Lecture 12

Object-Oriented Programming VIII

Assignment Operator, Operator Overloading, and the Rule of Three

Prof. Hyeong-Seok Ko
Seoul National University
Graphics & Media Lab

Contents

- The Assignment Operator (13.2)
- Rule of Three (13.3)

Function Overloading

```
class vec2 {
public :
   vec2() : x(0), y(0) {}
   vec2(float a, float b) : x(a), y(b) {}

   float x,y;
};

int add (const vec2& a, const vec2& b) {
   return vec2(a.x + b.x, a.y + b.y);
}

float add (float & a, float & b) {
   return a + b;
}

void main() {
   vec2 a(1.1f,0), b(1.3f,2.5f);
   vec2 x0 = add(a,b);
   float x = 1, y = 2, z;
   z = add(x,y)
}
```

Functions vs. Operators

```
Vec2 add(const Vec2& a, const Vec2& b) {
   // ……
}

Vec2 operator+(const Vec2& a, const Vec2& b) {
   // ……
}

Vec2 a(0,0), b(0,1), v;

v = add(a, b)
v = a + b;
```

Non-member Operator Overloading

```
class Vec2 {
public :
   Vec2() : x(0), y(0) {}
   Vec2(float a, float b) : x(a), y(b) {}

   float x,y;
};

Vec2 operator+(const Vec2& a, const Vec2& b) { return
Vec2(a.x + b.x, a.y + b.y); }

void main() {
   Vec2 a(1.1f,0), b(1.3f,2.5f);
   Vec2 x0 = a + b;
}
```

Member Operator Overloading

Another way of defining + on vec2

```
class vec2 {
public :
   vec2() : x(0), y(0) {}
   vec2(float a, float b) : x(a), y(b) {}

   vec2 operator+(const vec2& a)
   { return vec2(x + a.x, y + a.y); }

   float x,y;
};

void main() {
   vec2 a(1.1f,0), b(1.3f,2.5f);
   vec2 x0 = a + b;
}
```

Non-member or member operator?
Either way is fine, as long as only one of them is defined.

Some operators must be defined as a member operator.
Assignment must be overloaded as a member operator.

Operators which can be Overloaded

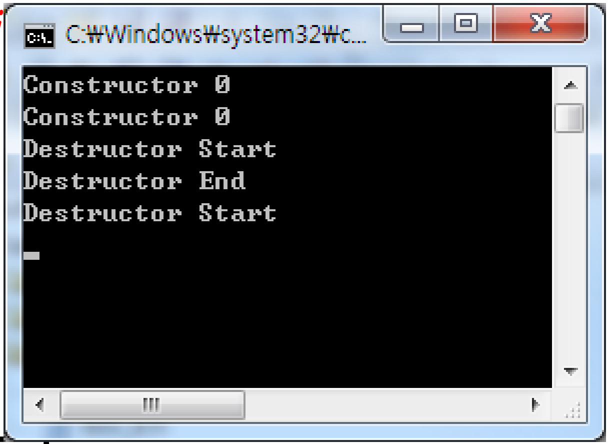
Operators which can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators which cannot be overloaded				
.	.*	::	?:	sizeof

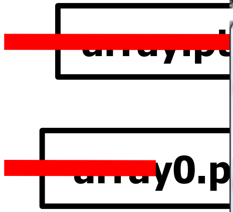
Default Assignment Operator

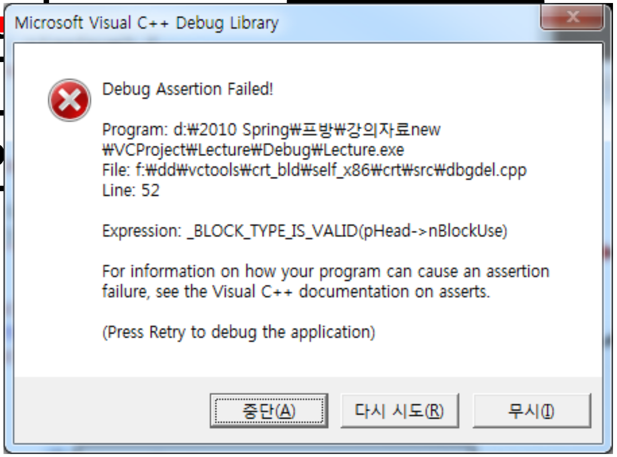
A problem exists in thi

```
#include <iostream>

class Array {
public :
   Array(std::size_t num) : size(num) {
      std::cout << "Constructor 0" << std::endl;
      ptr = new int[num];
   }
   Array(const Array& arr) : size(arr.size) {
      std::cout << "Copy Constructor" << std::endl;
      ptr = new int[size];
      for(std::size_t i=0;i<size;++i)
         ptr[i] = arr.ptr[i];
   }
   ~Array() {
      std::cout << "Destructor Start" << std::endl;
      if(ptr != NULL) delete [] ptr;
      std::cout << "Destructor End" << std::endl;
   }

   int *          ptr;
   std::size_t    size;
};

void f() {
   Array array(5);
   Array array0(10);
   array0 = array;
}

void main() {
   f();
}
```



```
Constructor 0
Constructor 0
Destructor Start
Destructor End
Destructor Start
_
```





Debug Assertion Failed!

