nie będziemy korzystać ze wskaźnika warto go wyzerować.

Przykład

```
int x=5;
int *pX =0; // pusty wskaźnik
pX=&x;      // pX wskazuje na zmienną x, czyli zawiera jej adres
int y=*pX;  // y=x
*pX+=5;     // x=10
```

Uwaga

Używając operatora wyłuskania i działań inkrementacji lub dekrementacji lepiej jest napisać operator wyłuskania wraz ze wskaźnikiem w nawiasach, bo kompilator może zrozumieć, że działamy na wskaźniku a nie na pamięci przez niego wskazywanej (co może powodować błędy).

```
int x=5;
int *pX =&x; // pX zawiera adres x
(*pX)++;     // zwiększamy x o 1
*pX++;  // zwiększamy wskaźnik o rozmiar typu wskaźnika
        // (rozmiar int = 4), pX nie wskazuje już na x
```

Każdy wskaźnik ma taki sam rozmiar. Jeżeli nie wiadomo na jaki typ ma dany wskaźnik wskazywać można go zrobić wskaźnikiem typu **void** (tak samo, gdy ma on obejmować wiele różnych typów), a następnie go rzutować na wskaźnik typu jaki potrzebujemy. Istnieje również wskaźnik na wskaźnik o następującej definicji:

```
typ **nazwa;
```

Może się on przydać np. w przypadku tablic wskaźników lub przekazywaniu wskaźników do funkcji, gdzie mogą być one zmienione.

Przykład

```
char a; //znak
char* b; //wskaźnik na znak
char** c; //wskaźnik na wskaźnik na znak
char d[10]; //tablica znaków
char* e[2]; //tablica wskaźnikow na znaki
char* (*f)[]; // wskaźnik na tablicę wskaźnikow na znak

//2 napisy po 10 znaków każdy
char tab[2][10]={"123456789","ABCDEFGHI"};
e[0]=tab[0];
e[1]=tab[1];
char** tabptr = e;

a=**tabptr;          //pierwszy znak pierwszego napisu
printf("%c\n",a);// 1
a=**(tabptr+1);      //pierwszy znak drugiego napisu
printf("%c\n",a);// A
a=*(*tabptr+1);      //drugi znak pierwszego napisu
printf("%c\n",a);// 2
a=*(*(tabptr+1)+1); //drugi znak drugiego napisu
printf("%c\n",a);// B
```

Alokacja pamięci na stercie

Aby tworzyć duże obiekty na stercie należy zarezerwować na nie pamięć. Służą do tego funkcje dostępne w bibliotece **stdlib.h**. Zwracają one wskaźnik na dowolny typ (wskaźnik ten wskazuje początek zarezerwowanego bloku pamięci), który musimy rzutować na typ, którego chcemy używać.

```
void* malloc (size_t size);//alokuje podaną ilość pamięci zwracając
                           //wskaźnik do niej (lub NULL gdy błąd)
void* calloc (size_t num, size_t size);//alokuje i zeruje num*size
     // bajtów pamięci zwracając wskaźnik do niej (lub NULL)
void* realloc (void* ptr, size_t size);//zwraca wskaźnik na pamięć
     // powstałą z ptr po zmianie rozmiaru na size w przypadku sukcesu
     // kopiuje* i zwalnia* pamięć pod starym wskaźnikiem
void free (void* ptr);// zwalnia pamięć pod adresem ptr
```

Przykład

```
int *wsk=(int*)malloc(1024*sizeof(int));// rzutowanie na int*
if(wsk!=0){
    /* jakieś operacje na pamięci */
}
free(wsk);
```

Należy pamiętać o tym by to co było alokowane za pomocą **malloc** (**calloc** czy też **realllloc**) było usuwane za pomocą **free**.

Należy też ostrożnie używać realokacji pamięci by nie doprowadzić do wycieku pamięci. Z tego względu dobrze jest użyć drugiego wskaźnika.

Przykład

```
int *wsk=(int*)malloc(1024*sizeof(int));// rzutowanie na int*
if(wsk!=0){
    /*
    jakieś operacje na pamięci
    */
    // chcemy zmienić rozmiar zarezerwowanej pamięci
    int *tmp=(int*)realloc(wsk,2048*sizeof(int));// dodatkowy wskaźnik
    if(tmp!=0){
        wsk=tmp;
        /*
        dalsze operacje na powiększonej pamięci
        */
    }else{
        // nie udało się zmienić rozmiaru, ale dalej możemy coś robić
        // na pamięci wskazywanej przez wsk
    }
}
free(wsk);
```

Liczby pseudolosowe

Często by testować pewne operacje na liczbach przydałoby się mieć je wypełnione pewnymi losowymi wartościami.
W bibliotece **stdlib.h** znajdują się funkcje związane z pseudolosowością.

```
srand(unsigned int ziarno);
```

funkcja ta inicjuje generator liczb pseudolosowych. Jeżeli **ziarno** jest stałe, to po każdym uruchomieniu będziemy mieć stały ciąg liczb pseudolosowych. Dlatego jako ziarno najlepiej stosować np. obecny czas w sekundach uzyskany za pomocą **time(0)** z biblioteki **time.h**. Inicjację generatora liczb losowych wystarczy wywołać raz na początku programu.

```
int rand();
```

funkcja ta zwraca wartość będącą liczbą pseudolosową w zakresie od 0 do **RAND_MAX** (ma to być co najmniej 15 losowych bitów, w Windows jest to $2^{15} - 1 = 32767$, a w Linux $2^{31} - 1 = 2147483647$).
Jeżeli chcemy mieć liczby losowe należące do jakiegoś przedziału $[m, n]$ ($n - m <$ RAND_MAX), to możemy to zrobić kodem:

```
liczba=m+(rand()%(n+1-m));
```

Przykład

```
int n=1024*1024;
int *tabl=(int*) malloc(n*sizeof(int));
if(tabl==0)
    printf("Brak pamieci\n");
else{
    long long srednia=0;
    for(int i=0;i<n;i++){
        tabl[i]=rand()%100;
        srednia+=tabl[i];
    }
    printf("srednia rand: %f\n",((double) srednia/n));
}
free(tabl);
```

Kopiowanie i porównywanie bloków pamięci i łańcuchów znaków

Gdy mamy dwie tablice i chcemy skopiować zawartość jednej do drugiej, to możemy do tego celu użyć pętli. Niestety nie jest to najefektywniejsze podejście. Lepiej jest skorzystać z pewnych zoptymalizowanych funkcji znajdujących się w bibliotece **string.h**. W tej bibliotece mamy również inne ciekawe funkcje, które tutaj przedstawimy.

```
memcpy(void *cel, const void *zrodlo, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** podaną **ilość bajtów** (**size_t** -typ całkowity bez znaku) do pamięci wskazanej przez **cel**.

```
memccpy(void *cel, const void *zrodlo, int znak, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** do pamięci wskazanej przez **cel** aż do napotkania **znaku** (**unsigned char**). Jeżeli go nie napotka, to kopiuje podaną **ilość bajtów**.

```
memmove(void *cel, const void *zrodlo, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** do pamięci wskazanej przez **cel** podaną **ilość bajtów**, przy czym pamięci te mogą na siebie zachodzić.

```
memset(void *cel, int wartosc, size_t ilosc_bajtow);
```

wypełnia podaną **ilość bajtów** pamięci wskazanej przez **cel** wskazaną **wartością** (**unsigned char**).

```
memchr(void *zrodlo, int wartosc, size_t ilosc_bajtow);
```

szuka pierwszego wystąpienia **wartości** (**unsigned char**) w pierwszej **ilość bajtów** pamięci wskazanej przez **źródło**.

```
int memcmp(const void *zrodlo1, const void *zrodlo2, size_t dlugosc);
```

porównuje zadaną **długość** bajtów w dwóch **źródłach** pamięci. Zwraca 0 gdy są równe, liczbę dodatnią gdy pierwszy bajt nie będący taki sam w obu blokach był większy w **pierwszym źródle**, wartość ujemna w przeciwnym wypadku.

Przykład

```
int n=1024;
char *pTab1=(int*) malloc(n);
char *pTab2=(int*) malloc(n);
if(pTab1==0||pTab2==0)
    printf("Za malo pamieci\n");
else{
    memset(pTab1,0,n);// zerujemy tablicę 1
    memcpy(pTab1,pTab2,n);// kopiujemy ją do tablicy 2
}
free(pTab1);
free(pTab2);
```

PetleTablic.c

Można oczywiście stosować wcześniejsze funkcje dla łańcuchów znaków zakończonych zerem, ale są dla nich dostępne specjalne funkcje.

```
strcpy(char *cel, const char *zrodlo);
```

kopiuje tekst ze **źródła** do **celu**.

```
strncpy(char *cel, const char *zrodlo, size_t dlugosc);
```

kopiuje podaną **ilość znaków** ze **źródła** do **celu** (gdy **ilość znaków** $<$ długość **źródła**, to trzeba dodać kod **cel[n]=0;**).

```
int strcmp(const char *lancuch1, const char *lancuch2);
```

porównuje dwa łańcuchy znaków ze sobą, zwraca 0 gdy są równe, liczbę dodatnią gdy pierwszy znak nie będący taki sam w obu łańcuchach był większy w **pierwszym łańcuchu**, wartość ujemna w przeciwnym wypadku.

```
int stricmp(const char *lancuch1, const char *lancuch2);
```

działa tak samo jak **strcmp** tylko ignoruje wielkość liter.
Powyższe dwie funkcje nie radzą sobie z polskimi znakami. Jeżeli chcemy porównywać z uwzględnieniem polskich liter, to musimy wcześniej zdeklarować, że używamy języka polskiego (używając polecenia **setlocale(LC_ALL,"Polish");** z biblioteki **locale.h**) a następnie skorzystać z odpowiedniej funkcji

```
int strcoll(const char *lancuch1, const char *lancuch2);
int stricoll(const char *lancuch1, const char *lancuch2);
```

będących kolejno odpowiednikiem **strcmp** i **stricmp**.

Przykład

```
char szA[32]={"Nowak Jan"};
char szB[32]={"Kowalski Piotr"};
char szTmp[32];
if(strcmp(szA,szB)>0){// sortowanie
    strcpy(szTmp,szA);
    strcpy(szA,szB);
    strcpy(szB,szTmp);
}
```

```
size_t strlen(const char *lancuch);
```

zwraca długość **łańcucha** znaków.

```
char * strchr(const char *lancuch, int znak);
```

szuka w **łańcuchu** pierwszego wystąpienia **znaku** (w ASCII). Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
char * strrchr(const char *lancuch, int znak);
```

szuka w **łańcuchu** ostatniego wystąpienia **znaku** (w ASCII). Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
size_t strspn(const char *lancuch1, const char *lancuch2);
```

zwraca indeks pierwszego znaku z **pierwszego łańcucha**, który nie występuje w **drugim łańcuchu** znaków. Jeżeli wszystkie znaki występowały, to zwraca długość **pierwszego łańcucha**.

```
size_t strcspn(const char *lancuch1, const char *lancuch2);
```

zwraca indeks pierwszego znaku z **pierwszego łańcucha**, który występuje w **drugim łańcuchu** znaków. Jeżeli żaden ze znaków nie występował, to zwraca długość **pierwszego łańcucha**.

```
const char * strstr(const char *lancuch1, const char *lancuch2);
```

szuka w **pierwszym łańcuchu** pierwszego wystąpienia **drugiego łańcucha**. Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
strcat(char *lancuch1, const char *lancuch2);
```

dopisuje **drugi łańcuch** na końcu **pierwszego łańcucha**.

```
strncat(char *lancuch1, const char *lancuch2, size_t  liczba_znakow);
```

dopisuje co najwyżej podaną **liczbę znaków** z **drugiego łańcucha** na końcu **pierwszego łańcucha**.

Większość z wymienionych funkcji z **string.h** ma swoje odpowiedniki dla typu **w_char_t** zamiast **char** i są one dostępne w bibliotece **wchar.h**.

Przykład

```
char napis[]="Numer1234wtekscie";
char cyfry[]="0123456789";
char lancuch[256];
lancuch[0]=0;
int pocz=strcspn(napis,cyfry); // pocz=5
if(pocz<strlen(napis)){
    int kon=strspn(napis+pocz,cyfry); // kon=4
    printf("Poczatek numeru: %d, dlugosc: %d\n",pocz,kon);
    strncpy(lancuch,(napis+pocz),kon); // lancuch="1234"
    lancuch[kon]=0;
}else
    printf("W tekscie nie ma numeru.\n");
strncat(lancuch,napis,5); // lancuch="1234Numer";
printf("%s,lancuch);
```

Przykład

```
char napis[]="Jakies krotkie zdanie. I kolejne.";
char *pocz,*kon;
char lancuch[256];
kon=strchr(napis,'.');// szukamy kropki
if(kon!=0){
    *kon=0;
    pocz=strrchr(napis,' ');// szukamy spacji
    if(pocz!=0)
        strncpy(lancuch,pocz+1,kon-pocz);
    else
        strncpy(lancuch,napis,kon-napis);
    printf("%s",lancuch);// "zdanie"
}else
    printf("Nie ma kropki.");
```

Funkcje

- Funkcja jest podprogramem, który wykonuje pewne konkretne zadanie.
- Funkcja może pobierać pewne argumenty, może także zwracać pewne wartości.
- Jeżeli w programie mamy kod, który często się powtarza, to warto zastąpić go funkcją.
- Poszczególne argumenty funkcji są oddzielone przecinkami.
- Rozróżniamy deklarację i definicję funkcji. Definicja polega na podaniu typu zwracanego przez funkcję (**void** gdy nic nie zwraca), argumentów funkcji wraz z ich typami, a także kodu samej funkcji. Deklaracja natomiast zawiera typ zwracany oraz typy argumentów (można podać również nazwy zmiennych). Jeśli funkcja jest wywoływana w kodzie po swojej definicji, to nie wymaga deklaracji.
- Funkcja zwraca wartość za pomocą **return** (wartość lub wyrażenie);
  Jeżeli funkcja nic nie zwraca, to albo używamy **return**; bez wartości zwracanej albo nie używamy go wcale.
- Wywołanie funkcji polega na podaniu wartości argumentów funkcji i przypisaniu wartości funkcji do zmiennej czy też wyświetleniu jej.
- Nazwy argumentów funkcji mogą być takie same jak zmiennych globalnych (przesłanianie).

```
typ nazwa(typ1, ...,typn);// deklaracja - sam prototyp funkcji
:
typ nazwa(typ1 arg1, ...,typn argn){// definicja
    /*                                // = prototyp + treść funkcji
        instrukcje
    */
    return (wyrażenie);
}
```

Przykład - n^n

```
#include <stdio.h>
void oblicz(unsigned);// deklaracja
void oblicz(unsigned n){// definicja
   int m=1;// złożoność czasowa O(n)
   for(int j=0;j<n;j++)// n iteracji
       m*=n;
   printf("%u ^ %u = %u\n",n,n,m);
}
int main(){
    unsigned n;
    printf("Podaj liczbe");
    scanf(" %u",&n);
    oblicz(n);// wywołanie funkcji
    return 0;
}
```

Przekazywanie parametrów poprzez wartości

- Przy wywołaniu funkcji tworzone sa lokalne kopie argumentów wejściowych (na stosie). W obliczeniach modyfikowane sa wartości tych kopii a nie samych zmiennych.
- Argumentami przekazywanymi podczas wywołania funkcji mogą być stałe, zmienne lub wyrażenia.
- Tablica jest wskaźnikiem, więc można ją modyfikować wewnątrz funkcji.
- W przypadku tablic wielowymiarowych konieczne jest podanie wszystkich jej wymiarów w argumencie funkcji.

Przykład

```
// double potega(double, int); - prototyp
int n=5;
potega(1.2,n); // stałe i zmienne
potega(1.2+2.3,n+4); // wyrażenia
```

Przekazywanie wartości poprzez wskaźniki

Uwaga W C++ występuje przekazywanie poprzez referencje ale nie występuje ono w C (wskaźniki i referencje to prawie to samo, referencje są łatwiejsze w użytkowaniu).

- Do funkcji przesyłane sa wskaźniki do argumentów
- zmienna w definicji funkcji jest wskaźnikiem i trzeba użyć operatora wyłuskania * aby zmodyfikować zmienną wewnątrz funkcji oraz operatora referencji & by przekazać funkcji adres zmiennej.
- Modyfikacje zmiennych wskazywanych wewnątrz funkcji sa widoczne po wykonaniu funkcji.
- Argumentami przekazywanymi do funkcji jako wskaźniki mogą być wyłącznie zmienne.

