

## 4 3 章 構造体・クラス

```
int main(){  
    int yuusya_HP;  
  
    int yuusya_Attack;  
  
    int yuusya_Speed;  
  
    int mahotukai_HP;  
  
    int mahotukai_Attack;  
  
    int mahotukai_Speed;  
  
}
```

ゲームを作るとき、

あなたは「勇者」「魔法使い」という職業を設計したいとします。

必要なステータスは「HP」「攻撃力」「すばやさ」です。

めっちゃくちゃ見づらいです。

それに。。。

勇者のマイケル・魔法使いのマイクみたく、職業を付与すると

```
int main(){
    int yuusya_Michael_HP;

    int yuusya_Michael_Attack;

    int yuusya_Michael_Speed;

    int mahotukai_Mike_HP;

    int mahotukai_Mike_Attack;

    int mahotukai_Mike_Speed;

}
```

2人ならまだですが、プレイヤーが100人になったりすると変数が大変なことになります。

## そこで構造体の出番です！！

```
struct yuusya{
    int HP;

    int Attack;

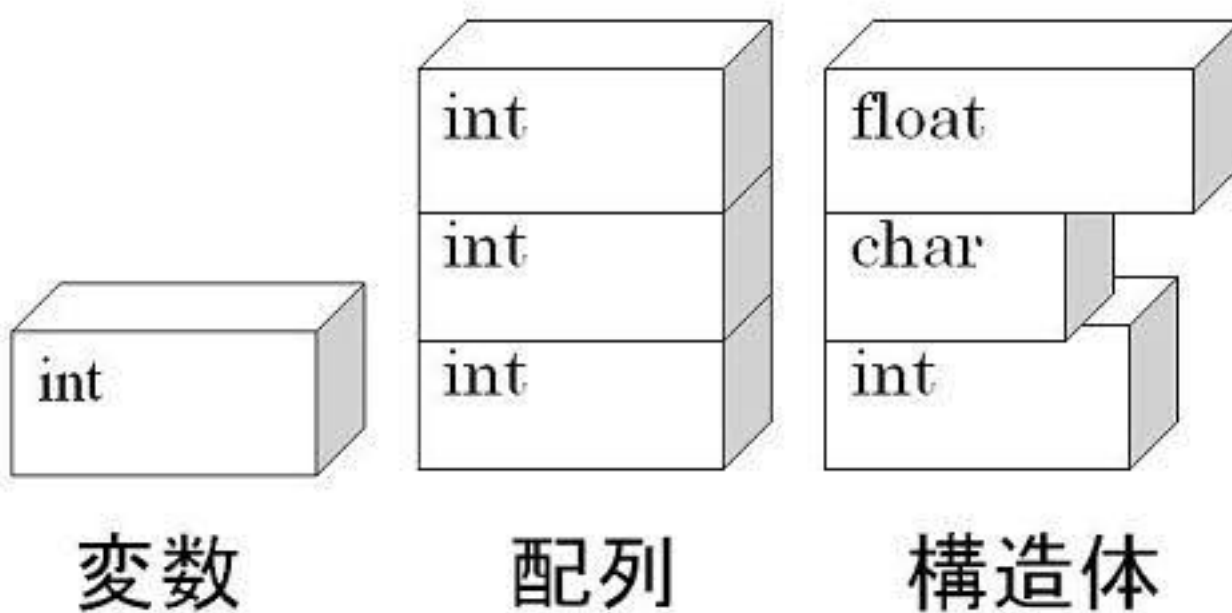
    int Speed;
};
```

変数をまとめることができます。

勇者 構造体を作りました。

構造体はこのようになっています。

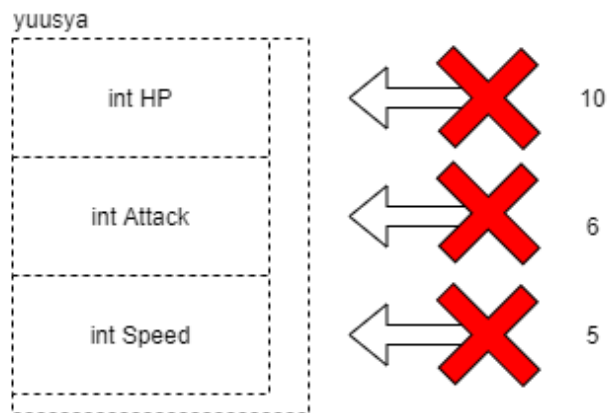
```
struct 構造体名{  
    型 変数名;  
    型 変数名;  
    ⋮  
};
```



構造体は配列のように、複数の変数を宣言できます。

ただ、配列とは違って変数の型は自由です。また、それぞれの変数に名前を付けられます

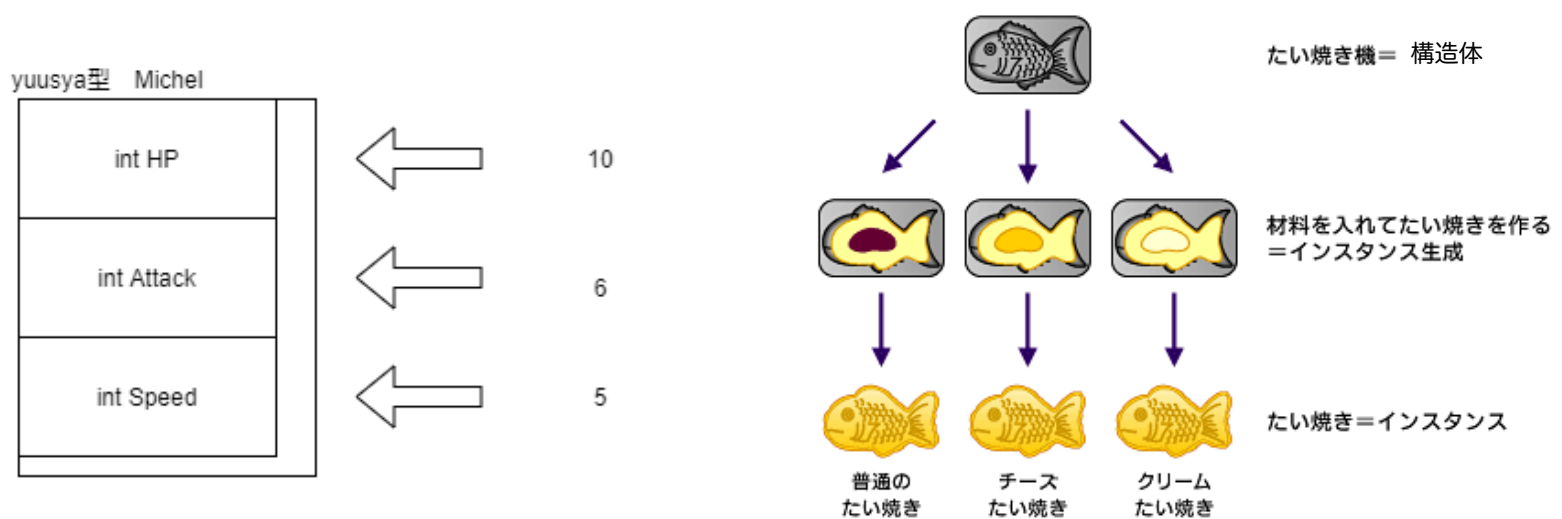
しかしこのままでは構造体の中の変数に値を入れることはできません。なぜならさっき宣言したものはたいやきの型のように構造体を作る「型」に過ぎないからです。



そこで、このように変数を作ります。

```
yuusya Michel;
```

こんな感じで型名を構造体の名前にして変数を作ります。



するとこのように型から構造体が作られます。このように型から作ったものを「インスタンス」といい、型から作ることを「インスタンス化」といいます。MMORPGをしてる人なら聞いたことあるんじゃないでしょうか。

このyuusya型のMichel構造体の中にあるAttackには以下のようにアクセスできます。

```
Michel.Attack=100;  
  
cout << Michel.Attack << endl;
```

構造体内の変数への代入は宣言するときであれば配列のように中括弧カンマ区切りで一度にすることができます。

```
yuusya Bob = {10, 5, 1};  
  
cout << Bob.HP << endl;  
  
cout << Bob.Attack << endl;
```

10

5

また構造体も配列やvectorに入れることができます。

```
yuusya players[10];
vector<yuusya> playerList;
for(int i = 0; i<10; i++){
    players[i].HP = i;
}
playerList.push_back({5,2,6});
```

配列内の構造体の中の変数にアクセスするときは上のように配列名[インデックス].変数名とします。

上のプログラムではfor文で配列内の各構造体の変数にそれぞれ違う値を入れています。型を作ってからインスタンスを作る仕組みになっていることで、同じ項目があるが中の値は異なるものが大量に作れるので、上の勇者のステータス管理のように複数の同じような項目を持つ概念をプログラムで表現するとき便利になります。

vectorのpush\_back時も{}での一括代入ができます。

構造体を関数に渡すことができます。

```
void who_win(yuusya first_player , yuusya second_player){
    if(first_player.HP - second_player.Attack > second_player.HP - first_player.Attack){
        cout << "first player win !!" << endl
    }else{
        cout << "second player win!!" << endl;
    }
}
```

これは、2人のHPと攻撃力を比較して、どっちが勝ったかを判定する関数です。

## 問題 1

生徒の名前、国語・数学・英語の点数を記録する構造体を作り、

その構造体で入力を受け取って下さい

```
>> tarou 50 60 70
```

```
//構造体studentにこの情報が入る。
```

## 問題 2

上の構造体を使い、4人の情報を受け取って下さい

その際、vectorを使用してください。

```
>> tarou50 60 70
```

```
>> suzuki 55 80 40
```

```
>> tanaka 100 90 95
```

```
>> mori 80 80 75
```