Program: d:₩2010 Spring₩프방₩강의자료new ₩VCProject₩Lecture₩Debug₩Lecture.exe
File: f:₩dd₩vctools₩crt_bld₩self_x86₩crt₩src₩dbgdel.cpp
Line: 52

Expression: _BLOCK_TYPE_IS_VALID(pHead->nBlockUse)

For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

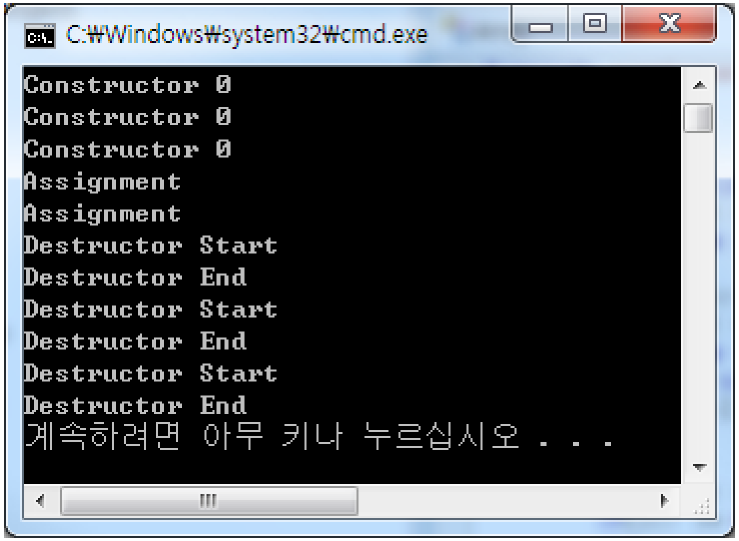
(Press Retry to debug the application)

```
array0.ptr = array.ptr;
array0.size = array.size;
// top-level member-wise assignment
```

Your own assignment operator needs to be defined…

How to Define Your Own Assignment Operator?

```
class Array {
public :
   Array(std::size_t num) : size(num) {
      std::cout << "Constructor 0" << std::endl;
      ptr = new int[num];
   }
   Array(const Array& arr) : size(arr.size) {
      std::cout << "Copy Constructor" << std::endl;
      ptr = new int[size];
      for(std::size_t i=0;i<size;++i)
         ptr[i] = arr.ptr[i];
   }
   ~Array() {
      std::cout << "Destructor Start" << std::endl;
      if(ptr != NULL) delete [] ptr;
      std::cout << "Destructor End" << std::endl;
   }
   Array& operator=(const Array& arr) {
      std::cout << "Assignment" << std::endl;
      if(ptr != NULL) delete [] ptr;
      size = arr.size;
      ptr = new int[arr.size];
      for(std::size_t i=0;i<size;++i)
         ptr[i] = arr.ptr[i];
      return (*this);
   }

   int *         ptr;
   std::size_t   size;
};
void f() {
   Array array(5);
   Array array0(10), array1(10);
   array1 = array0 = array;
}
void main() { f(); }
```



```
Constructor 0
Constructor 0
Constructor 0
Assignment
Assignment
Destructor Start
Destructor End
Destructor Start
Destructor End
Destructor Start
Destructor End
계속하려면 아무 키나 누르십시오 . . .
```

Rule of Three

- The rule of three is a rule of thumb in C++ that advises that if a class explicitly defines one of the following, probably it should explicitly define all three.
  - destructor
  - copy constructor
  - assignment operator

- Above three functions are special member functions that are automatically created by the compiler if they are not explicitly defined by the programmer.

- If one of these had to be defined by the programmer, it means that the compiler-generated versions of the other two probably do not fit the needs of the class, thus need to be redefined.

Rule of Three

- In the previous example

```
class Array {
public :
   Array(std::size_t num) : size(num) {
      std::cout << "Constructor 0" << std::endl;
      ptr = new int[num];
   }
```

```
   Array(const Array& arr) : size(arr.size) {
      std::cout << "Copy Constructor" << std::endl;
      ptr = new int[size];
      for(std::size_t i=0;i<size;++i)
         ptr[i] = arr.ptr[i];
   }
```

Copy constructor

```
   ~Array() {
      std::cout << "Destructor Start" << std::endl;
      if(ptr != NULL) delete [] ptr;
      std::cout << "Destructor End" << std::endl;
   }
```

Destructor

```
   Array& operator=(const Array& arr) {
      std::cout << "Assignment" << std::endl;
      if(ptr != NULL) delete [] ptr;
      size = arr.size;
      ptr = new int[arr.size];
      for(std::size_t i=0;i<size;++i)
         ptr[i] = arr.ptr[i];
      return (*this);
   }
```

Assignment Operator

```
   int *          ptr;
   std::size_t   size;
};
```