初心者用
Ｃ＋＋講座

第 1 版 2000 年 11 月 27 日

文責：斎藤輪太郎、永安悟史

0. はじめに

このテキストではC++のごく基本的なことを学びます。C＋＋はCの上位互換言語であり、C言語の長所を生かしつつ、オブジェクト指向プログラミングができるので大変強力です。このテキストを読むにあたっては、C言語の知識があるとより理解しやすいでしょう。

1. 簡単な入力と出力

C++の基本の第一歩として、文字列の出力をしてみましょう。

```
#include <iostream.h>
main(){
   cout << “Hello, world!”;
}
```

これは Hello, world!という文字列を画面上に出力するプログラムです。そしてその中の最初の一行目の#include 文は入出力に関する情報を取り込んでいます。これにより cout << などができるようになります。cout << は画面に（正確にいうと標準出力）文字列を出力せよという意味です。なお、main はここがプログラムが最初に実行されるところであることを示します。

出力できるのは文字列だけではありません。数値も同様に出力することができます。

```
#include <iostream.h>
main(){
   cout << 1 + 2;
}
```

1 + 2 が計算されて 3 が出力されます。

<< の記号は以下のように続けて使うこともできます。

```
#include <iostream.h>
main(){
   cout << “1 + 2 = ” << 1 + 2;
```

```
}
```

これではっきりしたと思いますが、上の例の場合、二重引用符(double quotation, “ ～ “)で囲まれた部分は文字列を表し、そうでない部分は数値演算を表します。

変数の中身を表示することもできます。

```
#include <iostream.h>
main(){
   int sum;
   sum = 4 + 6;
   cout << sum;
}
```

上の例では sum の変数の中身である 10 が出力されます。ちなみに、int は sum という変数名の、整数を扱う変数を使うことを宣言します。

もちろん、出力できるのは整数用の変数 int の中身だけではありません。小数用の変数 double の中身や、文字列用の変数 char *の中身も表示できます。

```
#include <iostream.h>
main(){
   double prod;
   char *message;
   prod = 1.2 * 0.2;
   message = “The product is: ”;
   cout << message << prod;
}
```

実際に上のプログラムを入力して実行してみましょう。

課題：12.34 x 14 の答えを”12.34 x 14 = “の文字列とともに出力しましょう。

これまでのところで、変数の中身の出力を取り扱ってきましたが、プログラムの実行後、ユーザが変数へ値を入力できるようにする方法もあります。

```
#include <iostream.h>
main(){
   int x;
   cout << “Type one number. “;
   cin >> x;
   cout << “The number you typed is: “ << x;
}
```

cin >> は標準入力を変数に格納します。この場合、標準入力はユーザの入力です。上のプログラムを実際に打ち込んでみましょう。次のような出力が得られるはずです。

```
% a.out
Type one number. 123     ← ユーザ入力
The number you typed is: 123
%
```

標準入力の格納先として、小数の変数(double 型)や文字列の変数(char *型)を指定することもできます。

課題：2 つの数をユーザに入力してもらい、その積を出力するプログラムを書きましょう。

2.メモリ確保

C++では比較的手軽にメモリ領域を（原則的に）任意の大きさで確保することができます。例えばいくつかの数を入力してもらい、平均と、各々の数が平均からどれだけ離れているか（偏差）を出力するプログラムを考えます。まずは以下のようなプログラムが考えられるでしょう。

```
#include <iostream.h>
main(){
   int num[20];
   int n, sum;
   double average;
   int i;
   cout << “Input number of numbers:”;
   cin >> n;
```

```
  for(i = 0;i < n;i ++){
    cout << “Input number “ << i + 1 << ‘ ‘;
    cin >> num[i];
  }

  sum = 0;
  for(i = 0;i < n;i ++)sum = sum + num[i];
  average = 1.0 * sum / n;
  cout << “Average: “ << average << ‘¥n’;
  for(i = 0;i < n;i ++)
    cout << i << “: ” << num[i] << ‘ ‘ << num[i] – average << ‘¥n’;
}
```

※for(i = 0;i < n;i ++)は i の値を 1 つずつ増やしながら次にある文を n 回実行する文です。
※大きな配列を宣言するときは static int num[100]; というように static をつけましょう。

int num[10]で 10 個分の数を格納できるような配列を宣言しています。しかし、これは 20 個の数を入力するときは小さいし、4 個しか数を入力しないときは残りの 6 個分の変数のスペースが無駄になってしまいます。

