Object – Oriented Programming

Week 10

Overloaded operators and Move Ctor

Weng Kai

Defining a stream extractor

从流中提取数据的操作(输出)
相对于用户来说，从用户提取数据，叫输入

- Has to be a 2-argument free function
  - First argument is an `istream&`
  - Second argument is a *reference* to a value

```
istream&
operator>>(istream& is, T& obj) {
    // specific code to read obj
   return is;
}
```

obj不是const，因为要写入

- Return an `istream&` for chaining

```
cin >> a >> b >> c;
((cin >> a) >> b) >> c;
```

Creating a stream inserter

- Has to be a 2-argument free function 输出cout
  - First argument is an ostream&
  - Second argument is any value

```
ostream&
operator<<(ostream& os, const T& obj) {
    // specific code to write obj
   return os;
}
```

- Return an `ostream&` for chaining

```
cout << a << b << c;
((cout << a) << b) << c;
```

Creating manipulators

操作符

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
    ...
    return out;
}
ostream& tab ( ostream& out ) {
    return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

没有operator，因为是自定义的，没有重载

Copying vs. Initialization

```
MyType b;
MyType a = b;
a = b;
```

Example: CopyingVsInitialization.cpp

Automatic operator= creation

- The compiler will automatically create a **type::operator=(type)** if you don't make one.
- *memberwise assignment*

成员逐一赋值

- Example: AutomaticOperatorEquals.cpp

Assignment Operator

- Must be a member function
- Will be generated for you if you don't provide one
  - –Same behavior as automatic copy ctor -- memberwise assignment
- Check for assignment to self
- Be sure to assign to all data members
- Return a reference to *this

```
A = B = C;
// executed as
A = (B = C);
```

```
MyClass& operator=(const MyClass& other) {
// 自我赋值检查
if (this == &other) {
  return *this;  // 如果是自我赋值，则直接返回当前对象
}

// 执行赋值操作
x = other.x;

return *this;
}
};
```

Skeleton assignment operator

框架形赋值运算符

```
T&   T::operator=( const T& rhs ) {
    // check for self assignment
    if ( this != &rhs)  {
        // perform assignment
    }
    return *this;
}
```

//This checks address vs. check value (*this != rhs)

Example: SimpleAssignment.cpp

Assignment Operator

- For classes with dynamically allocated memory declare an assignment operator (and a copy constructor)
- To prevent assignment, explicitly declare operator= as private

Value classes

- Appear to be primitive data types
- Passed to and returned from functions
- Have overloaded operators (often)
- Can be converted to and from other types
- Examples: Complex, Date, String

User-defined Type conversions

- A conversion operator can be used to convert an object of one class into
  - an object of another class
  - a built-in type
- Compilers perform implicit conversions using:
  - Single-argument constructors
  - implicit type conversion operators

Single argument constructors

```
class PathName {
    string name;
public:
  // or could be multi-argument with defaults
  PathName(const string&);
  ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;          // OK abc => PathName
```

参数是别的类型，用于类型转换；参数是自己这个类型就是拷贝构造

编译器会用abc将类型转换给xyz然后赋值给xyz

Example: AutomaticTypeConversion.cpp

Preventing implicit conversions

- New keyword: explicit

```
class PathName {
    string name;
public:
   explicit PathName(const string&);
   ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;         // error!
```

需要被显示调用；赋值时不会被编译器隐式调用

Example: ExplicitKeyword.cpp

Conversion operations

- Operator conversion
  - Function will be called automatically
  - Return type is same as function name

```
class Rational {
public:
    ...
    operator double() const;  // Rational to double
}
Rational::operator double() const {
  return numerator_/(double)denominator_;
}
Rational r(1,3); double d = 1.3 * r; // r=>double
```

General form of conversion ops

- X::operator *T* ()
  - –Operator name is any type descriptor
  - –No explicit arguments
  - –No return type
  - –Compiler will use it as a type conversion from X ⇒ *T*

C++ type conversions

- Built-in conversions
  - *Primitive*

```
char ⇒ short ⇒ int ⇒ float ⇒ double
                ⇒ int ⇒ long
```

  - *Implicit (for any type T)*

```
T ⇒ T&          T& ⇒ T          T* ⇒ void*
T[ ] ⇒ T*       T* ⇒ T[ ]       T ⇒ const T
```

- User-defined T ⇒ C
  - *if* `C(T)` *is a valid constructor call for* `C`
  - *if* ***operator*** `C()` *is defined for* `T`
- BUT
  - See: TypeConversionAmbiguity.cpp

两个都能实现类型转换，但不能同时存在

或者一个加explicit，不会被隐式调用

Do you want to use them?

