2020/05/25_C++_第14课_运算符重载、智能指针

笔记本: C++
创建时间: 2020/5/25 星期一 15:21
作者: ileemi
标签: 运算符重载, 智能指针

运算符重载

可重载运算符/不可重载运算符

下面是可重载的运算符列表：

双目算术运算符	+ (加)，-(减)，*(乘)，/(除)，% (取模)
关系运算符	==(等于)，!= (不等于)，< (小于)，> (大于>，<=(小于等于)，>=(大于等于)
逻辑运算符	||(逻辑或)，&&(逻辑与)，!(逻辑非)
单目运算符	+ (正)，-(负)，*(指针)，&(取地址)
自增自减运算符	++(自增)，--(自减)
位运算符		(按位或)，& (按位与)，~(按位取反)，^(按位异或)，<< (左移)，>>(右移)
赋值运算符	=, +=, -=, *=, /= , % = , &=, |=, ^=, <<=, >>=
空间申请与释放	new, delete, new[ ] , delete[]
其他运算符	`()`(函数调用)，`->`(成员访问)，`,`(逗号)，`[]`(下标)

下面是不可重载的运算符列表：

- `.`：成员访问运算符
- `.*`，`->*`：成员指针访问运算符
- `::`：域运算符
- `sizeof`：长度运算符
- `?:`：条件运算符
- `#`：预处理符号

语法糖：可有可无，但是运算符重载是语法糖里比较重要的一个。

C++三大主要语法：类、继承、多态

常规的运算符只限适用于对基本类型数据的操作，解决类于类之间的数据操作，提高源码的可读性，提高类的易用性。
示例：

在没有运算符重载的情况下，对字符串进行拼接，就需要进行下面的操作：

```
CString str("Hello");
str.Append("World!");
```

当程序中含有运算符重载的时候，就可以使用下面的式子，对字符串进行拼接操作：

```
str += "World";
```

运算符重载的简单使用

关键字：operator

赋值（=）运算符

```
#include <iostream>
using namespace std;

// 操作整数的类
class CInteger
{
public:
  void SetVal(int nVal)
  {
    m_nVal = nVal;
  }

  int GetVal()const
  {
    return m_nVal;
  }

  void operator=(int nVal)
  {
    m_nVal = nVal;
```

```
    }
private:
    int m_nVal;
};
int main()
{
    CInteger n;
    n.SetVal(75);
    cout << n.GetVal() << endl;

    //n = 999;
    // 在没有定义对用的运算符重载时，= 仅仅支持基本的数据类型，不支持类类型

    // 为此，就需要定义对应的运算符重载

    n = 999;
    // 编译阶段，编译器调用的时候，还是通过成员函数对其进行调用的，将转换成下面的方式
    n.operator=(999);
    // 此时 operator 为成员函数，可以通过对象调用，和上一句的结果一样
    /*
      n = 999;
    00861A50  push        3E7h
    00861A55  lea         ecx,[n]
    00861A58  call        CIntageer::operator= (0861320h)
      n.operator=(666);
    00861A5D  push        29Ah
    00861A62  lea         ecx,[n]
    00861A65  call        CIntageer::operator= (0861320h)
    */

    cout << n.GetVal() << endl;

    return 0;
}
```

基本类型可以连续为变量进行赋值操作，类是否也可以这样做?

```
    int n0, n1, n2;
    n0 = n1 = n2 = 666;

    CInteger n3, n4, n5;
    n3 = n4 = n5 = 999;
    // n5.operator(999); --> 返回值 void 类型
    // n3 = n4 = void?
```

```
// 编译失败
// error C2679: 二进制“=”: 没有找到接受“void”类型的右操作数的运算符(或没有可接受的转换)
```

上面的程序想要编译通过，需要将对应的运算符重载方法的返回值进行更改

```
int operator=(int nVal)
{
  m_nVal = nVal;
  return m_nVal;
}
```

上面程序正常运行，还可以使用类的引用

```
CInteger& operator=(int nVal)
{
  m_nVal = nVal;
  return *this;
}
```