そこで new 演算子を使うとプログラム実行中に任意の大きさの配列を確保することができます。実際に以上のプログラムを打ち込んで実行してみましょう。

```
#include <iostream.h>
main(){
  int *num;
  int n, sum;
  double average;
  int i;
  cout << “Input number of numbers:”;
  cin >> n;
  num = new int[n];
  for(i = 0;i < n;i ++){
    cout << “Input number “ << i + 1 << ‘ ‘;
    cin >> num[i];
  }
```

```
  sum = 0;
  for(i = 0;i < n;i ++)sum = sum + num[i];
  average = 1.0 * sum / n;
  cout << “Average: “ << average << ‘¥n’;
  for(i = 0;i < n;i ++)
    cout << i << “: ” << num[i] << ‘ ‘ << num[i] – average << ‘¥n’;
  delete num;
}
```

任意の大きさの配列を確保するには、

① まず変数名（配列名、上の例では num）と型（上の例の場合は int 型）を決めます。’*’ は変数が配列（ポインタ）であることを示します。

```
  int *num;
```

② 次に new 演算子で配列を確保します。使用方法は、

配列変数名 = new 型名[配列の大きさ];

です。配列の大きさは以下のように変数にできるのが特徴です。

```
  num = new int[n];
```

③ 確保した配列を使う必要がなくなったら、delete 演算子で配列を開放します。コンピュータの資源を節約するため、使う必要がなくなった配列はなるべく開放するようにします

```
  delete num;
```

応用課題：C 言語のポインタの扱いが初心者という方にはやや高度になりますが、入力された名前を番号をつけてそのまま順番に表示するプログラムを作成してみましょう。名前は 20 字以内であるとします。

出力例：

```
How many persons? : 3    ←　ユーザ入力
Name 1: Manaka           ←　ユーザ入力
Name 2: Inaba            ←　ユーザ入力
Name 3: Furuta           ←　ユーザ入力

1. Manaka        ←　番号をつけて名前を出力
2. Inaba
3. Furuta
```

まず、名前は文字列です。そして文字列はC言語ではchar型の配列になります。さらに複数の人の名前を記憶する関係上、名前の配列を作らなければいけないので、名前を指す変数は配列の配列、つまり二次元配列になります。

```
  char **names;
```

人数分の名前を登録する変数を確保するには、

```
  names = new char *[number_of_person];
```

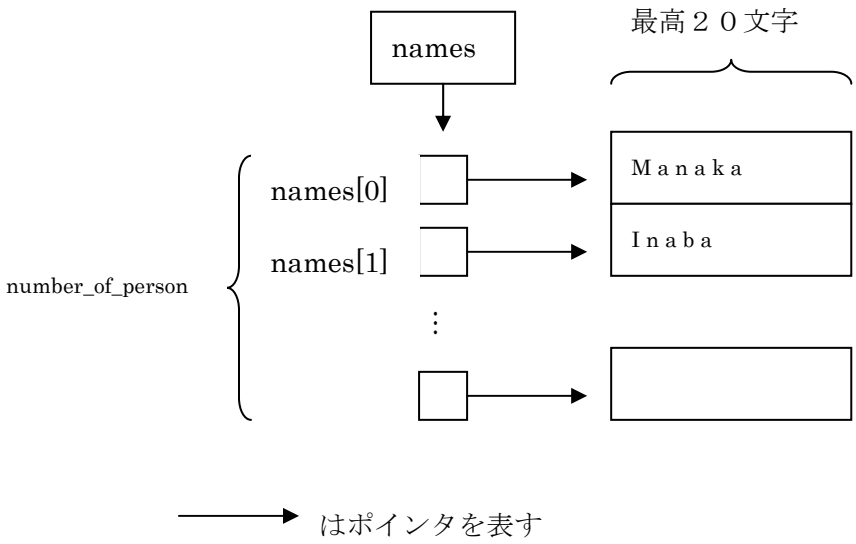
とします。しかしこれだけでは、名前が number_of_person 分あるといっているだけなので（正確には名前へのポインタを人数分確保しただけ)、次に名前を入れるスペースを確保します。

```
  for(i = 0;i < number_of_person;i ++)names[i] = new char[20];
```

これらの変数を開放するときは、

```
  for(i = 0;i < number_of_person;i++)delete names[i];
  delete names;
```

