Język ANSI C

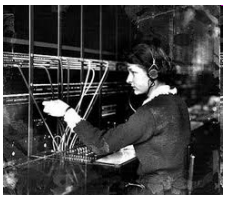
część 3

Podstawy języka C

Jarosław Gramacki
Instytut Informatyki i Elektroniki

Operatory



Operatory

- operator - w programowaniu konstrukcja językowa jedno-, bądź wieloargumentowa zwracającą wartość
- zagadnienie operatorów w języku C jest wbrew pozorom dosyć trudne
  - jest ich dużo i mają wielopoziomową hierarchię (priorytety)
  - mają czasami słabo czytelną postać
    - » `&&` oraz `||` zamiast np. `and` oraz `or`
  - niektóre operatory całkowicie zmienia swoje znaczenie w zależności od kontekstu !
    - » `&, *, =, ==, !, !=, &, &&, |, ||`

Operatory

Operator	Łączność
`() [] -> .`	L
`! ~ ++ -- + - * & (typ) sizeof`	P
`* / %`	L
`+ -`	L
`<< >>`	L
`< <= > =>`	L
`== !=`	L
`&`	L
`^`	L
`|`	L
`&&`	L
`||`	L
`?:`	P
`= += -= *= /= %= ^= |= <<= >>=`	P
`,`	L

- "ciekawsze" operatory zaznaczono strzałkami
  - » dlaczego nazwano je "ciekawsze"?

Operatory - przykłady

Dalej pokażemy kilka wybranych "anomalii" dotyczących operatorów

() operator rzutowania

```
float fNum;
int iNum1 = 6, iNum2 = 4;
fNumber = iNum1 / iNum2;                // 1.0
fNumber = (float)iNum1 / (float)iNum2;  // 1.5
MyFun((long)iNum1);                     // MyFyn wymaga arg. long
```

* dostęp pośredni za pomocą wskaźnika, deklaracja wskaźnika

```
int* piNum;                  // w deklaracji
iNum1 = *piNum;     // w wyrażeniu
```

& pobranie adresu obiektu

```
(long)iNum2 = &iNum1;
```

++ przedrostkowe (ang. prefix)

```
int iTab[10], w = 3;
iTab[ w++ ] = 7;    iTab[ ++w ] = 7;
```

\+ w wyniku otrzymujemy wartość argumentu, ze zmienionym znakiem. wprowadzono dla symetrii z jednoargumentowym operatorem "–"

~ dopełnienie jedynkowe (operator bitowy, zamiana każdego bitu 1 na 0 i odwrotnie + dopełnienie jedynkami z lewej strony do wielkości zależnej od platformy). Argument musi być typu całkowitego.
np.:
`~0` wynik: 16 lub 32 jedynki (zależnie od platformy)
`~0xFF` wynik: `FF00` (dla 16 bitowego środowiska), `FFFFFF00` (dla 32 bitowego środowiska)

Operatory - przykłady

?: `z = (a>b) ? a : b;` //to samo co: `if (a>b) z=a; else z=b;`
nawiasy zbędne, gdyż priorytet operatora `?:` jest bardzo niski, ale lepiej pisać `(a>b)`

```
printf("Masz %d cz%ss", n, n == 1 ? "ć" : "ci");
```

<< uwaga na przesunięcia liczb ze znakiem! Przykłady:

przesuwanie w lewo (mnożenie przez 2):

```
int s, ws;                          // ze znakiem, int 16-bitowy
unsigned int u, wu;                 // bez znaku
s = -12;     ws = s << 2;           // -48 OK.
s =  16385; ws = s << 1;            // BLAD -32766 zamiast +32770
01000000 00000001 --> 10000000 00000010 =
s = (2^14 + 1)         ws = -2^15 + 2 = -32766


s = -32768; ws = s << 1; // BLAD 0 zamiast +32768
s = (-2^15)
```

przesuwanie w prawo (dzielenie przez 2):
przy przesuwaniu w prawo dla liczb bez znaku, najbardziej znaczące bity (zwolnione bity) są zawsze uzupełniane zerami
przy przesuwaniu w prawo dla liczb ze znakiem, najbardziej znaczące bity (zwolnione bity) są na pewnych maszynach powielane bitami znaku a na innych wypełniane zerami

```
u = 45;      wu = u >> 2;           // 11 (część całk. Z 45/4)
s = -22;     ws = s >> 2;           // -6
    -22 (00000000 00101010) -->      -6 (11111111 11111010)
```

