07장 자료 유형

7.9 파일과 입력/출력

아래 두 하위 절을 각기 상호작용적 I/O와 파일 기반의 I/O에 할당한다. 그러고 나서 7.9.3절은 텍스트 파일의 자주 나오는 특별한 경우를 다룬다.

【7.9.1】 상호작용적 I/O

현대 기계에서 일반적으로 상호작용적 I/O는 창, 메뉴, 스크롤바, 버튼, 슬라이더 등을 지원하는 마우스, 키보드와 비트로 매핑된 스크린을 사용하는 그래픽 사용자 인터페이스(GUI, "구이") 시스템을 통해 발생한다. GUI의 특성은 마이크로소프트, 매킨토시와 유닉스의 X11마다 확연히 다르다. 이 차이는 응용프로그램의 플랫폼 간의 이식을 어렵게 하는 근본적 이유 중 하나다.

단일 플랫폼 내에서 GUI 시스템의 편리성은 일반적으로 (윈도우를 생성하고 텍스트를 출력하고 다각형을 그리기 위한) 라이브러리 루틴의 형태를 갖는다. 입력 이벤트(마우스를 움직임, 버튼을 누름, 키 누름)를 프로그램에서 접근 가능하거나 이벤트가 발생했을 때 운영체제가 호출하는 **이벤트 처리기** 서브루틴에 묶인 큐에 위치시키게 한다. 외부에서 처리기를 시작하게 하기 때문에 그 행동을 메인 프로그램의 행동과 **동기화**해야 한다. 12.3절에서 동기화에 대해 더 논해본다.

일부 프로그래밍 언어, 특히 스몰토크와 자바는 GUI 메커니즘의 표준 집합을 언어 내로 통합하려 한다. 스몰토크의 설계 조직은 1970년대 초에 마우스와 윈도우 기반의 인터페이스를 발명했던 제록스의 팔로 알토 연구 센터(PARC)에 있던 최초의 그룹 중 일부다. 불행히도 스몰토크의 GUI가 언어의 영역 내로 한정시키는 데는 성공했지만

그것이 실행되는 호스트 시스템의 "룩 앤 필(look-and-feel)"과 잘 통합하지 않는 경향이 있었다. 비슷한 예로 자바의 원래 GUI 편의 기능(Abstract Window Toolkit (AWT))은 최소 공배수 중 일부를 보여주게 했다. 스몰토크의 GUI는 언어의 기초적인 부분이다. 자바의 것은 라이브러리의 표준 집합 형태를 띤다. 자바 루틴과 인터페이스는 시간상에 있어서 상당히 향상됐었다. 좀 더 최신의 "Swing" 라이브러리는 장착형 룩 앤 필을 가지고 있어서 그것을 다양한 운영체제와 좀 더 쉽게 통합하게 (그리고 좀 더 쉽게 이식하게) 한다.

프로그램과 상호작용적 시스템에 특징을 부여하는 인간 사용자의 "병렬 실행"은 함수형 언어에서 획득하기 어렵다. (파일로부터의 입력을 얻고 출력을 파일에 쓰는) 일괄 방식으로 동작하는 함수형 프로그램을 입력에서 출력으로의 함수로 모델화할 수 있다. 하지만 사용자와 상호작용하는 프로그램은 프로그램 순서 정하기의 매우 구체적인 개념을 필요로 한다. 이후의 입력이 이전 입력에 의존할 수도 있기 때문이다. 입력과 출력 모두가 토큰의 정렬된 순서의 형태라면 6.6.절의 "무한" 리스트의 상기시키는 방법으로 게으른 자료 구조를 사용해 상호작용적 I/O를 모델화할 수 있다. 함수형 프로그램에서 좀 더 일반적인 (GUI 기반의) I/O를 다루기 위한 방법은 미해결 문제로 남아있다. 10.4절과 10.7절에서 다시 이런 문제를 다룰 것이다.

【7.9.2】 파일 기반의 I/O

지속적인 파일은 다른 시간대에 실행되는 프로그램이 서로와 통신하는 주요 방법이다. 일부 언어 제안(예를 들어 아르고스(Argus)[LS83]와 χ[SH92])은 보통의 변수가 프로그램의 한 호출로부터 다음 호출까지 지속하게 한다. 그리고 일부 실험적인 운영체제(예를 들어 오팔[CLFL94]과 헴록[GSB+93])은 언어 외부의 변수에 대한 지속성을 제공한다. 게다가 스몰토크와 커먼 리스프에 대한 것과 같은 언어 한정적인 프로그래밍 환경은 지속적인 이름이 붙은 변수를 포함하는 작업 공간의 개념을 제공한다. 하지만 이런 예는 규칙뿐인 예외다. 특정 프로그램 실행보다 오래 살아남을 필요가 있는 자료는 파일에 저장할 필요가 있다.

