Module03-09
C++ 面向对象编程：操作符重载

面向对象编程 - 操作符重载

- 操作符重载：
  - 操作符重载概览
  - 友元
  - 赋值操作 operator=
  - 下标操作 operator[]
  - 函数调用 operator()
  - 去引用 * 和 operator->
  - 递增与递减 operator++ 和 --
  - 隐式转换与 explicit 构造
  - 类型转换

操作符重载 - 概览

- C++ 操作符重载机制：
  - 让自定义类型的行为更象基本类型
  - 模拟易于接受的操作符操作方式（而不是使用普通的函数）

操作符重载 - 概览

- 可以被重载的操作符
  - `+  -  *  /  %  ^  &  |  ~`
  - `!  ,  =  <  >  <=  >=  ++  --`
  - `<<  >>  ==  !=  &&  ||  +=  -=  /=`
  - `%=  ^=  &=  |=  *=  <<=?  >>=  []  ()`
  - `->  ->*  new  new[]  delete  delete[]`
- 不可被重载的操作符
  - `::` （域操作符）
  - `.` （类成员选择）
  - `.*` （类成员选择，选择成员函数指针）
  - `?:` （三元关系表达式）

操作符重载 - 概览

- 百无一用的 Integer 类

```
// Integer.cpp

class Integer {
public:
    enum Base { DEC, HEX, OCT, BIN };

    // Constructors and destructors
    Integer(const Base& b = DEC);
    Integer(const int& val, const Base& b = DEC);
    Integer(const Integer& other);
    Integer& operator=(const Integer& other);
    Integer& operator=(const int& other);
    ~Integer();

    // Getters and Setters
    void setValue(const int& val);
    int getValue() const;
    void setFormat(const Base& b);
    Base getFormat() const;
```

操作符重载 - 概览

- 百无一用的 Integer 类（续）

```
// Integer.cpp

    // Arithmetic operations
    Integer add(const Integer& right);    // +
    Integer& added(const Integer& right); // +=
    // Other arithmetic operations or bitwise operations
    // such as: multiply, bitwise and ...

    // // Relational operators
    bool isLessThan(const Integer& right) const; // <
    bool equals(const Integer& right) const;     // ==

private:
    int value;
    Base base;
};
```

操作符重载 - 概览

- 使用 Integer 类

```
int main() {
    Integer m(8);
    Integer n(6);

    m.added(n);  // m.value == 14
    Integer x = m.add(n);  // x.value == 20

    cout << m.equals(n) << endl;
    cout << x.isLessThan(m) << endl;
    cout << m.getValue() << endl;

    // Other operations ...
}
```

操作符重载 - 概览

- 再看 Integer 类
  - 其中定义的 add() 、 added() 、 equals() 等可以工作得很好
  - 但是，既然是算术运算和关系运算，使用 + 、 += 、 == 等操作符是否会更自然？
  - 因为 C++ 语言支持操作符重载

操作符重载 - 概览

- 改革过的 Integer 类（部分）

```
// Relational operators
bool operator<(const Integer& right) const;
bool operator==(const Integer& right) const;

// Arithmetic operators
Integer operator+(const Integer& right);
Integer operator+(const int& right);
Integer& operator+=(const Integer& right);
Integer& operator+=(const int& right);
```

```
// 使用Integer
Integer m(8);
Integer n(6);

m += 6;  // m.value == 14
Integer x = m + n;  // x.value == 20

cout << (m == n) << endl;
cout << (x < m) << endl;
```

操作符重载 - 友元

- 友元 (Friend)
  - 一个类常规的非静态成员函数声明描述了三件在逻辑上不相同的事情：
    1. 能访问类声明的 private 部分
    2. 该函数置于类的作用域中
    3. 该函数必须由一个该类的对象去激活（有一个隐含的 this 指针）
  - 一个 static 成员函数，我们可以让它只具有前 2 种性质
  - 将一个函数声明为该类的友元，则可以使它只具有第一种性质

操作符重载 - 友元

- 友元 (Friend)（续）
  - 一个类的友元可以象该类的成员一样，访问该类的任何成员
  - 一个类的友元可以是：类、类的成员函数、一般函数
  - 一个类的友元不是该类的成员
  - 友元不具有对称性，即声明 B 是 A 的友元，不代表 A 是 B 的友元
  - 一个类的友元可以声明在类的 public 区或 private 区，没有任何差别
- 什么时候需要友元
  - 当一个类 A 或函数 F 需要方便访问另一个类 T，但并不需要成为类 T 的成员时，将 A 或 F 当作 T 的友元是最好的方式