Operatory - przykłady

= Nawet z operatorem przypisania możliwe błędy:

```
double x;
int j;
x = j = 1.2;   // cel: przypisz x = 1.2,  j = 1 (część całkowita z 1.2)
```

W praktyce wyrażenie zostanie opracowane tak:

```
x = (j = 1.2); // uzyskamy x = 1, bo łączność z Prawej do Lewej
```

Operator == jest często mylony z =, przy czym powstałe wyrażenie jest całkowicie poprawne np:

```
if (a=1)  // przypisanie, wartością wyrażenie jest 1 czyli TRUE
if (a==1) // porównanie, TRUE lub FALSE w zależności od wartości a
```

, Uwaga !!! Język C nie określa kolejności obliczania wartości operatora – źródło wielu błędów np. (tzw. efekt uboczny):

```
x = f() + g();
printf(„%d %d\n”, ++n, power(2,n));
Tab[ i ] = i++;
```

&& Wykonywana jest najmniejsza liczba operacji koniecznych do obliczenia wartości całego wyrażenia logicznego

```
x && y++
```

drugi operand, `y++`, jest wykonywany (obliczany) tylko gdy `x` jest prawdą (nie zero). Dlatego `y` NIE zostanie zwiększone gdy `x` jest fałszem (0)

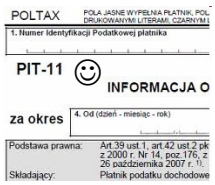
! `if( ! valid )` zamiast `if( valid == 0 )`. Często stosowana konstrukcja

& `if ((num & MASK) == 0)` Wszystkie nawiasy są tu niezbędne

Deklaracja a definicja

- **deklaracja** informuje o właściwościach (typie) zmiennej
  - odnosi się do miejsca, w którym okresla się naturę zmiennej, lecz nie przydziela jej pamieci
- **definicja** dodatkowo powoduje rezerwacje pamięci
  - odnosi się do miejsca, w którym zmienna jest faktycznie tworzona
- przykład deklaracji i definicji zmiennych zewnętrznych:

```
int n;                  // definicja
int tab[100];           // definicja

extern int n;           // deklaracja
extern int tab[100];    // deklaracja
// extern - pamięć będzie zarezerwowana gdzie indziej
```


POLTAX
POLA JASNE WYPEŁNIA PŁATNIK, POL. DRUKOWANYMI LITERAMI, CZARNYM I
1. Numer identyfikacji Podatkowej płatnika
PIT-11
INFORMACJA O
za okres
4. Od (dzień - miesiąc - rok)
Podstawa prawna:
Art.39 ust.1, art.42 ust.2 pk z 2000 r. Nr 14, poz.176, z 26 października 2007 r.).
Składający:
Płatnik podatku dochodowe

Zmienne statyczne

- deklaracja poprzedzane słowem kluczowym **static**
  - opisuje tzw. klasę pamięci
- dotyczy zmiennych zewnętrznych (globalnych) i wewnętrznych
- zasięg statycznych zmiennych zewnętrznych (i funkcji!) ogranicza się do jednego pliku źródłowego
  - np. funkcje w innych plikach nie będą miały dostępu do tych zmiennych
- zmienne statyczne wewnętrzne istnieją miedzy wywołaniami funkcji, nie trącą wartości po zakończeniu działania funkcji (stanowią prywatną stałą pamięć danej funkcji)

```
int funkcja()
{
  static int i = 0; // zmienna wewnętrzna
  return i++;
}

int main (void)
{
  for (int i=1; i<10; i++)
    printf("%d ", funkcja());
  return 0;
}
```


Zmienne statyczne

```
static int zm1;           // zmienna zewnętrzna
static int zm2;           // j.w.

int fun1 (void) {...}

float fun2 (void) {...}
```

- żadna inna funkcja nie będzie miała dostępu do tych zm1, zm2 a ich nazwy nie będą kolidować z takimi samymi nazwami w plikach jednego programu
  - w pewnym sensie ukryto więc zmienne zm1 i zm2

```
static int fun1(int argument);
```

- bez `static` nazwa funkcji widoczna jest dla całego programu
- po dodaniu `static` jej nazwa jest widoczna tylko w pliku zawierającym jej deklarację

