基于 Linux 的 C++

第十讲　操作符重载

提　纲

四则运算符重载

关系操作符重载

下标操作符重载

赋值操作符重载

流操作符重载

操作符重载总结

四则运算符重载

设计一个数偶类，定义专用的四则运算

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b) { }
  Couple operator+( const Couple & c );
  Couple operator*( const Couple & c );
private:
  int _a, _b;
};
```

四则运算符重载

```
Couple Couple::operator+( const Couple & c )
{
  Couple _t( this->_a + c._a, this->_b + c._b );
  return _t;
}
Couple Couple::operator*( const Couple & c )
{
  Couple _t( this->_a * c._a, this->_b * c._b );
  return _t;
}

int main()
{
  Couple a( 1, 2 ), b( 3, 4 ), c, d;
  c = a + b;          // 等价于 c = a.operator+(b)  ==> c( 4, 6 )
  d = a + b + c;      // 等价于 d = a.operator+(b).operator+(c)  ==> d( 8, 12 )
  c = a * b;          // 等价于 c = a.operator*(b)  ==> c( 3, 8 )
  return 0;
}
```

四则运算符重载

问题一：假设需要数偶倍乘运算（标量运算），将整数k同时乘到两个成员上。如何实现？

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b) { }
  Couple operator*( const int & k );
private:
  int _a, _b;
};
Couple Couple::operator*( const int & k )
{
  Couple _t( this->_a * k, this->_b * k );
  return _t;
}
```

四则运算符重载

```
int main()
{
  int k = 3;
  Couple a( 1, 2 ), b( 3, 4 ), c, d;
  c = a + b;        // 等价于 c = a.operator+(b)  ==> c( 4, 6 )
  d = a + b + c;  // 等价于 d = a.operator+(b).operator+(c)  ==> d( 8, 12 )
  c = a * b;        // 等价于 c = a.operator*(b)  ==> c( 3, 8 )
  c = c * k;        // 等价于 c = c.operator*(k)  ==> c( 9, 24 )
  d = d * 2;        // 等价于 d = d.operator*(2)  ==> d( 16, 24 )
  return 0;
}
```

四则运算符重载

问题二：参数必须是Couple类对象的常引用吗？

- 可以不使用引用，但会产生对象拷贝动作，降低效率
- 可以不是常引用，但无法限制函数内部对参数的修改
- 可以使用指针，但与常规数学公式使用方式不符

问题三：返回值必须是Couple类的对象吗？返回引用是否可行？

- 可以返回引用，但必须是全局对象或通过参数传递进去的Couple对象的引用，不能引用函数内部的局部变量
- **不建议使用引用类型的返回值**
- 需要将右操作数累加到左操作数上并返回左操作数时，此时应该重载加赋等操作符，减赋、乘赋、除赋与余赋类似

四则运算符重载

问题四：四则运算符必须重载为成员函数吗？

- 不。可以重载为类的友元函数或普通函数。注意：普通函数无法访问类的私有成员
- **建议重载为友元函数**

重载为友元函数

- 优势：显式具有双操作数，且格式一致；操作不局限于当前对象本身，且不要求左操作数必须为本类的对象
- 劣势：显式具有双操作数，不能省略左操作数

四则运算符重载

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b) { }
  friend Couple operator+( const Couple & c1, const Couple & c2 );
  friend Couple operator*( const Couple & c1, const Couple & c2 );
  friend Couple operator*(const Couple & c, const int & k );
  friend Couple operator*( const int & k, const Couple & c );
private:
  int _a, _b;
};
```

四则运算符重载

数偶倍乘运算重载的说明

- 应重载为类的友元函数
- 若非友元函数，当倍数为左操作数时，无法解析乘法运算，编译会出错
- 将左操作数k转换为Couple类的对象可以解决上述问题，但意义已不同
- 上述转换要求提供一个单参数的从整数到Couple类的构造函数，如果使用explicit修饰该构造函数，隐式类型转换会被禁止；虽然即使不禁止，很多编译器也不进行此转换
- 左右操作数不可互换，重载函数必须提供两个版本，它们的函数签名不同

四则运算符重载

```
Couple operator+( const Couple & c1, const Couple & c2 )
{
  Couple _t(c1._a + c2._a, c1._b + c2._b );
  return _t;
}
Couple operator*( const Couple & c1, const Couple & c2 )
{
  Couple _t( c1._a * c2._a, c1._b * c2._b );
  return _t;
}
Couple operator*( const Couple & c, const int & k )
{
  Couple _t( c._a * k, c._b * k );
  return _t;
}
Couple Couple::operator*( const int & k, const Couple & c )
{
  Couple _t( k * c._a, k * c._b );
  return _t;
}
```

四则运算符重载

```
int main()
{
  int k = 3;
  Couple a( 1, 2 ), b( 3, 4 ), c, d;
  c = a + b;       //  等价于 c = operator+(a, b)  ==> c( 4, 6 )
  d = a + b + c; //  等价于 d = operator+(operator+(a, b), c)  ==> d( 8, 12 )
  c = a * b;       //  等价于 c = operator*(a, b)  ==> c( 3, 8 )
  c = k * c;       //  等价于 c = operator*(k, c)  ==> c( 9, 24 )
  d = 2 * d;       //  等价于 d = operator*(2, d)  ==> d( 16, 24 )
  return 0;
}
```

关系操作符的重载

为数偶类定义专用的关系操作符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b){ }
  friend bool operator==( const Couple & c1, const Couple & c2 );
  friend bool operator!=( const Couple & c1, const Couple & c2 );
private:
  int _a, _b;
};
```