### 問題 3

上の構造体の配列(vector)を受け取り、それぞれの科目の平均点を求めるプログラムを書いてください。

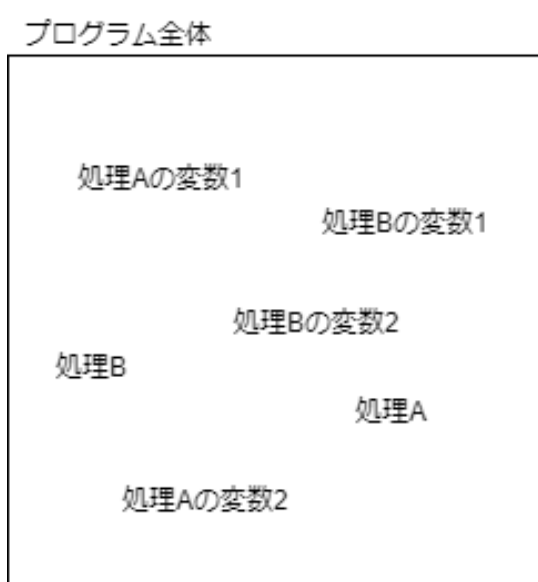


# クラス

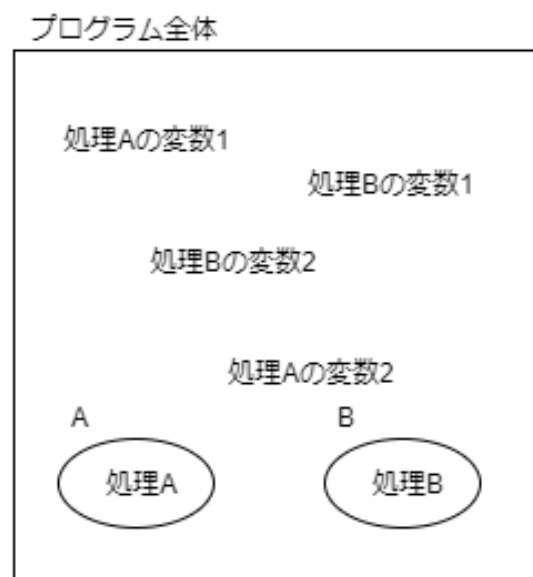
クラスとは、一言でいうならば「関数が入れられるようになった構造体」です。クラスは構造体のような変数だけではなく、関数を宣言し、構造体のようにインスタンス.関数名()の形式で使うことができます。

構造体に関数を入れられるようになったことで何ができるのでしょうか？

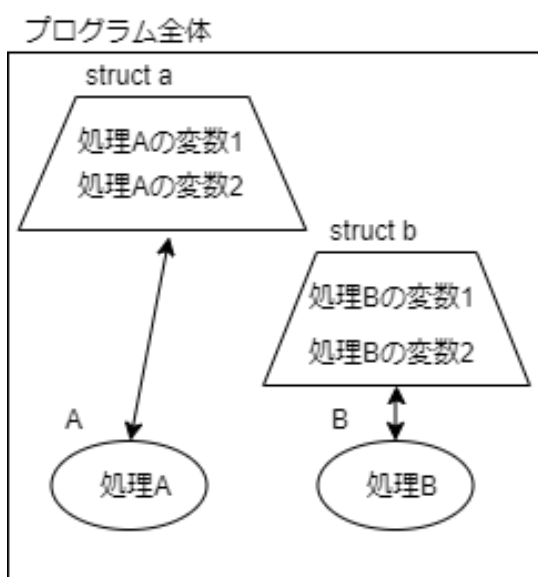
クラスは値だけでなく関数も登録することができるので、「処理」とそれに関係する「値」をひとまとめにし、新たな型名をつけて何に使われるものなのかわかりやすくすることができます。そしてプログラムは「値」と「処理」でできているものですから、それらをひとまとめにできることによって、これまで関数では処理しか、構造体では値しか分割できなかったプログラムが、自由に分割できるようになります。



バラバラな値、処理



関数で処理のみ分割し名付け



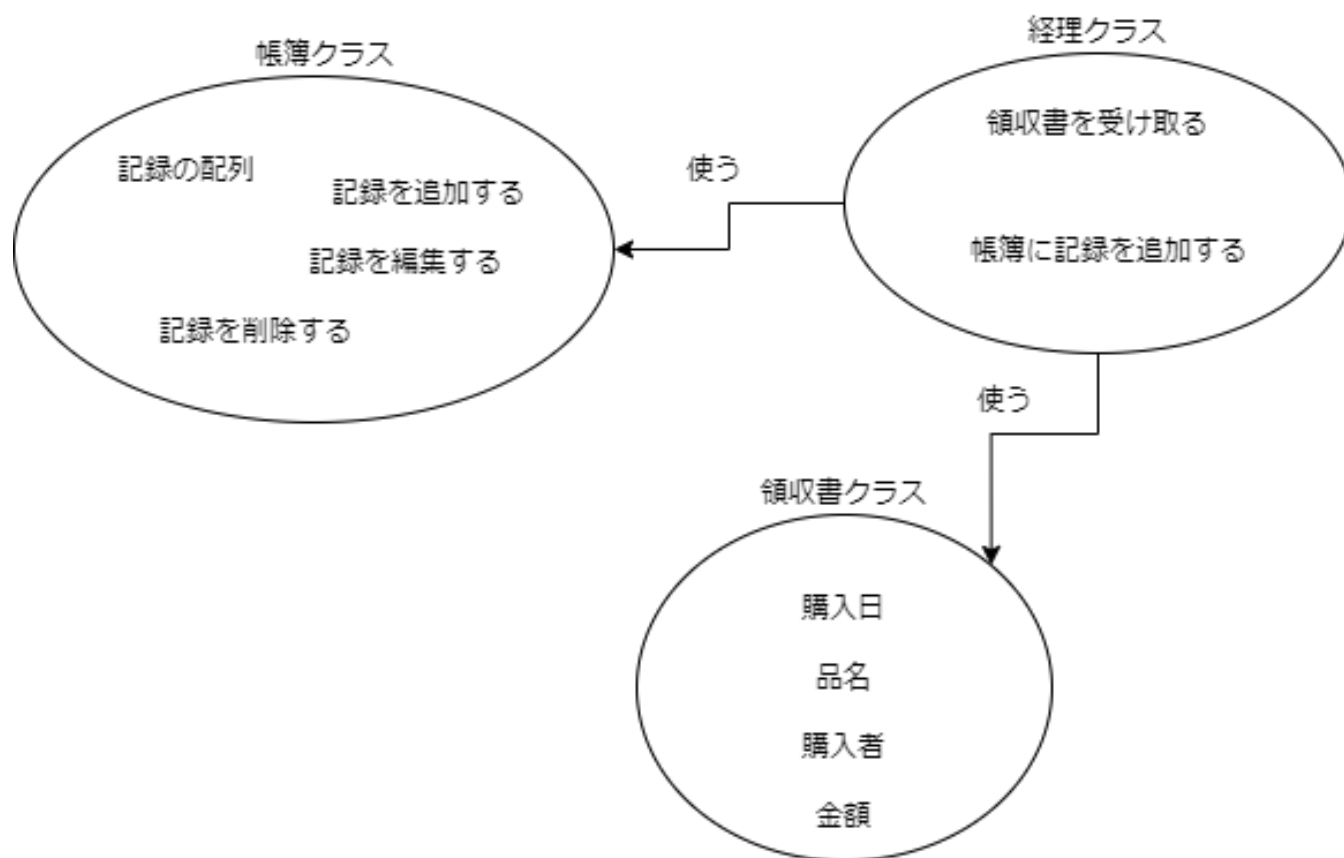
関数と構造体で  
処理と値を分割して名付け  
処理Aとそれに関係する構造体a、処理B  
とそれに関係する構造体bをまとめた



関数と値をクラスで  
ひとまとめにして名付け  
クラスをつかって値と処理をまとめた

クラスは自由にプログラムを分割できるため、どこで分割するかが重要になります。このとき、「何らかの概念」ごとにクラスを分けていくことで様々なメリットがあります。

経理システムを作る際は以下の図のように経理に関わる物を分割してクラスを作り、それぞれが持つ値と処理を実装していきます。(以下の図では末尾が単語のものが値、動詞のものが処理になっています)



経理システムであれば領収書を表す「領収書クラス」、帳簿を表す「帳簿クラス」、領収書をもとに帳簿に登録する橋渡しをする「経理クラス」の3つに分けられます。このようにクラスで物事を分割してプログラムすることで、現実にあるシステムのようにプログラムを構築していくことができます。

このようにプログラムで何かをするとき、プログラムしたいものを概念ごとに値と処理に分け、それぞれクラスにし、それらを組み合わせてプログラムすることによって現実にある仕組みのようにプログラムを構築できる上に、プログラムが何処で分割されるのかが明確になるため、複数人で同時に開発をすすめることが用意になります。さらにクラスに用意されている便利な機能によってif文・switch文の数を減らしたり、プログラムにおける様々なミスを防いだり、記述量を減らしながら開発することができるため、1人で開発する際もクラスには様々なメリットがあります。

このように何らかの概念を表現した「モノ」と「モノ」を組み合わせることでプログラムを作っていく手法を「オブジェクト指向プログラミング」と言います。

クラスはArduinoのライブラリ内部でも使われています。中でもSerial関係はクラスに関する様々な機能を駆使しており、Arduinoのユーザーはもちろん、新たに通信関係のライブラリを作る開発者も簡単にプログラムできるように作られています。

## クラスの作成

構造体に含まれる要素: メンバ ← 前回

クラスに含まれる要素: メンバ

クラスに含まれる変数: メンバ変数

クラスに含まれる関数: メンバ関数(メソッド)

と呼びます。

一般化すると以下の通り

```
class [クラス名]
{
public:
    //メンバ関数

private:
    //メンバ変数

};
```

## アクセス修飾子

クラスの{}内にある「public:」、「private:」:を「アクセス修飾子」と呼びます。

```
class CSample
{
public:
    void function() {
        // 関数の処理
    }
private:
    int m_num;
};
```

アクセス修飾子

## インスタンス化

クラスのインスタンス化は、構造体と同様です。

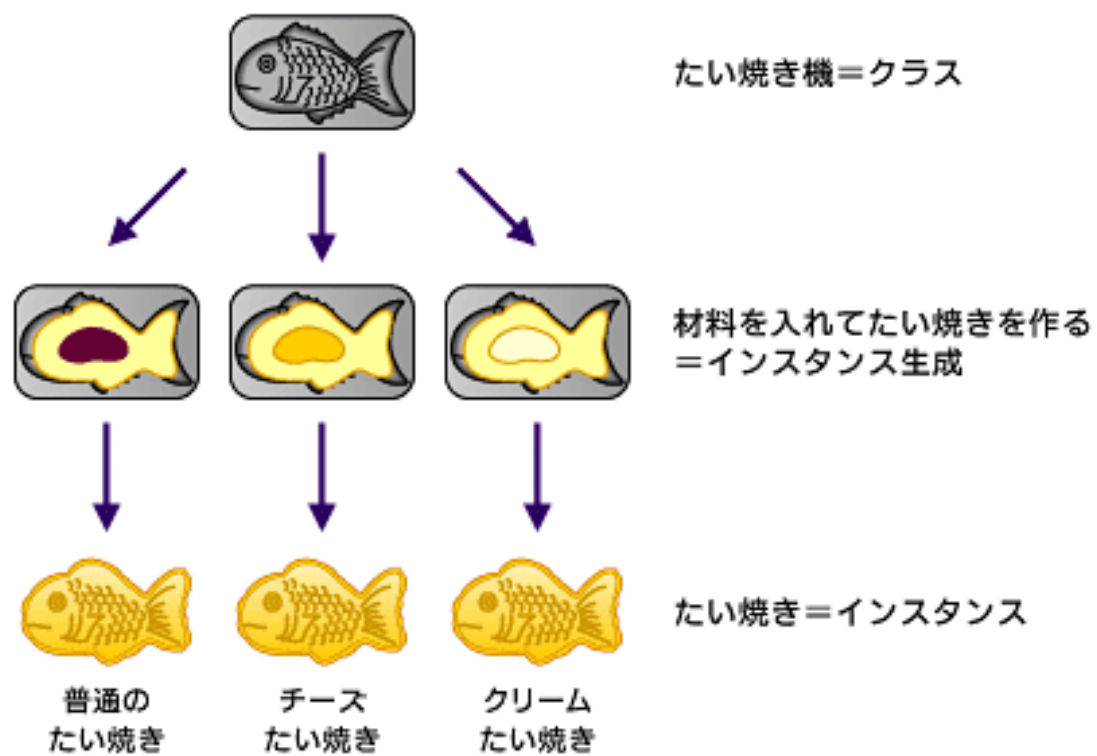
```
[クラス名][インスタンス名];
```

## コード例

```
class CSample
{
public:
    void function() {
        // 関数の処理
    }

private:
    int m_num;
};

CSample obj; // インスタンス化
```



