Gotchas when using linker sections to arrange data, part 1

devblogs.microsoft.com/oldnewthing/20181108-00

November 8, 2018

Raymond Chen

We saw last time that you can use linker sections to arrange the order in which data appears in the module. We ended with a diagram like this:

`mydata$a`	`firstInitializer`	`main.obj`	
`mydata$g`	`DoThisSooner3`	`file3.obj`	unspecified order
	`DoThisSooner4`	`file4.obj`	
`mydata$m`	`Function2`	`file2.obj`	unspecified order
	`Function1`	`file1.obj`	
	`Function3`	`file3.obj`	
`mydata$t`	`DoThisLater2`	`file2.obj`	unspecified order
	`DoThisLater4`	`file4.obj`	
`mydata$z`	`lastInitializer`	`main.obj`	

Based on this table, we would be tempted to write code like this:

```
// Code in italics is wrong.
void NaiveInitializeAllTheThings()
{
    const INITIALIZER* initializer = &firstInitializer + 1;
    while (initializer < &lastInitializer) {
      (*initializer++)();
    }
}
```

From a language lawyer standpoint, this code is not valid because it dereferences a pointer beyond the end of an object, and because it compares two pointers which are not part of the same aggregate. We can fix this by switching to `uintptr_t` as our currency.

```
// Code in italics is still wrong.
void LessNaiveInitializeAllTheThings()
{
    auto begin = (uintptr_t)&firstInitializer
                 + sizeof(firstInitializer);
    auto end = (uintptr_t)&lastInitializer;
    for (auto current = begin; current < end;
         current += sizeof(INITIALIZER)) {
      auto initializer = *(const INITIALIZER*)current;
      initializer();
    }
}
```

The conversion between pointers and `uintptr_t` is implementation-defined (rather than undefined), so this avoids the undefined behavior problems of using pointers to walk between two global variables.

But the code is still not right, because it fails to take into account another detail of linker sections: intra-section padding.

The linker will add padding after a fragment in order to satisfy any alignment requirements of the subsequent fragment. That's expected.

What most people aren't aware of is that the linker is permitted but not required to add padding after each fragment, up to the section's alignment. In practice, you are likely to see this "unnecessary" padding when using incremental linking.

In all cases, the padding bytes (if any) will be zero.

`mydata$a`	`firstInitializer`	`main.obj`	
	Optional padding		
`mydata$g`	`DoThisSooner3`	`file3.obj`	unspecified order
	Optional padding		
	`DoThisSooner4`	`file4.obj`	
	Optional padding		
`mydata$m`	`Function2`	`file2.obj`	unspecified order
	Optional padding		
	`Function1`	`file1.obj`	
	Optional padding		

	`Function3`	`file3.obj`	
	Optional padding		
`mydata$t`	`DoThisLater2`	`file2.obj`	unspecified order
	Optional padding		
	`DoThisLater4`	`file4.obj`	
	Optional padding		
`mydata$z`	`lastInitializer`	`main.obj`	
	Optional padding		

To accommodate padding, we need to skip over any possible null pointers.

```
void InitializeAllTheThings()
{
    auto begin = (uintptr_t)&firstInitializer
                 + sizeof(firstInitializer);
    auto end = (uintptr_t)&lastInitializer;
    for (auto current = begin; current < end;
         current += sizeof(INITIALIZER)) {
      auto initializer = *(const INITIALIZER*)current;
      if (initializer) initializer();
    }
}
```

We'll look at another consequence of padding next time.

Raymond Chen

Follow