Mismatching scalar and vector new and delete

devblogs.microsoft.com/oldnewthing/20040203-00

February 3, 2004

Raymond Chen

In a previous entry I alluded to the problems that can occur if you mismatch scalar "new" with vector "delete[]" or vice versa.

There is a nice description of C++ memory management in C++ Gotchas: Avoiding Common Problems in Coding and Design on www.informit.com, and I encourage you to read at least the section titled Failure to Distinguish Scalar and Array Allocation before continuing with this entry, because I'm going to use that information as a starting point.

Here's how the Microsoft C++ compiler manages vector allocation. Note that this is internal implementation detail, so it's subject to change at any time, but knowing this may give a better insight into why mixing scalar and vector new/delete is a bad thing.

The important thing to note is that when you do a scalar "delete p", you are telling the compiler, "p points to a single object." The compiler will call the destructor once, namely for the object you are destructing.

When you do "delete[] p", you are saying, "p points to a bunch of objects, but I'm not telling you how many." In this case, the compiler needs to generate extra code to keep track of how many it needs to destruct. This extra information is kept in a "secret place" when the vector is allocated with "new[]".

Let's watch this in action:

```
class MyClass {
public:
  MyClass(); // constructor
  ~MyClass(); // destructor
  int i;
};
MyClass *allocate_stuff(int howmany)
{
    return new MyClass[howmany];
}
```

The generated code for the "allocate_stuff" function looks like this:

```
    push    esi
    mov     esi, [esp+8] ; howmany
   ;  eax = howmany * sizeof(MyClass) + sizeof(size_t)
    lea     eax, [esi*4+4]
    push    eax
    call    operator new
    test    eax, eax
    pop     ecx
    je      fail
    push    edi
    push    OFFSET MyClass::MyClass
    push    esi
    lea     edi, [eax+4] ; edi = eax + sizeof(size_t)
    push    4 ; sizeof(MyClass)
    push    edi
    mov     [eax], esi ; howmany
    call    `vector constructor iterator'
    mov     eax, edi
    pop     edi
    jmp     done
fail:
    xor     eax, eax
done:
    pop     esi
    retd    4
```

Translated back into pseudo-C++, the code looks like this:

```
MyClass* allocate_stuff(int howmany)
{
  void *p = operator new(
     howmany * sizeof(MyClass) + sizeof(size_t));
  if (p) {
    size_t* a = reinterpret_cast<size_t*>(p);
    *a++ = howmany;
    vector constructor iterator(a,
      sizeof(MyClass), &MyClass::MyClass);
    return reinterpret_cast<MyClass*>(a);
  }
  return NULL;
}
```

In other words, the memory layout of the vector of MyClass objects looks like this:

howmany
MyClass[0]
MyClass[1]

…
MyClass[howmany-1]

The pointer returned by the new[] operator is **not** the start of the allocated memory but rather points to MyClass[0]. The count of elements is hidden in front of the vector.

The deletion of a vector performs this operation in reverse:

```
void free_stuff(MyClass* p)
{
    delete[] p;
}
```

generates

```
    mov     ecx, [esp+4] ; p
    test    ecx, ecx
    je      skip
    push    3
    call    MyClass::`vector deleting destructor`
skip
    ret     4
```

Translated back into pseudo-C++, the code looks like this:

```
void free_stuff(MyClass* p)
{
  if (p) p->vector deleting destructor(3);
}
```

The vector deleting destructor goes like this (pseudo-code):

```
void MyClass::vector deleting destructor(int flags)
{
  if (flags & 2) { // if vector destruct
    size_t* a = reinterpret_cast<size_t*>(this) - 1;
    size_t howmany = *a;
    vector destructor iterator(p, sizeof(MyClass),
      howmany, MyClass::~MyClass);
    if (flags & 1) { // if delete too
      operator delete(a);
    }
  } else { // else scalar destruct
    this->~MyClass(); // destruct one
    if (flags & 1) { // if delete too
      operator delete(this);
    }
  }
}
```

The vector deleting destructor takes some flags. If 2 is set, then a vector is being destructed; otherwise a single object is being destructed. If 1 is set, then the memory is also freed.

In our case, the flags parameter is 3, so we will perform a vector destruct followed by a delete. Observe that this code sucks the original “howmany” value out of its secret hiding place and asks the vector destructor iterator to run the destructor that many times before freeing the memory.

So now, armed with this information, you should be able to describe what happens if you allocate memory with scalar “new” and free it with vector “delete[]” or vice versa.

Bonus exercise: What optimizations can be performed if the destructor MyClass::~MyClass() is removed from the class definition?

Answers to come tomorrow.

Raymond Chen

Follow