↑ クラスとインスタンスの関係

## 使用例 入力した数字をそのまま返すプログラム

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

// クラス宣言
class CSample
{
public:
    void set(int num) // m_numに値を設定する関数
    {
        m_num = num;
    }
    int get() // m_numの値を取得する関数
    {
        return m_num;
    }
private:
    int m_num;
};

int main()
{
    CSample obj; // CSampleをインスタンス化
    int num;

    cout << "整数を入力: ";
    cin >> num;

    obj.set(num); // CSampleのメンバ変数をセット
    cout << obj.get() << endl; // メンバ変数の値を出力

    return 0;
}
```



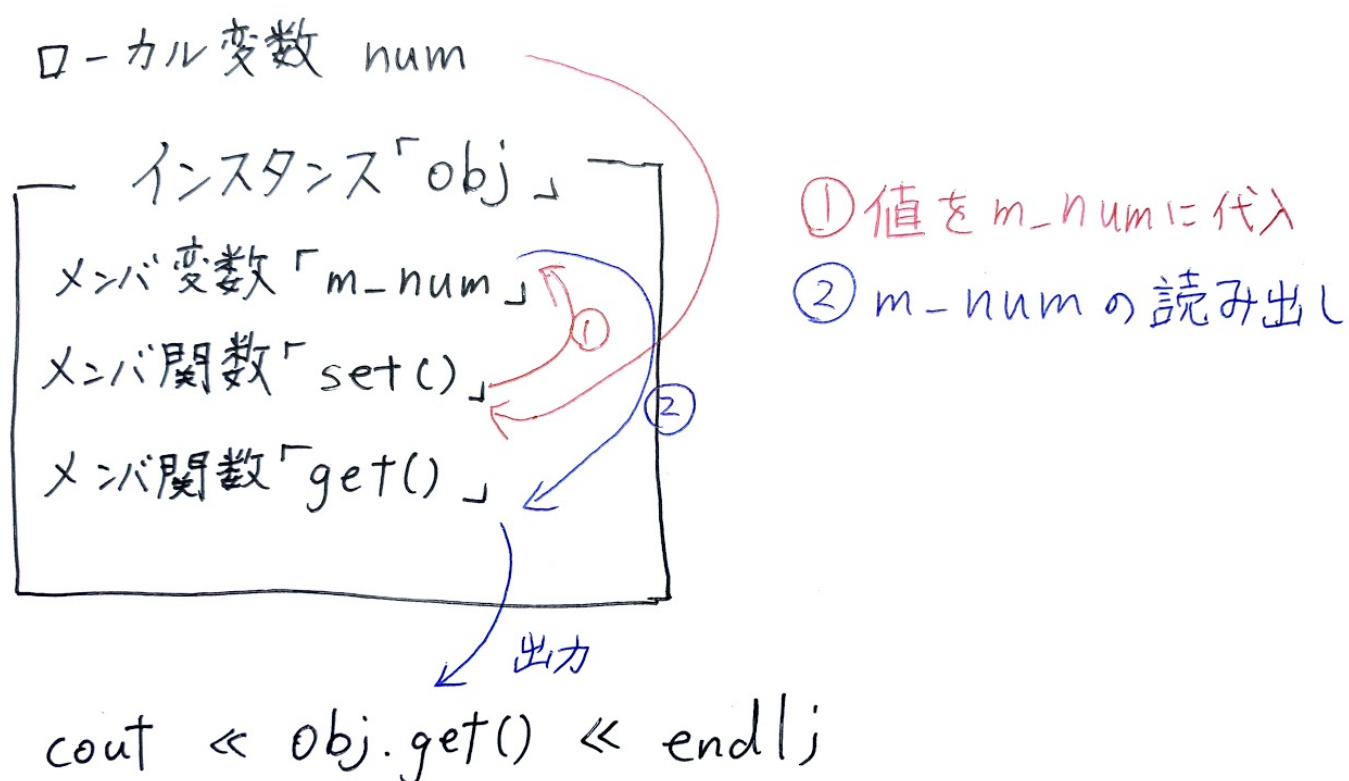
## 出力例

整数を入力: 2

2

## 以上のプログラムの処理の流れ

このプログラムでは、ローカル変数numに格納した整数を、set関数でメンバ関数m\_numに格納しています。その後、get関数でそれを戻り値にしています。



## メンバ関数の使い方

構造体と同様です。

## 例

```
obj.get() //インスタンス名.メンバ関数名();
```



## 複数のインスタンス

### コード例

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

// クラス宣言
class CSample
{
public:
    void set(int num) // m_numに値を設定する関数
    {
        m_num = num;
    }
    int get() // m_numの値を取得する関数
    {
        return m_num;
    }
private:
    int m_num;
};

int main()
{
    CSample obj1; // CSampleをインスタンス化
    CSample obj2; // CSampleをインスタンス化
    int num;

    obj1.set(1); // CSampleのメンバ変数をセット
    obj2.set(2); // CSampleのメンバ変数をセット
    cout << obj1.get() << endl; // メンバ変数の値を出力
    cout << obj2.get() << endl; // メンバ変数の値を出力

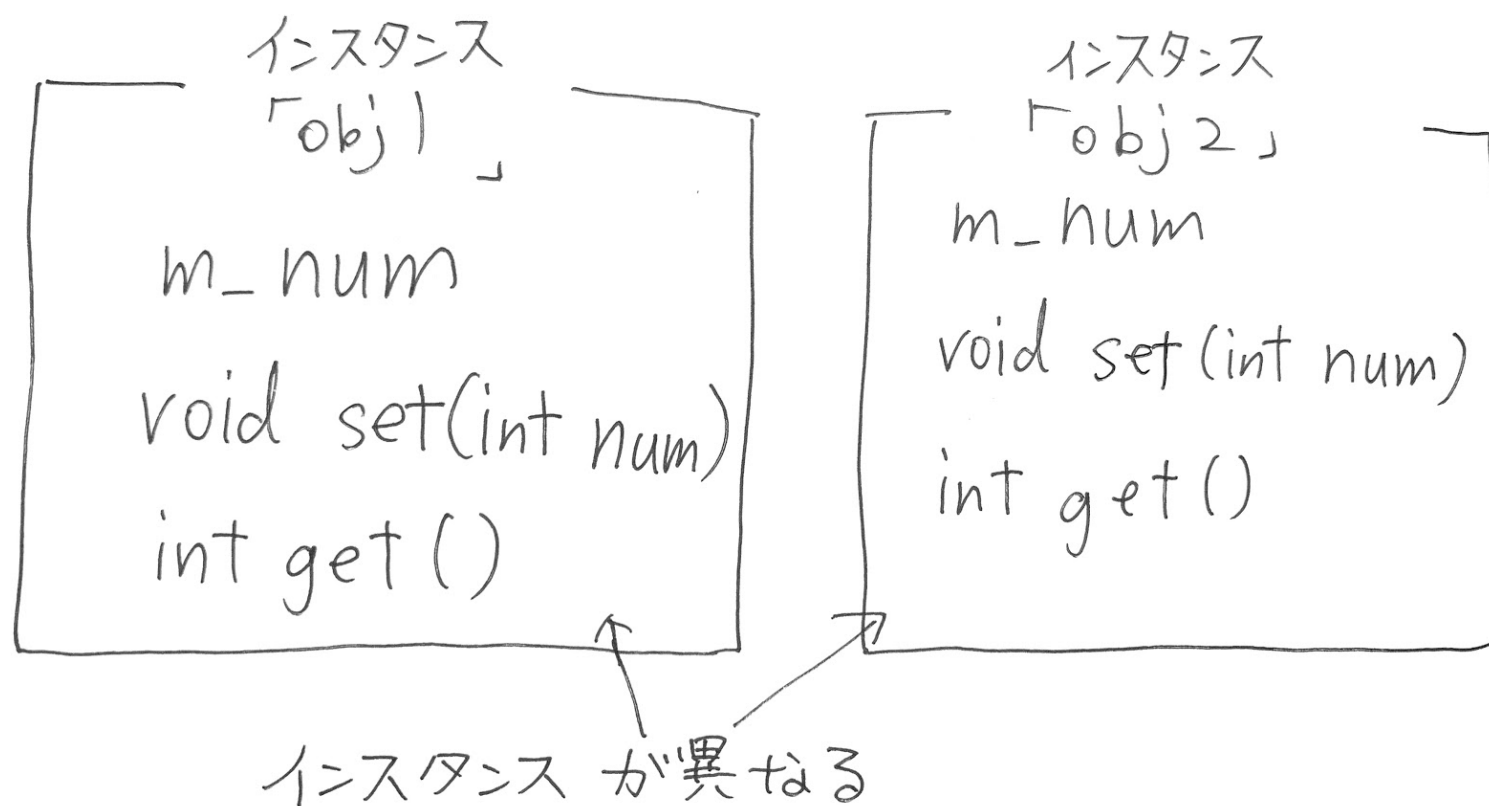
    return 0;
}
```

## 出力

1

2

以上のプログラムの様に同じクラスで複数のインスタンスを生成できます。同じ名前の変数、関数であってもインスタンスが違えば別物ということです。



## 問題 1

以下のプログラムを実行すると「実行結果1」のように出力されます。このプログラム内のクラスKeisanに、メンバ変数a,bの引き算を帰すメンバ関数sub()を追加して「実行結果2」のように出力されるように変更してください。↓まずはこれをエディタにコピー&ペースト

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

class Keisan {
public:
    int a;
    int b;
    int add()
    {
        return a + b;
    }
};

int main() {
    Keisan k;
    k.a = 4;
    k.b = 3;
    cout << k.a << " + " << k.b << " = " << k.add() << endl;
    return 0;
}
```

### 実行結果1