상호작용적 I/O와 마찬가지로 파일을 직접적으로 언어에 통합시키거나 라이브러리 루틴을 통해 제공한다. 후자의 경우에서 플랫폼 간 프로그램의 이식성을 향상시키기 위해 언어 설계자가 표준 라이브러리 인터페이스를 제안하는 것은 여전히 좋은 생각이다. 알골 60에서 이런 표준의 부재는 언어의 광범위한 사용을 방해했다. I/O를 언어 속으로 통합시키는 주요 원인 중 하나는 특별한 구문의 사용이다. 일부 언어, 특히 포트란과 파스칼은 가변 개수의 매개변수를 취하는 유형적으로 안전한 "서브루틴"을 얻기 위해 고유 I/O 편의 기능을 제공한다. 그 매개변수 중 일부는 선택적일 수도 있다.

프로그래머의 필요와 호스트 운영체제의 성능에 따라 파일 내의 자료를 메모리상에

있는 것처럼 이진 형태로나 텍스트로 표현할 수 있다. 이진 파일에서 32비트 값 10000101010_2는 숫자 1066_{10}을 나타낸다. 텍스트 파일에서는 ASCII 문자열 "1066"으로 표현할 것이다. 속도와 편리함을 목적으로 임시 파일을 보통 이진 형태로 유지한다. 지속적인 파일은 일반적으로 두 형태 모두로 저장한다. 텍스트 파일을 시스템 간 좀 더 쉽게 이식할 수 있다. 워드의 크기, 바이트의 순서, 정렬, 부동 소수점의 형식 등의 문제가 발생하지 않기 때문이다. 텍스트 파일도 사람의 가독성에 이점이 있다. 그래서 텍스트 편집기와 관련된 도구로 조작할 수 있다. 불행히도 텍스트 파일은 크기가 큰 경향이 있다. 특히 숫자 자료를 저장하기 위해 사용할 때 그렇다. 배정도 부동소수점 숫자는 이진 형태로 8바이트만을 차지한다. 하지만 10진 표기법으로 (주위 공백 문자를 세지 않는다면) 24개의 문자만큼 필요로 한다. 또한 텍스트 파일은 출력할 때 이진에서 텍스트로의 변환하고 입력할 때 텍스트에서 이진으로 변환하는 비용이 발생한다. 자료 압축을 사용해 적어도 기록 저장 공간에 대해 크기 문제를 다룰 수 있다. 텍스트/이진 변환을 제어하기 위한 방법은 I/O의 가장 복잡한 부분인 경향이 있다. 이에 대해서는 따라오는 하위 절에서 다룬다.

예 7.129
고유형으로서 파일

I/O가 언어에 내장돼 있을 때 파스칼에서처럼 보통 고유형 생성자를 사용해 파일을 선언한다.

```
var my_file : file of foo;
```

예 7.130
open 연산

라이브러리 루틴이 I/O를 제공하면 라이브러리는 보통 파일을 표현하기 위해 불투명한 유형을 제공한다. 각 경우에 각 파일 변수는 open 연산을 이용한 외부의 운영체제가 지원하는 파일에 한정된다. 예를 들면 C에서 다음처럼 작성할 수 있다.

```
my_file = fopen(path_name, mode);
```

fopen의 첫 번째 인자는 호스트 운영체제의 이름 규정을 사용한 파일의 이름이다. 두 번째 인자는 파일을 읽을 수 있거나 쓸 수 있거나 아니면 둘 다 해야 하는지, 아직 존재하지 않으면 파일을 생성해야 하는지, 그리고 파일이 이미 존재한다면 덮어쓸 수 있는지나 덧붙일지를 지시하는 문자열이다.

예 7.131
close 연산

프로그램이 파일 사용을 종료할 때 그것은 close 연산을 사용해 파일 변수와 외부 객체 간의 관계를 깰 수 있다.

```
fclose(my_file);
```

close의 호출에 대한 응답으로 운영체제는 잠겨 있는 파일을 푸는 것(그래서 다른 프로그램이 그것을 사용해도 되게 함), 테이프 드라이브를 감거나 버퍼 내용을 디스크로 출력하게 강제하는 것과 같은 특정 "종결" 연산을 수행할 수도 있다.

