Chapter 18: Declaration vs Definition

Section 18.1: Understanding Declaration and Definition

A declaration introduces an identifier and describes its type, be it a type, object, or function. A declaration is what the compiler needs to accept references to that identifier. These are declarations:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

A definition actually instantiates/implements this identifier. It's what the linker needs in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

A definition can be used in the place of a declaration.

However, it must be defined exactly once. If you forget to define something that's been declared and referenced somewhere, then the linker doesn't know what to link references to and complains about a missing symbols. If you define something more than once, then the linker doesn't know which of the definitions to link references to and complains about duplicated symbols.

Exception:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

This exception can be explained using concepts of "Strong symbols vs Weak symbols" (from a linker's perspective) . Please look here (Slide 22) for more explanation.

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                  /* defines anX */
```