クラスの復習

C++プログラミングIII

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

授業の方法

このままの状況が続けば14回対面で行う予定です。

前半部分で講義を行い、

後半部分で演習(プログラムの作成)を行います。

講義はまとめて1人の先生が行い、演習は各クラスに分かれます。

各自のクラスはCoursePowerで確認してください。

ZOOMでのオンライン授業になった場合、講義と演習でZOOMの情報が異なるので注意してください。

成績評価について

中間と期末試験を実施できる場合には、

中間試験48点＋期末試験48点＝96点満点を予定。

（ただし、中間試験は再試験を行わない）

中間と期末試験を実施できない場合には、

各回の演習時の回答を採点して成績に反映する予定。

演習問題については、翌週水曜日24時まで提出（CoursePower）。

注意点

これまでの経験上、様々な人が様々な目的でプログラムを書きます。
したがって、本来、プログラムの使われる場所や目的が明確に
ならないと、最適なプログラムは決まらないということになります。

私が作成する練習問題やプログラム例は、
わかりやすさとバグが生じにくいことを重視して
作成しているつもりです。
このため、メモリ効率や処理速度はあまり考えていないものが
多いので、その点はご承知おきください。
(これは私がメモリ効率や処理速度などを考える必要がなく、
第三者がソースを見たときの分かりやすさと、
バグが絶対に許されない環境で仕事をしていたからです)

余計なアドバイス

1. 既に上手く動いているプログラムをベースとして、
   それを改変するのがおススメ。
   正しく動くものに新たに書き足していって動かなくなったら、
   新しく足した部分に問題がある可能性が高い。
   (弊害として変数名を書きかえるのを忘れ、
   実行結果がおかしくなることがあるのでこちらも注意！)

2. 値を随時細かくチェックすること。特に配列などに値を代入したら、
   値を出力してチェック。(適宜コメントアウトしながら！！)

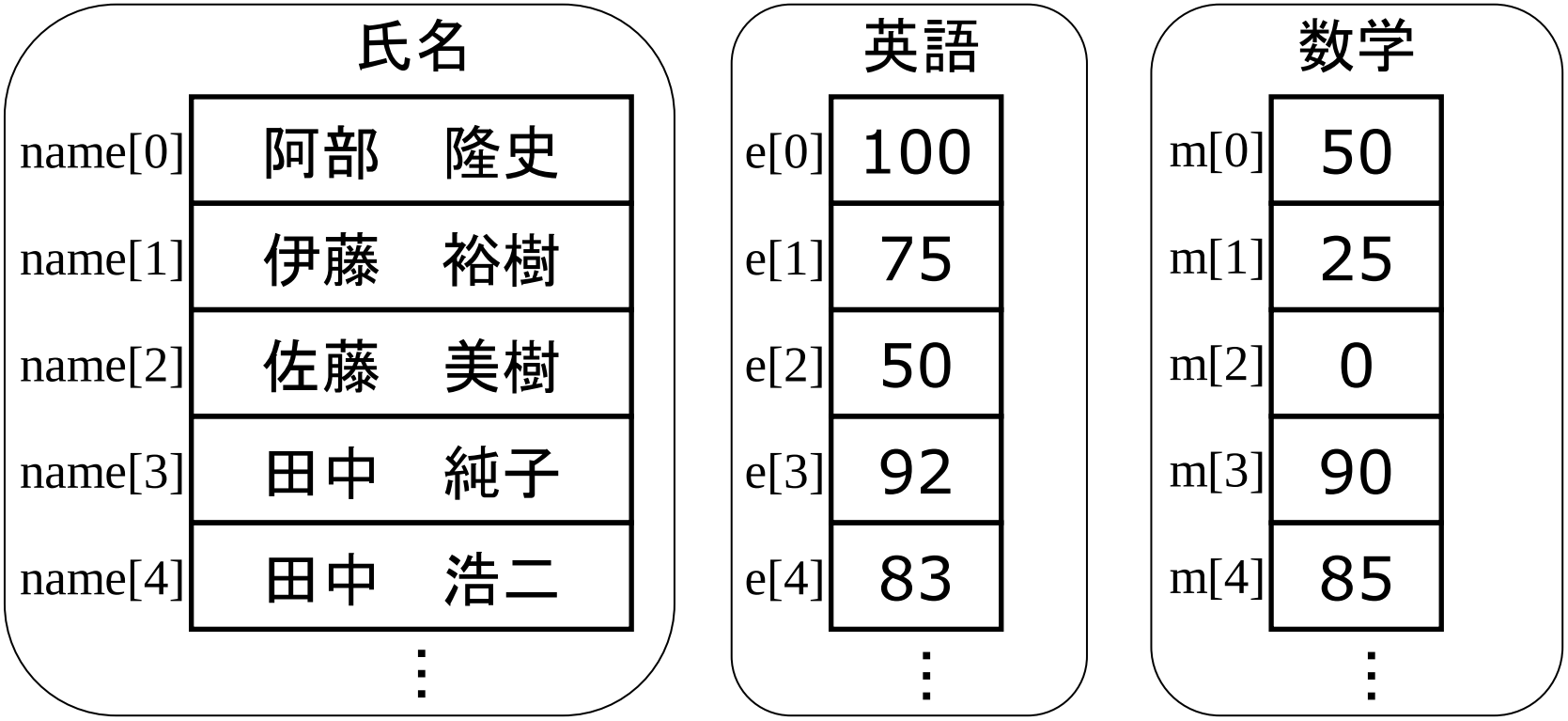
3. プログラムを勉強する目的は大きく2つある。
   1つは、自分でプログラムが書けるようになること。
   もう1つは、他人の書いたプログラムが読めること。

