Better C としての C++ 入門

2012-06-01

はじめに

研究のための実験プログラムを作る上で, C++ が使えるようになるための導入的解説を行う. ただし, C プログラミング経験者が C++ を覚えるために最低限必要な事項を説明するだけであって, この資料で C++ のすべてが理解できるわけではない. 特にオブジェクト指向プログラミングを本格的に行うための手法 (クラスを自作するなど) やテンプレートを使った高度な手法には触れない. あくまでも, C++ を自習する際の効率をよくするために行うものである.

というわけで, 各自**本を読むなどして**, **必ず自習をすること**. 特に, 卒業研究に着手してからでは遅いので, 夏休み明けまでの時間がある時に練習をしたほうがよい.

注意　今後の演習でリファレンスとして活用するので, チュートリアルが終わるまでは**常に携帯**すること. なお, 以下のサイトはライブラリのリファレンスとして信頼できるので, ブックマークなどしていつでも見れるようにしておくとよい.

- C/C++ リファレンス `http://www.cppll.jp/cppreference/`
- C++ reference (英語) `http://en.cppreference.com/w/cpp`

資料中のコードについて　この資料では, 左右のコードを対比している. 特に断りがないかぎり, 左のコードの方が推奨される書き方である. すべてのコードは, C++ として正しく動作するものである.

1　C++ と C の文法的な違い

C++ (C 互換)

```
#include <cstdio>

int main()
{


  std::printf("Hello, World.\n");

  // sum of squares
  int x = 0;
  for(int i=0; i<10; ++i)
  {
    x += i*i;
  }

}
```

C (C++ でもコンパイル可能)

```
#include <stdio.h>

int main(void)
{
  int i, x = 0;

  printf("Hello, World.\n");

  /* sum of squares */
  for(i=0; i<10; ++i)
  {
    x += i*i;
  }

  return 0;
}
```

要点

- C のソースコードは基本的に C++ のソースとしてもコンパイル可能.
- C の標準ライブラリのヘッダファイルは `c` を頭に追加し, `.h` を除いた新しいヘッダファイルを使用する.
- 引数を取らない関数を示す `void` は省略できる.[1]
- 新形式の C ライブラリヘッダおよび C++ のすべての標準関数, グローバル変数, および型は `std` 名前空間 (namespace) に属し, スコープ解決演算子 `::` を使用して明示しなければならない.
- C++ では C 形式の `/* */` によるコメントに加えて, `//` による行末までのコメントが使える.
- C では変数宣言がブロックの先頭でしか行えなかったが, C++ では任意の場所におくことができる. 変数宣言はなるべく最初に使用する直前におくのがよい.
- 制御文の条件内で変数宣言ができる. この場合, ブロックを抜けた時点でこの変数は消滅する.
- `main` 関数の最後の `return 0` は省略してもよい.

注意　C++ ソースファイルの標準的な拡張子は, `.cpp`, `.cc`, `.C` である. また, ヘッダファイルでは `.h` または `.hpp` となっていることが多い.

[1] C 言語でプロトタイプ宣言がない関数の引数が空の場合は異なる意味に解釈されることに注意. またそのような記述は推奨されない.

2 C++ のストリームと C での標準入出力の比較

I/O ストリーム

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string s;
  int i;

  cin >> s >> i;
  cout << "string: " << s << "; "
       << "value: " << i << "\n";
  clog << "done.\n\n";
}
```

C 互換 I/O 関数

```
#include <cstdio>


using namespace std;

int main()
{
  char s[100];
  int i;

  scanf("%s %d", s, &i);
  printf("string: %s; value: %d\n",
                                s, i);
  fprintf(stderr, "done.\n\n");
}
```

要点

- C++ では <iostream> が提供する入出力が使用できる．これはストリームと呼ばれる．
- C++ では文字列の取扱いを簡単に行える string 型が <string> で提供されている．
- 名前空間は using namespace ディレクティブを書くことで，以降，曖昧でない限り指定を省略することができる．この例では，cin, cout, clog, string, scanf, printf, fprintf で省略されている．
- string 型の変数は通常の変数と同じように宣言し，char 型配列と同じように使用できる．
- 配列は予め指定された大きさを持ち，変更することはできないが，string は自動的にその大きさが増減される．
- 標準入力から値を取り出すには，cin 変数を使う．これは iostream で定義されているグローバル変数である．任意の変数を >> 演算子で結合する．[2] scanf の様にアドレスを渡す必要はない．[3]
- 標準出力は cout 変数に対して，<< 演算子を使用する．
- 標準エラー出力は cerr または clog を使う．この二つの違いは，バッファリングの有無で，前者はバッファリングされず即座に出力される．

注意 例で用いている scanf はエラー処理が難しいため通常は使われず，sscanf を使う．また，cin を用いた処理では，通常はさらにエラー処理が必要である．

注意 標準入出力で C++ のストリームと C の標準入出力関数を混用することは基本的にできない．必ずどちらを使うか統一すること．なお，混用するためには同期を取る特別な処理が必要である．

[2] これは C++ でもビットシフト演算子であるが，cin に対して使用する場合の定義が変更されている．この機能を演算子オーバーロードという．
[3] これは C++ の参照 (refenrece) という機能によって実現されている．

3 ストリームによるファイルの入出力

I/O ストリーム

