4장. 파일 타입과 컴파일 도구

4장에서는 빌드 시스템에서 등장하는 파일 포맷을 간단히 다룬다. 큰 그림에서 보자면 4장은 소스 트리의 각 파일을 오브젝트 트리의 파일로 변환하는 컴파일 도구에 초점을 맞춘다. 오브젝트 파일을 또 다른 오브젝트 파일로 변환하게 확장할 수도 있으며 이때의 두 파일은 모두 오브젝트 트리에 저장된다.

> 💡 이번 장은 다른 언어에 관한 이야기도 존재한다. 나의 목표는 C와 C++이 목표기 때문에 그 부분은 참고정도로 하고 읽어보는 정도로 진행했고 전체적인 이야기 말고 중요하다고 생각되는 부분만 목표!

빌드 도구와 컴파일 도구의 차이점을 상기해보자. 4장에서는 하나 이상의 입력 파일을 변환해 그에 상응하는 출력 파일을 만드는 컴파일 도구에만 초점을 맞춘다. 반면 Make나 Art 같은 빌드 도구는, 어떤 파일을 컴파일 해야하고 어떤 컴파일 도구를 사용해야 할지의 결정과 같이 좀 더 상위 수준에서 빌드 프로세스를 조율할 책임이 있다.

오브젝트 파일

오브젝트 파일은 기계어 코드 명령을 담는 컨테이너다. 이 파일은 다른 오브젝트 파일과 모든 필요 라이브러리와 함께 링크돼야 하므로 아직은 컴퓨터가 이를 실행할 수 없다. 앞에서 살펴본 내용과 같이 소스 파일을 오브젝트 코드로 컴파일하기 위해서는 gcc 옵션에 -c를 사용해야 한다.

```
$ gcc -c hello.c
```

오브젝트 파일은 사람이 읽을 수는 없지만 유닉스의 file 명령을 사용해 파일 내용의 상위 수준 정보를 얻을 수 있다. 이 경우에 file은 앞서 만들었던 파일의 타입이 다음과 같음을 확인해준다.

- 오브젝트 파일은 다양한 프로그램 컴포넌트를 저장하기 위해 실행 및 링크 가능 포맷 구조를 사용한다. ELF는 a.out과 COFF 같은 이전 포맷을 대체하는 일반적인 오브젝트 파일 포맷이다.
- 파일은 새로운 64비트 프로그램이거나 과거의 16비트 프로그램이 아닌 32비트 프로그램이다.
- 데이터는 빅 에디언이 아닌 리틀엔디언으로 저장된다.
- 머신 명령은 인텔 x86 계열의 프로세서만을 지원하고, MIPS와 파워 PC를 비롯한 그 밖의 많은 CPU타입은 지원하지 않는다.
- 파일은 여전히 재배치 가능하며 아직은 최적화되지 않았다. 이는 파일이 다른 오브젝트 파일이나 라이브러리와 함께 링크하는데 필요한 정보를 포함하고 있지만, 메모리에 로드 돼 실행되기에 충분한 정보를 갖고 있지 않음을 의미한다.

빌드 시스템이 단 하나의 CPU타입을 지원하는 프로그램을 만든다면 이는 단지 흥미로운 정보일 뿐이다. 하지만 여러 CPU 타입의 시스템을 지원하게 컴파일하려면 모든 파일 타입의 세부 사항이 올바른지 확인하는 데 많은 관심을 기울여야 한다. 빌드 시스템이 예기치 않게 서로 다른 타입의 오브젝트 파일을 함께 섞어버린다면 파일이 서로 링크될 수 없는 혼란스러운 에러에 수없이 마주해야 한다. 심지어 CPU계열이 같은 32비트와 64비트 오브젝트 파일을 서로 섞으려는 시도조차 불분명한 컴파일 에러의 원인이 될 수 있다. 오브젝트 파일을 살펴보는 또 다른 방법으로, 파일에서 정의하거나 파일이 필요로 하는 심볼은 무엇인지 확인하는 방식이 있다. 이는 C 소스 파일에서 어떤 함수와 변수를 정의하는지 확인하거나, 해당 파일이 사용하지만 다른 파일에 정의된 함수와 변수는 무엇인지 확인하는 방법이다. 함수 프로토타입을 정의하기 위해 어떻게 헤더 파일을 사용하고 있음을 확인할 수 있다.

유닉스의 nm 명령은 필요한 함수가 실행 프로그램에 링크되지 않았을 때 발생하는 컴파일 에러, 즉 정의되지 않은 심볼해결에 매우 유용하다. 모든 오브젝트 파일에 nm 명령을 실행하면 누락된 심볼을 참조하는 위치와 해당 심볼을 정의한 위치를 확인 할 수 있다.

고급 사용자라면 유닉스 `objdump` 를 알아보기를 바란다. → 충분한 가치 존재

정적 라이브러리

리눅스 운영체제는 정적 링크 라이브러리(statically linked library)와 동적 링크 라이브러리(dynamically linked library)를 모두 지원한다. 동적 라이브러리는 런타임 시에 로드되고 필요한 함수를 프로그램이 직접 호출하는 반면, 정적 라이브러리는 오브젝트 파일을 하나의 실행 프로그램으로 링크하는 오브젝트 파일의 아카이브일뿐임을 상기하자.

동적 라이브러리

동적 링크 라이브러리의 처리 과정은 더욱 복잡하다. 특히 링크 작업이 프로그램을 컴파일할 때가 아니라 런타임에 일어난다는 사실을 이해해야한다. 따라서 컴파일과정에서의 변경이 필요하다.

이제는 위치 독립코드(Position-independent code)를 사용해 모든 오브젝트 파일을 생성해야하며, 이를 통해 프로그램이 요청하는 어떤 메모리 위치든 오브젝트 파일을 로드할 수 있다. 공유 라이브러리를 마치 독립 실행 프로그램인 것처럼 생성할 수 있으며, 단지 공유 라이브러리를 동적 라이브러리로 만들기 위해 -shared 옵션을 사용한다는 점이 다르다.

새롭게 생성한 공유 라이브러리를 사용하기 위해서는 표준 GCC 링커의 실행라인에 해당 라이브러리의 이름을 지정해야 한다. 예제의 -l 옵션은 링커에게 libmymath.so 라이브러리를 포함하게 요청한다. -L 옵션은 해당 라이브러리를 찾을 수 있는 위치 를 링커에게 알려주기 위해 사용한다

```
$ gcc -c main.c
$ gcc -o prog main.c -L. -l mymath
```

모든 것이 올바르게 동작함을 확인하기 위해 메인 프로그램을 실행할 때 어떤 동적 라이브러리가 메모리에 로드되는지 보여주는 유닉스 `ldd` 명령을 사용한다!
→ 추후에 공부 필요!