The perils of the accidental C++ conversion constructor

devblogs.microsoft.com/oldnewthing/20210115-00

January 15, 2021

Raymond Chen

Consider this class:

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

You can create an uninitialized buffer with a particular capacity, or you can create an initialized buffer.

The one-parameter constructor also serves as a conversion constructor, resulting in the following:

```
Buffer buffer(24); // create a buffer of size 24
Buffer buffer({ 1, 3, 5 }); // create an initialized 3-byte buffer
```

Okay, those don't look too bad. But you also get this:

```
Buffer buffer = 24; // um...
Buffer buffer = { 1, 3, 5 };
```

These are equivalent to the first two versions, but you have to admit that the `= 24` version looks really weird.

You also get this:

```
extern void Send(Buffer const& b);
Send('c'); // um...
```

This totally compiles, but it doesn't send the character `'c'`, which is what it looks like. Instead, it creates an uninitialized buffer of size `0x63 = 99` and sends it.

If this is not what you intended, then you would be well-served to use the `explicit` keyword to prevent a constructor from being used as conversion constructions.

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

I made the first constructor explicit, since I don’t want you to pass an integer where a buffer is expected. However, I left the initializer list as a valid conversion constructor because it seems reasonable to let someone write

```
Send({ 1, 2, 3 });
```

Raymond Chen

Follow