```
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
using namespace std;

int main()
{
  const double pi = 3.1415926535;
  const int n = 18; // # of division
  const string fn = "sin.txt";

  {
    ofstream ofs(fn.c_str());
    ofs << "# sine curve\n";
    for(int i=0; i<n; ++i)
    {
      const double a = i*(360.0/n);
      ofs << a << " "
        << sin(a*(pi/180.0)) << "\n";
    }
  }

  cout << "Output " << fn << "\n";
}
```

C 互換 I/O 関数

```
#include <cstdio>
#include <string>
#include <cmath>

using namespace std;

int main()
{
  const double pi = 3.1415926535;
  const int n = 18; // # of division
  const string fn = "sin.txt";


  FILE* fp = fopen(fn.c_str(), "w");
  fprintf(fp, "# sine curve\n");
  for(int i=0; i<n; ++i)
  {
    const double a = i*(360.0/n);
    fprintf(fp, "%f %f\n",
              a, sin(a*(pi/180.0)));
  }
  fclose(fp);

  printf("Output %s\n", fn.c_str());
}
```

要点

- ファイルの入出力は <fstream> が提供する, ofstream と ifstream を使用する.[4]
- 変数の宣言に () で引数を与える構文で, **コンストラクタ**と呼ばれる特別な関数によって初期化が行われる. ofstream のコンストラクタは指定されたファイルを出力モードで開く.
- ofstream は指定したファイルに対して出力を行う. 基本的な使用方法は cout と同じである.[5]
- ofstream などで開いたストリームはその変数の消滅時に自動的に閉じられる. ここでは, {} でブロックを明示的に作って, 変数の生存期間を制限している.
- 従来の C 標準ライブラリ関数や, 多くの C++ のクラスは, string 型の文字列に対応していない. C スタイルの文字列 char* に string を渡すためには, string::c_str() を使う.[6] この様に変数に対して呼び出される関数を**メソッド**という.[7]

[4] これらは fstream 型に入出力のフラグを指定したものと同等である.
[5] これらは, 同じ ostream から継承されている.
[6] c_str() が返すポインタは, その時限り有効なので, 他でも使う必要がある場合は都度このメソッドを使うか, char 配列にコピーをする.
[7] メソッドは通常所属するクラスなどの型を明示して string::c_str() の様に表記される.

4 メモリの動的確保とバイナリ出力

C++ 的な書き方

```
#include <fstream>

using namespace std;
typedef unsigned char pixel;

int main()
{
  const int W = 640, H = 480;
  pixel* G = new pixel[W*H];

  {
    ofstream ofs(
         "pattern.pgm", ios::binary);

    // PGM header
    ofs << "P5\n"
      << W << " " << H << "\n255\n";
    // pixel values
    ofs.write(
      reinterpret_cast<const char*>(G),
      W*H);
  }

  delete[] G;
}
```

C 互換 ファイル出力関数

```
#include <cstdio>
#include <cstdlib>
using namespace std;
typedef unsigned char pixel;

int main()
{
  const int W = 640, H = 480;
  pixel* G
    = static_cast<pixel*>(malloc(W*H));

  FILE* fp
       = fopen("pattern.pgm", "wb");

  // PGM header
  fprintf(fp,
          "P5\n%d %d\n255\n", W, H);

  // pixel values
  fwrite(G, W*H, 1, fp);

  fclose(fp);

