<<

重载要点：

1 函数返回值为一个 ostream的引用

2 第一形参 为输出流的引用

第二参数 为自定义类的常量引用

3 输出运算符必须是非成员函数 because

如果写成成员函数就是下面这种写法

Sales\_data data; data<<cout;

但是我们一般是这样 cout<<data;

ostream &operator<<(ostream & os, const Sales\_data& data){

return os;

}

>>

重载要点：

1 函数返回值为一个 ostream的引用

2 第一形参 为输出流的引用

第二参数 为自定义类的非常量引用，因为这是要读入到这个类中

3 输出运算符必须是非成员函数 because

一般我们调用这种函数是

Sales\_data data; data<<<cout;

如果是成员函数 就成了 data.<<这种形式那么就很别捏；

istream &operator<<(istream & is, const Sales\_data& data){

return is;

}

算数和关系运算符 （一般定义为非成员函数）

Sales\_data operator+(const Sales\_data & a,const Sales\_data & b ){

Sales\_data sum=a;

sum +=b;

return sum; //我们一般习惯性返回一个临时变量

}

相等运算符

bool operator==（(const Sales\_data & a,const Sales\_data & b ）{

return a.song==b.song&&a.xiao==b.xiao;

}

bool operator!=（(const Sales\_data & a,const Sales\_data & b ）{

return a.song!=b.song&&a.xiao!=b.xiao;

}

大于小于运算符

一般只有这个类有逻辑关系我们才定义；

赋值运算符（一般为成员函数）

我们先看一下他的相关操作

vector<string>v;

v={12,13,14};//这个就是他的作用；

Sales\_data &Sales\_data：：operator=（initiallizer\_list<string> il)

{

this;

}

复合赋值运算符

Sales\_data &Sales\_data：：operator+=（const Sales\_data &rhs）{

return \*this；

}

下标运算符

一般需要定义两个版本一个是常量和非常量

string &operator[](std::size\_t n){

return elements[n];

}

const string &operator[](std::size\_t n)const{

return elements[n];

}//常量版本是为了防止为常量赋值。

递增递减运算符

一般争对迭代器的类

一般为了区别前置递增和后置递增

我们把后置递增增加一个int 的形参

StrBlobPtr &operator++(){

check(currr,"fasfasfasf"); //我们先进行检查

++currr;

return \*this;

}//前置递增

StrBlobPtr &operator++(int){

StrBlobPtr ret=\*this；

++\*this； //从这里我们可以看出我们其实是调用了前置递增来实现后置递增

return ret;

}

成员访问运算符

string & operator\*()const{

auto p=check(curr,"dereferences past end");

return (\*p)[curr] //(\*p)是对象所指的vector；

}

string \* operator->()const{

return &this->operator\*();

}

**函数调用运算符**

写在类中

int operator（）（int abc）const{

return abc<0 ?-abc:abc

}

实例

song a;

a(42);

**在这一个运算符下我们探讨了lambda表达式其实是被翻译成一个未命名类的未命名的对象**

stable\_sort(word.begin,words.end(),

[ ](const string &a,const string &b) {return a.size()<b.size()})

**这个lambda表达式被编译器看成下面这个类**

**默认只产生一个函数调用运算符**

class shorterstring

{

int operator（）（const string &a,const string &b）const{

return a.size()<b.size()；}

}

**如果采用引用来捕获变量那么就可以直接使用，但是你必须保证他存在，如果是拷贝**

**那么该表达式被翻译过来就会加上一个构造函数，同时会创建一个变量**

stable\_sort(word.begin,words.end(),

[4](const string &a) {return a.size()<sz})

class shorterstring

{

**shortcomp（size\_t n）:sz(n);**

int operator（）（const string &a）const{

return a.size()<sz；}

private:

**size\_t sz;**

}

**这里插入一个标准库定义的函数对象**

**我们就可以直接在算法中调用相关函数对象实现排序的简化**

**比如：**

sort（sev.begin(),svec.end(),greater<string>()}; **//具体看510页**

**可调用对象与function**

**c++中存在几种可调用对象 函数 函数指针 lambda表示式 bind创建对象**

**因为这几种可调用对象 实际上是几种不同的类型**

**看下面例子**

int add(int i,int j){return i+j；}

auto mod=【】(int i,int j) {return i+j；}

struct divide{

int operator（）(int i,int j) {return i+j；}

}

map<string , int (\*)(int ,int)> binops;

binops.insert({"+",add});  **true**

binops.insert({"+",mod});  **false 因为 int (\*)(int ,int)是函数类型 lambda表达式与其类型不匹配所以报错**

binops.insert({"+",divide);  **false 同一道理**

map<string , int (\*)(int ,int)> binops;

**把这个变成** map<string , function< int (\*)(int ,int)>> binops; **就可以了**

**function函数是一个模板 具体看P512面资料**

**1 类型转换运算符**

**这里涉及到两个函数**

**转换构造函数**

**类型转换运算符 的形式**  operator type（）const；

**一个构造函数接收一个不同于其类类型的形参，可以视为将其形参转换成类的一个对象。像这样的构造函数称为转换构造函数。**

class IntClass

{

private:

int value;

public:

**//转换int的转换构造函数**

IntClass(int intValue) {

value = intValue;  **其形参转换成类的一个对象**

}

operator int（） const { return value; **类型转换运算符**

}};

**注意这里都是隐式转换所以我们的类型转换符号不能有参数和返回值**

**想较与隐式转换我们可以使用显示转换**

**但是在条件语句中会把显示转换变成隐式转换 static\_cast**

song si;

static\_cast<int>(si)+3;

**我们要注意转换的二重定义性**

**这里注意一种**

**我们一般通过函数的调用方式就可以知道是成员函数还是非成员函数**

**但是当我们有类型转换函数时就可能产生二重性**

class song{

int data；

song （int=0）;  **//注意这是一种简化的转换**

song operator +（song &a,song&b）{return a+b};

operator int（）{return data； }

}

song a ,b;

auto c=a+b;**这里没有任何异议**

auto d=a+0;  **//如果0被转换成song类型就会执行成员函数，如果a被转换成int就会执行加法非成员函数所以会产生异议 我们需要注意**