6 复习complex的实现过程 为什么我们把+定义为非成员函数

标准库里的代码

```
#ifndef __COMPLEX__
#define __COMPLEX__
...
#endif
class complex
{ public: complex (double r=0, double i=0)
    : re(r), im(i) { }
  complex& operator += (const complex&);
  double real() { return re; }
  double imag() { return im; }
private:
  double re, im;
  friend complex&
  __doapl (complex*, const complex&)
};
complex::operator += (const complex& r)
{ return __doapl (this, r);
}
inline complex&
__doapl (complex *ths, const complex& r)
{ ths->re += r.re;
  ths->im += r.im;
  return *ths;
}
```

全局函数

```
inline complex
operator + (const complex& x, const complex& y)
{
  return complex (real(x)+real(y),
                  imag(x)+imag(y));
}
```

这里的接口为什么不采用reference.

这里用临时对象返回.

再考虑重构

正是因为我们需要重构.

所以需要定义非成员函数

```
cout << c1;
#include<iostream>
operator << (ostream& os,
             const complex& x)
{ os << '(' << real(x) << ',' <<
     imag(x) << ')';
}
```

其return type是什么?

ostream&

传出去的东西如果不是local object. 则可return reference

7. 三大函数: 拷贝构造. 拷贝复制. 析构

class with pointer member(s)

string

```
#ifndef __MYSTRING__
#define __MYSTRING__
class string
{
};
#endif
```

```
int main()
{
  String s1();
  String s2("hello");
  String s3(s1);
  cout << s3 << endl;
  s3 = s2; 赋值. 拷贝
  cout << s3 << endl;
}
```

如何处理拷贝构造与拷贝赋值

复数没有写, 但是编译器会自己构造

但是string不行. 因为其中含有指针的复制

如果你的类中有指针. 自然需要动态分配.

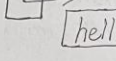
需要析构函数

```
class String
{ public:
  String(const char* cstr=0);
  String(String& str);  构造函数 拷贝构造
  String& operator= (const String& str);
                        拷贝赋值
  ~String();  析构函数
              当对象死亡时被调用
private:
  char* m_data;
};
```

字符串长度

size

结束符



② ③ ④ Big Three

```
inline
String::String(const char* cstr=0)
{ if (cstr){
    m_data = new char[strlen(cstr)+1];
    strcpy(m_data, cstr);
  }
  else{
    m_data = new char[1];
    *m_data = '\0';
  }
}
inline
String::~String()  不执行则意味着内存泄露
{                  memory leak
  delect[] m_data;
}
```

Big Three

拷贝构造 拷贝赋值 析构函数

class with pointer member(s) 必须有 copy ctor 和 copy op=

如果不这样做，编译器为你生成 会一个bit一个bit地复制

指针两个都指向了一个位置 那么其中被赋值者产生内存泄露

此时是浅拷贝 而我们写的是深拷贝

```
inline
String::String(const String& str)
  { m_data = new char[strlen(str.m_data)+1];
    strcpy(m_data, str.m_data);
  }
```

拷贝构造函数

不一般不会主动自我赋值

但是隐式

copy assignment operator (拷贝赋值函数)

s2 = s1

1. 清空s2
2. 分配
3. 赋值

```
inline
String& String::operator=(const String& str)
{ if (this == &str)        检测自我赋值
    return *this;          self assignment
  1. delete[] m_data;
  2. m_data = new char[strlen(str.m_data)+1];
  3. strcpy(m_data, str.m_data);
     return *this;
}
```

如果不写检测自我赋值 会发现拷贝出错 ✓

会发不明确行为

(undefined behavior)