本日お話する内容

1. クラスの復習

クラスの必要性①

学生10人の氏名と英語・数学の試験の得点を整理したい。
配列を利用すると・・・



一人の学生に着目するために共通のインデックスを使っているだけで、一人分としてまとまりがない。
自分だけでプログラムを使う分には良いが、他人と分担してプログラムを作成する場合には、説明が必要！

英語と数学の平均点を計算するプログラム(配列)

```
int main(){

    string name[100];
    int e[100];
    int m[100];

    name[0]="阿部";     e[0]=100;     m[0]=50;
    name[1]="伊藤";     e[1]=75;      m[1]=25;
    name[2]="佐藤";     e[2]=50;      m[2]=0;
    int averageEnglish=(e[0]+e[1]+e[2])/3;
    int averagemath=(m[0]+m[1]+m[2])/3;
    cout << "英語の平均点=" << averageEnglish << "¥n";
    cout << "数学の平均点=" << averagemath << "¥n";

    return 0;
}
```

実行例

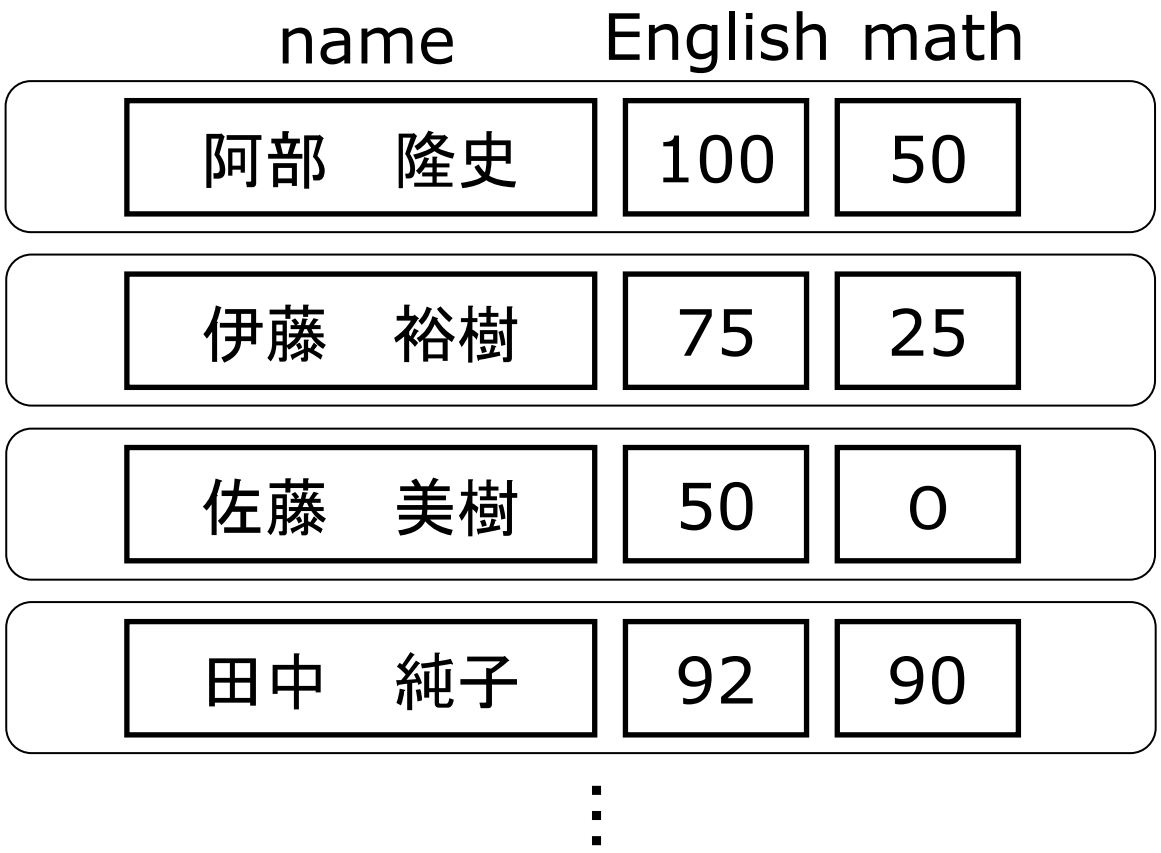
```
英語の平均点=75
数学の平均点=25
```

クラスの必要性②

氏名と英語・数学の得点は、
一人の学生が持つデータとして整理すべき。

配列と違って、異なるデータをひとつにまとめることができる。
データをひとつにまとめたからといって、
何か新しいことができるわけではないが、他人に分かりやすい！

クラスによるデータメンバの記述




```
class Student {
   public:
        string  name; //名前
        int English;    //英語の得点
        int math;       //数学の得点
};
```

英語と数学の平均点を計算するプログラム(クラス①)

```
class Student {
    public:
        string name;  //名前
        int English;    //英語の得点
        int math;       //数学の得点
};

int main(){

        Student abe, itou, sato; //Stringと同じで宣言が必要。

        abe.name="阿部";    abe.English=100;     abe.math=50;
        itou.name="伊藤";   itou.English=75;     itou.math=25;
        sato.name="佐藤";   sato.English=50;     sato.math=0;
        int averageEnglish=(abe.English+itou.English+sato.English)/3;
        int averagemath=(abe.math+itou.math+sato.math)/3;
        cout << "英語の平均点=" << averageEnglish << "¥n";
        cout << "数学の平均点=" << averagemath << "¥n";

        return 0;
