コーディングスタイルについて

コーディングスタイルとはソースコードのレイアウトのことです。
空白やカッコをどこにいくつ入れるか、どこで改行するか、コメントはどこにつけるかといったこともろもろを、全部まとめてコーディングスタイルと言います。

Ｃはフリーフォーマットな言語です。Web で、The International Obfuscated C Code Contest（難解 C コード国際コンテスト）の入賞コードを見つけて参照してみてください。Ｃのコーディングスタイルにはこれほどの自由度があります。

しかしこのようにわかりにくいコードを書いてバグが出たときに、果たしてデバッグは可能でしょうか（入賞者たちはきっとデバッグに非常な苦労をしたでしょう)。また、ちょっとした機能を追加するとき、ためらいなく手をつける気になれますか？

仕事では、わかりやすく、保守が容易で、一貫しているコーディングスタイルが求められます。実際にはプロジェクトごとにコーディングスタイルを決めることも多いようです。

以下にあるのはコーディングスタイルガイドの一例です。研修ではここに書いてあるコーディングスタイルに従うものとします。最初は面倒かもしれませんが、これもマナーの１つと考えてください。

1. インデントとタブ

インデント（行ごとの字下げ段差）は４桁とします。

```
if (a > 1)  {
    x = a + 1;
    for (i = 0; i < n; i++)  {
        array[i] = func(x, i);
        if (array[i] < 0)  {
            fprintf(stderr, “ERROR: Invalid value %d¥n”,
        }
        else {
            foo(array[i]);
                :
```

インデントにはタブが利用できますが、タブ幅の設定は人によって異なり、４桁に設定している人と８桁の人とがいます。タブ幅設定が異なると、自分のエディタでは問題なく表示されても他の人のエディタではガタガタに表示されてしまいます。これを防ぐため、タブキーを押したときには空白文字が入力されるようにエディタを設定しておくことをお薦めします。

（サクラエディタでの設定例）

```
「設定」→「タイプ別設定」→「スクリーン」タグ
                          →「SPACE の挿入」にチェック
```

2. 関数宣言

関数宣言は、

```
int foo(
    char        *name,          /* 名前 */
    int         age)            /* 年齢 */
```

のように書きます。

あるいは、

```
/*
 * 関数名 : foo
 * 機能   : 名前と年齢から、その人の寿命を求める
 * 引数   : char        *name   : 名前
 *          int         age     : 年齢
 * 戻り値 : (int) 寿命
 */
int foo(char *name, int age)
```

と関数へのコメントとあわせて書いても良いです。

3. 変数宣言

```
    char        *str;           /* 説明 */
    int         x;              /* 説明 */
```

のように書きます。
型が同じなら
(1)

```
    char        *from;          /* 説明 */
    char        *to;            /* 説明 */
```

のようにも書けるし、カンマで区切って
(2)

```
    char        *from, *to;     /* 説明 */
```

のようにも書けます。
しかし、追加や削除のしやすさ、およびコメントとの対応を重視して、(1)を採用します。

変数宣言に対するコメントは、できるだけつけるようにします。コメントが難しい、つまり変数の役割をひとことで表すことができないならば、それは設計がまずいといえます。変数の役割を明確にしましょう。

一つの変数を異なる目的で使うことはしないこと。わずかな記憶領域をけちるより、わかりやすさを優先します。

上の例では char と *from がとても離れているように感じます。もうすこし近くてもよい

かもしれません。しかし構造体変数を使うようになると、このくらいがちょうどよく思えてきます。

```
char            *buf;
struct stat     st;
```

要は、buf と st を揃えたいのです。次のようだと、ぱっと見てわかりにくいです。

```
char    *buf;
struct stat     st;
```

ポインタをあらわす * （アスタリスク）の位置には、二通りの書き方があります。

```
  char          *ptr;   //ポインタの書き方１
```

```
  char*         ptr;    //ポインタの書き方２
```

研修では、前者の書き方を採用します。

4. 関数呼び出しと制御文のカッコ

次のように、関数名とカッコは離さずに書きます。

```
c = fgetc(fp);
```

if, for, while, switch といった制御文では、カッコの前にスペースを置きます。

```
if (a > 0)
```

```
switch (c)
```