이진과 텍스트 파일 대부분을 문자, 워드나 레코드의 선형 나열로 저장한다. 열린 파일은 모두 그 나열의 어떤 원소로의 묵시적인 참조인 현재 위치의 개념을 가지고 있다. 각 read나 write 연산은 이 참조를 묵시적으로 한 위치만큼 전진시킨다. 결국 연속

적인 연산은 연속적인 원소를 자동적으로 접근한다. 순차 파일에서 이 자동적인 전진은 현재 위치를 변경시키는 유일한 방법이다. 순차 파일은 보통 프린터와 테이프와 같은 미디어와 대응된다. 그것에서 현재 위치는 변경하기 어려운 물리적 표현(얼마나 많은 페이지를 출력할지, 각 스풀당 얼마나 테이프를 소비하는지)을 가지고 있다.

임의 접근 파일에서 프로그래머는 `seek` 연산을 호출해 현재 위치를 임의의 값으로 변경할 수 있다. 일부 프로그래밍 언어(예를 들어 코볼과 PL/I)에서 임의 접근 파일(직접 파일이라고도 함)은 현재 위치의 개념을 가지고 있지 않다. 오히려 그것을 어떤 키에 인덱싱한다. 그리고 매 `read`나 `write` 연산은 키를 지정해야 한다. 순차적이고 키 모두로 접근할 수 있는 파일을 **인덱싱된 순차**라고 한다.

임의 접근 파일은 보통 자기디스크나 광학디스크와 같은 미디어에 대응된다. 여기서 현재 위치를 상대적으로 쉽게 변경할 수 있다(기술에 따라 현대의 디스크는 어디에서나 새로운 장소의 자리를 찾는 것이 5에서 200ms 정도 걸린다. 하지만 테이프 드라이브는 1분 이상 걸릴 수 있다. 5ms, 즉 2GHz 프로세서상에서 1,000만 주기가 여전히 긴 시간이어서 자리 찾기(seeking)를 경솔히 실행시키면 안 됨을 기억하자). 개인의 구현이 비표준 언어 확장으로 임의 접근을 지원하게 한다 하더라도 일부 언어, 특히 파스칼은 임의 접근 파일을 제공하지 않는다.

【7.9.3】 텍스트 I/O

텍스트 파일은 행의 나열로 구성되고 각 행이 문자로 구성되어 있는 것으로 생각하는 것이 일반적이다. 좀 더 오래된 시스템, 특히 펀치 카드의 은유에 입각해 설계된 것에서는 파일 자체의 조직에 행을 반영한다. 예를 들면 `seek` 연산은 행 숫자를 인자로 취할 수도 있다. 좀 더 흔히 텍스트 파일은 단순히 문자의 나열이다. 이 나열 안에 (출력되지 않는) 제어 문자는 행 사이의 경계를 의미한다. 불행히도 행 종료 규정은 표준화가 되어 있지 않다. 유닉스에서는 텍스트 파일의 각 행은 새 줄("control-J") 문자, 아스키 값 10으로 끝난다. 매킨토시에서 각 행은 캐리지 리턴("control-M") 문자, 아스키 값 13으로 끝난다. MS-DOS와 윈도우에서 각 행은 캐리지 리턴/라인피드 쌍으로 끝난다. 텍스트 파일은 보통 순차적이다.

행 중단에 대한 혼란스러운 규정에도 불구하고 텍스트 파일은 이진 파일보다 훨씬 더 이식성 있고 가독성이 있다.[1] 이진 파일이 내부 자료의 구조를 반영하지 않기 때문에 텍스트 파일은 입력과 출력에 확장된 변환을 필요로 한다. 고려해야 할 문제는 정수 값에 대한 기저(그리고 10진수의 표현), 부동소수점 값의 표현(숫자의 개수, 소수점의 위치, 지수에 대한 표기법), 열거형과 기타 수가 아닌 문자열이 아닌 유형, 그리고 있다면 열(오

주1. 물론 여기서 파일을 평범한 아스키 혹은 유니코드 파일에 대해서 말하고 있다. 특정 폰트, 크기와 색상으로 구성된 형식화된 텍스트 소위 "풍부한 텍스트" 파일은 완전히 다른 문제일 것이다. 워드 프로세서는 일반적으로 이진과 아스키 자료의 조합을 가지고 풍부한 텍스트를 표현한다. 그러나 이식성을 향상시키기 위해 아스키 유일 표준을 사용할 수 있다.

른과 왼쪽 자리맞춤, 0 혹은 공백 채우기, 코볼의 "부동" 달러 기호) 내의 위치 잡기를 포함한다. 하드웨어는 이 문제 중 일부(예를 들어 부동소수점 숫자에 자리의 개수)에 영향을 준다. 하지만 애플리케이션의 요구와 프로그래머의 선호는 대부분을 지정한다.