}
```

実行例

```
英語の平均点=75
数学の平均点=25
```

英語と数学の平均点を計算するプログラム(クラス②)

```
class Student {
    public:
        string name;  //名前
        int English;    //英語の得点
        int math;       //数学の得点
};

int main(){

    Student gakusei[100]; //配列で宣言することも可能。

    gakusei[0].name="阿部";   gakusei[0].English=100;   gakusei[0].math=50;
    gakusei[1].name="伊藤";   gakusei[1].English=75;    gakusei[1].math=25;
    gakusei[2].name="佐藤";   gakusei[2].English=50;    gakusei[2].math=0;
    int averageEnglish=(gakusei[0].English+gakusei[1].English+gakusei[2].English)/3;
    int averagemath=(gakusei[0].math+gakusei[1].math+gakusei[2].math)/3;
    cout << "英語の平均点=" << averageEnglish << "¥n";
    cout << "数学の平均点=" << averagemath << "¥n";

    return 0;
}
```

実行例

```
英語の平均点=75
数学の平均点=25
```

英語と数学の平均点を計算するプログラム(クラス②)

```
class Student {
    public:
        string name;  //名前
        int English;    //英語の得点
        int math;       //数学の得点
};

int main(){

        Student gakusei[100];

        gakusei[0].name="阿部";    gakusei[0].English=100;     gakusei[0].math=50;
        gakusei[1].name="伊藤";    gakusei[1].English=75;      gakusei[1].math=25;
        gakusei[2].name="佐藤";    gakusei[2].English=50;      gakusei[2].math=0;
        int averageEnglish=(gakusei[0].English+gakusei[1].English+gakusei[2].English)/3;
        int averagemath=(gakusei[0].math+gakusei[1].math+gakusei[2].math)/3;
        cout << "英語の平均点=" << averageEnglish << "¥n";
        cout << "数学の平均点=" << averagemath << "¥n";

        return 0;
}
```

プログラムを複数人で分担して開発する場合、
他の開発者が得点の値を利用するのは良くても
値を書き換えてほしくないことがある。
➡クラスはそのようなことにも対応できる

