Styl programowania

- komentowanie programów
- nazywanie zmiennych, typów i funkcji
- rozmieszczenie tekstu w plikach źródłowych

M. Kotowski. „Wysokie C”

Program w **C** powinien być:

- czytelny
- łatwy do testowania i usuwania błędów
- poprawny i niezawodny
- łatwy do modyfikowania i rozszerzania
- przenośny
- defensywny
- spójny stylistycznie
- efektywny
- uniwersalny

DEFENSYWNY: umiejący sobie radzić z błędami (brak pamięci, złe dane, brak pliku, ...)

IOCCC The International Obfuscated C Code Contest

W roku 1984 Wielką Nagrodę zdobył **Sjoerd Mullender** (Holandia) za następujący program:

```
short main[] = {
    277, 04735, -4129, 25, 0, 477, 1019, 0xbef, 0, 12800,
    -113, 21119, 0x52d7, -1006, -7151, 0, 0x4bc, 020004,
    14880, 10541, 2056, 04010, 4548, 3044, -6716, 0x9,
    4407, 6, 5568, 1, -30460, 0, 0x9, 5570, 512, -30419,
    0x7e82, 0760, 6, 0, 4, 02400, 15, 0, 4, 1280, 4, 0,
    4, 0, 0, 0, 0x8, 0, 4, 0, ',', 0, 12, 0, 4, 0, '#',
    0, 020, 0, 4, 0, 30, 0, 026, 0, 0x6176, 120, 25712,
    'p', 072163, 'r', 29303, 29801, 'e'
};
```

http://www.ioccc.org/

Proste przykłady:

```
if (Counter)
{
   ...
```

```
if (!Valid)
{
   ...
```

ZALECANE

```
if (St & BIT_A)
{
   ...
```

```
if (Counter != 0)
{
   ...
```

ZALECANE

```
if (Valid == 0)
{
   ...
```

```
if (St & BIT_A != 0)
{
   ...
```

BŁĘDNE

```
if (Counter != 0)
  State = 1;
else
  State = 0;
```

ZALECANE

```
State = (Counter ? 1 : 0);
```

```
State = !!Counter;
```

```
State = (Counter !=0 );
```

- nie ma jednego najlepszego stylu programowania
- należy wybrać sobie jeden z proponowanych stylów i **konsekwentnie** go stosować

Język **C** po angielsku czy po polsku?

Konsekwentnie: albo po polsku albo po angielsku.

Unikać jednoczesnego używania np. zmiennych **Licznik** i **Counter**, funkcji **GetData** i **WczytajPlik**, czy nawet **ReadPlik**.

Przykład:

```
if (!find_przyst(przyst_name))
  insert_przyst(przyst_name);
```

„Tak mi się napisało. Mogę po polsku i po angielsku”

Można przyjąć zasadę jaką wybrał tłumacz książki K&R: zmienne po angielsku, komentarze po polsku.

Komentowanie programów

Przykład samokomentującej się funkcji:

```
COLUMN IncrementColumn(COLUMN Column)
{
  return (Column < MAX_COLUMN ? Column+1 : 0);
}
```

Przykład samokomentującego się kodu:

```
while (GetNextEmployee(EmplPtr) != 0)
{
  ...
}
```

Ale nie wiadomo po co robiona jest ta pętla oraz komentarza może wymagać jej wnętrze.

Nagłówki komentarzowe funkcji

- nazwa funkcji
- krótki opis operacji realizowanych przez funkcję
- lista argumentów funkcji oraz ich opis
- lista zmiennych zewnętrznych, z których korzysta funkcja
- lista zmiennych zewnętrznych modyfikowanych przez funkcję;
- opis wartości zwracanej (ewentualnie informacje o błędach wykonania)

```
/*-------------------------------------------------|
| SearchInString                                   |
|                                                  |
| FUNKCJA ZNAJDUJE W CIĄGU LICZBĘ WYSTĄPIEŃ CIĄGÓW |
| SPEŁNIAJĄCYCH WZÓR WYSZUKIWANIA                  |
| ARGUMENTY:                                       |
|   Text   - ADRES CIĄGU PRZESZUKIWANEGO           |
|   Image  - ADRES WZORU WYSZUKIWANIA              |
|            (DOPUSZCZA ZNAKI * i ?)               |
|   Status - PARAMETR PRZESZUKIWANIA               |
|             - W_PRZOD, W_TYL                     |
|             - LITERY_WIELKIE, LITERY_MALE        |
|             - CALE_WYRAZY                        |
|               (ZDEFINIOWANE W SEARCH.H)          |
| WARTOŚĆ ZWRACANA:                                |
|   LICZBA ZNALEZIONYCH WYSTĄPIEŃ                  |
|-------------------------------------------------*/

int SearchInString (const char *Text, const char *Image,
                    int Status)
{
  ...
}
```

Komentarze w kodzie

Komentarz blokowy:

```
/* OBLICZANIE ILU PRACOWNIKÓW W ZBIORZE MA GRUPĘ A */
while (GetNextEmployee(EmplPtr) != 0)
{
  ...
}
```

Komentarz w linii:

```
TotalLines=LinesCounter+StartLines+EndLines;
                /* ŁĄCZNA LICZBA LINII TEKSTU NA STRONIE */
```

W osobnym wierszu, jeśli się nie mieści, ale z dużym wcięciem.