대부분의 언어에서 프로그래머는 각 문자만 읽고 쓰기 위해 언어나 라이브러리 방법을 사용해 그것 모두를 명시적으로 작성함으로써 입력과 출력 형식화의 완벽한 제어를 취할 수 있다. 하지만 이런 저수준에서의 I/O는 장황하고 대부분의 언어는 좀 더 고수준의 연산도 제공한다. 이 연산은 구문과 프로그래머가 I/O 형식을 지정하게 하는 정도에 따라 완전히 다르다. 포트란, 에이다, C와 C++, 이렇게 4가지 명령형 언어의 예를 이용해 가능성의 폭을 설명한다.

포트란의 텍스트 I/O

예 7.132
포트란에서의 형식화된 출력

포트란에서 문자열, 정수와 10개의 부동소수점 숫자를 가진 배열을 다음과 같이 작성할 수 있다.

```
character s*20
integer n
real r (10)
...
write (4, '(A20, I10, 10F8.2)'), s, n, r
```

write문에서 4는 특정 출력 파일을 식별하는 **단위수**를 의미한다. 작은따옴표로 둘러싸이고 괄호로 둘러싸인 수식을 **형식**이라고 한다. 이것은 출력될 변수를 표현하는 방법을 지정한다. 이 경우 20열 크기의 아스키 문자열, 10열 크기의 정수와 8열 크기(값의 소수 부분을 위해 각각 2열 크기를 확보해놓는)의 부동소수점 숫자 10개를 요청한다. 포트란은 형식의 내부에서의 사용을 위한 이 편집 디스크립터의 매우 풍부한 집합을 제공한다. 코볼, PL/I와 펄은 매우 다른 구문을 가지고 있지만 이에 필적하는 편의 기능을 제공한다.

예 7.133
레이블을 붙인 형식

포트란은 어떤 형식을 간접적으로 지정하게 한다. 그래서 2개 이상의 입력문이나 출력문에서 사용할 수도 있다.

```
write (4, 100), s, n, r
...
100 format (A20, I10, 10F8.2)
```

또한 형식을 실행 시간에 만들고 문자열에 저장하게 한다.

```
character(len=20) :: fmt
...
fmt = "(A20, I10, 10F8.2)"
```

```
...
write (4, fmt), s, n, r
```

프로그래머가 항목에 대한 열의 정확한 할당에 대해 알지 못하거나 신경 쓰지 않는 경우 형식을 생략할 수 있다.

```
write (4, *), s, n, r
```

이 경우에 실행 시간 시스템은 기본 형식 규정을 사용할 것이다.

예 7.134

표준 출력에 출력하기

표준 출력 스트림(즉, 터미널이나 대행자)에 쓰기 위해 프로그래머는 print문을 사용할 수 있다. 이는 단위수가 없는 write와 닮았다.

```
print*, s, n, r              ! *는 기본 형식을 의미
```

입력에 대해 표준 입력과 특정 파일에 대해 read를 사용한다. 전자의 경우에 추가적인 괄호의 집합과 함께 단위수를 생략한다.

```
read 100, s, n, r
...
read*, s, n, r           ! *는 기본 형식을 의미
```

포트란 90에서는 별(*)을 생략할 수 있을 것이다.

read, write와 print의 완전한 형태에서 단위수와 형식과 함께 괄호로 둘러싼 리스트에서 추가적인 인자를 제공할지도 모른다. 파일 종료에 뛰어드는 레이블, 다른 오류에 뛰어드는 레이블, 운영체제가 반환하는 상태 코드를 위치시키기 위한 변수, 출력 값에 덧붙이는 레이블의 집합("이름 목록"), 그리고 일반적인 자동적 파일의 다음 행으로의 전진을 덮어쓰기 위한 제어 코드를 포함한 다양한 부가적인 정보를 지정하기 위해 이를 사용할 수 있다. 선택적인 인자의 대다수가 있고 대부분을 거의 생략하기 때문에 일반적으로 8.3.3절에서 설명할 이름이 붙은(키워드) 매개변수 표기법을 사용해 지정한다.

read, write와 print의 약어 버전의 다양성은 그것이 가변 개수의 프로그램 변수에 연산한다는 사실과 함께 그것을 "보통의" 서브루틴으로 변환하는 것을 어렵게 한다. 포트란 90은 기본과 키워드 매개변수(8.3.3절)를 제공하지만 포트란 77은 그렇지 않다. 그리고 포트란 90에서도 임의의 수의 매개변수를 가진 서브루틴을 정의하는 방법이 없다.

