C 言語経験者のための C++ 入門

金森 由博
kanamori@cs.tsukuba.ac.jp

2011/12/2 第1.1版
2011/7/19 第1版

内容

- C++ を学び始めに引っかかりそうなところを概説

- この資料の目標は、C++ ソースの解読の補助程度
  - 書けるようになるには、既存のソースを真似るか本を参照

- 以下はこの資料ではあまり詳しく説明しない
  - 継承
  - テンプレート

  …このあたりこそ重要で奥が深い

コンパイル関係

- 拡張子は .c ではなく .cpp などにする
  - Visual Studio は拡張子でソースの種類を判定するので拡張子が .c で中身が C++ だとエラーになる
  - ヘッダファイルは .h でもよいが .hpp なども見かける
- GNU コンパイラを使う場合 gcc ではなく g++ を使う
  - 例：gcc hoge.c → g++ hoge.cpp
- C 言語由来のヘッダファイルには C++ 用のを使う
  - C 言語のも使えるが、C++ のものが推奨される
  - 頭に “c” をつけて末尾の “.h” を取り除いたファイル名
    例：stdio.h → cstdio / stdlib.h → cstdlib /
    math.h → cmath / string.h → cstring

コメント

- C 言語の /*　*/ というコメントも使えるが
  C++ では // 以降がコメントとして扱われる
- /*　*/ ではネスト不可だったが // なら OK

C 言語スタイル

```
/* This is a comment. */

/*    ※これはエラーになる
      /* Nested comment. */
  */
```

C ++ スタイル

```
// This is a comment.

//
//    // Nested comment.
//
```

- Visual Studio でのショートカット（※覚えて使うこと）
  - Ctrl-K + Ctrl-C ：選択した（複数の）行をコメントアウト
  - Ctrl-K + Ctrl-U ：選択した（複数の）行のコメントを解除

変数や関数などの宣言 (1/2)

- 変数は好きなところで宣言できる
  - C 言語のようにブロックの頭で宣言する必要はない
  - 使う必要のあるところでだけ宣言・定義すべき
    - 変数の有効な範囲（スコープ）が短くなる
    - コードが追いやすくなる
    - 使わない変数を無駄に宣言することがなくなる

C 言語スタイル

```
{
    int i, a, b;

    for (i = 0; i<N; i++) ....

    a = ....

    /* 変数 b を使っていない */
}
```

C ++ スタイル

```
{
    for (int i = 0; i<N; i++) ....

    int a = ....
}
```

変数や関数などの宣言 (2/2)

- 引数が void の関数の引数は省略できる
  - C 言語では func(void) と書いていたのを
    C++ では単に func() と書いて宣言 / 定義できる
- 構造体 (struct) は typedef しなくてよい
  - 参考: C++ では struct は class の一種として扱われる
    (メンバ変数/関数がすべて public であるような class)

C 言語スタイル

```
struct Hoge { … }; ←定義
struct Hoge a;  ←変数宣言

または

typedef struct { … } Hoge;
Hoge a;  ←変数宣言
```

C ++ スタイル

```
struct Hoge { … }; ←定義
Hoge a;  ←変数宣言
```

const, inline キーワード

- **const**: 定数や、クラス（後述）のうちでメンバ変数の値を変更しないメンバ関数（後述）につけるキーワード
  - ソースを追いやすくなる、コンパイラの最適化が期待できる
  - #define マクロによる定数ではなく const 変数を推奨

- **inline**: 関数につけるキーワード
  - 関数呼び出しのオーバーヘッドを避けるために、行数の少ない関数をソースに埋め込むようコンパイラに指示
  - クラス（後述）で定義する場合、ヘッダファイルに記述
  - 実際に埋め込むかどうかはコンパイラが判断

const, inline の使用例

- const … 値が変わらない、不変
  - **const** int N = 100;　　// 定数を宣言
  - class Hoge {
    …
    void hello() **const**;　　// メンバ変数の値を変えない関数
    };

- inline
  - **inline** float frand()　　// 主に処理の少ない関数に使う
    {
    return rand()/(float)RAND_MAX;
    }

参照 (reference)

- ポインタ同様、間接的に変数にアクセスできる
- 参照先の変数と全く同様に操作できる（≒別名定義）

ポインタ

```
int a = 3;
int *p = &a;  // ポインタ

*p += 1;
```

参照

```
int a = 3;
int &r = a;  // 参照

r += 1;
```

- ただし参照先は一度代入したら変更できない
- 実際の使い方
  - データサイズの大きな構造体などを、
    値をコピーせずに関数の引数として与えたいとき
    → void func(**const Hoge &hoge**)
    （※ただしこの関数内で hoge の値は変更しない前提）