Przykład - przekazywanie poprzez wskaźniki

```
short tab[]={12,14,17,102};
short minimum; // najmniejszy element tablicy
int zaDuze; // liczba elemtów większych od 100
int funkcja(short *pminimum,short tablica[],int n){
    int j=0;
    (*pminimum) = tablica[0];
    for(int i=0;i<n;i++){ // n-rozmiar tablicy
        if(tab[i]>100){
            tab[i]=100; // zmniejszamy za duże elementy
            j++;        // i zliczamy je
        }else
            if(tablica[i]<*pminimum)
                *pminimum=tablica[i];
    }
    return j;
}
:
:
zaDuze=funkcja(&minimum,tab,4); // minimum=12, zaDuze=1
                                // tab={12,14,17,100}
```

Wskaźniki **const** w funkcjach

Wśród wskaźników mamy specjalny ich rodzaj - wskaźniki **const**. Służą do zabezpieczania zmiennych przed przypadkową zmianą. Mamy ich trzy rodzaje:

```
const typ_zmiennej *nazwa_wskaznika;
typ_zmiennej * const nazwa_wskaznika = wskaznik;
const typ_zmiennej * const nazwa_wskaznika = wskaznik;
```

W pierwszym przypadku wskaźnik wskazuje na stałą wartość (a więc nie można zmienić tej wartości). W drugim przypadku wskaźnik jest stały i nie można przypisać mu innego miejsca w pamięci (można za to zmieniać zmienną na którą wskazuje). Trzeci przypadek jest połączeniem dwóch wcześniejszych (nie możemy zmieniać ani zmiennej na którą wskazuje ani zmienić samego wskaźnika).

Przykład

```
int A=1,B=7,C=2,D=3;
void funkcja(int *const pA, const int *pB, const int *const pC){
    (*pA)+=(*pB)+(*pC); // możemy zmieniać wartość *pA, ale nie pA
    pB=&D; // możemy zmieniać pB ale nie wartość którą wskazuje
           // pC ani *pC nie może być zmienione
}
.
.
.
funkcja(&A,&B,&C); // A=10 reszta bez zmian
```

Będąc przy wskaźnikach występujących jako parametry funkcji warto wspomnieć, że oprócz kwalifikatora **const** mamy również kwalifikator **restrict** informujący kompilator, że pozostałe wskaźniki odwołują się do innego adresu pamięci. Dzięki temu kompilator może wygenerować możliwie najszybszy kod.

Kurs C

Radosław Łukasik

Wykład 3

Wskaźniki na funkcje

Wskaźniki mogą również wskazywać na funkcję. Dzięki temu możemy wykonywać różne czynności na tych samych danych podając tylko funkcję, którą chcemy wywołać. Utworzenie tego wskaźnika, przypisanie mu jakiejś funkcji oraz wywołanie wygląda następująco:

```
typ_funkcji (*nazwa_wsk_funkcji) (typ1 arg1, ... ,typn argn);
nazwa_wsk_funkcji=nazwa_funkcji;
zmienna=nazwa_wsk_funkcji(arg1, ... ,argn);
```

Taki wskaźnik na funkcję można również umieścić bezpośrednio w innej funkcji.

Przykład

```
int dodawanie(int a, int b){
    return a + b;
}

int mnozenie(int a, int b){
    return a * b;
}

int oblicz(int a, int b, int( *pDzialanie )(int, int)){
    return pDzialanie(a, b);
}

int main(){
    printf("Wynik mnozenia  = %d\n",oblicz(3, 4, mnozenie));
    printf("Wynik dodawania = %d\n",oblicz(3, 4, dodawanie));
    return 0;
}
```

Przy okazji wskaźników na wskaźnik mówiliśmy, że można je wykorzystać gdy chcemy zmienić nie zawartość tablicy a jej położenie czy rozmiar.

Przykład

```
void funkcja(int **pTab,int *n){
    if(*n<2048){
        int *tmp=(int*)realloc(*pTab,2048*sizeof(int));
        if(tmp!=0){
            *pTab=tmp;// zmieniamy wskaźnik tab z main()
            *n=2048;
            // jajkieś operacje na tablicy, o której już
        }   // wiemy, że ma rozmiar >=2048
    }
}

int main(){
    int n=1024;// rozmiar tablicy
    int *tab=(int*)malloc(n*sizeof(int));// początkowo 1024 el.
    if(tab!=0){
        funkcja(&tab,&n);
    }
    printf("Rozmiar %d\n",n);
    return 0;
}
```

Rekurencja funkcji

Przez rekurencję rozumiemy wywołanie funkcji wewnątrz niej samej (bezpośrednio lub poprzez inną funkcję). Używając jej należy się upewnić, że w pewnym momencie funkcja już nie będzie wywoływana oraz że ilość wywołań nie zapełni nam stosu (poprzez tworzenie zmiennych lokalnych i odkładanie parametrów funkcji na stosie).

Przykład

Rozpatrzmy funkcję obliczającą nam wyrazy ciągu Fibonacciego.

```
// deklaracja
int fib(int n);
:
:
// definicja
int fib(int n){//złożoność wykładnicza
    if(n<3){
        return 1;
    }else{
        return (fib(n-2)+fib(n-1));
    }
}
```

W niektórych przypadkach da się uniknąć rekurencji (zmniejszając złożoność pamięciową, a czasem nawet zmniejszając złożoność czasową) stosując inne rozwiązania.

Przykład

Dla ciągu Fibonacciego możemy zapamiętywać ostatnie dwa obliczone wyrazy ciągu zamiast liczyć je kilka razy. Da nam to złożoność liniową.

```
int fib(int n){//złożoność liniowa
    if(n<3){
        return 1;
    }else{
        int a=1;
        int b=1;
        int temp;
        for(int c=3;c<=n;c++){
            temp=b+a;
            a=b;
            b=temp;
        }
        return b;
    }
}
```

Kolejny przykład rekurencji, którą rozwiązaliśmy wcześniej uzyskując taką samą czasową złożoność liniową. W poniższym rozwiązaniu złożoność pamięciowa (stos) wynosi $\mathcal{O}(n)$.

Obliczanie silni

```
unsigned long long SILNIA(unsigned n){
    if(n<2)
        return 1;
    return n*SILNIA(n-1);
}
```

Funkcje o zmiennej liczbie argumentów

Uwaga w C++ występują funkcje o parametrach domyślnych. Standard C tego nie zawiera.
Mamy za to możliwość używania funkcji o zmiennej liczbie parametrów. Potrzebna jest do tego biblioteka **stdarg.h**. W samej funkcji za wszystkimi argumentami, które występują na stałe we funkcji dajemy 3 kropki.

```
typ nazwa(argumenty_stale, int ostatni_arg_staly, ... ){
    // treść funkcji
}
```

Żeby móc używać zmiennych wymienionych za ostatnim stałym parametrze najpierw musimy utworzyć listę i powiązać ją z naszymi dodatkowymi zmiennymi funkcji (występujące po ostatnim argumencie stałym)

```
va_list nazwa_listy;
va_start( nazwa_listy, ostatni_arg_staly );
```

Dostęp do kolejnych parametrów dostarcza nam funkcja zwracająca zmienną typu, który podamy

```
va_arg( nazwa_listy, typ_kolejnego_parametru );
```

Na sam koniec musimy po sobie posprzątać używając

```
va_end ( nazwa_listy );
```

Przykład

```c
#include <stdio.h>
#include <stdarg.h>

void PrintFloats(int n, ...){
    double val;
    printf("Wyswietlanie liczb:");
    va_list vl;
    va_start(vl,n);
    for (int i=0;i<n;i++){
        val=va_arg(vl,double);
        printf(" [%.2f]",val);
    }
    va_end(vl);
    printf ("\n");
}

int main (){
    PrintFloats(3,3.14,2,1.41421);
    return 0;
}
```

Funkcje inline

Funkcja **inline** (dostępna od C99) jest funkcją, której kod wstawiany jest w miejsce wywołania zamiast wywoływania tej funkcji. Jeżeli funkcja jest krótka i wywołana tylko kilka razy, to czasem lepiej wstawić jej kod w miejsce wywołania niż wywoływać ją jako funkcję. Nowoczesne kompilatory mimo użycia dyrektywy **inline** same decydują o tym czy funkcję traktować jako **inline** czy nie. Czasami mogą występować błędy podczas linkowania dla funkcji inline. Wystarczy wtedy taką funkcję uczynić statyczną (**static** przed **inline**).

Przykład

```
// deklaracja
inline float kwadrat(float);
:
:
// definicja
inline float kwadrat(float x){
    return (x*x);
}
:
:
float y=2;
float z=kwadrat(x); // ta linia zostaje przetłumaczona na kod
                    // float z=x*x;
:
:
```

Argumenty funkcji main

Funkcja **main** może mieć argumenty. Służą one jako parametry wywołania programu. Mają one postać:

```
int main(int argc, char *argv[]){
    ⋮
}
```

- Zmienna **argc** mówi o rozmiarze tablicy **argv** (o jeden więcej), pierwszy element wskazuje na nazwę programu (wraz ze ścieżką), ostatni element tej tablicy jest wskaźnikiem zerowym.
- Jeżeli są jakieś parametry wywołania programu, to **argc**$\geqslant 2$.

Przykład - wypisanie parametrów wywołania

```
#include <stdio.h>

int main(int argc, char *argv[]){
    for(int k = 0; k<argc; k++)
        printf("%s\n",argv[k]); // wypisujemy kolejne parametry
    printf("\nWcisnij enter, aby zamknac program.");
    char c;
    scanf("%c",&c);
    return 0;
}
```

Wyszukiwanie binarne

Można je stosować tylko do posortowanych tablic. Polega na porównywaniu poszukiwanego elementu z elementem w środku tablicy, a następnie zawężaniu przeszukiwanej tablicy do połówki w której może występować ten element.

```
int WyszukiwanieBinarne(int *tablica, unsigned n, int elem){
    unsigned pocz=0, koniec=n-1;
    while(pocz<=koniec){
        unsigned srodek=(pocz+koniec)>>1;
        if(elem<tabl[srodek]){
            koniec=srodek-1;
        }else if(elem>tabl[srodek]){
            pocz=srodek+1;
        }else return srodek; // zwraca pozycje lub
    }
    return n; // zwraca n gdy nie znaleźliśmy
}
```

Złożoność tego wyszukiwania wynosi $\mathcal{O}(\log_2 n)$.

Szybkie sortowanie - kod w C

```
void sortowanie_szybkie(int tab[], unsigned lewy, unsigned prawy){
    int i=(lewy+prawy)>>1;
    int j=lewy;
    int temp;
    temp=tab[i];tab[i]=tab[j];tab[j]=temp;
    for(i=lewy+1;i<=prawy;i++)
        if(tab[i]<tab[lewy])
            temp=tab[i];tab[i]=tab[++j];tab[j]=temp;
    temp=tab[lewy];tab[lewy]=tab[j];tab[j]=temp;
    if(lewy+1<j) sortowanie_szybkie(tab,lewy,j-1);
    if(j+1<prawy) sortowanie_szybkie(tab,j+1,prawy);
}
```

Powyższy algorytm wywołujemy z prawym końcem równym rozmiarowi -1, a lewym równym 0.

Istnieją pewne optymalizacje szybkiego sortowania, które zapobiegają złożoności czasowej $\mathcal{O}(n^2)$ i złożoności $\mathcal{O}(n)$ na stosie. Przykładem jest tu sortowanie introspektywne będące połączeniem sortowań szybkiego, przez kopcowanie i wstawianie. Ma ono złożoność czasową $\mathcal{O}(n \log n)$ oraz pamięciową $\mathcal{O}(\log n)$.

Sortowanie przez scalanie - kod w C

```
void sortowanie_scalanie(int tab[],int pom[],int i_p, int i_k){
    int i_s,i1,i2,i;    // wywołane z i_p =0, i_k =n
    i_s=(i_p + i_k)>>1;
    if(i_s-i_p>1)
        sortowanie_scalanie(tab,pom,i_p,i_s);
    if(i_k-i_s>1)
        sortowanie_scalanie(tab,pom,i_s,i_k);
    i1=i_p;
    i2=i_s;
    for(i=i_p;i<i_k;i++)
        pom[i] = ((i1 == i_s) || ((i2 < i_k) && (tab[i1]>tab[i2]))) ?
                    tab[i2++] : tab[i1++];
    for(i=i_p;i<i_k;i++)
        tab[i]=pom[i];
}
```

Istnieją wersje tego algorytmu niewymagające rekurencji (i przez to nieco szybsze). By przyspieszyć działanie, dla małych długości podtablic można wewnątrz wywołać sortowanie przez wstawianie zamiast sortowania przez scalanie. Można sprawdzić, że szybkie sortowanie jest nieco szybsze dla losowych danych nie zawierających zbyt wielu powtórzeń elementów, gdy wystąpień tego samego elementu jest dość dużo, to quicksort staje się dość powolny i może powodować przepełnienie stosu (rozmiar tablicy rzędu 1MB, liczby od 0 do 255 lub mniej).

Sortowanie przez kopcowanie - kod w C

```
void sortowanie_kopcowanie(int tab[],int n){
    int i,j,k,m,x,temp;
    while(i<n){// Budujemy kopiec
        k=i>>1; j=i; x = tab[i++];
        while((k>0) && (tab[k]<x)){
            tab[j] = tab[k]; j=k; k>>=1;
        }
        if(tab[0]<x){
            tab[j]=tab[0]; j=0;
        }
        tab[j] = x;
    }
    for(i=n-1;i>0;i--){// Rozbieramy kopiec
        temp=tab[0];tab[0]=tab[i];tab[i]=temp;
        j = 0; k = 1;
        while(k<i){
            m=k;
            if((k+1<i) && (tab[k+1]>tab[k])) m++;
            if(tab[m]<=tab[j]) break;
            temp=tab[m];tab[m]=tab[j];tab[j]=temp;
            j=m; k=m<<1;
        }
    }
}
```

Sortowanie kubełkowe - kod w C

```
void sortowanie_kubelkowe(int tab[],int n,int vmin,int vmax){
    int m=vmax-vmin+1;
    int *L=(int*)malloc(m*sizeof(int));
    if(L==0)
        printf("Pamiec pelna - ");
    else{
        memset(L,0,m*sizeof(int));//zerowanie pamięci
        int i=n,j=0;
        while(i>0)
            (L[tab[--i]-vmin])++;
        for(;i<m;i++)
            while(L[i]>0){
                tab[j++]= i+vmin;
                (L[i])--;
            }
    }
    free(L);
}
```

Wadą tego algorytmu jest zapotrzebowanie na pamięć, dla małych zakresów nie jest to problem, ale sortowanie to na pewno nie nadaje się do tablic o zakresie długości większej od 2^{29} ze względu na funkcję alokującą pamięć. Złożoność pamięciowa wynosi $\mathcal{O}(m)$, a dokładniej tworzymy tablicę o rozmiarze m.
Złożoność czasowa jest łatwa do policzenia - najpierw zerujemy tablicę liczników, co jest liniowo zależne od m. Następnie przebiegamy całą tablicę by zwiększać liczniki - robimy to w n krokach. Na samym końcu mamy dwie połączone pętle - zewnętrzna wykona się m razy, przy czym tylko dla n przypadków będziemy przypisywać wartość do tabeli.
Pamiętajmy, że powyższy algorytm działa tylko i wyłącznie dla liczb całkowitych (poprzednie algorytmy prosto przerabia się na liczby zmiennoprzecinkowe). Istnieją pewne modyfikacje, które pozwalają działać sortowaniu kubełkowemu również na liczbach zmiennoprzecinkowych (czy też pełnym zakresie **int**), ale jest to już bardziej skomplikowane choć pozwala na osiągniecie złożoności pamięciowej $\mathcal{O}(n)$ przy zachowaniu złożoności czasowej $\mathcal{O}(n)$. W rozwiązaniu tym kubełki są przedziałami (jest ich około n). Tworzymy również listę wszystkich wartości i wskaźników na najmniejszy element w kubełku (każda ma więc po n elementów). W sumie dla typu **int** potrzebujemy więc $3 * n*$**sizeof**(**int**) wolnej pamięci na stercie.