Zmienne rejestrowe

- deklaracja poprzedzane słowem kluczowym **register**
- dla zmiennych typu całkowitego wskazówka dla kompilatora aby taka zmienna była w miarę możliwości umieszczana w rejestrze maszyny, bo zależy nam na szybkim dostępie do niej
  - czyli sytuacja dotycząca intensywnie używanej zmiennej
- potencjalnie deklaracja register może przyśpieszyć działanie programu, ale może też być ignorowana przez kompilator

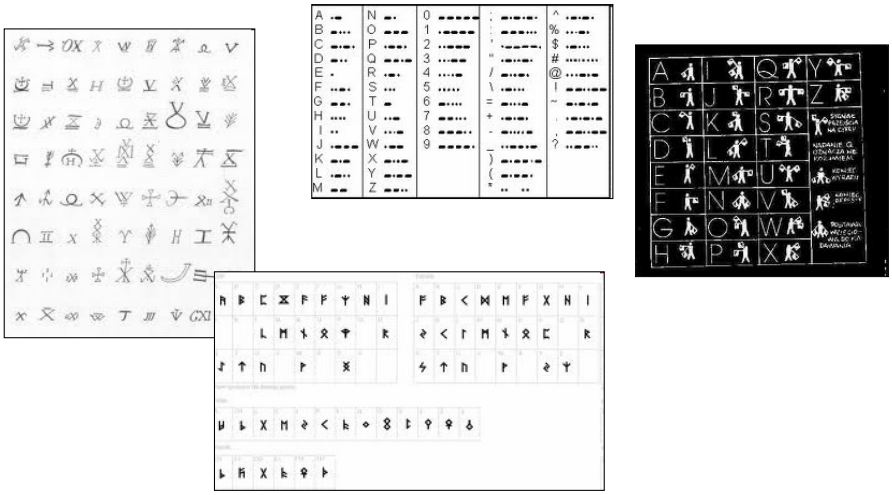
```
register int zm1 = 5;
```


Zmienne const (kwalifikator typu const)

- deklaracja poprzedzane słowem kluczowym **const**
- wskazujemy, że zmienna nie będzie zmieniała wartości
- zmienna const umieszczana jest w pamięci chronionej przed zapisem (i być może wpłynie to na optymalizację programu)

```
const int n = 5;

int strlen (const char[]);  // po co funkcji strlen mozliwość
                            // modyfikacji argumentu ?
```

Kody i znaki specjalne w funkcjach input / output



Kody formatujące (większość)

```
int main(void)
  {
  int a = -5;
  char b = 'a';
  float c = 0.1;
  char TabStr[5]="abcd";  // cztery znaki

  printf("Moje dane: %d, %c, %d, %x, %f", a, b, b, b, c);
  printf("Czy to 0.1 ?: %.80f", c);
  printf("Napis: %s", TabStr);
  return 0;   }

%c       znak
%s       napis (tablica znaków)
%d       liczba całkowita ze znakiem
%f       liczba zmiennoprzecinkowa
%u       liczba całkowita bez znaku
%x       liczba szesnastkowa (5abf)
%X       j.w. (5ABF)
%o       liczba ósemkowa
%e       liczba zmiennoprzecinkowa w notacji wykładniczej
%g       krótszy zapis z %f i %e

przedrostki:
l        długa liczba całk. (%ld, %lu, %lx, %lo)
l        typ double (%lf, %le)
L        typ long double (%Lf, %Le)
h        typ short

schemat:
% [flagi][szerokość][.precyzja][F | N | h | l | L] znak_typu
   -, +
```

