컴파일러 개발자가 들려주는 C 이야기

Expert C programming
by Peter van der Linden

컴파일러 개발자가 들려주는 C 이야기:
아무도 알려주지 않던 심오한 C의 비밀

전자책 1쇄 발행 2022년 2월 15일 **지은이** 페터르 판데르린던 **옮긴이** 정기훈 **펴낸이** 한기성 **펴낸곳** (주)도서출판인사이트
편집 신승준, 송우일, 정수진 **등록번호** 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울시 마포구 연남로5길 19-5
전화 02-322-5143 **팩스** 02-3143-5579 **블로그** http://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-342-4

프로그래밍 인사이트

컴파일러 개발자가 들려주는 C 이야기

아무도 알려주지 않던 심오한 C의 비밀

페터르 판데르린덴 지음 | 정기훈 옮김

인사이트

헌정사(獻呈詞)

이 책을 피자, 달마티안 개, 해먹에서 보낸 일요일 오후, 코미디에 바친다. 이런 것들이 더 많아진다면 세상은 훨씬 좋아질 것이다. 이제 집필을 모두 마쳤으니 이것들을 다시 가까이 하려고 한다.

정말로 나는 다음 주 일요일 오후에 흔들거리는 해먹에 누워 내 달마티안 개가 피자를 먹으려고 애쓰는 걸 보면서 미소 짓고 있을 것이다.

아울러 영국 요크셔의 텍스턴(Theakston) 브루잉 회사의 에일 맥주도 훌륭했음을 밝힌다.

차례

옮긴이의 글

고전을 읽는 이유에는 여러 가지가 있겠지만 고전 속 지혜를 현재와 미래에 투영하여 새로운 시대의 지혜로 승화시키고자 하는 것이 주된 목적일 것이다. 동시에 고전이 쓰인 당시의 시대 상황을 엿봄으로써 지금의 세상과는 달랐던 과거를 간접 체험하는 재미를 느끼는 것도 또 다른 이유가 될 것이다.

인문학에서의 고전은 수백 년, 길게는 천여 년 전에 쓰인 것도 있지만, 발전 속도의 차이가 현저한 공학 분야에서는 고전의 기준이 다르다. 인류 문명은 수렵 활동을 시작으로 현재까지의 발전을 이룩하는데 1만 년에 달하는 시간이 필요했지만, 컴퓨터공학 분야는 굳이 무어의 법칙을 언급할 필요도 없이 이미 통상적으로 사용하는 보조 저장 장치의 용량이 1990년대 1.44MB에서 2020년대 16GB로 이미 1만 배가 넘을 정도로 기술의 발달은 상상을 초월하고 있다. 시간과 배수를 직접적으로 비교하는 것은 다소 비약적이지만 공학기술 관점에서 볼 때 1990년대에 서술한 공학기술은 2020년대 시점에서 보면 충분히 고전으로 느낄 수 있는 기술적, 시대적 간극이 있다. 특히 C 언어는 마치 철학과도 같이 컴퓨터공학과 함께 발전해오면서 프로그래밍 언어학 이상의 의미를 지니게 되었다.

《컴파일러 개발자가 들려주는 C 이야기》는 C 언어의 가장 밑바닥을 다루는 컴파일러 개발자인 저자가 언어학적, 실무적 경험을 담아 집필한 책이다. 저자가 밝힌 바와 같이, 이 책은 C 언어에 대해 어느 정도 경험을 가진 사람이 더 깊은 지식과 개발 전략을 익혀 한 단계 더 높은 수준으로 성장하는 데 도움을 주는 역할을 한다. 일반 C 언어 서적이 개념만 언급하고 넘어가는 부분을 이 책에서는 밑바닥까지 파고들어 그렇게 될 수밖에 없었던 기술적 혹은 시대적 원인까지 알려준다. 참으로 아이러니한 것은 (정도는 다르지만) 당시에 사용하던 개발 전략이 지금도 적용이 가능할 뿐만 아니라, 과거에 큰 피해를 주었던 버그가 현재에도 비슷한 영향력을 행사(?)한다는 것이다. 마치 고전을 통해 과거의 경험을 현대에 반추하듯 이 책을 통해 21세기의 소프트웨어 개발 전략을 점검하고 더 넓은 영역으로 확대할 수 있으

리라 기대한다.

