良いプログラムの書き方

2012-06-01

はじめに

この資料では, バグの発生を抑え, 後で読んでも理解できる, メンテナンス性の高いプログラムを作るための方法について説明する. 基本的には, C/C++ 以外でも役立つ事項である.

なお, この資料は, 両面印刷左綴じで読みやすいように作ってある.

1　良いコメントを書こう

人間は忘れる生き物だ．まだプログラミングに慣れいてないうちは，自分の書いたコードは完全に理解していて，忘れることなんてないと錯覚しがちである．

しかし経験論的に，数週間，数ヶ月前に自分で書いたコードは，他人が書いたコードであると思うことになるだろう．

1.1　外部仕様 — 使う人のための仕様

ライブラリのような他人の書いたプログラムを利用するためには，**どのように使うか**を知らなければならない．多くのライブラリではリファレンスマニュアルが整備されているが，唯一コンパイラが理解できるのはその関数やクラスの**プロトタイプ**である．

研究用の実験プログラムで仕様書を用意するのはなかなか現実的ではない．少なくとも，その関数の使い方を明確にするために，適切なコメントを付けておく必要がある．

例　`<cstring>` に含まれる関数プロトタイプ

```
char *strcat(char *dest, const char *src);
```

を考えよう．

この関数に対して，次のようなコメントは不適切である．

```
// 文字列を結合する
```

これは以下の点で良くない．

- 引数の意味が説明されていない．destination, source などと英単語の意味で推測させようとしている．
- 返り値が何か不明

関数のコメントには，すべての引数と返り値に関する説明が必要である．また，その引数の範囲や，範囲外の場合の処理を明確に述べなければならない．C のリファレンスでは，これは以下のような説明がされている．

```
// 文字列 src のコピーを文字列 dest の後に追加する
// dest の終端文字は src の最初の文字で上書きされる
// dest は結果を保存するのに十分な大きさの配列で無ければならない
// 返り値は dest と同じものを返す
```

無駄なく必要事項をコメントとして入れるには，**すべての引数を文に含むように**説明を構成し，その後さらに引数に関する詳細な事項を列挙する書き方を推奨する．

1.2　内部仕様 — 未来の自分への仕様

ある関数が実際にどのようなアルゴリズムで実装されているかが，内部仕様である．

内部仕様を明確にするためにも，適切なコメントが必要である．あまりに複雑なコメントが必要であるならば，より細かい関数に分割できないか考えるべきである．

コメントに対してコーディングする　コードを書いてから, コメントを付けるのは非常に面倒に感じることが多い. お勧めなのは, 先にアルゴリズムをコメントで書いてから, そのコメントに対してコードを付けていくことである. 例えば, 以下のようなコメントから始めて, 次第に細かく詳細を書いていく.

```
// 2 つの画像 I, J を読み込む
// 初期エネルギー E0 を計算する
// E が初期値 E0 から 0 に近づくまで反復計算をする
```

ダメなコメントの例　以下のようなコメントは, 意味が無いか, 曖昧である. 理由を考えよ.

```
(a) i = i + 2; // 2 増やす
(b) const double a = 0; // 角度
(c) fp = fopen(fn, "rb"); // バイナリモードでファイルを開く
```

以下にコメントを書くときに, 何を考えればいいかをいくつか挙げておく.

- なぜそのような計算を行うのか
- その変数の値はどういう規則 (単位や, 原点の位置) なのか
- その変数はどのような範囲の値を取るのか
- その変数が特殊な意味を持つのはどのような値か
- ループの終了条件は何を意味しているのか
- ループが終了することで何が求まるのか

たかしのかあちゃんからのコメント　J(’ー ‘) し

> たかしへ
> たかしの書いたソースコード読みました。コメントにメソッドで実現したいことが書いてあり大変読みやすいです。設計上の判断も書いてあってとてもよいです。「○○しようと思ったけど、デメリット××があったからやめた。」というコメントは修正するときに大変役立ちます。
>
> たかしへ
> たかしの書いたソースコード読みました。一応動いているみたいですね。空行が 2 行以上なのは、一度しか参照しないローカル変数にはどんな意味がありますか？変数の定義と使用箇所が離れすぎています。ループの中身を関数にすると読みやすくなります。次はもう少し丁寧に書いてみて下さい。
>
> http://togetter.com/li/299807 「最強の IT 系かあちゃんからたかしへのアドバイス」より

2 const の活用

const キーワードによって, 様々なものの定数性が保証される. これを const correctness という.