```
4 + 3 = 7
```

### 実行結果2

```
4 + 3 = 7
4 - 3 = 1
```

## 問題 2

以下のプログラムを改造し、クラスMinMaxのメンバ関数max(),min()の引数の数を3にし、期待される実行結果にならい、3つの数の最大値・最小値を出せるようにプログラムを改造してください。

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

class MinMax {
public:
    int max(int n1, int n2) {
        if (n1 > n2) {
            return n1;
        }
        return n2;
    }
    int min(int n1, int n2) {
        return -max(-n1, -n2);
    }
};

int main() {
    MinMax m;
    int a = 4;
    int b = 2;
    cout << a << "と" << b << "のうち、最大のものは" << m.max(a, b) << endl;
    cout << a << "と" << b << "のうち、最小のものは" << m.min(a, b) << endl;
    return 0;
}
```

## 実行結果1

4と2のうち、最大のものは4  
4と2のうち、最小のものは2

## 実行結果2

4と2と8のうち、最大のものは8  
4と2と8のうち、最小のものは2

問題1、2のソースコードの出典: 「一週間で身につくC++言語の基本 C++入門編」

## コンストラクタ、デストラクタ

コンストラクタ、デストラクタ: クラスに含まれる特殊なメソッド

下のコード例を元に説明致します。

### コード例

```
#include <iostream>

using namespace std;

// 自動車クラス
class CCar {
public:
    // コンストラクタ
    CCar() : m_fuel(0), m_migration(0)
    {
        cout << "CCarオブジェクト生成" << endl;
    }
    // デストラクタ
    ~CCar()
    {
        cout << "CCarオブジェクト破棄" << endl;
    }
    // 移動メソッド
    void move()
    {
        // 燃料があるなら移動
        if (m_fuel >= 0) {
            m_migration++; // 距離移動
            m_fuel--;      // 燃料消費
        }
        cout << "移動距離:" << m_migration << endl;
        cout << "燃料" << m_fuel << endl;
    }
    // 燃料補給メソッド
    void supply(int fuel)
    {
        if (fuel > 0) {
            m_fuel += fuel; // 燃料補給
        }
        cout << "燃料" << m_fuel << endl;
    }
private:
    int m_fuel;        // 燃料
    int m_migration;   // 移動距離
};

int main() {
    CCar c;
    c.supply(8); // 燃料補給
    c.move();    // 移動
    c.move();    // 移動
    return 0;
}
```

## 出力

```
CCarオブジェクト生成
燃料8
移動距離:1
燃料7
移動距離:2
燃料6
CCarオブジェクト破棄
```

長いコード例なので気分を悪くされた方もいらっしゃるかも知れませんが、今回理解して頂きたいのはコード例の冒頭、**赤字になっているコンストラクタとデストラクタの部分**のみです。

### ・コンストラクタ

コンストラクタは、そのクラスをインスタンス化したときに、自動的に呼び出される特別なメンバ関数です。先程のコード例の7~10行目にあるCCar()メソッドがコンストラクタです。ですから、インスタンスが生成された時、CCar()関数内が実行され「CCarオブジェクト生成」と表示されます。尚、コンストラクタの名前はクラス名と同じにします。

7行目に以下の記述が有ります。

```
CCar() : m_fuel(0),m_migration(0)
```

ここでは、メンバ変数の初期化処理を行っており、m\_fuelに0を、m\_migrationに0をそれぞれ代入しています。

つまり、インスタンスが生成されコンストラクタが実行されると、メンバ変数の初期化処理やコンストラクタの{}が実行されます。

### ・コンストラクタの初期化処理

```
クラス名() : メンバ変数1(初期値1),メンバ変数2(初期値2)...
```

このように、間を,(コンマ)で区切り、()内に値を入れると、メンバ変数をその値で初期化することができます。なお、メンバ変数の並び順は、定義順にすることが推奨されています。



その順番にしたがっていなくても文法上は間違いではありませんが、そのほうがソースコードが理解しやすくなりますので、そう心がけましょう。

このような方法でメンバ変数を初期化するのは、あくまでも、その値が定数であったり、外部からコンストラクタの引数として渡されるパラメータである場合（詳細は後述）に限られます。値を決めるのに何らかの処理が必要な場合は、あくまでもコンストラクタの処理の中で値を設定してもかまいません。

また、通常のメソッド同様、コンストラクタの中で諸処理は、`{}`の中に記述されます。実行結果からみてもわかるとおり、ここに記述された処理は、特に呼び出されなくても自動的に実行されることがわかります。ここでは、インスタンス生成時の様々な初期化処理が行われます。

## ・デストラクタ

次に、デストラクタについて説明します。デストラクタとは、クラスのインスタンスが解放されるときに、解放の直前で自動的に呼び出されます（下図参照）。解放されるタイミングは、そのインスタンスのスコープを抜けるときです。例えば、ある関数内でインスタンス化した場合、その関数を抜ける段階で解放されます。この例でも、`main()`の処理が終わるときにデストラクタが呼ばれていることがわかります。

デストラクタの名前は、クラス名の先頭に `~`(チルダ) を付けたものになります。したがって、このサンプルの場合、クラス名が `CCar` ですから、デストラクタの名前は、`~CCar()` になります。また、これ以外の名前を使うことはできません。

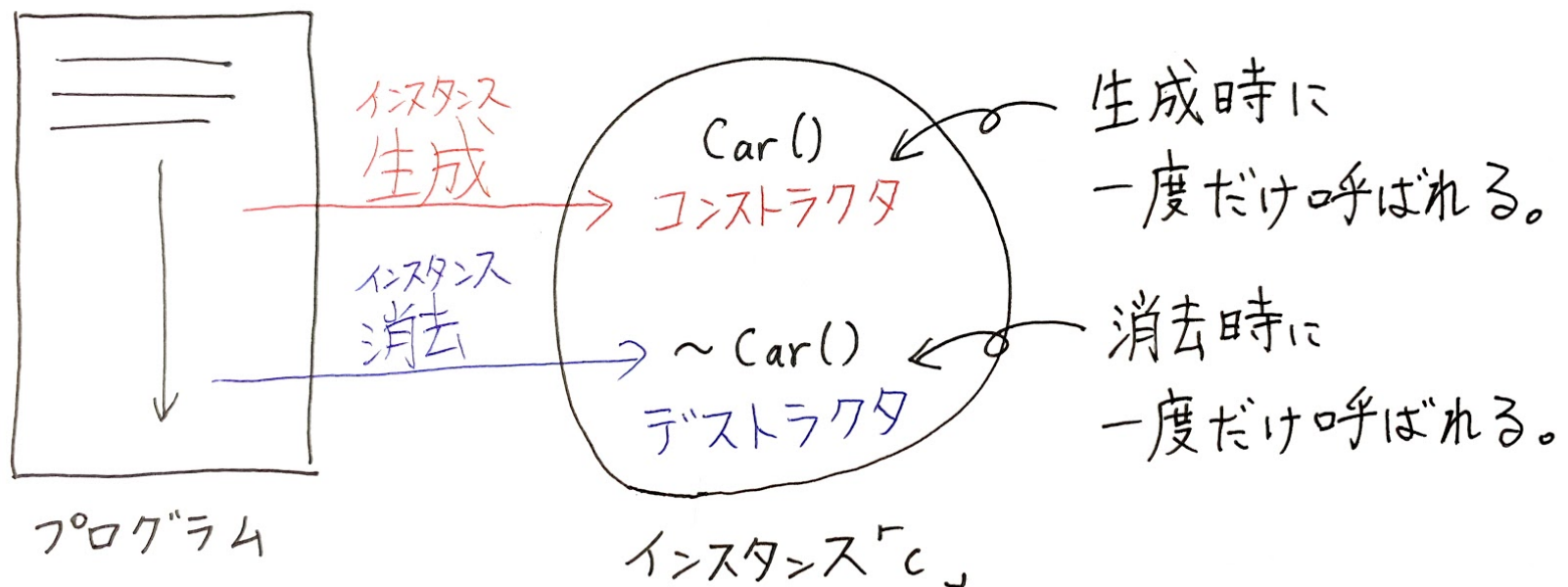
デストラクタで行う処理は自由ですが、基本的には終了処理を行うことが一般的です。終了処理というのは、動的に確保されたメモリの解放や、オープンされたままのファイルをクローズすることなどです。デストラクタの処理の実行を終えた時点で、そのクラスのインスタンスはメモリから解放されて無くなってしまいます。



コンストラクタやデストラクタは、必要がなければ作る必要はありません。省略した場合でも、コンパイラが自動的にコンストラクタおよびデストラクタを生成しています。ただ、そこで何をしているのかを知ることはできません。「メンバ変数」などの項で示していたサンプルにコンストラクタとデストラクタが省略されていても実行できたのはことためです。

出典: 「一週間で身につくC++言語の基本 C++入門編」

クラス名「Car」



## 問題

以下のプログラム内のクラスにコンストラクタとデストラクタをつけ、インスタンス生成時に「インスタンス生成」、インスタンス消去時に「インスタンス消去」と出力されるように変更してください。

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

class MinMax {
public:
    int max(int n1, int n2) {
        if (n1 > n2) {
            return n1;
        }
        return n2;
    }
    int min(int n1, int n2) {
        return -max(-n1, -n2);
    }
};

int main() {
    MinMax m;
    int a = 4;
    int b = 2;
    cout << a << "と" << b << "のうち、最大のものは" << m.max(a, b) << endl;
    cout << a << "と" << b << "のうち、最小のものは" << m.min(a, b) << endl;
    return 0;
}
```

## 出力例