Sortowanie kubełkowe rozszerzone - kod w C I

```
typedef struct{// pomocnicza struktura
    int nastepnik;// indeks większego elementu
    int dane;// wartość
} lista;

void sortowanie_kubelkowe_ext(int *tab,int n,int vmin,int vmax){
    unsigned char k=0;
    long long zakres=vmax;zakres-=vmin;zakres++;// zakres zmiennych
    while(n<(zakres>>k)) k++;// szerokość kubełka to 2^k
    int *K=(int*)malloc(n*sizeof(int));
    lista *L=(lista*)malloc((n*sizeof(lista)));
    if(L==0 || K==0){
        printf("Pamiec pelna - ");
    }else{
        int i=0,j=0,ikb;
        memset(K,-1,n*sizeof(int));// kubełki puste
        for(;i<n;i++){// przypisujemy elementy do kubełków
            int we=tab[i];
            L[i].dane=we;
            ikb=(((unsigned)(we-vmin))>>k);// nr. kubełka
            int ip=-1;
            int ib=K[ikb];// ib = -1 lub indeks najmniejszego
```

Sortowanie kubełkowe rozszerzone - kod w C II

```
            while((ib>=0)&&(L[ib].dane<we)){
                ip = ib;
                ib = L[ib].nastepnik;
            }
            L[i].nastepnik=ib;
            if(ip==-1)
                K[ikb]=i;
            else
                L[ip].nastepnik=i;
        }
        for(ikb=0;ikb<n;ikb++){// sortowanie
            i=K[ikb];// i=-1 lub indeks najmniejszego elementu
            while(i>=0){
                tab[j++] = L[i].dane;
                i=L[i].nastepnik;// i=-1 lub kolejny element
            }
        }
    }
    free(L);free(K);
}
```

Podsumowanie sortowań

Można się pytać oczywiście jakie sortowanie mamy stosować znając rozmiar n tablicy i zakres m wartości z tablicy. Poniższa tabela została opracowana na podstawie wyników z mojego komputera i na innym sprzęcie może wyglądać inaczej (tabela typu **int**, dla **double** jest podobna, ale bez zwykłego sortowania kubełkowego).

n	m	wolna pamięć na stercie	sortowanie
$\leqslant 128$	-	-	wstawianie
> 128	$< \alpha_n n$	$\geqslant m$	kubełkowe
		$\in [3n, m)$	kubełkowe2
		$< 3n < m$	introspektywne
	$\geqslant \alpha_n n$	$\geqslant 3n$	kubełkowe2
		$< 3n$	introspektywne

Występujące tutaj α_n rośnie wraz z n od wartości 5 (poniżej 256tyś.) aż do 22 (powyżej 1mln). Wynika to z istnienia pamięci podręcznej procesora i jej wielkości jak i działania w tym samym czasie innych programów.

Sortowanie.cpp Sortowanie2.cpp

Funkcje matematyczne

Używając biblioteki **math.h** mamy dostęp do pewnych funkcji matematycznych. Większość z nich występuje dla wszystkich trzech typów liczb zmiennoprzecinkowych (**float**, **double**, **long double**), ale przedstawiając je tutaj będziemy w uproszczeniu mówić tylko o **double**.
Gdy w poniższych funkcjach występują kąty, to są one zawsze w radianach.

Funkcje trygonometryczne:

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);// x w [-1,+1]
double acos(double x);// x w [-1,+1]
double atan(double x);// wartość w [-pi/2,+pi/2]
double atan2(double y,double x);// atan(y/x), wartość w [-pi,+pi]
```

Funkcje hiperboliczne:

```
double sinh(double x);
double cosh(double x);// wartości w (1,+∞)
double tanh(double x);// wartości w (-1,1)
```

Funkcje wykładnicze i logarytmiczne:

```
double exp(double x);// e^x
double ldexp(double x,int exp);// zwraca x*2^exp
double frexp(double x,int* exp);// zwraca mantysę z [0,1) i wykładnik
                                // dla x=mantysa*2^exp
double log(double x);// ln x
double log10(double x);// log_10 x
```

Funkcje potęgowe:

```
double pow(double base,double exponent);// base^exponent
double sqrt(double x);// pierwiastek kwadratowy
```

Przybliżanie i reszty:

```
double floor(double x);// podłoga - cecha
double ceil(double x);// sufit
double fmod(double numer,double denom);// reszta z dzielenia obciętego
        // zwraca numer - trunc(numer/denom)*denom
```

wartość	(int)	floor	ceil
2.3	2.0	2.0	3.0
3.8	3.0	3.0	4.0
5.5	5.0	5.0	6.0
-2.3	-2.0	-3.0	-2.0
-3.8	-3.0	-4.0	-3.0
-5.5	-5.0	-6.0	-5.0

Inne:

```
double fabs(double x);// wartość bezwzględna
```

Stałe:

```
HUGE_VAL // maksymalna dodatnia wartość dla double
```

Jeżeli jest to nam potrzebne, to możemy sobie pomocniczo zdefiniować potrzebne nam ważne stałe matematyczne (można je otrzymać wykorzystując wcześniej omówione funkcje):

```
#define M_E   2.71828182845904523536
#define M_PI  3.14159265358979323846
```

Biblioteka complex.h

Biblioteka ta pozwala operować na liczbach zespolonych. Aby utworzyć taką liczbę zespoloną należy dodać do naszego typu (całkowitego lub rzeczywistego) dopisać typ zespolony (przed lub za tym typem). Funkcje **scanf** i **printf** niestety nie obsługują formatu zespolonego więc musimy sami zadbać o wczytanie i wyświetlanie liczb zespolonych, wykorzystując to, że liczba zespolona jest zapisywana jako tablica dwuelementowa.

```
double complex z;
complex int s=2+3*I;// I jest zdefiniowaną stałą w bibliotece
double * wsk=(double*)&z;
scanf("%lf %lf",wsk,wsk+1);
```

Wszystkie funkcje występujące w bibliotece mogą występować w trzech typach: operujące na **double** (domyślne), operujące na **float** (nazwa funkcji ma przyrostek "f") oraz operujące na **long double** (nazwa funkcji ma przyrostek "l").

Podstawowe operatory:

```
double creal(x);// zwraca część rzeczywistą
double cimag(x);// zwraca część zespoloną
double cabs(x);// zwraca moduł
double carg(x);// zwraca argument główny (w radianach)
double complex conj(x);// zwraca sprzężenie liczby zespolonej
```

Pozostałe funkcje zespolone:

```
double complex  cacos (x);
double complex  casin (x);
double complex  catan (x);
double complex  ccos (x);
double complex  csin (x);
double complex  ctan (x);
double complex  cacosh (x);
double complex  casinh (x);
double complex  catanh (x);
double complex  ccosh (x);
double complex  csinh (x);
double complex  ctanh (x);
double complex  cexp (x);
double complex  clog (x);
double complex  cpow (x, y);// x^y
double complex  csqrt (x);// cz. rzeczywista >=0
double complex  cproj (x);// rzutowanie na sferę Riemanna
```

Przykład

```
double complex x=-4;
double *wsk=(double*)&x;
printf("%f %+f i\n",creal(x),cimag(x));// -4.0 + 0.0 i
x=csqrt(x);
printf("%f %+f i\n",*wsk,*(wsk+1));// 0.0 + 2.0 i
```

Biblioteka ctype.h

W bibliotece tej mamy funkcję operującą na pojedynczych znakach.

Sprawdzanie rodzaju znaku:

```
int isdigit(int c);// czy znak jest cyfrą
int isxdigit(int c);// czy znak jest cyfrą w systemie szesnastkowym
int isalnum(int c);// czy znak jest cyfrą lub literą
int isalpha(int c);// czy znak jest literą
int islower(int c);// czy znak jest małą literą
int isupper(int c);// czy znak jest dużą literą
int isspace(int c);// czy znak jest białym znakiem
int isprint(int c);// czy znak jest znakiem drukowalnym
int isgraph(int c);// czy znak jest drukowalny i nie jest spacją
int iscntrl(int c);// czy znak jest znakiem kontrolnym
int ispunct(int c);// czy znak jest drukowalny ale nie alfanumerycznym
                   // i nie spacją
```

Wszystkie powyższe funkcje zwracają wartości logiczne ($0 =$ **false**).

Konwersja znaku:

```
int tolower(int c);// zmienia literę na małą
int toupper(int c);// zmienia literę na dużą
```

Funkcje te zwracają przekształconą literę.

Przykład

```
#include <ctype.h>
#include <stdbool.h>
int main(void){
    char znak;
    bool bznakprzed=true;
    puts("Podaj zdanie: ");
    do{
        znak=getchar();
        if(znak==EOF) break;
        if(isspace(znak)){
            if(!bznakprzed)
                putchar(' ');
            bznakprzed=true;
        } else if(isalpha(znak)){
            if(bznakprzed)
                znak=toupper(znak);
            putchar(znak);
            bznakprzed=false;
        } else if(!iscntrl(znak)){
            bznakprzed=false;
            putchar(znak);
        }
    }while(znak!='\n');
    return 0;
}
```

Biblioteka stdlib.h

Wcześniej wspominaliśmy o pewnych funkcjach dostępnych w bibliotece standardowej, teraz co nieco dopowiemy o innych funkcjach tam dostępnych.

Konwersja tekstu na liczbę:

```
double strtod (const char* str, char** endptr);
```

zwraca liczbę typu double zapisaną w łańcuchu znaków **str**, w **endptr** (o ile nie jest zerowy) zostaje zapisany wskaźnik na pierwszy znak z łańcucha, który nie był związany z konwertowaną liczbą. Łańcuch powinien zawierać cyfry (przed nimi może być znak), między którymi może wystąpić jedna kropka a także może zawierać litery "e" lub "E" oznaczające wykładnik (który również może zawierać znak). Ewentualnie liczba ta może być zapisana szesnastkowo z przedrostkiem "0x" lub "0X" (z kropką i wykładnikiem jak powyżej). W przypadku błędu konwersji funkcja zwraca 0, a w przypadku przekroczenia zakresu najbliższą + lub - maksymalną wartość dla **double** (**HUGE_VAL**).

```
float strtof (const char* str, char** endptr);
long double strtold (const char* str, char** endptr);
```

podobnie jak wcześniej tylko zwracają liczbę typu **float** lub **long double**.

```
long int strtol(const char* str, char** endptr, int base);
unsigned long int strtoul(const char* str, char** endptr, int base);
long long int strtoll(const char* str, char** endptr, int base);
unsigned long long int strtoull(const char* str, char** endptr, int
    base);
```

przekształcają liczbę zapisaną w tekście na odpowiedni typ całkowity (l - **long**, ll - **long long**, u - **unsigned**), **base** oznacza bazę systemu (od 2 do 26), w której zapisana jest liczba, przy czym wartość 0 oznacza, że przedrostek będzie mówił o tym jaki jest system (ósemkowy lub szesnastkowy). Liczba może zwierać cyfry lub litery (w zależności od systemu) a także prefiks "0x" lub "0X" w systemie szesnastkowym albo "0" dla ósemkowego. Funkcje te zwracają 0 w przypadku błędu lub najbliższą maksymalną/minimalną możliwą wartość danego typu, gdy przekroczony był zakres.
Wszystkie wymienione wcześniej funkcje usuwają z wejścia spacje poprzedzające zapisaną tekstowo liczbę.

Przykład

```
char szOrbity[] = "365.24 29.53";
char* pKoniec;
double d1, d2;
d1 = strtod(szOrbity, &pKoniec);
d2 = strtod(pEnd, NULL);
printf("Księżyc w ciągu roku okrąża Ziemię %.2f razy.\n", d1/d2);
```

Przykład

```
char szLiczby[] = "2001 60c0c0 -11011101001101001000000 0x6fffff";
char* pKoniec;
long int li1, li2, li3, li4;
li1 = strtol(szLiczby,&pKoniec,10);
li2 = strtol(pKoniec,&pKoniec,16);
li3 = strtol(pKoniec,&pKoniec,2);
li4 = strtol(pKoniec,NULL,0);
```

Oprócz wymienionych wcześniej funkcji konwertujących łańcuch na tekst mamy również pewne ich uproszczenia:

```
double atof(const char* str);
```

dla liczb rzeczywistych oraz

```
int atoi(const char* str);
long int atol(const char* str);
long long int atoll(const char* str);
```

dla liczb całkowitych o podanym typie.
Niestety w przypadku nieprawidłowych znaków wartość zwracana jest trudna do przewidzenia, więc lepiej je stosować gdy mamy pewność, że łańcuch znaków jest poprawny.
Żeby przekonwertować liczbę na ciąg znaków możemy wykorzystać **sprintf**:

```
char szLiczba[50];
float liczba = 23.34;
sprintf(szLiczba, "%f", liczba);
printf("Liczba na tekst: %s", szLiczba);
```

W bibliotece **stdlib.h** mamy jeszcze dwie ważne funkcje:

```
void qsort(void* tablica, size_t rozm_tab, size_t rozm_el,
           int (*f_por)(const void*,const void*));
```

pozwala na posortowanie (zmodyfikowany algorytm quicksort) tablicy elementów dowolnego typu. Należy podać wskaźnik na tablicę (**tablica**), rozmiar tablicy (**rozm_tab**) oraz rozmiar pojedynczego elementu (**rozm_el**). Ponadto musimy podać wskaźnik na funkcję (**f_por**), która porównuje dwa elementy naszej tablicy i zwraca 0 gdy są równe, wartość <0 gdy pierwszy parametr jest mniejszy niż drugi lub wartość >0 gdy pierwszy parametr jest większy niż drugi.
Druga funkcja, to:

```
void* bsearch (const void* klucz, const void* tablica,
               size_t rozm_tab, size_t rozm_el,
               int (*f_por)(const void*,const void*));
```

która wyszukuje binarnie w posortowanej tablicy dowolnego typu podanego przez nas elementu (**klucz**). Również tutaj wymagana jest funkcja porównująca dwa elementy z tablicy. Funkcja ta zwraca element pasujący do wyszukiwania lub 0 gdy nie udało się go odnaleźć.

Przykład

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char tablica[3][16] = {"Kowalski", "Nowak", "Chrobry"};
int f_por(const char* a, const char* b){
    return strcmp(a,b);
}

int main(void){
    char nazwisko[16];
    char* wsk;
    qsort(tablica,sizeof(tablica)/sizeof(tablica[0]),
          sizeof(tablica[0]),f_por);
    for(int i=0;i<3;i++)
        printf("%s\n",tablica[i]);
    printf("Podaj nazwisko do wyszukania: ");
    scanf("%16s",nazwisko);
    wsk= bsearch(nazwisko,tablica,sizeof(tablica)/sizeof(tablica[0]),
                 sizeof(tablica[0]),f_por);
    if(wsk!=0)
        printf("Znaleziono element %s\n",wsk);
    else
        printf("Nie znaleziono.\n");
    return 0;
}
```

W bibliotece **stdlib.h** jak i w innych (**stddef.h stdio.h string.h**, **time.h**, **wchar.h**) można napotkać typ całkowity bez znaku **size_t**. Jest on używany w różnych funkcjach (np. związanych z alokacją pamięci czy odczytem/zapisem do pliku) do przechowywania rozmiarów czy liczników. W zależności od architektury może on zajmować od 2 do 8 B. Jego maksymalną wartość można oczytać ze stałej **SIZE_MAX**. Wyświetlanie lub wczytywanie tego typu za pomocą **printf** i **scanf** jest możliwe dzięki "z" występującym przed jednym ze specyfikatorów **d, i, u, o, x, X** wyświetlających liczby całkowite.

Biblioteka stdint.h

Może się zdarzyć, że chcemy kompilować nasz kod na różnych architekturach (np. 32 lub 64 bitowych). Niektóre typy całkowite mają zmienną wielkość zależną właśnie od architektury. Aby mieć pewność, że nasze typy danych mają dokładnie tyle bitów ile chcemy możemy użyć biblioteki **stdint.h**. Znajdziemy w niej następujące typy całkowite:

ze znakiem	bez znaku	opis
int8_t	**uint8_t**	8 bitów
int16_t	**uint16_t**	16 bitów
int32_t	**uint32_t**	32 bity
int64_t	**uint64_t**	64 bity
intmax_t	**uintmax_t**	maksymalna dostępna ($\geqslant$ 64 bity)

W niektórych dziwnych konfiguracjach sprzętowych typy o ustalonej ilości bitów mogą nie istnieć. Z tego względu mamy do dyspozycji typy, które zamiast **int** zawierają **int_least** lub **int_fast**. Mają one co najmniej tyle bitów ile jest podanych, przy czym wersja "fast" ma wielkość dopasowaną do jak najszybszych działań na tych typach, a wersja "least" jak najmniejszy rozmiar.

