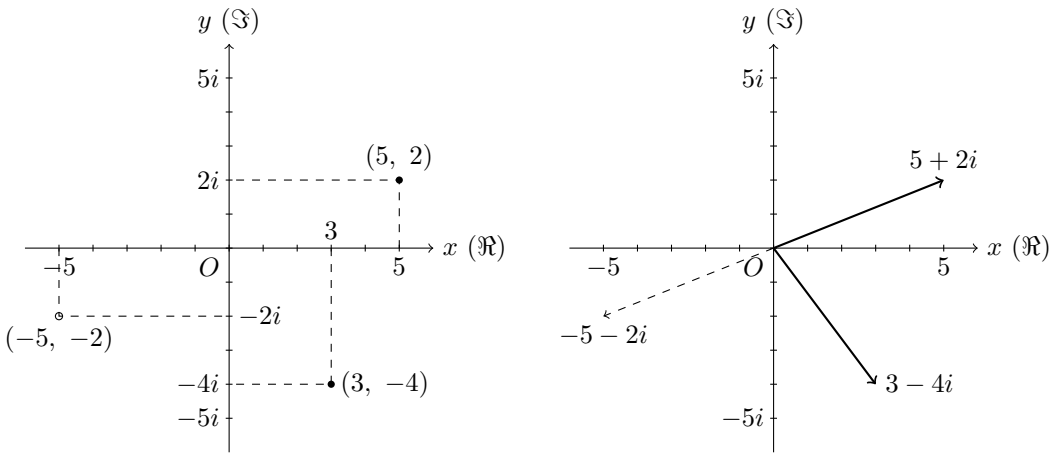
8.2 複素数

実数と虚数はそれぞれの数直線上に表せるが、複素数とは実際どんな数なのだろう。複素数 0 が両数直線に共通していることから、数直線と虚数直線を 0 で交差させた**複素平面**[1]が作れる。複素数は複素平面上で表される数なのである。複素数 $a+bi$ を、実数単位の 1 と虚数単位の i を強調して書くと $a\cdot 1+b\cdot i$ であるから、複素数はそれぞれの単位の係数である $(a,\ b)$ の組で表してもよい。a を複素数の**実部**、b を複素数の**虚部**という。このことから、複素数は複素平面に自然な形で示すことができる。



実は、複素数は 2 通りの見方ができる。平面上の点として見るか、平面上の**ベクトル**として見るか、である。実数も数直線上で点やベクトルで表せたので似たようなものだ。図は $(3,\ -4)$ と $3-4i$ などを異なる見方で描いたが、$(3,\ -4)$ と $3-4i$ はまったく同じものを表していることを注意しておく。また実数は、数直線上で正反対の数は正負が異なっていたように、複素数も原点 O に対して正反対の点・ベクトルで正負が異なる。

それでは、**C++**で複素数はどのように扱うのだろう。さすがの **C++**でも、`int c = 3;` みたいに、`complex c = 3-4i;` と認識してくれるほど気が利いているわけではない。そこで複素数用のクラスが必要になる。

programming list [Complex'.cpp]

```
1: #include <iostream>
2:
3: class Complex {
4: public:
5:     double re;
6:     double im;
7:
```

[1] ガウス平面とも言う。

```
 8:     Complex(double x, double y) {
 9:         re = x;
10:         im = y;
11:     }
12: };
13:
14: int main() {
15:     Complex u = Complex(3,-4);
16:     Complex v = Complex(5, 2);
17:
18:     std::cout << u.re << " + " << u.im << "i" << std::endl;
19:     std::cout << v.re << " + " << v.im << "i" << std::endl;
20:
21:     return 0;
22: }
```

この場合のクラスは、新たな型を定義するものである。この旅では int 型などを用いて、int n; のように整数変数などを使用してきた。ところが複素数は実部、虚部の組を持つので、これまでのように単純に代入するわけにはいかない。そこで、複素数が扱えるように型を決めようということである。