파스칼에서는 포트란77처럼 모든 서브루틴이 가진 매개변수의 개수와 유형을 고정한다. 그러므로 파스칼의 read, readln, write와 writeln "루틴"을 언어에 설치한다. 라이브러리의 일부가 아니다. 각기 가변 개수의 인자를 취한다. 그 중 첫 번째 인자는 선택적으로 특정 파일을 지정할 수도 있다. 불행히도 파스칼의 형식 지정 방법은 포트란보다 훨씬 덜 유연하다. 가끔 프로그래머가 개별 문자의 입력과 출력에 대해서만

read와 write를 사용해 형식 지정을 직접 구현하게 한다. 모듈라 3의 설계에서 니클라우스 워드는 I/O를 언어 밖으로 옮겨서 표준 라이브러리에 내장시키는 선택을 했다. 모듈라 2의 I/O 라이브러리는 상대적으로 원시적이다. 파스칼에서의 문자 하나하나의 I/O에 대한 적절한 향상만 있다. 에이다 라이브러리는 훨씬 더 비싸고 다중 정의와 기본 매개변수를 많이 사용한다.

에이다의 텍스트 I/O

에이다는 I/O에 대한 5개의 표준 라이브러리를 제공한다. sequential_IO와 direct_IO 패키지는 이진 파일을 위한 것이다. 원하는 원소 유형에 대해 인스턴스되는 제네릭 파일 유형을 제공한다. IO_exceptions와 low_level_IO 패키지는 각기 오류 조건과 장치 제어를 다룬다. text_IO 패키지는 문자의 순차 파일의 형식화된 입력과 출력을 제공한다.

예 7.135
에이다의 형식화된 출력

text_IO를 사용해 원래 3개의 변수를 가진 포트란 출력문은 에이다에서 다음과 같이 작성할 수 있다.

```
s : array (1..20) of character;
n : integer;
r : array (1..10) of real;
...
set_output(my_file);
put(n, 10);
put(s);
for i in 1..10 loop put(r(i), 5, 2); end loop;
new_line;
```

(루프 내에서) r의 원소의 put에서 두 번째 매개변수는 포트란에서처럼 (소수점을 포함한) 전체 수의 폭이 아닌 소수점 이전의 자리수를 지정한다. s의 put은 문자열의 자연 길이를 사용한다. 다른 길이를 원한다면 프로그래머는 공백을 작성해야 하거나 부분 문자열을 명시적으로 put해야 할 것이다. 정수와 실수에 대해 정확한 출력 위치 지정을 원하지 않는다면 put 호출에서 추가적인 매개변수를 생략할 수 있다. 이 경우 실행 시간 시스템은 표준 기본값을 사용한다. 프로그래머는 원할 때 이 기본값을 변경하기 위해 추가적인 라이브러리 루틴을 사용할 수 있다. set_output의 호출은 비슷한 방법을 실행한다. 그것은 출력 파일의 기본 개념을 변경한다.

모든 고유형에 대해 2가지 다중 정의된 형태가 있다. 하나는 첫 인자로 파일 이름을 취한다. 그리고 나머지는 아니다. 위의 마지막 5행을 다음과 같이 작성할 수 있다.

예 7.136
다중 정의된 put 루틴

```
put(my_file, n, 10);
put(my_file, s);
```

```
for i in 1..10 loop put(my_file, r(i), 5, 2); end loop;
new_line(my_file);
```

물론 프로그래머는 임의의 사용자 정의형에 대해 get과 put의 추가적인 형태를 정의할 수 있다. 이 편의 기능의 모든 것은 표준 에이다 기법에 의존한다. 포트란과 달리 I/O를 위한 지원은 언어 자체에 내장하지 않는다.

C의 텍스트 I/O

C는 I/O를 stdio라 하는 라이브러리 패키지를 통해 제공한다. 에이다처럼 I/O을 위한 지원을 언어 자체에 내장하지 않는다. 하지만 많은 C 구현은 I/O 함수의 지식을 컴파일러 안에 설치한다. 그래서 그것은 인자의 부정확한 사용이 나타나면 경고를 발생시킨다.

