# Zasady dobrego stylu programowania

Metody Numeryczne II A

wersja 1.1

Jakub Narębski

22 kwietnia 2001 roku

Poniższe zasady nie mają w żadnej mierze charakteru wiążącego. Jednak, ponieważ przy ocenie programu brana jest także postać kodu warto ich przestrzegać. Za przestrzeganie poniższych zasad można dostać dodatkowe punkty. Punktacja będzie obniżana wyłącznie za kod bardzo nieczytelny, zawikłany i trudny w odczytaniu.

Przy oddawaniu programów za pomocą e-mail powinny bardziej przestrzegać poniższych zasad. Ważne jest szczególnie odpowiednie nazewnictwo funkcji oraz stosowanie komentarzy. Absolutnym minimum jest komentarz nagłówkowy programu zawierający nazwę zadania, autorów programu i ich grupę ćwiczeniową oraz kompilator (Borland C, gcc) którym program był kompilowany. Ogólnie: należy dbać o przejrzystość kodu programu.

Program oddawany za pomocą e-mail musi się kompilować pod Linuksem (ewentualnie można przysłać skompilowany program wykonujący się pod DOS-em wraz z niezbędnymi plikami — należy jednak liczyć się z dłuższym okresem sprawdzania zadań).

## 1 Zasady ogólne

Dbaj o prostotę kodu. Staraj się nie stosować złożonych konstrukcji bez potrzeby. Staraj się, by kod odzwierciedlał twoje intencje. Przykładowo, jeśli funkcja przyjmuje jako jeden z argumentów jednowymiarową tablicę, deklaruj ten argument raczej jako np. **float** arr [] niż **float** *arr.

Ponieważ w nazwach funkcji nie można stosować liter spoza zakresu a–zA–z, nazwy funkcji staraj się pisać po angielsku. Jeśli jednak nie znasz na tyle angielskiego, pisz je całkowicie po polsku. Dbaj o jednolitość: pisz albo wszystkie nazwy po angielsku, albo wszystkie (i w całości!) po polsku bez polskich literek. Komentarze pisz po polsku, do wyboru z użyciem polskich znaków narodowych (w standardzie czytelnym w danym środowisku) lub bez.

Dbaj o spójność stylistyczną kodu, t.j. stosuj wszędzie te same zasady. Pisz kod tak, by móc po roku zrozumieć jak dany program działa.

## 2 Komentowanie

Komentarze można pisać według wyboru: albo wszystko wielkimi literami, wszystko małymi, albo jako pełne zdania.

Każdy program powinien mieć nagłówek komentarzowy, w którym znaleźć powinny się przynajmniej nazwa programu, numer i/lub nazwa zadania, numer grupy i autorzy programu. Nagłówek może zawierać datę napisania (lub oddania) programu, opis zadania i użytej w rozwiązaniu metody. Można także w nagłówku wymienić ważniejsze funkcje programu. Jeśli program wymaga do kompilacji specjalnych parametrów, albo wykorzystuje rozszerzenia kompilatora czy też funkcje biblioteczne specyficzne dla danego środowiska należy podać nazwę i numer wersji kompilatora, za pomocą którego kompilowano program i jego parametry kompilacji (np. dołączane biblioteki czy model pamięci w przypadku programów DOS-owych).

Każda funkcja (z wyjątkiem być może oczywistych przypadków) powinna mieć nagłówek komentarzowy zawierający przynajmniej krótki opis funkcji (co ona robi), np.

```
/* równanie różniczkowe oscylatora harmonicznego */
void harm(float t, float x [], float res [])
```

Do wyboru pozostaje, czy komentarz pisany jest przed czy po nagłówku funkcji. W nagłówku komentarzowym znaleźć się może także nazwa funkcji, opis funkcji i zastosowanego algorytmu (lub przynajmniej podana nazwa zastosowanego algorytmu), opis argumentów funkcji (w szczególności jeśli powinny one spełniać jakieś warunki), opis wartości zwracanej (jeśli wartość jest zwracana [także] w jednym lub kilku argumentach należy to napisać), opis modyfikowanych zmiennych globalnych oraz rodzaj błędów i sposób ich sygnalizacji.

Komentarze w kodzie powinny być pomocą w zrozumieniu sensu operacji, a nie je opisywać, np.

```
Counter++; /* zwiększenie zmiennej Counter o 1 */      — źle
Counter++; /* następny krok metody */                 — dobrze
```

Komentarze w kodzie powinny być dobrze oddzielone od kodu. Nie należy przeplatać zdania kodu i zdania komentarza. Komentarze odnoszące się do jednego zdania języka można pisać za nimi, w tej samej linii lub linię niżej odpowiednio wyrównane, np.:

```
t += dt; /* tutaj krótki komentarz */
tmp = f(a, b) + 0.5*g(a, b, c) + 0.6*f(b, a);
  /* tutaj długi komentarz, który nie zmieścił by się umieszczony w linii */
```

Komentarz dotyczący grupy zdań powinny być umieszczony przed nimi, wyrównany stosowanie do komentowanego kodu.