実行例

```
英語の平均点=75
数学の平均点=25
```

英語と数学の平均点を計算するプログラム(クラス③)

```
class Student {
    private:
        string name;    //名前
        int English;    //英語の得点
        int math;       //数学の得点
    public:
        Student(string a, int b, int c){ name=a; English=b; math=c; }
        //上記のデータメンバの初期値を設定する関数(コンストラクタ)
        string get_name( ){ return name; }   //名前を返す関数
        int get_English( ){ return English; }  //英語の得点を返す関数
        int get_math( ){ return math; }        //数学の得点を返す関数
};

int main(){

    Student abe("阿部", 100, 50);        //Studentクラスabeの初期化
    Student ito("伊藤", 75, 25);         //Studentクラスitoの初期化
    Student sato("佐藤", 50, 0);         //Studentクラスsatoの初期化

    int averageEnglish=(abe.get_English()+ito.get_English()+sato.get_English())/3;
    int averagemath=(abe.get_math()+ito.get_math()+sato.get_math())/3;
    cout << "英語の平均点=" << averageEnglish << "¥n";
    cout << "数学の平均点=" << averagemath << "¥n";

    return 0;
}
```

private宣言すると、gakusei[0].nameではアクセスできないようになる。

➡get_nameという名前を返すメンバ関数を作成し、この関数をpublicにしアクセスできるようにする。

実行例

```
英語の平均点=75
数学の平均点=25
```

クラスの定義

```
class クラス名{
      private:    // 省略してもpublicまでのメンバはすべてprivateになる
                  メンバの宣言（非公開部）
                        主にデータメンバ
      public:
                  メンバの宣言（公開部）
                        主にメンバ関数の定義
};
```

上記のように、データメンバは非公開にして、メンバ関数を公開することにより、他のクラスからデータメンバへの直接のアクセスを制限することができる。

データメンバの値を他のクラスに対して公開したい場合には、単一のデータメンバの値を取得して返却するゲッタと呼ばれるメンバ関数を作成し、公開すればよい。

また、データメンバの値を他のクラスから設定したい場合には、データメンバの値を設定するセッタと呼ばれるメンバ関数を作成し、公開すればよい。

宣言とデータメンバ・メンバ関数の呼び出し方

オブジェクト（インスタンス）の宣言

クラス名　オブジェクト名；

例：Student abe, gakusei[100];

データメンバの呼び出し

オブジェクト名．データメンバ名；

例：abe.math;　//阿部の数学の得点

//但しデータメンバmathがpublicでないとアクセスできない

メンバ関数の呼び出し

オブジェクト名．メンバ関数名();

例：abe.get_math();　// 阿部の数学の得点を返すメンバ関数

//但しメンバ関数get_math()がpublicでないとアクセスできない

クラスの例(複素数)

```
class Complex {
        double  real, imaginary;   // real：実数部、imaginary:虚数部
    public:
        Complex (double a, double b) { real=a; imaginary=b;}      //コンストラクタ
        void print() {    // 複素数をa+biまたはa-biの形式で出力
            cout << real;
            if(imaginary>=0) { cout << '+'; }
            cout << imaginary << 'i' << "¥n";
        }

};

int main(){
        Complex d(1,2), e(3,-4);          //Complex型の変数dとeの宣言および
        d.print();          //複素数dの値を出力          コンストラクタを使った初期化
        e.print();          //複素数eの値を出力
        return 0;
}
```

実行例

```
1+2i
3-4i
```

コンストラクタとデストラクタ

- コンストラクタ（constructor）
  - クラス名と同名のメンバ関数
  - オブジェクト生成時に自動実行される関数。
  - オブジェクトのデータメンバの初期化が主な目的。（データメンバ変数は、宣言時に初期化できないから）。
  - リターン型を持たない。
  - 多重定義すれば、色々な方法でオブジェクトを初期化可能。
  - 明示的に呼び出すと、新しいオブジェクトを生成する。

- デストラクタ（destructor）
  - クラス名の前に~(tilde:チルダ)を付けた名前のメンバ関数
  - オブジェクト消滅時に自動実行される関数。
  - リターン型を持たない。
  - 多重定義はできない。