Biblioteka inttypes.h

Wyświetlanie czy pobieranie typów z **stdint.h** może powodować czasem pewne problemy. Z tego względu w bibliotece **inttypes.h** mamy zdefiniowane pewne wartości, które służą do obsługi tych typów całkowitych w funkcjach **printf** i **scanf**:

przedrostek	typ	przyrostek	bity
PRI	d, i	brak	8, 16, 32, 64
	u	LEAST	
SCN	o	FAST	
	x	MAX	brak

Przykład

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
intmax_t duza;
uint32_t liczba;
scanf("%" SCNdMAX,&duza);
printf("duza = %" PRIdMAX ", liczba 32 bit = %" PRIu32, duza, liczba);
```

Operacje na plikach

Jeżeli chcemy wykonywać operacje na plikach, to możemy wykorzystać bibliotekę **stdio.h**. Do obsługi pliku potrzebny jest nam uchwyt (czyli wskaźnik) do pliku

```
FILE * nazwa_uchwytu;
```

Domyślnie mamy zdefiniowane 3 takie uchwyty: **stdin** - oznaczające standardowe wejście, czyli klawiaturę; **stdout** - oznaczające standardowe wyjście, czyli konsolę; **stderr** - standardowe wyjście błędów, które domyślnie też jest powiązane z wyświetlaniem w konsoli.
Zanim przejdziemy do otwierania plików, to powiedzmy sobie najpierw o innych możliwych operacjach. Plik możemy usunąć lub zmienić jego nazwę za pomocą funkcji:

```
int remove(const char* nazwa_pliku);
int rename(const char* stara_nazwa, const char* nowa_nazwa);
```

Obie funkcje zwracają 0, gdy operacja się powiedzie.

Aby otworzyć plik używamy funkcji

```
FILE* fopen(const char *nazwa_pliku, const char *tryb);
```

przy czym tryb mówi w jaki sposób ma być otwarty plik i może być równy:

"r"	odczyt z pliku, plik musi istnieć
"w"	zapis do pliku (nadpisywanie)
"a"	dopisywanie do pliku (na końcu)

Za powyższymi znakami mogą występować również (mogą być wymieszane w dowolnej kolejności)

"+"	oznacza, że plik jest otwarty do aktualizacji (odczyt i zapis)
"b"	oznacza tryb binarny otwarcia, czyli nietekstowy
"x"	od C11 tylko razem z "w", oznacza, że plik musi istnieć

W trybie dopisywania (czyli zawierającym "a") nie działają funkcje, które zmieniają aktualną pozycję w pliku. Jeżeli uda się otworzyć plik, to zwracany wskaźnik jest niezerowy.
Każdy otwarty plik należy zamknąć, gdy już nie będziemy z niego korzystać. Aby to zrobić należy wywołać funkcję:

```
void fclose(FILE* nazwa_pliku);
```

Jeśli tego nie zrobimy, to system operacyjny będzie uważał plik za otwarty co może uniemożliwić innym programom jego modyfikację.
Zamknięty uchwyt do pliku może być ponownie wykorzystany do otwarcia innego pliku.
Standardowe strumienie **stdin**, **stdout**, **stderr** nie zamykamy (o ile chcemy ich jeszcze użyć), chyba, że wcześniej zmieniliśmy je, tak by wskazywały na pliki.
Taką zmianę możemy zrobić za pomocą

```
FILE*freopen(const char* nazwa_pliku,const char* tryb,FILE *strumien);
```

gdzie tryb przyjmuje wartości jak dla **fopen**.

Przykład

```
#include <stdio.h>
int main (){
  freopen ("wyjscie.txt","w",stdout);
  printf("Zapisujemy do pliku a nie wypisujemy w konsoli.");
  fclose(stdout);
  printf("Teraz już wypisujemy w konsoli.");// nic się nie wyświetli
  return 0;
}
```

W trybie tekstowym do pliku możemy zapisywać za pomocą funkcji podobnych do znanych **printf** i **scanf**:

f		zapis do pliku
s		zapis do łańcucha znaków
v	printf	wyświetlenie zmiennej ilości danych
vf		zapis zmiennej ilości danych do pliku
vs		zapis zmiennej ilości danych do łańcucha znaków

Analogiczne funkcję występują dla **scanf**. Mają one składnię:

```
int fprintf(FILE* wsk_do_pliku, const char* format, ... );
int fscanf(FILE* wsk_do_pliku, const char* format, ... );
int vfprintf(FILE* wsk_do_pliku, const char* format, va_list arg );
int vfscanf(FILE *wsk_do_pliku, const char* format, va_list arg );
int sprintf(char* lancuch, const char* format, ... );
int sscanf(const char* lancuch, const char* format, ...);
int vsprintf(char* lancuch, const char* format, va_list arg );
int vsscanf(const char* lancuch, const char* format, va_list arg );
int vprintf(const char* format, va_list arg );
int vscanf(const char* format, va_list arg );
```

Funkcje wypisujące/zapisujące zwracają ilość znaków pomyślnie wypisanych/zapisanych. Funkcję wczytujące zwracają ilość zmiennych pomyślnie wypełnionych.

Przykład

```c
#include <stdio.h>
int main(){
    int len=0;
    char bufor[256];
    FILE *plik;
    plik=fopen("wyjscie.txt","r+");
    if(plik){
        printf("Otwarto plik. Pierwsze 26 znaki beda zmienione:\n");
        fscanf(plik,"%[26]s",bufor);
        bufor[26]=0;
        while(len<26){
            bufor[len]='A'+len;
            len++;
        }
        len=fprintf(plik,"%s",bufor);
        printf("Zapisano znakow %d\n",len);
    }else{
        printf("Nie udalo sie otworzyc pliku.\n");
    }
    fclose(plik);
    return 0;
}
```

Kurs C

Radosław Łukasik

Wykład 4

Znaki odczytywać lub zapisywać możemy również za pomocą innych funkcji:

```
int fgetc(FILE* wsk_do_pliku);// odczyt znaku
char* fgets(char *lancuch, int ilosc, FILE* wsk_do_pliku);// odczyt
                                                          // łańcucha
int fputc(int znak, FILE* wsk_do_pliku);// zapis znaku
int fputs(const char* lancuch, FILE* wsk_do_pliku);// zapis łańcucha
```

Funkcja **fgetc** zwraca aktualny znak, przesuwa pozycję w pliku o 1. W przypadku gdy nie można odczytać znaku (np. koniec pliku) zwraca wartość **EOF**, przy czym dla końca pliku ustawiony jest odpowiedni znacznik, który można zbadać za pomocą funkcji

```
int feof(FILE* wsk_do_pliku);// zwraca wartość !=0 gdy osiągnięto
                             // koniec pliku
```

lub w przypadku błędów odczytu (dla zapisu też działa ta funkcja)

```
int ferror(FILE* wsk_do_pliku);// zwraca wartość !=0 gdy
                               // wystąpił błąd
```

Czasem się zdarza, że pomimo wystąpienia błędu chcemy dalej kontynuować pewne operacje na pliku (np. wystąpił błąd zapisu, ale chcemy coś z pliku jeszcze odczytać). Wówczas musimy wyzerować znacznik błędu za pomocą

```
void clearerr(FILE* wsk_do_pliku);
```

Funkcja **fgets** odczytuje co najwyżej **ilosc -1** znaków, chyba, że wcześniej napotka znak końca linii (który nie jest odczytywany) lub dotrze do końca pliku. Zwraca wskaźnik na łańcuch znaków do którego zapisano znaki lub 0 gdy wystąpi błąd odczytu (włączając w to dojście do końca pliku).
Funkcja **fputc** zapisuje jeden znak do pliku. Zwraca zapisany znak lub w przypadku błędu **EOF** i ustawia znacznik błędu.
Funkcja **fputs** zapisuje ciąg znaków zakończony zerem (nie zapisuje znaku '\0'). Zwraca wartość niezerową w przypadku powodzenia lub 0 gdy pojawi się błąd (oraz ustawia znacznik błędu).
Jeżeli chcemy wiedzieć jaki dokładnie wystąpił błąd podczas otwierania pliku, odczytu lub zapisu do niego, to możemy to uzyskać korzystając z zdefiniowanego strumienia **stderror** oraz funkcji:

```
void perror(const char* tekst);
```

która wyświetla komunikat błędu poprzedzany podanym przez nas tekstem (może on być wskaźnikiem zerowym).

Przykład

```c
#include <stdio.h>
int main(){
    char znak;
    FILE *plik;
    plik=fopen("wyjscie.txt","r");
    if(plik){
        while((znak=fgetc(plik))!=EOF)
            putchar(znak);
        if(feof(plik))
            printf("\nOdczytano caly plik.\n");
        if(ferror(plik))
            printf("\nWystąpil blad odczytu.\n");
    }else
        perror("Nie udalo sie otworzyc pliku: ");
    fclose(plik);
    return 0;
}
```

Jeżeli chcemy odczytać lub zapisać coś do pliku binarnego, to możemy użyć

```
size_t fread(void* bufor, size_t rozmiar, size_t ilosc,
             FILE* wsk_do_pliku);
```

aby zapisać do bufora (tablicy) podaną **ilość** elementów, przy czym każdy element ma podany przez nas **rozmiar**. Podobnie dla zapisu mamy

```
size_t fwrite(const void* bufor, size_t rozmiar, size_t ilosct,
              FILE* wsk_do_pliku);
```

Funkcje te zwracają ilość prawidłowo odczytanych/zapisanych elementów. Jeżeli jest to ilość różna od podanej przez nas, to zostaje ustawiony znacznik błędu.

Przykład

```
#include <stdio.h>
int tablica[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int main(){
    FILE *plik; plik=fopen("dane.dat","wb");
    if(plik){
        if(fwrite(tablica,sizeof(int),16,plik)!=16)
            perror("Wystąpil blad zapisu: ");
    }else
        perror("Nie udalo sie otworzyc pliku: ");
    fclose(plik);
    return 0;
}
```

Wcześniejsze odczyty lub zapisy były zawsze od momentu w którym ostatnio skończyliśmy. Nie zawsze musi tak być. Możemy odczytywać lub przesuwać aktualną pozycję w pliku za pomocą:

```
int fseek(FILE* wsk_do_pliku, long int przesuniecie, int p_wyjsciowy);
```

gdzie przesuniecie jest liczbą całkowitą i oznacza pozycję względem punktu wyjściowego, który może być równy:

SEEK_SET	początek pliku
SEEK_CUR	aktualna pozycja w pliku
SEEK_END	koniec pliku

Funkcja ta zwraca 0 w przypadku powodzenia. W przypadku błędu zostaje ustawiony znacznik błędu. Może ona nie działać w plikach otwartych jako tekstowe.
Aby odczytać aktualną pozycję możemy użyć funkcji (może nie działać w plikach tekstowych):

```
long int ftell(FILE* wsk_do_pliku);
```

Mamy również funkcję, która pozwala nam skoczyć zawsze na początek pliku

```
void rewind(FILE* wsk_do_pliku);
```

Pomocne jest to szczególnie dla plików otwartych jednocześnie do odczytu i zapisu do przełączania się pomiędzy zapisem i odczytem.

Przykład

```c
#include <stdio.h>
char napis[]="\nKoniec pliku";
int main(){
    int len;
    FILE *plik;
    plik=fopen("wyjscie.txt","rb+");
    if(plik){
        fseek(plik,0,SEEK_END);
        len=ftell(plik);
        printf("Otwarto plik. Rozmiar: %d.\n",len);
        fwrite(napis,1,sizeof(napis)-1,plik);
        rewind(plik);
        fread(napis,1,sizeof(napis)-1,plik);
        printf("%s",napis);
    }else{
        perror("Blad: ");
    }
    fclose(plik);
    return 0;
}
```

W plikach tekstowych funkcje **fseek** i **ftell** mogą zwracać nieprawidłowe wartości z tego względu, że znaki mogą zajmować więcej niż jeden bajt w zależności od kodowania. Dla plików tekstowych mamy więc specjalne funkcje służące do odczytu i zapisu bieżącej pozycji w pliku:

```
int fgetpos(FILE* wsk_do_pliku, fpos_t* pos);// odczyt pozycji
int fsetpos(FILE* wsk_do_pliku, const fpos_t* pos );// ustawienie
    pozycji
```

Funkcje te zwracają 0 w przypadku powodzenia. Musimy w nich podać wskaźnik typu **fpos_t** skąd zostanie pobrana lub gdzie zostanie zapisana aktualna pozycja. Oczywiście zawartość tej zmiennej nie może nam powiedzieć gdzie dokładnie jesteśmy w pliku, ale możemy zapamiętywać i odtwarzać tą pozycję.

Przykład

```c
#include <stdio.h>
int main(){
    char znak;
    fpos_t pos;
    FILE *plik;
    plik=fopen("wyjscie.txt","r");
    if(plik){
        fgetc(plik);
        fgetpos(plik,&pos);
        znak=fgetc(plik);
        printf("2 znak: %c.\n",znak);
        fgetc(plik);
        fsetpos(plik,&pos);
        znak=fgetc(plik);
        printf("2 znak: %c.\n",znak);
    }else
        perror("Blad: ");
    fclose(plik);
    return 0;
}
```

Mamy również możliwość tworzenia plików tymczasowych. Pliki takie mają tworzoną pewną "losową" nazwę, tak by nie powtarzała się z nazwami innych plików oraz po zamknięciu są automatycznie usuwane. Zastosowanie tych plików to chwilowe zastąpienie pamięci RAM, gdy pośrednie wyniki zajmują więcej miejsca niż jest dostępne na stercie.
Aby utworzyć plik tymczasowy należy użyć funkcji:

```
FILE* tmpfile(void);
```

która tworzy i otwiera plik tymczasowy, a także zwraca wskaźnik do niego (zerowy wskaźnik oznacza niepowodzenie utworzenia pliku tymczasowego). Plik jest otwarty w trybie "wb+" i żeby go usunąć wystarczy go zamknąć funkcją **fclose**.
Możemy również tworzyć tymczasowe nazwy plików (nie będą się one powtarzać z istniejącymi już plikami):

```
char * tmpnam(char* str);
```

W funkcji tej podajemy wskaźnik na łańcuch znaków o długości takiej jak stała **L_tmpnam**. Jeżeli podamy wskaźnik zerowy, to zostaniem nam zwrócony wskaźnik na pewien bufor z nazwą tymczasową pliku (zawartość tego bufora może się zmieniać). Jeżeli podaliśmy wskaźnik na łańcuch znaków, to zostanie zwrócony ten wskaźnik, ewentualnie 0 w przypadku niepowodzenia.
Liczba gwarantowanych unikalnych tymczasowych nazw jest określona za pomocą stałej **TMP_MAX**.

Typy złożone

Zdarza się, że zamiast wartości zmiennych wolimy używać pewnych nazw z nimi związanych. Do tworzenia typu wyliczeniowego - czyli tablic z nazwami stałych typu **unsigned int** - służy polecenie **enum**:

```
enum nazwa_typu_wyliczenia{
    nazwa_wyliczenia0 = 7,     // wyliczenia oddzielamy przecinkiem
    nazwa_wyliczenia1 = 3,
    :
    nazwa_wyliczeniaN = 5     // po ostatnim wyliczeniu bez przecinka
};
```

Jeżeli przy pierwszym wyliczeniu nie ma przypisanej wartości, to przypisane zostaje do niej 0. Jeżeli przy którejś niżej ma miejsce brak przypisania, to przypisana zostaje wartość wcześniejszego wyliczenia powiększona o 1.
By przypisać zmiennej całkowitej wartość z naszego typu wyliczeniowego należy użyć kodu

```
zmienna=nazwa_wyliczenia;
```

Typ wyliczeniowy może służyć do definiowania zmiennych, przy czym zmienna ta może przyjmować nie tylko wartości będące wyliczeniami (w C++ jest inaczej).

Przykład

```
enum color{
    red =4,
    blue,     // blue =5
    white =8
};
:
int n=red; // przypisujemy do n wartość red, czyli 4
enum color zmienna;     // tworzymy zmienną typu color
zmienna = blue;    // zmienna = 5
```

Czasem zdarza się, że mamy zmienną składającą się z długiego słowa bądź z kilku słów. Możemy wówczas wprowadzić dla niej swoją krótszą nazwę używając polecenia **typedef**:

```
typedef typ_zmiennej nazwa;
```

Przykład

```
typedef unsigned int uint;
:
uint x=5;
```

Struktury

Czasem chcemy opisać jakieś rzeczy kilkoma parametrami. Przydałby się nam więc pewien obiekt w którym możemy mieć kilka zmiennych i to nawet różnych typów. To tego służy właśnie **struct**, przy czym mamy kilka możliwości:

```
typedef struct dluga_nazwa_struktury{// definiujemy nowy typ
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    :
    typ nazwa_elementuN;
}krotka_nazwa;
struct dluga_nazwa_struktury nazwa_zmiennej;
krotka_nazwa nazwa_zmiennej;
```

przy czym długa nazwa może być pominięta jeśli chcemy używać tylko krótkiej nazwy.

