オブジェクト指向と
ゲームプログラミング

○　C++編 － 第９回　演算子のオーバーロード，フレンド　○

演算子のオーバーロード

C++では、単項演算子と２項演算子のほとんどをオーバーロードできます。ただし、以下のような制約があります。

・基本型どうしの演算を行う演算子はオーバーロードできません。すなわち、int型どうしやdouble型とint型の演算を行う、といった演算子はオーバーロードできません。
・以下の演算子はオーバーロードできません。

演算子	定　義	演算子	定　義
.	直接メンバ演算子	?	条件演算子
.*	メンバへのポインタ	#	プリプロセス指令
::	スコープ解決演算子	##	プリプロセス指令

・演算子の優先順位と結合規則は変更できません。+演算子を*演算子よりも優先して演算するようにしたり、「左から右」に評価される式を「右から左」に評価させるようにする、といったことはできません。
・単項演算子を２項演算子(またはその逆)としてオーバーロードすることはできません。
・新しい演算子を創出して言語を拡張することはできません。

演算子のオーバーロードを行う目的は、クラスどうしまたはクラスと基本型や構造体の変換、操作を「自然にわかりやすく」記述するためです。注意しなければならないこととして、「演算子の本来持つ意味に反するようなオーバーロードを行っても何のエラーも起きない」ということが挙げられます。たとえば、+演算子をオーバーロードし、その内部で減算を行っても、文法さえ正しければエラーにはなりません。しかし、まったく脈絡のない処理や演算子から想像しづらい処理を割り当てても、後々のメンテナンスで苦労するだけです。

オーバーロード演算子の実装

オーバーロード演算子は、関数として記述されます。それらはグローバル関数としても、クラスのなかでメンバ関数としても定義できます。いずれの場合も、以下のような構文で関数を宣言します。

戻り値の型 operator演算子(引数)

キーワードoperatorに続く「演算子」の部分にオーバーロードしたい演算子の記号(=や+など)を記述します。以下のCharacterクラスでは、代入演算子=と、加算代入演算子+=をオーバーロードしています。

```
// 座標構造体
struct POINT {
    double   x;
    double   y;
};

// ベクトル構造体
struct VECTOR2 {
    double   x;
    double   y;
};

// キャラクタークラス
class Character {
public:
    Character& operator=(const POINT&);      // 代入演算子の定義
    Character& operator+=(const VECTOR2&);  // 加算代入演算子の定義

private:
    double   x_;     // x座標
    double   y_;     // y座標
};
```

```
// 代入演算子の実装
Character& Character::operator=(const POINT& pt)
{
    x_ = pt.x;        // x座標を代入
    y_ = pt.y;        // y座標を代入
    return *this;     // 結果を返す
}

// 加算代入演算子の実装
Character& Character::operator+=(const VECTOR2& vec)
{
    x_ += vec.x;      // x座標の移動
    y_ += vec.y;      // y座標の移動
    return *this;     // 結果を返す
}
```

２つの演算子のオーバーロードにより、以下のような演算が可能になります。

```
Character    chara;
POINT        pt  = {320.0, 240.0};    // 座標
VECTOR2      vec = {100.0, -50.0};    // 移動量

chara  = pt;     // 座標の設定
chara += vec;    // 移動
```

演算子の実装で気をつける点は、引数がクラスや構造体の場合、その参照を受け取るようにすることです。値渡しにすると、呼び出しごとに一時的なコピーを生成し、それを渡すコードが生成されてしまいます。これは効率が悪いので、引数の指定は"&"を付けて参照型にします。

また、オーバーロード関数の最後で処理結果を返すために「return *this;」と記述しています。これは、これ以外に自分自身の参照を返す方法がないためです。文法上は、どんな型でも返すようにできますが、実際にはその意味上ほとんどの場合、クラスへの参照を返すことになります。

フレンド

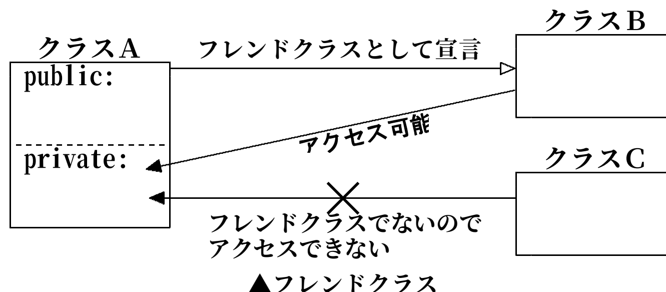
オブジェクト指向では、データのカプセル化を行ってクラスの独立性を高めることが重要です。クラスのメンバ変数をprivateで宣言すると、クラスの外部からはアクセスできなくなります。この機構を用いると、クラスを利用するほかのプログラムとの相互依存性を最小限に抑え、クラスの実装を変更した際に生ずる影響の度合いを少なくすることができます。

しかし、相互のクラスの依存を始めから許容したい場合もあります。通常は、お互いのクラスに備わったpublicなメンバ関数を介してアクセスするように設計しますが、こうすると、アクセスされたくない別のクラスからもアクセスできてしまいます。

C++では、この問題を解決するため、フレンド(friend)という概念が導入されています。フレンドを使うと、指定したクラスまたは関数にだけprivateメンバを含むすべてのメンバに直接アクセスを許可することができます。

