クラス　Class
メンバ関数と演算子のオーバーロード(1)

榎本　理恵

講義内容

回	講義・実験内容
1	クラスの復習1
2	クラスの復習2(オペレータのオーバーロード)
3	ポインタの復習とメモリの動的確保
4	スタック
5	キュー
6	連結リスト①
7	連結リスト②(再帰)
8	二分木
9	中間テスト
10	ハッシュ
11	クイックソート
12	マージソート
13	ヒープソート
14	文字列探索
	学期末テスト

問題

次のような実行結果となるプログラムを作成したい.

今日: 2022/9/29.
Taroさんの誕生日 : 2002/9/29.
Peach-kunさんの誕生日 : 1923/3/3.
Jakobさんの誕生日 : 1654/12/27.

Taroさんは今 20 歳です
今日が誕生日です。お誕生日おめでとう！
Peach-kunさんは今 99 歳です
今年の誕生日は過ぎてしまいました
Jakobさんは今 367 歳です
誕生日まであと89日です
Jakob is released.
Peach-kun is released.
Taro is released.

問題(続き)

これは、名前と生年月日を与えておいて、今日の日付が誕生日を超えていない場合は、年齢と今年の誕生日までの日数を出力してくれるプログラムである。ここでは、うるう年かどうかは気にせず、すべてうるう年でないとしてよい。さらに、

- 今日の日付が誕生日を超えていたら、

  今年の誕生日は過ぎてしまいました

- 今日の日付が誕生日であったら、

  今日が誕生日です。お誕生日おめでとう！

と出力するようにしたい。

途中まで記述してあるファイルがCoursePowerにprac01_skel.cppという名前で置かれているので、利用してプログラムを完成させなさい。

作成したプログラムprac01.cppと実行結果を、CoursePowerで割り当てられたクラスに水曜24時までに提出しなさい。

解答

```cpp
#include <iostream>
using std::cout,std::cin,std::string;

// クラス Date 定義部分
// Dateクラスに不備がある場合は訂正すること
class Date
{
private:
    int year;  // 年
    int month; // 月
    int day;   // 日
public:
    Date(){}                                    //Personクラスのデータメンバになっているため、Personクラスの
                                                //オブジェクトが生成される際Dateのコンストラクタが呼び出される
                                                //(const Date d;)
                                                //このためデフォルト引数の指定か引数なしのコンストラクタが必要
    //コンストラクタ:引数の値をデータメンバに格納
    Date(const int y, const int m, const int d){
        year = y;
        month = m;
        day = d;
    }
    // データメンバyear の値を返す
    int get_year() const{ return year; }
    // データメンバmonth の値を返す
    int get_month() const{ return month; }
    // データメンバday の値を返す
    int get_day() const{ return day; }
    // プロトタイプ宣言
    int days_from_newyearsday();
};
```

```
// データメンバに格納されている日にちについて元日からの日数を計算する
int Date::days_from_newyearsday(){
    int days_in_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int days = day;

    // 終了条件が (month-1) であることに注意.（例）2月5日の場合には 31+5 になる
    for (int i = 0; i < (month - 1); i++)
    {
        days += days_in_month[i];
    }
    return days - 1; // 元旦からの日数なので 1 を引く.（例）1月1日の場合は 0
}
//クラス Person 定義部分
class Person
{
private:
    string name;    // 名前
    Date birth_day; // 生年月日
public:
    // コンストラクタ:引数の値をデータメンバに格納
    Person(const string fn, const Date d){
        name = fn;
        birth_day = d;
    }
    // データメンバname の値を返す
    string get_name(){return name;}
    // 名前と生年月日を出力する.（例）「Taroさんの誕生日:2002/9/29.」を出力
    void print_birth() const{cout << name << "さんの誕生日:" << birth_day.get_year() << "/" << birth_day.get_month() << "/" <<
birth_day.get_day() << ". ¥n";}
    int tobirthday(Date today); // 関数のプロトタイプ宣言
    int ages(Date today);       // 関数のプロトタイプ宣言
    // Person クラスのデストラクタの設定.「名前＋is released.」を表示する
    ~Person(){cout << name << " is released.¥n";}
};
```

```
// 引数から誕生日までの日数を計算し返す(「今日の日付」と「誕生日」との差を比較するため負の値になる可能性があることに注意)
int Person::tobirthday(Date today){ return birth_day.days_from_newyearsday() - today.days_from_newyearsday();}
// 引数から誕生日までの年数を計算し返す
int Person::ages(Date today){
    // 今日と誕生日の日付を比較して，今年誕生日がきているかどうかで場合分けする.
    if (tobirthday(today) > 0){ return today.get_year() - birth_day.get_year() - 1;
    }else{return today.get_year() - birth_day.get_year();}
}
void print_day(int date){
    if (date > 0){ cout << "誕生日まであと" << date << "日です¥n";
    }else if (date < 0){cout << "今年の誕生日は過ぎてしまいました¥n";
    }else{cout << "今日が誕生日です。お誕生日おめでとう！¥n";}
}
int main()
{
    Date today(2022, 9, 29); // 今日の日付
    Person taro("Taro", Date(2002, 9, 29));
    Person peach("Peach-kun", Date(1923, 3, 3));
    Person jakob("Jakob", Date(1654, 12, 27));
    cout << "今日: " << today.get_year() << "/" << today.get_month() << "/" << today.get_day() << ". ¥n";
    taro.print_birth();
    peach.print_birth();
    jakob.print_birth();

    cout << "¥n";
    cout << taro.get_name() << "さんは今 " << taro.ages(today) << " 歳です¥n";
    print_day(taro.tobirthday(today));
    cout << peach.get_name() << "さんは今 " << peach.ages(today) << " 歳です¥n";
    print_day(peach.tobirthday(today));
    cout << jakob.get_name() << "さんは今 " << jakob.ages(today) << " 歳です¥n";
    print_day(jakob.tobirthday(today));

    return 0;
}
```

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 出力演算子のオーバーロード
5. 入力演算子のオーバーロード

演算子の多重定義(オーバロード)

- 関数のオーバロード
  C++の関数では、シグニチャ(関数名，引数の型および個数のこと)が異なることによって、
  同名の関数が複数存在することが出来る。
- 演算子のオーバロード
  四則演算子をはじめとする演算子を、
  新たに定義したクラスを取り扱うように
  多重定義することが出来る。

複素数の加減乗除の関数

Complexクラスの加減乗除をメンバ関数で表現

```
class Complex {
    double  real, imaginary; // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    Complex add(Complex);          //自分と引数cを加算する
    Complex sub(Complex);          //自分から引数cを減算する
    Complex mul(Complex);          //自分と引数cを乗算する
    Complex div(Complex);          //自分から引数cを除算する
    void print();
};
Complex Complex::add(Complex c){       //自分と引数cを加算する
    Complex sum;
    sum.real=real+c.real;    sum.imaginary=imaginary+c.imaginary;
    return sum;
}
Complex Complex::sub(Complex c){       //自分から引数cを減算する
    Complex sum;
    sum.real=real-c.real;    sum.imaginary=imaginary-c.imaginary;
    return sum;
}
```

