Item 16: Use the same form in corresponding uses of `new` and `delete`.

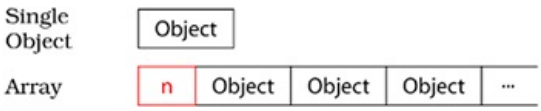
What's wrong with this picture?

```
std::string *stringArray = new std::string[100];

...

delete stringArray;
```

Everything appears to be in order. The `new` is matched with a `delete`. Still, something is quite wrong. The program's behavior is undefined. At the very least, 99 of the 100 `string` objects pointed to by `stringArray` are unlikely to be properly destroyed, because their destructors will probably never be called.

When you employ a *`new` expression* (i.e., dynamic creation of an object via a use of `new`), two things happen. First, memory is allocated (via a function named `operator new`—see Items 49 and 51). Second, one or more constructors are called for that memory. When you employ a *`delete` expression* (i.e., use `delete`), two other things happen: one or more destructors are called for the memory, then the memory is deallocated (via a function named `operator delete`—see Item 51). The big question for `delete` is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called.

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? It's a critical question, because the memory layout for single objects is generally different from the memory layout for arrays. In particular, the memory for an array usually includes the size of the array, thus making it easy for `delete` to know how many destructors to call. The memory for a single object lacks this information. You can think of the different layouts as looking like this, where `n` is the size of the array:



This is just an example, of course. Compilers aren't required to implement things this way, though many do.

When you use `delete` on a pointer, the only way for `delete` to know whether the array size information is there is for you to tell it. If you use brackets in your use of `delete`, `delete` assumes an array is pointed to. Otherwise, it assumes that a single object is pointed to:

```
std::string *stringPtr1 = new std::string;

std::string *stringPtr2 = new std::string[100];

...

delete stringPtr1;                      // delete an object
```

```
delete [] stringPtr2;                    // delete an array of objects
```

What would happen if you used the "[]" form on stringPtr1? The result is undefined, but it's unlikely to be pretty. Assuming the layout above, delete would read some memory and interpret what it read as an array size, then start invoking that many destructors, oblivious to the fact that the memory it's working on not only isn't in the array, it's also probably not holding objects of the type it's busy destructing.

What would happen if you didn't use the "[]" form on stringPtr2? Well, that's undefined too, but you can see how it would lead to too few destructors being called. Furthermore, it's undefined (and sometimes harmful) for built-in types like ints, too, even though such types lack destructors.

The rule is simple: if you use [] in a new expression, you must use [] in the corresponding delete expression. If you don't use [] in a new expression, don't use [] in the matching delete expression.

This is a particularly important rule to bear in mind when you are writing a class containing a pointer to dynamically allocated memory and also offering multiple constructors, because then you must be careful to use the *same form* of new in all the constructors to initialize the pointer member. If you don't, how will you know what form of delete to use in your destructor?

This rule is also noteworthy for the typedef-inclined, because it means that a typedef's author must document which form of delete should be employed when new is used to conjure up objects of the typedef type. For example, consider this typedef:

```
typedef std::string AddressLines[4];   // a person's address has 4 lines,
                                       // each of which is a string
```

Because AddressLines is an array, this use of new,

```
std::string *pal = new AddressLines;   // note that "new AddressLines"
                                       // returns a string*, just like
                                       // "new string[4]" would
```

must be matched with the *array* form of delete:

```
delete pal;                            // undefined!

delete [] pal;                         // fine
```

To avoid such confusion, abstain from typedefs for array types. That's easy, because the standard C++ library (see Item 54) includes string and vector, and those templates reduce the need for dynamically allocated arrays to nearly zero. Here, for example, AddressLines could be defined to be a vector of strings, i.e., the type vector<string>.

Things to Remember

- If you use `[]` in a `new` expression, you must use `[]` in the corresponding `delete` expression. If you don't use `[]` in a `new` expression, you mustn't use `[]` in the corresponding `delete` expression.

PREV < Day Day Up > NEXT