```
struct nazwa_struktury{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    :
    typ nazwa_elementuN;
};
struct nazwa_struktury nazwa_zmiennej;
```

```
struct nazwa_struktury{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
}nazwa zmiennej;
```

Żeby odwoływać się do składowych struktury używamy kropki między nazwą zmiennej a nazwą elementu

```
nazwa_struktury zmienna; //tworzymy nową zmienną
zmienna.nazwa_elementu=wartosc;
```

Możemy również zainicjować wartości początkowe naraz używając

```
nazwa_struktury zmienna={wartosc1,wartosc2,...,wartoscN};
```

gdzie kolejne wartości odpowiadają kolejnym elementom struktury.
Od C99 można inicjować również wybrane pola struktury oddzielone przecinkami.
Niewymienione pola są wypełnione zerami.

```
nazwa_struktury zmienna={.nazwa_elementu=wartosc};
```

Jeśli mamy wskaźnik na strukturę, to oprócz domyślnego dla wskaźników odwoływania się do elementów możemy również odwoływać się używając operatora wskazywania ->

```
nazwa_struktury *wskaznik;
(*wskaznik).nazwa_elementu=wartosc; // domyślny sposób
wskaznik->nazwa_elementu=wartosc;   // też działa
```

Mamy również inną przydatną rzecz - możemy skopiować zawartość jednej struktury do innej (tego samego typu) używając po prostu =

```
nazwa_struktury zmienna1,zmienna2;
:
:
zmienna2=zmienna1;     // kopiujemy wszystkie elementy
```

Należy tutaj pamiętać, że jeśli elementem struktury jest wskaźnik, to zostaje skopiowany wskaźniki i nie jest tworzona kopia zmiennej na którą on wskazuje, przez co należy przy zwalnianiu zadbać o wyzerowanie również wskaźnika ze skopiowanej struktury (lub utworzeniu kopii elementu wskazywanego).
Jeżeli chcemy wiedzieć ile pamięci zajmuje struktura, to możemy posłużyć się

```
sizeof(nazwa_struktury)
```

Gdy mamy tablicę złożoną ze struktur, to użycie inkrementacji lub dekrementacji na wskaźniku na element tej tablicy powoduje odpowiednio dodanie lub odjęcie od wskaźnika rozmiaru struktury. Dzięki temu możemy przesunąć się w tablicy na następny lub poprzedni jej element.

Struktura może nie mieć nazwy, o ile jest elementem innej struktury (unii lub klasy). Dzięki temu możemy odwoływać się do jej elementów jakby to były elementy jej struktury nadrzędnej (odpowiedni przykład będzie przy uniach). Jeżeli mielibyśmy dwie struktury, i każda z nich odwołuje się wewnątrz do tej drugiej, to należy wcześniej zadeklarować przynajmniej jedną z nich (tą późniejszą w kodzie), bo bez tego otrzymamy błąd kompilacji. Aby to zrobić wystarczy po prostu napisać

```
struct nazwa_dalszej_struktury;
```

W pierwszej strukturze musimy za to użyć tej nazwy struktury poprzedzonej słowem **struct**. Podobnie ma to miejsce gdy chcemy użyć wskaźnika na strukturę wewnątrz jej samej. Nie trzeba jej deklarować wcześniej, ale trzeba użyć **struct** z długą nazwą naszej struktury.
I na koniec najważniejsze - struktury mogą być zwracane poprzez funkcję. Możemy używać również struktur do przypisywania pewnych ustalonych wartości czy też zwracania pewnej ustalonej struktury poprzez wymienienie jej pól.

Przykład

```
typedef struct kolejka{// długa nazwa jest nam potrzebna
    char imie[32];
    char nazwisko[32];
    struct kolejka *nastepny;// bo się do niej tu odwołamy
}Kolejka;
```

Przykład

```
struct samochod;// bez tego błąd kompilacji
typedef struct {// definicja kierowcy
    char imie[32];
    char nazwisko[32];
    struct samochod *woz;
}kierowca;
typedef struct samochod{// definicja samochodu
    char marka[32];
    char model[32];
    kierowca *szofer;
}samochod;
```

Przykład

```
typedef struct {
    double x, y;
} punkt_t;

punkt_t suma(punkt_t A, punkt_t B){
    return (punkt_t){A.x+B.x, A.y+B.y};
}
```

Przykład

```
typedef struct {
    char imie[32];
    char nazwisko[32];
    short wiek;
    short wzrost;
}osoba;
:
:
osoba *radek=(osoba*)malloc(sizeof(osoba));
osoba klon={"Bezimienny","Klon",0,0};// inicjacja bezpośrednia
printf("Podaj imie i nazwisko: ");
scanf("%s %s",radek->imie,radek->nazwisko);
printf("Podaj wiek i wzrost: ");
scanf("%hd %hd",&(radek->wiek),&(radek->wzrost));
klon=*radek;
printf("%s %s ma ",klon.imie,klon.nazwisko);
printf("%hd lat i %hd cm wzrostu\n",klon.wiek,klon.wzrost);
free(radek);
```

Pola bitowe

Tworząc struktury możemy sprawić by zmienne całkowite zawierały tylko liczbę bitów, którą im ustalimy. Może to być przydatne np. w celu optymalizacji pamięci lub pewnych operacji na bitach, ale ma też swoje wady, bo działania wykonywane w ten sposób mogą być wolniejsze.
Pole bitowe tworzymy dodając za nazwą zmiennej dwukropek i liczbę bitów. Należy pamiętać, że typ liczby musi być wyłącznie całkowity (od **char** do **long long**) ze znakiem lub bez. Ten typ decyduje na starcie ile miejsca jest rezerwowanego dla tego pola bitowego i kolejnych pól bitowych tego samego typu (jeśli występują). Jeżeli kolejne pole bitowe przekroczyłoby wstępnie zarezerwowany rozmiar, to zostaje one przeniesione na początek nowej zmiennej tego typu. Jeżeli chcemy wymusić wyrównanie bitów do rozmiaru zmiennej podanego typu, to wystarczy utworzyć pole bitowe bez nazwy zmiennej i z :0 po nazwie typu.
Wyświetlanie bezpośrednie pól bitowych działa natomiast nie da się do nich wczytać bezpośrednio danych od użytkownika - trzeba to zrobić za pomocą pomocniczych zmiennych.

Przykład

```
typedef struct {// rok liczony od 1900
    unsigned short dzien:5;
    unsigned short miesiac:4; // 5+4=9<16
    unsigned short rok:7;     // 9+7=16
} pola_bitowe;// ta struktura zajmuje dwa bajty
pola_bitowe x;
x.rok=120;
x.dzien=15;
x.miesiac=8;
// scanf(" %hd",&(x.rok)); powoduje błąd
short temp;
scanf(" %hd",temp);
x.rok=temp;
printf("Rok:  %hd\n",x.rok);
printf("Miesiac:  %hd\n",x.miesiac);
printf("Dzien: %hd\n",x.dzien);
```

```
struct {
    char dzien:5;
    char miesiac:4;// 5+4=9>8
    char rok:7;// 4+7=11>8
} pola_bitowe;// ta struktura zajmuje trzy bajty
```

Unie

Unie są obiektami podobnymi do struktur ale z jedną zasadniczą różnicą: jej elementy nie są osobnymi zmiennymi tylko nachodzą na siebie (zaczynają się w tym samym miejscu w pamięci). Dzięki temu modyfikując jeden element zmieniamy jednocześnie wszystkie. Jest to pomocne gdy zmienna, którą chcemy przechować może przyjmować różne typy. Rozmiar unii jest maksymalnym rozmiarem jej elementów.

```
union nazwa_unii{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
};
```

```
typedef union dluga_nazwa_unii{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
}krotka_nazwa_unii;
```

Przykład

```
typedef union {
    unsigned DWORD;
    struct{// struktura bez nazwy
        short LOWORD;
        short HIWORD;
    };
} liczba;
liczba x;
x.DWORD=0x12345678;
printf("LICZBA: %#x\n",x.DWORD);
printf("HIWORD: %#x\n",x.HIWORD);
printf("LOWORD: %#x\n",x.LOWORD);
```

Biblioteka time.h

W bibliotece **time.h** mamy dostępne funkcje związane z pomiarem czasu i datą. Potrzebne nam będą następujące typy i struktury:

- **time_t** - służy do przechowywania czasu w sekundach, przeważnie liczonego od godziny 00:00 UTC, 1 stycznia 1970 roku;
- **clock_t** (=**long**) - przechowuje licznik cykli zegara, ilość cykli tego zegara na sekundę jest przechowywana w stałej **CLOCKS_PER_SEC**. Wynik może być zależny od systemu operacyjnego, w windows 7 zegar w przybliżeniu przyjmuje wielokrotności 16ms mimo, że ta stała wynosi 1000;
- struktura **tm** - służy do odczytywania daty i aktualnego czasu, ma następującą postać:

```
struct tm{
    int tm_sec;     // liczba sekund 0-59* (ewentualnie 61)
    int tm_min;     // liczba minut 0-59
    int tm_hour;    // liczba godzin 0-23
    int tm_mday;    // dzień miesiąca 1-31
    int tm_mon;     // miesiąc (od stycznia) 0-11
    int tm_year;    // rok od 1900
    int tm_wday;    // dzień tygodnia (0-niedziela ... 6-sobota)
    int tm_yday;    // dzień roku 0-364 lub 365
    int tm_isdst;   // >0 czas letni
};
```

W tej bibliotece mamy dostępne następujące funkcje:

```
time_t time(time_t * czas);
```

zwraca czas lokalny, jeżeli ***czas** jest różne od zera, to zapisuje tą wartość również do zmiennej **czas**.

```
double difftime(time_t czas2, time_t czas1);
```

zwraca różnicę w sekundach między **drugim czasem** a **pierwszym czasem**.

```
tm * gmtime(const time_t *czas);
```

zwraca wskaźnik na strukturę **tm**, której dane wyrażone są w czasie UTC.

```
tm * localtime(const time_t *czas);
```

zwraca wskaźnik na strukturę **tm**, której dane wyrażone są w czasie lokalnym.

```
time_t mktime(tm *dane_czasu);
```

przekształca lokalne **dane czasu** na czas typu **time_t**. Zwraca -1 gdy nie można przekształcić z powodu niepoprawnych danych. Ignoruje zawartość **tm_wday** i **tm_yday**.

```
clock_t clock();
```

zwraca liczbę cykli, które upłynęły od uruchomienia programu. Przydatna przy mierzeniu wydajności algorytmów.

Przykład

```
char dzien[7][16]={"niedziela","poniedzialek","wtorek","sroda",
                   "czwartek","piatek","sobota"};
struct tm * aCzas;
time_t czas=time(0);     // pobieramy czas
aCzas=localtime(&czas); // konwertujemy na czas lokalny
printf("Aktualny czas %d:%d:%d\n",(*aCzas).tm_hour,(*aCzas).tm_min,
       (*aCzas).tm_sec);
printf("Aktualna data %d/%d/%d\n",aCzas->tm_mday,aCzas->tm_mon,
       1900+aCzas->tm_year);
printf("%s, %d dzien roku\n", dzien[aCzas->tm_wday],aCzas->tm_yday);
```

Przykład

```
clock_t zegar1,zegar2;
zegar1=clock();
// kod algorytmu, jeśli zbyt szybki, to należy powtórzyć go pętlą
// tak, by dało się zmierzyć jego czas trwania
:
:
zegar2=clock();
printf("Czas trwania %fs\n",(double)(zegar2-zegar1)/CLOCKS_PER_SEC);
```

Jeżeli chcemy dokładniej odmierzać czas (lub cykle procesora), to należy użyć funkcji **___rdtsc** lub **___rdtscp** związanych z odczytem TSC (Time Stamp Counter), który przechowuje ilość instrukcji wykonanych od uruchomienia procesora. Instrukcje te, tak jak i inne instrukcje asemblerowe procesora, są dostępne po dodaniu odpowiedniej biblioteki zależnej od kompilatora: dla MSVC **intrin.h**, dla GCC **x86intrin.h**.

```
uint64_t __rdtsc();
uint64_t  __rdtscp(unsigned *IA32_TSC_AUX);// pod linuxem do
// IA32_TSC_AUX zapisuje numer wątku na którym wykonano instrukcję
```

Musimy pamiętać o dwóch ważnych rzeczach. Po pierwsze procesor swój czas przeznacza nie tylko na nasz program, stąd wyniki mogą być nieco zawyżone. Z drugiej strony tak działa system, więc jak tworzymy aplikację, to musimy to uwzględnić, więc nie ma co się tym przejmować. Po drugie mierzenie ilości cykli procesora nie musi zależeć od aktualnej częstotliwości procesora tylko może być wartością stałą związaną z bazową częstotliwością procesora. Informacje te da się wydobyć dla GCC instrukcją **___get_cpuid** dostępną w bibliotece **cpuid.h** lub dla MSVC instrukcją **___cpuid** z **intrin.h**. Można również użyć funkcji **clock()** wraz z **___rdtsc** by wyznaczyć przybliżoną częstotliwość procesora*.

Przykład

tsc.c

```c
#include <stdio.h>
#include <stdint.h>
#ifdef _MSC_VER // w zależności od kompilatora
# include <intrin.h> // dołączamy odpowiednią bibliotekę
#else
# include <x86intrin.h>
#endif
uint64_t readTSC();// nowsze wersje gcc wymagają deklaracji
uint64_t readTSCp();// funkcji inline lub poprzedzenia ją static
// prostsza wersja bez podkreślników z ewentualnymi poprawkami
inline uint64_t readTSC(){
//  _mm_mfence();// wymusza czekanie na wykonanie instrukcji przed
    uint64_t tsc = __rdtsc();
//  _mm_mfence();// wymusza poczekanie na wykonanie istrukcji rdtsc
    return tsc;
}
// jak wyżej, czeka na wykonanie instruckcji przed
inline uint64_t readTSCp(){
    unsigned dummy;// nie używamy, w nim zapiszemy IA32_TSC_AUX
    return __rdtscp(&dummy);
}
int main(){
    printf("Wartosc tsc: %llu\n",readTSC());
    printf("Wersja 2: %llu\n",readTSCp());
}
```

Biblioteki **intrin.h** lub **x86intrin.h** zawierają mnóstwo innych funkcji będących odpowiednikami instrukcji procesora. Można tam znaleźć m.in. pozwalające na używanie MMX, SSE, AVX czy też obrotów bitowych czy wyszukiwań bitowych. Instrukcje te mogą się przydać przy optymalizacji programu, lecz trzeba pamiętać, że są one zależne od sprzętu, więc musimy się upewnić, że platforma na której będzie uruchomiony program będzie miał te instrukcje procesora.
Trzeba dodać, że tutaj rpdzaj kompilatra decyduje czasem o tym gdzie ta funkcja jest i jaką ma składnie (MSVC i GCC mają tutaj czasem całkiem różne podejście).Przykładem mogą być tutaj obroty bitowe, które są dla MSVC są dostępne (w zależności od typy zmiennej całkowitej) jako **_rotl8**, **_rotl16**, **_rotl**, **_rotl64** (dla obrotów w prawo mamy **_rotr8**, **_rotr16**, **_rotr**, **_rotr64**), a w GCC jako **__rolb**, **__rolw**, **__rold**, **__rolq** (dla obrotów w prawo mamy **__rorb**, **__rorw**, **_rord**, **_rorq**).

Przydatnymi funkcjami mogą być wyszukiwania ustawianych bitów (z góry lub z dołu).

- Wyszukanie pierwszego ustawionego bitu w zmiennej **Mask** i zapisanie jego numeru do **Index**. Funkcje dla MSVC zwracają wartość 0 gdy brak ustawionego bitu, w pozostałych przypadkach zwracają wartość 1.

```
unsigned char _BitScanForward( unsigned long * Index, unsigned
    long Mask);// MSVC 32 bit
unsigned char _BitScanForward64( unsigned long * Index, unsigned
    long long Mask);// MSVC 64 bit
int __bsfd(int Mask);// GCC 32 bit
int __bsfq(long long Mask);// GCC 64 bit
```

- Wyszukanie ostatniego ustawionego bitu w zmiennej **Mask** i zapisanie jego numeru do **Index**. Funkcje dla MSVC zwracają wartość 0 gdy brak ustawionego bitu, w pozostałych przypadkach zwracają wartość 1.