複素数の加減乗除の関数(続き)

```
Complex Complex::mul(Complex c){      //自分と引数cを乗算する
    Complex sum;
    sum.real=real*c.real-imaginary*c.imaginary;
    sum.imaginary=real*c.imaginary+imaginary*c.real;
    return sum;
}
Complex Complex::div(Complex c){      //自分から引数cを除算する
    Complex sum;                      //除算をする際には、0でないことを確認して！
    double tmp=c.real*c.real+c.imaginary*c.imaginary;
    sum.real=(real*c.real+imaginary*c.imaginary)/tmp;
    sum.imaginary=(-real*c.imaginary+imaginary*c.real)/tmp;
    return sum;
}
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";    // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(5,0), e(2,1), f;        //Complex型の変数dとeとfの宣言および
    f=d.add(e);       f.print();                    //コンストラクタを使った初期化
    f=d.sub(e);       f.print();
    f=d.mul(e);       f.print();
    f=d.div(e);       f.print();
    return 0;
}
```

加算のわりには不自然。
f=d+e;と書けないか？

実行例

```
7+1i
3-1i
10+5i
2-1i
```

演算子のオーバロード

- 演算子は関数の特殊ケース

  例:　＋演算子　→　operator+という名の関数

- 二項演算子における2つの解釈

  obj1 + obj2

  obj1:第1オペランド
  obj2:第2オペランド

【方法1】operator+をobj1のクラスのメンバ関数と解釈

obj1.operator+(obj2) と等価

【方法2】両オペランドともoperator+の引数と解釈

operator+(obj1, obj2) と等価

この場合、operator+はメンバ関数とはならない。

演算子のオーバロードの方法1

先ほどの複素数の演算を演算子で計算できるようにする。

```
class Complex {
    double  real, imaginary; // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    Complex operator+(Complex);          //自分と引数cを加算する
    Complex operator-(Complex);          //自分から引数cを減算する
    Complex operator*(Complex);          //自分と引数cを乗算する
    Complex operator/(Complex);          //自分から引数cを除算する
    void print();
};
Complex Complex::operator+(Complex c){       //自分と引数cを加算する
    Complex sum;
    sum.real=real+c.real;    sum.imaginary=imaginary+c.imaginary;
    return sum;
}
Complex Complex::operator-(Complex c){       //自分から引数cを減算する
    Complex sum;
    sum.real=real-c.real;    sum.imaginary=imaginary-c.imaginary;
    return sum;
}
```

演算子のオーバロードの方法1(続き)

```
Complex Complex::operator*(Complex c){      //自分と引数cを乗算する
    Complex sum;
    sum.real=real*c.real-imaginary*c.imaginary;
    sum.imaginary=real*c.imaginary+imaginary*c.real;
    return sum;
}
Complex Complex::operator/(Complex c){      //自分から引数cを除算する
    Complex sum;//除算する際には、0でないことを確認したほうが本当は望ましい
    double tmp=c.real*c.real+c.imaginary*c.imaginary;
    sum.real=(real*c.real+imaginary*c.imaginary)/tmp;
    sum.imaginary=(-real*c.imaginary+imaginary*c.real)/tmp;
    return sum;
}
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";   // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(5,0), e(2,1), f;                  //Complex型の変数dとeとfの宣言および
                                                //コンストラクタを使った初期化
    f=d+e;      f.print();
    f=d-e;      f.print();
    f=d*e;      f.print();
    f=d/e;      f.print();
    return 0;
}
```

実行例

```
7+1i
3-1i
10+5i
2-1i
```

演算子のオーバロードの注意点1

```
int main(){
    Complex d(5,0), e(2,1), f; //Complex型の変数dとeとfの宣言
    f=d+e;
    f.print();
}
```

f = d.operator+(e);
と書いても良いが、そんな書き方をする人はいない・・・

演算子のオーバロードの注意点2

■オーバロード可能な演算子

+, −, ＊, /, %, ++, −−, <<, >>, ( ), [ ], <, >, <=, >=, ==, !=,
&&, ||, +=, −=, ＊=, /=, %=, &, ~, ! , ^, |, ->など

■オーバロード不可能な演算子

. :: .＊ ?:

■代入演算子=について

=は、同じクラスのオブジェクト同士であれば代入できるように
オーバロードされている演算子なので、
「代入」目的以外で使わない限り、
ユーザがオーバロードする必要はない。

■優先順位

オーバロードされた演算子でも、その演算子の元の優先順位や
結合規則は変えることができない。

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 出力演算子のオーバーロード
5. 入力演算子のオーバーロード

複素数と実数の和①

複素数の加算は解決できたが、複素数と実数の和はどうなるのか？

```
class Complex {
    double  real, imaginary;                    // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    Complex operator+(Complex);                 //自分と引数cを加算する
    void print();
};
Complex Complex::operator+(Complex c){      //自分と引数cを加算する
    Complex sum;
    sum.real=real+c.real;    sum.imaginary=imaginary+c.imaginary;
    return sum;
}
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";          // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                  //Complex型の変数dとeとfの宣言および
    f=d+5.0;            f.print();                  //コンストラクタを使った初期化
    return 0;
}
```

複素数と実数の和②

f=d+5.0;

Complex operator+(double)が無ければ、
d+5.0は、d.operator+(Complex)と解釈せざるを得ず、
5.0はComplex型に自動キャストすることになる。

d.operator+(Complex(5.0))

つまり5.0はコンストラクタにより 5.0+0.0i となって、
Complex operator+(Complex)のメンバ関数が使われる。
その結果、2.0+1.0iと5.0+0.0iの和になって、7.0+1.0iが得られる。

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 出力演算子のオーバーロード
5. 入力演算子のオーバーロード

実数と複素数の和①

複素数と実数の加算はできたが、実数と複素数の和はどうなるのか？

```
class Complex {
    double  real, imaginary;                          // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    Complex operator+(Complex);                       //自分と引数cを加算する
    void print();
};
Complex Complex::operator+(Complex c){      //自分と引数cを加算する
    Complex sum;
    sum.real=real+c.real;    sum.imaginary=imaginary+c.imaginary;
    return sum;
}
void Complex::print(){
    cout << real;        if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";              // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                        //Complex型の変数dとeとfの宣言および
    f=5.0+d;              f.print();          //コンストラクタを使った初期化
    return 0;
}
```

実数と複素数の和②

z=5. 0+d;

5.0は実数で、メンバ関数は持たないので、
5.0.operator+(d)とは解釈できないため、コンパイルエラーとなる。
このため演算子のオーバーロードのもう一つの方法を利用する必要がある。

演算子のオーバロードの方法2

```
class Complex {
    double  real, imaginary; //real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    friend Complex operator+(Complex, Complex);          //引数を加算する
    void print();
};
Complex operator+(Complex a, Complex b){
    Complex sum;
    sum.real=a.real+b.real;  sum.imaginary=a.imaginary+b.imaginary;
    return sum;
}
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";          // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                   //Complex型の変数dとeとfの宣言および
    f=5.0+d;            f.print();              //コンストラクタを使った初期化
    return 0;
}
```

メンバ関数ではなく、通常の関数として定義する。但し、そのままではデータメンバにアクセスできない。

friend宣言をすることにより、この関数からはデータメンバにアクセス可能になる

プログラムの実行結果　7+1i

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 出力演算子のオーバーロード
5. 入力演算子のオーバーロード

他の演算子のオーバーロードはどう行う?