2.1 ポインタ引数

関数の仕様を決める際に, ポインタに const をつけるかどかは**常に考えなければならない**. const によってそのポインタの指す領域は変更されないことを明示できる.

例 例えば, C 形式の文字列の長さを調べる関数が

```
// 文字列 str の文字列長を返す. str の内容は変更されない.
size_t strlen(char* str);
```

という宣言だった場合, この関数を利用した後では, str の指す領域は変更されていないとは保証されない. たとえ, コメントで書かれていてもである. C 標準ライブラリの <cstring> の関数は, この意図を正しく表現するために, 以下のようになっている.

```
// 文字列 str の文字列長を返す.
size_t strlen(const char* str);
```

注意 ポインタや参照以外の引数に const を付加されていることは通常ない. プロトタイプとしてはそのような const は無視される. ただし, 関数の定義では, 通常のローカル変数同様, その内部で const 性を持つことを示すことができる.

注意 ポインタ引数の宣言には可能性として以下のものがある.

```
1.   int * x
2a.  const int * x
2b.  int const * x
3.   int * const x
4a.  const int * const x
4b.  int const * const x
```

重要なのは, * の前後で const の意味が異なる点である. したがって, a, b のペアは同じ意味である. 2. は x の指す領域への操作に対する const 性であり, x[0] = 1 のようなコードはコンパイラによってエラーとみなされる. それに対して, 3. は x そのものの定数性で, プログラムコードで x 自体 (つまりアドレス値) の変更ができないことを示している. 前項の説明から, ライブラリなどのヘッダファイルで使われるのは 1., 2. のみである.

2.2 メソッドの状態不変性

あるクラスにおいて, メソッドがそのオブジェクトの状態を変化させない場合, const によってその不変性を表す.[1]

[1] 状態を変更しない, つまり const のついたメソッドをインスペクタ (inspector), 状態を変更するメソッドをミューテータ (mutator) と呼ぶこともある.

例　以下の string のメソッドは, 1 つ目はそのサイズを調べるだけなので, 宣言の最後に const が付き, 2 つ目は, 内容を消去するので const はつかない.

```
size_t string::size() const;
void string::clear();
```

2.3　ローカル変数の定数性

関数の実装で使われるローカル変数が, 初期化以外で一切変更されないことを示すために const を使う.

例　以下のコードでは, 最初に定義された変数が関数の最後まで変化しないことを const によって表現している. もしこれがない場合は, 途中で変更している可能性があるため, デバッグの労力が増加する.

```
// image size
const int W = 640;
const int H = 480;

... // 長いコード

for(int y=0; y<H; ++y) {
  ...
```

```
// image size
int W = 640;
int H = 480;

... // 長いコード

for(int y=0; y<H; ++y) {
  ...
```

TBW

- #define よりも const 変数
-

3　アサーションプログラミング

関数に与えるべき引数の条件や得られる結果が常に正しいものである保証を与えるためのプログラミングテクニックにアサーションがある. これは, OpenCV などのライブラリでも一般に用いられている技法であり, どんな小さなプログラムでも, なるべく使うべきである.

3.1　事前条件

関数の引数として, 意味の無い値が与えられたとき, どのような振る舞いをするべきか. 大きく分けて以下のどれかになるだろう.

(a) 何かしらの適当な答えを返す. (例: 範囲外の点の描画は, 無視する.)

(b) エラーを通知する (例: 返り値によって, エラーなら非ゼロの値を返す.)

(c) アボートする

((d) 仕様にかかれている動作以外は保証しない)

一般に (a) は, 関数の仕様が分かり辛くなる傾向があるため, 行うべきではない. (b) はいつでも有効であるが, プログラムの使用者がエラーを無視してしまう可能性がある. またエラーの通知方法が煩雑になる.[2] (c) のアプローチは, 正しい使用方法以外では一切プログラムの実行を継続させないため, プログラムの使用者が常にその要求に答える必要がある. このような関数の引数に対する制限を**事前条件**または公開要件という.

C/C++ では, 事前条件を表明 (assert) するために, `<cassert>` に `assert()` マクロが提供されている. `assert()` マクロは, 与えた式が真であれば何もせず, 偽である場合は, 行番号などの情報を表示してアボート (異常終了) する. この様に, 条件を満たさない限り実行を継続できないようにする手法を**アサーションプログラミング**という. このような技法によって, 関数の仕様をコード化することができ, 単にコメントで書くのとは大きく異なる.