关系操作符的重载

```
bool operator==( const Couple & c1, const Couple & c2 )
{
  return (c1._a == c2._a) && (c1._b == c2._b);
}
bool operator!=( const Couple & c1, const Couple & c2 )
{
  return (c1._a != c2._a) || (c1._b != c2._b);
}

int main()
{
  Couple a( 1, 2 ), b( 3, 4 );
  if( a == b )
    cout << "a == b" << endl;
  else
    cout <<   “a != b" << endl;
  return 0;
}
```

下标操作符重载

下标操作符重载的场合与目的

- 如果对象具有数组成员，且该成员为主要成员，可以重载下标操作符
- 目的：以允许在对象上通过数组下标访问该数组成员的元素

下标操作符必须重载两个版本

- 常函数版本用于处理常量

数组下标越界错误

- 可以在重载函数中处理数组下标越界错误，或使用异常处理机制

下标操作符重载

为数偶类定义专用的下标操作符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) {  _a[0]=a,  _a[1]=b;  }
  int & operator[]( int index );
  const int & operator[]( int index ) const;
private:
  int _a[2];
};
```

下标操作符重载

```
int & Couple::operator[]( int index )
{
  if( index < 0 || index > 1 )
    throw std::out_of_range( "Index is out of range!" );
  return _a[index];
}
const int & Couple::operator[]( int index ) const
{
  if( index < 0 || index > 1 )
    throw std::out_of_range( "Index is out of range!" );
  return _a[index];
}
int main()
{
  Couple a( 1, 2 ), b( 3, 4 );
  cin >> a[0] >> a[1];
  cout << b[0] << " " << b[1] << endl;
  return 0;
}
```

赋值操作符重载

- 赋值操作符重载的一般形式
- 复合赋值操作符重载
- 赋值构造与拷贝构造
- 浅拷贝与深拷贝
- 移动语义（C++11）

赋值操作符重载的一般形式

为数偶类定义专用的赋值操作符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b){ }
  Couple( const Couple & c ) : _a(c._a), _b(c._b){ }
  Couple & operator=( const Couple & c );
private:
  int _a, _b;
};
```

赋值操作符重载的一般形式

```
Couple & Couple::operator=( const Couple & c )
{
  if( *this == c )
    return *this;
  _a = c._a,  _b = c._b;
  return *this;
}

int main()
{
  Couple a( 1, 2 ), b( 3, 4 );
  cout << a << endl;
  a = b;
  cout << a << endl;
  return 0;
}
```

复合赋值操作符重载

为数偶类定义专用的简写四则运算符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b){ }
  Couple( const Couple & c ) : _a(c._a), _b(c._b){ }
  Couple & operator+=( const Couple & c );
  Couple & operator*=( const Couple & c );
  Couple & operator*=( const int & k );
private:
  int _a, _b;
};
```

复合赋值操作符重载

```
Couple & Couple::operator+=( const Couple & c )
{
  _a += c._a,  _b += c._b;
  return *this;
}

Couple & Couple::operator*=( const Couple & c )
{
  _a *= c._a,  _b *= c._b;
  return *this;
}

Couple & Couple::operator*=( const int & k )
{
  _a *= k,  _b *= k;
  return *this;
}
```

递增递减操作符重载

为数偶类定义专用的递增递减操作符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b){ }
  Couple( const Couple & c ) : _a(c._a), _b(c._b){ }
  Couple & operator=( const Couple & c );
  Couple & operator++();      //  前缀递增
  Couple operator++( int );   //  后缀递增
private:
  int _a, _b;
};
```

递增递减操作符重载

```
Couple & Couple::operator++()
{
  ++_a,  ++_b;
  return *this;
}

Couple Couple::operator++( int t )
{
  Couple _t( *this );
  _a++,  _b++;
  return _t;
}
```

赋值操作符的返回值

- 除后缀递增递减操作符，应返回对象的引用，以于C++本身语义相符
- 返回对象需要额外的对象构造，降低效率
- 如果不需要使用返回值以进行连续赋值，可以将返回值设为void，但要注意此时重载的操作符语义与C++不符，不推荐

赋值构造与拷贝构造

赋值也是构造

拷贝、赋值与析构三位一体，一般同时出现

- **缺省赋值构造与拷贝构造为浅拷贝**
- **如果对象没有指针成员，缺省行为即可满足要求，无需实现或重载这三个函数**
- **如果对象有指针成员，一般需要重载这三个函数**

浅拷贝

```
class A
{
public:
  A() : _n(0), _p(NULL) { }
  explicit A( int n ) : _n(n), _p(new int[n]) { }
  A( int n, int * p ) : _n(n), _p(p) { }
  A( const A & that ) : _n(that._n), _p(that._p) { }
  A & operator=( const A & that ) { _n = that._n, _p = that._p; return *this; }
  virtual ~A() { if(_p){ delete[] _p, _p = NULL; } }
public:
  int & operator[]( int i );
  const int & operator[]( int i ) const;
private:
  int _n;
  int * _p;
};
```