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

class Complex {
    double  real, imaginary;                          //real:実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;}//コンストラクタ
    friend Complex operator+(Complex, Complex);              //引数を加算する
    void print();
};
Complex operator+(Complex a, Complex b){
    return Complex( a.real+b.real, a.imaginary+b.imaginary );
}
void Complex::print(){
    cout << real;        if(imaginary>=0) { cout << '+'; }
    cout << imaginary << 'i';                         // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                                //Complex型の変数dとfの宣言および
    f=5.0+d;             f.print(); cout << "¥n";     //コンストラクタを使った初期化
    return 0;
}
```

cout << f << "¥n";のように出力できるようにしたい。

プログラムの実行結果

```
7+1i
```

演算子のオーバーロードによる複素数の出力①

```
cout << f;
```

上記は、operator<<(cout, f)の形式でかかれている。
「<<」の左のcoutはostreamクラスで、fは複素数クラスである。
このため、方法1のメンバ関数による演算子のオーバーロードを行うためには、
ostreamクラスのメンバ関数として記述しなければならず難しい。
したがって、方法2の通常の関数として記述することになるが、
fのデータメンバであるrealやimaginaryはprivateであるから、
そのままではアクセスできない。
アクセスする手法は、get_real()やget_imaginary()などのgetterを作成する手法でもよいが、複素数クラスにoperator<<をフレンド関数として登録する手法でも実現できる。

演算子のオーバーロードによる複素数の出力②

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

class Complex {
    double  real, imaginary;                          //real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    friend Complex operator+(Complex, Complex);               //引数を加算する
    friend ostream& operator<<(ostream &, Complex);           //複素数を出力する
};
Complex operator+(Complex a, Complex b){
    return Complex( a.real+b.real, a.imaginary+b.imaginary );
}
ostream& operator<<(ostream &stout, Complex f){
    stout << f.real;       if(f.imaginary>=0) { stout << '+'; }
    stout << f.imaginary << 'i';                      // a+biまたはa-biの形式で表示
    return stout;
}
int main(){
    Complex d(2,1), f;                                //Complex型の変数dとfの宣言および
    f=5.0+d;             cout << f << "¥n";             //コンストラクタを使った初期化
    return 0;
}
```

プログラムの実行結果	7+1i

演算子のオーバーロードによる複素数の出力③

getterでも実装できるという参考例

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

class Complex {
    double  real, imaginary;                          //real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    double get_real(){ return real; }
    double get_imaginary(){ return imaginary; }
};
Complex operator+(Complex a, Complex b){
    return Complex( a.get_real()+b.get_real(), a.get_imaginary()+b.get_imaginary() );
}
ostream& operator<<(ostream &stout, Complex f){
    stout << f.get_real();        if(f.get_imaginary()>=0) { stout << '+'; }
    stout << f.get_imaginary() << 'i';                // a+biまたはa-biの形式で表示
    return stout;
}
int main(){
    Complex d(2,1), f;                                //Complex型の変数dとfの宣言および
    f=5.0+d;              cout << f << "¥n";            //コンストラクタを使った初期化
    return 0;
}
```

プログラムの実行結果	7+1i

出力演算子の関数表現

- 出力演算子 <<

```
cout << f << "¥n";
```

operator<<(cout, f)

```
ostream& operator<<( ostream& , Complex );
```

coutを受け取り、

coutを返す。

☞ cout(標準出力)は実行環境に1つなので、参照変数で受け渡す。
☞ 複素数クラスComplexのメンバ関数ではない！

入出力演算子の戻り値が参照である理由

ostream& operator<<(ostream &stout, Complex f){
istream& operator>>(istream &stin, Complex& f){

入出力演算子の戻り値は参照である。
例えば、複素数fを出力演算子を用いて出力することを考えると、
cout << f << "¥n";
となるが、この処理は、
operator<<(cout,f) << "¥n";
と解釈され、次に2個目の「<<」が解釈され、
operator<<(operator<<(cout,f), "¥n")
を意味する。
したがって、「operator<<(cout,f)」の戻り値がcoutでないと、
「¥n」を標準出力に出力することができない。
つまり、連結して「<<」を使用できるようにするためには、coutを戻す必要があり、
さらにcoutは唯一であるため、コピーではなくそのものを戻す必要がある。
(戻り値をvoidにしても「cout << f」は実行できるが、
「cout << f << "¥n"」のように連結することはできなくなる)

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 出力演算子のオーバーロード
5. 入力演算子のオーバーロード

複素数の入力プログラム

```
class Complex {
    double  real, imaginary;                          //real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;}//コンストラクタ
    friend ostream& operator<<(ostream &, Complex);       //複素数を出力する
    void input();//複素数を入力する
};
ostream& operator<<(ostream &stout, Complex f){
    stout << f.real;       if(f.imaginary>=0) { stout << '+'; }
    stout << f.imaginary << 'i';                      // a+biまたはa-biの形式で表示
    return stout;
}
void Complex::input(){
    cin >> real >> imaginary;
    return;
}
int main(){
    Complex f;                                        //Complex型の変数fの宣言および
    cout << "数値を2つ入力してください¥n";  //コンストラクタを使った初期化
    f.input();             cout << f << "¥n";
    return 0;
}
```

cin >> f;　で入力できるようにしたい

プログラムの実行結果

```
数値を2つ入力してください
7 1
7+1i
```

演算子のオーバーロードによる複素数の入力①

```
cin >> f;
```

上記は、operator>>(cin, f)の形式で記述されている。
「>>」の左のcinはistreamクラスで、fは複素数クラスである。
このため、方法1のメンバ関数による演算子のオーバーロードを行うためには、
istreamクラスのメンバ関数として記述しなければならず難しい。
したがって、方法2の通常の関数として記述することになるが、
fのデータメンバであるrealやimaginaryはprivateであるから、
そのままではアクセスできない。
アクセスする手法は、set_realやset_imaginaryなどのsetterを作成する手法でもよいが、
複素数クラスにoperator>>をフレンド関数として登録する手法でも実現できる。

演算子のオーバーロードによる複素数の入力②