这里需要考虑下面的问题：

```
CInteger n3, n4, n5;
n3 = n4 = n5 = 999;

// 此时在类中没有定义对应的运算符重载方法时，可以这样写
// 编译器会为其自动生成一个对应的默认的运算符重载的方法
n3 = n4;  // 类似于拷贝构造，浅拷贝

// 考虑分析
/*
自右向左
n3 = n4 = n5 = 999;
第一步：n5 = 999 n5.operator=(999) --> 返回n5的引用，n3 = n4 = n5;
第二步：n4 = n5 调用默认的运算符重载
第三步：n3 = n4 调用默认的运算符重载
*/
n3 = n4 = n5 = 999;


// 为上述的程序，手动添加一个默认的运算符重载方法
CInteger& operator=(const CInteger& obj)
{
  m_nVal = obj.m_nVal;
  return *this;
```

```
}


    // 在这个程序中，为了便于运算符重载的方法可以和基本数据类型的数值进行相关操作
    // 所以这里就将，运算符 重载方法的返回值 以及 默认的运算符重载方法改写为 int类型
    CInteger nVal0, nVal1;
    int nVal2, nVal3;

    // 更改后，下面的句子就可以
    nVal0 = nVal2 = nVal1 = nVal3 = 999;


    // 类中对应的修改如下
    int operator=(int nVal)
    {
        m_nVal = nVal;
        return m_nVal;
    }

    int operator=(const CInteger& obj)
    {
        m_nVal = obj.m_nVal;
        return m_nVal;
    }
```

尝试使用引用做运算符重重载的返回值（查看多个类是否可以进行赋值操作）：

运算符重载的原则

- 不能改变原有运算符的用法
- 不能改变原有运算符的意义（方法内部 = 操作，定义为 + 操作）

在没有运算符重载的情况下，运算符重载支持 默认的复制拷贝运算符

运算符重载的返回值可以是 引用类型 也可以是 int 类型

运算符的分类

算数运算符

```
+, -, *, /, +=, -=, *=, /=, %=, ++, --
```

加号（+）运算符

```
// 算数运算符

class CInteger
{
public:
  CInteger(int nVal): m_nVal(nVal)
  {

  }
  void SetVal(int nVal)
  {
    m_nVal = nVal;
  }

  int GetVal()const
  {
    return m_nVal;
  }

  int operator+(int nVal)
  {
    //此句在这里，不能改变原有运算符的意义
    // m_nVal = nVal;
    return m_nVal + nVal;
  }
private:
    int m_nVal;
};
int main()
{
    CInteger n(999);
    int nRes = n + 1; // 注意：不能改变原有运算符的意义
    return 0;

}
```

运算符重载函数的方法也可以写成全局的：

```
 int mian()
 {
    CInteger n(999);
    int nRes = n + 1; // 注意：不能改变原有运算符的意义
```

```
    CInteger n1(6), n2(7), n3(8);
    // 在没有对程序添加局部的拷贝构造的，这样写，编译不通过
    n + 99 + n1 + 66 + n2 + 32 + n3;
    // error C2677 : 二进制“ + ” : 没有找到接受“CIntager”类型的全局运算符(或没有可接受的转换)

    /*
    分析：对应加法的运算过程，从左向右
    n + 99 + n1 + 66 + n2 + 32 + n3;
    第一步：n + 99, 调用自定义的函数运算符重载 --> 返回值为int 类型
    第二步：返回的int类型 和 n1（对象）
    999 + n1  //999.operator(n) --> int.operator(CIntager)
    */
}
```

```
//对于上面所述的问题，可以进行以下修改
// 算数运算符
class CInteger
{
public:
    CInteger(int nVal): m_nVal(nVal)
    {

    }
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }

    int GetVal()const
    {
        return m_nVal;
    }

    int operator+(int nVal)
    {
        //此句在这里，不能改变原有运算符的意义
        // m_nVal = nVal;
        return m_nVal + nVal;
    }
      // 运算符重载一般写成友元
  friend int operator+(int nVal, CInteger& interobj)
  {
    return nVal + interobj.GetVal();
```

```
  }
private:
    int m_nVal;
};

// 运算符重载的参数数量需要写出来（局部运算符重载）
int operator+(int nVal, CInteger& interobj)
{
  return nVal + interobj.GetVal();
}
```

