GameLib コーディング作法について

平山尚

2008 年 7 月 6 日

1 はじめに

この文書では、本書のサンプルコードの作法を紹介する。しかしそれだけにとどまらず、web や書籍をいくらか調査した結果どのような方法が多く使われていて、どういう理由でどんなスタイルを選んだのか、という語り方をする。単純に列挙して欲しい人には申し訳ないが、こうしなければ意図が伝わらないだろうと思ってそのようにした。

ただし、著者である平山はルールに関してさほど厳密性を求めない性分であり、これらのルールには例外がいくらでもある。

また、このルールは読者に強制するものではないし、これが最上だと主張するものでもない。おそらく一年後には私もルールを修正しているだろう。

2 スタイル、レイアウト

2.1 タブ幅

2 から 4 が良いとされている。タブキー 1 回がインデント 1 段に当たる方式、規定の幅だけ半角スペースを並べる方式、タブ幅を 8 に固定し、8 に達するまでは半角スペースで、8 に達したらタブ文字で置換する形式の三つが広く用いられている。

なお、google で「インデント　タブ　スペース　混在」で検索した結果上位 20 位までは全て混在を否定する意見であった。問題は emacs が標準でそのような混在を行う設定になっていることで、メンバのエディタが統一されていない状況下では emacs 使いの率によって結論が異なるケースが多いかもしれない。結局話し合いで少しづつ統一を進めてゆく他ないだろう。なお、タブとスペースのどちらを用いるかについては決定的な結論は見当たらない。タブ幅の変更に対して頑健であるためには行頭以外はタブを使うべきでないため、であれば全てスペースにすべきだという意見もある。

この文書ではインデント幅を 2 にしているが、2 を推奨しているわけではなく、単なるスペースの都合だ。私が選んだスタイルは、タブ文字を用いたインデントであり、VisualStudio の標準設定のまま 4 の幅として使っている。タブ文字であるから開くエディタによって見える幅は異なるが、それでレイアウトが崩れるような事は一切していない。スペースを使ったレイアウトの調整は一切行っていないからである。

2.2 空白

以下の場所には空白を入れる。

1. 2 項、3 項演算子の左右
2. if,while,for 等の予約語の右
3. (の右、) の左
4. , の右
5. [の右、] の左

以下に厳密に適用した例を示す。

```
grossRate[ census[ groupID ].gender ]
if ( int i = 0 ; i < N ; ++i ){
*index = 5;
int a = ( i > 5 ) ? 5 : i;
if ( ( a == b ) || ( c == d ) ){...
int a = ( ( 4 * 5 ) + ( 3 / 4 ) ) * 4;
```

連続する括弧や、比較的長い数式の場合など、厳密に適用するとかえって読みにくいと思われる場合もある。このあたりは比較的柔軟に対応しても問題が少ない部分であると言える。

私は本文中のサンプルコードで幅が足りない場合を除いて、ほぼ厳密に上記の規則を適用している。if,while,for の横に空白を入れるのは、関数との区別の意味合いもある。

2.3 ブロックの表現

以下のレイアウトを用いる。

```
if ( foo ){
  ++a;
}
```

コードコンプリートで推奨されない書き方として以下のようなものがある。

```
if ( foo ){
          ++a;
}
```

括弧の中の行頭位置が条件文の長さに依存している。他の行の構成要素の長さが変化した時にレイアウトを直さねばならないような方法は全て変更コストの面で劣り、用いるべきではない。

```
if ( foo )
{
  ++a;
}
```

のようなスタイルについてはコードコンプリートでは推奨されていないが、一般的に用いられており、ここでは敢えて否定はしないし、私も C#で書く時には Visual Studio が勝手にそのように整形してくれるのでそのままにしている。

2.4 複数行を揃える必要はない

「他の行の構成要素の長さが変化した時にレイアウトを直さねばならないような方法は全て変更コストの面で劣り、用いるべきではない。」という原則が常に適用される。

```
int aaa = 0;
int aa  = 0;
int a   = 0;
```