예 7.137
C의 형식화된 출력

예제의 출력문은 C에서는 다음과 같을 것이다.

```
char s[20];
int n;
double r[10];
...
fprintf(my_file, "%20s%10d", s, n);
for (i = 0; i < 10; i++) fprintf(my_file, "%8.2f", r[i]);
fprintf(my_file, "\n");
```

fprintf의 인자는 파일, 형식 문자열과 수식의 나열이다. 형식 문자열은 구문이 완전히 다르지만 포트란의 형식과 유사한 특성을 가지고 있다. 일반적으로 형식 문자열은 "위치 지정자"를 내재한 문자의 나열로 구성되어 있다. 각 위치 지정자는 % 기호로 시작된다. 위치 지정자 %20s는 20개의 문자의 길이를 갖는 문자열을 의미한다. %d는 10진 표기로 나타낸 정수를 나타낸다. %8.2f는 소수점의 오른편에 2자리를 갖는 8개의 문자 길이를 갖는 부동소수점 숫자를 의미한다.

예 7.138
형식 문자열에서 텍스트

포트란처럼 형식을 계산하고 문자열에 저장할 수 있고 단일 fprintf문은 임의의 수의 수식을 출력할 수 있다. 배열을 출력하기 위해서는 에이다처럼 명시적인 for 루프를 필요로 한다. 흔히 형식 문자열도 레이블을 붙인 텍스트와 공백문자를 포함한다.

```
strcpy(s, "four");            /* "four"를 s로 복사 */
n = 20;
char *fmt = "%s score and %d years ago\n";
fprintf(my_file, fmt, s, n);
```

% 기호를 두 번 쓰면 그것을 출력할 수 있다.

```
printf(my_file, "%d%%\n", 25);  /* "25%"를 출력 */
```

예 7.139

C에서 형식화된 입력

C의 입력은 비슷한 형태를 띤다. fscanf 루틴은 파일, 형식 문자열과 변수로의 포인터의 나열을 인자로 받는다. 일반적인 경우에 형식 이후의 모든 인자는 "~로의 포인터" 연산자가 앞에 오는 변수 이름이다.

```
fscanf(my_file, "%s %d %lf", &s, &n, &r[0]);
```

이 호출에서 %s 위치 지정자는 공백 문자를 포함하지 않는 최대 길이를 가진 문자열과 일치할 것이다. 이 문자열이 20 글자(s의 길이)보다 길면 fscanf는 문자열에 대한 저장 공간의 끝을 넘어 기록할 것이다(scanf의 이런 취약점은 인터넷 해커가 이용하는 소위 "버퍼 오버런" 버그의 원천 중 하나다). 문자 3개로 된 %lf 위치 지정자는 라이브러리 루틴에게 대응하는 인자가 double임을 알려준다. 문자 2개의 나열로 된 %f는 float에 읽을 것이다.[2] 잘못된 크기의 변수에 대한 위치 지정자를 우연하게 사용하는 것은 C의 좀 더 오래된 구현에서 흔한 오류다. 따라오는 인자에 대해 &를 잊는 것도 또 하나의 오류다. 현대 C 컴파일러가 fscanf의 특수 경우 지식을 이용해 가끔 그런 실수를 잡아내지만 유형적으로 안전한 I/O를 사용하는 언어에서는 항상 잡힐 것이다. r의 단일 원소를 읽기 때문에 전체 배열을 읽기 위해 fprintf를 이용하는 것처럼 for 루프를 필요로 함을 알자.

위에서 I/O 루틴이 가변 개수의 인자를 취하게 하기 위해 포트란과 파스칼의 I/O 루틴을 언어 내에 넣음을 깨달았다. 또한 에이다에서 I/O를 라이브러리 내로 이동시키는 것이 모든 출력 수식에 대해 put의 별개 호출을 만들게 강제함을 알았다. 그래서 fprintf와 fscanf가 어떻게 동작하는가? C가 가변 개수의 매개변수를 가진 함수를 허용함을 안다(이런 함수에 대해 8.3.3절에서 좀 더 자세히 설명한다). 불행히도 따라오는 매개변수의 유형은 정해져 있지 않다. 이는 일반적인 경우에 컴파일 시에 가변 길이 인자 리스트의 유형 검사가 불가능하게 한다. 게다가 C의 실행 시간 유형 디스크립터가 없음도 실행 시간 검사를 불가능하게 막는다. 동시에 (fprintf와 fscanf를 포함한) C 라이브러리가 언어 표준의 일부이기 때문에 이 루틴의 특별한 지식을 컴파일러 내에 내장시킬 수 있다. C의 I/O 루틴을 공식적으로 "보통의" 함수로 정의하지만 일반적으로 이를 포트란과 파스칼의 그것과 동일한 방법으로 구현한다. 결과적으로 fprintf나 fscanf에 대한 인자가 형식 문자열과 일치하지 않을 때 C 컴파일러는 가끔 좋은 오류 진단을 제공할 것이다.