とします。これらの配列の関係を図示すると以下のようになります。



あとはがんばってこの応用課題に取り組んでみましょう。

3.参照

C++では、関数の中でそこに渡された引数の変数の値を比較的簡単に変えることができます（C言語の場合、ポインタを使わなければなりません)。これには参照という方法を使います。以下のプログラムを打ち込んで実行してみましょう。

```
#include <iostream.h>

plus_one(int &x){
   x = x + 1;
}

main(){

   int num;
   cout << “Input one number:” ;
   cin >> num;
   plus_one(num);
   cout << num;
}
```

‘&’という記号が参照に使われる演算子です。これを関数のパラメータ変数につけることで参照ができるようになります。これにより、その変数の値を変えると、呼び出し元の変数の値も変わるようになります。

```
plus_one(int &x){
   x = x + 1;
}
```

上の例では、x の値が 1 つ足されていますが、呼び出し元の変数 num の値も変わります。

```
   plus_one(num);
```

関数のパラメータから&をとると、x の値はもちろん 1 つ足されますが、呼び出し元の変数 num の値は変化しなくなります。

課題：2 つの整数型変数 x と y を受け取り、x と y の数を入れ替えるような関数を参照を使って作成しましょう。

4.クラス

クラスを使うことで、変数とそれに対する手続きをひとまとめにして定義することができます。このクラスについて順番に学んでいきましょう。

4.1 クラスの定義と宣言

まず、クラスは関連するデータをひとまとめにして扱うことができます（C 言語の struct と同じような感覚で使うことができます)。

以下のプログラムを見て下さい。

```
#include <iostream.h>
class Cube {
public:
   int height, width, depth;
};
```

```
main(){

   Cube cube1;

   cube1.height = 10;
   cube1.width = 20;
   cube1.depth = 15;

   cout << "Height: " << cube1.height << " Width: " << cube1.width
      << " Depth: " << cube1.depth << '¥n';
}
```

クラスを定義するときは以下のような書式を使います。

class クラス名 {

   クラスの内容

};

従って、上の例では Cube という名前のクラスを定義していることになります。

プログラムを見るとその中で public という文字が出てきます。これは変数が外部からアクセス可能であることを示します。詳しくは後ほど説明します。

その後の、

```
   int height, width, depth;
```

は height, width, depth という変数が Cube というクラスに含まれていることを示しています。これによりこの３つの変数を Cube というクラスの中でまとめて扱うことができます。

さて、これだけでは Cube というクラスがどんなものであるかは示していますが、実際にどの変数が Cube という型のクラスであるかを示していません。

それを示すのが、

```
Cube cube1;
```

です。これを宣言といいます。これによって cube1 という変数が Cube という型のクラスになります。

クラスの宣言の方法は一般的に、

クラス名 変数名;

です。

そしてクラスの中の変数にアクセスするには例えば、

```
cube1.height = 10;
```

とします。上の例の場合、Cube というクラスの cube1 という変数の height という変数の値を 10 に設定しています。

クラスの中の変数や関数をメンバと呼びます。メンバにアクセスするためには、

変数名.メンバ

のように変数名とメンバの間にピリオドを入れます。

課題：上記の例にならって、長方形を表すクラス`Rectangle`を定義し、`width`と`height`という変数をメンバにしてみましょう。次にこのクラスの変数を宣言し、`width`と`height`にそれぞれ `10` と `20` を代入して表示してみましょう。

4.2 メンバ関数

上の例では Cube というクラスの中に width, height, depth という変数を導入しました。せっかくですから、ここから体積を計算することはできないでしょうか？

実は、クラスのメンバの中には変数の他に関数を含めることができます。その関数の中で

width x height x depth を計算すれば、体積が求められます。以下のプログラムを打ち込んで実行してみましょう。

```
#include <iostream.h>
class Cube {
public:
   int height, width, depth;
   int volume();
};

int Cube::volume(){
   return height * width * depth;
}

main(){
   Cube cube1;
   cube1.height = 10; cube1.width = 20; cube1.depth = 15;
   cout << cube1.volume() << '¥n';
}
```

クラスの中で関数を定義するときは、まず Cube の定義の中で関数名を定義します。

```
class Cube {
public:
   int height, width, depth;
   int volume(); // 関数名をとりあえずここに書く
};
```

そしてその関数の手続きを後で以下のように記述します。

```
int Cube::volume(){
   return height * width * depth;
}
```

これで、cube1.volume() によってvolume() が呼ばれ、上のプログラムのケースでは、cube1の中のheight、width、depthの積が計算され、体積が返ってきます。

