治郎吉コーディング規約

2005年9月13日　椎名

ここでコーディングをすることになったインターン生や
アルバイトの人は、「治郎吉コーディング規約」という緑色のファイルと
Webページ(http://jirokichi.jp/products/download/JSDN/references/coding_standard.html)
を参照し、実際のソースコードを読ませてもらえば、おおよその規約は
わかるはずである。
この文書は、ここ治郎吉商店におけるコーディング規約を
補足する目的で書かれたものである。
既存のコーディング規約にはない慣習を
　　1.コーディング規約の補足
　　2.コメント
の2つに分けて記した。

1．　コーディング規約の補足

「治郎吉コーディング規約」にないものを補足する。

1.1 命名規則

1.1.1 bool型の値を返す関数名は 'Is' で始める。

(例)

```
// タグオブジェクトであるか
if( IsTagObj( *ioMetaDataPtr ) == false )
{
	return kYAO_Failure;
}
```

(ポイント)

Has などではじめることもある。
要するに条件判定に使われる関数であることが
一目でわかるような工夫をしろということ。

1.1.2 型名

(ポインタ)

```
(1) YAO_METADATA*   metaDataPtr;
(2) YAO_METADATA  *  metaDataPtr;
(3) YAO_METADATA   *metaDataPtr;
```

いろいろな流儀があるが(1)を採用するのがよい。
"*"までで1つの「ポインタ型」であることが明確に示されており、
「ポインタ型」に対する検索がやりやすい。
他の方法でも良いが、どれか1つに決めたら一貫してその
方法を採用することが重要である。

(文字列型)

次の変数は文字列として扱われる。

```
char* filePathPtr;
```

後ろにPtrをつけると、ポインタ変数であることが一目でわかるが
一般的にいって char* は「char へのポインタ」
というより「文字列型」という非ポインタ型として認識されることが多い。
よってこの場合に限り、接尾語Ptrは必要ない。

(型名の置き換え)

治郎吉コーディング規約によると型名は
全て大文字で、単語ごとに'_'で区切ることになっている。
構造体に対してはこの規則を適用しなければならない。

```
typedef struct YAO_FILE
{
	FILE* filePointer;
	YAO_METADATA currentMetaData;
	bool currentExist;
} YAO_FILE;
```

defineまたはtypedefによるデータ型名の置き換えはこの限りではない。
次の例では型名に小文字を含んでいる。

```
#define YAOStatus  long
#define YAOTag     long
#define YAOSize    unsigned long
```

1.1.3　その他

治郎吉コーディング規約によると定数名の頭には次のように k が付く。

```
kYAO_Success = 0;
```

このような定数名はMacのAPIの中にも見られる。例えば

```
enum {
	...
	kCurrentProcess = 2
};
```

など。他にも、治郎吉における命名規則にはMacのAPIを
踏襲していると思われるものが多いので、
Macによるプログラミングを扱ったWebサイトや書籍を
参照するとよい。
（参考）

```
Apple Developer Connection (http://developer.apple.com)
```

1.2 関数に渡す出力用引数

結果の出力先を引数にとる関数について、出力先は引数リストの最後の方で
指定させる。なお出力用引数の名前は必ず out （またはio） で修飾される。
（例）

```
//-----------------------------------------------
// 関数 : YAO_GetData
// 機能 : カレントのデータを取得する
// 返値 : kYAO_Success            ... 成功
//        kYAO_Failure            ... 失敗
//        kYAO_FileIO_Err         ... ファイル入出力エラー
//        kYAO_NoCurrent          ... カレントが無い
//-----------------------------------------------
YAOStatus YAO_GetData( YAO_FILE * ioFile, YAOSize inBufferLen, void * outBuff );
```

この関数はファイルから読み込んだデータを与えられたバッファ outBuff に格納する。
出力用引数は、引数リストの最後で指定する。
これは、入力を処理した結果を「最後に」格納するという意図を明確にするためである。

1.3 **return**文

return <複雑な式>; という形のreturn文を書いてはいけない。
極力return <変数>; の形にする。
（例）

```
position = (YAOSize)ftell( ioFile->filePointer );
return position;
```

この例では、position への代入後、その値を参照するコードを挿入する
必要が生じたときに、修正がしやすくなる。

1.4 視覚的要素

1.4.1 可能ならば、代入や変数宣言が連続する箇所でタブを用いて、見やすくする。
（例） 変数宣言

```
YAO_METADATA        newMetaData;
YAO_METADATA        prevTagMetaData;
YAO_METADATA        nextTagMetaData;
```