```
インスタンス生成
4と2のうち、最大のものは4
4と2のうち、最小のものは2
インスタンス消去
```

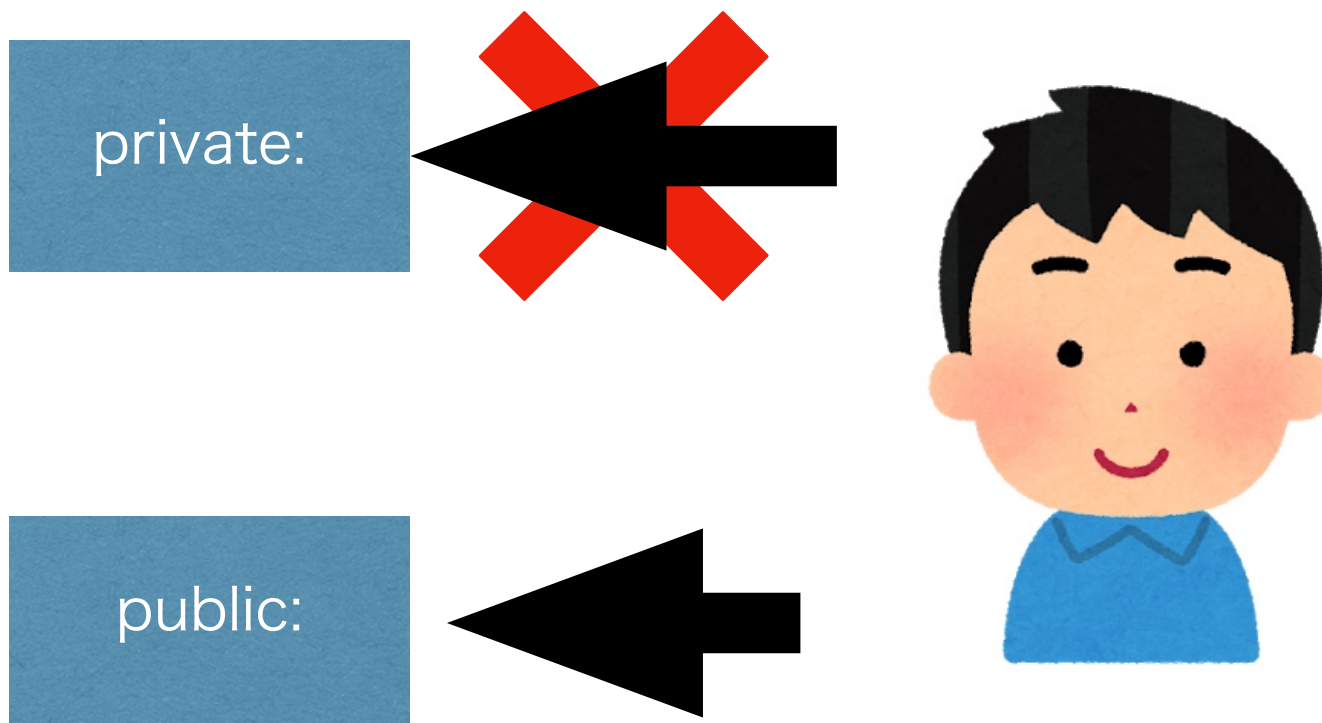
## PUBLICとPRIVATE

PrivateとPublicというワードは何なのでしょう？

Privateはクラス内でしか使えず、クラス外でアクセスしようとするエラーがでます。

Publicはクラス外でも使えます。

クラスの外の  
ジョージ君



```
class CSample
{
public:
    void function() {
        // 関数の処理
    }

private:
    int m_num;
};

CSample obj; // インスタンス化
obj.function( ); //publicなのでオッケー！
int x=obj.m_num; //privateなのにアクセスしてエラー！！
```

何故こんなめんどくさいことをするのでしょうか？

全部publicに置いておけばいいのでは？

## カプセル化

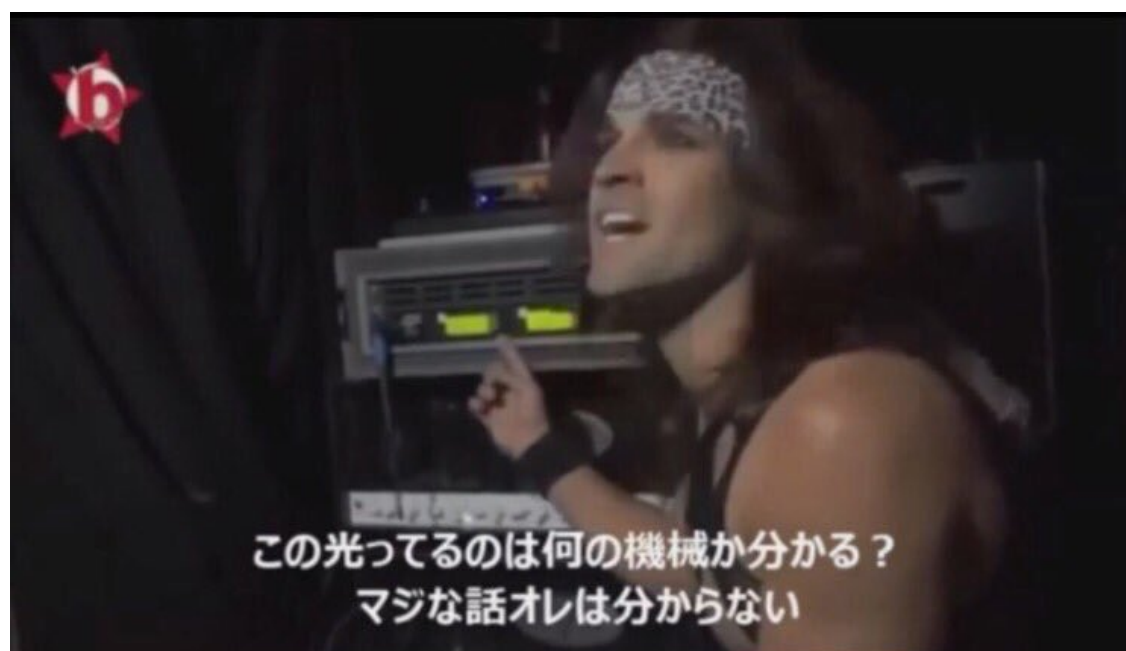
(情報隠蔽とも言うらしい)

僕たちがSerial.println()

を使う時、その中で変数がどう動いているかなんて気にしないと思います。

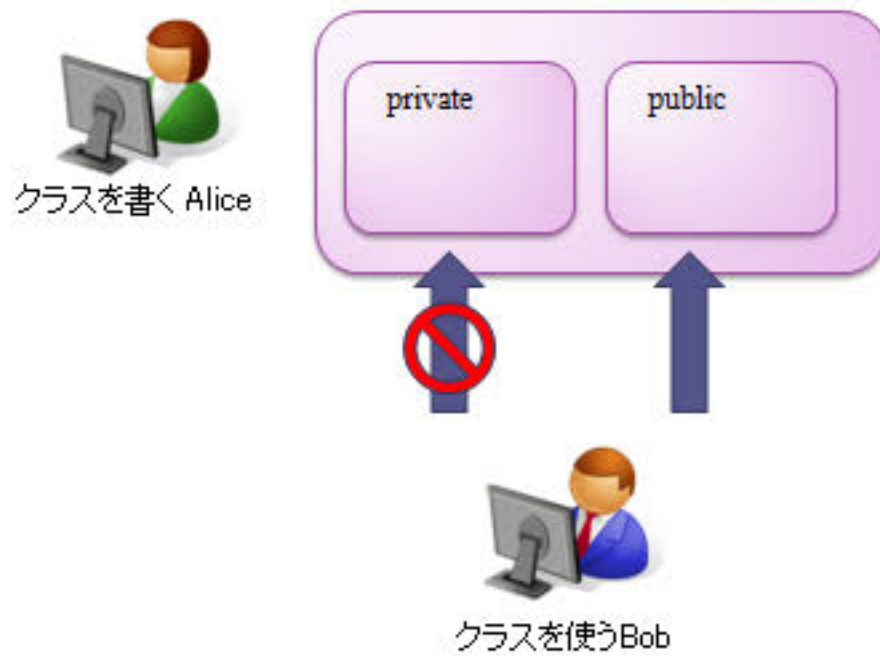
それは中の動きが隠蔽されているからです。

よく考えなくても動くようにしてくれているのですね



また、アクセスを制御するという意味もあります。

## アクセス制御・・・privateとpublic



使う人には書き換えて欲しくないパラメータ、LEDの流れるスピードなどをprivateに置いておき

変更されないようにしておけます。

あくまでも外から変更できないだけで、クラスの中では変更できます。



外部から変更したい変数を作る時はどうしたらいいのでしょうか?

Publicに変数を置くこともできますが、変数をprivateに置いた上でgetterとsetterという物を作ることで便利ことがあります。

getter setterを使って、関数から値を変更します。

```
class CSample
{
public:
    void setNum(int input){
        m_num=input;
    }
    int getNum(){
        return m_num;
    }
private:
    int m_num;
};
```

setterは整数を受け取り、その値をprivateなm\_numに代入します。

getterはm\_numの値を返します。

これを行うことでm\_numの値が変更されたときに、何らかの処理をすることが出来ます。

(setter関数を通すため)

ただしgetter、setterを使う必要がないときは普通にpublicにメンバ変数をおいたほうがプログラムが単純になってわかりやすいです。

## 問題

x座標とy座標を記録するクラスPointのgetterとsetterを作ってください。

```
class Point{  
    private:  
        int x;  
        int y;  
}
```



# 継承

## 継承とは

### ・継承の概念

C++言語に限らず、オブジェクト指向言語に備わっている重要な特性の1つに継承（インヘリタンス）があります。継承は、あるクラスのメンバを、他のクラスに引継ぐ（継承させる）という効果があります。

では、実際に継承とはどういうものか、さらに詳しく説明しましょう。すでに述べたように、クラスは、インスタンス、およびオブジェクトの「設計図」です。自動車の例を出すのならば、自動車の設計図がクラス、実際に工場で生産された自動車本体が、インスタンス、およびオブジェクトということになります。

### ・親クラスと子クラス



このように、基本となるクラスの性質を受け継ぎ、独自の拡張をすることを、オブジェクト指向では、継承（けいしょう）と呼びます。継承のもととなるクラスのことを、親クラス、スーパークラスなどと呼びます。それにたいし、親クラスの機能を継承し、独自の機能を実装したクラスのことを、子クラス、もしくは、サブクラスと呼びます。

前述の自動車の例で言うのならば、車クラスが親クラス、トラックや救急車などが、サブクラスということになります。

## ・ 継承の仕方

では実際に、この継承を実装してみましょう。ここでは、前述の、自動車と、救急車の例をプログラムにしてみます。少し長いですが、以下のプログラムを実行してみてください。

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

// 自動車クラス
class CCar {
public:
    // コンストラクタ
    CCar() : m_fuel(0), m_migration(0)
    {
        cout << "CCarオブジェクト生成" << endl;
    }
    // デストラクタ
    virtual ~CCar()
    {
        cout << "CCarオブジェクト破棄" << endl;
    }
    // 移動メソッド
    void move()
    {
        // 燃料があるなら移動
        if (m_fuel >= 0) {
            m_migration++; // 距離移動
            m_fuel--;      // 燃料消費
        }
        cout << "移動距離:" << m_migration << endl;
        cout << "燃料" << m_fuel << endl;
    }
    // 燃料補給メソッド
    void supply(int fuel)
    {
        if (fuel > 0) {
            m_fuel += fuel; // 燃料補給
        }
        cout << "燃料" << m_fuel << endl;
    }
private:
    int m_fuel;          // 燃料
    int m_migration;     // 移動距離
};
```

```

#include <iostream>
using namespace std;

// 自動車クラス
class CCar {
public:
    // コンストラクタ
    CCar() : m_fuel(0), m_migration(0)
    {
        cout << "CCarオブジェクト生成" << endl;
    }
    // デストラクタ
    virtual ~CCar()
    {
        cout << "CCarオブジェクト破棄" << endl;
    }
    // 移動メソッド
    void move()
    {
        // 燃料があるなら移動
        if (m_fuel >= 0) {
            m_migration++; // 距離移動
            m_fuel--;      // 燃料消費
        }
        cout << "移動距離:" << m_migration << endl;
        cout << "燃料" << m_fuel << endl;
    }
    // 燃料補給メソッド
    void supply(int fuel)
    {
        if (fuel > 0) {
            m_fuel += fuel; // 燃料補給
        }
        cout << "燃料" << m_fuel << endl;
    }
private:
    int m_fuel;          // 燃料
    int m_migration;     // 移動距離
};