名前空間 (namespace)

- 大規模開発の際に、変数名や関数名が衝突するのを避けるための仕組み
  - 名前空間が異なると別々の変数/関数として扱われる

```
namespace Hoge {
  int a;
  void hello();
}

namespace Geho {
  int a;
  void hello();
}
```

```
Hoge::a = 1;
Geho::a = 2;        別扱い
Hoge::hello();
Geho::hello();      別扱い
```

- “using namespace Hoge” と書くと “Hoge::” を省略可
  - ただしヘッダファイルで “using namespace …” は厳禁
    （#include のたび “using …” してしまい名前が衝突するかも）

関数のオーバーロード

- 引数が違うだけで処理は同じ、という関数を同じ関数名で宣言/定義できる

```
int hoge(int a);
int hoge(int a, int b);

void geho(int n);
void geho(double d);
```

- ただしむやみに同じ名前の関数を作るとバグの温床になりやすいので避ける

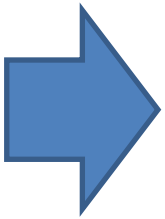
演算子のオーバーロード

- 既存の二項演算子 (=, +, -, *, / など) を使って別の演算や、ユーザ定義型の処理を書ける

例：3次元ベクトルの足し算

```
struct Vec3 {
    float x, y, z;

    Vec3 operator+(const Vec3 &v)
    {
        Vec3 tmp;
        tmp.x = x + v.x;
        tmp.y = y + v.y;
        tmp.z = z + v.z;
        return tmp;
     }
};
```



```
Vec3 a, b, c;

// c に a と b の和が代入される
c =  a + b;
```

ストリーム入出力

- 変数の型を意識せずに使える入出力機構
  - C 言語の printf や scanf も使える、そちらの方が高速
- #include <iostream> して使えるもの (std 名前空間)
  - cout: 標準出力、cerr: 標準エラー出力、cin: 標準入力
  - endl: 改行(と内部バッファのフラッシュ)

標準出力

```
std::cout << “Hello, World!” << std::endl;
```

標準エラー出力

```
int a = 1;
std::cerr << “a = “ << a << std::endl;
```

```
int b;
std::cin >> b;
```

標準入力

- ファイル入出力 (fstream)、文字列入出力 (stringstream) などもある

メモリの確保・解放 (new/delete)

- new/delete 演算子：malloc のようなキャストは不要
  - delete で配列を解放するときは [] が要ることに注意

C 言語スタイル

```
// メモリの確保
Vec3 *p = (Vec3 *)malloc( sizeof(Vec3) );
....
free(p);   // メモリの解放

// 配列のメモリの確保
int *a = (int *)malloc( 3 * sizeof(int) );
....
free(a);   // メモリの解放
```

C++ スタイル

```
// メモリの確保
Vec3 *p = new Vec3();
....
delete p;   // メモリの解放

// 配列のメモリの確保
int *a = new int[ 3 ];
....
delete [] a;   // メモリの解放
```

- new したら delete, malloc したら free, 混ぜるな危険
  - new も delete もオーバーロードが可能で、メモリ確保の際に内部的処理が通常と異なる場合があるため

クラス (class)

- 構造体 (struct) を拡張して、メンバ変数だけでなく
  - メンバ関数：そのクラスならではの処理を書いた関数
  - 公開/非公開 (public, protected, private) の属性
  - 基本となるクラスを拡張したクラス（継承）

  などを記述できる

- 「オブジェクト指向プログラミング」(OOP) で重要
  - 大規模開発向き
  - 設計の定石あり (cf. *デザインパターン*)
  - 内部設計をできるだけ隠蔽することで、モジュール間のつながりが疎になり、仕様変更に対応しやすくなる

参考：C 言語で OOP をやるなら

外部から変数にアクセス可能

仕様が変わったら、この変数を直接参照している部分は全部書き直し！？

```
typedef struct {
    int a;
} Hoge;

void Hoge_Init(Hoge *h)
{
    h->a  =  0;
}

void Hoge_Set_Value(Hoge *h, int a)
{
     h->a = a;
}

int Hoge_Get_Value(Hoge *h)
{
    return h->a;
}
```