のような書き方は不適切であり、

```
int aaa = 0;
int aa = 0;
int a = 0;
```

と書けば良い。「美観」と「変更のコスト」が対立した場合には、変更コストが優先することが多い。関係する行数が大きくなればなおさらである。すでに述べたとおり、「スペースを使ったレイアウト調整は一切しない」というルールがある以上当然の帰結である。

2.5 複雑な条件文

可能な限り論理的なまとまりで改行し、さらには補助的な bool 変数などを用いて見やすくすると良い。

```
if ((('a'<=c) && (c<='z')) || (('0'<=c)&&
  (c<='9')) || (('A'<=c) && (c<='Z'))){
```

のような条件文は、

```
if (
  ((c >= 'a') && (c <= 'z')) ||
  (('0' <= c) && (c <= '9')) ||
  (('A' <= c) && (c <= 'Z'))){
```

のように意味の区切りごとに改行して不等号の向きはそろえ、できれば

```
bool isLower = (c >= 'a') && (c <= 'z');
bool isUpper = (c >= 'A') && (c <= 'Z');
bool isNumber = (c >= '0') && (c <= '9');
if (isLower || isBig || isNumber){
```

のように補助的な bool 変数を用いて整理する。ただし、「どこからが複雑か」はその場の状況や気分によるため、厳密な指針はない。

2.6 継続行の処理

行が長くなった場合の分割には一貫性を持たせる。

```
if ( ( a > b ) ||
  ( c > d ) ||
  ( e > f ) ){
```

```
if ( ( a > b )
  || ( c > d )
  || ( e > f ) ){
```

いずれも演算子の位置は後ろか前に統一されている。前者は演算子が行末に来ることで継続していることがわかりやすく、後者は演算子が縦にならんで見やすいといわれているが、どちらが良いとは言いがたい。重要なのは一貫性であり、一人の人の中では統一されているべきである。

私は前者、つまり演算子が行末にあるやり方を採用している。行の継続がわかりやすいことがより重要だという立場だからである。

2.7 複数行にまたがる関数コール

コードコンプリートでは以下のような形式が推奨されている。

```
foo(
  a,
  b,
  c
);
```

インデントを通常と同じだけ入れ、最後の括弧を別行に関数の先頭と同じインデントにすることで引数の範囲が明に示される。引数のインデントをfoo(の括弧にあわせる流儀は関数名が変わった時に弱いのと、インデントがタブの個数でなく文字の数になるのが面倒である。ただし、この方式は最後の引数のカンマを間違ってつけてしまいやすいのが欠点であるとはいえる。); を最後の引数の行に書いてしまうのもおそらく悪い流儀ではないだろう。

そのため、私の流儀は

```
foo(
  a,
  b,
  c );
```

のようなものになっている。

2.8 行数を無理に減らそうとしない

複雑な文を短く表記することは、複雑な内容を簡単に見せかけるという意味で偽りである。複雑な処理は複雑に見えるように書くべきだ。また、一行に複数の文を書くと流し読みの際に障害になる、コメントアウトが面倒、ブレークポイントを貼りにくいなどの弊害が生じる。一行に一文、かつ単純さを保つべきである。

```
while ( *t++ = *s++ );
```

のような文は複雑さを隠している悪い例であり、

```
do {
  ++t;
  ++s;
  *t = *s;
}while ( *t != '0' );
```

と素直に書けばよい。これであればバグも入りにくい。

私はよほどの場合を除いて、後置インクリメント（i++）は完全に追放している。また、インクリメントは独立した文とし、それを比較や代入に用いることは禁じている。

2.9 switch 文

以下のように整形する。

```
switch ( c ){
  case 'a':
    hoge();
    break;
  default:
    fuge();
    break;
}
```