一般的な記述方法は、

返り値の型 クラス名::メンバ関数名(引数){

　関数の手続き

}

です。

メンバ関数の定義は以下のようにクラスの定義の中に組み込むこともできます。この場合、この関数をインライン関数と呼びます。

```
class Cube {
public:
  int height, width, depth;
  int volume(){ return height * width * depth; }
};
```

インライン関数を定義した場合、プログラムの実行速度が理論的には若干（多くの場合、ほとんど分からないくらい）上がりますが、プログラムの容量が増えてしまいます。

課題：上のクラス Cube に表面積を計算する surface()を定義し、それを呼び出してみましょう。次に surface()をインライン関数にしてみましょう。

4.3 コンストラクタとデストラクタ

Cube のプログラムの例で、

```
  Cube cube1;
```

と宣言した時点では、cube1.height、cube1.width、cube1.depth にはどんな値が入っているかは不定です。何とか初期値を入れることはできないでしょうか？これを実現するのがコンストラクタ（構築子関数）です。

コンストラクタはクラスが宣言された時点で呼ばれます。以下の例では、Cubeのクラスが宣言されると、height, width, depthの全ての変数に 1 が入ります。

```
class Cube {
public:
   int height, width, depth;
   Cube();
};

Cube::Cube(){
   height = width = depth = 1;
}
```

Cube()というのがコンストラクタです。クラス名と同じですね。しかも Cube のメンバ関数であることが分かります。

以上をまとめると、クラス名と同じ名前のメンバ関数を定義すると、それはコンストラクタとなり、クラス宣言時に呼ばれます。

デストラクタ（消滅子関数）というのもあります。これもメンバ関数の一種で、宣言したクラスのある関数の実行が終わるときに呼ばれます。関数名は

~クラス名()

となります。

実際に以下のプログラムを打ち込んで実行し、コンストラクタと、デストラクタの役割を調べてみましょう。

```
#include <iostream.h>

class Cube {
public:
   int height, width, depth;
   Cube();  // コンストラクタ
   ~Cube(); // デストラクタ
```

```
   int volume();
};

Cube::Cube(){  // コンストラクタ
   height = width = depth = 1;
   cout << “Function Cube() called.¥n”;

}

Cube::~Cube(){ // デストラクタ
   cout << “Function ~Cube() called.¥n”;
}

int Cube::volume(){
   return height * width * depth;
}

main(){
   Cube cube1;
   cout << cube1.volume() << ‘¥n’;
   cube1.height = 10; cube1.width = 20; cube1.depth = 15;
   cout << cube1.volume() << ‘¥n’;
}
```

課題：上記の例にならって、長方形を表すクラス `Rectangle` を定義し、`width` と `height` という変数をメンバにしてみましょう。そして上の例にならって初期値をそれぞれ 10 と 20 にするようなコンストラクタを作成しましょう。

4.4 public と private

上の例の Cube の例では、体積を知りたいとき、毎回 volume()が呼ばれていました。そして volume()の中では毎回、掛け算が行われていました。しかし、width、height、depth の値が変わらないなら、掛け算は 1 回だけで済ませたいですね。width、height、depth のいずれかの値が変わったときだけ、掛け算を再実行するようにすることはできないでしょうか？

以下のクラス定義を見て下さい(メンバ関数は全てインラインにしてあります)。

```
class Cube {
public:
   int height, width, depth;
   int vol;
   Cube(){ height = width = depth = 1; } // コンストラクタ
   new_height(int h){ height = h; vol = height * width * depth; }
   new_width(int w){ width = w; vol = height * width * depth; }
   new_depth(int d){ depth = d; vol = height * width * depth; }
   int volume(){ return vol; }
};
```

今までは、cube1 の中の height の値を例えば 5 にセットするときは、

```
   cube1.height = 5;
```

としていましたが、上のクラス定義を使えば、

```
   cube1.new_height(5);
```

とすることができます。すると、

① メンバ関数の中で h に 5 が代入され、 new_height(int h)
② それが height に代入され、 height = h;
③ 体積も自動的に計算されて、vol という変数に入ります。
vol = height * width * depth;

体積が知りたいときは例えば、

```
  cout << cube1.volume();
```

とすれば、以下のように、メンバ関数の中であらかじめ計算された vol の値が返るだけなので、体積の再計算は行われません。

```
  int volume(){ return vol; }
```