需要注意一点，就是运算符重载一般写成友元，写在类内，需要在返回值类型前，添加关键字 frined，当类的数据成员比较多的时候，可以使用友元对其，进行直接访问，提高访问速度等

运算符重载一般写成友元，写在类内，友元最合适的运用场景（在类中定义运算符重载）

前++运算符

前++运算符

加法的操作结果是一个右值，不能被修改

```
class CInteger
{
public:
    CInteger(int nVal) : m_nVal(nVal)
    {

    }
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }

    int GetVal()const
    {
        return m_nVal;
    }

    int operator+(int nVal)
    {
        //此句在这里，不能改变原有运算符的意义
        // m_nVal = nVal;
        return m_nVal + nVal;
```

```
    }
    // 运算符重载一般写成友元
    friend int operator+(int nVal, CInteger& interobj)
    {
        return nVal + interobj.GetVal();
    }

    // 前++
    //int operator++()
    //{
    //  m_nVal++;
    //  return m_nVal;
    //}

    // 前++
    CInteger& operator++()
    {
        m_nVal++;
        return *this;
    }
private:
    int m_nVal;
};

int main()
{
    CInteger n(999);
    //++n;
    //++(++n);  // n, 返回一个整数，其值为右值，不能再次修改

    //对于上面的问题，需要修改运算符的返回值类型，应该修改成返回其类型的引用
    ++(++n);

    int n1 = 666;
    ++(++n1);

    //(n1 + 999) = 100; // n1 是一个右值(加法的操作是个右值)，不能被修改
    return 0;
}
```

后++运算符

前++ 可以连续，而后++ 不能连续++
代码示例：

```
class CInteger
{
public:
  CInteger(int nVal) : m_nVal(nVal)
  {

  }
  void SetVal(int nVal)
  {
    m_nVal = nVal;
  }

  int GetVal()const
  {
    return m_nVal;
  }

  int operator+(int nVal)
  {
    //此句在这里，不能改变原有运算符的意义
    // m_nVal = nVal;
    return m_nVal + nVal;
  }
  // 运算符重载一般写成友元
  friend int operator+(int nVal, CInteger& interobj)
  {
    return nVal + interobj.GetVal();
  }

  // 前++
  CInteger& operator++()
  {
    m_nVal++;
    return *this;
  }
   // 后++，和前++的区别再于参数类型不一样，后++多了一个类型占位符
  int operator++(int)
  {
    int nOld = m_nVal;
    m_nVal++;
    // 返回++前的值
    return nOld;
  }
  //CIntager operator++(int)
  //{
  //  int nOld = m_nVal;
```

```
    //  m_nVal++;
    //  // 返回++前的值
    //  return CIntager(nOld);

    //  // main 函数中的 int nVal = n2++; 编译不通过
    //  //error C2440: “初始化”: 无法从“CIntager”转换为“int”
    //}

private:
    int m_nVal;
};

int main()
{
    CInteger n(999);
    //++n;
    //++(++n);  // n, 返回一个整数，其值为右值，不能再次修改

    //对于上面的问题，需要修改运算符的返回值类型，应该修改成返回其类型的引用
    ++(++n);

    int n1 = 666;
    ++(++n1);
    //(n1 + 999) = 100; // n1 是一个右值(加法的操作是个右值)，不能被修改

    CInteger n2(50);
    ++n2;

    int nVal = n2++;
    //(n2++)++;   //前++ 可以连续，而后++ 不能连续++
    return 0;
}
```

关系运算符

```
==, !=, &&, ||, >=, <=, ||
```

```
class CString
{
public:
    CString(const char* str)
    {
        m_str = new char[strlen(str) + 1];
```

```
        strcpy(m_str, str);
    }
    // ==
    bool operator==(const char* str)
    {
        return strcmp(m_str, str) == 0;
    }
    // !=
    bool operator!=(const char* str)
    {
        // return strcmp(m_sz, str) != 0;
        // 调用 == 的结果，对其取反就是不等于
        return !(*this == str);
    }

    friend bool operator==(const char* str, CString& pStrObj)
    {
        return pStrObj == str;
    }

private:
    char* m_str;
};
int main()
{
    CString str1("Hello");
    bool bRet = (str1 == "Hello");
    cout << bRet << endl;

    // !=
    bRet = (str1 != "Hello");
    cout << bRet << endl;


    if (!("World" == str1))
    {
    cout << false << endl;
    };
    return 0;
}
```

