# 03장 이름, 유효 범위, 바인딩

## 3.7 분리 컴파일

가장 이해하기 쉬운 분리 컴파일 기법은 모듈라 2, 모듈라 3, 에이다 등과 같은 모듈 기반 언어에서 찾아볼 수 있을 것이다. 이러한 언어에서는 모듈이 선언부(또는 헤더)와 구현부(또는 몸체)로 나뉠 수 있다. 3.3.4절에서 살펴봤듯이 헤더는 모듈 사용자에게 필요한(또는 이러한 사용자를 컴파일하기 위해 컴파일러에게 필요한) 모든 정보만을 포함한다. 몸체는 나머지를 포함한다.

소프트웨어 공학의 관례상 설계 팀은 주로 프로젝트 초기에 모듈 인터페이스를 정의하고 이러한 인터페이스를 모듈 헤더의 형태로 체계화한다. 그 후 개별적인 팀원이나 하위 팀이 모듈 몸체를 구현한다. 이들은 몸체를 구현하는 동안 다른 모듈의 헤더를 사용해서 성공적으로 자신의 코드를 컴파일할 수 있다. 또 몸체들의 임시 복사본을 이용해서 어느 정도의 테스팅도 수행할 수 있다.

간단한 구현에서는 모듈 몸체만이 실행 가능한 코드로 컴파일된다. 컴파일러는 모듈 M의 몸체를 컴파일할 때와 M을 사용하는 모듈의 몸체를 컴파일할 때 모듈 M의 헤더를 읽을 수 있다. 좀 더 복잡한 구현에서 컴파일러는 M의 헤더를 심볼 테이블로 변환함으로써 반복적인 어휘 분석, 구문 분석, M의 헤더를 분석할 때의 오버헤드를 피할 수 있다. M의 몸체나 M을 사용하는 모듈의 몸체를 컴파일할 때 컴파일러는 이 심볼 테이블에 바로 접근할 수 있다. 대부분의 에이다 구현은 모듈 헤더를 컴파일한다. 모듈라 2와 모듈라 3의 구현은 다양해서 일부는 몸체만 컴파일하며 나머지는 헤더까지 컴파일한다.

## 【3.7.1】 C의 분리 컴파일

C의 초기 버전은 1970년경에 벨 연구소에서 설계되었다. 그 후 C는 상당히 발전했지만 분리 컴파일과 관련된 부분은 그다지 발전하지 못했다. 분리 컴파일 분야에 있어 C는 상대적으로 기초적인 단계에 머물러있다. 특히 C에서는 일반적으로 컴파일러나 링커(분리 컴파일된 파일들을 하나의 최종 프로그램으로 결합시키는 프로그램)가 여러 파일에 존재하는 이름의 선언이나 사용 간의 모순을 탐지할 수 없다. C89 표준 위원회는 링크 개념에 기반한 분리 컴파일의 새로운 설명을 도입했지만 이는 주로 의미를 명확하게 하는 것이지 의미를 변경하는 것은 아니었다. C의 현재 규칙은 다음과 같이 요약할 수 있다(특정 세부 사항과 특수한 경우는 생략했다).

- 전역 객체(변수나 함수)의 선언이 단어 `static`을 포함하면 이 객체는 내부 링크를 가지며 동일한 파일에서 내부적으로 링크된 동일한 이름의 선언과 동일시된다(링크된다).
- 함수 선언이 키워드 `static`을 포함하고 있지 않다면 이 함수는 외부 링크를 가지며 프로그램의 아무 파일에 있는 동일한 함수의 (비정적) 선언과 동일시된다(함수 선언은 헤더만으로 구성될 수 있다).
- 변수 선언이 키워드 `extern`을 포함하면 변수는 파일의 좀 더 앞부분에 나오는 동일한 이름의 내부적으로나 외부적으로 링크된 볼 수 있는 선언과 동일한 링크를 가진다. 좀 더 앞선 선언이 없다면 변수는 외부 링크를 가지며, 프로그램의 아무 파일에 있는 동일한 외부 변수의 선언과 동일시된다. 다시 말하면 일치되는 외부 변수 선언을 포함하는 동일한 프로그램의 파일들은 실제로 동일한 변수를 공유한다. 전역 변수는 그 선언이 `static`이나 `extern`라고 명시되지 않으면 외부 링크를 가진다.
- 객체는 내부 링크나 외부 링크 모두로 선언되면 프로그램의 동작은 정의되지 않는다.
- 외부적으로 링크된 객체(변수나 함수)는 프로그램에서 정확히 한 파일에 정의되어야 한다. 변수는 초기 값을 가질 때 정의되며 `extern` 키워드 없이 전역 수준에서 선언된다. 함수는 몸체(코드)가 주어질 때 정의된다.