浅拷贝

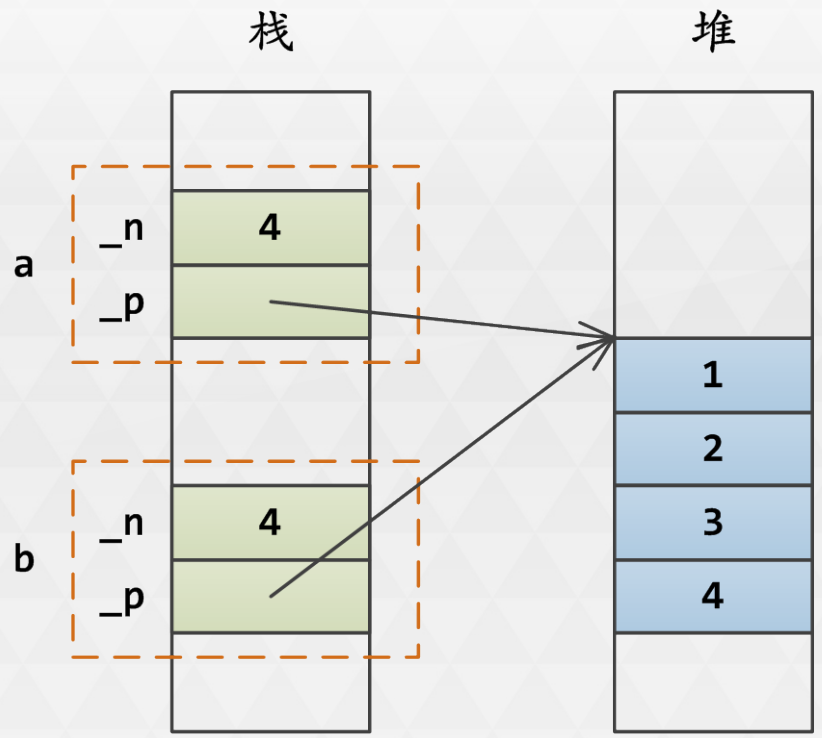
```
int & A::operator[]( int i )
{
  if( i < 0 || i >= _n )
    throw std::out_of_range( "Out of range when trying to access the object...");
  return _p[i];
}

const int & A::operator[]( int i ) const
{
  if( i < 0 || i >= _n )
    throw std::out_of_range( "Out of range when trying to access the object...");
  return _p[i];
}
```

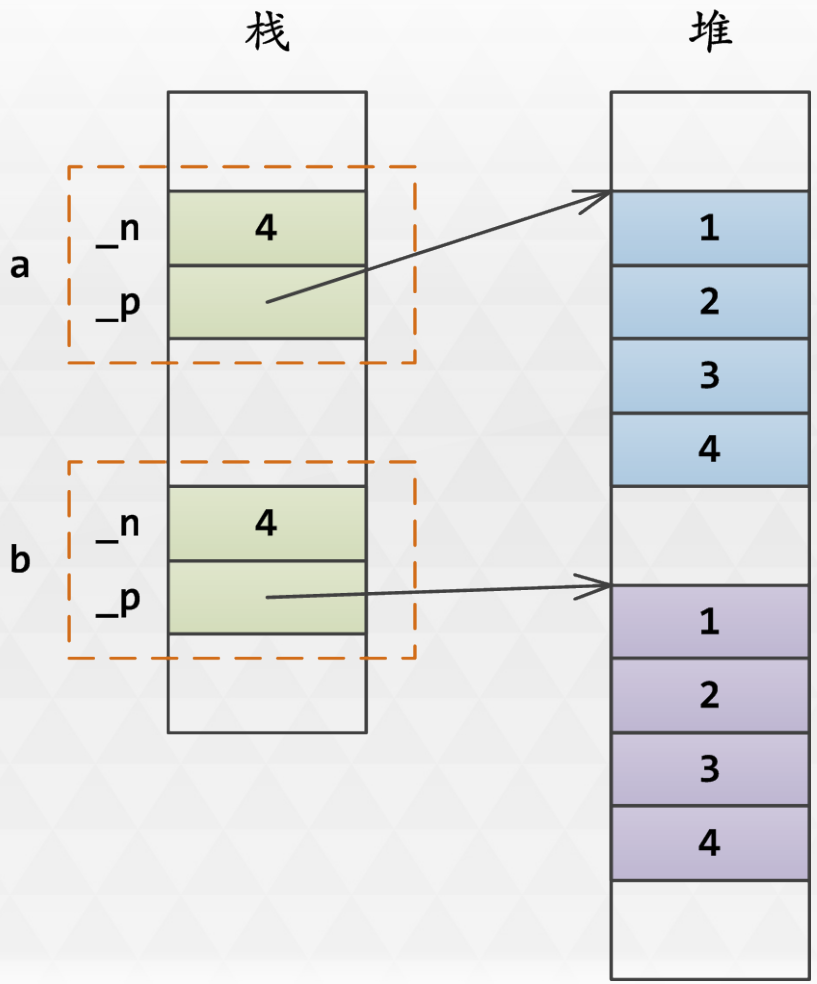
浅拷贝

```
int main()
{
  A a(4), b;
  for( int i = 0; i < 4; i++ )
    a[i] = i + 1;
  std::cout << "Before object assignment:" << std::endl;
  for( int i = 0; i < 4; i++ )
    std::cout << a[i] << " ";
  std::cout << std::endl;
  b = a;
  std::cout << "After object assignment:" << std::endl;
  for( int i = 0; i < 4; i++ )
    std::cout << b[i] << " ";
  std::cout << std::endl;
  return 0;    // 程序结束时，系统崩溃
}
```