実際に以上のクラス定義をプログラムの一部として書いて、height や width に様々な値を代入して体積を表示してみましょう。

課題:上記の例にならって､長方形を表すクラス Rectangle を定義し､width と height、area(面積)という変数をメンバにしてみましょう。そして上の例にならって width や height に値を代入し面積を area に代入するメンバ関数、面積の値を返すメンバ関数を作成してみましょう。

変数への代入を関数に任せる、なかなかいい方法ですが、１つ注意が必要です。変数への代入を常にメンバ関数に任せていれば、volの値は常にwidth x height x depthと等しくなります。したがって、volume()関数では単にvolの値を返すだけで十分です。

しかし、ひとたび変数への代入が関数を通さず直接行われてしまうと、volの値がwidth x height x depthと等しいという保証がなくなります。

例えば次のような文は変数の値を直接変えてしまうものです。

```
  cube1.height = 5;
```

そこで C++では変数の値を直接変える（言いかえれば、メンバ関数の中以外のところで変数の値を変える）ことを避けるために private というアクセス指示子があります。

これを使って Cube の定義を書きかえると以下のようになります。

```
class Cube {
private:
   int height, width, depth;
   int vol;

public:
   Cube(){ height = width = depth = 1; } // コンストラクタ
   new_height(int h){ height = h; vol = height * width * depth; }
   new_width(int w){ width = w; vol = height * width * depth; }
   new_depth(int d){ depth = d; vol = height * width * depth; }
```

```
   int volume(){ return vol; }
};
```

private 指示子はそれに続く変数や関数のメンバ関数外からのアクセスを禁止するものです。上の例では、height, width, depth, vol の 4 つの変数にアクセス制限がかかります。

試しに、private 指示子を入れて

```
   cube1.height = 5;
```

という文を入れてコンパイルしてみて下さい。コンパイル時にエラーが出るのが分かると思います。

これに対し、public 指示子はそれに続く変数や関数のメンバ関数外からのアクセスを許可します。

応用課題 : ここまでの知識を生かして、文字列を保持するクラス Mojiretsu を作ってみましょう。Mojiretsu のメンバ変数は、

```
char *str;    // 文字列を指し示す
int length;   // str の長さ
```

の 2 つとします。これらは private にします。メンバ関数は、

```
Mojiretsu();         // コンストラクタ。文字列を'¥0'の 1 文字だけにして、
                     // length を 0 に設定。
~Mojiretsu();        // 文字列を削除 : str が指すメモリ空間を開放する。
char *ret();         // 文字列 str を返す。
clear();             // 文字列を削除 : 文字列を'¥0'の 1 文字だけにする
append(char *s);     // 文字列 s を str の後に追加し、length を増やす。
```

の 4 つです。

まず、文字列の操作があるので、

```
#include <string.h>
```

をプログラムの先頭につけます。

文字列の長さを 0 にするときは、

```
   str = new char[1];  // ‘¥0’の文のメモリを確保
   str[0] = ‘¥0’;
   length = 0;
```

のようにするといいでしょう。ちなみに、’¥0’は文字列の終わりの意味です。

str に新たな文字列 s を追加するときは、

```
   char  *new_str;
   length = strlen(str) + strlen(s);  // strlen は文字列の長さを求める
   new_str = new char[length + 1];    // 新しい文字列のためのメモリを確保
   strcpy(new_str, str);   // new_str に str をコピー
   strcat(new_str, s);     // さらにその後に、s を追加
   delete str;             // 古い文字列のメモリを削除
   str = new_str;          // 新しい文字列を str に代入
```

のようにするといいでしょう。あとはがんばってみましょう。ちゃんと文字列を出力して動作確認を忘れずに。以下が確認するためのプログラム例（一部）です。

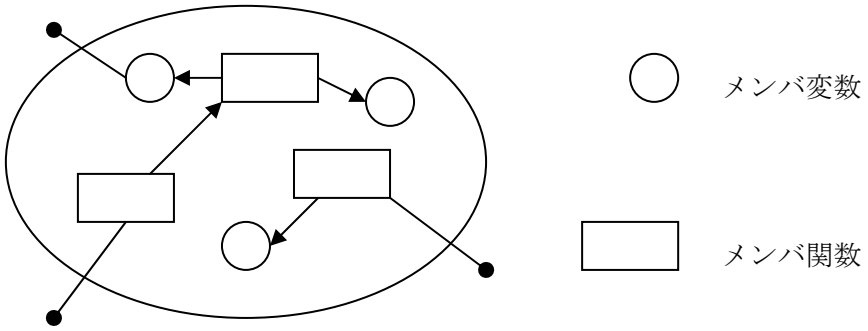
```
Mojiretsu str1;

str1.append(“enjoy”);
cout << str1.ret();
str1.append(“ment”);
cout << str1.ret();
```