例　事前条件は, 以下のように関数の先頭で `assert` マクロの羅列で書かれる. なお, 文字列リテラルは常に真として評価されることを利用して, 事前条件の意味を書いておくと, アボート時に表示されるので便利である.

```
void setPixel(pixel* img, int W, int H, int x, int y, int v)
{
  assert(img != NULL && "img must be specified");
  assert(W > 0 && H > 0 && "image size must be positive");
  assert(0 <= x && x < W && "x coordinates must be in image range");
  assert(0 <= y && y < H && "y coordinates must be in image range");
  assert(0 <= v && v <= 255 && "pixel value must be in [0, 255]");

  // 以下関数のコード
  ...
}
```

注意　アサーションの条件式ではプログラムの**動作を変更するような記述**を書いてはならない.

[2] C++ では**例外機構**という言語サポートがあるが, この資料では解説しない.

3.2 事後条件

関数の使用者は関数の結果は常に正しいものが得られることを前提にプログラミングを行うことができるべきである. 関数が常に正しい結果を返すことを保証することを**事後条件**または公開契約という. この場合にも assert マクロを利用できる.

例

```
int countColoredPixel(const pixel* img, int W, int H)
{
  int n;
  ...
  // n に黒でないピクセルの個数が数え上げられている

  assert(0 <= n && n <= W*H);
}
```

3.3 不変表明

プログラムが複雑になってくると, 意図しない結果が連鎖的に誤りを発生してしまうことがある. あるタイミングで必ず成り立たなければならない条件を表明することを**不変表明**という.

3.4 アサーションとエラーの区別

アサーションは, 常に正しくあるべきことをコード化する手法である. これは, プログラムのエラー処理とは異なる.

例 ファイルを開く関数が, そのファイルを見つけられずに処理を続行できないというのは, 想定される事態であって, プログラムのバグではない. これはその後の処理で回避する. これを**エラー**という. それに対して与えるファイルのパスが意味のある文字列でない場合は, ロジックの誤りであり, 想定された事態ではない. これには常に正しいパス文字列を与えなければいけないというアサーションをすることができる.

3.5 アサーションの取扱い

一般に

$$\text{プログラムにバグがない} \Longrightarrow \text{すべてのアサーションが真である}$$

が成り立つ. したがって一度書き込んだアサーションは取り除く必要はない. 逆は成り立たないが, アサーションが増えれば増えるほど, バグの発生がおきにくいと考えられる. したがって, 関数の実装や関数の利用では, なるべく多くのテストを行い, どんな場合でもアサーションが真であるように努めることが目標となる.

アサーションは少なからず計算時間がかかるため, 十分テストが済んで, プログラムを実際に利用したり公開しようとする場合には, 消し去ってしまいたいと思うだろう. C/C++ では, NDEBUG マクロを事前に定義することで, assert マクロは完全に消滅するようになっている.

4 コマンド ライン引数を与えるプログラムの設計

実験プログラムでは, 複数の入力に対して何度も実行することがよくある. そのようなプログラムは, 入力情報をコマンド ライン引数として与えられるようにする. コマンド ライン引数を扱う標準的な方法はないが, いくつか有名なライブラリが存在する. ここではそのようなものを扱わずに, 手軽にコードがかける雛形を紹介する.

```
#include <iostream>

namespace
{
  enum Args
  {
    argExec = 0,
    argInput, // 入力画像のファイル名
    argOutput, // 出力ファイル名
    argEnd
  };

  printUsage()
  {
    clog
      << "Invert PGM image\n\n"
      << "Usage:\n"
      << "\tinvertPGM input output\n\n"
      << "\tinput: Input PGM image file\n"
      << "\toutput: Output PGM image file\n\n";
  }
}

int main(int ac, char** av)
{
  // check parameters
  if(ac != argEnd)
  {
    cerr << "! Too few or too much parameters\n\n";
    printUsage();
    return 1;
  }

  // av[argInput], av[argOutput] がそれぞれ与えられた引数文字列である
  ...
}
```

要点

- コマンドライン引数の番号を列挙体 `enum` で定義する．この時, 最初と最後は特別な変数名 (ここでは, `argExec` と `argEnd`) を与える．コマンドライン引数にはそれぞれわかりやすいコメントを書いておくと良い.
- コマンドライン引数がすべて与えられた場合は `ac == argEnd` となる.
- コマンドライン引数がすべて与えられなかった場合は, 関数の使い方 (usage) を表示し, 非ゼロの値を返してプログラムを正常終了する.

5 課題

1. 前回作成した関数に適切なコメントをつけなさい.
2. さらに事前条件を `assert` マクロで記述しなさい. 実際に条件を満たさない引数を与えて, どのような結果になるか確かめること.
3. 可能な限り `const` を利用したコードにしなさい.