```

```

class CAmbulance : public CCar {
public:
    // コンストラクタ
    CAmbulance() : m_number(119)
    {
        cout << "CAmbulanceオブジェクト生成" << endl;
    }
    // デストラクタ
    virtual ~CAmbulance()
    {
        cout << "CAmbulanceオブジェクト破棄" << endl;
    }
    // 救急救命活動
    void savePeople() {
        cout << "救急救命活動" << endl << "呼び出しは" <<
m_number << "番" << endl;
    }
private:
    int m_number;
};

int main() {
    CCar c;
    c.supply(10);    // 燃料補給
    c.move();        // 移動
    c.move();        // 移動
    CAmbulance a;
    a.supply(10);
    a.move();
    a.savePeople();
    return 0;
}

```

## 出力結果

```
CCarオブジェクト生成
燃料10
移動距離:1
燃料9
移動距離:2
燃料8
CCarオブジェクト生成
CAmbulanceオブジェクト生成
燃料10
移動距離:1
燃料9
救急救命活動
呼び出しは119番
CAmbulanceオブジェクト破棄
CCarオブジェクト破棄
CCarオブジェクト破棄
```

あるクラスが、別のクラスを親クラスとするときの定義は、

親クラスを継承した、子クラスの定義の方法

class 子クラス名 : public 親クラス名

のように定義します。つまり、CAmbulanceクラスは、CCarクラスを継承したクラスです。そのため、CCarクラスが親クラス、CAmbulanceクラスが、子クラスという組み合わせになります。なお、ambulanceとは、英語で救急車を表す言葉であり、このクラスは、「救急車クラス」ということになります。

このことから、publicメンバである、move()メソッド、および、supply()メソッドを利用することができます。ただ、CCarクラスのprivateメンバ変数である、m\_fuelおよび、m\_migrationには、直接アクセスすることはできません。

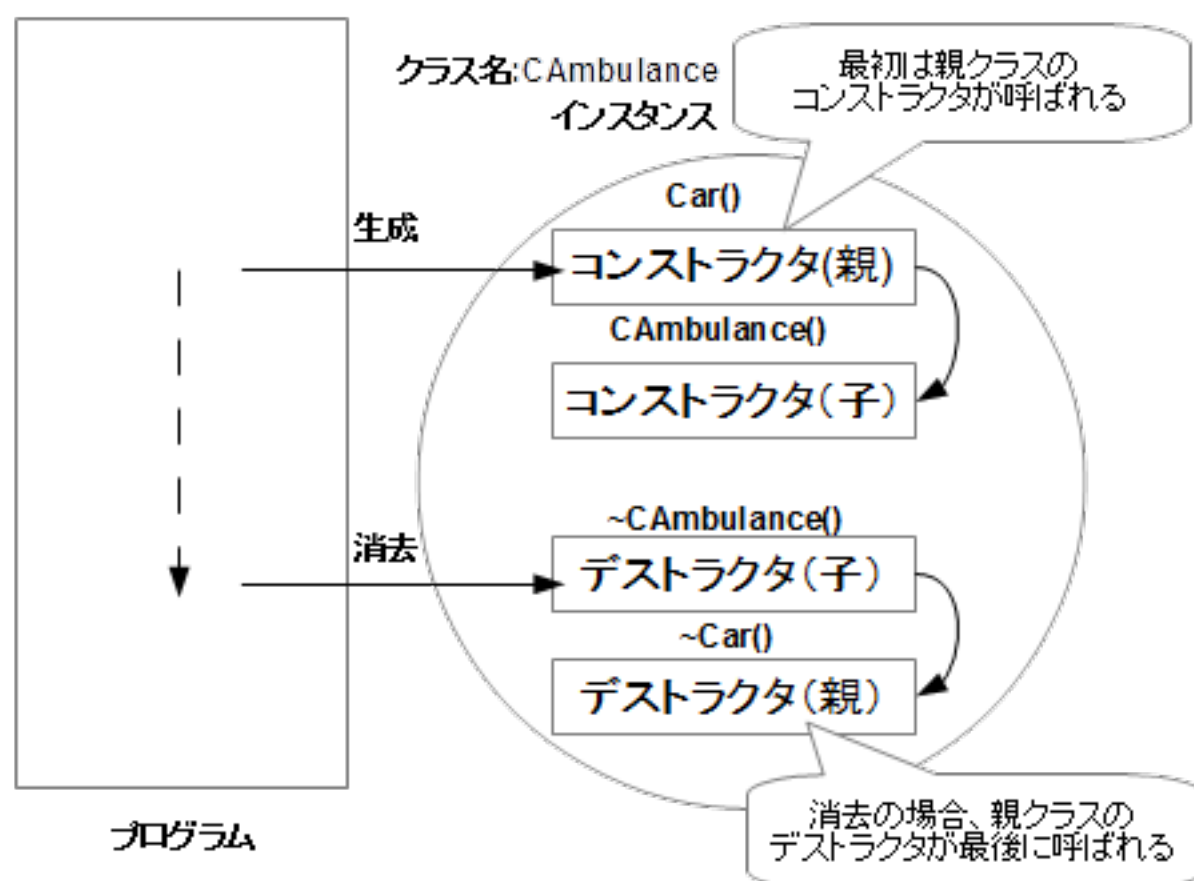
また、CAmbulanceクラスは、さらに独自のメンバである、メンバ変数のm\_number、および、savePeopleメソッドは、CAmbulanceクラスで利用することができますが、CCarクラスでは利用することはできません。

## ・サブクラスのコンストラクタとデストラクタ

次に、子クラスのコンストラクタとデストラクタの組み合わせについて注目してみましょう。プログラムをみてもわかるとおり、この子クラスにも、親クラスと同様に、コンストラクタ(CAmbulance)および、デストラクタ(~CAmbulance)が定義されています。

また、実行結果からもわかるとおり、実はサブクラスが生成される際、子クラスのコンストラクタが実行される前に、親クラスのコンストラクタが実行され、逆にdeleteでインスタンスが破棄されるときは、子クラスのデストラクタが実行され、そのあと親クラスのデストラクタが実行されることがわかります。

このように、継承が利用された場合、親クラスのコンストラクタ・デストラクタも利用されることがわかります。



なお、CCarおよび、CAmbulanceクラスについている、virtual(バーチャル)修飾子に関しては、必ずつけるようにしてください。通常、C++言語では、継承を用いる場合、virtualをデストラクタにつけるように推奨されています。理由は難しいのでここでは省略しますが、

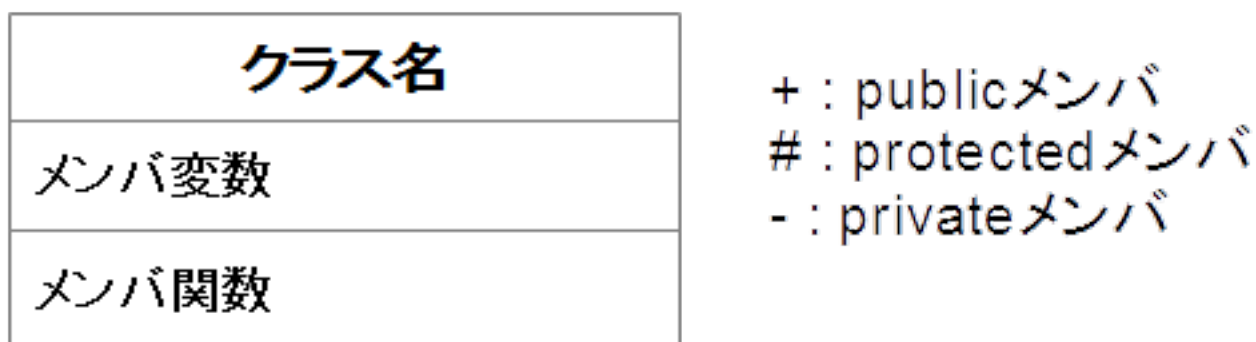


どのようなクラスでも、そのサブクラスが作られる可能性があるので、今後、基本的にクラスのコンストラクタには、`virtual`をつけるようにしましょう。

## ・ UML

なお、プログラムの動作には直接関係はありませんが、ここでC++に限らず、オブジェクト指向言語の設計に用いられる、UMLというツールの中にある、クラス図を用いて、このクラスでこのクラスを表記してみることにします。

まず、クラス図ですが、クラスを以下のような方法で表記します。

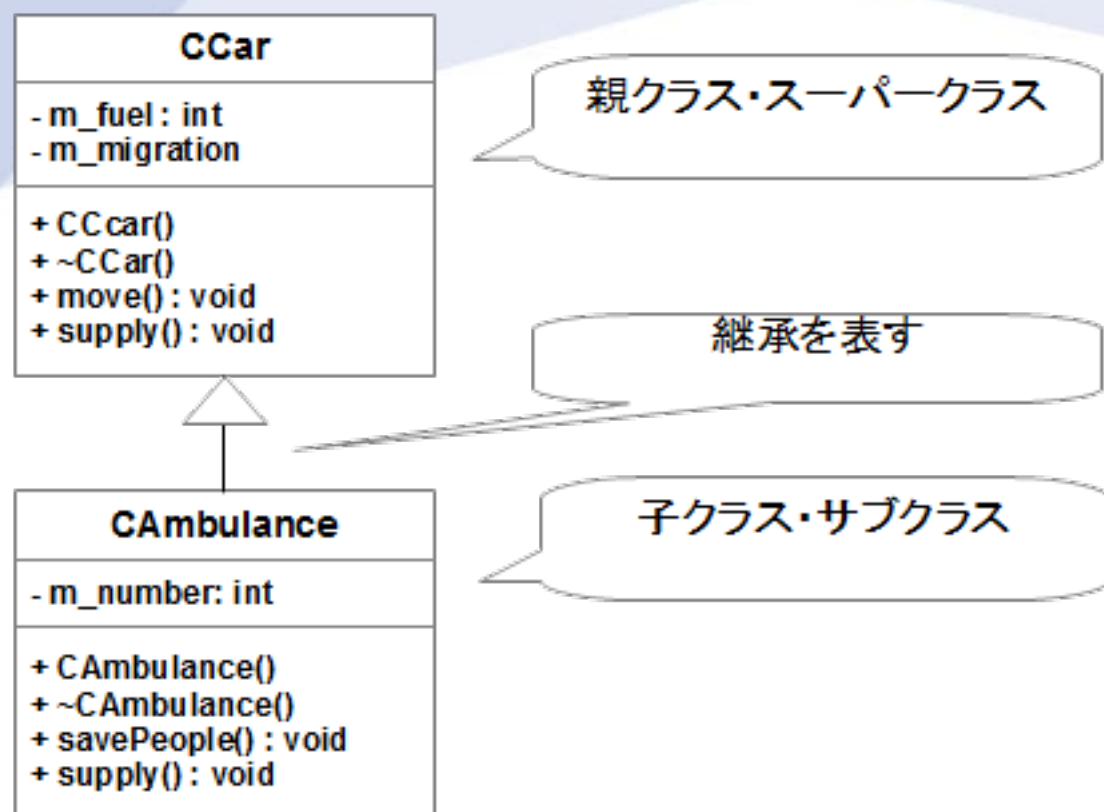


このように、3つに区切られた四角形の中に、上からクラス名、メンバ変数（属性）、メンバ関数（操作）を記入します。（クラス名以外は省略可能）複数のクラスが存在する場合には、これにさらにクラス間の関係性が記入されます。また、メンバ変数およびメンバ関数の前には、+、-、#といった記号を記述します。これらはそれぞれ、`public`、`private`、`protected`を表します。