5.クラスの継承

5.1 カプセル化

オブジェクト指向言語では、変数や関数などの要素をカプセル化して扱います。カプセル化とはどのような概念なのでしょうか。継承の理解には、カプセル化を理解することが最低限必要ですので、最初に簡単にカプセル化について述べます。



クラス

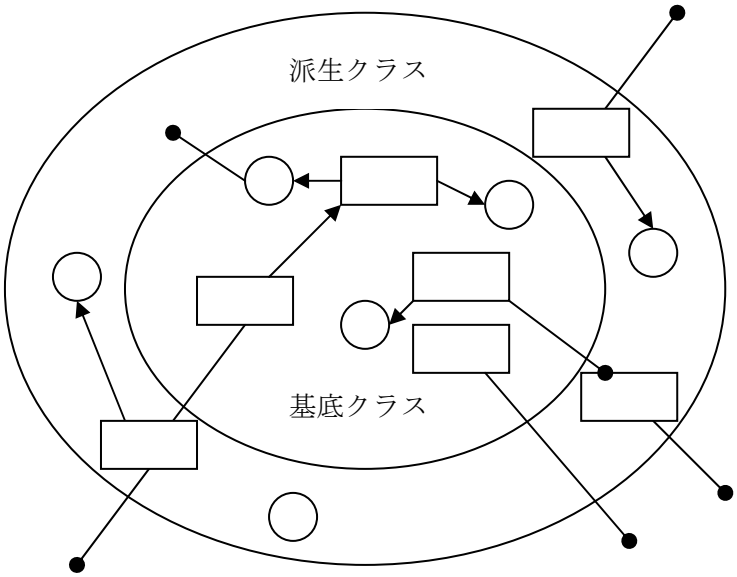
「カプセル化」とは、変数や関数などを、あたかもカプセルに閉じ込めるように、クラスの中にまとめてしまうことです。クラスの中にまとめられた変数や関数は、それぞれ「メンバ変数」、「メンバ関数」などと呼ばれます。

メンバには、外から見えるもの（public なもの）や、外からは見えず、そのクラスの中だけで見えるもの（private）なものがあります。

変数や関数などを操作する場合には、そのカプセルの外に見えている部分（public なメンバ）だけを通して操作することになります。

5.2 クラスの再利用

C++を始めとする主なオブジェクト指向言語では、作成したクラスを元に、それを発展させることで、元のクラスを再利用することができます。これを「クラスを継承する」、「クラスを派生させる」と言います。



元となるクラスのことを、「基底クラス」または「スーパークラス」と呼び、継承したクラスを「派生クラス」または「サブクラス」と呼びます。

派生クラスでは、基底クラスのメンバ関数やメンバ変数を利用することができます。基底クラスの資産を使うことによって、同じようなクラスを何度も設計する手間を省くのが、「C++のクラスの再利用性」です。

5.3 継承の実際

それでは、派生の実際を見てみましょう。

```
class A {
public:
  A();

  void func1();
  void func2();
  void func3();
};
```

ここに、A というクラスがあります。クラス A は、コンストラクタの他に、func1()、func2()、

func3()というメンバ関数を持っています。

次に、このAというクラスを継承して、クラスBというクラスを作ってみます。

```
class B : public A {         // <- Aから継承してBを定義
public:
  B();
};
```

クラス B では、コンストラクタ以外のメンバ関数は定義していませんが、実際には、クラス A の持つメンバ関数を利用することができます。

以下に、実際に動くソースコードを載せますので、実際に試してみてください（basic.cpp）。
※行番号は入力しないように！！

```
 1: //
 2: // basic.cpp
 3: //
 4: #include <iostream.h>
 5:
 6: class A {
 7: public:
 8:   A() {
 9:     cout << "A::A() called." << endl;
10:   }
11:
12:   void func1() {
13:     cout << "A::func1() called." << endl;
14:   }
15:   void func2() {
16:     cout << "A::func2() called." << endl;
17:   }
18:   void func3() {
19:     cout << "A::func3() called." << endl;
20:   }
21:
22: };
```

```
23:
24: class B : public A {
25: public:
26:   B() {
27:     cout << B::B() called." << endl;
28:   }
29: };
30:
31: int main(void)
32: {
33:   A a;
34:
35:   a.func1();
36:   a.func2();
37:   a.func3();
38:
39:   B b;
40:
41:   b.func1();
42:
43:   return 0;
44: }
```