慣例として、return 文にはカッコをつけません。

```
return 0;
```

```
return a + b;
```

5. 制御文の中カッコの位置

もっとも論争が起きる点です。いくつか例を挙げてみましょう。

```
if (a > 0) {
    printf( ... );
}
```

例 1　　←これを採用します

```
if (a > 0)
{
    printf( ... );
}
```

例 2

```
if (a > 0)
    {
    printf( ... );
    }
```

例 3

書籍などでは例 1 が多いようです。これは、C言語の「バイブル」と呼ばれたりする『プログラミング言語C』(カーニハン・リッチー著　共立出版) という本で採用しているスタイルであること、また行数が少なくて済むということなどが理由でしょう。
ただし、関数の最も外側の中カッコは通常１カラム目に書きます。

```
int  func( void )
{
    int     a;
        :
        :
}
```

例 2 は if の判定文と if の中身が連続した行にならないので見やすい、カッコの対応がわかりやすいなどのメリットがあります。

例3の書き方は、GNUのコーディング規約にありますが、それ以外ではあまり見かけません。

6. 改行について

一行に複数の文を書くことは、避けます。

```
x = a + 1;  y = b + 2;                    // 不可
```

if 文や if 文の else 節、while 文なども、改行します。

```
if (v < 0)  printf(stderr, "...");      // 不可
```

```
if  (a > 1) {
    x = a + 1;
}
else {
    y = b + 2;
}
while (i < n) {
  *p++ = str[i++];
}
```

コンマ式も、for 文の中以外では使用しません。

```
z = c + 1,  c++;                                  // 不可
```

```
for (i = 0, j= 0; i < n && j < m; i++, j++) {  // 可
          :
```

7. 制御文での{中カッコ}の省略

次のような場合です。

```
if (fp == NULL)
    fp = stdin;
```

このような場合、fp == NULL のときの処理を追加しようとして、中カッコをつけ忘れることがあります。

```
if (fp == NULL)
    fprintf(stderr, “標準入力より読み込みます。¥n”);
    fp = stdin;
```

すると、条件（fp == NULL）にかかわらず fp = stdin; は実行されてしまいます。このようなミスの予防も兼ねて制御文が実行するコードが１行だけのときも中カッコでくくるようにします。

```
if (fp == NULL) {
    fp = stdin;
}
```

8. switch 文のインデント

switch 文も様々な書き方がありますが、研修中は以下のように統一します。

```
switch (errcode) {
    case MANY_ARG:
    case LESS_ARG:
            puts(“NG”);
            break;
    case NORMAL:
            break;
    default:
            puts(“INVALID_ERROR”);
            break;
}
```

中カッコの位置は「6. 制御文の中カッコの位置」に準拠します。処理の内容と見易さを考慮し、”:” 以降には（コメント以外は）何も書かず、各処理と “break;” はインデントします。

9. 演算子周りのスペース

次のように、演算子の前後にはスペースを置きます。

```
area = width_x * width_y;
```

ただし、++ や -- にはつけません。
また、ポインタ演算子としての * は、次のように書きます。

```
c = *p;
```

アロー演算子にはスペースを空けません。

```
size = moon->magic;
```

10.式まわりのカッコ

if 文では条件式が複雑になることがあります。

```
if (a < b || c < d && e < f)
```

と

```
if ((a < b) || ((c < d) && (e < f)))
```

では、後者のほうが悩んだり間違えたりする余地が少なくなります。

演算子の順序も忘れがちです。

```
val = higher << 16 + lower;
```

これは次のコードと同じです。

```
val = higher << (16 + lower);
```

シフト演算子は優先順位が低いので要注意。次のように書きます。

```
val = (higher << 16) + lower;
```

結論として、優先順位で悩まないよう、極力カッコをつけるようにすること。
誤解の余地のないコードを書きましょう。

11.コメントについて

コメントには、独立した行に書くコメントと、行末コメントがあります。
独立した行に書くコメントのインデントは、コメントしようとするコードにあわせます。

```
/* ... */
for ( ... ) {
    /*  ... */
     ...;
}
```

行末コメントは、前後の行でカラムをそろえるようにします。

```
    char        *from;          /* 説明 */
    char        *to;            /* 説明 */
```

特に、関数の引数や変数宣言では、コメントを必ずつけるようにします。

コメントの重要な目的は「意図」を記述することです。つまり「何をしようとしているか」を読む人に知らせることです。読み手にとって、コードがどのような動作をするかを理解するよりも、その動作の意図を理解することの方が難しいものです。

時々、修正時に削除した部分をコメントアウトして残しておく、というものが見受けられます。

```
/*  if (a == 0) {       2007.12 削除 */
/*      hogehoge();     2007.12 削除 */
/*  }                   2007.12 削除 */
```