クラスの例(コンストラクタをちょっと修正)

```
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;}//コンストラクタ
        void print(); // プロトタイプ宣言のみ行い、クラスの外で関数定義している
};
void Complex::print(){   //クラス外での関数定義
        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(1,2), e(3),f;          //Complex型の変数dとeとfの宣言および
        d.print();          //複素数dの値を出力          コンストラクタを使った初期化
        e.print();          //複素数eの値を出力
        f.print();          //複素数eの値を出力
        return 0;
}
```

実行例

```
1+2i
3+0i
0+0i
```

コンストラクタの引数に具体的な値を書いておくと、
引数の値が指定されなかった際に、
その値で初期化される。

プロトタイプ宣言とインライン関数

```
class Complex {
        double  real, imaginary;   // real：実数部、imaginary:虚数部
    public:
        Complex (double a, double b) { real=a; imaginary=b;}      //コンストラクタ
        void print(); // プロトタイプ宣言のみ行い、クラスの外で関数定義している
};
void Complex::print(){   //クラス外での関数定義
        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(1,2), e(3,-4);          //Complex型の変数dとeの宣言および
        d.print();     //複素数dの値を出力          コンストラクタを使った初期化
        e.print();     //複素数eの値を出力
        return 0;
}
```

インライン関数にすると、実行ファイルサイズは大きくなるが実行速度は速くなることが多い。
一方、クラスの定義と関数定義を別ファイルで管理し、関数定義はオブジェクトファイルで
供給することが多いため、上記のプロトタイプ宣言の形式のように、メンバ関数を
クラス定義の外部で記述する方が多い。

デストラクタの例

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

class Person {
    string  name;
    int  age;
  public:
    Person(string tmp="", int y=20){  // コンストラクタ
        name=tmp;   age=y;
        cout << "Object " << name << " has been generated.¥n";
    }
    ~Person(){   // デストラクタ
        cout << "Object " << name << " is released...¥n";
    }
    void print(){
        cout << "Name :  " << name << "¥n";
        cout << "Age  :  " << age << "¥n";
    }
};
```

Personクラスが宣言された時に、初期化の引数が指定されれば、その値をnameとageに格納。

指定されなければ、tmp="", y=20で初期化

オブジェクトが消滅する際に、「Object name(=変数) has been released.」を画面に出力する。

デストラクタの例(続き)

```
int main(void)
{
    Person a("Candy",21), b("Dick",18);
    // この生成の際にそれぞれのコンストラクタが実行
    a.print();
    b.print();

    {   // 内部ブロックIB
      Person a("Rolla");//aの生成とコンストラクタの実行
      a.print();
      b.print();
    }   // このブロック終端でaが消滅され、その際にデストラクタが実行される

    a.print();
    b.print();
    return 0;
}  // main関数の終了においてa,bが消滅されるが、その際にそれぞれ
   // デストラクタが実行される
```

実行例

```
Object Candy has been generated.
Object Dick has been generated.
Name :  Candy
Age  :  21
Name :  Dick
Age  :  18
Object Rolla has been generated.
Name :  Rolla
Age  :  20
Name :  Dick
Age  :  18
Object Rolla is released...
Name :  Candy
Age  :  21
Name :  Dick
Age  :  18
Object Dick is released...
Object Candy is released...
```

クラスの例(メンバ関数追加版)

```
class Complex {
        double  real, imaginary;   // 実数部realと虚数部imaginary
    public:
        Complex (double a, double b) { real=a; imaginary=b;}      //コンストラクタ
        Complex add(Complex c){//自分と引数cを加算する(インライン関数で記述)
            Complex sum(0,0);
            sum.real=real+c.real;       sum.imaginary=imaginary+c.imaginary;
            return sum;
        }
       void print(); // プロトタイプ宣言のみ行い、クラスの外で関数定義している
};
void Complex::print(){  //クラス外での関数定義
        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(1,2), e(3,4), f(0,0);         //Complex型の変数dとeとfの宣言および
        f=d.add(e);       //複素数eをdに加算しfに代入       コンストラクタを使った初期化
        f.print();          //複素数fの値を出力
        return 0;
}
```

実行例

```
4+6i
```

クラスのメンバ関数と普通の関数の違い

- 例：円の面積を求める関数menseki

仮引数は、
その関数に
必要なデータを渡す
ためのもの

```
double menseki(double r){
    return r*r*M_PI;
} // 普通の関数
```

メンバ関数は、
そのオブジェクト（クラス）の
データメンバを
使用することができる。