浅拷贝



深拷贝



深拷贝

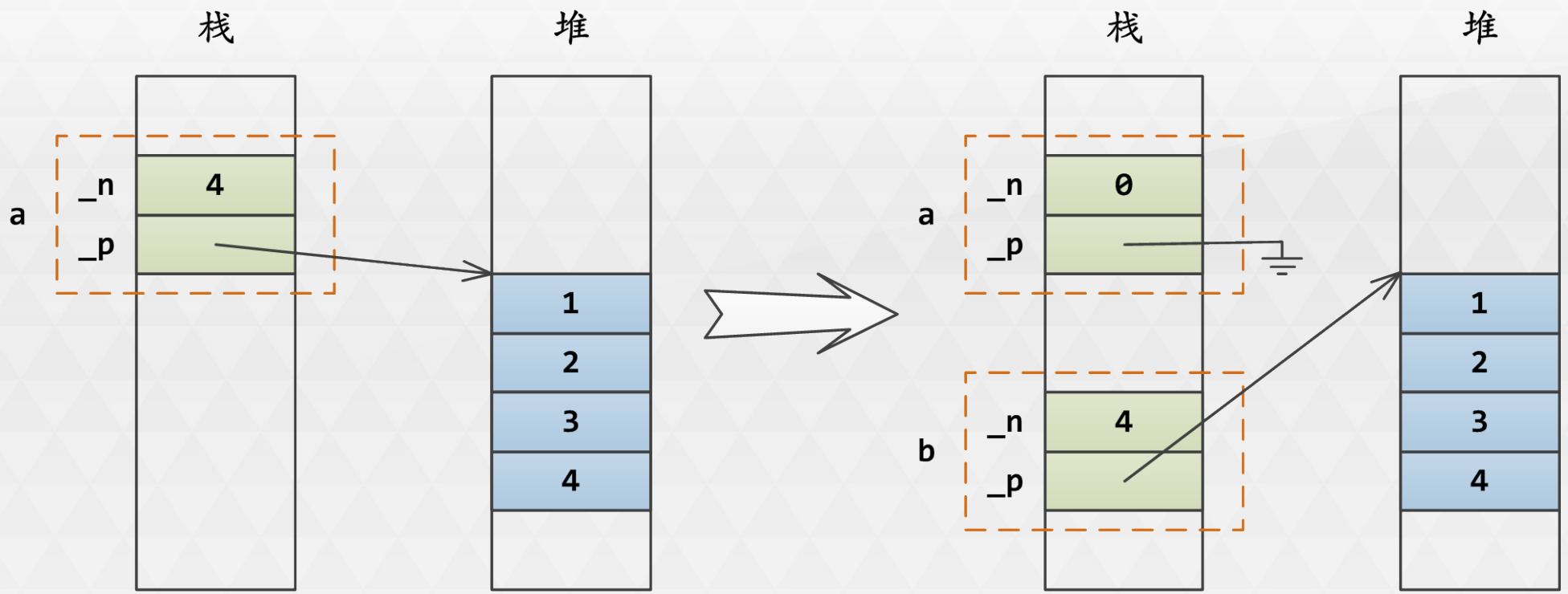
```
class A
{
public:
  A() : _n(0), _p(NULL) { }
  explicit A( int n ) : _n(n), _p(new int[n]) { }
  A( int n, int * p ) : _n(n), _p(p) { }
  A( const A & that );
  A & operator=( const A & that );
  virtual ~A() { if(_p){ delete[] _p, _p = NULL; } }
public:
  int & operator[]( int i );
  const int & operator[]( int i ) const;
private:
  int _n;
  int * _p;
};
```

深拷贝

```
A::A( const A & that )
{
  this->_n = that._n;
  _p = new int[_n];
  for( int i = 0; i < _n; i++ )
    _p[i] = that._p[i];
}

A & A::operator=( const A & that )
{
  this->_n = that._n;
  if( _p )
    delete[] _p;
  _p = new int[_n];
  for( int i = 0; i < _n; i++ )
    _p[i] = that._p[i];
  return *this;
}
```

移动语义



移动语义

- 左值与右值
- 左值引用与右值引用
- 移动赋值与移动构造
- 移动语义重载

左值与右值

C原始定义

- 左值：可以出现在赋值号左边或右边
- 右值：只能出现在赋值号右边

C++定义

- 左值：用于标识非临时对象或非成员函数的表达式
- 右值：用于标识临时对象的表达式或与任何对象无关的值（纯右值），或者用于标识即将失效的对象的表达式（失效值）

左值引用与右值引用

左值引用：&

右值引用：&&

- 深拷贝需要频繁分配和释放内存，效率较低
- 移动语义的目的：所有权移交，不需要重新构造和析构
- 为与构造函数兼容，移动语义必须为引用，而不能是指针或普通量
- 普通引用传递左值，以允许函数内部修改目标数据对象
- 为区分左值引用，实现移动语义时必须传递右值引用
- 为保证能够修改目标数据对象，在函数内部必须将右值引用作为左值引用对待

移动赋值与移动构造

```
class A
{
public:
  A() : _n(0), _p(nullptr)  {  }
  explicit A( int n ) : _n(n), _p(new int[n])  {  }
  A( int n, int * p ) : _n(n), _p(p)  {  }
  A( A && that );
  A & operator=( A && that );
  virtual ~A() {  if(_p){  delete[] _p, _p = nullptr;  }  }
public:
  int & operator[]( int i );
  const int & operator[]( int i ) const;
private:
  int _n;
  int * _p;
};
```

移动赋值与移动构造

```
A::A( A && that )
{
  //  nullptr：C++11预定义的空指针类型nullptr_t的常对象
  //  可隐式转换为任意指针类型和bool类型，但不能转换为整数类型，以取代NULL
  _n = that._n,  _p = that._p,  that._n = 0,  that._p = nullptr;
  //  *this = that;    //  此代码不会调用下面重载的赋值操作符函数
  //  具名右值引用that在函数内部被当作左值，不是右值
  //  匿名右值引用才会被当作右值；理论上如此……
  //  *this = static_cast<A &&>( that );    //  等价于 *this = std::move( that );
  //  上一行代码可以调用下面重载的移动赋值操作符，但是有可能导致程序崩溃
  //  因this指向的本对象可能刚刚分配内存，_p字段所指向的目标数据对象无定义
}
A & A::operator=( A && that )
{
   if( _p )    delete[] _p;    //  删除此行代码可能会导致内存泄露
  //  可以测试是否为同一对象，以避免自身复制操作，但意义不大
  _n = that._n,   _p = that._p,  that._n = 0,  that._p = nullptr;
  return *this;
}
```

移动语义重载

```
class A
{
public:
  A() : _n(0), _p(nullptr) { }
  explicit A( int n ) : _n(n), _p(new int[n]) { }
  A( int n, int * p ) : _n(n), _p(p) { }
  // 可以同时提供拷贝语义与移动语义版本，前者使用常左值引用
  // 不能修改目标数据对象的值，后者则可以修改
  A( const A & that );
  A( A && that );
  A & operator=( const A & that );
  A & operator=( A && that );
  virtual ~A() { if( _p ) { delete[] _p, _p = nullptr; } }
  ……
};
```

