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.

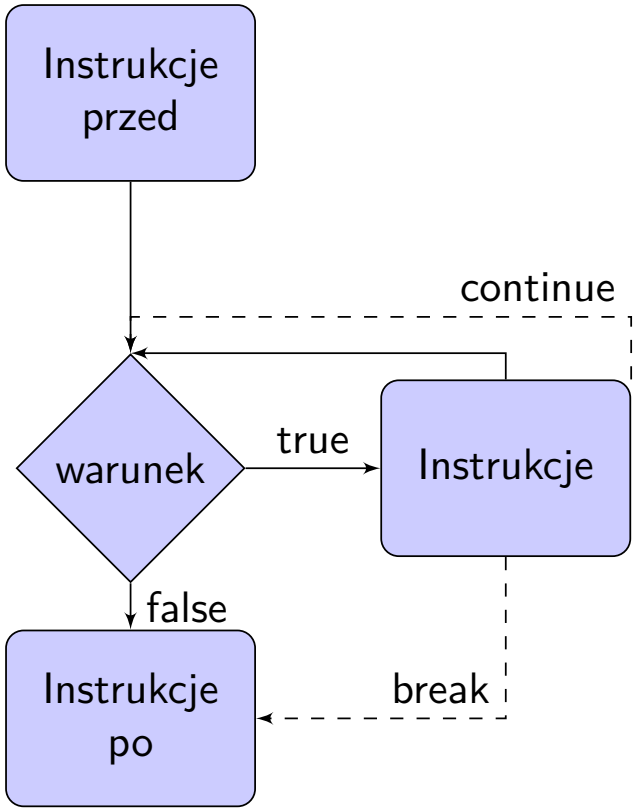
Każdą z powyższych pętli można symulować pozostałymi.

Instrukcja **while**

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

gdy tylko jedna instrukcja w pętli:

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



Przykład - Algorytm Euklidesa

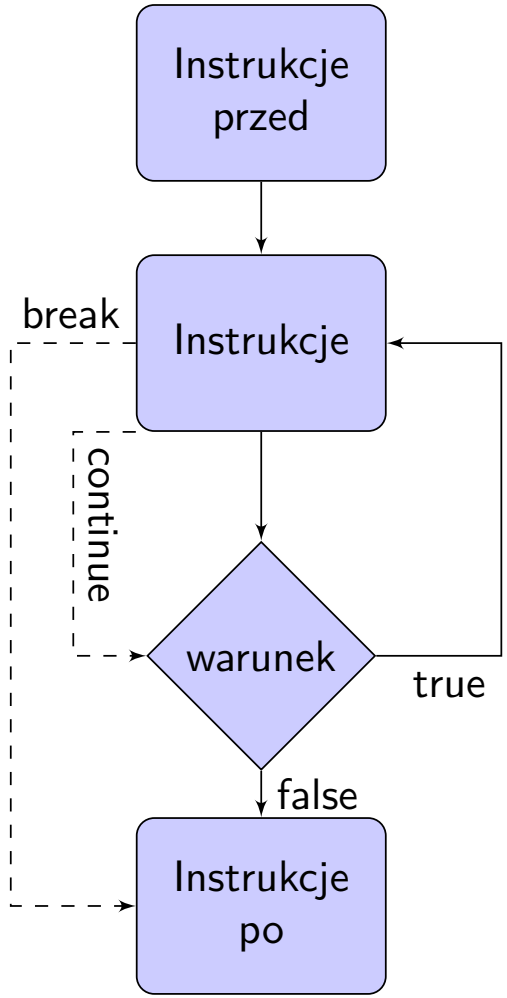
```
unsigned a,b;
printf("Podaj dwie liczby całkowite nieujemne\n");
scanf(" %u %u",&a,&b);
while(b!=0){
    unsigned r=a%b; //Obliczanie reszty
    a=b;
    b=r;
}
printf("NWD= %u\n",a);
```

Instrukcja **do...while**

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

gdy tylko jedna instrukcja w pętli:

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



Przykład - menu programu

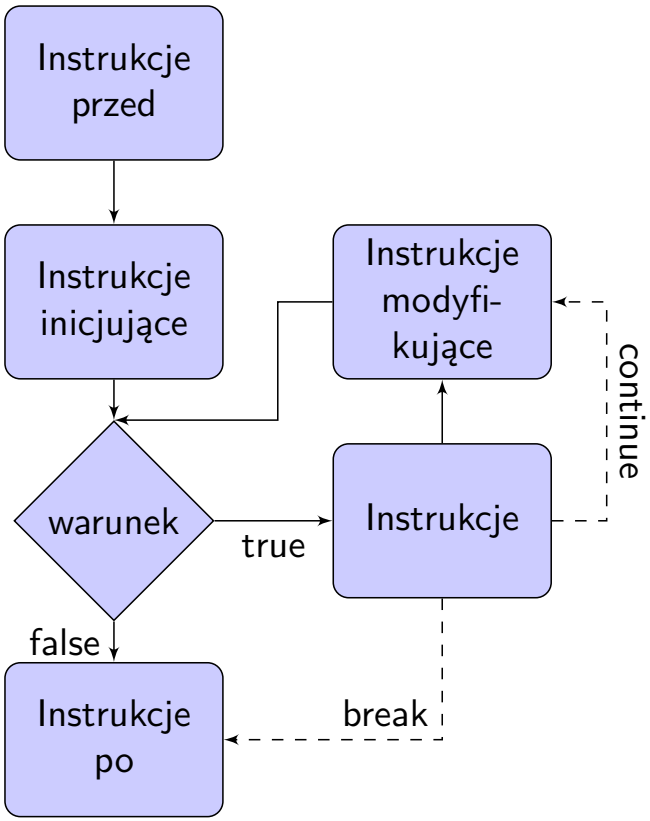
```
int main(){
    char znak;
    do{
        /*
            instrukcje w menu
        */
        printf("Zakonczyc program? (T/N): ");
        scanf(" %c",&znak);
    }while((znak!='t') && (znak!='T'));
    return 0;
}
printf("Koniec programu.\n");
```

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);
}
```