```
class Complex {
    double  real, imaginary;                    //real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;}//コンストラクタ
    friend ostream& operator<<(ostream &, Complex);      //複素数を出力する
    friend istream& operator>>(istream&, Complex&);      //複素数を入力する
};
ostream& operator<<(ostream &stout, Complex f){
    stout << f.real;       if(f.imaginary>=0) { stout << '+'; }
    stout << f.imaginary << 'i';                 // a+biまたはa-biの形式で表示
    return stout;
}
istream& operator>>(istream &stin, Complex& f){
    stin >> f.real >> f.imaginary;
    return stin;
}
int main(){
    Complex f;                                   //Complex型の変数fの宣言および
    cout << "数値を2つ入力してください¥n";  //コンストラクタを使った初期化
    cin >> f;              cout << f << "¥n";
    return 0;
}
```

プログラムの実行結果

```
数値を2つ入力してください
7 1
7+1i
```

入力演算子の関数表現

- 入力演算子 >>

```
cin >> f;
```

```
operator>>(cin, f)
```

```
istream& operator>>( istream&, Complex& );
```

cinを受け取り、

cinを返す。

参照変数であることに注意！

☞ cin(標準入力)は実行環境に1つなので、参照変数で受け渡す。
☞ 複素数クラスComplexのメンバ関数ではない！
☞ 入力された値を代入するので、第2引数も参照変数。

本日はここまでです
お疲れさまでした

質問があれば、下記までお願いします
11号館4階1405号室

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

練習問題①

行列の足し算・引き算・掛け算を
オーバーロードする
プログラムを作りなさい。

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    Gyouretsu operator+(Gyouretsu);
    Gyouretsu operator-(Gyouretsu);
    Gyouretsu operator*(Gyouretsu);
    void print();
};
Gyouretsu Gyouretsu::operator+(Gyouretsu g){        //自分と引数gを加算する




}
Gyouretsu Gyouretsu::operator-(Gyouretsu g){        //自分と引数gを減算する




}
```

練習問題①続き

```
Gyouretsu Gyouretsu::operator*(Gyouretsu g){        //自分と引数gを乗算する



}
void Gyouretsu::print(){
    cout << a << "¥t" << b << "¥n";
    cout << c << "¥t" << d << "¥n";
    cout << "¥n";
}
int main()  {
    Gyouretsu g(1,2,3,4),f(5,6,7,8),z;
    z=g+f;          // 行列の掛け算
    z.print();      // zを表示
    z=g-f;          // 行列の掛け算
    z.print();      // zを表示
    z=g*f;          // 行列の掛け算
    z.print();      // zを表示
}
```

プログラムの実行結果

```
6       8
10      12

-4      -4
-4      -4

19      22
43      50
```

練習問題①解答

行列の足し算・引き算・掛け算を
オーバーロードする
プログラムを作りなさい。

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    Gyouretsu operator+(Gyouretsu);
    Gyouretsu operator-(Gyouretsu);
    Gyouretsu operator*(Gyouretsu);
    void print();
};
Gyouretsu Gyouretsu::operator+(Gyouretsu g){        //自分と引数gを加算する
    Gyouretsu sum;
    sum.a=a+g.a;      sum.b=b+g.b;          sum.c=c+g.c;          sum.d=d+g.d;
    return sum;
}
Gyouretsu Gyouretsu::operator-(Gyouretsu g){        //自分と引数gを減算する
    Gyouretsu sum;
    sum.a=a-g.a;      sum.b=b-g.b;          sum.c=c-g.c;          sum.d=d-g.d;
    return sum;
}
```

練習問題①解答続き

```
Gyouretsu Gyouretsu::operator*(Gyouretsu g){        //自分と引数gを乗算する
    Gyouretsu sum;
    sum.a=a*g.a+b*g.c;   sum.b=a*g.b+b*g.d;   sum.c=c*g.a+d*g.c;   sum.d=c*g.b+d*g.d;
    return sum;
}
void Gyouretsu::print(){
    cout << a << "¥t" << b << "¥n";
    cout << c << "¥t" << d << "¥n";
    cout << "¥n";
}
int main()  {
    Gyouretsu g(1,2,3,4), f(5,6,7,8), z;
    z=g+f;          // 行列の掛け算
    z.print();      // zを表示
    z=g-f;          // 行列の掛け算
    z.print();      // zを表示
    z=g*f;          // 行列の掛け算
    z.print();      // zを表示
}
```

プログラムの実行結果

```
6       8
10      12

-4      -4
-4      -4

19      22
43      50
```

本日お話する内容

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 複素数と実数の和
3. 実数と複素数の和(外部関数による演算子の記述)
4. 演習問題のための補足

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

練習問題②

先ほどの複素数の演算をメンバ関数ではなく、通常の関数で記述しなさい。

```
class Complex {
    double  real, imaginary; // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
                                                              //自分と引数cを加算する
                                                              //自分から引数cを減算する
                                                              //自分と引数cを乗算する
                                                              //自分から引数cを除算する
    void print();
};
```

練習問題②(続き)

```
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";    // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                  //Complex型の変数dとeとfの宣言および
                                                コンストラクタを使った初期化
    f=5.0+d;            f.print();
    f=5.0-d;            f.print();
    f=5.0*d;            f.print();
    f=5.0/d;            f.print();
    return 0;
}
```

実行例

```
7+1i
3-1i
10+5i
2-1i
```

練習問題②解答

先ほどの複素数の演算をメンバ関数ではなく、通常の関数で記述しなさい。

```
class Complex {
    double  real, imaginary; // real：実数部、imaginary:虚数部
public:
    Complex (double a=0, double b=0) { real=a; imaginary=b;} //コンストラクタ
    friend Complex operator+(Complex, Complex);//自分と引数cを加算する
    friend Complex operator-(Complex, Complex);//自分から引数cを減算する
    friend Complex operator*(Complex, Complex);//自分と引数cを乗算する
    friend Complex operator/(Complex, Complex);//自分から引数cを除算する
    void print();
};
Complex operator+(Complex a, Complex b){    //自分と引数cを加算する
    Complex sum;
    sum.real=a.real+b.real;          sum.imaginary=a.imaginary+b.imaginary;
    return sum;
}
Complex operator-(Complex a, Complex b){   //自分から引数cを減算する
    Complex sum;
    sum.real=a.real-b.real;          sum.imaginary=a.imaginary-b.imaginary;
    return sum;
}
```

練習問題②解答(続き)

```
Complex operator*(Complex a, Complex b){   //自分と引数cを乗算する
    Complex sum;
    sum.real=a.real*b.real-a.imaginary*b.imaginary;
    sum.imaginary=a.real*b.imaginary+a.imaginary*b.real;
    return sum;
}
Complex operator/(Complex a, Complex b){   //自分から引数cを除算する
    Complex sum;//除算する際には、0でないことを確認したほうが本当は望ましい
    double tmp=b.real*b.real+b.imaginary*b.imaginary;
    sum.real=(a.real*b.real+a.imaginary*b.imaginary)/tmp;
    sum.imaginary=(-a.real*b.imaginary+a.imaginary*b.real)/tmp;
    return sum;
}
void Complex::print(){
    cout << real;       if(imaginary>=0) { cout << '+'; }
    cout << imaginary << "i¥n";          // a+biまたはa-biの形式で表示
}
int main(){
    Complex d(2,1), f;                   //Complex型の変数dとeとfの宣言および
    f=5.0+d;            f.print();       //コンストラクタを使った初期化
    f=5.0-d;            f.print();
    f=5.0*d;            f.print();
    f=5.0/d;            f.print();
    return 0;
}
```

実行例

