```
// b.cpp:
int *b = new int;       // runtime initialization of file-scope variable
int setB(int i)         // Initialize b.
{
    *b = i;             // Populate the allocated heap memory.
    return 0;           // Return successful status.
}

extern int setA(int);   // declaration (only) of setter in other TU
int x = setA(5);        // Initialize a and b.
int main()              // main program entry point
{
    return 0;           // Return successful status.
}
```

These two translation units will be initialized before `main` is entered in some order, but regardless of that order, the program in the example above will wind up dereferencing a null pointer before entering `main`:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.out
  Segmentation fault (core dumped)
```

Suppose we were to instead move the file-scope **`static`** pointers, corresponding to both `setA` and `setB`, inside their respective function bodies:

```
// a.cpp:
extern int setB(int);   // declaration only of setter in other TU
int setA(int i)         // Initialize this static variable; then that one.
{
    static int *p = new int;  // runtime init of function-scope static
    *p = i;                   // Populate this static-owned heap memory.
    setB(i);                  // Invoke setter to populate the other one.
    return 0;                 // Return successful status.
}

// b.cpp: (make analogous changes)
```

Now the program reliably executes without incident:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.out
$
```

In other words, even though no order exists in which the translation units as a whole could have been initialized prior to entering `main` such that the *file*-scope variables would be valid before they were used, by instead making them *function*-scope **`static`**, we are able to guarantee that each variable is itself initialized before it is used, regardless of translation-unit-initialization order.