3 Names, Scopes, and Bindings

3.8 Separate Compilation

Probably the most straightforward mechanisms for separate compilation can be found in module-based languages such as Modula-2, Modula-3, and Ada, which allow a module to be divided into a declaration part (or *header*) and an implementation part (or *body*). As we noted in Section 3.3.4, the header contains all and only the information needed by users of the module (or needed by the compiler in order to compile such a user); the body contains the rest.

As a matter of software engineering practice, a design team will typically define module interfaces early in the lifetime of a project, and codify these interfaces in the form of module headers. Individual team members or subteams will then work to implement the module bodies. While doing so, they can compile their code successfully using the headers for the other modules. Using preliminary copies of the bodies, they may also be able to undertake a certain amount of testing.

In a simple implementation, only the body of a module needs to be compiled into runnable code: the compiler can read the header of module *M* when compiling the body of *M*, and also when compiling the body of any module that uses *M*. In a more sophisticated implementation, the compiler can avoid the overhead of repeatedly scanning, parsing, and analyzing *M*'s header by translating it into a symbol table, which is then accessed directly when compiling the bodies of *M* and its users. Most Ada implementations compile their module headers. Implementations of Modula-2 and 3 vary: some work one way, some the other.

As a practical matter, many languages allow the header of a module to be subdivided into a "public" part, which specifies the interface to the rest of the program, and a "private" part, which is not visible outside the module, but is needed by the compiler, for example to determine the storage requirements of opaque types. Ideally, one would include in the header of a module only that information that the programmer needs to know to use the abstraction(s) that the module provides. Restricted exports, and the public and private portions of headers, are compromises introduced to allow the compiler to generate code in the face of separate compilation.

At some point prior to execution, modules that have been separately compiled must be "glued together" to form a single program. This job is the task of the *linker*. At the very least, the linker must resolve cross-module references (loads, stores, jumps) and *relocate* any instructions whose encoding depends on the location of certain modules in the final program. Typically it also checks to make sure that users and implementors of a given interface agree on the version of the header file used to define that interface. In some environments, the linker may perform additional tasks as well, including certain kinds of interprocedural (whole-program) code improvement. We will return to the subject of linking in Chapters 14 and 15.

3.8.1 Separate Compilation in C

The initial version of C was designed at Bell Laboratories around 1970. It has evolved considerably over the years, but not, for the most part, in the area of separate compilation. Here the language remains comparatively primitive. In particular, there is in general no way for the compiler or the linker to detect inconsistencies among declarations or uses of a name in different files. The C89 standards committee introduced a new explanation of separate compilation based on the notion of *linkage*, but this served mainly to clarify semantics, not to change them. The current rules can be summarized as follows (certain details and special cases are omitted):

- If the declaration of a global object (variable or function) contains the word `static`, then the object has *internal linkage*, and is identified with (linked to) any other internally linked declaration of the same name in the same file.
- If the declaration of a function does not contain the keyword `static`, then it has *external linkage*, and is identified with any other (nonstatic) declaration of the same function in any file of the program. (A function declaration may consist of just the header.)
- If the declaration of a variable contains the keyword `extern`, then the variable has the same linkage as any visible, internally or externally linked declaration of the same name appearing earlier in the file. If there is no earlier declaration, then the variable has external linkage, and is identified with any other declaration of the same external variable in any file of the program. In other words, files in the same program that contain matching external variable declarations actually share the same variable. A global variable also has external linkage if its declaration says neither `static` nor `extern`.
- If an object is declared with both internal and external linkage, the behavior of the program is undefined.
- An object (variable or function) that is externally linked must have a *definition* in exactly one file of a program. A variable is defined when it is given an initial value, or is declared at the global level without the `extern` keyword. A function is defined when its body (code) is given.

Many C implementations prior to C89 relaxed the final rule to permit zero or one definitions of an external variable; some permitted more than one. In these

implementations, the linker unified multiple definitions, and created an implicit definition for any variable (or set of linked variables) for which the program contained only declarations.

The "linkage" rules of C89 provide a way to associate names in one file with names in another file. The rules are most easily understood in terms of their implementation. Most language-independent linkers are designed to deal with *symbols*: character-string names for locations in a machine-language program. The linker's job is to assign every symbol a location in the final program, and to embed the address of the symbol in every machine-language instruction that makes a reference to it. To do this job, the linker needs to know which symbols can be used to resolve unbound references in other files, and which are local to a given file. C89 rules suffice to provide this information. For the programmer, however, there is no formal notion of *interface*, and no mechanism to make a name visible in some, but not all files. Moreover, nothing ensures that the declarations of an external object found in different files will be compatible: it is entirely possible, for example, to declare an external variable as a multifield record in one file and as a floating-point number in another. The compiler is not required to catch such errors, and the resulting bugs can be very difficult to find.

Header Files

Fortunately, C programmers have developed conventions on the use of external declarations that tend to minimize errors in practice. These conventions rely on the *file inclusion* facility of a macro preprocessor. The programmer creates files in pairs that correspond roughly to the interface and the implementation of a module. The name of an interface file ends with `.h`; the name of the corresponding implementation file ends with `.c`. Every object *defined* in the `.c` file is *declared* in the `.h` file. At the beginning of the `.c` file, the programmer inserts a directive that is treated as a special form of comment by the compiler, but that causes the preprocessor to include a verbatim copy of the corresponding `.h` file. This inclusion operation has the effect of placing "forward" declarations of all the module's objects at the beginning of its implementation file. Any inconsistencies with definitions later in the file will result in error messages from the compiler. The programmer also instructs the preprocessor at the top of each `.c` file to include a copy of the `.h` files for all of the modules on which the `.c` file depends. As long as the preprocessor includes identical copies of a given `.h` file in all the `.c` files that use its module, no inconsistent declarations will occur. Unfortunately, it is easy to forget to recompile one or more `.c` files when a `.h` file is changed, and this can lead to very subtle bugs. Tools like Unix's `make` utility help minimize such errors by keeping track of the dependences among modules.