Znaki specjalne

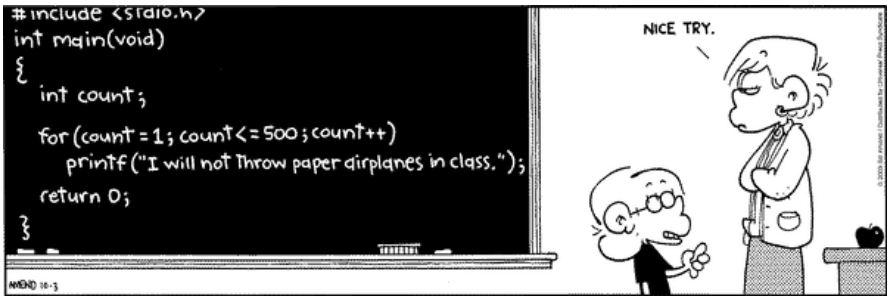
```
int main(void)
  {
  printf("Jakis tekst \n \? i tak\ndalej.\n \x1F");
  }

\n       nowa linia
\t       tabulator poziomy (zwykły)
\b       backspace
\r       powrót karetki
\v       tabulator pionowy
\a       sygnał alarmu
\f       nowa strona

%%       wypisanie znaku % (nie \% - czy potrafisz to wyjaśnić?)

\'       apostrof
\"       cudzysłów
\?       znak ? (jest zastrzeżony w dla operatora ?:)
\\       znak \

\xhh     kod ASCI szesnastkowo (\x i dwie cyfry szesnastkowe)
\ooo     kod ASCI ósemkowo (trzy cyfry ósemkowe)
```

Znaki sterujące

Wprowadzanie znaków zastrzeżonych

Wprowadzanie kodów znaków

Instrukcje sterujące (sterowanie)



- Instrukcje sterujące, pętla `for`
  - **instrukcja** - wyrażenie zakończone średnikiem
    ```
    x = 15;
    n++;
    ```
  - **blok** - instrukcja złożona, sekwencja wyrażeń ograniczona nawiasami {}
    ```
    {
      z = z + x;
      x = 15;
      printf("%d", x);
    }
    ```

```
// forloop
// prints numbers from 0 to 9
#include <stdio.h>
void main(void)
  {
  int count;
  for (count=0; count<10; count++)   // from 0 to 9
    printf("\ncount=%d", count);     // \n - new line
  }
```

brak ;

```
// forloop2
// prints numbers from 0 to 9, keeps running
#include <stdio.h>
void main(void)
   {
   int count, total;
   for (count=0, total=0; count<10; count++, total += 2)
      {
      total = total + count;
      printf("\ncount=%d, total=%d", count, total);
      }
   }
```

Wiele inicjacji w pętli for

Wiele instrukcji w pętli

- Instrukcje sterujące, pętla `for`

```
// asctab
// prints table of ascii characters
#include <stdio.h>
void main(void)
   {
   int i;
   printf("\n");
   for (i = 32; i < 256; i++)        // 0-31 control codes
      printf ("%3d = %c\t", i, i);   // print as number
   }                                 // and as character
```

```
// line.c
// draws a solid line using rectangular graphics character
#include <stdio.h>
void main(void)
   {
   int cols;

   printf("\n");
   for (cols=1; cols<40; cols++)
      printf("%c", '\xDB');
   }
```

jeden znak. Stąd '' a nie ""

- Instrukcje sterujące, pętla `for`

```
// multab
// generates the multiplication table
#include <stdio.h>
void main(void)
   {
   int cols, rows;
   for(rows=1; rows < 13; rows++)      // outer loop
      {
      printf("\n");
      for(cols=1; cols < 13; cols++)   // inner loop
         printf("%3d ", cols*rows );   // adjust column - "%3d "
      }
   }
```

Zagnieżdżone pętle for

```
// fill.c
// fills square area on screen
#include <stdio.h>
void main(void)
   {
   int cols, rows;
   for (rows=1; rows<=22; rows++)     // from row to row
      {
      printf("\n");
      for (cols=1; cols<=40; cols++)  // from column to column
         printf("\xDB");              // or printf("%c", '\xDB');
      }
   }
```

Zagnieżdżone pętle for

- Instrukcje sterujące, ...
  - pozostałe instrukcje (z braku czasu) zostaną jedynie zasygnalizowane
  - instrukcje `if, if-else`
  - instrukcja `switch`
  - pętla `while`
  - pętla `do-while`
  - instrukcje `break` i `continue` w pętlach
  - instrukcja `goto` i etykiety

- Instrukcje sterujące, ...