コンパイルして、実際に動作させてみます。

```
 1: % g++ basic.cpp
 2: % ./a.out
 3: A::A() called.
 4: A::func1() called.
 5: A::func2() called.
 6: A::func3() called.
 7: A::A() called.
 8: B::B() called.
 9: A::func1() called.
10: %
```

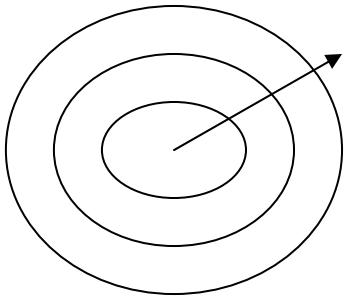
ここには、いくつか重要な情報が含まれています。

33 行目では､クラス A のオブジェクトを作成しています｡オブジェクトを作成したときに、コンストラクタが呼ばれ（3 行目)、初期化の処理が行われます。続く 35、36、37 行目では、クラス A のメンバ関数をそれぞれ呼び出しています（4、5、6 行目)。ここまでは、クラス A に関する操作になっています。

次に、39 行目において、クラス B のオブジェクトを作成しています。オブジェクトを作成した時、クラス A とクラス B のコンストラクタが順に呼ばれています（7、8 行目)。

これは､クラス B がクラス A から派生している為に､まず派生する元のスーパークラス（クラス A）のコンストラクタが呼ばれ、その後にクラス B のコンストラクタが呼ばれていることを意味します。

あるクラスから派生したクラスを作成する場合、そのクラスのルーツを辿るようにして、大元のクラスのコンストラクタから順番に呼び出されることになります。



基底クラス（内側）が作られてから
派生クラス（外側）が作られる

つまり､クラス B がクラス A から派生している場合､クラス B はクラス A の一種な訳ですから、クラス A としての準備を整えて（コンストラクタを呼び出して）から、クラス B のコンストラクタを呼び出します。

このようにして､「object of class B inherited from A」が作成されます。

次に、今作成したクラス B のオブジェクトを介して、メンバ関数 func1()を呼び出しています。ソースコードからも明らかなように、クラス B の設計においては、func1()というメンバ関数は作成していません。ただ、クラス A を継承してクラス B を定義しただけです。にも関わらず､「A::func1() called.」という結果が表示されています。

結果の 9 行目から分かるように、41 行目の「b.func1()」で呼び出されたのは、クラス B の

func1()ではなく、クラス A の func1()です。

これが、「継承」の基本です。

継承というのは、「受け継ぐ」ことですから、継承する前に持っていたメンバ変数や関数は、継承した後の派生クラスにおいて、あたかもそのクラスが持っているかのように利用することができるのです。

逆に、クラスが壊されるときには、デストラクタは派生クラス（外側）から呼び出され、基底クラスのデストラクタは一番最期に呼び出されます。

課題： 以下のクラス A の定義を見て下さい。

```
#include <iostream.h>
#include <string>

class A {
  string s;

  A() {
    s = "This is A.";
  }

  void print() {
    cout << s << endl;
  }
};
```

ここでクラス A を継承して、クラス B を作り（定義し）、クラス B のオブジェクトからクラス A のメンバ関数 print()を呼び出してみましょう。main 関数は、

```
int main()
{
  B b;
```

```
  b.print();

  return 0;
}
```

のようになります。

課題： 適当な基底クラスと派生クラスを作成し、デストラクタが派生クラスから呼び出されることを確認しましょう。

5.4 メンバ関数（メソッド）のオーバーライド

今回、クラス B ではメソッドを一切定義しませんでした。クラス A のメンバ関数を利用できたのは、クラス B で同じ名前のメンバ関数を定義しなかったからです。

それでは、クラス B でクラス A の持っているメンバ関数と「同じ型」の「同じ名前」のメンバ関数を作成したらどうなるのでしょうか。

実際に見てみましょう。

```
//
// override.cpp
//
#include <iostream.h>

class A {
public:
  void func() {
    cout << "A::func()" << endl;
  }
};

class B : public A {
public:
  void func() {
```

```
    cout << "B::func()" << endl;
  }
};

int main(void)
{
  A a;
  B b;

  a.func();
  b.func();

  return 0;
}
```