다른 여타 고전도 그렇겠지만, 이 책을 번역하는 데는 굉장한 시간과 노력이 필요했다. 과거의 기술과 시대적 배경을 조사하고 이를 현대에 맞춰 재해석하는 과정은 얼음 위를 걷듯 신중한 선택의 연속이었으며, 고증과 검증 속에서도 저자의 경험과 위트가 적절한 밸런스를 이룰 수 있도록 하는 것은 굉장히 힘들고 까다로운 작업이었다. 장황한 변명으로 시작하는 이유는 이 책을 번역하기까지 인고의 시간을 함께해준 분들에 대한 고마움과 미안함을 표현하기 위함이다. 먼저, 역자를 믿고 이 책을 번역할 수 있도록 기회를 준 도서출판 인사이트 한기성 대표님, 완주할 수 있도록 독려와 격려를 병행해 준 신승준 부장님, 송우일 에디터님, 정수진 에디터님께 깊은 감사를 드린다. 아울러 얄궂게 투덜거리면서도 곁에서 응원해준 아내와 윤서, 태원에게도 지면을 빌어 고마움과 미안함을 전한다.

정기훈

머리말

최근에 서점을 돌아보면서 C와 C++ 서적 대부분이 딱딱하게 집필된 것을 보고 실망했다. 누구나 프로그래밍을 즐길 수 있다는 취지로 쓴 책도 더러 있었지만, 길고 지루하고 단조로운 글로 인해 그 의미가 퇴색해 버렸다. 졸지 않고 그 책들을 읽을 수 있다면 아마도 여러분에게 유용할지도 모른다. 하지만 프로그래밍을 그렇게 만만하게 봐서는 안 된다.

프로그래밍은 경이롭고 중요하고 도전적인 활동이다. 그러므로 프로그래밍 책은 프로그래밍에 대한 열정이 넘쳐 나야 한다. 이 책은 교육적이면서도 재미있게 구성하려고 노력했다. 이 책이 여러분에게 즐거움을 선사한다고 생각된다면 여러분 곁에 두기(구매하기) 바란다(고맙다!). 만일 그렇지 않다면 이 책을 다시 원래 자리에 놓아두면 된다.

이 책까지 포함하여 이제 세상에는 C 프로그래밍에 관한 책이 수백 권 있다. 과연 이 책이 여타 C 프로그래밍 책과 다른 점은 무엇일까?

《컴파일러 개발자가 들려주는 C 이야기》는 모든 프로그래머의 두 번째 C 책이어야 한다. 이 책에 수록된 대부분의 설명, 팁, 기술은 다른 책에서 찾기 어려울 것이다. 보통 이런 내용은 잘 작성된 설명서의 여백이나 옛 인쇄물의 뒷면에 끄적거린 형태로 볼 수 있다. 이 책에 실린 지식은 썬 마이크로시스템즈(Sun Microsystems, 이하 썬)[1]의 컴파일러 및 운영 체제 부서에서 일한 나와 동료들이 수년간 C 프로그래밍을 하면서 축적해 온 것들이다. 또 인터넷에 연결된 자판기, 우주 공간의 소프트웨어 문제, C 버그로 인한 AT&T 장거리 전화망 장애 등 흥미로운 C 이야기와 일화도 담았다. 마지막 장은 쉬운 C++ 자습서로, C의 파생물로 시작해 대형 언어가 된 C++를 익히는 데 도움이 될 것이다.

이 책에 수록된 내용은 PC와 유닉스(UNIX) 시스템의 근간이 된 안시(ANSI) 표준

1 (옮긴이) 2010년 오라클에 인수됐다.

C에 적용된다. 가상 메모리 등 유닉스 플랫폼에서 찾아볼 수 있는 정교한 하드웨어와 관련된 C의 고유한 측면도 자세히 설명한다. 아울러 PC 메모리 모델과 인텔(Intel) 8086 제품군이 C 코드에 미친 영향에 대해서도 자세히 설명한다. C의 기초를 이미 습득한 사람이라면 이 책에서 한 프로그래머가 수년간 고른 팁, 힌트, 손쉬운 방법 등을 가득 발견하게 될 것이다. 이 책은 많은 C 프로그래머들이 혼란스러워하는 다음과 같은 주제를 다룬다.

- `typedef struct bar {int bar;} bar;`의 정확한 의미는?
- 함수에 다른 크기의 다차원 배열을 전달하려면 어떻게 해야 할까?
- 도대체 왜 `extern char *p;`가 다른 파일의 `char p[100];`에 연결되지 않나?
- 버스 오류(bus error)란 무엇인가? 세그먼테이션 위반(segmentation violation)이란 무엇인가?
- `char *foo[]`와 `char (*foo)[]`의 차이점은?

답하지 못하는 질문이 있거나 C 전문가들이 어떻게 대처하는지 알고 싶은 내용이 있다면 이 책을 계속 읽기 바란다. 위 질문의 답뿐 아니라 C에 대한 그 밖의 지식을 이미 전부 알고 있더라도 지식을 더욱 강화하기 위해 책을 구매하기 바란다. 서점 직원에게는 "친구에게 사 주는 것"이라고 말하면 된다.