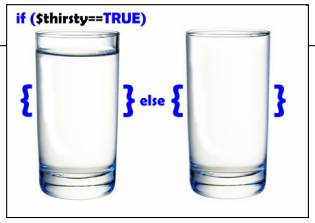
C Reference Card ANSI (fragment)

Flow of Control	
statement terminator	`;`
block delimeters	`{ }`
exit from `switch, while, do, for`	`break`
next iteration of `while, do, for`	`continue`
go to	`goto` label
label	label:
return value from function	`return` expr
Flow Constructions	
`if` statement	`if` (expr) statement `else if` (expr) statement `else` statement
`while` statement	`while` (expr) statement
`for` statement	`for` (expr 1; expr2; expr3) statement
`do` statement	`do` statement `while` (expr);

`switch` statement	`switch` (expr) {
	`case` const1: statement1 `break`;
	`case` const2: statement2 `break`;
	`default`: statement
	}

- Instrukcje sterujące - `if-else`

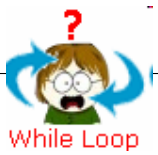
```
if (x > 6 && v < 2)
  {
    n++;
    k--;
  }
else              // else można pominąć
  {
    n--;
    k++;
  }
```



- Instrukcje sterujące - `switch`

```
switch(x)
{
  case 1: printf("1");
  case 2: printf("2");
}

switch(znak)
{
  case 'a': case 'o': case 'e':
  case 'i': case 'u': case 'y':
    printf("Samogloska\n");
    break;
  default:
    printf("Spolgloska\n");
}
```

- Instrukcje sterujące - pętla `while`



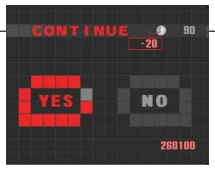

```
int i = 10;

while (i != 0) // while (i)
{
  printf("%d ", i);
  i--;
}
```

- Instrukcje sterujące - pętla `do-while`

```
int i = 10;

do
{
  printf("%d ", i);
  i--;
}
while (i != 0);  // while (i)
```


- Instrukcje sterujące - `break, continue`

```
int i;

for (i = 10; i != 0; i--)
{
  if (i % 2 == 1) continue;          // idź do następnego kroku pętli
  printf("i=%d i%%2=%d ", i, i%2);
}
```

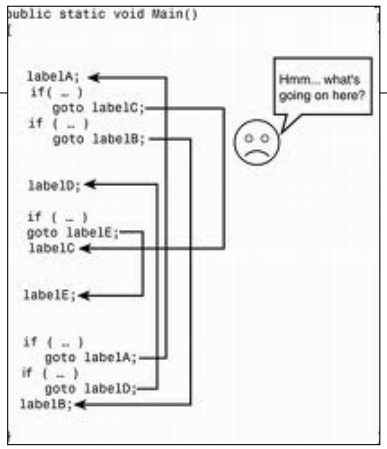



```
int i, j;

for (i = 1; i<=4; i++)
{
  for (j=1; j<=10; j++)
  if (j == 3) break;       // wyskok z najbardziej zagnieżdżonej pętli
    printf("i=%d j=%d ", i, j);
}
```

- Instrukcje sterujące - `goto`

```
for (...)
{
  for (...)
  {
  if (niepowodzenie) goto error;
  ...
  }
}
...
error:
...
```

  - generalnie zawsze możliwa do zastąpienia bardziej „kontrolowalnymi" konstrukcjami
  - czasami jednak użyteczna
    - » np. wyjście z bardzo zagłębionych w sobie pętli

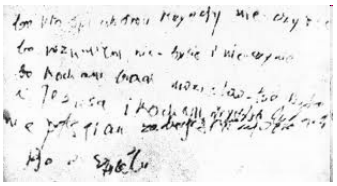


Czytelny kod programu !

```
Wczoraj do północy można było składać zapisy na akcje GPW w transzy dla inwestorów
indywidualnych. Zainteresowanie akcjami warszawskiej giełdy przerosło wszelkie
oczekiwania. Oferta GPW przyciągnęła więcej inwestorów niż PZU czy Tauron. Od 18 do 27
października na akcje GPW zapisało się 323 308 inwestorów.

Pobiliśmy wszystkie możliwe rekordy, akcjonariat obywatelski zatriumfował – powiedział
minister skarbu Aleksander Grad.

Do drobnych graczy trafi 30 proc. akcji z puli 26,8 mln sztuk, co daje 8,036 mln
akcji. Jeden inwestor mógł złożyć zapisy na maksymalnie 100 akcji po cenie maksymalnej
43 zł za sztukę.
```

Wczoraj do północy można było składać zapisy na akcje GPW w transzy dla inwestorów indywidualnych.Zainteresowanie akcjami warszawskiej giełdy przerosło wszelkie oczekiwania.Oferta GPW przyciągnęła więcej inwestorów niż PZU czy Tauron.Od 18 do 27 października na akcje GPW

zapisało się 323 308 inwestorów.Pobiliśmy wszystkie możliwe rekordy,akcjonariat obywatelski zatriumfował - powiedział minister skarbu Aleksander Grad.Do drobnych graczy trafi 30 proc. akcji z puli 26,8 mln sztuk,co daje 8,036 mln akcji.Jeden inwestor mógł złożyć zapisy na maksymalnie 100 akcji po cenie maksymalnej 43 zł za sztukę.