これを実行すると、

```
% g++ override.cpp
% ./a.out
A::func()
B::func()
%
```

となります。それぞれ、クラス A／クラス B の同名のメンバ関数が呼ばれています。先程はクラス B に void func() というメンバ関数がなかったので、クラス A の void func() が呼ばれましたが、今回はクラス B の void func() が呼ばれています。

派生クラスにおいて、基底クラスのメンバ関数と同じ名前のメンバ関数を作成することを、メンバ関数（メソッド）を「オーバーライド（override）する」と言います。

このように、オーバーライドをすることによって、元のメンバ関数とは違う内容のメンバ関数を同じ名前で呼び出すことができます。

課題：以下の Fruit クラスを継承して、Apple、Grape、Banana クラスをそれぞれ作りましょう。それぞれのクラスの、name 関数、color 関数で、各フルーツの名前と色を表示するようにメンバ関数をオーバーライドしましょう。

```
#include <iostream.h>
#include <string>

class Fruit {
  Fruit() {
    myname = "fruit";
    mycolor = "unknown";
  }

  void name() {
    cout << "Name: " << myname << endl;
  }

  void color() {
    cout << "Color: " << mycolor << endl;
  }

  string name;
  string color;
};
```

課題：以下の空の基底クラス（Formula）を継承して、二次方程式の解を解くクラス（QuadraticFormula）を作成し、二次方程式を解くプログラムを作成しましょう。各項の係数は利用者が入力できるようにして下さい。二次方程式の解は二つ出るので、それぞれans1()関数とans2()関数を用意し、二つの関数でそれぞれの解を表示してみましょう。

```
class Equation {
public:
  Equation() {
  }

  double calculate() {
  }
};
```

応用課題 : 基底クラスと派生クラスにおいて同じ型、同じ名前のメンバ変数があった場合、基底クラス内、派生クラス内において、それぞれどのような挙動をするか確認するソースコードを書きましょう。その結果、何が分かりましたか？

応用課題 : 継承を利用する際のメリットは？

5.5 has-A と is-A

クラスの継承を覚えると、何がなんでも継承を用いようとする人がいます。継承を利用することで、コードがエレガントになったように感じるからです。しかし、似たようなクラスがある時に、安直に継承を用いることは後に混乱を引き起こすことになります。

また、そうでなくとも、どのような場合に継承させるべきなのか、またどのような場合には継承させるべきでないのかを判断しかねる場合などがあります。

ここでは、二つのクラスが存在した場合に、クラスの継承を用いるべきか用いるべきでないか、どのように判断すべきかどうかを解説します。

C++を始めとするオブジェクト指向言語では、継承を考える時に「has-A か is-A か」という話がよくされます。ここでは、has-A と is-A について解説します。

まずは、has-A から。

「has-A」とは、その文字の通り、「A を持っている」という状況を表します。言い換えれば「A is a part of なにがし」という状況です。コンピュータは CPU を持っていますし、メモリやディスクも持っています。これは、CPU やメモリ、ディスクがコンピュータの「構成要素」である場合を表します。このような場合には、そのクラスのメンバとして設計するのが正しくなります。

```
class Computer {
  CPU cpu;
  Memory memory;
  HardDisk disk;
};
```

以下のように、CPU から継承して Computer を作成するのは正しくないことが分かるでしょう。

```
class Computer : public CPU {

};
```

次に is-A について。

「has-A」に対して、「is-A」という関係があります。これも、その通り「なにがし is A」という関係になります。上の例を踏まえて考えれば、

```
class Computer {
  CPU cpu;
  Memory memory;
  HardDisk disk;

  void type() {
    cout << "computer" << endl;
  }
};

class Macintosh : public Computer {
  void type() {
    cout << "Macintosh" << endl;
  }
};

class SPARCStation : public Computer {
  void type() {
    cout << "SPARCStation" << endl;
  }
};

class iMac : public Macintosh {
```

```
  void type() {
    cout << "iMac" << endl;
  }
};
```

のような形になります。

これは、Macintosh は Computer の一種であり、iMac は Macintosh の一種であることを示しています。また、SPARCStation が Computer の一種であることを示しています。

このように、継承というのは「is-A」の場合に使うのが正しく、「has-A」の関係の場合に使うものではありません。

ここでは、継承を使うのが明白な例ばかりですが、実戦的なプログラムを作成し始めると、継承を使うべきなのか、メンバとするべきなのか悩む場合が出てきます。そのような場合には、「is-A」なのか「has-A」なのか、基本に戻って考えるとどちらを使うのかが分かってきます（分かっていても悩む場合もあるのですが）。