操作符重载 - 友元

- Integer 类对象的局限
  - 下列代码的问题：

```
Integer n(6);
Integer k = 8 + n;  // ?????
```

  - 有 integer::operator+(int)，但是没有 int::operator+(Integer)
  - 需要定义友元函数来解决上述问题

操作符重载 - 赋值操作 operator=

- 赋值操作符 (Assignment Operator)
  - 为类的成员函数，用于复制赋值
  - 函数的声明 (T 为类类型)：

```
T& operator=(const T& other);
```

操作符重载 - 下标操作 operator[]

- 下标操作符 (Subscripting Operator)
  - 必须是类的成员函数，为顺序容器 (如数组、 vector 等) 的下标操作，或为关联容器 (map 等) 的下标操作
  - 函数的声明 (以下 Item 、 Value 、 Key 均为类型) :

    顺序容器:

```
const Item& operator[](const size_t& index)
const;
Item& operator[](const size_t& index);
```

    关联容器:

```
const Value& operator[](const Key& key) const;
Value& operator[](const Key& key);
```

操作符重载 - 函数对象 operator()

- 函数对象 (Functional Object or Functor)
  - 如果一个对象的行为象函数，那么该对象就是函数对象
  - 函数对象就是重载了函数调用操作符 () 的类的对象
  - 函数对象可以象函数一样被使用
  - 函数对象广泛应用于标准库的算法库
  - 必须为成员函数
  - operator() 的声明一般为:

```
return_type operator()(args_list) cv-qualifier
```

如:

```
void operator()(int& src) {
        src += 8;
}
```

操作符重载 - 去引用 * 和成员选择 ->

- 去引用 (Dereference) 和成员选择操作符
  - 必须为成员函数
  - 常用于智能指针
  - 函数的声明（如智能指针 TPtr、目标类型 T ）：

```
T& operator*();
T* operator->();
```

操作符重载 - 去引用 * 和成员选择 ->

- 去引用 (Dereference) 和成员选择操作符（续）
  - 示例：

```
class Aptr {
    A* a;
public:
    Aptr(A* ap) {
        a = ap;
    }

    A& operator*() {
        return *a;
    }

    A* operator->() {
        return a;
    }
};
```

```
int main() {
    A b(8);
    Aptr ap(&b);

    // (ap.operator->())->print();
    ap->print();
    (*ap).print();
}
```

操作符重载 - 递增 ++ 和递减 --

- 递增 (Increment) 和递减 (Decrement) 操作符
  - 必须为成员函数
  - 函数的一般声明:

```
T& operator++();    // ++t
T operator++(int); // t++
T& operator--();    // --t
T operator--(int); // --t
```

操作符重载 - 隐式转换和 explicit 构造

- 隐式转换
  - 按默认规定，只有一个参数的构造函数也定义了一个隐式转换

```
class A {
    int n;
    int k;
public:
    A(const int& m) :
        n(m), k() {
    }

    void print() const {
        cout << n << endl;
    }
};
```

```
void f(const A& a) {
    a.print();
}

int main() {
    f(8);

    A b = 12;
    b.print();
}
```

操作符重载 - 隐式转换和 explicit 构造

- explicit 构造
  - 如果要阻止隐式转换的行为，则在单参数的构造函数前加上关键字 explicit

```
class A {
    int n;
    int k;
public:
    explicit A(const int& m) :
        n(m), k() {
    }

    void print() const {
        cout << n << endl;
    }
};
```

```
void f(const A& a) {
    a.print();
}

int main() {
    f(8);  // Error

    A b = 12;   // Error
    b.print();
}
```

操作符重载 - 类型转换

- 类型转换 (Type Conversion)
  - 显式定义将一种类型转换成另一种类型
  - 必须为成员函数
  - 一个类中可以有多个类型转换函数
  - 函数的声明:

```
operator Type();
```

如向 int 类型的转换:

```
operator int();
```

操作符重载 - Bjarne's Advices

- Bjarne's Advices
  - 如果默认复制构造对于一个类合适，最好直接使用它
  - 如果默认复制构造对于一个类并不合适，重新定义它或禁用它
  - 对于对称的操作符采用非成员函数
  - 用 () 作为多维数组的下标
  - 要注意引进隐式转换的问题
  - 用成员函数表达那些需要左值作为其左操作数的操作符