C89 이전의 많은 C 구현은 0~1개의 외부 변수 선언을 허용하기 위해 마지막 규칙을 완화했으며, 일부 구현에서는 둘 이상의 선언을 허용했다. 이러한 구현들에서 링커(분리 컴파일된 파일들을 하나의 프로그램으로 결합하는 프로그램)는 다중 정의를 통합하고 프로그램이 선언만을 포함하는 모든 변수(또는 링크된 변수들의 집합)에 대해 묵시적 정의를 생성한다.

C89의 "링크" 규칙은 한 파일의 이름을 다른 파일의 이름과 연결하는 방법을 제공한

다. 이 규칙은 구현의 관점에서 보면 가장 쉽게 이해할 수 있다. 대부분의 언어 독립적 링커는 기호(기계어 프로그램에서의 위치에 대한 문자열 이름)를 처리하게 설계된다. 링커의 작업은 모든 기호를 최종 프로그램의 위치에 할당하고 기호를 참조하는 모든 기계어 명령어에 해당 기호의 주소를 포함시키는 것이다. 이를 수행하기 위해서 링커는 어느 기호가 다른 파일들에 있는 바인딩되지 않은 참조를 결정하는 데 사용될 수 있는지와 어느 기호가 주어진 파일에 지역적인지를 알아야 한다. C89 규칙은 이 정보를 제공하기에 충분하다. 프로그래머의 관점에서 보면 이 규칙에는 **인터페이스**의 공식적인 개념이 없으며, 어떤 이름을 모든 파일이 아닌 일부 파일에서만 볼 수 있게 만들 수 없다. 더욱이 어떤 것도 다른 파일에서 찾은 외부 객체의 선언이 호환 가능하다고 보장해주지 않는다. 예를 들어 하나의 외부 변수를 한 파일에서는 다중 항목 레코드로 선언하고 다른 파일에서는 부동점 실수로 정의하는 데에는 전혀 문제가 없다. 컴파일러는 이러한 오류를 잡을 필요가 없으며 이로 인한 오류는 매우 찾기 어려울 수 있다.

## 헤더 파일

다행히도 C 프로그래머들은 실질적으로 오류를 최소화할 수 있는 외부 선언의 사용에 대한 관례를 개발했다. 이러한 관례는 매크로 전처리기의 파일 포함 도구에 의존한다. 프로그래머는 대략 인터페이스와 모듈 구현에 대응되는 한 쌍의 파일을 생성한다. 인터페이스의 이름은 `.h`로 끝나며 이와 대응되는 구현 파일은 `.c`로 끝난다. `.c` 파일에서 정의된 모든 객체는 `.h` 파일에서 선언된다. 컴파일러는 `.c` 파일의 첫 부분을 특수한 형태의 주석으로 처리하지만 프로그래머는 전처리기가 이 파일과 대응되는 `.h` 파일의 축어적 복사본을 포함하게 하는 지시어를 삽입한다. 이러한 포함 연산은 모든 모듈에 있는 객체의 "전방" 선언을 구현 파일의 시작 부분에 위치시키는 효과를 가진다. 파일에서 나중에 나오는 정의와 모순되면 컴파일러는 오류 메시지를 생성한다. 프로그래머는 각 `.c` 파일의 상단에 있는 전처리기가 `.c` 파일이 의존하는 모든 모듈에 대한 `.h` 파일의 복사본을 포함하게 지시할 수도 있다. 어떤 `.h`의 모듈을 사용하는 모든 `.c` 파일에서 전처리기가 주어진 `.h` 파일의 동일한 복사본을 포함하는 한 모순되는 선언은 절대 일어나지 않는다. 불행히도 `.h` 파일이 변경되었을 때 하나 이상의 `.c` 파일에 대해 재컴파일을 잊어버리기 쉬우며, 이는 매우 잡기 어려운 오류를 야기할 수 있다. 유닉스의 `make` 유틸리티와 같은 도구는 모듈 간의 의존성을 유지함으로써 이러한 오류를 최소화하는 데 도움을 준다.