표준 입력 스트림에서의 I/O와 출력 스트림으로의 I/O를 단순화시키기 위해 stdio는 fprintf와 fscanf의 초기 인자를 생략하는 printf와 scanf라 부르는 루틴을 제공한다. 프로그램 내에서 문자열의 형식화를 편리하게 이용하기 위해 stdio는 sprintf와 sscanf이라 부르는 루틴도 제공한다. 이 함수는 fprintf와 fscanf의

주2. C의 double은 대부분의 구현에서 2배 정밀도 IEEE 부동소수점 숫자다. float는 보통 단순정밀도다. %s에 대한 안정성의 결여는 fscanf와 관련된 일부 문제 중 하나일 뿐이다. 이외에 오류가 있는 입력을 지나치는 능력이 없음과 충분치 못한 입력이 있을 때 정의되지 않은 행동이 있다. fscanf 대신 경험이 많은 C 프로그래머는 (길이를 제한한) 입력을 문자열로 읽는 fgets를 사용하고 이후에 strtol(문자열에서 long으로), strtod(문자열에서 double로) 등을 사용하는 경향이 있다.

초기 인자를 문자의 배열로의 포인터로 대체한다. sscanf 함수는 이 배열로부터 "읽고", sprintf는 그것에 "쓴다." 포트란 90은 소위 내부 파일을 통해 프로그램 내부의 형식화를 위한 비슷한 지원을 지원한다.

C++에서의 텍스트 I/O

C의 후계자로서 C++는 이전 하위 절에서 설명한 stdio 라이브러리를 지원한다. 언어의 객체지향 기능을 이용하는 iostream이라 하는 새로운 I/O 라이브러리를 지원한다. iostream 라이브러리는 stdio보다 좀 더 융통성 있고 (취향의 문제이긴 하지만) 좀 더 훌륭한 구문을 제공할 것이고 유형적으로 완전히 안정적이다.

예 7.140
C++의 형식화된 출력

C++의 스트림은 보통 비트 단위 시프트에 대해 사용하는 <<와 >> 기호를 마음대로 사용하기 위해 연산자 다중 정의를 사용한다. iostream 라이브러리는 각 고유형에 대해 <<와 >>의 다중 정의된 버전을 제공하고 프로그래머는 새로운 유형에 대한 버전을 정의할 수 있다. C++에서는 문자열을 출력하기 위해서 다음처럼 작성한다.

```
my_stream << s;
```

문자열과 정수를 출력하기 위해 다음처럼 작성한다.

```
my_stream << s << n;
```

이 코드는 my_stream이 iostream 라이브러리에 정의된 ostream 클래스(출력 스트림)의 인스턴스기를 요구한다. << 연산자는 3.6.2절에서 설명한 "연산자 함수" ostream::operator<<에 대한 문법적 편의성이다. <<가 왼쪽에서 오른쪽으로 결합하기 때문에 위 문장은 다음과 동일하다.

```
(my_stream.operator<<(s)).operator<<(n);
```

ostream::operator<<가 그것의 결과물로서 첫 인자로의 참조를 반환하기 때문에 이 코드는 동작한다(8.3.1절에서 확인하겠지만 C++은 변수에 대해 값 모델과 참조 모델을 지원한다).

예 7.141
스트림 조작자

이제까지 봐온 것처럼 ostream으로의 출력은 기본 형식화 규정을 사용한다. 규정을 변경하기 위해 소위 스트림 조작자를 << 연산의 나열 속에 내장시킬 수도 있다. (기본 10진 표기가 아닌) 8진수 표기로 n을 출력하기 위해 다음과 같이 작성할 수 있다.

```
my_stream << oct << n;
```

s와 n이 차지할 열의 수를 명시하기 위해 다음과 같이 쓸 수 있다.

```
my_stream << setw(20) << s << setw(10) << n;
```

oct 조작자는 스트림이 이후의 모든 숫자 출력을 8진수로 출력하게 만든다. setw

조작자는 다음 문자열이나 숫자 출력을 지정된 최소 폭의 영역에 표시하게 한다(이 행동은 하나의 출력 이후에 기본값으로 복귀한다).

ostream을 매개변수로 취하고 그것의 결과물로서 ostream의 참조를 만들어내는 함수로 oct 조작자를 선언한다. 빈 괄호가 뒤에 오지 않기 때문에 위의 출력 나열에서 oct의 출현은 oct에 대한 호출이 아니다. 오히려 oct에 대한 참조를 오른쪽 인자로 조작자 함수를 기대하는 <<의 다중 정의된 버전에 전달한다. <<의 이 버전은 함수를 호출해 스트림(<<의 왼쪽 인자)을 인자로 전달한다.