Komentarz w linii nie powinien powtarzać tego co opisano w kodzie:

```
Counter++; /* ZWIEKSZENIE ZMIENNEJ Counter O 1 */
```

Lepiej:

```
Counter++; /* UAKTUALNIENIE LICZBY ZLICZANYCH DOKUMENTÓW */
```

Przy złożonych instrukcjach dobrze komentować nawiasy klamrowe:

```
while (GetNextEmployee() != 0)
{
   ...
   if (EmployeeNumber != 0)
   {
      ...
   } /* if (EmployeeNumber != 0) */
   ...
} /* while (GetNextEmployee() != 0) */
```

Komentowanie definicji zmiennych

Zaleca się komentowanie struktur i unii w komentarzu blokowym nad nimi.

Komentarze jako szkielety funkcji

```
PARAGRAPH *AllocatePar(char *Text)
{
  /* ALOKACJA PAMIĘCI NA STRUKTURĘ PARAGRAPH */
  /* ALOKACJA PAMIĘCI NA TEKST */
  /* SKOPIOWANIE TEKSTU DO STRUKTURY I POWRÓT */
}
```

Funkcja zostanie napisana później ale już zawczasu opisano co będzie robić krok po kroku.

Nazywanie zmiennych, typów i funkcji

- nazwy zmiennych i funkcji powinny odzwierciedlać ich sens
- nazwy o pokrywającym się zasięgu powinny różnić się w sposób istotny np.

  Source i **Dist** zamiast **String1** i **String2**
- nazwy nie powinny wyglądać podobnie np.

  LastWord i **LostWord**

  counter i **Counter**

  WordCounter i **Word_counter**

  Index i **Indeks**

- nie tworzyć nazw trudnych do wymówienia np.

  zamiast **FInm** lepiej **FileName**

  zamiast **WrkrIndx** lepiej **WorkerIndex**

  WRZWSZ - propozycja instrukcji w LOGO

- nazwy nie powinny być przegadane

  zamiast **Ptr_To_Employee_Code** lepiej **PtrEmployeeCode** lub **PtrCode**

  zamiast **EmployeeArray[EmployeeIndex] -> EmployeeStatus** lepiej **Empl[Index] -> Status**

- zmienne o podobnym lub identycznym charakterze powinny mieć podobne lub identyczne nazwy np.

  funkcje realizujące podobne operacje na identycznych argumentach powinny mieć parametry identycznie nazwane i w tej samej kolejności,

  jeśli zmienne lokalne w różnych funkcjach przechowują te same dane, to powinny nazywać się identycznie np. **EmplPtr** a nie w jednej **EmplPtr**, w innej **Ptr_To_Empl** a jeszcze w innej **TempPtr**

- im szerszy zasięg zmiennej tym dłuższa nazwa

- nazwy jednoliterowe tylko dla zmiennych lokalnych o małym zasięgu

  stosuje się zasady:

  c - typ **char**

  i, **j**, **k** - typ **int**

  p, **q**, **r** - wskaźniki

  s, **t** - napisy (wskazania na znak)

  x, **y** - całkowite do przechowywania wsp. kartezjańskich

  unikać:

  char i;

  long c;

- różne obiekty powinny mieć różne nazwy np.

  poprawne ale w złym stylu:

```
struct Name
{
  int Name;
}

struct Name Name;
...
k=Name.Name;
```

Tworzenie skrótów

ClearArray - **ClrArr**
InitiateEmployeeStructure - **InitEmplStruc**
IncrementColumn - **IncrCol**

Przykłady standardowych funkcji:
strcpy - string copy
strlen - string length
strcmpi - string compare ignoring case
memset - memory set

stdio - standard input-output
stddef - standard definitions
stdlib - standard library

Notacja węgierska

Nazwa od narodowości jej autora Charlesa Simonyi (twórcy Word i Excel) z firmy Microsoft.

Zasada: na początku nazwy zmiennej umieścić informację o jej typie.

Przedrostki i podstawowe typy:

- **a** - tablica (array)
- **b** - wartość logiczna (bool)
- **c** - znak (character)
- **f** - znacznik bitowy (flag)
- **h** - numer pliku (handle)
- **i** - liczba całkowita (integer)
- **l** - liczba całkowita długa (long)
- **p** - wskaźnik (pointer)
- **s** - struktura (structure)
- **u** - liczba bez znaku (unsigned)
- **z** - ciąg znaków zakończony zerem

Standardowe nazwy własne:

Temp - zmienna tymczasowa

Sav - zmienna, w której przechowuje się pewną wartość a później odtwarza

Prev - wartość przechowana, pochodząca z poprzedniej iteracji

Cur - wartość bieżąca

Next - wartość następna w pewnej iteracji

Nazwy funkcji (zaczynają się z wielkiej litery):

[Typ][Operacja][Argumenty]