このようにして、クラスを表記し、さらに、クラス間の関係性を表記することも可能です。継承もその関係性の一つです。実際にCCarクラスと、CAmbulanceクラスを用いて表記する



と、以下のようになります。



図のように、UMLのクラス図で継承を表記する場合親クラスと子クラスの間を△のついた線で結びます。△の上には親クラス（スーパークラス）、下には子クラス（サブクラス）がきます。

## protectedメンバ(後の項目でより詳しく扱います)

すでに学んだとおり、privateメンバは、同一クラス内から、publicメンバは、クラス内・外を問わず、アクセスすることができます。しかし、三つ目のprotectedについては、詳しく説明していませんでしたので、この場で詳しく説明します。

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

// 二次元ベクトルクラス
class Vector2D {
protected:
    int m_x;
    int m_y;
public:
    // コンストラクタ
    Vector2D()
    {
        init();
    }
    // 値の設定
    void setValue(int x, int y)
    {
        m_x = x; m_y = y;
    }
    // X座標の取得
    int getX()
    {
        return m_x;
    }
    // Y座標の取得
    int getY()
    {
        return m_y;
    }
protected:
    // 初期化
    void init()
    {
        m_x = 0; m_y = 0;
    }
};
```

```

class Position2D : public Vector2D {
public:
    // 位置のリセット
    void resetPosition() {
        init();
    }
    // 移動
    void move(int dx, int dy)
    {
        m_x += dx;
        m_y += dy;
    }
};

int main(int argc, char** args) {
    Position2D p;
    p.setValue(1, 1);
    p.move(2, 3);
    cout << "p:(" << p.getX() << "," << p.getY() << ")" << endl;
    p.resetPosition();
    cout << "p:(" << p.getX() << "," << p.getY() << ")" << endl;
    return 0;
}

```

## 出力

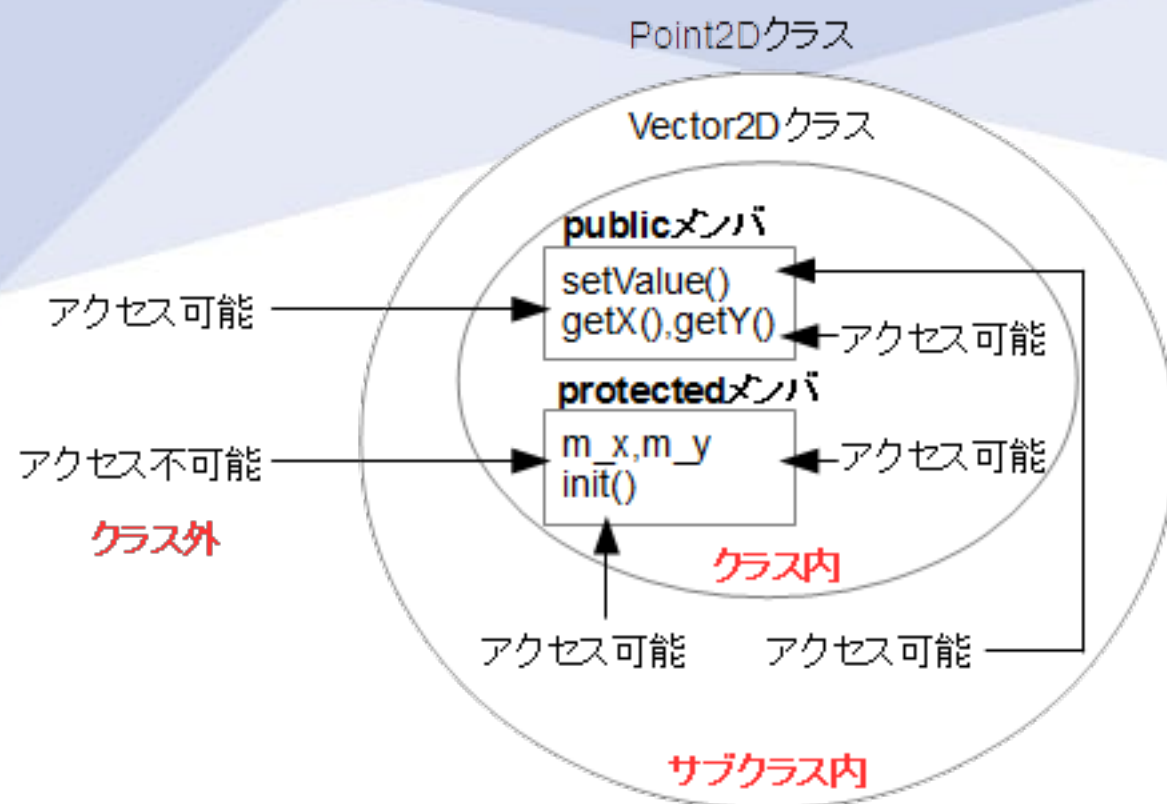
```

p:(3,4)
p:(0,0)

```

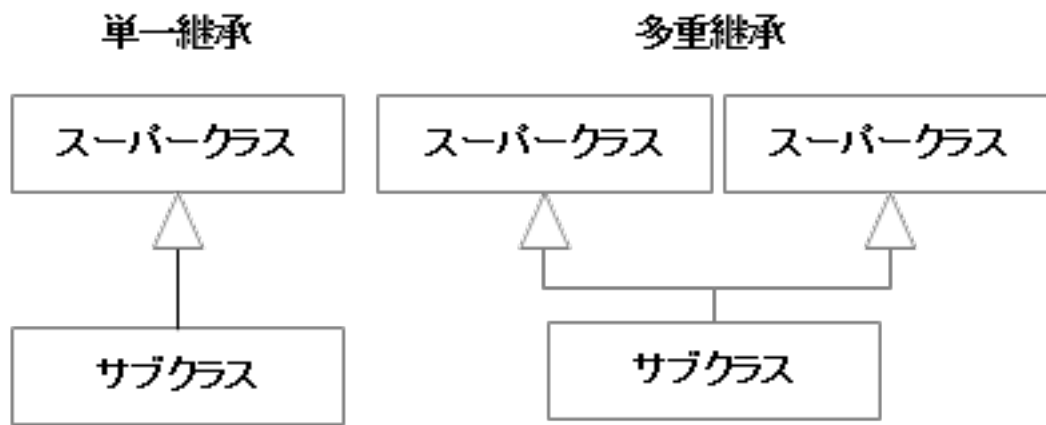
Position2Dクラスは、Vector2Dクラスを継承しています。そのため、親クラスのpublicなメンバである、setValue(),getX(),getY()関数を利用できます。また、Vector2Dで定義されているメンバ変数のm\_x、m\_yおよび、メンバ関数init()にも、クラス内からアクセスすることが可能です。

ただ、protectedメンバは、privateメンバ同様、クラス外からのアクセスはできません。つまり、protectedメンバは、子クラスから見ればpublicとように、クラス外から見ればprivateのようにふるまうことができるのです。このように、サブクラスのみにアクセスを許すメンバには、protected修飾子をつけます



## 多重継承

ここまで、C++言語の継承について説明してきましたが、今までの例では、親クラスが一つしか存在しませんでした。このように、親クラスが一つしかないような継承の仕方を、単一継承(たんいつけいしょう)と言います。ただ、C++では、ひとつのクラスに複数の親クラスを設定することができます。これを、多重継承(たじゅうけいしょう)と言います。



現在用いられているオブジェクト指向言語は、Java,C#など様々ですが、基本的に多重継承を許しているのはC++のみです。

### ・実装

では、C++で多重継承を実装するにはどのようにすればよいのでしょうか？ そのやり方は、簡単で、以下のような記述方法になります。そのため、現在では、純粹仮想クラスなどのように、多言語のインターフェースに類するような特殊なクラスの継承に用いられるのがほとんどです。なので、基本的に継承を用いる際には、原則的に子クラスは親クラスを原則一つしか持たないようにするように心掛けましょう。

```
class Sub : public SupA,public SupB{  
...  
}
```

このように、継承する親クラスの間を、,(コンマ)で区切れば、複数の親クラスを持つことが可能です。ただ、多重継承を用いるのは、技術的な問題点も多く、あまり推奨されていません。

## 問題 1

以下のプログラムに、飛行機クラスAirplaneを継承した戦闘機クラスFighterを指定されたとおりの仕様で追加し、期待された実行結果通りにプログラムが動くように改造してください。

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

// 飛行機クラス
class Airplane {
public:
    // 飛行する
    void fly() {
        cout << "飛行する" << endl;
    }
};

int main() {
    Fighter f; // 戦闘機クラス
    Airplane a; // 飛行機クラス
    // 飛行機が飛行する
    a.fly();
    // 戦闘機が飛行する
    f.fly();
    // 戦闘機が戦闘する
    f.fight();
    return 0;
}
```

期待される実行結果

```
飛行する← Airplaneクラスのfly()メソッドによる処理
飛行する← Fighterクラスのfly()メソッドによる処理
戦闘する← Fighterクラスのfly()メソッドによる処理
```

### クラスFighterの仕様(メンバ関数)

関数名	戻り値の型	引数	概要
fight	void	なし	「戦闘する」と表示し、改行する。



## 問題 2

以下のプログラムに、基本計算クラスFundCalcを継承した新計算クラスNewCalcを指定されたとおりの仕様で追加し、期待された実行結果通りにプログラムが動くように改造してください。このさい、必要があれば、FundCalcクラスの必要な部分を改造してください。