移动语义重载

```
int main()
{
  A a( 4 );
  for( int i = 0; i < 4; i++ )
    a[i] = i + 1;

  A b( a );          //  调用拷贝构造版本
  b = a;             //  调用普通赋值版本

  //  把左值引用转换为右值引用，否则会调用左值版本
  A c( static_cast< A && >( a ) );          //  调用移动构造版本
  c = static_cast< A && >( a );             //  调用移动赋值版本

  return 0;
}
```

移动语义再认识

左值引用同样可以实现移动语义

```
class A
{
public:
  A() : _n(0), _p(nullptr)  {  }
  explicit A( int n ) : _n(n), _p(new int[n])  {  }
  A( int n, int * p ) : _n(n), _p(p)  {  }
  A( A & that );                              //  重载非常量版本；移动构造语义
  A( const A & that );                        //  重载常量版本；深拷贝构造语义
  A & operator=( A & that );                  //  重载非常量版本；移动赋值语义
  A & operator=( const A & that );            //  重载常量版本；深拷贝赋值语义
  virtual ~A() {  if(_p){  delete[] _p, _p = nullptr;  }  }
public:
  int & operator[]( int i )  throw( std::out_of_range );
  const int & operator[]( int i ) const  throw( std::out_of_range );
private:
  int _n;
  int * _p;
};
```

移动语义再认识

```
A::A( A & that )
{
  _n = that._n,  _p = that._p,  that._n = 0,  that._p = nullptr;
}
A::A( const A & that )
{
  this->_n = that._n;
  _p = new int[_n];    for( int i = 0; i < _n; i++ )    _p[i] = that._p[i];
}
A & A::operator=( A & that )
{
   if( _p )    delete[] _p;
  _n = that._n,   _p = that._p,  that._n = 0,  that._p = nullptr;
  return *this;
}
A & A::operator=( const A & that )
{
  this->_n = that._n;
  if( _p )    delete[] _p;
  _p = new int[_n];    for( int i = 0; i < _n; i++ )    _p[i] = that._p[i];
  return *this;
}
```

移动语义再认识

```
// “Main.cpp”
int main()
{
  A a1;                              // 缺省构造
  const A a2;                        // 缺省构造

  A a3( a1 );                        // 调用A::A( A & )，移动构造
  A a4( a2 );                        // 调用A::A( const A & )，深拷贝构造
  // 对于非常量，必须转型为常量才能进行深拷贝
  A a5( const_cast< const A & >( a1 ) ); // 调用A::A( const A & )

  A a6, a7, a8;                      // 缺省构造
  a6 = a1;                           // 调用A::operator=( A & )，移动赋值
  a7 = a2;                           // 调用A::operator=( const A & )，深拷贝赋值
  a8 = const_cast< const A & >( a1 ); // 调用A::operator=( const A & )

  return 0;
}
```

右值引用的意义

右值引用可以使用文字作为函数实际参数

```
// 不接受文字作为实际参数，因无法获取文字的左值
int f( int & x ) { return ++x; }

// 接受文字作为实际参数，传递右值引用
// 具名右值引用作为左值，匿名右值引用作为右值
// 在函数内部理论如此，但实际上……
int f( int && x ) { return ++x; }

int main()
{
  // 错误代码，++操作符的操作数必须为左值
  // std::cout << ++10 << std::endl;
  // 可能有问题，传递右值引用，但部分编译器可能将其作为左值
  std::cout << f(10) << std::endl;    // 11?
  return 0;
}
```

右值引用的意义

避免编写过多的构造与赋值函数

- 不管是左值引用还是右值引用，若同时提供拷贝语义与移动语义，需要2对（4个）构造和赋值函数
- 若通过单独提供成员值的方式构造对象，单成员至少需要2对（4个）构造和赋值函数，双成员至少需要4对（8个）构造和赋值函数
- 使用右值引用，通过函数模板可以缩减代码编写量

实现完美转发

流操作符重载

- 流操作符重载的一般形式
- 流与文件
- 文件输入输出

流操作符重载的一般形式

为数偶类定义专用的流操作符

```
class Couple
{
public:
  Couple( int a = 0, int b = 0 ) : _a(a), _b(b)  {  }
  //  必须使用此格式，以与流的连续书写特性保持一致
  friend ostream & operator<<( ostream & os, const Couple & c );
  friend istream & operator>>( istream & is, Couple & c );
private:
  int _a, _b;
};
```

流操作符重载的一般形式

```
// 注意：此处实现的流输入输出格式不统一
ostream & operator<<( ostream & os, const Couple & c )
{
  os << "( " << c._a << ", " << c._b << " )" << endl;
  return os;
}
istream & operator>>( istream & is, Couple & c )
{
  is >> c._a >> c._b;
  return is;
}

int main()
{
  Couple a( 1, 2 ), b;
  cin >> b;
  cout << a << endl;
  return 0;
}
```

流与文件

- 标准流类库
- 流格式
- 操纵符
- 文件流
- 流状态
- 流定位

标准流类库

流：数据从源到目的的流动

输入输出流类

- 输入输出流：iostream；输入流：istream；输出流：ostream

字符串流类

- 输入输出字符串流：strstream；输入字符串流：istrstream；输出字符串流：ostrstream

文件流类

- 输出文件流：ofstream；输入文件流：ifstream；输入输出文件流：fstream

标准流类库

全局流对象

- std::cout：标准输出流对象，一般对应标准输出设备
- std::cin：标准输入流对象，一般对应标准输入设备
- std::cerr：标准错误流对象，一般对应标准错误输出设备
- std::clog：标准日志流对象，一般对应标准日志输出设备
- std::cout、std::cerr与std::clog为std::ostream类的对象；std::cin为std::istream类的对象

