

[戻る](#)

- [クラスとは](#)
- [どうやって使うの？](#)
- [実際にやってみる](#)
- [クラスとオブジェクト](#)
 - [オブジェクトの生成](#)
 - [メンバにアクセスする](#)
 - [動的なオブジェクトの生成](#)
- [メンバへのアクセスの制限](#)
 - [privateメンバ](#)
 - [publicメンバ](#)
 - [カプセル化](#)
- [コンストラクタ](#)
 - [コンストラクタの定義](#)
- [デストラクタ](#)
 - [デストラクタの必要性](#)
 - [デストラクタの定義](#)

クラスとは

クラスとはC++やjava, Rubyなどのオブジェクト指向言語にとって重要なものではどんなものなのでしょうか？

「関数の入る列挙体」「1つのメイン関数みたいなもの」など、各言語によって使い方に多少の違いがありますが、一番分かりやすいものとしては

```
モノの性質や、それに関わる機能をまとめたもの
```

と言えます

どうやって使うの？

とある参考書の受け売りですが、例えば車を想像して下さい

車には「ガソリンの残量」「ナンバー」などのデータ(=性質)を持っています

また「ガソリンを入れる」「ナンバーを決める」「それらを表示する」などの動作(=機能)も必要です
それらの性質や機能をクラスとしてまとめるには、次のようなコードを記述します

```
class 車{  
    ナンバー;  
    ガソリン量;  
    ナンバーを決める;  
    ガソリンを入れる;  
    ナンバーとガソリン量を表示する;  
}
```

広義的なクラスの宣言は、次のようにします

```
class クラス名{  
    アクセス指定子:  
    変数の宣言;  
    . . .  
    関数の宣言;  
    . . .  
};
```

クラスの場合、変数を「データメンバ」、関数を「メンバ関数」と呼びます

しかしこのままだと、メンバ関数の処理が定義されていないので実行できません
よって、メンバ関数の本体はクラスの外側で定義します

```
戻り値の型 クラス名::メンバ関数名(引数)  
{  
    . . .  
}
```

クラス名の後ろにある「: :」はスコープ解決演算子といいます
これを使って、メンバがどのクラスのメンバ関数であるのかを指定します

実際にやってみる

先ほどの車の例を使って、「Car (車)」クラスを宣言します

```
class Car{
public:
    int num;
    double gas;
    void show();
};

void Car::show()
{
    printf("車のナンバーは%d, ガソリン量は%fです \n", &num, &gas);
}
```

本来C++ではprintfは使わないのですが、分かりやすいよう敢て使っています
このCarクラスは、次のようなデータメンバとメンバ関数を持っています

データメンバ	説明
num	ナンバーを格納する変数
gas	ガソリン量を格納する変数
メンバ関数	説明
show()	ナンバーとガソリン量を出力する関数

クラスとオブジェクト

オブジェクトの生成

クラスを宣言しただけでは、車は生成されません
言ってしまうと、クラスは車の「設計図」みたいなものです
なので、クラスを使って「現物」の車を生成しなければなりません

```
Car car1;
```

「Carクラスの値」を記憶できる変数car1を宣言します
言い換えれば「Carクラスを元に、車car1を作った」こととなります
この出来上がった車(=変数)のことをオブジェクトと言います
一般にオブジェクト生成は次のように行います

```
クラス名 変数名;
```

メンバにアクセスする

車は生成されましたが、ナンバーもガソリンも無い状態では車は走れません
(豆知識: 私有地の道路(私道)ではナンバーが無くても法律上問題ないそうです)
「ナンバー」「ガソリン」を入力して、初めて車として機能します
それではまず、car1にナンバーを付けてガソリンを入れましょう

```
car1.num = 1234;
car1.gas = 25.5;
```

構造体と同じようにドット演算子(.)を使ってメンバにアクセスします
これと同じようにしてメンバ関数にもアクセス出来ます

```
car1.show();
```

動的なオブジェクトの生成

ポインタを使った動的なオブジェクトの生成は、new演算子を使います

```
Car* pCar;  
  
pCar = new Car;  
  
pCar->num = 1234;  
pCar->gas = 25.5;  
  
delete pCar;
```

動的に確保したメモリのアドレスを、ポインタpCarに格納します
そしてdelete演算子で動的メモリを開放します
なお、ポインタからメンバにアクセスする場合、構造体と同じようにアロー演算子(->)を使います

メンバへのアクセスの制限

privateメンバ

メンバへのアクセス方法を紹介しましたが、これでは問題が起こる場合があります
「ナンバーの偽装」や「ガソリンの盗難」など、世知辛い世の中です
なので、クラスの外から勝手にアクセスできないメンバにするわけです

```
class Car{  
    private:  
        int num;  
        double gas;  
        . . .  
};
```

アクセス指定子をprivateにすると、その下のメンバは全てクラスの外からアクセス出来なくなります
アクセス指定子は省略すると、全てprivateメンバになります

publicメンバ

ナンバーの偽装やガソリンの盗難は防げましたが、アクセスが出来なくなってしまったので
このままではナンバーの変更も給油も儘なりません
そこで使うのがpublicメンバです

```
class Car{  
    private:  
        int num;  
        double gas;  
    public:  
        void show();  
        void setNumGas();  
};  
  
void Car::show()  
{  
    printf("車のナンバーは%d、ガソリン量は%fです\n", &num, &gas);  
}
```

```

void Car::setNumGas()
{
    int n;
    double g;

    scanf("%d",&n);
    scanf("%f",&g);

    num = n;
    gas = g;
}

```

publicメンバにすることで、クラスの外からでもメンバにアクセス出来ます
publicなメンバ関数を通して、privateメンバを変更することも出来ます

カプセル化

上の2つのように、クラスの中にデータ（データメンバ）と機能（メンバ関数）をひとまとめにし、保護したいメンバにprivateをつけてアクセス制限する機能をカプセル化といいます
一般的には

データメンバ	privateメンバ
メンバ関数	publicメンバ

と指定します

コンストラクタ

コンストラクタの定義

出来ればすぐ乗れるように、ナンバーもガソリンも全部揃った状態で購入したいものですね
そこでコンストラクタを定義します

```

class Car{
    private num;
    double gas;
public:
    Car();
    void show();
};

void Car::show()
{
    printf("車のナンバーは%d、ガソリン量は%fです\n",&num,&gas);
}

Car::Car()
{
    num = 1234;
    gas = 20.0;
    printf("車を作成しました\n");
}

```

コンストラクタを定義すると、オブジェクトが生成された直後に定義されたコンストラクタが自動的に呼び出されます
これで車の完成と同時に、ナンバーとガソリンが与えられます
なお、一般的にコンストラクタの定義は次のように行います

クラス名::クラス名(引数)

(注) コンストラクタは必ず、クラス名を関数名にします
また、戻り値を持たず型の宣言も必要ありません

デストラクタ

デストラクタの必要性

デストラクタはコンストラクタとは逆で、オブジェクトを破棄する際に自動的に呼び出される関数です。では、何故呼び出す必要があるのでしょうか？

例えば、オブジェクトを動的に確保した場合、そのままオブジェクトを破棄するとメモリが開放されずに残ってしまいます

そこでdelete演算子をデストラクタに入れて定義すると、オブジェクトの破棄と同時にメモリを開放することが出来るわけです

デストラクタの定義

デストラクタの定義は、コンストラクタの定義によく似ています

```
クラス名::~クラス名()
```

コンストラクタと違うのは、スコープ解決演算子の後ろに~（チルダ）を付けます