```
unsigned char _BitScanReverse( unsigned long * Index, unsigned
    long Mask);// MSVC 32 bit
unsigned char _BitScanReverse64( unsigned long * Index, unsigned
    long Mask);// MSVC 64 bit
int __bsrd(int Mask);// GCC 32 bit
int __bsrq(long long Mask);// GCC 64 bit
```

Innymi przydatnymi funkcjami są:

- Kopiowanie **n** elementów tablicy **src** do tablicy **dst**. Rozmiar elementu tablicy to 1, 2, 4 lub 8 bajtów (przy czym 8 dla procesorów 64 bitowych), odpowiednio dla rozmiaru funkce te nazywają się **__movsb**, **__movsw**, **__movsd**, **__movsq**. Nie będziemy tu bardziej wchodzić w szczegóły tych funkcji, bo **memcpy** działa równie szybko jak one, a nawet szybciej (zależnie od kompilatora, czasem dopiero po włączeniu optymalizacji).
- Wstawianie podanej wartości **data** do **n** elementów tablicy **dst**. Rozmiar elementu tablicy to 1, 2, 4 lub 8 bajtów (przy czym 8 dla procesorów 64 bitowych). Pamiętajmy, że **memset** wypełnia tylko "jednobajtowo", co nie zawsze może nam odpowiadać. Poniższe funkcje działają równie szybko jak **memset**.

```
// dla MSVC
void __stosb(unsigned char* dst, unsigned char data, size_t n);
void __stosw(unsigned short* dst, unsigned short data, size_t n);
void __stosd(unsigned long* dst, unsigned long data, size_t n);
void __stosq(unsigned long long* dst, unsigned long long data,
    size_t n);
```

  Dla GCC niestety nie mamy dostępnych tych funkcji, musimy je sami zdefiniować używając wstawek asemblerowych.

```
static inline void __stosb(int8_t *Dest, int8_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosb" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosw(int16_t *Dest, int16_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosw" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosd(int32_t *Dest, int32_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosl" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosq(int64_t *Dest, int64_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosq" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}
```

bits.c

rep.c

Wyrównanie danych (C11)

Dane przetrzymywane w pamięci mogą zaczynać się w pewnych nieoptymalnych dla procesora miejscach, przez co dostęp do nich może być wolniejszy. Domyślnie dane są wyrównane do potęg liczby 2 odpowiadających rozmiarowi naszej zmiennej. W przypadku tablic jest to wyrównanie do wyrównania pojedynczego elementu tablicy. W przypadku struktur do wyrównania największego wyrównania spośród typów zmiennych w strukturze. Pamięć alokowana na stercie również jest domyślnie wyrównywana przynajmniej do największego wyrównania standardowych typów zmiennych (typ ten nazywa się **max_align_t** i jest dostępny w bibliotece **stddef.h**).
Wyrównanie danych można zmieniać jak i sprawdzać za pomocą funkcji z biblioteki **stdalign.h**.

```
alignof(typ_danych)
```

zwraca wyrównanie (jako typ **size_t**) podanego typu danych.

```
alignas(wyrównanie) typ_danych nazwa_zmiennej;
```

ustawia wyrównanie dla danej zmiennej.

Wyrównanie danych

align.c

```c
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
struct test{
    char var1;
    char var2[8];
} test;

struct testalign{
    char var1;
    alignas(8) char var2[8];
} testalign;

int main(void){
    printf("Wyrownanie max_align_t = %zu\n", alignof(max_align_t));
    printf("Wyrownanie int[13] = %zu\n", alignof(int[13]));
    printf("Wyrownanie test = %zu\n", alignof(test));
    printf("Rozmiar test = %zu\n", sizeof(test));
    printf("Wyrownanie test = %zu\n", alignof(testalign));
    printf("Rozmiar testalign = %zu\n", sizeof(testalign));
}// u mnie kolejno: 16, 4, 1, 9, 8, 16
```

Typy generyczne (C11)

W zależności od pewnych rzeczy, pod jedną nazwą chcielibyśmy mieć funkcje operujące na różnych typach zmiennych. Żeby to zrobić można użyć właśnie typów generycznych. Składnia wygląda następująco:

```
_Generic (nazwa, lista_powiazan);
```

gdzie elementy listy powiązań składa się z dwóch elementów

```
typ_danych: wyrazenie
```

Poszczególne elementy listy są oddzielone przecinkami. Typ danych może też być domyślny (**default**), który oznacza tutaj wszystkie pozostałe mogące się pojawić typy.

Przykład typu generycznego

generic.c

```c
#include <stdio.h>
#include <math.h>
#define cbrt(X) _Generic((X), \
    long double: cbrtl(X), \
    default: cbrt(X),  \
    float: cbrtf(X),  \
    int: llround(cbrt(X)) \
)

int main(void){
    double x = 8.0;
    int y=9;
    printf("cbrt(8.0) = %f\n", cbrt(x)); // 2.000
    printf("cbrt(9) = %lld\n", cbrt(y)); // 2
    return 0;
}
```

Funkcje bez powrotu i długie skoki (C11)

Istnieją funkcje, które nigdy nie powracają do miejsca ich wywołania. Takie funkcje tworzy się poprzez dodanie słowa **_Noreturn** lub makra **noreturn** (z biblioteki **stdnoreturn.h**) przed zwracany typ funkcji (który to oczywiście musi być **void**). W funkcji takiej oczywiście nie używamy **return** i kod musi albo zakończyć działanie programu (funkcja **exit**) albo wykonać długi skok do innego miejsca w programie. Długie skoki mają na celu obsługę nieoczekiwanych błędów i są analogiem obsługi wyjątków w innych językach programowania. Potrzebna nam będzie biblioteka **setjmp.h**. W niej mamy typ **jmp_buf**, a zmienna tego typu przechowuje nam adres do kodu wykonywanego przez procesor. Adres taki można zapisać do zmiennej używając funkcji

```
int setjmp(jmp_buf env);
```

która to zwraca 0 gdy nie jest wywołana w kodzie przez długi skok lub w przypadku użycia długiego skoku jest to wartość, która poprzez niego jest przekazywana (nigdy wtedy nie jest zerem).

Sam długi skok jest realizowany za pomocą

```
_Noreturn void longjmp( jmp_buf env, int status );
```

gdzie **env** jest zapisanym adresem powrotu do kodu, a **status** jest wartością do zwrócenia w funkcji **setjmp** (gdy jest ona równa 0, to zostaje poprawiona na wartość 1).

Przykład

```
#include <stdio.h>
#include <setjmp.h>
#include <stdnoreturn.h>

jmp_buf jump_buffer;

noreturn void a(int count){
    printf("a(%d) called\n", count);
    longjmp(jump_buffer, count+1); // setjmp zwróci count+1
}

int main(void){
    volatile int count = 0; // volatile ze względu na setjmp
    if (setjmp(jump_buffer) != 9)
        a(count++);
    return 0;
}// wyświetli się "a(0) called" do "a(8) called"
```

Przykład

longjmp.c

```c
#include <stdio.h>
#include <setjmp.h>
#define DIVIDE_EXCP 1
#define NEG_EXCP 2
double divide(const double x, const double y, jmp_buf env){
    if(y == 0){
        longjmp(env, DIVIDE_EXCP);
    }else if(y < 0){
        longjmp(env, NEG_EXCP);
    }
    return x/y;
}
void oldway(const double x, const double y){
    jmp_buf env;
    int catch_excp = setjmp(env);
    if(catch_excp == 0){
        printf("x/y = %f\n",divide(x, y, env));
    }else if(catch_excp == DIVIDE_EXCP){
        printf("Dzielenie przez zero jest niedozwolone\n");
    }else{
        printf("W sumie bez bledu, y<0\n");
    }
}
int main(int argc, char **argv){
    double x, y;
    printf("Podaj dwie liczby rzeczywiste: ");
    scanf(" %lf %lf",&x,&y);
    oldway(x, y);
    return 0;
}
```

Kurs C

Radosław Łukasik

Wykład 5

Wątki a procesy

Procesy:

- osobne programy wykonywane równolegle;
- każdy proces ma osobny stos, zmienne, pamięć.

Wątki:

- funkcje z jednego programu wykonywane równolegle;
- każdy wątek ma osobny stos;
- zmienne i pamięć są współdzielone.

Wielowątkowość w C

POSIX threads - realizowane przy pomocy biblioteki **pthread.h**, dostępne dla Windows, Linux, BSD, MacOS X. Jest to standaryzowana biblioteka (wersja na Windows https://sourceware.org/pthreads-win32/ nie jest!). MinGW zawiera własną wersję pliku **pthread.h** dla Windows, więc (o ile nie ma z nią jakichś problemów) nie trzeba dodawać dodatkowych bibliotek (pod linuxem w Code::Blocks należy dodać -pthread do Project Build Options -> Linker Settings -> Other linker options.

C11 threads - realizowane za pomocą biblioteki **thread.h**, bazuje na **POSIX threads**, działanie zależne od kompilatora i systemu operacyjnego (w Windows nie działa!).

Podstawowe typy i atrybuty związane z wątkami

Prawie wszystkie funkcje związane z POSIX zwracają 0 w przypadku sukcesu, w przeciwnym wypadku zwracają kod błędu, najczęstsze z nich, to **EINVAL** - nieprawidłowa wartość parametru (któregokolwiek), **ENOMEM** - brak pamięci do wykonania operacji, **EPERM** - odmowa dostępu (np. zbyt niski priorytet procesu), **EBUSY** - zablokowany wątek, mutex itp., **EDEADLK** - wykryto zakleszczenie (deadlock), **EAGAIN** - wyczerpanie dostępnych zasobów (nie pamięć), osiągnięcie limitu rekurencji, **ENOTSUP** - nie obsługiwany parametr.
Zanim przejdziemy do funkcji tworzących wątek zacznijmy od przedstawienia podstawowych typów z nimi związanymi:

- **pthread_t** - identyfikator wątku;
- **pthread_attr_t** - atrybuty wątku, odczytywane lub ustawiane pośrednio poprzez odpowiednie funkcje zaczynające się odpowiednio od **pthread_attr_get** i **pthread_attr_set**. Każda zmienna przechowująca atrybuty wątku musi być przed użyciem zainicjowana, a gdy przestaje być potrzebna, to należy zwolnić zasoby z nią powiązane. Robi się to funkcjami:

```
int pthread_attr_init(pthread_attr_t *attr);// inicjacja
int pthread_attr_destroy(pthread_attr_t *attr);// zwalnianie
```

Funkcje odczytujące/zmieniające atrybuty wątku (wszystkie poniższe funkcje zwracają 0 w przypadku sukcesu):

- rodzaj wątku:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

  zmienna **detachstate** może być jedną z dwóch wartości: **PTHREAD_CREATE_JOINABLE** (domyślna) lub **PTHREAD_CREATE_DETACHED**. Wątki typu **JOINABLE** nie usuwają po sobie danych (byśmy mogli z nich jeszcze skorzystać) i musimy sami to zrobić za pomocą funkcji **pthread_join**. Wątki typu **DETACHED** automatycznie czyszczą po sobie pamięć. Każdy wątek typu **JOINABLE** można przekształcić na typ **DETACHED** (ale nie odwrotnie!) za pomocą funkcji:

```
int pthread_detach(pthread_t id);
```

Wątki mogą być wykonywane z różnymi priorytetami. Aby uszczegółowić te rzeczy możemy użyć poniższych funkcji:

- dziedziczenie ustawień:

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int *inheritsched);
```

  Zmienna **inheritsched** przyjmują jedną z dwóch wartości: **PTHREAD_INHERIT_SCHED** - domyślna, oznacza dziedziczenie rodzaju szeregowania z wywoływanego wątku lub **PTHREAD_EXPLICIT_SCHED** - oznacza, że parametry są zapisane w atrybutach.

- polityka przełączania się na liście wątków:

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr,int *policy);
```

  gdzie **policy** może przyjmować wartości **SCHED_OTHER** - domyślny, **SCHED_FIFO**, **SCHED_RR** - cykliczna lista wątków (przełączanie po pewnym czasie na kolejny) Ogólnie rzecz biorąc zdarza się, że działa tylko **SCHED_OTHER**. Wynika to z tego, że pozostałe opcje wymagają uprawnień na poziomie system.

- priorytet wątku:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
            const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
            struct sched_param *param);
int pthread_setschedparam(pthread_t thread, int policy,
            const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *policy,
            struct sched_param *param);
```

Zmienna **policy** oznacza politykę, **sched_param** jest strukturą, która zawiera co najmniej jedno pole **sched_priority** typu **int**, oznaczające priorytet, przy czym wartość ta powinna być pomiędzy minimalną i maksymalną możliwą, które można pobrać za pomocą funkcji:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Pod linuxem przy polityce **SCHED_OTHER** powyższe dwie funkcje zwracają 0 i jest to jedyny priorytet do wyboru.

- rozmiar stosu wątku:

```
int pthread_attr_setstacksize(pthread_attr_t *attr,size_t size);
int pthread_attr_getstacksize(pthread_attr_t *attr,size_t *size);
```

size=0 oznacza taki sam rozmiar stosu jak w wątku tworzącym. Podawany rozmiar musi być między pewna minimalną i maksymalną wartością na którą zezwala system.

Tworzenie wątku

Aby utworzyć wątek możemy użyć funkcji:

```
int pthread_create(pthread_t *id, const pthread_attr_t *attr,
                   void* (*fun)(void*), void* arg);
```

W **attr** należy podać atrybuty, o których mowa była wcześniej. Jeśli podamy tutaj wartość **NULL**, to będą to domyślne parametry. Należy również podać funkcję, która będzie wykonywana w wątku, ma ona jeden parametr-wskaźnik, przez który możemy przesłać jakieś dane. Zwraca również wskaźnik na dowolny typ, poprzez który możemy coś zwracać o ile jest taka potrzeba. Po zwróceniu wartości wątek kończy swoje działanie. W **id** zostanie zapisany identyfikator utworzonego wątku. Dla wątków możliwe mamy sprawdzenie czy są to te same za pomocą:

```
int pthread_equal(pthread_t id1, pthread_t id2);// 0 gdy różne
```

lub pobranie samego identyfikatora wątku wewnątrz jego funkcji:

```
pthread_t pthread_self();
```

Istnieje również funkcja, która wymusza (wewnątrz wątku) zakończenie działania wątku:

```
void pthread_exit(void *value_ptr);
```

wartość podana jako parametr będzie zwrócona na zakończenie wątku. Należy tutaj pamiętać, że jeżeli główny proces zakończy swe działanie, to wszystkie jego watki zostaną zmuszone do zakończenia swego działania.
W głównym procesie programu może się zdarzyć, że będziemy czekać aż pewne wątki zakończą swe działanie. Do tego służy funkcja:

```
int pthread_join(pthread_t id, void **value_ptr);
```

która czeka na zakończenie wątku **id** i zapisuje wskaźnik na wartość przez niego zwracaną.
Jeżeli wewnątrz wątku chcemy oddać sterowanie do innych wątków, to możemy to zrobić za pomocą:

```
int sched_yield(void);
```

przy czym może się zdarzyć, że funkcja z powrotem uruchomi wątek ją wywołujący.