「Hoge_Set/Get_Value() を介してのみアクセスする」と約束すれば修正は最小限で済むが、紳士協定なのでその約束を守るかどうかはプログラマ次第

構造体を初期化するために明示的に関数呼び出しが必要

関数名を工夫して Hoge に関連することをアピール

プログラマが自分で対象のインスタンスを引数で指定

使用例 (※次頁と比較されたい)

```
Hoge hoge;

Hoge_Init(&hoge);
Hoge_Set_Value(&hoge, 1);
int b = Hoge_Get_Value(&hoge);
int c = hoge.a;     // 変数に直接アクセス
```

※ Linux カーネルはこのような C 言語による OOP 的なコードになっている

クラスの例

クラスの定義

```
class Hoge {   // sturct の定義と類似
public:     // これ以降は外部に公開
    Hoge () {}     // コンストラクタ
    ~Hoge()  {}    // デストラクタ

    void setValue(int _a) {
       a = _a;
    }

    int getValue() const
    {
        return a;
     }

private:    // これ以降は外部に非公開
     int a;   // 外部から参照するとエラー
};  // 最後に ";" が必要 … struct と同様
```

```
// 関数の定義は次のようにして
//  class Hoge {  … }; の外側で書いてもよい

Hoge::Hoge() {}     // コンストラクタ
~Hoge::Hoge() {}   // デストラクタ

void Hoge::setValue(int _a) { a = _a; }

int Hoge::getValue() const { return a; }
```

クラスの使用例 (※前頁と比較されたい)

```
Hoge hoge;   // Hoge クラスのインスタンス
                // 初期化は自動的に行われる
hoge.setValue( 1 );
int b = hoge.getValue();
int c = hoge.a;     // コンパイル時にエラー
```

コンストラクタ (constructor)

- クラスのインスタンスが生成されるときの
  メモリの割り当てや変数の初期化の方法を定義

```
class Geho {
public:
    // デフォルトコンストラクタ
    Geho() : a(0), h(1) {}

    Geho(int _a, int _b)
    : a(_a), h(_b) {}

    // コピーコンストラクタ
    Geho(const Geho &g)
    : a(g.a), h(g.h) {}

private:
    int a;  Hoge h;
};
```

初期化リスト

メンバ変数の初期化方法を “:” の後に記述

省略すると、各メンバ変数のデフォルト
コンストラクタが自動的に呼び出される

Geho() { a = 0; h = Hoge(1); } という風にも
書けるが、その場合は各メンバ変数の
コンストラクタが 2 回呼ばれることになり無駄

コピーコンストラクタ

代入 = によって値がコピーされるときに
呼ばれるコンストラクタ

デストラクタ (destructor)

- クラスのインスタンスがメモリから解放されるときに呼ばれ、メモリの解放の方法を定義

```
class Hage {
public:
   // コンストラクタ
   Hage()
    : p(0) {}   // ポインタを 0 で初期化

   // デストラクタ
   ~Hage()
   {  // メモリ割り当てられていれば解放
      if ( p ) delete p;
    }

private:
   int *p;  // int 型のポインタ
};
```

```
{
   Hage h;

   /*  ここで何か処理を行う  */

}  // このブロックが終わるときに
   // デストラクタが呼ばれる
```

```
{
   Hage *h = new Hage();

   /*  ここで何か処理を行う  */

   delete h;    // デストラクタが呼ばれる
}
```

継承 (inheritance)

- 基本となるクラス (基底クラス) の変数・関数を受け継いだクラス (派生クラス) を定義できる

クラスの継承の例

```
class MyBase {
public:
   MyBase(int _a) : a(_a) {}
   int getValue()  const { return a; }
   void hello() { cout << “Base” << endl; }
protected:   // 派生クラスからもアクセス可
   int a;
};

// MyBase クラスを継承したクラス
class ClassA : public MyBase {
public:
   ClassA(int _a) : MyBase(_a) {}
   void hello() { cout << “ClassA” << endl; }
};
```

```
MyBase b(1);
b.getValue();   // 結果は 1
b.hello();   // 結果は “Base”

ClassA a(2);
a.getValue();   // 結果は 2
a.hello();    // 結果は “ClassA”
```

継承すると、派生クラスのコンストラクタより先に基底クラスのコンストラクタが自動的に呼ばれる

継承を使ったプログラミングの例

- 仕様変更があっても修正がしやすい
  - 例えば swtich 文を使った実装を置き換えられる

Switch 文による実装

```
void display() {
   switch (method_param) {
   case Method_A: …
   case Method_B: …
   ….
   default: ….
   }
}

void animate() {
   switch (method_param) {
   case Method_A: …
   case Method_B: …
   ….
   default: ….
}
```

実装し忘れると
実行時にエラー

継承を使った実装

```
// 個別にクラスを用意しておく
// MethodBase は基底クラスとする

MethodBase *m = new MethodA();

m->display();

m->animate();
```