  free(G);
}
```

要点

- メモリの動的確保は, new 演算子で行うことができる. new は指定した型の領域へのポインタを返す. それに対して, malloc は常に void* を返すが, C++ では C と違い void* からの暗黙の型変換は行われないので, 明示的なキャストが必要である.[8] どんなキャストでも行える C スタイルのキャストに対して, static_cast<> は単純な型の変換にのみ使用できる.
- バイナリ出力をする場合は第 2 引数に std::ios::binary を指定する.[9]
- ostream::write は書き込み先の領域への char* 型ポインタとバイトサイズを指定する.
- reinterpret_cast<> は C++ スタイルのキャストで, 対象の解釈を変更する. この場合は, write の引数の型に合わせるためにポインタの読み替えをしている.[10] fwrite の第 1 引数は void* なので, 明示的なキャストは必要ない.
- new で動的確保した領域は delete で解放する. 配列領域の場合は, delete[] でなければならない.

[8] void* への変換は暗黙的に行われる.

[9] std::ios は std::ios_base から継承されていて, 文献によってはこちらを指定していることもある. どれでも特に問題は無いらしい.

[10] なお, reinterpret_cast<>はポインタ以外の型とも変換できてしまうため, より安全には static_cast<T*>(static_cast<void*>(x)) という void* を経由した 2 段階のキャストをすればいいが, 煩雑である.

5　静的関数, 静的グローバル変数

グローバル変数や関数は, デフォルトで外部に公開される. つまり, 他のモジュールから (意図してかしてないかに関わらず) アクセスできる. モジュール内でのみ使う関数や変数は隠蔽をしなければならない. C では `static` をつけることで達成されるが, C++ では**無名名前空間** (unnamed namespace) を使うことが推奨される.

C++ 的な書き方

```
#include <iostream>

namespace {
  const double PI = 3.1416;

  void savePGM(const pixel* img)
  {
    ...
  }
} // end of unnamed namespace

int main()
{
  ...
}
```

C 互換の書き方

```
#include <iostream>


static const double PI = 3.1416;

static void savePGM(const pixel* img)
{
  ...
}


int main()
{
  ...
}
```

要点

- 外部に公開されないグローバル変数や関数は, 無名名前空間の中に書くことが推奨される. 無名名前空間は `namespace` に続いて何も名前を指定しないブロックによって定義できる.
- 同一のモジュール内で複数の無名名前空間を定義した場合は, すべて同じ名前空間を共有する. 異なるモジュールで定義された無名名前空間はそれぞれ互いに異なる名前空間となり, 他のモジュールからは参照できない.
- 無名名前空間の内容が長くなると, 閉じ括弧を誤って消去してしまったり, どれが閉じ括弧かが分かり辛くなるので, コメントで補足しておくと良い.

注意　名前空間を明示的に指定していない部分はグローバル名前空間という. `main` 関数は常にグローバル名前空間に置かなければならない.

6 ポインタと参照

参照は, ポインタを隠蔽するための新しい文法である. 多くのライブラリでポインタの代わりに使われている. 配列のようなポインタ演算が必要な処理は行えないので, 必ずしもポインタをすべて参照に置き換えられるわけではない.

参照

```
#include <iostream>
using namespace std;

namespace {
  void Swap(int& a1, int& a2)
  {

    const int old_a1 = a1;
    a1 = a2;
    a2 = old_a1;
  }
}

int main()
{
  int x = 100, y = 50;
  cout << x << " " << y << "\n";
  Swap(x, y);
  cout << x << " " << y << "\n";
}
```

ポインタ

```
#include <iostream>
using namespace std;

namespace {
  void Swap(int* a1, int* a2)
  {
    // a1, a2 が NULL だと例外が発生[11]
    const int old_a1 = *a1;
    *a1 = *a2;
    *a2 = old_a1;
  }
}

