构造函数

Class本体

- inline内联函数：在类的内部声明的时候及进行了定义
- 访问级别：public:可以外部直接访问，调用

  private：不可以直接访问，调用
  - 构造函数：
    1. 函数名称一定要跟类的名称一致，可以拥有参数，无返回值类型。构造函数是用来创造对象的
    2. 构造函数特有的语法：初值列，初始列

```
class complex
{
    public:
        complex (double r = 0,double i = 0)
        :re(r),im(i)//将r的值赋给re，将i的值赋给im
        {}
    private:
    double re,im.
}
构造函数不需要调用，只需要创建对象。
complex c1(2,1);
```

3.构造函数可以有很多个（重载）

4.const member functions

```
class complex
{
    public:
        complex(int m,int n):re(m),im(n){}
        double real() const{return re;}//注意这里要写const
                                        //class 内部分为会改变数据的和不会改变数据的，不会改
                                        如果不加const，下面这种情况调用的时候就会报错
                                        const complex c1（1，2）
                                        //如果还不理解就看侯捷老师的视频P4：09:00

        double imag() const{return im;}
    private:
        double re,im;
    }
    调用：
{
```

```
    complex c1(2,1);
    cout << c1.real();
    cout << c1.imag();
}
下面这种调用就会出错，如果在类里面不写const
{
    const complex c1(2,1);
    cout << c1.real();
    cout << c1.imag();
}
```

5.参数传递：pass by value vs pass by reference(to const)

值传递与引用：在这里主要知道传值的时候最好使用引用来传

6.返回值的传递也尽量用reference(引用)

7.友元

```
class complex
{
    public:
        double real() const{return re;}//注意这里要写const
        double imag() const{return im;}
    private:
        double re,im;

    frined complex& _doapl(complex&,const complex&);//友元函数
}

友元函数可以取到Private成员
```

8.返回值类型为引用的格式

```
double &
int &
class &
```

9.引用的实质就是转换成指针

```
int a =1;
int& arr = a; //编译器自动转换成 int * const arr = &a
定义引用时其实就是将引用跟初始值绑定到一起，而不是将初始值拷贝给引用
```

```
在调用引用的时候，可以将函数引用看成一种数据类型。比如：函数通过引用传递参数
可以写成（int& r），也可写为（int&）
```

10.C++中对象的含义

```
C++中将通过类定义出来的变量叫做对象
宏观一些，int这种数据类型也可以被叫做类，因为类就是那些有共同属性的一组事物
```

11.相同class的各个object互为friends（友元）

```
class complex
{
    public:
        double real() const{return re;}//注意这里要写const
        double imag() const{return im;}
        int func(const complex& param)//这里，param也是complex类的对象，所以可以访问
            {return param.re;}          //它的私有属性（友元）
    private:
        double re,im;
    frined complex& _doapl(complex&,const complex&);//友元函数
}
调用：
{
    complex c1(2,1);
    complex c2;
    c2.func(c1)
}
注意：在c2.func()函数参数调用的时候就能看出来为什么这个函数在定义的时候参数部分
const complex& param写法的原由
就相当于complex& param = c1;这就是引用初始化的方法

```

12.什么情况下不能使用引用传递参数和传递返回值

- **什么情况下不能return by reference**

```
返回值的引用不能传递一个临时的地址，也就是在函数内定义的变量，因为在函数运行之后空间就释放了
一个拥有临时地址的变量不能用引用的方式返回
```

13.operator overloading (操作符重载)

格式：

- this指针：谁调用这个函数，那个谁(对象)就是这个指针（任何一个成员函数都有一个隐藏的this指针）：

例如

```
c2 += c1;
在这里c2就是this指针
```

- 操作符重载：将操作符作用在左边身上
- 编译器会自动将操作符左边变量的地址传到this指针

```
操作符重载的例子：
思考：
1、会什么这个函数第一个参数用指针？第二个参数用引用

2、为什么操作符重载中第一个参数是指针，但是按照操作符重载的规则，传入的是对象？
3、思考形参的传递 ，要一直想着int& arr = a；a是对象 引用是对象的另外一个标签
4、操作符重载函数的返回值类型为什么是 complex& ,因为考虑到了链式运算（c1+=c2+=c3）
5、因为操作符重载函数的返回值类型是complex&,则_doapl函数返回值类型也为complex&
inline complex& _doapl(complex *ths,const complex& r)
{
    ths->re += r.re;//
    ths->im += r.im;:
    return *ths
}
/*
操作符重载：+= 操作符左边是this，右边是r
*/
inline complex& complex::operator += (const complex& r)
{
    return _doapl(this,r)//右边加到左边
}
/*
相当于隐藏了this（但是不能写出来this）
inline complex& complex::operator += (this,const complex& r)
{
    return __doapl(this,r);
}
*/
调用：
{
    complex c1(1,2);
```

```
    complex c2;

    c2 += c1;//调用了操作符的重载 将右边作用在左边身上
}
```

```
omplex
{
        public:
                complex(double r = 0, double i = 0) : re (r), im (i)
                { }
                double real() const { return  re;}
                double imag() const { return  im;}
        complex & operator += (complex &);
        private:
                double re,im;
        friend complex &doapl(complex *,complex&);
};
inline complex& doapl(complex * ths,complex & r)
{
        ths->re += r.re;
        ths->im += r.im;

        return *ths;
}
inline complex& complex :: operator += ( complex& r)//这里不能有const，会编译出错，我现在也
{
        return doapl(this,r);
}
int main(int argc ,char ** argv)
{
        complex c1(1,2);
        complex c2(2,4);
        const complex c3(4,5);
        c1 += c2;
        std::cout << c1.real() << std::endl;
        std:: cout << c1.imag() <<std::endl;

        std:: cout << c3.imag() <<std::endl;
```

```
34
35          return 0;
36  }
```