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

class FundCalc {
private:
    double m_number1;    // 一つ目の数
    double m_number2;    // 二つ目の数
public:
    // コンストラクタ
    FundCalc() : m_number1(0), m_number2(0)
    {
    }
    // 一つ目の数を設定
    void setNumber1(double number)
    {
        m_number1 = number;
    }
    // 二つ目の数を設定
    void setNumber2(double number)
    {
        m_number2 = number;
    }
    // 一つ目の数を設定
    double getNumber1()
    {
        return m_number1;
    }
    // 二つ目の数を設定
    double getNumber2()
    {
        return m_number2;
    }
    // 二つの数の和を出力
    double add()
    {
        return m_number1 + m_number2;
    }
    // 二つの数の差を出力
    double sub()
    {
        return m_number1 - m_number2;
    }
};
```

```

int main() {
    NewCalc n;
    n.setNumber1(10);    // 一つ目の数を設定
    n.setNumber2(2);     // 二つ目の数を設定
    cout << n.getNumber1() << " + " << n.getNumber2() << " = " <<
n.add() << endl;
    cout << n.getNumber1() << " - " << n.getNumber2() << " = " <<
n.sub() << endl;
    cout << n.getNumber1() << " * " << n.getNumber2() << " = " <<
n.mul() << endl;
    cout << n.getNumber1() << " / " << n.getNumber2() << " = " <<
n.div() << endl;

    return 0;
}

```

期待される実行結果の例

```

10 + 2 = 12
10 - 2 = 8
10 * 2 = 20
10 / 2 = 5

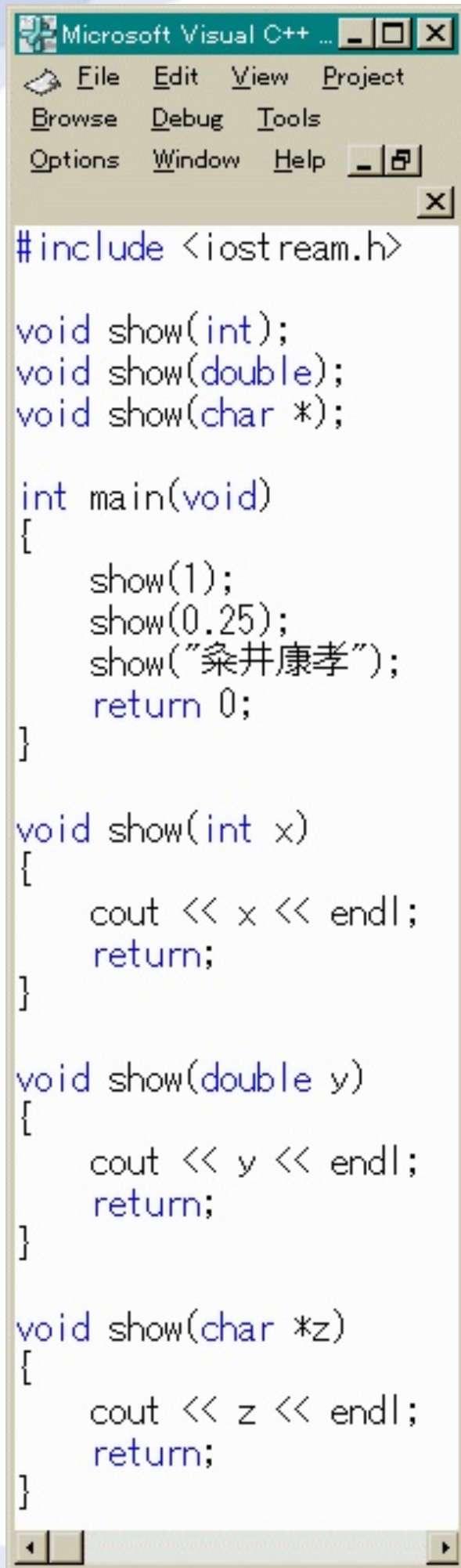
```

#### クラスNewCalcの仕様(メンバ関数)

関数名	戻り値の型	引数	概要
mul	double	なし	メンバ変数m_number1、m_number2の積
div	double	なし	メンバ変数m_number1、m_number2の商

## オーバーロード

(多重定義とも言うらしいです)



```
#include <iostream.h>

void show(int);
void show(double);
void show(char *);

int main(void)
{
    show(1);
    show(0.25);
    show("桑井康孝");
    return 0;
}

void show(int x)
{
    cout << x << endl;
    return;
}

void show(double y)
{
    cout << y << endl;
    return;
}

void show(char *z)
{
    cout << z << endl;
    return;
}
```

これは引数によって、呼び出す関数の内容を変えようという機能です。

左の例だと、関数showを3つの引数で場合分けしています。

これだけです。

コンストラクタもオーバーロードすることが出来ます。

```
class Point{
    private:
        int x;
        int y;
    public:
        Point(void){
            x=10;
            y=10;
        }
        Point(int num1,int num2){
            x=num1;
            y=num2;
        }
};
```

このコンストラクタに何も渡さないと(void)のコンストラクタが呼ばれデフォルトの値として10が入ります。

これだけです。

## 問題

x座標とy座標を受け取るクラスPointを作り

コンストラクタは何も受け取らない(0で初期化)、 x、 y どちらも受け取るの2つを定義してください。

## PROTECTED

継承って便利だ！！

```
class Animal{
    private:
        int age:
};
class Dog : public Animal{
    public:
        Dog(int num){
            age=num;    //ここがだめ。
        }
};
```

エラーが起きてしまいました。

Dogは継承によって変数ageを受け継いではいませんが、privateなのでアクセスできません。

これではまずい。

privateではなくprotectedにすると、privateだけど継承先でも呼び出せるってことになります。

```
class Animal{
    protected:
        int age:
};
class Dog : public Animal{
    public:
        Dog(int num){
            age=num;    //おっけー
        }
};
```

## 問題

x座標とy座標を持つクラスPointを作り

それを継承したクラスDistanceを作ってください。

また、Distanceの中には点(0,0)からの距離を求める関数を作ってください。

## オーバーライド

前章のオーバーロードと似た言葉で関数にも関係ありますが、オーバーロードとはちょっと違います。

オーバーライドとは継承先のクラス内で継承元と同じメソッド名、戻り地型、引数のメソッドを作成することです。

コード例

```
class Point{
    public:
        virtual int add(int a,int b){
            return a + b;
        }
};

class PointWithBonus : public Point{
    public:
        int add(int a,int b) override{
            return a + b + 5;
        }
};
```

オーバーライドされる継承元のクラスのメソッドには"virtual"というものを戻り値型の前につけ、オーバーライドする継承先のメソッドには引数の後ろに"override"とつけなければなりません。(正確にはつけなくても動きますがなにかミスをしていたときろくなことにならないのでつけておきましょう)virtualがついている関数のことを仮想関数と言います。

継承元のメソッドが仮想関数だったとしても、継承先でそれがオーバーライドされていなければ通常通り継承元のほうが呼び出されます。

あくまで同じメソッド名、同じ戻り値型、同じ引数でなければオーバーライドにはなりません。同じでなければそれはオーバーロードです。

オーバーライドすることによって継承先のクラスで機能を追加するだけでなく、既存の機能を書き換えることもできるようになります。



このように、継承元の関数(今回はoutput())が仮想関数(message())を呼び出していて、継承

```
#include <string>
#include <iostream>
using namespace std;
class MessageHoge{
public:
    void output(){
        cout << message() << endl;
    }
    virtual string message(){
        return "Hoge";
    }
};
class MessageFuga : public MessageHoge{
public:
    string message() override {
        return "Fuga";
    }
};
int main(){
    MessageHoge hoge;
    hoge.output();
    MessageFuga fuga;
    fuga.output();
}
```

Hoge

Fuga

先でオーバーライドしているときは、output()が継承元のメンバとして呼び出されているのか、それとも継承先のメンバとしてなのかで実際に呼び出される関数が変わります。

小クラスのオーバーライドしたメンバ関数から親クラスのオーバーライドされたメンバ関数を呼び出すには<親クラスの型名>::<オーバーライドした関数名>(引数);というふうに呼び出します。例えば上のコードでMessageFuga内のメンバ関数でMessageHogeのmessage()を呼び出す場合MessageHoge::message();とします。

## 純粋仮想関数

継承を使いながらクラスを作っていくと、「抽象クラスで処理を定めなくて、サブクラスで絶対に処理を決めるようにしたい」と思うことがあります。

そのようなとき下のようにメソッドを書くことで、継承先のクラスで処理を実装しない上で、継承先でオーバーライドして処理を実装することを強制させることができます。

```
virtual method = 0;
```

このようなメンバ関数のことを「純粋仮想関数」と言います。

純粋仮想関数が含まれるクラスはインスタンス化できません

### コード例

```
#include <iostream>
using namespace std;
class Base{
    public:
        void say(){
            message();
        }
    protected:
        virtual void message() = 0;
};
class Piyo : public Base{
    private:
        void message() override {
            cout << "Piyo" << endl;
        }
};
class Wan : public Base{
    private
        void message() override {
            cout << "Wan" << endl;
        }
};
```

```
int main(){
    // Base base; <- エラー
    Piyo piyo;
    Wan wan;
    piyo.say();
    wan.say();
}
```

ArduinoのSerial及びSoftwareSerialでは純粋仮想関数が使われています。

これら2つのクラスでは両方とも"readStringUntil"というメンバ関数が使えます。これは両方ともStreamというクラスを継承しており、その中にreadStringUntilが定義されているからです。

しかし、名前の通りreadStringUntilはread()を簡単にする関数ですからread()を使用するわけですが、そのread()の中でする処理はSerialとSoftwareSerialでは全く異なります。またStreamクラスはSerialとSoftwareSerial等の共通処理をまとめているものですから、read()の処理は実装したくありませんが、read()というメンバ関数があることは定義しなければなりません。

Streamではread()を純粋仮想関数にすることでSerial、SoftwareSerialでそれぞれ処理を実装することを強制した上で、Stream内のメンバ関数がread()を使うことを可能にしています。

## 問題

1. 以下のPointクラスを継承し、家庭科の得点を保存するメンバ変数homeを追加し、calc()メンバ関数を呼び出した際に家庭科の値も加算するように変更したクラスPointFor1を作り、またPointクラス内の一部を正しいプログラムになるように書き換えよ。

```
class Point {
    public:
        Point(int math,int japanese,int english,int chemistry)
            :math(math),japanese(japanese),english(english),chemistry(chemistry)
        {}
        int calc(){
            return math + japanese + english + chemistry;
        }
    private:
        int math,japanese,english,chemistry;
};
```

2. 以下の2クラスに共通する親クラスShapeを作成せよ

```
class Triangle: public Shape{
    public:
        Triangle(double bottom,double height):bottom(bottom),height(height){}
        double getArea() override{
            return bottom * height / 2;
        }
    private:
        double bottom,height;
};
```

```
#define _USE_MATH_DEFINES
#include <math.h>
class Triangle: public Shape{
public:
    Triangle(double radius):radius(radius){}
    double getArea() override{
        return radius * radius * M_PI;
    }
private:
    double radius;
};
```