Watki (slajd 1)

Watki.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <x86intrin.h>
#include <time.h>
#define N 16
#define ROZM 1024*64
typedef struct{
    unsigned long long cykle;
    unsigned int rdzen;
    int prior;
    int* tab;
}wynik;

struct sched_param param;
pthread_attr_t attr;
pthread_t id[N];// miejsce na identyfikatory watków
wynik wyniki[N];
int i,zwrot,pmin,pmax;
```

Watki (slajd 2)

```
void* watek(void* _arg){// funkcja wątków
    unsigned long long licz1, licz2;// przechowywanie liczników cykli
    int polityka, *tab, stala=640000;
licz1=__rdtscp(&(((wynik*)_arg)->rdzen));// odczyt cykli i rdzenia*
    pthread_getschedparam(pthread_self(), &polityka, &param);
    ((wynik*)_arg)->prior =param.sched_priority;
    tab=((wynik*)_arg)->tab;
    if(tab!=0){
        for(int i=0;i<ROZM;i++) tab[i]=rand();
        int wi,wj,value;
        for(wi=1;wi<ROZM;wi++){// sortowanie przez wstawianie
            value=tab[wi];
            for(wj=wi-1;wj>=0 && tab[wj]>value;wj--)
                tab[wj+1]=tab[wj];
            tab[wj+1]=value;
            if(wi*wi==stala){// pozwalamy innym wątkom
                sched_yield();// na włączenie się
                stala+=stala;
            }
        }
    }
licz2=__rdtscp(&(((wynik*)_arg)->rdzen));// odczyt cykli i rdzenia*
    ((wynik*)_arg)->cykle =licz2-licz1;// zapis cykli
    return 0;
}
```

Watki (slajd 3)

```
int main(){
    srand(time(0));
    for(i=0;i<N;i++)// rezerwacja miejsca na tablice
        wyniki[i].tab=malloc(ROZM*sizeof(int));
    pthread_attr_init(&attr);// inicjacja atrybutów
    pmin=sched_get_priority_min(SCHED_OTHER);// pobranie minimalnego
    pmax=sched_get_priority_max(SCHED_OTHER);// i maks. priorytetu
    printf("Zakres priorytetow %d - %d\n",pmin,pmax);
    for (i=0;i<N;i++){/* utworzenie kilku wątków */
        param.sched_priority=((pmax-pmin)*i)/(N-1)+pmin;// priorytety
        pthread_attr_setschedparam(&attr,&param);
        zwrot = pthread_create(&id[i], &attr, watek, &wyniki[i]);
        if(zwrot) printf("Nie powiodło sie pthread_create\n");
    }
    for (i=0;i<N;i++){/* oczekiwanie na zakończenie wątków */
        if(pthread_join(id[i], NULL)) perror("Blad watku");
        else{
            printf("Watek: %2d, prior. %3d, cykle %11llu, rdzen %u\n",
    i,wyniki[i].prior,wyniki[i].cykle,wyniki[i].rdzen);
            free(wyniki[i].tab);// zwalniamy pamięć tablic
        }
    }
    pthread_attr_destroy(&attr);// zwalniamy miejsce atrybutów wątków
    return 0;
}
```

Mutexy

Mutex (MUTual EXclusion) - jest mechanizmem wzajemnego wykluczania, który chroni dane współdzielone przez kilka wątków. W danym momencie, tylko jeden wątek może mieć dostęp do danych współdzielonych.
Z Mutexami w pthreads związane są struktury: **pthread_mutex_t** dla samego mutexu oraz **pthread_mutexattr_t** opisującej jego atrybuty.
Najpierw należy zainicjować mutex za pomocą:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutattr);
```

przy czym atrybuty mogą być wskaźnikiem **NULL**. Gdy już przestaje nam potrzebny nalezy go zwolnić za pomocą:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mamy również możliwość statycznej inicjacji mutexu podczas kompilacji przypisując do niego jedną ze stałych: **PTHREAD_MUTEX_INITIALIZER** - zwykły, **PTHREAD_RECURSIVE_MUTEX_INITIALIZER** - rekursywny, **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER** - bezpieczny. Mutexy takie nie wymagają obsługi błędów, ale również na koniec należy je zwolnić za pomocą **pthread_mutex_destroy**.
Wątek może pozyskać blokadę na trzy sposoby:

- "do skutku" - czeka w nieskończoność aż mutex zostanie zwolniony poprzez inne wątki:

```
int pthread_mutex_lock(pthread_mutex_t *mut);
```

- jednorazowa - jeżeli mutex jest zablokowany, to zwraca od razu błąd **EBUSY**:

```
int pthread_mutex_trylock(pthread_mutex_t *mut);
```

- "ograniczona czasowo" - czeka określoną ilość czasu aż mutex zostanie zwolniony poprzez inne wątki, gdy się nie doczeka zwraca błąd **ETIMEDOUT**:

```
int pthread_mutex_timedlock(pthread_mutex_t *mut,
                            const struct timespec *timeout);
```

gdzie struktura **timespec** ma dwie składowe **tv_sec** typu **time_t** oraz **tv_nsec** typu **long** oznaczające kolejno czas w sekundach plus czas w nanosekundach, w którym należy zakończyć oczekiwanie (czas bezwzględny, do pobrania aktualnego czasu użyjmy **time(0)**).

Zwalnianie blokady następuje zawsze funkcją:

```
int pthread_mutex_unlock(pthread_mutex_t *mut);
```

Jeżeli chodzi o atrybuty mutexu, to żeby ich użyć należy najpierw je zainicjować:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mutattr);
```

natomiast, gdy już go nie potrzebujemy, to należy go usunąć za pomocą:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mutattr);
```

Do dyspozycji mamy następujące atrybuty:

- współdzielenie mutexu z innymi procesami:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mutattr,
                                 int pshared);
int pthread_mutexattr_getpshared(const pthread_mutexattr_t
                                 *mutattr, int *pshared);
```

gdzie **pshared** może przyjmować wartości **PTHREAD_PROCESS_PRIVATE** (domyślny, bez współdzielenia) lub **PTHREAD_PROCESS_SHARED** (współdzielenie).

- typ mutexu:

```
int pthread_mutexattr_settype(pthread_mutexattr_t *mutattr,
                              int type);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *mutattr,
                              int *type);
```

  gdzie typ jest jednym z:

  - **PTHREAD_MUTEX_NORMAL** - normalny, nie wykrywa zakleszczeń (wątki czekające nawzajem na zakończenie drugiego). Wątek blokujący powtórnie bez wcześniejszego odblokowania powoduje zakleszczenie. Odblokowywanie niezablokowanego lub zablokowanego przez inny wątek mutexu powoduje nieprzewidziane zachowanie.
  - **PTHREAD_MUTEX_ERRORCHECK** - bezpieczny, pozwala na zwracanie błędów w przypadku próby blokowania lub zwalniania już zablokowanego przez inny wątek mutexu, czy też zwolnienia niezablokowanego mutexu.
  - **PTHREAD_MUTEX_RECURSIVE** - rekursywny, pozwala na wielokrotne blokowania (zliczanie w obrębie jednego wątku), które wymagają do zwolnienia taką samą liczbę odblokowań. Nie występuje tutaj zakleszczenie. Błędy są zwracane w przypadku próby odblokowania mutexu zablokowanego przez inny wątek lub niezablokowanego mutexu.

Dla wątków z polityką **SCHED_FIFO** lub **SCHED_RR** znaczenie mają również:

- zmiana protokołu kolejności żądań blokad - pozwala na uwzględnienie priorytetów wątków przy ustalaniu kolejności wątków blokujących mutexy:

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
                                  int *protocol);
```

  gdzie protokół może być równy:

  - **PTHREAD_PRIO_NONE** - domyślny, bez uwzględniania;
  - **PTHREAD_PRIO_INHERIT** - wątek blokujący inne wątki o większym priorytecie uzyskuje chwilowo większy priorytet (maksimum z wątków blokujących), by wykonać zablokowaną część szybciej;
  - **PTHREAD_PRIO_PROTECT** - jak wyżej, ale priorytet nie zależy od priorytetów wątków lecz od priorytetu mutexu opisanego poniżej.

- zmiana priorytetu mutexu:

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t
                                     *attr, int *prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *mut,
                                 int prioceiling);
int pthread_mutex_getprioceiling(const pthread_mutex_t *mut,
                                 int *prioceiling);
```

Typy mutexów (slajd 1) TypyMutexow.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_t id[2];
pthread_attr_t attr;
pthread_mutex_t mut;
pthread_mutexattr_t mutattr;
int x;
int typmut[2]={PTHREAD_MUTEX_RECURSIVE,PTHREAD_MUTEX_ERRORCHECK};
char typmutopis[2][32]={"PTHREAD_MUTEX_RECURSIVE","
    PTHREAD_MUTEX_ERRORCHECK"};

void funkcja(void){
    int zwrot;
    puts("Wywolano funkcje");
    zwrot=pthread_mutex_lock(&mut);
    if(zwrot) perror("Blad blokowania (funkcja)");
    else{
        puts("Pomyslnie zablokowano (funkcja)");
        x++;
        zwrot=pthread_mutex_unlock(&mut);
        if(zwrot) perror("Blad odblokowania (funkcja)");
        else puts("Pomyslnie odblokowano (funkcja)");
    }
}
```

Typy mutexów (slajd 2)

```
void* watek1(void* _arg){
    int zwrot;
    puts("Uruchomiono watek 1");
    zwrot=pthread_mutex_lock(&mut);
    if(zwrot) perror("Blad blokowania (watek)");
    else{
        puts("Pomyslnie zablokowano (watek)");
        funkcja();
        zwrot=pthread_mutex_unlock(&mut);
        if(zwrot) perror("Blad odblokowania (watek)");
        else puts("Pomyslnie odblokowano (watek)");
    }
    return 0;
}

void* watek2(void* _arg){
    if(!pthread_mutex_lock(&mut)){
        x*=3;
        pthread_mutex_unlock(&mut);
    }
    return 0;
}

void* (*wskfwatki[2])(void*)={watek1,watek2};
```

Typy mutexów (slajd 3)

```
int main(){
    int i,j,zwrot;
    if(pthread_mutexattr_init(&mutattr)){
        perror("Blad inicjacji mutexu"); exit(1); }
    for(i=0;i<2;i++){
        x=3;
        printf("Proba %s\n",typmutopis[i]);
        pthread_mutexattr_settype(&mutattr,typmut[i]);
        if(pthread_mutex_init(&mut,&mutattr)){
            perror("Blad inicjacji mutexu"); exit(2); }
        for(j=0;j<2;j++){
            zwrot=pthread_create(&id[j], NULL, wskfwatki[j], NULL);
            if(zwrot) printf("Nie utworzono watku %d\n",j);
        }
        for(j=0;j<2;j++){
            zwrot = pthread_join(id[j], NULL);
            if(zwrot) perror("Blad watku");
            else printf("Watek %d zakonczony prawidlowo\n",j);
        }
        printf("x=%d\n",x);
        pthread_mutex_destroy(&mut);
    }
    pthread_mutexattr_destroy(&mutattr);// zwalniamy atrybuty mutexu
    return 0;
}
```

Blokowanie ograniczone czasowo (slajd 1) timedlock.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
pthread_mutex_t mut;
int x=1;
struct timespec czas;
void* watek1(void* _arg){
    puts("1");
    czas.tv_sec=time(0); czas.tv_nsec=400;
    if(pthread_mutex_timedlock(&mut,&czas)) puts("2");
    else{ puts("3"); x*=2;
        if(pthread_mutex_unlock(&mut)) puts("4");
        else puts("5");}
    return 0;
}
void* watek2(void* _arg){
    puts("A");
    czas.tv_sec=time(0); czas.tv_nsec=400;
    if(pthread_mutex_timedlock(&mut,&czas)) puts("B");
    else{ puts("C"); x*=3;
        if(pthread_mutex_unlock(&mut)) puts("D");
        else puts("E");}
    return 0;
}
```

Blokowanie ograniczone czasowo (slajd 2)

```
pthread_t id[2];
void* (*wskfwatki[2])(void*)={watek1,watek2};
int main(){
    int j;
    if(pthread_mutex_init(&mut,NULL)){
        perror("Blad inicjacji mutexu"); exit(2); }
    for(j=0;j<2;j++){
        if(pthread_create(&id[j], NULL, wskfwatki[j], NULL))
            printf("Nie utworzono watku %d\n",j);
    }
    for(j=0;j<2;j++){
        if(pthread_join(id[j],NULL)) printf("Blad watku %d\n",j);
        else printf("Watek %d zakonczony prawidlowo\n",j);
    }
    printf("x=%d\n",x);
    pthread_mutex_destroy(&mut);
    return 0;
}
```

Blokady zapisu lub odczytu

Mutexy zawsze blokowały dostęp do zmiennej niezależnie od tego czy chcieliśmy ją odczytywać czy zapisywać. Jeśli chcielibyśmy to rozróżniać, to możemy użyć blokad zapisu lub odczytu. Podobnie jak wcześniej mamy typ dla blokady **pthread_rwlock_t** oraz jej atrybutów **pthread_rwlockrattr_t**. Do inicjacji i odpowiednio zwolnienia zasobów blokady służą funkcje:

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

przy czym podanie wskaźnika do atrybutów jako pusty oznacza domyślne atrybuty. Można również zainicjować blokadę w sposób statyczny z domyślnymi parametrami za pomocą przypisania do niej stałej **PTHREAD_RWLOCK_INITIALIZER**. Podobnie do inicjacji i zwalniania atrybutów bariery mamy:

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Jedynym dostępnym atrybutem jest współdzielenie blokady z innymi procesami:

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared)
int pthread_rwlockattr_getpshared(pthread_barrierattr_t *attr,
                                  int *pshared);
```

z trybami **PTHREAD_PROCESS_PRIVATE** (domyślny) i **PTHREAD_PROCESS_SHARED**.

Do założenia blokady do odczytu mamy trzy funkcje różniące się sposobem oczekiwania:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);// w nieskończ.
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);// jedna próba
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
                    const struct timespec *abs_timeout);// czasowa
```

Podobnie do założenia blokady do zapisu:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);// w nieskończ.
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);// jedna próba
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
                    const struct timespec *abs_timeout);// czasowa
```

Do zwalniania powyższych blokad służy:

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

W blokadach ograniczonych czasem, czas jest podawany jako bezwzględny. Jeżeli nie uda się uzyskać blokady przed upłynięciem podanego czasu, to zwracany jest błąd **ETIMEDOUT**. Przy blokadzie do odczytu dostęp może wiele wątków na raz, w przypadku blokady do zapisu tylko jeden.

Blokady (slajd 1)

Blokady.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#ifdef _WIN32
    #include <windows.h>
#else
    #include <time.h>
#endif
#define N 5 /* liczba wątków odczytujących */
#define K 2 /* liczba wątków zapisujących */
pthread_rwlock_t blokada=PTHREAD_RWLOCK_INITIALIZER;
int wartosc; // obiekt chroniony blokadą
void ms_sleep(unsigned czas){
#ifdef _WIN32
    Sleep(czas);
#else
    struct timespec req;
    req.tv_sec  = (czas / 1000);
    req.tv_nsec = (czas % 1000 * 1000000);
    nanosleep(&req, NULL);
#endif
}
```

Blokady (slajd 2)

```
void* pisarz(void* numer){
    for(int i=0;i<2;i++){
        printf("%c", (int)numer+'0');
        if(!pthread_rwlock_wrlock(&blokada)){
            printf("%c",(int)numer+'3'); ms_sleep(100);
            printf("%c", (int)numer+'6');
            if(pthread_rwlock_unlock(&blokada))
                printf("blad zw. blokady z %d\n",(int) numer);
        } else printf("blad blokady z %d\n",(int) numer);
        ms_sleep(400);
    } return 0;
}
void* czytelnik(void* numer){
    for(int i=0;i<2;i++){
        printf("%c", (int)numer+'a');
        if(!pthread_rwlock_rdlock(&blokada)){
            printf("%c", (int)numer+'A'); ms_sleep(20);
            printf("%c", (int)numer+'A');
            if(pthread_rwlock_unlock(&blokada))
                printf("blad zw. blokady o %d\n",(int) numer);
        } else printf("blad blokady o %d\n",(int) numer);
        ms_sleep(20);
    } return 0;
}
```

Blokady (slajd 3)

```
int main() {
    pthread_t odczyt[N];
    pthread_t zapis[K];
    int i;
    pthread_rwlock_init(&blokada, NULL);
    for(i=0;i<K;i++)
        if(pthread_create(&zapis[i],0,pisarz,(void*)i))
            printf("Blad tworzenia o %d\n",i);
    for(i=0;i<N;i++)
        if(pthread_create(&odczyt[i],0,czytelnik,(void*)i))
            printf("Blad tworzenia o %d\n",i);
    for(i=0;i<N;i++)
        if(pthread_join(odczyt[i],0))
            printf("Blad watku o %d\n",i);
    for(i=0;i<K;i++)
        if(pthread_join(zapis[i],0))
            printf("Blad watku z %d\n",i);
    pthread_rwlock_destroy(&blokada);
    return 0;
}
```

Zmienne warunkowe

Zmienne warunkowe są jednym ze sposobów synchronizacji między wątkami. Mają za zadanie wysłanie sygnałów (z pewnego wątku) do wątków oczekujących na dany sygnał. Jest ona zawsze używana z mutexem (nierekursywnym).
Ze zmiennymi warunkowymi są związane typy **pthread_cond_t** oraz **pthread_condattr_t** do obsługi atrybutów. Zmienna warunkowa wymaga inicjacji, a gdy staje się zbędna, to należy ją zwolnić:

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Wskaźnik na atrybuty może być zerowy, wtedy są one domyślne. Możliwa jest statyczna inicjacja przy pomocy stałej o domyślnych atrybutach **PTHREAD_COND_INITIALIZER**. Mamy dwa możliwe atrybuty:

- współdzielenie zmiennej warunkowej między procesami:

```
int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);
int pthread_condattr_getpshared(pthread_condattr_t *attr,
                                int *pshared);
```

gdzie **pshared** może być **PTHREAD_PROCESS_PRIVATE** - bez współdz., domyślny lub **PTHREAD_PROCESS_SHARED** - współdz.

- ustawienie identyfikatora zegara używanego do odmierzania czasu:

```
int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
int pthread_condattr_getclock(pthread_condattr_t *attr,
                              clockid_t *clock_id);
```

  domyślnie zegar jest zegarem systemowym, na Windows nie ma innych, na linux są **CLOCK_REALTIME** (domyślny), **CLOCK_MONOTONIC** i inne, przy czym w Linux jeśli uwzględniamy atrybuty, to należy jakikolwiek ustawić, by działo poprawnie oczekiwanie ograniczone czasowo.

Wątek, który zmienia warunek może to sygnalizować (jednemu, nieokreślonemu) lub rozgłaszać (wszystkim) innym wątkom poprzez funkcje:

```
int pthread_cond_signal(pthread_cond_t *cond);//sygnalizacja
int pthread_cond_broadcast(pthread_cond_t *cond);//rozgłaszanie
```

Natomiast oczekujące wątki do czekania na sygnał wykorzystują:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t* mut);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t* mut,
                           const struct timespec *abstime)
```

gdzie **abstime** oznacza czas bezwzględny.
Należy tutaj uważać, by sygnał był wysyłany gdy inny wątek już na niego czeka (w szczególności wątek oczekujący został już uruchomiony).
Przy oczekiwaniu blokada mutexu zostaje zwolniona aż do uzyskania sygnału.