插入与提取

插入

- 目的：将数据对象插入到流中
- 插入操作符也称输出操作符
- `std::cout << "Hello World!";  // 将字符串插入到输出流`

提取

- 从流中提取数据对象
- 提取操作符也称输入操作符
- `int a; std::cin >> a;  // 从输入流中提取整数`

注意：因为流可能被重定向或束定，有时使用输入输出描述流操作可能会让人迷惑

常用输入输出流函数

判断流是否已结束

- cin.eof()

读取单个字符

- cin.get(istream::char_type & c)

读取字符串

- cin.get(istream::char_type * s, streamsize n, istream::char_type delimiter = '\n')

读取单行文本

- cin.getline(istream::char_type * s, streamsize n, istream::char_type delimiter = '\n')

输出单个字符

- cout.put(ostream::char_type c)

流格式标志

```
class ios_base
{
public:
  typedef int fmtflags;
  enum
  {
    left = 0x0001,  right = 0x0002,  internal = 0x0004,
    dec = 0x0008,  hex = 0x0010,  oct = 0x0020,
    fixed = 0x0040,  scientific = 0x0080,  boolalpha = 0x0100,
    showbase = 0x0200,  showpoint = 0x0400,  showpos = 0x0800,
    skipws = 0x1000,  unitbuf = 0x2000,  uppercase = 0x4000,
    adjustfield = left | right | internal,
    basefield = dec | oct | hex,
    floatfield = scientific | fixed
  };
};
```

流格式标志

位掩码

- 使用32位整数的位代表流格式标志：ios_base::fmtflags
- 每个标志位可单独设置与清除

设置预定义标志位

- cout.setf(ios_base::showbase);　// 输出整数前缀，十六进制前添加“0x”

清除预定义标志位

- cout.unsetf(ios_base::showbase);　// 清除上述标志

位组：特定标志位集合，位组中的标志位互相排斥

- ios_base::adjustfield、ios_base::basefield与ios_base::floatfield
- cout.setf(ios_base::hex, ios_base::basefield);
- // 设置十六进制输出格式，使用单参数版本无效果

流格式标志

设置用户自定义参数：单参数版本

- 设置时传递用户指定值
- std::cout.width(8);　　// 将最小输出宽度定为8个字符
- std::cout.precision(8);　　// 将输出精度定为8位
- std::cout.fill('?');　　// 使用 ‘?’ 填充空白字符位置

获取用户自定义参数：无参数版本

- // 获得当前的输出精度值
- std::streamsize precision = std::cout.precision();

操纵符

操纵符的目的：控制流的输入输出格式

- 无参数操纵符：函数指针
- 单参数操纵符：函子，即带有函数指针功能的操纵符类的对象，实现上为重载了函数调用操作符的操纵符类

操纵符示例

- 头文件：“iomanip”
- std::cout << "Hello World!" << std::endl;
- int n = 1024; std::cout << std::dec << n << '\n' << std::hex << n << std::endl;

预定义操纵符

操纵符	I/O	功能描述
std::boolalpha	I/O	字符格式的bool值
std::dec	I/O	十进制
std::endl	O	插入换行符并刷新流缓冲区
std::ends	O	插入字符串结束符
std::fixed	O	使用定点数格式表示浮点数
std::flush	O	刷新流缓冲区
std::hex	I/O	十六进制
std::internal	O	在内部填充字符
std::left	O	左对齐，右边多余部分使用填充字符填充
std::noboolalpha	I/O	复位std::boolalpha设置
std::noshowbase	O	复位std::showbase设置

预定义操纵符

操纵符	I/O	功能描述
std::noshowpoint	O	复位std::showpoint设置
std::noshowpos	O	复位std::showpos设置
std::noskipws	I	复位std::skipws设置
std::nounitbuf	O	复位std::unitbuf设置
std::nouppercase	O	复位std::uppercase设置
std::oct	I/O	八进制
std::right	O	右对齐，左边多余部分使用填充字符填充
std::resetiosflag(std::ios_base::fmtflags mask)	I/O	复位格式标志
std::scientific	O	使用科学计数法表示浮点数
std::setbase(int base)	I/O	设置整数的基（进制），可能为8、10或16
std::setfill(std::basic_ios::char_type c)	I/O	设置填充字符

预定义操纵符

操纵符	I/O	功能描述
std::setiosflag(std::ios_base::fmtflags mask)	I/O	设置格式标志
std::setprecision(int n)	O	设置数值精度
std::setw(int n)	I/O	设置最小字段宽度
std::showbase	O	输出整数前缀，十六进制前添加“0x”
std::showpoint	O	浮点数输出时强制显示小数点
std::showpos	O	设置非负整数显示正号标志
std::skipws	I	设置忽略空格标志
std::unitbuf	O	每次格式化操作后都刷新流缓冲区
std::uppercase	O	数值格式化输出时使用大写字母
std::ws	I	忽略空格

文件流

文件特性

- 文件一般保存在外部存储设备上
- 文件生命周期可能远远超过创建它的程序本身

文件操作：读、写

- 一般使用文件指针，该指针代表文件的当前访问位置
- 老式的C语言使用文件句柄（handle）或文件描述符（file descriptor）表示某个打开的文件数据对象

文件流的使用

- 头文件：“fstream”
- 按照特定格式重载类的流操作符
- 创建文件流对象，输入输出

文件流

文件打开模式

- std::ios_base::app：每次插入都定位到文件流的尾部
- std::ios_base::binary：使用二进制而不是文本格式打开文件流
- std::ios_base::in：流用于输入目的，允许提取，此为std::ifstream流缺省设置
- std::ios_base::out：流用于输出目的，允许插入，此为std::ofstream流缺省设置
- std::ios_base::trunc：若文件存在，清除文件内容，此为std::ofstream流缺省设置
- std::ios_base::ate：若文件存在，定位到文件尾部文件一般保存在外部存储设备上