case 文の行末にあわせて case ブロック行頭を調整するような書き方は変更コストが高いので避けるべきである。

case を switch と同じ列に書く書き方はインデントが減って楽だし、私も昔はそうしていたが、コードコンプリートの記述を読んでこちらに改めた。

2.10 変数宣言

一行に一つにした方が良い。これも流し読みしやすくするなどの効果がある。これによって、古い本に多く見られる

```
int *foo;
```

という変数名に*がつくスタイルでなくプログラミング言語 C++ などで採用されている。

```
int* foo;
```

というスタイルを用いても問題が発生しなくなる。

しかし、無論のこと例外はあって、ポインタ型以外ではカンマで区切って複数置くことも少なくない。ただしポインタ型を複数置くことだけはないようにしている。

2.11 関数定義

関数定義は以下のような形式を取るのが良いといわれている。引数リストが一行に入りきる場合は、

```
bool foo( int a, int b, int c ){
  ...
}

bool foo( int a, int b, int c )
{
  ...
}
```

引数リストが複数行に渡る場合には、

```
bool foo(
  int a,
  int b,
  int c ){

  ...
}

bool foo(
  int a,
  int b,
  int c )
{
  ...
}
```

括弧の位置については if 文などと一貫性のあるスタイルを取ることが望ましい。ただし、C 言語のバイブルである「プログラミング言語 C」(通称 K&R) においては if 文などでは { を同じ行に置きつつも関数定義では { を別に置くスタイルをとっており、これは広く使われている。

私は括弧は行末にくっつけて書き、引数を改行して書く場合にはインデントしないことにしている。つまり、以下のようなものだ。

```
bool foo(
int a,
int b,
int c ){
  ...
}
```

括弧の位置については if や for に合わせただけだが、引数のインデントは理由を説明する必要があるだろう。見難いことは重々承知しているが、インデントしてしまうと関数の中身と区別がつきにくくなり、これを嫌ったのである。

```
bool foo(
  int a,
  int b,
  int c ){
  a = 5;
  b = 3;
  c = 4;
}
```

というような状態よりは、

```
bool foo(
int a,
int b,
int c ){
```

```
  a = 5;
  b = 3;
  c = 4;
}
```

の方が私は見やすいと判断した。ただし、関数宣言で中身がない場合には、インデントしている。

```
class A{
   void foo(
      int a,
      int b,
      int c );
   void bar( int a );
```

このあたりの不統一は自分でも整理がついておらず、近いうちに全てインデントする方式に修正する可能性は高い。

```
bool foo(
  int a,
  int b,
  int c ){

  a = 5;
  b = 3;
  c = 4;
}
```

のように最初に空行を置けばそれほど問題もないからだ。

2.12 クラス定義のレイアウト

クラス内の宣言順は、

1. public
2. protected
3. private

の順番とし、public、protected、private の各部は

1. コンストラクタとデストラクタ
2. enum
3. 内部クラス
4. 定数
5. 関数
6. 変数
7. static 変数

の順が良いかと考える。コードコンプリートでは常にコンストラクタとデストラクタは private であっても先頭に配置されるとしているが、私は「ヘッダを使う人間にとって必要な情報は public だけであり、それが目立つ場所にあるのが理にかなう」という考え方を採る。また、typedef や friend 宣言等についてはあまり資料がないので確たることは言えない。それらの性質に応じて適した場所に配置すれば良いと考える。

また、そもそも private なメンバはヘッダ内にあまり大量に置かれるべきではないのだ、ということも思い起こす必要があろう。適切な手段で cpp 内部に隠蔽してしまうべきなのである。

3 命名

以下では命名について規則を並べるが、最も重要であることは、それが何であるのかが容易く理解されることである。変数の名前であれば単に適切な名前をつけるという程度の話になるが、変数名と型名のように、異なる種類のものに別のルールを設けることで確実に識別することを義務付けるような場合には、その種類の識別が重要である必要がある。取り立てて識別する必要がないものを規則まで作って識別するのは無駄であるということに注意されたい。また、一部の規則（メンバ変数の識別など）はコンパイラのエラーチェック機能の不足を補うという意味で、通常の命名規則よりも一段と重要である。