フレンドクラス

クラス宣言のなかで、キーワードfriendを伴って宣言されたクラスは、そのクラスのすべてのメンバにアクセスすることが可能になります。このようなクラスをフレンドクラスと呼びます。



▲フレンドクラス

次のようなクラスAとクラスBがあったとします。

```
// クラスの前方参照
class B;

// クラスA
class A {
    // フレンドクラスの宣言
    friend class B;      // クラスBにすべてのメンバへのアクセスを許可する

protected:
    double   d_;

private:
    int      i_;
};

// クラスB
class B {
public:
    void InitA(A& a)
    {
        a.d_ = 0.0;      // privateやprotectedメンバに直接アクセスしても、
        a.i_ = 0;        // フレンド宣言されているのでエラーにならない
    }
};
```

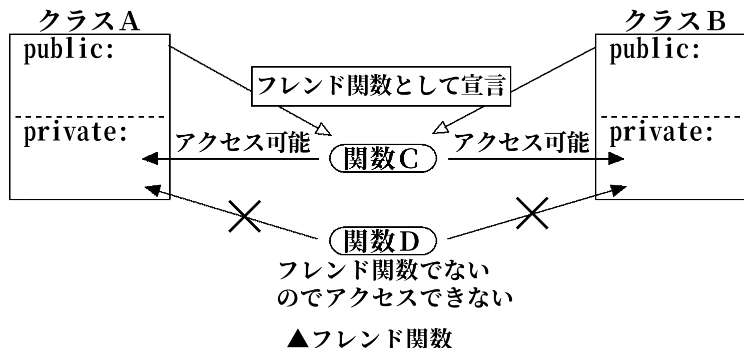
クラスAは、privateメンバとprotectedメンバがあり、クラスBをフレンドクラスとして宣言しています。クラスBのメンバ関数InitAでは、クラスAのprivateメンバとprotectedメンバに直接アクセスし、それぞれを初期化しています。クラスBがクラスAのフレンドクラスでなければ、このような直接アクセスはできないのです。

このように、別のクラスのprivateメンバに直接アクセスする必要が生ずる典型的なものが、単項演算子をオーバーロードする場合です。フレンドクラスは、別のクラスが自クラスに対する処理を行う場合にはとても有効ですが、まったく必然性のないクラスをフレンドクラスにしてしまうと、結果的にすべてのメンバをpublicにしたのと同じになり、カプセル化のメリットが無くなってしまうので注意しましょう。

フレンド関数

クラス宣言のときに、friendを伴って宣言された関数は、そのクラスのprivateメンバにアクセスすることが可能になります。このような関数をフレンド関数と呼びます。

クラスAとクラスB、そしてクラスAとクラスBのメンバ変数にアクセスする必要のあるグローバル関数Cがあったとします。通常、クラスのメンバ変数はprivateなので関数Cがそれらにアクセスするには、メンバ関数をとおして間接的に行うしかありません。カプセル化の概念から、メンバ変数をpublicに変更することは当然できません。しかし、ここでクラスAとクラスBにおいて関数Cをフレンド関数として宣言しておけば、関数Cのみ、それらのクラスのすべてのメンバにアクセスすることができるようになります。



▲フレンド関数

クラスのprivateメンバにアクセスする必要のある関数とは、当然そのクラスに関係のある処理をしていると考えられます。そのような関数の典型的なものが演算子のオーバーロード関数です。フレンド関数を用いてオーバーロードするのは、ほとんどの場合２項演算子をオーバーロードするときです。

あるクラスＡと別のクラスＢを加算するといった、クラスどうしの処理を行う演算子を定義する場合には、フレンド関数以外にアクセス制限を超えて記述できる方法はありません。なお、単項演算子のオーバーロード関数は、メンバ関数およびフレンド関数のどちらを使っても記述できます。

```
// 前方参照
class A;

// 関数プロトタイプ
A& operator+(const A&, const A&);

// クラスA
class A {
    // フレンド関数の宣言
    friend A& operator+(const A&, const A&);     // 関数operator+に対してすべてのメンバに
                                                 // アクセスすることを許可する
public:
    A(const double d, const int i) : d_(d), i_(i) {}     // コンストラクタ
    A& operator=(const A&);                              // 代入演算子の定義

protected:
    double   d_;

private:
    int      i_;
};

// 代入演算子の実装
A& A::operator=(const A& op)
{
    if(&op != this) {            // 自分自身を代入する(a = a)を回避
        d_ = op.d_;              // メンバ変数をコピー
        i_ = op.i_;
    }
    return *this;                // 自分自身を返す
}

// 加算演算子の定義
A& operator+(const A& op1, const A& op2)
{
    static A   op3(0.0, 0);      // 作業用オブジェクト
    op3.d_ = op1.d_ + op2.d_;    // それぞれのメンバを加算
    op3.i_ = op1.i_ + op2.i_;
    return op3;                  // 加算した結果を返す
}
```

代入演算子と加算演算子のオーバーロード関数により、以下のように記述することができるようになります。

```
A   a(100.0, 50), b(-50.0, -50), c(89.3, 3);     // オブジェクトの生成
a = a + b + c;                                   // 加算して結果を代入
```