実装のミスは
コンパイラがチェック

Method_C, Method_D, …
が増えたら？

void model(), void simulate() …
が増えたら？

テンプレート (template)

- 型をパラメータとして新しい関数やクラスを定義できる
  - コンパイル時に型が決まる
  - マクロによる関数定義の強化版とも言える？

テンプレート関数の例

```
template <class T>
const T& MyMax(const T &a, const T &b) {
    return (a > b) ? a : b;
}

cout << MyMax(3, 4) << endl;   // int 型, 4 が表示される
cout << MyMax(6.2, 3.8) << endl; // double 型, 6.2 が表示される
```

- 変数の型が自明であればコンパイラが自動で推定
- 型が曖昧なら MyMax<int>(3, 4) のように明記する

標準テンプレートライブラリ (STL)

- C++ で標準的に提供される、テンプレートを用いて実装されたライブラリ
- 様々なデータ構造、アルゴリズムなどを簡単に利用できる

- 例:
  - **std::vector** …可変長配列、非常によく利用する
  - std::list … リスト / std::map … キーと値のペアを格納
  - std::queue … キュー
  - std::priority_queue …優先度付きキュー

std::vector の利用例

- std::vector に値を登録する例
  - push_back 関数で配列の最後に値を追加
  - 入りきらなくなったら自動で配列サイズを増やしてくれる

```
vector<int> a;
a.push_back( 3 );
a.push_back( -2 );
a.push_back(  1 );
```

```
// 最初に要素数を指定
vector<int> a(3);
a[0] = 3;
a[1] = -2;
a[2] =  1;
```

```
vector<int> a;
a.resize(3); // 後で要素数を指定
a[0] = 3;
a[1] = -2;
a[2] =  1;
```

- std::vector の値を参照する例

```
for (int i=0; i<(int)a.size(); i++)
   cout << a[i] << endl;
```

```
vector<int>::const_iterator  itr  =  a.begin();
while ( itr != a.end() ) {
   cout << *itr << endl;
   ++itr;
}
```

反復子 (イテレータ) を使った例

反復子 (Iterator; イテレータ)

- データ構造によらず要素にアクセスするための仕組み
  - 普通の配列や vector ならランダムアクセス (a[i] など) 可能
  - しかし list などは先頭から順番に辿るしかない
  - **データ構造の変更時に要素アクセスの変更が最小限に**
- 使い方はポインタとほぼ同じ
  - イテレータ itr の指す要素にアクセスするなら *itr
  - 先に進むときは ++itr や itr++ (前者の方が効率がよい)

- STL のデータ構造が提供する特別なイテレータ
  - begin(): 先頭の要素を指すイテレータ
  - end(): 最後の要素**のひとつ先**を指すイテレータ

std::algorithm の利用例

- データのソーティング

クラスの定義

```
class Hoge {
public:
   Hoge(int _a) : a(_a) {}
   int getValue() const { return a; }

   // この関数を用意しておく
   bool operator<(const Hoge &h) const {
      return a < h.a;
   }

private:
   int a;
};
```

STL の利用例

```
#include <vector>
#include <algorithm>

using namespace std;

{
   vector<Hoge> hoges;

   // 値の格納
   hoges.push_back(  Hoge(8)  );
   hoges.push_back(  Hoge(2)  );
   hoges.push_back(  Hoge(3)  );

   // 値のソート
   sort(hoges.begin(), hoges.end());
}
```

付録：Visual C++ のショートカットコマンド

- ショートカットを覚えて作業効率を上げる

コマンド	説明
Ctrl-K + Ctrl-C	選択した（複数の）行をコメントアウト
Ctrl-K + Ctrl-U	選択した（複数の）行のコメントを解除
Ctrl-K + Ctrl-F	選択した（複数の）行を適切にインデント
Ctrl-G + 行番号	指定した行番号にジャンプ
Ctrl-K + Ctrl-K	現在の行にブックマークを設定または削除
Ctrl-K + Ctrl-N	次のブックマークに移動
F5	必要に応じてビルドしてデバッグ**あり**実行
Ctrl-F5	必要に応じてビルドしてデバッグ**なし**実行
Ctrl+Space	入力中の変数名・関数名の補完
Ctrl+F	検索ウィンドウの表示
Ctrl+H	置換ウィンドウの表示

より詳しくは…

- オンライン資料
  - WisdomSoft: http://wisdom.sakura.ne.jp/
- 書籍
  - Bjarne Stroustrup 著:「プログラミング言語 C++」
  - その他…