位运算符

```
<< >> & | ^ ~
```

```
class CString
{
public:
    CString(const char* str)
    {
    m_str = new char[strlen(str) + 1];
    strcpy(m_str, str);
    }

    //friend void operator<<(ostream& os, CString& str)
    //{
    //  os << str.m_str;
    //}
    friend ostream& operator<<(ostream& os, CString& str)
    {
    os << str.m_str;
    //之所以返回 ostream 类对象的引用，是为了能够连续读取复数
    return os;
    }
private:
    char* m_str;
};


int main()
{
    //char szBuff[0xff] = { 0 };
    //cin >> szBuff;
    //cout << szBuff;

    CString str("Hello");
    cout << str;

    /*
    error C2296: “<<”: 非法，左操作数包含“void”类型
    friend void operator<<(ostream& os, CString& str){os << str.m_str;}

    修改返回值类型，修改如下：
    friend ostream& operator<<(ostream& os, CString& str){os <<
str.m_str; return os}

    */
    cout << str << 999 << " " << true << endl;

    return 0;
}
```

new 和 delete

new和delete可以自动帮助我们申请内存和释放内存，我们还可以使用运算符重载，自己来实现
new 申请内存，初始化调用构造

手动实现new、delete时运用的场景：当程序频繁的申请内存且申请的次数过大时，可以考虑自己手动来实现。手动实现new、delete，此时的申请效率和一条赋值语句的效率基本一致。

```
// new 和 delete
class CString
{
public:
    CString(const char* str)
    {
        m_str = new char[strlen(str) + 1];
        strcpy(m_str, str);
    }
    // new
    void* operator new(size_t nSize)
    {
        return malloc(nSize);
    }

    // delete
    void operator delete(void* p)
    {
        free(p);
    }
private:
    char* m_str;
};

int main()
{
    // new：初始化内存，调用构造，这里申请了一块CString*大小的内存空间
    CString* str = new CString("Hello");
    delete(str);
    return 0;
}
```

智能指针

类对象的定义方式：全局、栈、堆
栈区和全局的申请的对象编译器会自动释放其申请的内存，而堆区申请的内存就需要手动释放申请的空间。

解决问题：在通过 new 对象的时候，解决忘记使用 delete 释放内存

可以通过以下的方式在new 对象后，程序结束时自动进行释放内存

代码示例：

```
/*
智能指针：只需要new，不需要delete
*/
class CInteger
{
public:
  CInteger()
  {
    cout << "CInteger::CInteger" << endl;
  }
  ~CInteger()
  {
    cout << "CInteger::~CInteger" << endl;
  }
  void SetVal(int nVal)
  {
    m_nVal = nVal;
  }
  int GetVal()const
  {
    return m_nVal;
  }
private:
  int m_nVal;
};

/*
将一个类指针放置在一个类中，这个指针是指向这个类的
这里类仅限于全局、栈区、局部的
*/
class CPointer
{
public:
    CPointer(CInteger* pInt) : m_pInt(pInt)
    {
    }
    // 为了让程序自动调用delete，将delete放置在该类中的析构函数内
    // 当该对象被销毁的时候，程序会自动调用该析构，以到达自动释放的效果
```

```
    ~CPointer()
    {
        delete m_pInt;
    }
    CInteger*  operator->()
    {
        return m_pInt;
    }
private:
    CInteger* m_pInt;
};
int main()
{
    // 堆申请的对象生命周期，new 开始 delete 结束
    //CInteger* pInt = new CInteger;
    //delete pInt;

    // 有时候会忘记手动释放申请的内存，可不可以让程序自动进行释放申请过内存？

    CPointer pt(new CInteger);
    CPointer pt1(new CInteger);
    CPointer pt2(new CInteger);

    //pt.m_pInt->GetVal();

    // 重载->运算符，使其在通过CPointer类对象直接调用CInteger中的方法
    pt->GetVal();
    return 0;
}
```

