

Re: Problem with linker

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-06/msg00487.html>

- *From:* Joseph M. Newcomer <newcomer@xxxxxxxxxxxxx>
 - *Date:* Fri, 08 Jun 2007 01:10:17 -0400
-

See below...

On Thu, 07 Jun 2007 19:19:14 -0500, "Doug Harrison [MVP]" <dsh@xxxxxxxxx> wrote:

On Thu, 07 Jun 2007 14:38:39 -0400, Joseph M. Newcomer <newcomer@xxxxxxxxxxxxx> wrote:

In fact, I agree that the it is the result of "carelessness", but the correct way to handle it in C# would be

```
Something();
Something(string);
Something(int);
```

so I believe the correct solution was not to add a default parameter to the constructor,
but to have actually written a default constructor.

Note that if it was

```
Something();
Something(LPCTSTR);
Something(int);
```

then we still have an ambiguity problem if someone calls

```
Something(NULL);
```

In terms of being able to create Something objects, there is no difference between writing:

```
// NB: This is a default ctor!
Something(LPCTSTR = 0);
```

and its decomposition:

```
Something();
Something(LPCTSTR);
```

The former is just syntactic sugar for the latter. (Again, I'm ignoring what the ctors actually do and focusing on how they may be used to create objects.) That's what I was getting at when I said:

Re: Problem with linker

It's not default arguments per se that cause problems. A function declaration that uses default arguments can always be decomposed into a set of function declarations that do not. Ambiguities arise with carelessly designed overloading, such as overloading on int and pointer types. That's the real mistake in the example.

What you've done is decompose `Something(LPCTSTR = 0)` into the equivalent set of function declarations. Doing this doesn't alter your ability to create `Something` objects, nor does it affect ambiguity WRT function overloading.

Yes, I agree except for the last line. If I later add a constructor

`Something(int)`

then I have a problem because

`Something()`

resolves to

`Something(0)`

and therefore we now have an ambiguity between

`Something(int)`

and

`Something(LPCTSTR)`

I've been done in by this before I began to avoid certain kinds of default overloading (I still use default parameters, I'm just a lot more cautious about what I do with them)

My own preference to avoid this is to declare them as

`Something();`

`Something(const CString & s);`

`Something(int);`

so it is impossible to write `Something(NULL)`; the programmer is forced to write

`Something()` to get a default parameter.

Bad example (or language, take your pick <g>). Because `NULL` is `#defined` to `0` (or some other integral constant expression equal to zero), and any conversion of `0` to `CString` requires a user-defined conversion, which is worse than any integral conversion that may be necessary to convert `NULL` to `int`, `Something(NULL)` will select `Something(int)`.

Re: Problem with linker

My recollection is that there's a compilation error. I should try this again with VS.NET to see what happens

This often arises when the correct breakdown should have been

```
Something() { Init(NULL); }  
Something(LPCTSTR p) { Init(p); }  
Something(int n)
```

where I could consider implementations such as

```
{ CString s; s.Format(_T("%d"), n); Init(s); }
```

or

```
{ Init(n); }
```

given that I define functions

protected:

```
void Init(LPCTSTR);
```

```
void Init(int);
```

Either of the above would be better alternatives to using default parameters.

The issue

here is a reluctance to refactor code; default parameters make it easy to reuse the

existing code instead of adding another function to handle the indirection, and since

adding code is Always A Bad Thing (an attitude that sets my teeth on edge, and I think is

the result of poor education), the quick & dirty hack is to add a default parameter.

The thing is, `Init(NULL)` is going to be (1) ambiguous or (more likely) (2) select `Init(int)`. In any case, because any integral constant zero expression is a null pointer constant, the class will have to treat the integer value zero as a null pointer constant; that is, `T(0)`, where `T` is any integer type, and `LPCTSTR(0)` have to be treated the same way. If anything, using default arguments would have helped you here, because if you restored it to:

```
Something(LPCTSTR p = 0) { Init(p); }
```

Then `Something()` would call `Init(LPCTSTR)`, because then `p` would have the type `LPCTSTR` instead of `int`. To fix your default ctor, you would need to say:

```
Something() { Init(LPCTSTR(NULL)); }
```

I think the messiness we are discussing is a good argument against using default arguments. The implicit conversions are a serious problem.

Re: Problem with linker

I once worked in a language where it was trivial to write the equivalent to this (illegal C++) code:

```
class x_coordinate : public int {};  
class y_coordinate : public int {};
```

and it was impossible to write a function that took the wrong coordinate. I could even write

```
class x_screen_coordinate : public int {};  
class x_client_coordinate : public int {};
```

and there could never be any confusion about whether we were in screen or client coordinates. The language actually supported the concept of dimensional types, e.g.

```
class acceleration : public double {};  
class distance : public double {};  
class time : public double {}  
class velocity: public double {}
```

```
time_squared operator *(time t1, time t2);  
acceleration operator *(distance d, time_squared t2);  
distance operator *(velocity r, time t);  
velocity operator / (distance d, time t)
```

if you made any error, it was caught by the compiler. I could spend a week getting something to compile, and it ran perfectly the first time.

if I wanted to have an absolute value, I could use the casting operator: which in those languages was ! as an infix operator

```
acceleration G = distance ! 9.8 / time_squared ! 1.0;  
where I defined the ! casting operators in the class. (The compiler happily optimized the / 1.0 out of existence!) I could do things like
```

```
in one_inch = in ! cm ! 2.54;  
cm one_cm = cm ! 1;  
in d = in ! one_cm;
```

and the conversions would be done. I used it to write a storage allocator where we had raw storage, storage blocks, and user blocks as abstractions. We **always** knew what we were pointing to!

Needless to say, I consider C++ a step backwards...
joe

Joseph M. Newcomer [MVP]
email: newcomer@xxxxxxxxxxxxx
Web: <http://www.flounder.com>
MVP Tips: http://www.flounder.com/mvp_tips.htm

Re: Problem with linker

Re: Problem with linker