페터르 판데르린던, 캘리포니아 실리콘 밸리에서

감사의 글

다른 대부분의 책에서는 다음과 같은 부자연스러운 감사의 글이 나온다. 지은이에게 돈을 빌려준 모든 사람에게 하나 마나 한 감사를 하고 초등학교 친구를 시작으로 배우자의 친척을 언급하다가 지도 교수에게 굽실거리며 비위를 맞추려는 뻔뻔한 시도로 마무리한다(그리고 마지막으로 핵심적인 문제를 풀기 위해 화장실 휴지를 앞으로 거느냐 뒤로 거느냐로 수많은 연구를 한 막강한 영향력을 지닌 오즈 교수[1]에게 찬사를 보낸다). 이 책은 절대 그렇지 않다. 이 책을 집필하면서 내게 정말 많은 도움을 준 사람들을 모두 나열했다. 여기에 언급한 모든 사람에게 감사의 마음을 전한다. 그리고 타히티 해변에서 왕족과 같은 여유로운 시간을 보내며 이 사람들을 생각할 것이다. 정말이다!

가장 먼저 초고를 검토하고 많은 수정 사항과 제안을 보내 준 Phil Gustafson과 Brian Scearce에게 특별한 감사를 드린다. 그들은 자신의 몸을 과학에 맡겼다고 할 정도로 열정적으로 수고해 주었다.

다음은 책을 집필하는 과정에서 원고의 상당 부분을 읽어 준 친구와 동료들이다. 깊은 감사를 드린다. 이들은 원고에 대해 명료한 의견을 아낌없이 제시해 주었다.

Lee Bieber, Keith Bierman(명함에는 직함이 '대중 선동가'라고 적혀 있는데, 그는 확실히 그 직업에 딱 맞는 사람이다), Robert Corbett, Rod Evans, Doug Landauer, Joseph McGukin, Walter Nielen, Charlie Springer(손가락으로 이진수를 세는 방법을 가르쳐 주었다. 여러분도 이 방법으로 1023까지 셀 수 있다), Nicholas Sterling, Panos Tsirigotis, Richard Tuck. 이들은 원고에 대해 관대하면서도 명료한 의견을 제시하였다.

다음은 내게 도움을 준 사람들로, 때로는 내 끝없는 질문에도 참을성 있게 대답

1 (옮긴이) 《오즈의 마법사》의 주요 등장인물인 마법사를 패러디한 것으로 보인다. 원래 서커스 단원이었으나 사고로 오즈에 불시착하고 나서 여러 가지 속임수를 써서 '위대한' 마법사 행세를 하는 인물이다.

해 주었다.

Chris Aoki, Arindam Banerji, Mark Brader, Brent Callghan(오디오 기능을 해킹해 파헤친 사람), David Chase, Joseph T. Chew, Adrian Cockcroft, Sam Cramer, Steve Dever, Derek Dongray, Joe Eykholt, Roger Faulkner, Mike Federwisch, David Ford, 썬 독일 지사의 Burkhard Gerull, Rob Gingell, Cathy Harris(상식이 풍부했다), Bruce Hilderbrand(그리고 그의 놀라운 플라잉 자전거 묘기), Mike Kazar, Bob Jervis, Diane Kelly, Charles Lasner, Bill Lewis, Greg Limes, Tim Marsland, Marianne Mueller, Eugene N. Miya, Chuck Narad, Bill Petro(그의 영감 넘치고 끊임없는 역사 이야기), Trelford Pinkerton, Alex Ramos, Fred Sayward, Bill Shannon, Mark D. Smith, Kathy Stark, Dan Stein, Steve Summit, Paul Tomblin, Wendy van der Linden(312쪽 C++로 bob-for-apples 상속 예제를 작성하고, 39쪽 "두 개의 'l'이 있는 null" 구절의 운율을 다듬었다), Dock Williams, Nigel 'Gag Me' Witherspoon, Brian Wong, Tom Wong.

내 원고에 은유적인 표현을 더하고 여러 차례 내 문제로 밤을 새워 준 Karin Ellison 편집장에게 감사드린다. 그리고 프레임메이커(Framemaker)에 대한 많은 질문에 답해 준 Astrid Julienne와 썬 도서관의 Peter Van Coutren에게도 감사드린다.

또한 풍부한 지식을 제공해 준 프렌티스 홀(Prentice Hall)의 Mike Meehan, Camille Trentacoste, Susan Aumack, Eloise Starkweather, Nancy Boylan에게도 감사드린다.