Nie zagnieżdżaj komentarzy. Jest to nieprzenośne. Nie używaj komentarzy // w programach napisanych w C (i kompilowanych jako programy w C, a nie w C++).

Nie stosuj komentarzy do zasłaniania fragmentów kodu przed kompilatorem (wyłączania fragmentu z kompilacji). Nie do tego służy komentarz. Stosuj do tego dyrektywy warunkowe preprocesora z odpowiednią niezdefiniowaną stałą symboliczną lub z wartością 0, np.

```
#ifdef XXXXX_CODE_BLOCKED_XXXXX
  /* tutaj kod wyłączony z kompilacji */
  ...
#endif
```

W (X)Emacsie istnieje tryb (minor mode) Hide-Ifdef, służący do ukrywania „nieaktywnych” fragmentów kodu. Włącza się go za pomocą `M-x hide-ifdef-mode`. Oznaczany jest on za pomocą Ifdef w wierszu statusu (modeline). W tym trybie można zażądać, by kod wewnątrz konstrukcji **#ifdef**, które preprocesor C wyeliminuje (i nie będą podlegały kompilacji), nie był widoczny. Ukrywanie włącza się za pomocą `C-c @ h` (`M-x hide-ifdefs`), a wyłącza za pomocą `C-c @ s` (`M-x hide-ifdefs`). Ukrywanie zaznaczane jest za pomocą Hiding w wierszu statusu. Deklarować które stałe symboliczne są zdefiniowane/niezdefiniowane można za pomocą `C-c @ d` (`M-x hide-ifdef-define`) i `C-c @ u` (`M-x hide-ifdef-undef`).

## 3 Nazewnictwo

Nazewnictwo funkcji powinno być jednolite. Do wyboru pozostaje, czy nazwy funkcji piszemy małymi literami, oddzielając od siebie części nazwy za pomocą znaku podkreślenia _ (np. print_matrix), czy też rozpoczynając każdą część nazwy od wielkiej litery (np. PrintMatrix).

Nazwa funkcji nie powinna wprowadzać w błąd: np. jeśli funkcja liczy konkretną wartość pomocniczą jakąś metodą, nie powinna nazywać się jak ta metoda, sugerując ogólność której nie posiada. Jeśli funkcja zakłada coś o swoich argumentach, można to zapisać w jej nazwie. Jeśli argument wejściowy jest używany w wersji zmodyfikowanej (np. używa się i − 1 zamiast i), należy to zaznaczyć za pomocą komentarza.

Nazwy zmiennych powinny być rzeczownikowe, np. Row (czy też row). W miarę możliwości powinno się stosować takie oznaczenia jak w stosowanym algorytmie, opisując w komentarzach wszelkie rozbieżności (np. że A[i][k] dla $i < k$ to $l_{ik}$ w algorytmie). Nazewnictwo powinno być jednolite; jeśli np w jakiejś funkcji stosuje się nazwę buffer na bufor znakowy, to powinno się stosować tę samą nazwę we wszystkich funkcjach.

Nazwy jednoznakowe powinny być kodowane małymi literami (chyba że zgodność oznaczeń z metodą lub algorytmem wymaga inaczej: np. oznaczanie macierzy wielkimi literami). Wskazane jest stosowanie się do powszechnie przyjętych zasad nadawania zmiennym nazw jednoliterowych. I tak, zmienne zawierające indeksy pętli powinny być nazywane i, j, k (o ile indeks nie ma konkretnego znaczenia, np. step czy krok), zmienne znakowe c i d, parametry a, b (o ile parametry te nie mają konkretnego znaczenia jak omega czy data (dane)).

Stałe symboliczne i makrodefinicje definiowane za pomocą #**define** powinny być kodowane wielkimi literami, zwłaszcza jeśli szacują argumenty więcej niż jeden raz (tzn. co najmniej jeden argument pojawia się w ciele makrodefinicji więcej niż raz). Nazwy definiowane za pomocą **enum** i **typedef** też mogą być kodowane wielkimi literami (czasami inne względy, np chęć używania nowego typu jak typów standardowych, przemawiają za innym kodowaniem nazw).

Nie używaj „liczb magicznych". Poza prostymi przypadkami (np. inicjowanie wartością 0 licznika, czy indeksy w konkretnej metodzie wielokrokowej), stałe w programie definiuj symbolicznie (nie dotyczy to inicjalizacji zmiennych). Jeśli nie wymaga to włożenia dużej ilości pracy stałe które można łatwo zmieniać bez rekompilacji programu (np. ilość iteracji) zapisuj w zmiennych. Warto dać użytkownikowi możliwość zmiany parametrów, jeśli nie w obecnej wersji programu, to może w przyszłości.

Jeśli pewna grupa funkcji realizuje podobne zadania i operuje na identycznych argumentach, to nazwy tych argumentów powinny być identyczne.

# 4 Układ kodu

Należy stosować pionową adiustację kodu. Wszystkie zdania należące do tego samego bloku powinny się zaczynać w tej samej kolumnie. Wcięcia w kodzie powinny być jednolite. Bloki wewnętrzne powinny być wcięte w stosunku do bloku zewnętrznego. Jeśli stosujesz wcinanie za pomocą znaków tabulacji, ustaw tabulator w swoim edytorze na rozsądną wartość (np. 2 lub 5).