int main()
{
  int x = 100, y = 50;
  cout << x << " " << y << "\n";
  Swap(&x, &y);
  cout << x << " " << y << "\n";
}
```

要点

- 変数や仮引数の宣言に `&` をつけたものを**参照** (reference) という.
- 参照はポインタのようにアドレスを持つのでなく, それ自体が参照元の変数と同一視される. 従って, 呼び出し元の引数が直接変更される.
- ポインタには ヌルポインタという特殊な値が指定できるが, 参照は常に何かしらの変数による実体を指定しなければならない. 従って, **ヌルポインタチェックが必要ない**.
- 参照の引数には変数をそのまま渡す.[12]

以下は構造体 `struct Point { int x, y; };` を参照またはポインタで操作する方法をまとめたものである.

	参照	ポインタ
変数の型	`Point& r`	`Point* p`
メンバ変数のアクセス	`r.x`	`p->x` (`(*p).x` と書くことと等価)
変数のアドレス	`&r`	`p`

注意 `<algorithm>` (C++11 以降は `<utility>`) に任意の型に対応した `std::swap()` が用意されている. これは参照で定義されている.

[11] Null pointer exception. 俗に言うぬるぽである.
[12] 旧来の C ではこの様な引数は常にコピー渡しであったが, C++ では関数の呼び出しを見ただけでは, 変数が変更されるかどうかは一概に判断できない.

7 課題

基本

1. §3 のプログラムをコンパイル, 実行しなさい. また, 出力されたファイルを gnuplot でプロットしてみること.

2. §4 のプログラムをコンパイル, 実行しなさい.

3. PGM 書き出し部分を関数として独立させなさい. プロトタイプは

   ```
   // サイズ WxH のグレイスケール画像 img をファイル名 fn に PGM 画像として保存
   void savePGM(const pixel* img, int W, int H, const char* fn);
   ```

   とする. §5 の作法にしたがって, 外部から隠蔽すること.

4. 以下の関数を追加せよ.

   - グレイスケール画像全体を指定した値 v で塗りつぶす関数

     ```
     void fill(pixel* img, int W, int H, int v);
     ```

   - グレイスケール画像の指定した座標 x, y での値を設定及び取得する関数

     ```
     void setPixel(pixel* img, int W, int H, int x, int y, int v);
     pixel getPixel(const pixel* img, int W, int H, int x, int y);
     ```

   - グレイスケール画像を上下に反転させる関数および左右に反転させる関数

     ```
     void flipV(pixel* img, int W, int H);
     void flipH(pixel* img, int W, int H);
     ```

   - グレイスケール画像の画素値を反転させる関数 (画素値 x を $255 - x$ で置き換える)

     ```
     void invert(pixel* img, int W, int H);
     ```

発展

1. 上記のコードでは, 画像を扱う際に画素値の入ったバッファとそのサイズを別々に扱っている. これを一つにまとめるために, 以下のような構造体を導入する.

   ```
   struct PGM
   {
     pixel* data;
     int W, H;
   };
   ```

   「基本」で作成したコードをこの構造体を用いたものに書き換えなさい. なお, 構造体によって定義される型は, C 言語では struct PGM というキーワードとタグの組で書かれたが, C++ では PGM が型名として定義される.

2. 指定した 2 点間の直線を描画する関数

```
void drawLine(PGM& img, int x1, int y1, int x2, int y2, int v);
```

を追加しなさい. 実装は様々考えられるが, 特にブレゼンハムの直線アルゴリズム (Bresenham's line algorithm) が高速描画手法として知られている.

3. 上記の関数は参照渡しだが, 以下のような通常の引数およびポインタ渡しの場合はどうなるか考察しなさい.

```
void drawLine(PGM* img, int x1, int y1, int x2, int y2, int v);
void drawLine(PGM img, int x1, int y1, int x2, int y2, int v);
```

8 GCC で C++ プログラムをコンパイルする方法

GCC は, 拡張子から自動的にソースの種類を判断するため, gcc コマンドでもコンパイルすることが可能だが, 通常は標準ライブラリの指定が必要なのでリンクエラーになる. g++ コマンドはそのようなライブラリの指定をデフォルトで行うので, これを使えば良い.

出力ファイル名の指定 GCC はデフォルトで a.out という名前のファイルを生成する. -o でその名前を変更できる. なお, Linux では通常実行ファイルに拡張子はつけない.

```
$ g++ hogehoge.cpp -o hogehoge
$ ./hogehoge                    # 実行
```

<cmath> のライブラリ指定 g++ コマンドでは数学ライブラリの指定 -lm は自動的に与えられるので, 特に指定する必要はない.

警告の表示 警告を有効にすることで, さまざまなコーディング上の問題点を指摘してくれる. バグを未然に防ぐために**常に指定し**, 警告が一つもでないようにコードを保つことが望ましい. -Wall はほとんどすべての警告を, -Wextra はそれ以外のさらに慎重な警告を出力する.

```
$ g++ -Wall -Wextra hogehoge.cpp -o hogehoge
```

最適化 最適化を指定すると, 通常はプログラムの実効速度が向上する. その代わり, コンパイル時間が増加したり, 実行ファイルサイズが増加したりする. また, 最適化によって, バグが現れることもまれにある. デフォルトでは -O1 が指定されていて, 多くの場合は -O2 で十分な最適化がされる. 逆に -O0 によって完全に最適化を無効にすることもできる.

```
$ g++ -O2 -Wall -Wextra hogehoge.cpp -o hogehoge
```

マクロの指定 NDEBUG マクロのように, 状況に応じて指定したいマクロは -D オプションで与える. 間に空白を入れてもいれなくても構わない.

```
$ g++ -DNDEBUG -Wall -Wextra hogehoge.cpp -o hogehoge
```

これはプログラムの先頭で #define NDEBUG と書くことと等価である.

互換性のない機能の禁止 GCC は, いくつか仕様にはない機能を使えるようにしている.[13] これは -ansi と -pedantic によって禁止できる. ただしこれで完全に仕様準拠したものになるわけではないので注意.

[13] 例えば, C 言語では // によるコメントは仕様上使えないが, 警告もエラーもなく使える.