```
7+1i
3-1i
10+5i
2-1i
```

実数と行列の掛け算

行列f*2.0は、メンバ関数Gyouretsu operator*(double)が無ければ、
f.operator*(Gyouretsu)と解釈せざるを得ず、
2.0はコンストラクタにより、Gyouretsu型である$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$と
自動キャストされる。
このため、$f * \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$となり、正しい値が得られる。

一方、行列2.0*fの場合、2.0は実数でメンバ関数は持たないので、
2.0.operator*(f)とは解釈できないため、コンパイルエラーになる。
このため複素数の場合と同様に、外部関数でoperator*を定義し、
friend宣言することで、
Gyouretsuクラスのデータメンバにアクセスできるようにする。

練習問題①の解答修正版(乗算部分のみ)

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    friend Gyouretsu operator*(Gyouretsu, Gyouretsu);
    void print();
};
void Gyouretsu::print(){
    cout << a << "¥t" << b << "¥n";
    cout << c << "¥t" << d << "¥n";
    cout << "¥n";
}
Gyouretsu operator*(Gyouretsu h, Gyouretsu g){    //引数hと引数gを乗算する
    Gyouretsu sum;
    sum.a=h.a*g.a+h.b*g.c;   sum.b=h.a*g.b+h.b*g.d;
    sum.c=h.c*g.a+h.d*g.c;   sum.d=h.c*g.b+h.d*g.d;
    return sum;
}
```

練習問題①の解答修正版(続き)

```
int main()  {
    Gyouretsu g(1,2,3,4),f(5,6,7,8),z;
    z=g*f;                                  // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    z=f*2.0;                                // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    z=2.0*f;                                // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    return 0;
}
```

プログラムの実行結果

```
19      22
43      50

10      12
14      16

10      12
14      16
```

実数と行列の掛け算(続き)

なお、getterを設定することでも、
外部関数でoperator*を定義し、Gyouretsuクラスのデータメンバにアクセスできるようにすることはできるが、
getterはどの関数からでも利用することができるため、
friend宣言する方法とは異なる。

練習問題①の解答再修正版

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    double get_a(){ return a; }
    double get_b(){ return b; }
    double get_c(){ return c; }
    double get_d(){ return d; }
    void print();
};
void Gyouretsu::print(){
    cout << a << "¥t" << b << "¥n";
    cout << c << "¥t" << d << "¥n";
    cout << "¥n";
}
Gyouretsu operator*(Gyouretsu h, Gyouretsu g){   //引数hと引数gを乗算する
   return Gyouretsu( h.get_a()*g.get_a()+h.get_b()*g.get_c(), h.get_a()*g.get_b()+h.get_b()*g.get_d(),
            h.get_c()*g.get_a()+h.get_d()*g.get_c(), h.get_c()*g.get_b()+h.get_d()*g.get_d());
}       //この書式だとset関数が不要！
        //(もちろんset関数でsumに値を設定し、できたsumをリターンしてもよい)
```

練習問題①の解答再修正版(続き)

```
int main() {
    Gyouretsu g(1,2,3,4), f(5,6,7,8), z;
    z=g*f;                                  // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    z=f*2.0;                                // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    z=2.0*f;                                // 行列の掛け算
    z.print();          cout << "¥n";       // zを表示
    return 0;
}
```

プログラムの実行結果

```
19      22
43      50

10      12
14      16

10      12
14      16
```

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

練習問題

main関数が動作するように、
入出力演算子をオーバーロードしなさい。

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    friend Gyouretsu operator*(Gyouretsu, Gyouretsu);          //乗算



};
Gyouretsu operator*(Gyouretsu h, Gyouretsu g){     //引数hと引数gを乗算する
    Gyouretsu sum;
    sum.a=h.a*g.a+h.b*g.c;    sum.b=h.a*g.b+h.b*g.d;
    sum.c=h.c*g.a+h.d*g.c;    sum.d=h.c*g.b+h.d*g.d;
    return sum;
}
```

練習問題(続き)

main関数が動作するように、

入出力演算子をオーバーロードしなさい。

```
int main()  {
    Gyouretsu g, f, z;
    cin >> g;
    cin >> f;
    cout << "¥n";
    z=g*f;                    // 行列の掛け算
    cout << z << "¥n";        // zを表示
    z=f*2.0;                  // 行列の掛け算
    cout << z << "¥n";        // zを表示
    z=2.0*f;                  // 行列の掛け算
    cout << z << "¥n";        // zを表示
    return 0;
}
```

プログラムの実行結果

```
1 2
3 4

5 6
7 8

19      22
43      50

10      12
14      16

10      12
14      16
```

練習問題解答

main関数が動作するように、
入出力演算子をオーバーロードしなさい。

```
class Gyouretsu {
    double  a,b;// 2行2列の行列
    double  c,d;// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){ a=x;  b=y; c=z; d=w; }
    Gyouretsu(double x=1){ a=x;  b=0; c=0; d=x; } //コンストラクタ(多重定義)
    friend Gyouretsu operator*(Gyouretsu, Gyouretsu);          //乗算
    friend istream& operator>>(istream&, Gyouretsu&);          //行列を入力する
    friend ostream& operator<<(ostream&, Gyouretsu);           //行列を出力する
};
Gyouretsu operator*(Gyouretsu h, Gyouretsu g){     //引数hと引数gを乗算する
    Gyouretsu sum;
    sum.a=h.a*g.a+h.b*g.c;   sum.b=h.a*g.b+h.b*g.d;
    sum.c=h.c*g.a+h.d*g.c;   sum.d=h.c*g.b+h.d*g.d;
    return sum;
}
istream& operator>>(istream &stin, Gyouretsu& g){
    stin >> g.a >> g.b;
    stin >> g.c >> g.d;
    return stin;
}
```

練習問題解答(続き)

```
ostream& operator<<(ostream &stout, Gyouretsu g){
    stout << g.a << "¥t" << g.b << "¥n";
    stout << g.c << "¥t" << g.d << "¥n";
    stout << "¥n";
    return stout;
}
int main()  {
    Gyouretsu g, f, z;
    cin >> g;
    cin >> f;
    cout << "¥n";
    z=g*f;                          // 行列の掛け算
    cout << z << "¥n";              // zを表示
    z=f*2.0;                        // 行列の掛け算
    cout << z << "¥n";              // zを表示
    z=2.0*f;                        // 行列の掛け算
    cout << z << "¥n";              // zを表示
    return 0;
}
```

プログラムの実行結果

```
1 2
3 4

5 6
7 8

19      22
43      50

10      12
14      16

10      12
14      16
```