- Formatowanie kodu, notacja węgierska
  - jedna z propozycji zachowania porządku w naszych programach jeśli chodzi o nazewnictwo zmiennych
  - dosyć często stosowana przez programistów (choć są jej zagorzali krytycy)
  - istota zapisu: sposób zapisu zmiennych, polegający na poprzedzaniu właściwej nazwy zmiennej małą literą (literami) określającą rodzaj tej zmiennej

przedrostek	znaczenie
s	string (łańcuch znaków)
sz	string (łańcuch znaków zakończony bajtem zerowym - null'em)
c	char (jeden znak), również const - wartość stała (szczególnie w przypadku użycia wskaźników)
by	byte, unsigned char
n	short
i	int
x, y	int (przy zmiennych określających współrzędne)
cx, cy	int (przy zmiennych określających rozmiar, długość)
l	long
w	word
dw	dword
b	boolean (wartość logiczna: prawda lub fałsz)
f	flaga
fn	funkcja
h	handle (uchwyt)
p	pointer (wskaźnik)

- Formatowanie kodu, notacja węgierska
  - charakterystyczne dla notacji węgierskiej (tworzące jej "węgierskie brzmienie") są również złożenia przedrostków

przedrostek	znaczenie
lpcsz	"długi" ("daleki") wskaźnik na stały ciąg znaków zakończony bajtem zerowym
pfn	wskaźnik na funkcję

  - przykłady użycia notacji (fragment jakiegoś bliżej nieokreślonego programu)

```
void error( const char* cpszFormat, ... )
{
  va_list argList;
  va_start( argList, cpszFormat );
  if ( iColorOutput )
  {
   char szBuff[ 512 ];
   vsprintf( szBuff, cpszFormat, argList );
   attrset( COLOR_PAIR( COLOR_PAIR_ERROR ) );
   printw( szBuff );
   attrset( COLOR_PAIR( 0 ) );
  }
  else vfprintf( stderr, cpszFormat, argList );
   va_end( argList );
}
```

wszędzie **SPACJE**

- Formatowanie kodu, notacja węgierska
  - przykłady użycia notacji (fragment jakiegoś bliżej nieokreślonego programu)

```
/* vars */
int iShowHelp = 0;            // ShowHelp !!!
int iColorOutput = 0;         // iColorOutput zamiast icoloroutput
int iPrintWholeLine = 0;
int iManyPatterns = 0;
int iPatternsCount = 0;
char** pszPatternsArr = NULL;
int iUseTag = 0;
char* pszStartTag = NULL;
char* pszEndTag = NULL;


void shiftText( char* pszLine, int iEndAt, int iCount )
{
  int i;
  for ( i = strlen( pszLine ) - 2 - (iCount - 1); i >= iEndAt; --i )
   pszLine[ i + iCount ] = pszLine[ i ];
}

while ( pszVal != NULL )
{
  char* newVal = getPattern( pszVal );
  int len = strlen( newVal ) + 1;
  pszPatternsArr[ i ] = ( char* )malloc( len * sizeof( char ) );
  strcpy( pszPatternsArr[ i++ ], newVal );
  free( newVal );
  pszVal = ( char* )strtok( NULL, "," );
}
```

wydaje się, że warto pisać programy **"po angielsku"**

wszędzie **SPACJE**

- Formatowanie kodu - kilka prostych ale ważnych zasad
  - unikaj zbędnych nawiasów. Niepotrzebnie zwiększają objętość kodu

```
for(i=0; i<10; i++)
{
  printf ("ABC");
}
```

  - unikaj "ogromnych" wcięć i zbędnych spacji. Preferowane wcięcia na DWIE spacje

```
for(i=0 ; i<10 ; i++)
                    printf ("ABC");              gets (buf);
```