setw 조작자는 더 까다롭다. "함수 캡슐화기" 객체라 부를 수 있는 것으로의 참조를 반환하는 함수로 선언한다. 캡슐화기는 정수와 ostream과 정수를 인자로서 기대하는 함수로의 포인터, 이 두 가지를 포함한다. 위 코드에서 setw(20)를 숫자 20과 setw 조작기로의 포인터를 포함하는 캡슐화기에 대한 집합체로 간주할 수 있다(좀 더 정확하게 말하면 이는 캡슐화기 객체에 대해 생성자 함수의 호출이다. 9.3절에서 자세히 생성자를 논할 것이다). iostream 라이브러리는 오른편 인자로 캡슐화기를 기대하는 <<의 다중 정의된 버전을 제공한다. <<의 이런 버전은 함수 캡슐화기 내의 함수를 호출하고 그것에 인자로 스트림(<<의 왼편 인자)과 캡슐화기 내의 정수를 전달한다.

예 7.142
C++의 배열 출력

ostream의 형식화기 행태를 변경하기 위해 iostream 라이브러리는 다양한 조작기를 제공한다. 하지만 C++이 포인터와 배열에 대한 C의 처리를 물려받았기 때문에 ostream이 배열의 길이를 알 방법이 없다. 결과적으로 완전한 출력 예는 r 배열을 출력하기 위해 여전히 for 루프를 필요로 한다.

```
char s[20];
int n;
double r[10];
...
my_stream << setw(20) << s << setw(10) << n;
for (i = 0; i < 10; i++)
    my_stream << setiosflags(ios::fixed)
       << setw(8) << setprecision(2) << r[i];
my_stream << "\n";
```

여기서 for 루프의 연속된 출력 내의 조작기는 부동소수점 수에 대한 (정확히는 아닌) 영역의 폭이 8이고, 소수점 이후의 자리수가 2인 고정된 형식을 지정한다. setiosflags와 setprecision 조작기는 스트림의 기본 형식을 변경한다. 이 변경은 모든 후속 출력에 적용된다.

예 7.143
기본 형식을 변경하기

스트림 조작기를 반복적으로 호출하는 것을 피하기 위해 아래와 같이 위의 예를 변경할 수 있다.

```
my_stream.flags(my_stream.flags() | ios::fixed);
```

```
my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
```

setw 조작기는 한 항목만의 출력 폭에 영향을 준다. 기본값의 복구를 편리하게 하기 위해 flags와 precision 함수는 이전 값을 반환한다.

```
ios::fmtflags old_flags = my_stream.flags(my_stream.flags() | ios::fixed);
int old_precision = my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
my_stream.flags(old_flags);
my_stream.precision(old_precision);
```

C++의 형식화된 입력은 형식화된 출력과 유사하다. ostream 대신 istream을, << 대신 >> 연산자를 사용한다. 표준 입력과 출력 스트림상의 I/O는 다른 함수를 필요로 하지 않는다. 프로그래머는 단순히 연속된 입력 혹은 출력을 표준 스트림 이름 cin이나 cout으로 시작하면 된다(C의 전통을 따르면 오류 메시지에 대해서는 표준 스트림 cerr이다). 프로그램 내에서 문자열의 형식화를 지원하기 위해 strstream 라이브러리는 istream과 ostream에서 유도하고 스트림 변수를 파일 대신 문자열과 연결시킨 istrstream과 ostrstream 객체 클래스를 제공한다.

✓ 확인문제

68. 상호작용적과 파일 기반 I/O 간의 차이점, 임시와 지속적인 파일 간의 차이점과 이진과 텍스트 파일 간의 차이점을 설명하라(이 정보의 일부는 책에 있다).

69. 텍스트와 이진 파일의 상대적인 이점은 무엇인가?

70. 유닉스, 윈도우와 매킨토시 파일의 행 종료 규정을 설명하라.

71. 라이브러리 루틴을 통해 지원하는 것에 비해 I/O를 프로그래밍 언어 내에 구축하는 것의 장점과 단점이 무엇인가?

72. 포트란, 에이다, C와 C++이 채용한 텍스트 I/O에 대한 여러 가지 방법을 요약하라.

73. C에서 scanf 방법의 취약점 중 일부를 설명하라.

74. 스트림 조작기는 무엇인가? C++에서 어떻게 사용하는가?