3:行目より、名称は Complex 型としたのが分かる。型クラスは大文字で書き始める。4:行目で public: を宣言したのは、どこからでも参照できるようにするためだ。そして複素数には実部、虚部があるので、それらを 5:, 6:行目で double で宣言している。べつに float でもいいけど。

ただ、このままでは使えない。実部、虚部に値を代入できないからである。そのために、代入する機能を持つ Complex() 関数が必要になる（この関数はクラスに属する**メンバ関数**であるが、この場合は初期化のための**コンストラクタ**として機能する）。Complex() は 8:–11:行目にあるように、2 個の実数引数をとり、実部 re、虚部 im にそれぞれ代入する機能を持ったことになる。これで Complex() は、一応の役目が果たせる。また、クラスは関数と違うので、12:行目のように }; で閉じていることに注意をしてもらいたい。

Complex 型による初期化は、int 型の変数 n を 3 で初期化するのと同様だが、引数を 2 個とるので 15:, 16:行目のようにすればよい。これで u、v にはそれぞれ 3-4i、5+2i が代入されたことになるのだが、実際は u の re に 3 が、u の im に-4 が代入されているに過ぎない。そのため、それなりの複素数であることを見るには、18:, 19:行目のような出力をする必要がある。

しかし、これだけでは複素数の計算ができない。計算のための関数を加えよう。

programming list [Complex.cpp]

```
 1: #include <iostream>
 2:
 3: class Complex {
 4: public:
```

```
    double re;
    double im;

    Complex(double x, double y) {
        re = x;
        im = y;
    }
};

Complex Add(Complex x, Complex y) {
    return Complex(x.re + y.re, x.im + y.im);
}

Complex Sub(Complex x, Complex y) {
    return Complex(x.re - y.re, x.im - y.im);
}

Complex Mul(Complex x, Complex y) {
    return Complex(x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re);
}

Complex Div(Complex x, Complex y) {
    return Complex(
                (x.re * y.re + x.im * y.im) / (y.re * y.re + y.im * y.im),
                (-x.re * y.im + x.im * y.re) / (y.re * y.re + y.im * y.im));
}

int main() {
    Complex u = Complex(3,-4);
    Complex v = Complex(5, 2);

    Complex a = Add(u, v);
    Complex s = Sub(u, v);
    Complex m = Mul(u, v);
    Complex d = Div(u, v);

    std::cout << u.re << " + " << u.im << "i" << std::endl;
    std::cout << v.re << " + " << v.im << "i" << std::endl;
    std::cout << std::endl;
    std::cout << a.re << " + " << a.im << "i" << std::endl;
    std::cout << s.re << " + " << s.im << "i" << std::endl;
    std::cout << m.re << " + " << m.im << "i" << std::endl;
    std::cout << d.re << " + " << d.im << "i" << std::endl;

    return 0;
}
```

プログラムが妙に長いのは、複素数を一つ一つ `Complex` 型で初期化し、ご丁寧に和・差・積・商を一つずつ計算および表示している `main()` のせいだ。

計算のための関数は 15:–29:行目で定義した。和・差・積・商用にそれぞれ `Add()`、`Sub()`、`Mul()`、`Div()` である。中身は何のことはない、複素数の計算規則を `.re` プロパティと `.im` プロパティを用いて書き直しただけである。ちょっと、見づらいのは仕方ないけれど。で、これで、34:–37:行目のように四則計算を行うのである。

ただし、これでは計算ができるだけで、計算機能をもった `Complex` クラスを作ったことにならない。単にプログラムに計算用の関数を追加しただけである。使い勝手をよくするなら `Complex` クラス内に計算用の関数を用意する必要がある。

以上のように、**C++**では自らクラスを作ることができる。もし、何か特別な操作をしたくなったら、それ用のクラスを自分で作ればよい。ところで、この地まで見学に来てなんだが、**C++**にはちゃんとした複素数クラスが存在している。練習にもならない私のクラスを使うより、次節でれっきとしたクラスに乗り換えて旅を続けよう。