（例） 代入

```
ioMetaDataPtr->dataTag                = 0;                  // タグを0に初期化
ioMetaDataPtr->length                 = 0;                  // データの長さを0に初期化
ioMetaDataPtr->filePointer            = 0;                  // ファイル中の位置を0に初期化
ioMetaDataPtr->previousMemberPointer  = kMemberNotFound;    // 前のメンバなし
ioMetaDataPtr->nextMemberPointer      = kMemberNotFound;    // 次のメンバなし
ioMetaDataPtr->previousTagPointer     = kTagNotFound;       // 前のタグなし
ioMetaDataPtr->nextTagPointer         = kTagNotFound;       // 次のタグなし
```

1.4.2　1つの処理単位ごとに空行を置く。
（例）

```
// ファイルを読み書き両用、バイナリモードで開く
yp->filePointer = fopen( inFilePath, "w+b" );
if ( yp->filePointer == NULL )
{
	free( yp );
	return NULL;
}
```

```
                                                        <- 空行
    // ファイルヘッダを作成
    memset( &fileHeader, 0, sizeof(YAO_FILE_HEADER) );
    fileHeader.signature[0] = 'Y';
    fileHeader.signature[1] = 'A';
    fileHeader.signature[2] = 'O';
    fileHeader.version = kYAOVersion;
                                                        <- 空行
    // ファイルヘッダを書き込む
    rc = MyWriteData( yp, &fileHeader, sizeof(YAO_FILE_HEADER) );
    if( rc != kYAO_Success )
    {
        return NULL;
    }
```

1.5 **for**文, **if**文を単一の文として書かない。

(悪い例)

```
    if( rc != kYAO_Success )
        return NULL;
```

(良い例)

```
    if( rc != kYAO_Success )
    {
        return NULL;
    }
```

(ポイント)

{}でブロックを作っておくと、ブロックの中に処理を追加する変更がしやすくなる。

2.コメント

2.1 分かりにくいコメント

2.1.1 あいまいなコメント

(悪い例)

```
    // エラーチェック
    if( ioFile == NULL )
    {
        return;
    }
```

(良い例)

```
    // ファイルが指定されているか?
    if( ioFile == NULL )
    {
        return;
    }
```

(ポイント)

* 何をチェックするのかをはっきりさせる。
* 疑問符に1バイトコードの「?」ではなく2バイトコードの「?」を
  使用すると見やすいだけでなく、
  コメント部が検索されなくなるというメリットがある。
  (コメント部には極力日本語を使う。)
* エラーチェックしている部分を検索したいときは、
  キーワードに「?」を指定すればよい。

2.1.2 逐語訳

(悪い例)

```
    // 前のメンバが存在しないものとする
    previousMemberPointer = kMemberNotFound
```

(良い例)

```
    // 先頭メンバにする
    previousMemberPointer = kMemberNotFound
```

(ポイント)

悪い例のコメントは次の例と同じような、単なる逐語訳であり、
コメントする意味がない。

```
        // count に1を足す
        count = count + 1;
```

良い例のように、プログラマの意図がわかるようなコメントをつける。

2.2 コメントの位置

（例）

```
// 新しく挿入するメタデータを作成    ...(1)
InitMetaData( &newMetaData );
newMetaData.dataTag = ioFile->currentMetaData.dataTag;          // カレントと同じタグを指定 ...(2)
newMetaData.filePointer = MyGetCurrentFilePointer( ioFile );    // ファイルの末尾を指定      ...(2)
newMetaData.length = inDataLen;
```

（ポイント）

(1)...　1つの処理単位と認められるコード群の頭に、1行とって処理内容を記述

(2)...　各行に対する特記事項は横に記述

なお、どの範囲の処理に対してコメントしているのかわからないような
記述は避ける。とくに、複数の処理単位を対象としたコメントは必要ない。

（例）

```
//===== チェーン構造の書き換え =====//                    <- いらない
// 選択オブジェクトと同じタグを持つメンバーオブジェクトがまだあるか?
if( inCurrentDataInfo->nextMemberPointer == kMemberNotFound )
{
        ... 略
```

このようなコメントが必要になったときは、その部分の
モジュール化を検討すべきである。