演算子のオーバーロード

演算子を新たな意味で使う

- C＋＋の機能として、演算子（+, -, *, /, % 等）に対し新たな機能を割り当てることができる
  - ( ), [ ], ->, =, ==, &&, ||, ^, >>, != 等にも適用できる
  - ただし、全ての演算子が使えるわけではない
    - .(ドット), :: (スコープ), ?:　(3項演算子), .*　(メンバポインタを参照するドット演算子) の4つ、および、プリプロセッサ演算子は使用不可
  - デフォルト引数は使用できない

- 演算的な処理を置き換える
  - 記述が簡素になる
  - ()括弧と組み合わせて，複合的な処理が1行で書ける
  - ベクトル，行列演算等，演算子の扱えるデータのクラスを拡張できる
    - 演算子を汎用的に使用できる

演算子の宣言例

```
class Point {
private:
    int x, y;
public:
    Point (int _x = 0, int _y = 0) { x = _x; y = _y; }
    Point operator + (const Point& p) { // ベクトルの足し算
        return Point (x + p.x, y + p.y);
    int operator * (const Point& p) { // ベクトルの内積
        return x * p.x + y * p.y;
    }
};
```

演算子とメンバ関数の記法の違い

```
Point plus (Point& p) {
    return Point (x + p.x, y + p.y);
}

Point p2 = p0.plus(p1);

Point operator + (Point& p) {
    return Point (x + p.x, y + p.y);
}

Point p2 = p0 + p1; // p0.(operator +)(p1)
```

演算子の実行例

```
int main () {
    Point p0 (1,2);
    Point p1 (3,4);
    Point p2 = p0 + p1; // p2 ← (4, 6)
    int d = p2 * p1; // d ← 36
    Point p3 = p2 + 1; // p3 ← (5, 6)
                // operator + (Point& p) なので、
                // 1 → Point(1) が暗黙裡に呼ばれる
}
```

スカラー積の実装

Point.h

```
class Point {
    ...
    Point operator * (int k); // ベクトル * 倍数
    friend Point operator * (int k, Point pnt); // 倍数 * ベクトル
};
```

Point.cpp

```
Point Point::operator * (int k) { // ベクトル * 倍数
    return Point (x * k, y * k);
}
Point operator * (int k, Point pnt); //「Point::」は不要
    return Point (k * pnt.x, k * pnt.y);
}
```

メンバ関数でない演算子

- 左オペランドがクラスオブジェクトでない場合、二つの引数を取る
  - これにより、クラスのメンバ関数としては扱われなくなる（「クラス名::」のプレフィックスが不要）
- 隠蔽されたクラス変数にアクセスするために、friend 関数とする
  - 関数が宣言されるクラスの、全メンバにアクセスできる

スカラー積の実行例

```
int main () {
    Point p0(1,2);
    Point p1 = p0 * 3; // メンバ関数として呼ばれる
    Point p2 = 2 * p1; // 単なる関数として呼ばれる
    // p1 == (2, 6) , p2 == (4, 12) になる
}
```

operatorでの単項演算子作成法

- 単項演算子 +, -, !, ~ の作成には、引数は指定しない。
- 単項演算子以外のオーバーライドは不可

```
class Point {
    …
   Point operator – () {
        return Point (–this–>x, –this–>y);} // 正常に動作
   int operator ^ () {// ^は2項演算子なのでコンパイラエラー
        return this–>x * this–>y; }
```

const 修飾子の活用

```
class Point {
private:
    int x, y;
public:
    Point (int _x, int _y) { x = _x; y = _y; }
    Point operator + (const Point& p) { // p は更新不可
        return Point2D (x + p.x, y + p.y);
    }
    int operator * (Point& p) const { // x, y, p は更新不可
        return x * p.x + y * p.y;
    }
};
```

代入演算子のオーバーロード

- 代入演算子「=」を、オーバーロードできる

```
Point& operator = (const Point& p) {
    x = p.x; // x値のコピー
    y = p.y; // y値のコピー
    return *this; // 自分自身を返す
}
```

コピーコンストラクタ

- 自身のクラスを引数とする構築子は、コピーコンストラクタと呼ばれる。
- 代入と、コピーコンスタラクタは、似ているが異なるメソッドが呼び出されている。

```
Point(const Point& p) : x(p.x), y(p.y){}
```

代入演算子とコピーコンストラクタの違い

Point p0(1,2), p1(3,4); // 通常の構築子が呼ばれる

Point p2 = p1; // p2 の初期化と解釈され、コピーコンストラクタが呼ばれる（p2 の (x, y) は (3, 4) となる）

p2 = p1 = p0; // 代入演算子が呼ばれる。右から式が適用されるので、p2, p1, p0 の全座標が (1,2) となる

Point3D p3d(1,2,3); // Point から派生したクラス

p0 = p3d; // p0 は p3d から自身が含む変数のみをコピーするので、Point3D のインスタンスには変換されない！

Point p2d = p3d; // コピーコンストラクタにおいても、上記と同様に p2d は、Point3D のインスタンスには変換されない！

派生クラスに変換される場合

Point p;

Point3D p3d(1,2,3); // Point から派生したクラス

p = p3d; // Point3D には変換されない！

Point pc = p3d; // Point3D には変換されない！

Point &pr = p3d; // 参照なので、p3d の実(Point3D)が保存される

Point *pp = new Point3D(4,5,6); // アドレス値の代入なので、Point3Dの実体となる

pp = &p3d; // 上記と同様