```
class Circle{
    private:
        double r;
    public:
        double menseki(){ return r*r*M_PI; };
};
```

クラスのオブジェクトの代入

クラスをクラスに「=」を使って代入することができる。

この際、各データメンバはコピーされる。

```
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;} //コンストラクタ
        void print(); // プロトタイプ宣言のみ行い、クラスの外で関数定義している
};
void Complex::print(){  //クラス外での関数定義
        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(0,0), e(3,4);       //Complex型の変数dとeの宣言および
        d.print();                    //複素数dの値を出力
        d=e;                          //複素数eをdに代入
        d.print();                    //複素数dの値を出力
        return 0;
}
```

実行例

```
0+0i
3+4i
```

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

クラスの配列の初期化方法①

これまでのコンストラクタの使い方は以下のように、
宣言と初期化が一緒だった。
しかし、複素数を配列でとりたいときにはどうしたらよいか？

```
class Complex {
        double  real, imaginary;   // real：実数部、imaginary：虚数部
    public:
        Complex (double a, double b) { real=a; imaginary=b;}//コンストラクタ
        void 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(1,2);//Complex型の変数dの宣言、コンストラクタを使った初期化
        d.print();          //複素数dの値を出力
        return 0;
}
```

実行例

```
1+2i
```

クラスの配列の初期化方法②

複素数を配列でとりたいときには、コンストラクタを多重定義する。
まず引数なしのコンストラクタで、宣言のみ行う。
次に、引数のあるコンストラクタを使って初期化する。

```
class Complex {
        double  real, imaginary; // real：実数部、imaginary：虚数部
    public:
        Complex ( ) { }//引数なしのコンストラクタ
        Complex (double a, double b) { real=a; imaginary=b;}//コンストラクタ
        void 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[10];                  //Complex型の変数10個の配列dの宣言
        for(int i=0;i<10;i++){
            d[i]=Complex(i, i*2.0);     //引数のあるコンストラクタを使った初期化
        }                               //整数に型変換するint(2.5)などと同じ
        d[5].print();                   //複素数d[5]の値を出力
        return 0;
}
```

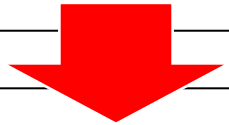
実行例

```
5+10i
```

コンストラクタのもう1つの使い方

複素数の加算を行う際にコンストラクタを使うと、
内部で代入用の複素数をとる必要がなくなる。

```
class Complex {
        double  real, imaginary;   // 実数部realと虚数部imaginary
    public:
        Complex (double a, double b) { real=a; imaginary=b;}      //コンストラクタ
        Complex add(Complex c){ //自分と引数cを加算する
            Complex sum(0,0);
            sum.real=real+c.real;       sum.imaginary=imaginary+c.imaginary;
            return sum;
        }
};
```



```
class Complex {
        double  real, imaginary;   // 実数部realと虚数部imaginary
    public:
        Complex (double a, double b) { real=a; imaginary=b;}      //コンストラクタ
        Complex add(Complex c){ //自分と引数cを加算する
            return Complex(real+c.real, imaginary+c.imaginary);
        }   //実数部がreal+c.realで虚数部がimaginary+c.imaginaryである
};          //一時オブジェクトを作り、これをreturnする
```

クラスの配列の初期化方法③

複素数を配列でとりたいとき、デフォルトコンストラクタでも実現できる。
しかし、多重定義でも実現できる場合には、
多重定義のほうが、他に与える影響が少ないので推奨されることが多い。

```
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
        void 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[10];                  //Complex型の変数10個の配列dの宣言
        for(int i=0;i<10;i++){
            d[i]=Complex(i, i*2.0);     //引数のあるコンストラクタを使った初期化
        }                               //整数に型変換するint(2.5)などと同じ
        d[5].print();                   //複素数d[5]の値を出力
        return 0;
}
```

実行例

```
5+10i
```

コンストラクタのさらにもう1つの使い方

Complexクラスには、特に定義をしなくても、
Complexクラスを引数にとるコピーコンストラクタ(Complex(const Complex & x))
が暗黙のうちに定義される。

```
class Complex {
        double  real, imaginary; // real：実数部、imaginary：虚数部
    public:
        Complex (double a, double b) { real=a; imaginary=b;}//コンストラクタ
        void 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 z(1,2);     //Complex型の変数zを1+2iの初期化
        Complex y(z);       //オブジェクトzの全データメンバをyにコピーする
        y.print();          //オブジェクトyの値を出力
        return 0;
}
```

実行例

```
1+2i
```