PnInsertObj(pObj)

p - wskaźnik

n - node (wierzchołek)

Insert - wstawianie

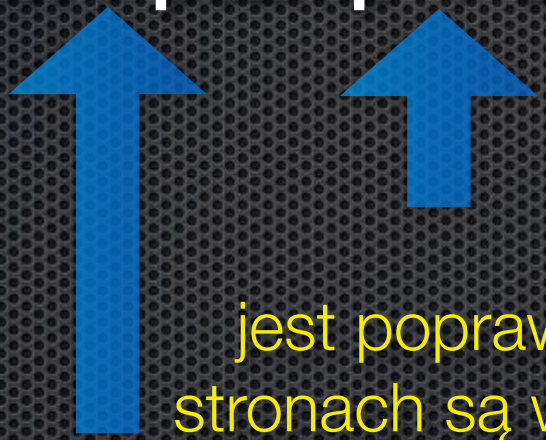
Obj - obiekt (nie jest precyzyjne bo to wskazanie na obiekt a nie on sam!)

„Rachunek typów”:

s - struktura

b - wartość logiczna

bShift = psEmp -> bWorking



jest poprawne bo **psEmp** jest wskazaniem na strukturę

jest poprawne bo po obu stronach są wartości logiczne

Ocena notacji węgierskiej

ZA	PRZECIW
• kontrola „rachunku typów” • sprawdza się w dużych projektach	• zmniejsza poziom abstrakcji • dziwne nazwy • utrudnia modyfikację programu (czasami bywa to zaletą) • więcej przeciwników niż zwolenników

Definiowanie stałych

```
if (Counter < 64)
...
```

nic nie mówi

```
#define PAGE_LENGTH 64
...
if (Counter < PAGE_LENGTH)
...
```

wszystko wyjaśnia

```
Number=fread(Buffer, 20, 30, Fp);
```

czy to przeczytanie 20 rekordów po 30 bajtów, czy 30 rekordów po 20 bajtów?

```
#define RECORD_LENGTH 20
#define NO_RECORDS    30
...
Number=fread(Buffer, RECORD_LENGTH, NO_RECORDS, Fp);
```

nie tylko czytelność programu ale i łatwość modyfikowania

Wyjątki: stałe **0**, **1**, **-1** należy pozostawiać zapisane explicite

Stałe szesnastkowe: **0xA230** lepiej niż **0XA230** albo **0xa230**.

Stałe będące liczbami zmiennoprzecinkowymi lepiej zapisać np. **23.0F** niż jako całkowita z rzutowaniem **(float)23**

```
/* NUMERY KLAWISZY FUNKCYJNYCH */

#define F1 (256+59)
#define F2 (256+60)
#define F3 (256+61)
#define F4 (256+62)
...

/* POLECENIA MENU */

#define HELP F1
#define LOAD_FILE F2
#define INSERT F3
#define DELETE F4
...

switch (PressedKey)
{
  case DELETE:
    ...
  case HELP:
    ...
}
```

Układ kodu (layout, wcięcia)

K&R	Allman

`if (x > y) {`
`  ...`
`}`
	`if (x > y)`
	`{`
	`  ...`
	`}`

Eric Allman twórca sendmail z Kalifornijskiego Uniwersytetu w Berkeley.

Wyjątek w stylu Allmana w przypadku **do-while**:

```
do
{
  ...
}
while (Function(x, y));
```

```
do
{
  ...
} while (Function(x, y));
```

zalecany

K&R oraz Allman stosuje się również przy zapisie funkcji i struktur.

Różne wariacje (za **C/C++ Users Journal** i **Microsoft Systems Journal**):

```
if (Expr1)
{  Expr2;
   Expr3;
}
```

```
if (Expr1)
{  Expr2;
   Expr3;}
```

```
if (Expr1) { Expr2;
             Expr3;}
```

Zalecenie: lepiej klamry w jednej linii.

```
if (x > 0)
  Func(x);
```

```
if (x > 0)
{
  Func(x);
}
```

Nawiasy są zbędne ale mogą się przydać podczas rozszerzania bloku w instrukcji **if**.

Wcięcia

- nie wcina się deklaracji i definicji zewnętrznych
- wcina się treść funkcji i zagnieżdżenia bloków (**for**, **if**, **do**, **while**, **switch**)
- najlepiej dla wcięć przyjąć pozycję 2, 4 albo 8

Różne kształty wcięcia:

```
while (Expr)
{
  x=y;
  ...
}
```

```
while (Expr)
  {
  x=y;
  ...
  }
```

```
while (Expr)
    {
      x=y;
      ...
    }
```

za głęboko

Koniec bloków:

```
    ...
    }
  }
}
```

zalecane

```
    ...
}
}
}
```

```
      ...
}}}
```

Instrukcja **if-else**:

```
if (Expr1)
{
   ...
}
else if (Expr2)
{
   ...
}
else if (Expr3)
{
   ...
}
else
{
   ...
}
```

```
if (Expr1)
{
   ...
}
else
   if (Expr2)
   {
      ...
   }
   else
      if (Expr3)
      {
         ...
      }
      else
      {
         ...
      }
```