3.1 ファイル名

中に記述された主クラスの名前を大文字小文字まで等しくそのままファイル名とする。「クラス名.cpp」と「クラス名.h」が出来ることになるだろう。

ファイルの中にはクラスは一つだけ存在することが望ましいが、ある基底クラスから派生した小さな

クラスが多数存在する場合など、一つのファイルに複数のクラスを配置することは合理的と判断されうる。その場合は最も代表的なクラス（この場合であれば基底クラスであろう）の名前をファイル名とすれば良い。しかし、基本的には 1 ファイルに 1 クラスとなる。

なお、クラスを格納しないもの、例えば小さな便利関数を多数集めたようなファイルである場合は、それらを一つの名前空間に格納し、その名前空間の名前をファイル名とするのが良い。MathUtil 名前空間に属する多数の関数宣言を格納するのであれば、MathUtil.h という名前は合理的である。

その他例外的なものについては、習慣的な名前があれば用い、その他は適宜判断する。例えばプリコンパイルヘッダなどは標準的な名前がある (StdAfx.h) し、データテーブルのヘッダなどは適切な名前をつければよかろう。

できる限りヘッダにテーブルの類を記述せず、ロードできるデータにした方がコンパイル時間、メモリ消費、柔軟性などの観点から望ましいが、それはこの文書で言及すべき範囲ではない。

3.2 型名

大文字で始まり小文字が続き、単語の先頭は大文字とする。

class、struct、enum、typedef などこれらは全てプログラム上で型名として分類される物であり、統一的なルールに従う。

なお、enum には可能な限り型名をつけるべきで、それによって引数のチェックが可能になる。例えば

```
enum Type{
  TYPE_A,
  TYPE_B,
  TYPE_C,
  TYPE_MAX,
};
```

のように Type という型名がついていれば、

```
Type getType( );
void setType( Type type );
```

のように関数の引数にでき、int をそのまま扱う場合に起きうる範囲エラーなどを除くことができる。

なお、名前空間については現状あまり資料がなく確かなことはいえないが、スコープ演算子 (::) が使える点で class や struct と同じく型名の一種として扱う場合が多いようではある。(C#の標準など)。しかし、小文字始まりの変数名と同じ規則を用いるのも広く見られるし、(C++ 標準ライブラリ)、広く確立されたルールは見当たらない。

私は名前空間がクラスに近い性質を持つ点を重視してクラスと同じ命名規則を使うようにしているが、異論もあろう。

ちなみに、クラス名の先頭に C をつける記法が流行した時代があったが、現在はあまり見られない。ユーザ定義型の大半がクラスであり、識別するにしても非効率であるのと、そもそも識別する必要がないのがその理由ではなかろうか。

3.3 関数名

小文字で始まり小文字が続く。途中の単語の先頭は大文字。また、英語であり、後に示す例外を除いて、動詞 + 目的語の構成をとる。simulate()、processInput() などだ。関数名はその内容を正しく表現できる程度の長さは必要で、平均的には 9-15 文字程度になると言われている。

特に bool を返す場合には「is+ 形容詞」「is+ 過去分詞」「has+ 過去分詞」「過去形」などの形式が英語として読みやすく、また識別しやすい。isReady()、hasGone()、failed() などである。なお、is か has かはたまだただの過去形か、などの判断は我々日本人にとってはわかりにくいため、全て is としても問題にはならないだろう。我々にはその方が読みやすいとすら言える。英語の感覚をお持ちの方はうまく使い分けると英語圏の人々に優しいコードとなる。

なお、戻り値がある場合は「get+ 戻り値の内容」「calc+ 戻り値の内容」のように、戻り値が何であるのかを示す名前にするのが良いとされている。同様に引数を設定する場合には「set+ 引数の内容」

が用いられる。setMaterial、calcFunctionCurve、getName などがその例である。

逆に、戻り値があるのにその戻り値の内容が関数名で示されていないような関数名は良い名前ではない。「execute() は bool を返し true であれば成功」というようなものは良いとは言えない。この場合は、

```
void execute( bool* succeeded );
```