指针的操作

-> ++ -- * & **

```
/*
智能指针：只需要new，不需要delete
*/
class CInteger
{
public:
   CInteger()
   {
       cout << "CInteger::CInteger" << endl;
   }
   ~CInteger()
   {
       cout << "CInteger::~CInteger" << endl;
```

```
    }
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }
    int GetVal()const
    {
        return m_nVal;
    }
    // 强转类型
    float operator()()
    {
        return m_nVal;
    }
private:
    int m_nVal;
};

/*
将一个类指针放置在一个类中，这个指针是指向这个类的
这里类仅限于全局、栈区、局部的
*/
class CPointer
{
public:
    CPointer(CInteger* pInt) : m_pInt(pInt)
    {
    }
    // 为了让程序自动调用delete，将delete放置在该类中的析构函数内
    // 当该对象被销毁的时候，程序会自动调用该析构，以到达自动释放的效果
    ~CPointer()
    {
        delete m_pInt;
    }
    // 重载 ->
    CInteger*  operator->()
    {
        return m_pInt;
    }

    // 对指针进行前++，可以进行连续++，需要是其对象的引用
    CPointer& operator++()
    {
        ++m_pInt;
        return *this;
    }
    // 对指针进行后++
```

```
CInteger* operator++(int)
{
    CInteger* pOld = m_pInt;
    ++m_pInt;
    return pOld;
// 或者返回 该类的对象 return CPointer(pOld); 返回值类型为CPointer
}

// 对指针进行前--，可以进行连续--，需要是其对象的引用
CPointer& operator--()
{
    --m_pInt;
    return *this;
}
// 对指针进行后++
CInteger* operator--(int)
{
CInteger* pOld = m_pInt;
--m_pInt;
    return pOld;
}

// 指针 取内容操作，返回值为指针 m_pInt 引用的对象
CInteger& operator*()
{
    return *m_pInt;
}

// 二级指针对指针取地址
CInteger** operator&()
{
    return &m_pInt;
}

// 对指针进行下标运算
CInteger& operator[](int nIdx)
{
    return m_pInt[nIdx];
}

// 指针加上某个数值
CInteger* operator+(int nVal);

// 对指针进行类型转换，转换运算符
CInteger* operator()()
{
    return m_pInt;
```

```
    }
private:
    CInteger* m_pInt;
};
int main()
{
    // 堆申请的对象生命周期，new 开始 delete 结束
    // CInteger* pInt = new CInteger;
    // delete pInt;

    // 有时候会忘记手动释放申请的内存，可不可以让程序自动进行释放申请过内存？

    CPointer pt(new CInteger);
    CPointer pt1(new CInteger);
    CPointer pt2(new CInteger);

    // pt.m_pInt->GetVal();

    // 重载->运算符，使其在通过CPointer类对象直接调用CInteger中的方法
    pt->GetVal();
    ++pt;
    pt++;
    /*
    重载后置++时，返回值类型为其类对象时，指针此时已经到达堆的边界处，
    调用析构进行释放时，会释放失败，这里的返回值以及返回值类型不应该是引用计数无名对象
    应该返回一个指针
    */
    --pt;
    pt--;

    // 指针取内容
    // (*pInt).GetVal(); //取玩指针的内容拿到这个对象，然后就可以调用类中的方法
    (*pt).SetVal(10);
    (*pt).GetVal();

    // 二级指针对指针取地址
    &pt;

    // 对指针进行下标运算
    pt[5].GetVal();

    // 对指针进行类型转换
    void* p = (void*)pt();
```

```
    CInteger n;
    cout << n() << endl; //对其进行强转，将 n 强转成其它的类型


    // 存在 Bug
    CPointer pt3(pt);
    return 0;
}
```

智能指针Bug

```
    CPointer pt(new CInteger);
    // 存在 Bug --> 重复释放析构，浅拷贝问题，解决办法：引用计数
    CPointer pt1(pt);
    return 0;
```