  - stosuj spacje i adekwatne nazwy zmiennych. Unikaj nazw jednoliterowych

```
int a,x,z,y,q,l;     // źle
wynik=fun(a,x,&l);  // źle

int tmp, i, j, wplata, suma, wynik;     // dobrze, ale:
                                        // zaleca się: jedna deklaracja, jedna linia
wynik = fun ( wplata, suma, &wynik);    // dobrze
```

  - poprawnie zamykaj nawiasy

```
for(i=0; i<10; i++)
{
  printf ("aaa");   // wcięcie na dwie spacje
  gets(buf1);
  {
   printf ("bbb");
   gets(buf2);
}}                  // źle
  }                 // dobrze
}                   // dobrze
```

- Formatowanie kodu - kilka prostych ale ważnych zasad
  - "zastrzeżone" nazwy liczników

```
i, j, k, l
```

  - zalecane przyrostki zmiennych wskaźnikowych

```
int * income_ptr,
```

  - zalecane przyrostki struktur

```
struct struct_name_s {
  type field_name;
};
```

  - zalecane przyrostki własnych typów

```
typedef type_definition type_name_t;
```

  - makra z dużych liter

```
#define MACRO_NAME macro text
#define MACRO_NAME(param_1, param_2)
{
  macro text
}

#define MAXTAB 100
#define MAX(A, B) ((A)>(B) ? (A) : (B))
// x = MAX (d+f, h); korzystanie z makra

#undef MACRO_NAME   // kasowanie makra
```


- Formatowanie kodu - kilka prostych ale ważnych zasad
  - nie wstawiaj niepotrzebnie wiele razy definicji. Pliki *.h powinny wyglądać tak:

```
#ifndef __FILENAME_H__
#define __FILENAME_H__
  /* file body */
#endif /* __FILENAME_H__ */
```

  - pliki nagłówkowe "lokalne" i "globalne"

```
#include "my_types.h"         // z katalogu bieżącego
#include <global.h>           // z katalogu INCLUDE
```

  - dziel długie linie na czytelne fragmenty

```
if ((x_pos == START_X || x_pos == END_X) &&
    (y_pos == START_Y || y_pos == END_Y)) {
...
}
```

  - jeszcze raz spacje - wyrażenia algebraiczne i logiczne

```
total_size = element_size * (old_cnt + new_cnt);
old_cnt = new_cnt;
(new_idx < old_idx) || (current_cnt == last_cnt)
```

- Formatowanie kodu - kilka prostych ale ważnych zasad

  – jeszcze raz spacje - operatory unarne - nie, rzutowanie - tak

```
!error;
i++;
*reg_ptr (int *) reg_val;
```

  – komentarze, spacja po dla // oraz przed i po dla /* */

```
// tekst_komentarza
/* tekst_komentarza */
```

  – podwójne nawiasy - bez spacji

```
((new_idx != old_idx) && (current_cnt != last_cnt))
```

  – nazwy wieloczłonowe

```
void PrintTable(int table*);            // dobrze
void print_table(int table*);           // dobrze
void printtable(int table*);            // źle
void PrintTablica(int table*);          // źle pol-ang !!!
int mvbck;                              // źle, niezrozumiały skrót
int move_back;                          // dobrze
```


- Formatowanie kodu - narzędzie

  – stosuj dostępne narzędzia ! Program **indent**

```
The 'indent' program can be used to make
code easier to read. It can also convert
from one style of writing C to another.

'indent' understands a substantial amount
about the syntax of C, but it also
attempts to cope with incomplete and
misformed syntax.
```

PRZED

```
#include <stdio.h>
int main(void)
{
//  zapis jednaj liczby ca³kowitej na plik
 int a=8388608+1;
    FILE * fptr;


if((fptr=fopen("file.bin","wb"))==NULL    )
   printf("\nCan't open file")  ;
         fwrite (&a,sizeof(a ),1,fptr);

 return 0;     }
```

```
NAME
  indent - changes the appearance of a C
  program by inserting or deleting whitespace.

SYNOPSIS
  indent [options] [input-files]
```

PO

```
#include <stdio.h>

int
main (void)
{
//  zapis jednaj liczby calkowitej na plik
  int a = 8388608 + 1;
  FILE *fptr;

  if ((fptr = fopen ("file.bin", "wb")) == NULL)
    printf ("\nCan't open file");

  fwrite (&a, sizeof (a), 1, fptr);
  return 0;
}
```

OGROMNE możliwości formatowania