Namespaces

Even with the convention of header files, C89 still suffers from the lack of scoping beyond the level of an individual file. In particular, all global names must be distinct, across all files of a program, and all libraries to which it links. Some coding standards encourage programmers to embed a module's name in the name

of each of its external objects (e.g., `scanner_nextSym`), but this practice can be awkward, and is far from universal.

To address this limitation, C++ introduced a `namespace` mechanism that generalizes the scoping already provided for classes and functions, breaks the tie between module and compilation unit, and strengthens the interface conventions of `.h` files. Any collection of names can be declared inside a `namespace`:

EXAMPLE 3.49
Namespaces in C++

```
namespace foo {
    class foo_type_1;           // declaration
    ...
}
```

Actual definitions of the objects within `foo` can then appear in any file:

```
class foo::foo_type_1 { ...     // full definition
```

Definitions of objects declared in different namespaces can appear in the same file if desired. ■

EXAMPLE 3.50
Using names from another namespace

A C++ programmer can access the objects in a namespace using *fully qualified* names, or by *importing* (`using`) them explicitly:

```
foo::foo_type_1 my_first_obj;
```

or

```
using foo::foo_type_1;
...
foo_type_1 my_first_obj;
```

or

```
using namespace foo;    // import everything from foo
...
foo_type_1 my_first_obj;
```

There is no notion of export; all objects with external linkage in a namespace are visible elsewhere if imported. Note that linkage remains the foundation for separate compilation: `.h` files are merely a convention. ■

3.8.2 Packages and Automatic Header Inference

EXAMPLE 3.51
Packages in Java

The separate compilation facilities of Java and C# eliminate `.h` files. Specifically, Java introduces a formal notion of module, called a *package*. Every *compilation unit*, which may be a file or (in some implementations) a record in a database, belongs to exactly one package, but a package may consist of many compilation

units, each of which begins with an indication of the package to which it belongs:

```
package foo;
public class foo_type_1 { ...
```

Unless explicitly declared as `public`, a class in Java is visible in all and only those compilation units that belong to the same package. ■

EXAMPLE 3.52
Using names from another package

As in C++, a compilation unit that needs to use classes from another package can access them using fully qualified names, or via name-at-a-time or package-at-a-time import:

```
foo.foo_type_1 my_first_obj;
```

or

```
import foo.foo_type_1;
...
foo_type_1 my_first_obj;
```

or

```
import foo.*;           // import everything from foo
...
foo_type_1 my_first_obj;
```

■

When asked to import names from package *M*, the Java compiler will search for *M* in a standard (but implementation-dependent) set of places, and will recompile it if appropriate (i.e., if only source code is found, or if the target code is out of date). The compiler will then *automatically* extract the information that would have been needed in a C++ `.h` file or an Ada or Modula-3 header. If the compilation of *M* requires other packages, the compiler will search for them as well, recursively.

C# follows Java's lead in extracting header information automatically from complete class definitions. Its module-level syntax, however, is based on the namespaces of C++, which allow a single file to contain fragments of multiple namespaces. There is also no notion of standard search path in C#: to build a complete program, the programmer must provide the compiler with a complete list of all the files required.

To mimic the software engineering practice of early header file construction, a Java or C# design team can create skeleton versions of (the public classes of) its packages or namespaces, which can then be used, concurrently and independently, by the programmers responsible for the full versions.

3.8.3 Module Hierarchies

In Modula and Ada, the programmer can create a hierarchy of modules within a single compilation unit by means of lexical nesting (module `C`, for example, may

EXAMPLE 3.53
Multipart package names

be declared inside of module `B`, which in turn is declared inside of module `A`). In a similar vein, the Ada 95, Java, or C# programmer can create a hierarchy of separately compiled modules by means of *multipart names:*

```
package A.B is ...          -- Ada 95

package A.B; ...            // Java

namespace A.B { ...         // C#
```

In these examples package `A.B` is said to be a *child* of package `A`. In Ada 95 and C# the child behaves as though it had been nested inside of the parent, so that all the names in the parent are automatically visible. In Java, by contrast, multipart names work by convention only: there is no special relationship between packages `A` and `A.B`. If `A.B` needs to refer to names in `A`, then `A` must make them public, and `A.B` must import them. Child packages in Ada 95 are reminiscent of derived classes in C++, except that they support a module-as-manager style of abstraction, rather than a module-as-type style. We will consider the Ada 95 facilities further in Section 9.2.4. ■

CHECK YOUR UNDERSTANDING

49. What purpose(s) does separate compilation serve?
50. What does it mean for an external variable to be *linked* in C?
51. Summarize the C conventions for use of `.h` and `.c` files.
52. Describe the difference between a compilation unit and a C++ or C# *namespace*.
53. Explain why Ada and similar languages separate the header of a module from its body. Explain how Java and C# get by without.

DESIGN & IMPLEMENTATION

Separate compilation

The evolution of separate compilation mechanisms from early C and Fortran, through C++, Modula-3, Ada, and finally Java and C#, reflects a move from an implementation-centric viewpoint to a more programmer-centric viewpoint. Interestingly, the ability to have zero definitions of an externally linked variable in certain early implementations of C is inherited from Fortran: the assembly language mnemonic corresponding to a declaration without a definition is `.common` (for `common` block). (And as we noted in Section 3.3.1 [page 123], the lack of type checking for common blocks was originally considered a feature, not a bug!)