のように、引数で明に受け取るべきである。引数に書き込むのは可能であれば避けるべきで良い作法とは言いがたいが、戻り値には名前をつけられないわけで、まだマシだろう。

なお、execute や process のような動詞はそれが何をするのかが具体的でなく、もしもっと良い名前があるのであればそれを用いるべきである。

関数名に用いられる動詞が対になる場合には、英語として正しい対にすることで理解を助ける効果がある。一部を列挙すると、

1. add/remove
2. increment/decrement
3. open/close
4. begin/end
5. insert/delete
6. show/hide
7. create/destroy
8. lock/unlock
9. source/target
10. get/put
11. next/previous
12. up/down
13. get/set
14. old/new
15. start/stop
16. first/last
17. min/max
18. visible/invisible

3.3.1 get の省略について

単にメンバ変数を返すだけの get 系関数は、誤解の恐れがない限りにおいて get を省略した。getSize()、getLength() と書く代わりに、size()、length() と書くということである。名詞のみの関数名は他に存在し得ず、メンバ変数は後述のように m から始まるので名前がかぶることはない。

この省略の条件は以下のようになる。

- 戻り値として返す場合。引数に入れる場合は省略しない。
- 結果の関数名が明瞭である場合。name()、size() 等はいいが、intersection() となると微妙である。
- 結果の関数名が予約語になる場合は省略しない。getInt() は int() になるので省略できない。
- 結果の関数名が標準ライブラリとかぶる場合は省略しない。getString() は string() になるので省略しない。

なお、この省略はルールの一貫性をいささか損なってでも簡潔さを向上させる、という考えに基づいたものであり、一貫性を重視するならば採用すべきでない。

3.4 変数名

小文字で始めて小文字が続き、途中の単語の先頭は大文字。関数と同じルールだが、変数は全体として名詞になるようにするし、関数は常に後ろに () が付くため容易に区別できる。

i、c、renderWidth、progressBar のようになる。

使い捨ての変数を除いて、一文字の変数は避けるのが良いとされている。いくつかは慣習的に定められており、整数の i、j、k、m、n や文字の c、d、e などがある。

なお、変数名の長さはスコープの大きさに比例することを原則とすると良い。使い捨てのローカル変数であれば一文字でも許されるが、ローカル変数であっても数行に収まらないほどのスコープを持つのであればもう少し内容のある名前にすべきだ。メンバ変数は関数の全区間に渡って存在するため十分に長い必要がある。そして、グローバル変数は十分に

長い名前にして絶対に重複や誤使用が起こらぬようにしておく必要がある。ただし、私はコードの短さを平均よりも重視するため、多少スコープが大きくても平気で一文字変数を使う傾向がある。自分では問題ない範囲だと思っているが、他人の意見を聞いて回ったことはないので陰で苦々しく思われている可能性もないとは言えない。

また、メンバ変数やグローバル変数は規則を別に設けて区別をすべきである。これは、これらの変数と同じ名前のローカル変数を定義してしまうことが可能であり、その結果メンバ変数を隠してしまうことがあるからだ。変数名を隠すのは多くの本でよくない習慣とされており、これをコンパイラがチェックをしないバグであるとみなすならば、ユーザ側で規則を設けて対処するのが合理的である。メンバ変数の先頭に m_を、グローバル変数の先頭に g_をつける流派が有名だが、先のルールに従えば単語の先頭は大文字になるのだから、単に m、g をつけるだけでも識別は可能である。例えば、mMaterial、mShader、gDirect3dDevice、のようになる。メンバ変数の識別法として末尾 (あるいは先頭) に_をつけるスタイルも短くかつ目立たないので使われているが、_の使用は望ましくないという意見もある。先の命名規則に従うのであれば m と g が望ましかろう。なお、C++ の仕様書によれば_から始まる名前は禁止されているようなので、_は仮につけるとしても末尾になるだろう。