## 네임스페이스

헤더 파일의 관례에도 불구하고 C89에서는 개별 파일 수준 이상으로 유효 범위를 지정하지 못하는 것이 여전히 문제로 남아있다. 특히 모든 전역 이름은 프로그램의 모든 파일과 이 프로그램이 링크하는 모든 라이브러리에서 유일해야 한다. 일부 코딩 표준

에서는 프로그래머가 외부 객체의 이름에 모듈의 이름을 포함시키게 장려하지만(예를 들어 scanner_nextSym) 이는 어색할 수 있으며 보편적인 것과는 거리가 멀다.

**예 3.38**

C++의 네임스페이스

이러한 제약을 해결하기 위해 C++는 클래스와 함수에 이미 제공되는 유효 범위 지정을 일반화하고 모듈과 컴파일 단위 간의 관계를 제거하며 .h 파일의 인터페이스 관례를 강화하는 namespace 기법을 도입했다. namespace 내부에서는 모든 종류의 이름 모음을 선언할 수 있다.

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

foo에서 선언된 객체의 실제 정의는 아무 파일에나 나올 수 있다.

```
class foo::foo_type_1 { ...          // 완전한 정의
```

원하는 경우 다른 네임스페이스에서 선언된 객체의 정의가 동일한 파일에 나올 수 있다.

**예 3.39**

다른 네임스페이스의 이름 사용하기

C++ 프로그래머는 **완전명**을 사용하거나 명시적으로 들여옴(using)으로써 네임스페이스의 객체에 접근할 수 있다.

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

또는

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

또는

```
using namespace foo;        // foo의 모든 것을 들여오기
...
foo_type_1 my_first_obj;
```

내보내기의 개념은 없다. 네임스페이스에서 외부 링크를 가지는 모든 객체는 들여오기를 수행하면 어딘가 다른 곳에서 볼 수 있다. 분리 컴파일의 기초는 여전히 링크라는 것에 주의하자. .h 파일은 단순한 관례다.

## 【3.7.2】 패키지와 자동 헤더 추론

**예 3.40**

**자바의 패키지**

자바와 C#의 분리 컴파일 도구는 .h 파일을 사용하지 않는다. 명확히 말해서 자바는 패키지라고 하는 모듈의 공식적 개념을 도입한다. 파일이나 (어떤 구현에서는) 데이터베이스의 레코드일 수 있는 모든 컴파일 단위는 정확히 하나의 패키지에 속하지만 패키지는 여러 컴파일 단위로 구성될 수 있으며, 각 컴파일 단위는 자신이 속한 패키지를 나타내는 것으로 시작한다.

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

명시적으로 public으로 선언되지 않는 한 자바의 클래스는 동일한 패키지에 속한 모든 컴파일 단위에서만 볼 수 있다.

**예 3.41**

**다른 패키지의 이름 사용하기**

C++에서와 같이 다른 패키지의 클래스를 사용할 필요가 있는 컴파일 단위는 완전명의 사용이나 "한번에-이름 들여오기"나 "한번에-패키지 들여오기"를 통해 접근할 수 있다.

```
foo.foo_type_1 my_first_obj;
```

또는

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

또는

```
import foo.*;              // foo의 모든 것을 들여오기
...
foo_type_1 my_first_obj;
```

자바 컴파일러는 패키지 M에서 이름을 들여오게 요청 받으면 표준(그러나 구현에 따라 다른) 위치 집합에서 M을 검색하고 적절한 경우(즉, 소스 코드만 찾은 경우나 목표 코드가 무효한 경우) 이를 재컴파일한다. 그 다음 컴파일러는 C++의 .h 파일이나 에이다나 모듈라 3 헤더에서 필요했던 정보를 자동으로 추출한다. M의 컴파일에 다른 패키지가 필요하다면 컴파일러는 물론 이런 패키지도 재귀적으로 검색한다.