- In General, no!
  - Cause lots of problems when functions are called unexpectedly.
  - See: CopyingVsInitialization2.cpp
- Use explicit conversion functions. For example, in class Rational instead of the conversion operator, declare a member function:

```
double toDouble() const;
```

Overloading and type conversion

- C++ checks each argument for a "best match"
- Best match means cheapest
  - –Exact match is cost-free
  - –Matches involving built-in conversions
  - –User-defined type conversions

Overloading

- Just because you can overload an operator doesn't mean you should.
- Overload operators when it makes the code easier to read and maintain.
- Don't overload && || or , (the comma operator)

对象初始化

```
//小括号初始化
string str("hello");

//等号初始化
string str = "hello";
```

```
//大括号初始化
struct Studnet
{
    char *name;
    int age;
};
Studnet s = {"dablelv", 18};//Plain of Data类型对象
Studnet sArr[] = {{"dablelv", 18}, {"tommy", 19}}; //
POD数组
```

列表初始化

```
class Test
{
    int a;
    int b;
public:
    Test(int i, int j);
};
Test t{0, 0};                          //C++11 only,
相当于 Test t(0,0);
Test *pT = new Test{1, 2};             //C++11 only,
相当于 Test* pT=new Test{1,2};
int *a = new int[3]{1, 2, 0}; //C++11 only
```

容器初始化

```
// C++11 container initializer
vector<string> vs={ "first", "second",
"third"};
map<string,string> singers ={ {"Lady Gaga",
"+1 (212) 555-7890"},{"Beyonce Knowles", "+1
(212) 555-0987"}};
```

type of function parameters and return value

way in

- void f(Student i);
  - a new object is to be created in f
- void f(Student *p);
  - better with const if no intend to modify the object
- void f(Student& i);
  - better with const if no intend to modify the object

way out

- Student f();
  - a new object is to be created at returning
- Student* f();
  - what should it points to?
- Student& f();
  - what should it refers to?

hard decision

```
char *foo()
{
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return p;
}
void bar()
{
    char *p = foo();
    printf("%s", p);
    delete p;
}
```

define a pair functions of alloc and free

let user take resp., pass pointers in & out

tips

- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer

Left Value vs Right Value

- 可以简单地认为能出现在赋值号左边的都是左值：
  - 变量本身、引用
  - *、[]运算的结果
- 只能出现在赋值号右边的都是右值
  - 字面量
  - 表达式
- 引用只能接受左值—>引用是左值的别名
- 调用函数时的传参相当于参数变量在调用时的初始化

右值引用

- int x=20; // 左值
- int&& rx = x * 2; // x*2的结果是一个右值，rx延长其生命周期
- int y = rx + 2; // 因此你可以重用它:42
- rx = 100; // 一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值
- int&& rrx1 = x; // 非法:右值引用无法被左值初始化
- const int&& rrx2 = x; // 非法:右值引用无法被左值初始化

右值参数

```
// 接收左值
void fun(int& lref) {
    cout << "l-value" << endl;
}
// 接收右值                                    构成重载
void fun(int&& rref) {
    cout << "r-value" << endl;
}

int main() {
    int x = 10;
    fun(x); // output: l-value reference
    fun(10); // output: r-value reference
}
```

```
void fun(const int& clref) {
  cout << "l-value const reference\n";
}
```

没有接受右值的函数时也能接受右值

DynamicArray

std::move()

```
vector<int> v1{1, 2, 3, 4};
vector<int> v2 = v1; // 此时调用用复制构造函数，v2是v1的副本
vector<int> v3 = std::move(v1); // 此时调用用移动构造函数
```

通过std::move将v1转化为右值，从激发v3的移动构造函数，实现移动语义

std::swap()

```
void swap(T& a, T& b) {
 T tmp{a}; // 调用用拷贝构造函数
 a = b; // 拷贝赋值运算符
 b = tmp; // 拷贝赋值运算符
}
```

```
void swap(T& a, T& b) {
 T temp{std::move(a)};
 a = std::move(b);
 b = std::move(tmp);
}
```

other ctor

using function

- The derived class is able to “using” functions of its parent class

```
class Base {
public:
    void f() {}
};

class Child : public Base {
public:
    using Base::f;
    void f(int i) {}
};
```

Delegating Ctor

- Put calling to another ctor in the initialization list.
- But this delegating ctor can not have other members initialized in its own initialization list
- To solve this, a private ctor can be used to provide initialization to other members.
- It is possible to create a chain of delegating ctors.