なお、同様の理由で関数の引数にも別のルールを設ける派もある。元の変数名の先頭が子音ならば a、元の変数名の先頭が母音ならば an を先頭につける派や、先頭に_をつける派などがある。m や g を前につけているのであれば a をつけるのが整合性を取る上では良いだろう。ただ、母音であれば an をつけるというのは英語的な都合であり、我々になじむかどうかは疑問ではある。なお、引数に規則を設けることについては、メンバ変数やグローバル変数と違って引数は定義がすぐ近くに存在しデバグが容易であることから、それほど重視はされていないようだ。

また、配列であれば a、ポインタであれば p、無符号 32bit 整数であれば ul といった接頭辞をつけるスタイルは広く使われているが、その合理性を説明した文書が見つからないので、ここでは取り上げない。.NET 以前の Windows プログラミングの本では盛んにそういった記法 (ハンガリアン記法) の優位性が喧伝されたこともあったが、現在では変数名に型情報を含める記法の優位性を主張する文書はあまり見られないようだ。型が変わったら変数名を変えねばならない、そもそも型の大半はユーザ定義クラスであり例外的ともいえる組み込み型を識別する意味を感じない、C++ などの言語では型チェックが厳しく型の間違いに由来するバグは発生しにくい、などの意見が多くみられる。

3.5 定数

全て大文字とし、単語境界に_を用いる。MIN_WIDTH、HEIGHT、TYPE_LINEAR のようになる。enum、static const は等しく定数とし、格別の区別はしないが、enum はその型名を先頭に付加することでどの enum の定数なのかを明に示す。

```
enum TextureFilter{
  TEXTURE_FILTER_LINEAR,
  TEXTURE_FILTER_CUBIC,
};

static const float CAMERA_NEAR = 1.f;
static const int LOCK_COUNT_MAX = 5;
```

先頭に k を付与する派もあるが、原典は見つけられなかった。#define のマクロによる定数定義は C++ では不要どころか有害であることから、それとの区別をするためという目的では接頭辞は不要になっていると考える。

なお、列挙に関してはコードコンプリートで使われている

```
enum TextureFilter{
  TextureFilter_Linear,
  TextureFilter_Cubic,
};
```

のようなルールもアリだとは思う。しかし私は C 言語時代の「定数は大文字ルール」をそのまま引きずっているのでどうにも慣れない。それに、C++ になってマクロが激減した結果、全てを大文字にする識別子がほとんどなくなってしまったため、定数や列挙にこれを割り当てるのは悪い選択ではないかなと思う。

3.6 インクルードガード

インクルードガードは良くファイル名の単語区切りを_にしたものが使われているが、C++ 言語においては_から始まる識別子や_が二つ並ぶ識別子を作ってはならないという掟がある。また、単にファイル名だけでは名前の重複が起こりうるため、危険である。そこで、「INCLUDED_モジュール名 or アプリ名_ファイル名」というものを提案してみる。

```
#ifndef INCLUDED_GAMELIB_SOUNDMANAGER_H
#ifndef INCLUDED_ROBOFIGHT_AI_PLAYER_H
```

INCLUDED を加えるのは、インクルードガード以外の何者でもないことを明に示し、偶然同じ名前のマクロが定義されて衝突することを避けるためである。その目的で昔から加えられてきた__は C++ では使用を禁止されているからだ。

また、SOUND_MANAGER のように単語境界で区切らないのは、Sound::Manager と SoundManager が同じになるのを防ぐためだ。インクルードガードは読みやすい文字列である必要はないので、これで問題はない。

```
#ifndef MANAGER_H
```

とだけ書いて他モジュールと重複するようなことを避けることであり、それだけでも十分に目的は達せられる。

4 その他

効率、可読性、安全性などの合理的な目的のために規則あるいは癖として身につけるべき項目をここに記す。

4.1 グローバル変数は避けよ

グローバル変数はカプセル化を妨げる。グローバルなアクセスを行いたい場合には変数をそのまま置くのではなく Singleton パターンを用いて安全性を高めるべきである。よって、先頭に g がつくようなグローバル変数そのものはほとんど皆無といってよい状態になるはずだ。

