The initializing constructor looks like an assignment, but it isn’t

devblogs.microsoft.com/oldnewthing/20210628-00

June 28, 2021

Raymond Chen

Some time ago, I warned about the perils of the accidental C++ conversion constructor: A single-parameter constructor is considered by default to be a conversion constructor; you can opt out of this by marking the constructor `explicit`.

I gave as an example this class:

```
class Buffer
{
public:
  Buffer(size_t capacity);
  Buffer(std::initializer_list<int> values);
};
```

The `size_t` constructor is not marked as `explicit`, so it is a conversion constructor. And that permits weird things like this:

```
Buffer b = 1; // um...
```

What exactly is happening here?

A common misconception is that what’s happening is that a temporary `Buffer` is created (with the capacity 1), and then that temporary buffer is assigned to the destination buffer `b`.

That’s not what’s happening. You can prove this by deleting the assignment operators.

```
class Buffer
{
public:
  Buffer(size_t capacity);
  Buffer(std::initializer_list<int> values);
  Buffer& operator=(Buffer const&) = delete;
  Buffer& operator=(Buffer&&) = delete;
};

Buffer b = 1; // still compiles
```

(Deleting the move assignment operator is redundant because declaring the copy assignment operator automatically suppresses the implicit move assignment operator. But I deleted it explicitly for emphasis.)

Even though there is an equal-sign in the statement, there is no actual assignment.

There can't be an assignment, if you think about it, because the assignment operator assumes that `this` refers to an already-constructed object. But we don't have a constructed object yet.

According to the language rules,

```
Buffer b = 1;
```

is a *copy-initialization*, and the copy initialization is performed by taking the thing on the right-hand side and, if the types don't match,[1] it looks for a conversion constructor.

The equals sign doesn't mean assignment here. It's just a quirk of the syntax.

[1] If the types do match, then "the initializer expression is used to initialize the destination object." At this point *copy elision* kicks in:

```
extern Buffer get_buffer();

Buffer b = get_buffer();
```

The `Buffer` returned by `get_buffer()` is placed directly into the memory occupied by `b`.

Copy elision also means that

```
Buffer b = Buffer(1);
```

does not create a temporary `Buffer` of capacity 1, and then construct `b` from that temporary buffer. Instead, the `Buffer` of capacity 1 is constructed directly into `b`. The result is the same as `Buffer b(1);`.

Since the copy elision rule can be repeated,

```
Buffer b = Buffer(Buffer(Buffer(1)));
```

is also the same as `Buffer b(1);`, because each repetition of the rule strips away one of the calls to `Buffer(...)`.

Raymond Chen

Follow