Staraj się, aby długość linii kodu nie przekraczała 80 znaków w linii. W Emacsie można włączyć tryb automatycznego łamania linii za pomocą `M-x auto-fill-mode` (przy użyciu mojego pliku `.emacs` funkcja ta podpięta jest pod kombinację klawiszy `C-x C-a`). Następne wywołanie wyłącza ten tryb; przy włączonym trybie w wierszu statusu pojawia się `Fill`.

Łamanie zarówno wywołań funkcji, jak i wyrażeń arytmetycznych powinno być jednolite. Łamać należy na przecinkach oddzielających argumenty funkcji i na operatorach o w miarę możliwości najniższym priorytecie. Operator powinien być pozostawiany na końcu linii przenoszonej. Używaj rozsądnie odstępów między operatorem a jego argumentami (zaznaczając np. porządek działań/-priorytet operatorów).

Przy łamaniu literałów znakowych należy je raczej kodować w postaci oddzielnych ciągów, sklejanych przez kompilator, niż je przenosić, np.

```
printf("Tutaj następuje bardzo długi opis jakiejś funkcji, "
       "dlatego zostanie podzielony na dwa ciągi.");
```

Unikaj kodowania w postaci jednego ciągu zdań języka. Staraj się je rozdzielać pustymi liniami, grupując polecenia. Deklaracje zmiennych oddzielaj od kodu przynajmniej jedną linią pustą.

Wybór stylu kodowania nawiasów klamrowych pozostaje w gestii programisty. Ważne jest zachowanie konsekwencji (z wyjątkiem nawiasów klamrowych w definicjach funkcji i definicjach struktur), w czym w przypadku pisania w Emacsie może pomóc odpowiedni wybór stylu kodowania (patrz informacje o CC Mode).

Oto przykłady używania nawiasów klamrowych; z lewej styl K&R (w.g. terminologii Emacsa "gnu"), z prawej styl Allmana (inaczej "bsd"):

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

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

Można stosować też inne style, byle konsekwentnie. Nawiasy klamrowe definicji funkcji należy pisać raczej w stylu Allmana czyli:

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

zaś w pętli **do** ...**while** należy pisać **while** w linii, w której znajduje się nawias klamrowy zamykający blok:

```
  do
  {
    ...
  } while (...);
```

# 5 Używanie danych

Unikaj stosowania zmiennych globalnych. Jeśli to możliwe, przekazuj odpowiednie dane jako argumenty funkcji.

Jeżeli dany argument w funkcji nie ma być modyfikowany, opatrz go kwalifikatorem **const**. O ile to możliwe, używaj zmiennych przekazywanych w wywołaniu funkcji jako zmiennych roboczych. Dane wejściowe funkcji powinny być (o ile to możliwe) tylko czytane, a wyjściowe tylko pisane.

Nie modyfikuj literałów (stałych) znakowych. Literały znakowe mające części wspólne (np. `"h(t) = sin(t)"` i `"h(t) = cos(t)"`) mogą być łączone: modyfikując jeden możesz przypadkowo zmodyfikować drugi.

# 6 Używanie kodu

Unikaj głębokich zagnieżdżeń kodu, chyba że wynika to w sposób naturalny z algorytmu (metody).

Nie porównuj (operatory ==, =!) zmiennych typu zmiennoprzecinkowego. Przykładowo używaj konstrukcji

```
fabs(a - b) <= EPS
```

(lub podobnej) do porównania dwu liczb zmiennoprzecinkowych.

Używając w kodzie stałych symbolicznych, staraj się w miarę możliwości by w wyrażeniach nie pojawiały się ze stałymi addytywnymi (chyba, że wynika to z algorytmu), np. zamiast

```
for(i = 0; i <= N-1; i++)
```

pisz

```
for(i = 0; i < N; i++)
```

Rozmiar funkcji nie powinien przekraczać kilkudziesięciu linii kodu, chyba że wynika to z algorytmu. W definicji funkcji zawsze podawaj jej typ (nawet jeśli jest to domyślny typ **int**): jeśli funkcja nie zwraca wartości zaznacz to wartością **void**. Jeżeli funkcja nie ma argumentów, pisz argument pusty **void**.

Staraj się, by funkcja realizowała tylko jedną operację, ale za to realizowała ją w sposób pełny i skończony. Jeżeli po wywołaniu danej funkcji (i być może innych) należy wywołać inną, powinno być to oczywiste (nie powinno być np. tak, że przy testowaniu kilku metod jedna z nich musi być wywołana jako ostatnia, gdyż zaszyte są w niej funkcje porządkowe dla całego zestawu metod).

# 7 Uwagi końcowe

Powyższa wersja zasad jest wersją wstępną. Nie powinno być większych zmian, mogą dojść dalsze porady. W następnych wersjach powinny dojść informacje o konfiguracji Emacsa (a dokładniej CC mode) przy stosowaniu określonych konwencji.