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

添字演算子のオーバーロードの例①

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

class Complex {
    double  real, imaginary;                    // 実数部realと虚数部imaginary
public:
    Complex(double a=0, double b=0){ real=a; imaginary=b; }
    double operator[](int num){                 //添字演算子はメンバ関数で実装
        // real,imaginaryを外部からf[0]とf[1]のように配列で呼び出せる
        if(num==0){return real;}
        else if(num==1){return imaginary;}
    }
};
ostream& operator<<(ostream &stout, Complex f){
    stout << f[0];                              // f[0]はrealのこと
    if(f[1]>=0) {stout << '+';}                 // f[1]はimaginaryのこと
    stout << f[1] << 'i';
    return stout;
}
int main()  {
    Complex f(7,1);                             //Complex型の変数dの宣言および
    cout << f << "¥n";                          //コンストラクタを使った初期化
    return 0;
}
```

プログラムの実行結果

```
7+1i
```

添字演算子のオーバーロードの例②

```
class Complex {
    double  real, imaginary;                    // 実数部realと虚数部imaginary
public:
    Complex(double a=0, double b=0){ real=a; imaginary=b; }
    double& operator[](int num){                //添字演算子の戻り値を参照に変更
        if(num==0){return real;}
        else if(num==1){return imaginary;}
    }
};
ostream& operator<<(ostream &stout, Complex f){
    stout << f[0];              if(f[1]>=0) {stout << '+';}
    stout << f[1] << 'i';
    return stout;
}
istream& operator>>(istream &stin, Complex& f){
    stin >> f[0] >> f[1];     //添字演算子が参照を返却するので、値を代入できる
    return stin;
}
int main()  {
    Complex f;                                  //Complex型の変数fの宣言および
    cout << "数値を2つ入力してください¥n";//コンストラクタを使った初期化
    cin >> f;          cout << f << "¥n";
    return 0;
}
```

プログラムの実行結果

```
数値を2つ入力してください
7 1
7+1i
```

添字演算子のオーバーロードの練習問題

空欄に何が入るか考えてみてください

```
class Double2 {
    double  hairetsu[100];          // 0から99までの100個の浮動小数点の配列
public:
    Double2( ){ for(int i=0;i<100;i++){ hairetsu[i]=i; } }  //初期化(値の理由は特にない)
    double& operator[](int n){                              //メンバ関数で実装
        // 負の値や、100以上の配列を参照するとエラーと表示する
        double gomi=-1000.0;
        if(n<0){cerr << "エラー:" << n << endl; return gomi;}
        else if(n>=100){cerr << "エラー:" << n << endl; return gomi;}
        else{return  ？？？ ;}
    }
};
int main()  {
    Double2 z;
    cout << z[99] << "¥n";
    z[99]=1000.0; //添字演算子の戻り値が参照なので代入が可能
    cout << z[99] << "¥n";
    cout << z[-1] << "¥n";
    return 0;
}
```

プログラムの実行結果

```
99
1000
エラー:-1
-1000
```

添字演算子のオーバーロードの練習問題解答

空欄に何が入るか考えてみてください

```
class Double2 {
    double  hairetsu[100];              // 0から99までの100個の浮動小数点の配列
public:
    Double2( ){ for(int i=0;i<100;i++){ hairetsu[i]=i; } }  //初期化(値の理由は特にない)
    double& operator[](int n){                              //メンバ関数で実装
        // 負の値や、100以上の配列を参照するとエラーと表示する
        double gomi=-1000.0;
        if(n<0){cerr << "エラー:" << n << endl; return gomi;}
        else if(n>=100){cerr << "エラー:" << n << endl; return gomi;}
        else{return hairetsu[n];}
    }
};
int main()  {
    Double2 z;
    cout << z[99] << "¥n";
    z[99]=1000.0; //添字演算子の戻り値が参照なので代入が可能
    cout << z[99] << "¥n";
    cout << z[-1] << "¥n";
    return 0;
}
```

プログラムの実行結果

```
99
1000
エラー:-1
-1000
```

行列の添字演算子のオーバーロード

```
class Gyouretsu {
    double  a[2][2];// 2行2列の行列
public:
    Gyouretsu(double x, double y, double z, double w){                    //コンストラクタ
        a[0][0]=x;  a[0][1]=y; a[1][0]=z; a[1][1]=w; }
    Gyouretsu(double x=1){a[0][0]=x;  a[0][1]=0; a[1][0]=0; a[1][1]=x;} //多重定義
    double* operator[](int num){                  //添字演算子の戻り値をポインタにする
        if(num==0){return a[0];}
        else if(num==1){return a[1];}
    }
    double& operator()(int i, int j){             //()演算子の場合は参照を戻せばよい
        return a[i][j];
    }
    friend istream& operator>>(istream&, Gyouretsu&);          //行列を入力する
    friend ostream& operator<<(ostream&, Gyouretsu);           //行列を出力する
};
istream& operator>>(istream &stin, Gyouretsu& g){
    stin >> g.a[0][0] >> g.a[0][1];
    stin >> g.a[1][0] >> g.a[1][1];
    return stin;
}
```

行列の添字演算子のオーバーロード(続き)

```
ostream& operator<<(ostream &stout, Gyouretsu g){
    stout << g.a[0][0] << "¥t" << g.a[0][1] << "¥n";
    stout << g.a[1][0] << "¥t" << g.a[1][1] << "¥n";
    return stout;
}
int main()  {
    Gyouretsu z;
    cin >> z;
    cout << "¥n";
    cout << z << "¥n";                    // zを表示
    z[1][0]=100;
    cout << z << "¥n";                    // zを表示
    z(1,0)=2000;
    cout << z << "¥n";                    // zを表示
    return 0;
}
```

プログラムの実行結果

```
1 2
3 4

1       2
3       4


1       2
100     4


1       2
2000    4
```

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

フレンド（friend）関数

自分のクラス以外の特定の関数に対して、
自分のクラスのprivateデータへのアクセスを許可したい場合、
その関数をfriend関数と宣言することによって、
それが可能となる。

フレンド関数を使用したプログラムの例