しかし、サンプル内には縦横無尽にグローバル変数が使われているし、私のシングルトンクラスは内部実体がグローバルで置かれているわけで、例外は多い。サンプルに関してはわかりやすさを重視した結果ということで納得して欲しいが、人によってはそれでも許容できないだろう。シングルトンの実体に関しては意見を聞きたい。全て名前なし名前空間に入れてあるので実害はないはずである。

4.2 コメント

コメントには三種類ある。逆に言えばコメントには三種類しかない。

1. コードでは表現できない情報
2. 意図を説明する
3. 概要を添える

コードでは表現できない情報とは、ある変数が何を表しているのかなどを名前だけでは表現しきれない場合や、処理の背景にある情報などである。意図とは、例えば一見不自然なコードであっても最適化の結果であると添えることで読む人に理由を説明するようなことを指す。「敢えてそうしている」の「敢えて」を説明する、と考えれば良かろう。概要とは、関数や一塊の処理のすぐ上に短くその処理が何をしているのかを書いておくことなどである。

コメントにすべきでないのは、コードで十分に説明されている情報や、メモなどであるが、メモに関しては常に否定されるとは断定し難い部分もある。しかしその場合でも、特定の文字列 (TODO、TOFIX など) で検索可能にするなどのルール作りは必要であろう。

なお、私のコードにおいては上記の三つからは若干外れるコメントもかなりある。教育用なのでコードを読めばわかるような内容も日本語で書いた方が

いい場合もあるし、本文中で説明するのに適さない項目などはコメントの形でかなりの長文が入っていたりもする。いずれも教育用途であるが故の例外と考えて欲しい。

4.3 単文の禁止

for,if,while の後の単文は禁止する。つまり、かならず {} でくくる。

```
for ( int i = 0 ; i < N ; ++i )
  ++a;
  ++b;
```

この手のコードが持つ問題を未然に防ぐのが目的である。最初にコードを書いた人間がよく理解した上で単文を用いていたとしても、保守の過程で他の人の手が入ることを考えれば安全を期しておいて損はない。

ベテランからは大変反発されるので敢えて私の考えを押し付けようとは思わないが、絶対に自分でこの書き方をすることはないだろう。

4.4 括弧の使い方

演算子の優先順位を知っていることが前提になるようなコードは書かず、全て括弧でくくる。

```
if ( ( a == b ) || ( c == d ) ){
x = ( 3 + 4 ) * ( 5 + 6 );
color = 0xff008080 | ( red<<16 );
(*(array.getIterator()))->someFunc();
```

などなど、例はいくらでもあるが、演算子が複数結合した式では必ず括弧で明に示すべきである。括弧の優先順位を全て完全に記憶している人は稀であり、標準的なスキルの人に合わせるべきだからである。

ただ、乗除算と加減算に関しては文字数が多い場所では括弧を省略していることもあり、厳密に守られているわけではない。

4.5 スコープは最短に

変数のスコープは可能な限り短くすべきである。何故なら、使っていない時間にその変数が存在すること自体が予期せぬ書き換えなどの問題の影響を受ける可能性を高めてしまうからである。new のオーバーヘッドを気にする時代ではすでにない。

ローカル変数定義は使う寸前で行うべきであり、クラスインスタンスも使う寸前で new し、使い終わったら速やかに delete すべきである。情報を保持する必要がないのであれば、再び使う予定があるとしてもその都度 new すれば良い。

なおローカル変数定義の場合にはそうすることで関連するコードが集中しやすくなり、読みやすくなる効果もある。

また、この原則からもグローバル変数が歓迎されない理由が理解できるだろう。

ただし、性能上の理由がある場合は別であり、これも厳密に守られているわけではない。

5 参考文献

1. コードコンプリート
2. プログラミング言語 C
3. Modern C++ Design
4. プログラミング言語 C++
5. google 先生