このようなコメントは、ソースプログラムを読みにくくする以外何の役にも立ちません。
変更の履歴は、ソースファイル中のコメントではなく、専用のツール（CVS、SCCS など）を使用して管理してください。

12.名前一般

変数や関数には、そのものをよく表す名前をつけます。そのような名前が浮かばない場合、その役割があいまいになってはいないでしょうか？ 異なる複数の目的のために使用されてはいないでしょうか？ 変数や関数はある単一の目的のためだけに使用されるべきで、そうでないなら分割します。

Ｃでは古くから関数名や変数名に小文字を用いるのが普通でした。最近ではそうでもなくなってきましたが、まあ小文字が基本でしょう。アンダーバー（_）も名前に使え、単語の区切りのように用いることができます。

```
get_filename(&buf, sizeof(buf));
```

大文字をまぜる命名法では、おそらく次のようになります。

```
getFilename(&buf, sizeof(buf));
```

動詞＋対象 という命名がよくされます。

#define したマクロでは、大文字で名前をつけます。

```
#define         MAX_NAME        10
```

一般に、次の変数は次のような目的で使われるものと思われています。

i, j, k	int 型で、ループカウンタに用いられます
c	char あるいは int 型で、文字を入れます
s	char * 型で、文字列
n	何らかの数やカウンタ（int 型？）
p	何らかのポインタ
fp	FILE * 型で、ファイルポインタ

このようなものは慣習と異なる別な意味で用いないようにします。もっと具体的な名前が考えられる時は、そちらを使いましょう。

13. 行数（サイズ）について

大きすぎる関数は、見通しが悪く、理解することもテストすることも難しくなります。したがって、バグも多く発生します。関数が大きすぎるということは、不必要に複雑であり、機能分割が不十分であることを意味します。
一つの関数のサイズは、空白行を除いて、およそ２００行以内にします。それを越すようでしたら、機能をよく整理し、複数の関数に分割します。

また、一つのファイルのサイズはできれば１０００行以内に、大きくても２０００行以内にします。

14.バグを未然に防ぐ assert

コーディングスタイルの話からはそれますが、assert の話。関数の引数チェックに用いることを「強く」推奨します。

assert は、デバッグ用の関数で assert(式);と書きます。式が成立していれば何もしないが、式が不成立の場合はエラーメッセージを出して終了します。

たとえば正の数を引数としてとる関数があるとします。設計では、正の数かのチェックはその関数を呼ぶ側が行い、正の数しか渡ってこない「ことになっています」。assert はこのような場合に用います。

```
#include        <assert.h>
#include        <stdlib.h>

int foo(
    int         count)          /* 正の数 */
{
    assert(count >= 0);         /* 引数のチェック */
        :
}
```

assert なしで、もし負の数が渡されたらどのような動作をするのでしょうか。正の数が来る「はず」なのでわかりません、でしょうか。わかりません、というのは厄介なバグを生む大きな原因です。一見正常そうに動作してしまうかもしれません。それは大変に危険です。

assert を追加すると、プログラムは極端に明解な動作をします。仮定していない値が渡されたらエラー終了するのです。

ここで assert は内部的な不整合を検出している点に注意してください。間違えているのはユーザーではなく、この関数を作った人、つまり開発者です。このあたりの切り分けはなかなか難しいのですが、大事なことです。

ユーザーの入力エラーに対しては、assert は用いず、もっと美しくエラー終了すべきです。はじめは assert にしておき、後でエラーメッセージ表示処理に置き換えていく方法もありますが。ともかくプログラムから「このようなときの動作はわかりません」という状態をなくすことです。

15.最後に

以上、研修で採用するコーディングスタイルの説明およびアドバイスでした。

コーディングスタイルは誰でも自分の慣れたスタイルが書きやすく読みやすいです。また、それぞれのスタイルにはそれぞれの良さがあり、どちらが優れていると判定できない項目が多数あります。

では何故プロジェクトで統一するかというと、個々がバラバラのスタイルを採用すると全体として非常に見難いプログラムが出来上がってしまうからです。ソースを納める前に indent コマンドを実行する事を求められるジョブもあるくらいです。(indent コマンドはソースコードを設定したコーディングスタイルに矯正する) プロジェクトでどのコーディングスタイルを採用するかはまちまちです。

ということで、以上に述べたコーディングスタイルは絶対ではありません。重要なのは、スタイルを統一することにより、読みやすく美しいプログラムを書く、ということです。

以上