```
#include <iostream>
#include <string>
using namespace std;
class Girl;          // Boyのメンバ関数print()の引数にGirlを指定しているため
                     // その前にGirlはクラスであることを明示しなければならない。
class Boy{
    string name;   //名前
    int age;       //年齢
public:
    Boy(string x, int y){ name=x;  age=y; }              //コンストラクタ
    void print(Girl);
};
class Girl{
    string name;              //名前
    string phone_num;         //電話番号
public:
    Girl(string p, string q){ name=p;  phone_num=q; }  //コンストラクタ
    friend void Boy::print(Girl);
            // Boyクラスのprint関数をフレンド関数であると宣言
};          // Boyクラスなので、Boy::がつく。外部関数の場合にはつかない
```

フレンド関数を使用したプログラムの例(続き)

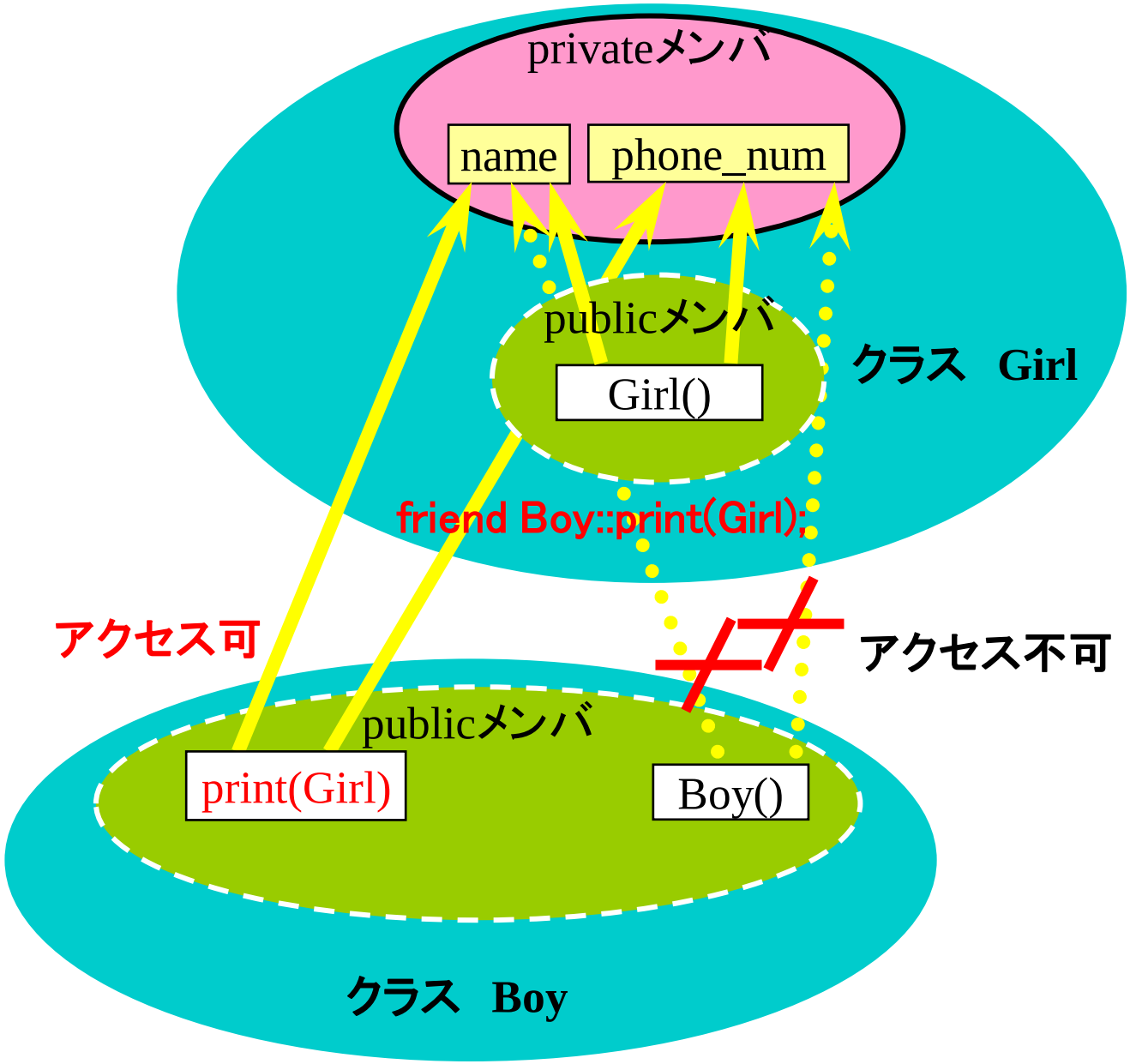
```
void Boy::print(Girl josei){
    cout <<"Name : "<< name <<",  Age : "<< age << "¥n";
    cout <<"Her name: "<< josei.name <<",  TEL : "<< josei.phone_num << "¥n";
  // Boy::print()はクラスGirlでfriend指定されているため、
  //クラスGirlのデータメンバnameおよびphone_numにアクセスすることができる。
}
int main(void){
    Boy dansei("Bill",21);
    Girl josei("Eluza","3210-0123");

    dansei.print(josei);
    return 0;
}
```

実行例

```
% example5
Name : Bill,  Age : 21
Her name: Eluza,  TEL : 3210-0123
%
```

フレンド関数


privateメンバ
name
phone_num
publicメンバ
Girl()
クラス　Girl
friend Boy::print(Girl);
アクセス可
アクセス不可
publicメンバ
print(Girl)
Boy()
クラス　Boy

付録

1. 複素数同士の和(メンバ関数による演算子の記述)
2. 実数と複素数の和(外部関数による演算子の記述)
3. 入出力演算子のオーバーロード
4. 添字演算子のオーバーロード
5. フレンド関数
6. フレンドクラス

フレンドクラス①

あるクラスのデータメンバに対して、
他のクラスの全メンバ関数からアクセスすることを許可したい場合には、
そのクラス自体を以下のようにフレンド指定することで可能となる。