流状态

流状态：表示操作成功或失败的状态信息

- std::ios_base::goodbit：流完好无损
- std::ios_base::badbit：流已出现致命错误，一般无法恢复
- std::ios_base::eofbit：流结束时设置
- std::ios_base::failbit：流操作失败时设置，可能恢复

流状态对流操作行为的影响

- 一旦流状态存在错误，所有I/O操作都失效
- 在出现std::ios_base::failbit与std::ios_base::badbit状态时，输出操作立即停止
- 在非std::ios_base::goodbit状态时，输入操作立即停止

流状态

流状态测试

- bool std::ios_base::good() const：没有出现任何错误时返回真
- bool std::ios_base::eof() const：设置std::ios_base::eofbit状态时返回真
- bool std::ios_base::fail() const：设置std::ios_base::failbit状态时返回真
- bool std::ios_base::bad() const：设置std::ios_base::badbit状态时返回真
- bool std::ios_base::operator !() const：与std::ios_base::fail() 效果相同
- std::ios_base::operator void*() const：std::ios_base::fail() 为真时返回空指针，否则非空

流定位

流位置指针

- 位置指针指向下一次读写操作时的数据对象在流中的位置，该指针会随着输入输出操作而不断变化
- 单向流：一个位置指针；双向流：两个位置指针

流位置指针的获取

- 成员函数tellp()：获取当前的流位置指针（写指针）
- 成员函数tellg()：获取当前的流位置指针（读指针）

流位置指针的定位

- 成员函数seekp()：将文件位置指针定位到某个特定位置，用于插入（输出）目的
- 成员函数seekg()：将文件位置指针定位到某个特定位置，用于提取（输入）目的的定位

流定位

流定位函数seekp()与seekg()

- 单参数版本：可以使用获取的位置指针
- 双参数版本：第一个参数为偏移量；第二个参数为定位基准

定位基准

- std::ios_base::beg：从流的开始位置开始计算偏移量
- std::ios_base::cur：从当前位置开始计算偏移量
- std::ios_base::end：从流的结束位置开始计算偏移量

文件输入输出

将点对象输出到文件中

```
#include <fstream>
#include "point.h"
using namespace std;

int main()
{
  ofstream ofs( "~/CPP/filestream/data.txt" );
  //  ofs.open( "~/CPP/filestream/data.txt" );
  Point2D pt2d( 1, 2 );
  Point3D pt3d( 3, 4, 5 );
  ofs << pt2d;
  ofs << pt3d;
  ofs.close();
  return 0;
};
```

文件输入输出

将Point2D、Point3D对象从文件中读取出来

```
#include <fstream>
#include "point.h"
using namespace std;

int main()
{
  ifstream ifs( "~/CPP/filestream/data.txt" );
  Point2D pt2d;
  Point3D pt3d;
  ifs >> pt2d  >> pt3d;
  ifs.close();
  return 0;
};
```

文件输入输出

```
class Point2D : public Point
{
public:
  friend ostream & operator<<( ostream & os, const Point2D & pt );
  friend istream & operator>>( istream & is, Point2D & pt );
};

class Point3D : public Point2D
{
public:
  friend ostream & operator<<( ostream & os, const Point3D & pt );
  friend istream & operator>>( istream & is, Point3D & pt );
};
```

文件输入输出

```
ostream & operator<<( ostream & os, const Point2D & pt )
{
  os << '(' << pt._x << ',' << pt._y << ')';
  return os;
}
// 逐字符分析，确保文件非致命改动不影响数据读取
istream & operator>>( istream & is, Point2D & pt )
{
  char _c;    int _a[2] = {0, 0}, _i = 0;    bool _started = false;
  _c = is.get();
  while( _c != '\n' ){
    if( _c == '(' ){    _started = true;  }
    else if( isdigit(_c) ){    if( _started )    _a[_i] = _a[_i] * 10 + _c - 48;   }
    else if( _c == ',' ){   _i++;    }
    else if( _c == ')' ){    _started = false;   break;    }
    _c = is.get();
  }
  pt._x = _a[0],    pt._y = _a[1];    return is;
}
```

文件输入输出

```
ostream & operator<<( ostream & os, const Point3D & pt )
{
  os << '(' << pt._x << ',' << pt._y << ',' << pt._z << ')';
  return os;
}
// 逐字符分析，确保文件非致命改动不影响数据读取
istream & operator>>( istream & is, Point3D & pt )
{
  char _c;    int _a[3] = {0, 0, 0}, _i = 0;    bool _started = false;
  _c = is.get();
  while( _c != '\n' ){
    if( _c == '(' ){    _started = true;    }
    else if( isdigit(_c) ){    if( _started )    _a[_i] = _a[_i] * 10 + _c - 48;    }
    else if( _c == ',' ){    _i++;    }
    else if( _c == ')' ){    _started = false;    break;    }
    _c = is.get();
  }
  pt._x = _a[0],    pt._y = _a[1],    pt._z = _a[2];    return is;
}
```

数据持久化

持久化：将数据保存在外部文件中，在程序运行时装入内存，在程序结束时重新写回文件

思考题

- **考虑下述编程任务。存在一个数据结构，需要将其数据流入流出。为提升程序效率，只在必要时进行数据持久化，即仅当内存中的数据发生变化时才写入文件。如何实现？提示：（1）需要考虑内存数据的来源和目的地对数据持久化的影响。（2）使用下一章将要讨论的动态型式转换技术，实现效果更佳。**