- inline表示内联函数

在Class内部定义的函数是内联函数

```
1  http://c.biancheng.net/view/2201.html
```

- return by reference 语法分析

传递者无需知道接收者是以referenc（引用）形式接收

```
1   inline complex& _doapl(complex *ths,const complex& r)
2  {
3      ths->re += r.re;
4      ths->im += r.im;
5      return *ths
6  }
7  返回的是一个对象，但是 返回值类型是引用。这就是 传递者无需知道接收者是以什么方式接收
8
9  inline complex& complex::operator += (const complex& r)
10 {
11     return _doapl(this,r)
12 }
13  c2 += c1;//调用了操作符的重载
14  这里也是
```

在我看来，在传形参的时候用另外一种角度更能理解

```
1  比如：
2  inline complex& complex::operator += (const complex& r)
3  {
4      return _doapl(this,r)
5  }
6   c2 += c1;//调用了操作符的重载
7   c1对应的是(const complex& r)；可以理解为：complex& r = c1;
8   c1是对象，r是引用；
9
```

14.非成员函数的重载（跟成员函数的区别是没有this指针）

```
inline conplex operator + (const complex& x,const complex& y)
{
    return complex (real (x) + real(y),imag(x) + imag(y);)
}
inline conplex operator + (const complex& x,double y)
{
    return complex (real (x) + real(y),imag(x) + imag(y);)
}
inline conplex operator + (cdouble x,const complex& y)
{
    return complex (real (x) + real(y),imag(x) + imag(y);)
}
注意这里必须返回的是value 不能返回reference，因为返回的值不是一个真实存在的地址，
函数运行完毕之后就会被释放
```

15.class之外的各种定义

```
①创建临时对象：typename():变量类型加括号。注意类也是一种数据类型
    临时对象在运行完就会被释放
②negate反相（取反）:这个也是操作符重载
inline complex operator + (const complex& x)
{
    return x;
}
inline complex operator -(const complex &x)
{
    return complex (-real (x),-imag (x));
        }
③特殊的操作符重载只能作为全局函数，比如<<。因为成员函数一定是将操作符重载作用在
操作符左边的对象。
如果操作符重载在成员函数里面调用的时候只能这样调用：c1<<count;但是不符合使用习惯
ostream& operator << (ostream& os,const complex& x)
{
    return os << '(' << real(x) << ', '
                << imag (x) << ')';
}
调用：
{
    complex c1(2,1);
```

```
    cout << c1;    //输出c1(2,-1)
}
```

16.拷贝函数

```
class String{
    public:
        String(const char* cstr = 0);//构造函数
        String(const String& str);//构造函数，接受的是自己类型的对象，所以是拷贝构造
        String& operator=(const String &str);//拷贝赋值，如果类带着指针，一定要写这个函数
        ~String();//这个类的对象死亡的时候析构函数就会被调用
        char * get_c_str()const {return m_data;}
    pprivate:
        char *m_data;
}
```

17.构造函数和析构函数

```
inline String::string(const char *cstr =0)
{
    if(cstr){
        m_data = new char[strlen(cstr)+1];
        strcpy(m_data,cstr);
    }
    else{ //未指定初值
        m_data = new char[1];
        *m_data = '\0';
    }
}
inline String::~String()//析构函数
{
    delete[] m_data;
}
```

18.class with pointer member 必须有拷贝构造和拷贝赋值

- 必须有拷贝构造是因为为了避免浅拷贝

浅拷贝是只拷贝了指针，指针后面的地址里面内容没有拷贝

深拷贝

```
inline String ::String(const String &str)
{
```

```
    m_data = new char[strlen (str.m_data) +1];
    strcpy(m_data,str.m_data);
}
```

- 拷贝赋值（先把左边的清空，然后申请右边一样大小的空间，再把右边的拷贝过来）

```
inline
String& String::operator=(const Strings &str)
{
    if(this == &str)
    return *this;//检测自我赋值（大家风范）这个一定要写
    delete[] m_data;//左边的清空
    m_data = new char[strlen(str.m_data)+1];//申请右边一样的空间
    strcpy(m_data,str.m_data);
    return *this;
}
```