C#도 자바와 같이 완전한 클래스 정의에서 자동으로 헤더 정보를 추출한다. 그러나 C#의 모듈 수준 구문은 한 파일이 여러 네임스페이스의 조각들을 포함할 수 있게 해주는 C++의 네임스페이스에 기반한다. C#에는 표준 검색 경로의 개념도 없다. 완전한 프로그램을 생성하기 위해 프로그래머는 컴파일러에 필요한 모든 파일의 완전한 목록을 제공해야 한다.

조기 헤더 파일 구성이라는 스프트웨어 공학 관례를 모방하기 위해 자바나 C# 설계팀은 패키지나 네임스페이스의 (공용(public) 클래스의) 골격 버전을 생성할 수 있다. 이러한 패키지나 네임스페이스는 추후에 완전한 버전을 담당하는 프로그래머가 병행적이고 독립적으로 사용할 수 있다.

**설계와 구현**

### 분리 컴파일

초기 C와 포트란에서부터 C++, 모듈라 3, 에이다, 끝으로 자바와 C#까지 이어지는 분리 컴파일 기법의 발전은 구현 중심적 관점에서 좀 더 프로그래머 중심적 관점으로의 이동을 반영한다. 흥미롭게도 일부의 초기 C 구현에서 외부적으로 링크된 변수의 정의를 가지지 않아도 되는 것은 포트란에서 따온 것이다. 정의 없는 선언과 대응되는 어셈블리어 연상어는 .common(common 블록의 경우)이다(그리고 3.3.1절에서 살펴봤듯이 common 블록에 대한 유형 검사의 부재는 본래 오류가 아니라 기능으로 간주되었다!).

## 【3.7.3】 모듈 계층

**예 3.42**

**다중 부분 패키지 이름**

모듈라와 에이다 프로그래머는 어휘적 중첩를 써서 단일 컴파일 단위 안에 모듈 계층을 생성할 수 있다(예를 들어 모듈 C는 모듈 A 안에서 선언된 모듈 B 안에서 선언될 수 있다). 이와 비슷한 맥락에서 에이다 95, 자바, C# 프로그래머는 다중 부분 이름을 사용해서 별도로 컴파일된 모듈들의 계층을 생성할 수 있다.

```
package A.B is ...          -- 에이다 95
package A.B; ...            // 자바
namespace A.B { ...         // C#
```

이 예에서 패키지 A.B를 패키지 A의 자식이라고 한다. 에이다 95와 C#에서 자식 패키지는 부모의 내부에 중첩된 것처럼 동작하므로 자동으로 부모의 모든 이름을 볼 수 있게 된다. 이와 달리 자바에서는 다중 부분 이름이 단지 관례며, 패키지 A와 A.B 사이에는 특별한 어떤 관계도 없다. A.B가 A에 있는 이름을 참조한다면 A는 이러한 이름을 공용으로 선언해야 하며 A.B는 이를 명시적으로 들여와야 한다. 에이다 95의 자식 패키지는 유형으로서의 모듈 방식이 아니라 감독자로서의 모듈 방식의 추상화를 지원한다는 점만 제외하면 C++의 유도된 클래스와 유사하다. 에이다 95의 자식 패키지 관련 내용은 9.2.3절에서 좀 더 자세히 다룬다.

## ✓ 확인문제

44. 분리 컴파일의 목적은 무엇인가?

45. C에서 외부 변수가 링크된다는 것은 어떤 의미인가?

46. `.h`와 `.c` 파일의 사용에 대한 C의 관례를 요약하라.

47. 컴파일 단위와 C++이나 C#의 네임스페이스 사이의 차이점을 설명하라.

48. 에이다와 유사 언어가 모듈 헤더를 몸체와 분리시키는 이유는 무엇인가? 자바와 C#는 왜 헤더와 몸체를 분리하지 않는가?