```
class Girl;          // Boyのメンバ関数print()の引数にGirlを指定しているため
                     // その前にGirlはクラスであることを明示しなければならない。
class Boy{
    string name;   //名前
    int age;       //年齢
public:
    Boy(string x, int y){ name=x;  age=y; }          //コンストラクタ
    void print(Girl);
};
class Girl{
    string name;           //名前
    string phone_num;      //電話番号
public:
    Girl(string p, string q){ name=p;  phone_num=q; }  //コンストラクタ
    friend class Boy; // Boyクラスがフレンドクラスであると宣言
};          // Boyのメンバ関数からGirlのデータメンバにアクセス可能
```

フレンドクラス②

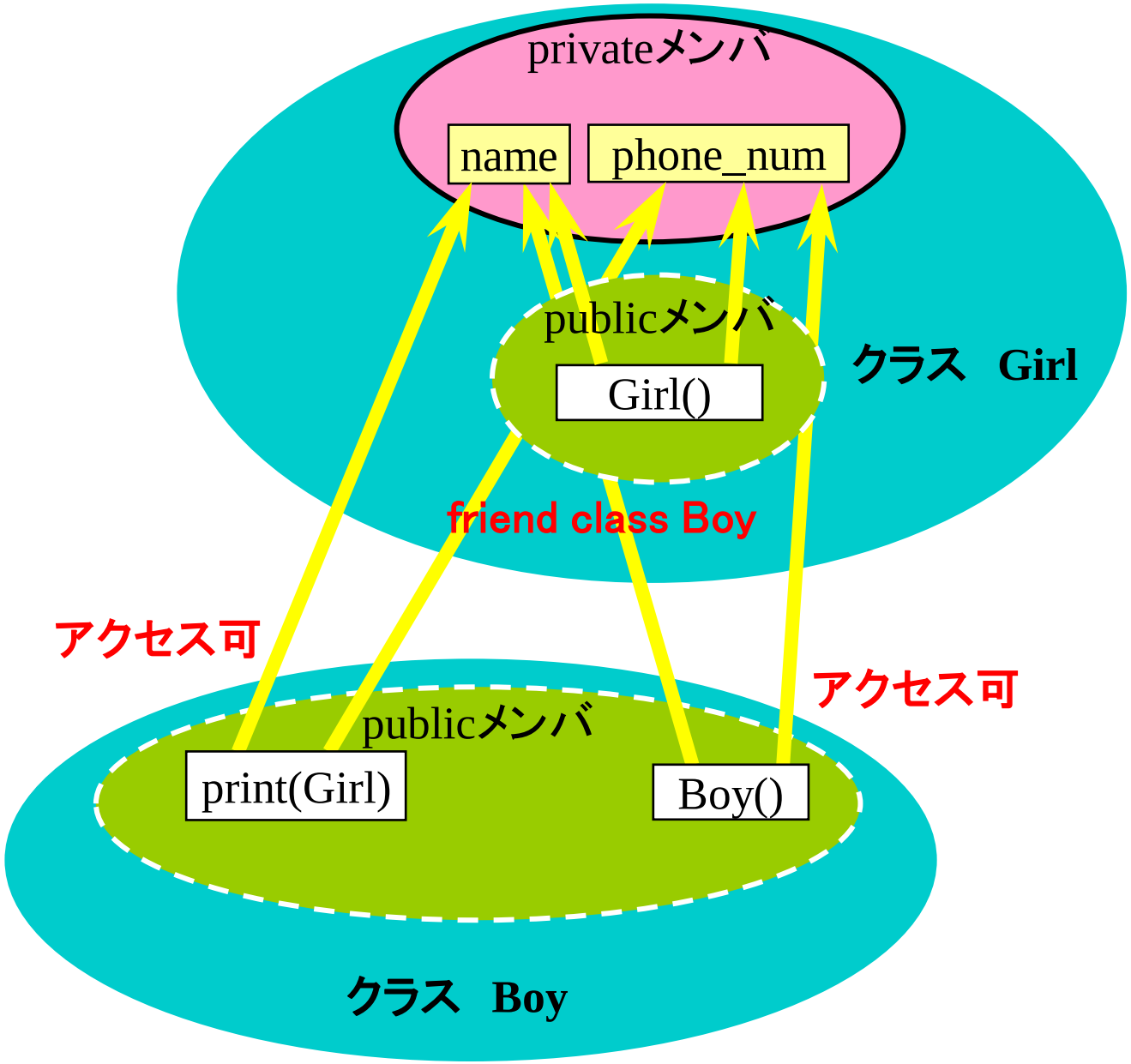
```
void Boy::print(Girl josei){
    cout <<"Name : "<< name <<",  Age : "<< age << "¥n";
    cout <<"Her name: "<< josei.name <<",  TEL : "<< josei.phone_num << "¥n";
    // Boy::print()はクラスGirlでfriend指定されているため、
    //クラスGirlのデータメンバnameおよびphone_numにアクセスすることができる。
}
int main(void){
    Boy dansei("Bill",21);
    Girl josei("Eluza","3210-0123");

    dansei.print(josei);
    return 0;
}
```

実行例

```
% example5
Name : Bill,  Age : 21
Her name: Eluza,  TEL : 3210-0123
%
```

フレンドクラス


privateメンバ
name
phone_num
publicメンバ
Girl()
クラス　Girl
friend class Boy
アクセス可
アクセス可
publicメンバ
print(Girl)
Boy()
クラス　Boy

フレンド関数による演算子のオーバロードの例

C++の数学ライブラリにある三角関数の引数は
単位がradianである。これをdegree単位で扱うことができる
新しいsinを作成する（関数のオーバロード）。
加えてdegree単位の角度の加算や比較をする必要が
あるため、演算子のオーバロードも行う。

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
class Degree{
    double deg;
public:
    Degree(double x=0){ deg=x; }     // コンストラクタ以外はすべてフレンド関数
    friend double sin(Degree);                              //度を引数とするsin関数
    friend ostream& operator<<(ostream& , Degree); //度の出力
    friend Degree operator+(Degree, Degree);          //度の加算
    friend bool operator<=(Degree, Degree);           //度の比較
};
double sin(Degree x){//度をラジアンに変換しオリジナルのsin関数の値を求めて戻す
    return( sin(x.deg*M_PI/180) );
}

ostream& operator<<(ostream& outst, Degree a){    //度の出力
    outst << a.deg;
    return outst;
}
```

フレンド関数による演算子のオーバロードの例

```
Degree operator+(Degree a, Degree b){      //度の加算
    Degree tmp;
    tmp.deg=a.deg+b.deg;
    return tmp;
}

bool operator<=(Degree a, Degree b){       //度の比較
    if(a.deg<=b.deg){
        return true;
    } else{
        return false;
    }
}                  //「n=0」は、「n=Degree(0)」と解釈される
                   //「n<=180」は、「operator<=(n,Degree(180))」と解釈される
int main(void){    //「n+10」は、「operator+(n,Degree(10))」と解釈される
  Degree n;
  for(n=0; n<=180; n=n+10){
     cout << "sin("
          << resetiosflags(ios::showpoint)                  //リセット
          << setiosflags(ios::fixed)                        //固定小数点方式
          << setprecision(0) << setw(3) << n << ")=";       //小数点は0桁、幅3文字で出力
     cout << setiosflags(ios::showpoint | ios::fixed)//小数点強制出力・固定小数点方式
          << setprecision(10) << sin(n) << "¥n";            //小数点以下10桁を出力
  }
  return 0;
}
```

プログラムの実行結果

```
sin(  0)=0.0000000000
sin( 10)=0.1736481777
sin( 20)=0.3420201433
sin( 30)=0.5000000000
sin( 40)=0.6427876097
sin( 50)=0.7660444431
sin( 60)=0.8660254038
sin( 70)=0.9396926208
sin( 80)=0.9848077530
sin( 90)=1.0000000000
sin(100)=0.9848077530
sin(110)=0.9396926208
sin(120)=0.8660254038
sin(130)=0.7660444431
sin(140)=0.6427876097
sin(150)=0.5000000000
sin(160)=0.3420201433
sin(170)=0.1736481777
sin(180)=0.0000000000
```

メンバ関数か、フレンド関数か？

1. 作成するクラスのオブジェクトを扱う関数は、
   まずメンバ関数で考える。
2. 関数（または演算子）のオペランドの順序により
   やむを得ない場合（入出力の取り扱いなど）や、
   同じ関数（または演算子）のオーバロードが
   うまく節約できるときには、フレンド関数を考える。
3. 無闇にフレンド関数を多用しないこと。
   フレンド関数は、所詮外部関数に過ぎない。
   本来のクラスの意味を忘れないこと。