# Using linker segments and __declspec(allocate(…)) to arrange data in a specific order

devblogs.microsoft.com/oldnewthing/20181107-00

November 7, 2018

Raymond Chen

You can declare a section and then start generating data into it.

```
#pragma section("mydata$a", read, write)
__declspec(allocate("mydata$a")) int i = 0;

#pragma section("mydata$b", read, write)
__declspec(allocate("mydata$b")) int j = 0;
```

The `#pragma section` directive lets you define a new section and assign attributes. You can then place data into that section with the `__declspec(allocate(...))` attribute.

When the linker combines all the little bits and pieces of data, it does the following:

- It takes the section names and splits them at the first dollar sign. (If there is no dollar sign in the section name, then the entire string is treated as the "before the first dollar sign" portion.)
- The portion before the dollar sign is the name of the section in the generated module.
- The portion after the dollar sign, if any, is used to sort the fragments within a section.

It is common to take advantage of the "sorts the data fragments alphabetically" step by generating data into a carefully-named sequence of sections so that they can iterate over all the objects in the middle section:

```
typedef void (*INITIALIZER)();

#pragma section("mydata$a", read)
__declspec(allocate("mydata$a")) const INITIALIZER firstInitializer = nullptr;

#define ADD_INITIALIZER_TO_SECTION(fn, s) \
    __declspec(allocate("mydata$" s)) \
    const INITIALIZER initializer##fn = fn

#pragma section("mydata$g", read)
#pragma section("mydata$m", read)
#pragma section("mydata$t", read)

#define ADD_EARLY_INITIALIZER(fn) ADD_INITIALIZER_TO_SECTION(fn, "g")
#define ADD_INITIALIZER(fn)       ADD_INITIALIZER_TO_SECTION(fn, "m")
#define ADD_LATE_INITIALIZER(fn)  ADD_INITIALIZER_TO_SECTION(fn, "t")

#pragma section("mydata$z", read)
__declspec(allocate("mydata$z")) INITIALIZER lastInitializer = nullptr;

// In various files

// file1.cpp
ADD_INITIALIZER(Function1);

// file2.cpp
ADD_INITIALIZER(Function2);
ADD_LATE_INITIALIZER(DoThisLater2);

// file3.cpp
ADD_INITIALIZER(Function3);
ADD_EARLY_INITIALIZER(DoThisSooner3);

// file4.cpp
ADD_EARLY_INITIALIZER(DoThisSooner4);
ADD_LATE_INITIALIZER(DoThisLater4);
```

The idea is that anybody who needs to add an initializer declares a function pointer in the `mydata$g`, `mydata$m`, or `mydata$t` section. The linker will collect all of those function pointers from same-named sections together, and then sort the sections, so that the final order of fragments in the `mydata` section is

| | | | |
|---|---|---|---|
| `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` | |

The `InitializeAllTheThings` function then walks through all the function pointers between `firstInitializer` and `lastInitializer` and calls each one.

The alphabetical ordering rule ensures that the `mydata$a` fragment comes first, so that `firstInitializer` has the lowest address. Next comes the `mydata$g` fragments, which contain the early initializers. Following that are the `mydata$m` fragments, which are the regular initializers. Next are the `mydata$t` fragments, which contain the late initializers. And finally the `mydata$z` fragment, which contains `lastInitializer`.

Now that we understand the principle behind section grouping and sorting, we can look at the gotchas next time.

Raymond Chen

**Follow**