操作符重载总结

哪些操作符可以重载？

- 不可重载操作符：“::”、“?:”、“.”、“.*”、“sizeof”、“#”、“##”、“typeid”
- 其他操作符都可重载

操作符重载原则

- 只能使用已有的操作符，不能创建新的操作符
- 操作符也是函数，重载遵循函数重载原则
- 重载的操作符不能改变优先级与结合性，也不能改变操作数个数及语法结构
- 重载的操作符不能改变其用于内部类型对象的含义，它只能和用户自定义类型的对象一起使用，或者用于用户自定义类型的对象和内部类型的对象混合使用
- 重载的操作符在功能上应与原有功能一致，即保持一致的语义

操作符重载总结

操作符重载的类型：成员函数或友元函数

- 重载为类的成员函数：少一个参数（隐含this，表示二元表达式的左参数或一元表达式的参数）
- 重载为类的友元函数：没有隐含this参数

成员函数与友元函数

- 一般全局常用操作符（关系操作符、逻辑操作符、流操作符）重载为友元函数，涉及对象特殊运算的操作符重载为成员函数
- 一般单目操作符重载为成员函数，双目操作符重载为友元函数
- 部分双目操作符不能重载为友元函数：“=”、“()”、“[]”、“->”
- 类型转换操作符只能重载为成员函数

操作符重载总结

重载的操作符参数：一般采用引用形式，主要与数学运算协调

- **示例：Couple a(1,2), b(3,4), c;　c = a + b;**
- **若有定义：Couple Couple::operator+(Couple*, Couple*) { …… }**
- **则需调用：Couple a(1,2), b(3,4), c;　c = &a + &b;**

操作符重载总结

操作符重载的函数原型列表（推荐）

- 普通四则运算
  - friend A operator + (const A & lhs, const A & rhs);
  - friend A operator - (const A & lhs, const A & rhs);
  - friend A operator * (const A & lhs, const A & rhs);
  - friend A operator / (const A & lhs, const A & rhs);
  - friend A operator % (const A & lhs, const A & rhs);
  - friend A operator * (const A & lhs, const int & rhs); // 标量运算，如果存在
  - friend A operator * (const int & lhs, const A & rhs); // 标量运算，如果存在
- 关系操作符
  - friend bool operator == (const A & lhs, const A & rhs);
  - friend bool operator != (const A & lhs, const A & rhs);
  - friend bool operator < (const A & lhs, const A & rhs);
  - friend bool operator <= (const A & lhs, const A & rhs);
  - friend bool operator > (const A & lhs, const A & rhs);
  - friend bool operator >= (const A & lhs, const A & rhs);

操作符重载总结

操作符重载的函数原型列表（推荐）

- 逻辑操作符
  - friend bool operator || (const A & lhs, const A & rhs);
  - friend bool operator && (const A & lhs, const A & rhs);
  - bool A::operator ! ( );
- 正负操作符
  - A A::operator + ();　　// 取正
  - A A::operator - ( );　　// 取负
- 递增递减操作符
  - A & A::operator ++ ();　　// 前缀递增
  - A A::operator ++ (int);　　// 后缀递增
  - A & A::operator --();　　// 前缀递减
  - A A::operator -- (int);　　// 后缀递减

操作符重载总结

操作符重载的函数原型列表（推荐）

- 位操作符
  - friend A operator | (const A & lhs, const A & rhs); // 位与
  - friend A operator & (const A & lhs, const A & rhs); // 位或
  - friend A operator ^ (const A & lhs, const A & rhs); // 位异或
  - A A::operator << (int n); // 左移
  - A A::operator >> (int n); // 右移
  - A A::operator ~ (); // 位反
- 动态存储管理操作符：全局或成员函数均可
  - void * operator new(std::size_t size) throw(bad_alloc);
  - void * operator new(std::size_t size, const std::nothrow_t &) throw( );
  - void * operator new(std::size_t size, void * base) throw( );
  - void * operator new[](std::size_t size) throw(bad_alloc);
  - void operator delete(void * p);
  - void operator delete [](void * p);

操作符重载总结

操作符重载的函数原型列表（推荐）

- 赋值操作符
  - A & operator = (A & rhs);
  - A & operator = (const A & rhs);
  - A & operator = (A && rhs);
  - A & operator += (const A & rhs);
  - A & operator -= (const A & rhs);
  - A & operator *= (const A & rhs);
  - A & operator /= (const A & rhs);
  - A & operator %= (const A & rhs);
  - A & operator &= (const A & rhs);
  - A & operator |= (const A & rhs);
  - A & operator ^= (const A & rhs);
  - A & operator <<= (int n);
  - A & operator >>= (int n);

操作符重载总结

操作符重载的函数原型列表（推荐）

- 下标操作符
  - T & A::operator [] (int i);
  - const T & A::operator [] (int i) const;
- 函数调用操作符
  - T A::operator () (...); // 参数可选
- 类型转换操作符
  - A::operator char * () const;
  - A::operator int () const;
  - A::operator double () const;
- 逗号操作符
  - T2 operator , (T1 t1, T2 t2); // 不建议重载

操作符重载总结

操作符重载的函数原型列表（推荐）

- 指针与选员操作符
  - A * A::operator & ( ); // 取址操作符
  - A & A::operator * ( ); // 引领操作符
  - const A & A::operator * ( ) const; // 引领操作符
  - C * A::operator -> ( ); // 选员操作符
  - const C * A::operator -> ( ) const; // 选员操作符
  - C & A::operator ->* (...); // 选员操作符，指向类成员的指针
- 流操作符
  - friend ostream & operator << (ostream & os, const A & a);
  - friend istream & operator >> (istream & is, A & a);

编程实践

10.1 使用面向对象技术完成习题7.1，并为有理数类重载必要的操作符。

10.2 new与delete操作符可以被重载。研究存储管理技术，重载这两个操作符以实现程序独有的动态存储管理策略。