Zmienne warunkowe (slajd 1) Warunkowe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define N 64*1024
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t war=PTHREAD_COND_INITIALIZER;
typedef struct{
    int* wsk;
    int n;
    float med;
}tablica;
void* mediana(void* _arg){
    struct timespec czas; tablica *t=(tablica*)_arg;
    czas.tv_sec=time(0)+4; czas.tv_nsec=400;
    if(pthread_mutex_lock(&mut)) puts("Blad blokowania w mediana");
    else{
        puts("Mediana czeka");
        if(pthread_cond_timedwait(&war,&mut,&czas)) puts("Blad");
        t->med=t->wsk[t->n >>1];
        if((t->n & 1)==0){ t->med+=t->wsk[(t->n >>1)-1]; t->med/=2; }
        pthread_mutex_unlock(&mut);
    }
    return 0;
}
```

Zmienne warunkowe (slajd 2)

```
void* sortuj(void* _arg){
    int pol,ind,wi,wj,value,*tab;
    int stala=(((tablica*)_arg)->n)*(((tablica*)_arg)->n)-640000;
    tab=((tablica*)_arg)->wsk;
    pol=((tablica*)_arg)->n >>1;
    if(pthread_mutex_lock(&mut)) perror("Blad blokowania");
    else{
        for(wi=((tablica*)_arg)->n -1;wi>0;wi--){
            ind=wi;
            for(wj=wi-1;wj>=0;wj--)
                if(tab[wj]>tab[ind]) ind=wj;
            value=tab[ind]; tab[ind]=tab[wi]; tab[wi]=value;
            if(wi==pol){
                puts("Wysylamy sygnal");
                pthread_cond_signal(&war);
            }
            if(stala>=wi*wi){// pozwalamy innym wątkom
                sched_yield();// na włączenie się
                stala=wi*wi -640000;
            }
        }
        pthread_mutex_unlock(&mut);
    }
    return 0;
}
```

Zmienne warunkowe (slajd 3)

```
pthread_t id[2];
int main(){
    int j;
    tablica tab;
    tab.wsk=malloc(N*sizeof(int)); tab.n=N;
    if(!tab.wsk){
        puts("Nie udalo sie utworzyc tablicy"); exit(1); }
    srand(time(0));
    for(j=0;j<N;j++)
        tab.wsk[j]=rand();
    if(pthread_create(&id[1], NULL, mediana, &tab)){
        puts("Nie utworzono watku mediana"); exit(5); }
    if(pthread_create(&id[0], NULL, sortuj, &tab)){
        puts("Nie utworzono watku sortujacego"); exit(4); }
    if(pthread_join(id[0],NULL)) puts("Blad wyjscia watku sortuj");
    else puts("Watek sortuj zakonczony prawidlowo");
    if(pthread_join(id[1],NULL)) puts("Blad wyjscia watku mediana");
    else puts("Watek mediana zakonczony prawidlowo");
    printf("%f\n",tab.med);
    pthread_cond_destroy(&war);
    pthread_mutex_destroy(&mut);
    return 0;
}
```

Bariery

Kolejną metodą synchronizacji wątków jest bariera. Polega ona na tym, że wątki są wstrzymywane aż do momentu dojścia ostatniego z nich do tej bariery. Podobnie jak wcześniej mamy typ dla bariery **pthread_barrier_t**oraz jej atrybutów **pthread_barrierattr_t**. Do inicjacji bariery służy funkcja

```
int pthread_barrier_init(pthread_barrier_t *barrier,
    pthread_barrierattr_t* attr, unsigned int count);
```

count mówi ile wątków jest powiązanych z barierą. **attr** może być wskaźnikiem zerowym - domyślne atrybuty. Do zwolnienia pamięci związanych z barierą służy:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Podobnie do inicjacji i zwalniania atrybutów bariery mamy:

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Jedynym dostępnym atrybutem jest współdzielenie bariery z innymi procesami:

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                   int pshared);
int pthread_barrierattr_getpshared(pthread_barrierattr_t *attr,
                                   int *pshared);
```

z trybami **PTHREAD_PROCESS_PRIVATE** (domyślny) i **PTHREAD_PROCESS_SHARED**.

Aby wstrzymać wątek do momentu dojścia wszystkich wątków do bariery należy użyć funkcji:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Funkcja ta tylko dla jednego z wątków zwraca wartość różną od zera i jest to **PTHREAD_BARRIER_SERIAL_THREAD**. Pozostałe zwracane wartości oznaczają błąd. Po zwrocie tej wartości bariera znów nadaje się do użycia (z ostatnio inicjowaną liczbą wątków w niej uczestniczących).

Bariery (slajd 1) Bariery.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <x86intrin.h>
#define N 8*1024*1024
#define W 4
pthread_barrier_t bar[W/2];
pthread_t id[W];
typedef struct{
    int* wsk;
    int* wsk2;
    int* pom;
}tablica;
tablica t;
void sortowanie_scalanie(int tab[],int pom[],int i_p, int i_k){
    int i_s=(i_p + i_k)>>1,i1=i_p,i2=i_s,;
    if(i_s-i_p>1) sortowanie_scalanie(tab,pom,i_p,i_s);
    if(i_k-i_s>1) sortowanie_scalanie(tab,pom,i_s,i_k);
    for(int i=i_p;i<i_k;i++)
        pom[i]=((i1==i_s)||((i2<i_k)&&(tab[i1]>tab[i2])))?tab[i2++]:
    tab[i1++];
    memcpy(tab+i_p,pom+i_p,(i_k-i_p)*sizeof(int));
}
```

Bariery (slajd 2)

```
void* sortuj(void* _arg){
    int ktory=(int)_arg, bity=1,i_s,i1,i2,nrbar;
    int i_p=(ktory*N)/W, i_k=((ktory+1)*N)/W, *tab=t.wsk,*pom=t.pom;
    sortowanie_scalanie(tab,t.pom,i_p,i_k);
    while(bity<W){
        nrbar=(ktory/(bity*2))*bity;
        switch(pthread_barrier_wait(&bar[nrbar])){
            case 0:
            case PTHREAD_BARRIER_SERIAL_THREAD:
                break;
            default:
                puts("b");
        }
        if((ktory&bity)==0){
            i1=i_p; i_s=i_k; i2=i_s;
            i_k=((2*bity+ktory)*N)/W;
            for(int i=i_p;i<i_k;i++)
                pom[i]=((i1==i_s)||((i2<i_k)&&(tab[i1]>tab[i2])))?tab[
    i2++]:tab[i1++];
            memcpy(tab+i_p,pom+i_p,(i_k-i_p)*sizeof(int));
            bity<<=1;
        }else break;
    }
    return 0;
}
```

Bariery (slajd 3)

```
int main(){
    int j; unsigned long long l1,l2;
    t.wsk=malloc(N*sizeof(int)); t.wsk2=malloc(N*sizeof(int));
    t.pom=malloc(N*sizeof(int));
    for(j=0;j<W/2;j++)
        if(pthread_barrier_init(&(bar[j]),0,2))
            printf("Nie utworzono bariery %d\n",j);
    if(!t.wsk || !t.wsk2 || !t.pom){ puts("Za malo pamieci");
        free(t.wsk); free(t.wsk2); free(t.pom); exit(1); }
    srand(time(0));
    for(j=0;j<N;j++){ t.wsk[j]=rand(); t.wsk2[j]=t.wsk[j]; }
l1=__rdtscp(&j);
    for(j=0;j<W;j++)
        if(pthread_create(&id[j], NULL, sortuj, (void*)j)){
            printf("Nie utworzono watku %d",j); exit(4); }
    for(j=0;j<W;j++)
        if(pthread_join(id[j],NULL)) printf("Blad watku %d",j);
l2=__rdtscp(&j); printf("Wielowatkowe sortowanie %llu\n",l2-l1);
l1=__rdtscp(&j);
    sortowanie_scalanie(t.wsk2,t.pom,0,N);
l2=__rdtscp(&j); printf("Jednowatkowe sortowanie %llu\n",l2-l1);
    free(t.pom); free(t.wsk); free(t.wsk2);
    for(j=0;j<W/2;j++) pthread_barrier_destroy(&bar[j]);
    return 0;
}
```

Semafory

Semafory są podobne do mutexów, przy czym tutaj możemy zadecydować ile wątków jednocześnie może mieć dostęp do chronionego obszaru. Ogólnie rzecz biorąc semafor utożsamiany jest z liczbą nieujemną o pewnej wartości początkowej. Każdy dostęp do niego zmniejsza tą wartość o 1, a zwolnienie dostępu zwiększa o 1, przy czym dostęp jest możliwy gdy wartość ta nie jest zerem. Aby ich użyć należy dołączyć bibliotekę **semaphore.h**. Z semaforami związany jest typ **sem_t**. Zmienna tego typu najpierw musi być zainicjowana przy pomocy:

```
int sem_init(sem_t *semafor,int pshared,int value);
```

przy czym **pshared** może przyjmować wartości **PTHREAD_PROCESS_PRIVATE** (domyślny, bez współdzielenia) lub **PTHREAD_PROCESS_SHARED** (współdzielenie między procesami), a **value** oznacza liczbę wątków mających jednoczesny dostęp do obszaru (musi być większe niż zero! i nie przekraczać **SEM_VALUE_MAX**). Gdy semafor już przestaje nam potrzebny należy go zwolnić za pomocą:

```
int sem_destroy(sem_t *semafor);
```

Powyższe funkcje zwracają 0 w przypadku sukcesu, w przypadku błędu zwracana jest -1.

Blokowanie semafora podobne jak dla mutexów może odbywać się na trzy sposoby:

```
int sem_wait(sem_t *semafor);// czeka do skutku
int sem_trywait(sem_t *semafor);// nie czeka
int sem_timedwait(sem_t *semafor, const struct timespec *abstime);
    // czeka co najwyżej podany czas
```

Do odblokowywania (właściwie zwiększania licznika semafora) służy funkcja:

```
int sem_post(sem_t *semafor);
```

Możemy również sprawdzić aktualny licznik semafora za pomocą

```
int sem_getvalue(sem_t *semafor,int *value);
```

Powyższe funkcje zwracają 0 w przypadku sukcesu (dla funkcji blokujących zmniejszany jest wówczas licznik semafora, a dla odblokowujących zwiększany o 1), w przypadku błędu zwracana jest -1 (licznik semafora nie ulega wtedy zmianie), kod błędu można uzyskać z **errno** (tak jak zwykle **perror** może nam wyświetlić szczegóły). Błędy, które mogą się tu pojawić, to **EINTR** - przerwane wywołanie przez obsługę sygnałów*, **EINVAL** - błędny semafor (lub czas dla blokowania z limitem czasu), **EAGAIN** - dla wersji blokowania bez czekania, oznacza zablokowany semafor, **ETIMEDOUT** - upłynął podany czas oczekiwania na blokowanie, **EOVERFLOW** - odblokowanie osiągnęło maksymalną dostępną wartość (**SEM_VALUE_MAX**).

Semafory.c

Podsumowanie wątków

- Blokowanie innych wątków:
  - mutex (pojedynczy dostęp);
  - semafor (dostęp kilku naraz - sami decydujemy ilu);
  - blokady zapisu/odczytu (dostęp wielu naraz przy odczycie, pojedynczy dostęp przy zapisie).
- Informowanie innych wątków o zakończeniu pewnych działań - zmienne warunkowe (+mutex).
- Oczekiwanie na zakończenie działań przez kilka wątków - bariera.

Typy atomowe (C11)

W przypadku programów wielowątkowych mamy pewne problemy z dostępem do tej samej zmiennej. W prosty sposób możemy sobie zagwarantować, że w zmiennej tej będą zapyswane wszystkie wyniki.

Inkrementacja w dwóch wątkach dla zmiennej

nieatomowej:

wątek 1	wątek 2	zmienna
odczyt		0
modyfikacja	odczyt	0
zapis	modyfikacja	1
	zapis	1
		1
		1

atomowej:

wątek 1	wątek 2	zmienna
odczyt		0
modyfikacja		0
zapis		1
	odczyt	1
	modyfikacja	1
	zapis	2

Żeby uczynić zmienną atomową należy użyć składni: atomic.c

```
_Atomic typ nazwa_zmiennej;
// lub
_Atomic(typ) nazwa_zmiennej;
// można też skorzystać z gotowych typów z biblioteki stdatomic.h, np.
atomic_int nazwa_zmiennej;
```

Sygnały

Program może otrzymywać pewne sygnały o występujących zdarzeniach (np. zamykanie programu). Możemy te sygnały odbierać za pomocą własnych funkcji. Potrzebna do tego nam będzie biblioteka **signal.h**. Mamy w niej dwie funkcje:

```
void (*signal(int sig, void (*func)(int)))(int);
```

która wiąże z sygnałem **sig** funkcję, która go obsługuje. Zamiast własnej funkcji można tutaj podać pewne automatyczne funkcje - domyślną **SIG_DFL** lub ignorującą **SIG_IGN**. Sygnał **sig** może być również jednym z domyślnych sygnałów

sygnał	znaczenie
SIGABRT	Nienormalne zakończenie działania programu (np. funkca **abort**)
SIGFPE	Niedozwolona operacja arytmetyczna
SIGILL	Niedozwolona instrukcja procesora
SIGINT	Przerwanie procesu
SIGSEGV	Naruszenie ochrony pamięci
SIGTERM	Do programu zostało wysłane żadanie jego zakończenia

Funkcja **signal** zwraca wskaźnik na poprzednią funkcję obsługującą dany sygnał, a jeśli się nie powiedzie to zwraca **SIG_ERR**.

Drugą funkcją jest

```
int raise (int sig);
```

która wysyła sygnał **sig** do naszego programu. Zwraca ona 0 w przypadku powodzenia, lub inną wartość w przypadku błędu.
W bibliotece **signal.h** mamy jeszcze typ **sig_atomic_t**, który może być przydatny do współdzielenia zmiennej całkowitej (atomowej).

Przykład

signals.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#ifdef __linux__
#include <time.h>
void ms_sleep(unsigned czas){
    struct timespec req;
    req.tv_sec = (czas / 1000);
    req.tv_nsec = (czas % 1000 * 1000000);
    nanosleep(&req, NULL);
}
#elif _WIN32
#include <windows.h>
void ms_sleep(unsigned czas){
    Sleep(czas);
}
#endif
void sighandler(int signum) {
    printf("Zlapano sygnal %d, konczenie...\n", signum);
    exit(1);
}
int main () {
    signal(SIGINT, sighandler);
    printf("Nacisnij ctrl+c by wyjsc\n");
    while(1) {
        printf("Sleep(1s)...\n");
        ms_sleep(1000);
    }
    return(0);
}
```