아울러 집필 과정에서 나를 성가시게 하지 않았던 사람들에게도 감사를 표한다. 나와 적절한 거리를 유지했고, 심각한 상황을 만들지도 않았다. 괜찮은 사람들이다.

Dirk Wibble-O'Doolery, P. A. G. Embleton

이 책의 내용 중 일부는 대화, 이메일, 인터넷 게시물 및 업계 동료의 제안에서 영감을 받았다. 이 경우 최대한 출처를 밝혔지만, 혹시라도 간과한 부분이 있다면 지면을 빌어 사과드린다.

페터르 판데르린덴, 캘리포니아 실리콘 밸리에서

들어가는 글

C 코드. C 코드 실행. 실행 코드 실행…. 제발!

— 바버러 링(Barbara Ling)

모든 C 프로그램은 똑같은 일을 한다: 글자를 보고 아무것도 하지 않는다.

— 피터 와인버거(Peter Weinberger)

《C 함정과 실수》(피어슨에듀케이션코리아, 2004), 《The C Puzzle Book》(Addison-Wesley, 1998), 《Obfuscated C and Other mysteries》(Wiley, 1992)처럼 시사하는 바가 많은 제목이 붙은 C 언어 책은 많은데 다른 프로그래밍 언어에는 그런 책이 없다는 것을 눈치챘는가? 이에 대한 아주 설득력 있는 이유를 들자면 다음과 같다.

C 프로그래밍은 완벽해지는 데 수년이 걸리는 기술이다. 머리가 영리한 사람이라면 C의 기초는 아주 빨리 배울 수 있을 것이다. 그러나 언어의 뉘앙스를 터득하고 다양한 종류의 프로그램을 많이 작성하면서 C 프로그래밍 전문가가 되는 데는 훨씬 오랜 시간이 걸린다. 자연 언어에 비유하자면, 파리에서 커피를 주문할 수 있는 것과 지하철에서 파리 토박이에게 어디에서 내릴지 물어볼 줄 아는 것의 차이라고 할 수 있다. 이 책은 안시 C 프로그래밍 언어의 고급 설명서로, C 프로그램은 이미 작성할 수 있으나 전문가의 통찰력과 기술을 빠르게 습득하려는 사람들을 위한 책이다.

전문 프로그래머는 수년에 걸쳐 기술 도구들을 구축한다. 여기에는 관용구, 코드 조각, 세련된 기술 등이 포함된다. 이것들은 시간이 흐르면서 천천히 습득하게 되는데, 경험이 풍부한 동료의 어깨너머로 배우거나 다른 사람들이 작성한 코드를 관리하면서 배운다. 그 외 C의 다른 부분은 독학으로 체득하는데, 다음은 대다수 초급 C 프로그래머가 실수하는 대표적인 표기 오류다.

`if (i==3)` 대신 `if (i=3)`이라고 쓴다.

한 번 겪고 나면 비교문을 쓸 자리에 대입문을 쓰는 이 고통스러운 오류는 거의 되풀이하지 않는다. 어떤 프로그래머는 아예 다음과 같이 상수를 먼저 쓰는 습관을 키우기도 한다.

```
if (3==i)
```

이렇게 하면 등호 한 개가 실수로 생략되는 경우, 컴파일러가 '상수에 대입하려는 시도가 있다'며 불평할 것이다. 물론 변수 두 개를 비교하려다 저지르는 오류에서 여러분을 보호하지는 못하지만, 그래도 약간의 도움은 될 것이다.

2000만 달러짜리 버그

1993년 봄, 비동기 I/O 라이브러리에 문제를 일으키는 '최우선순위' 버그가 썬의 운영 체제 개발 부서에 보고됐다. 이 라이브러리 기능이 필요했던 고객에게 2000만 달러 상당의 하드웨어를 판매할 예정이었으나 버그 때문에 중단되었고, 우리는 원인을 찾기 위해 아주 필사적이었다. 여러 번의 집중적인 디버깅을 마친 후에야 문제의 원인이 다음 문장 때문이라는 것을 알아낼 수 있었다.

```
x==2;
```

원래 대입문이었는데 오타가 난 것이다. 우연히 프로그래머의 손가락이 = 키를 한 번이 아니라 두 번 눌렀고, 그 결과 이 문장은 x를 2와 비교한 다음 참인지 거짓인지 알아보고는 결과를 무시했다.

C는 표현식(expression) 언어에 불과해서 컴파일러는 평가가 끝난 표현식에 대해 불평하지 않고 부작용이 없으면 결과를 단순히 버릴 뿐이다. 우리는 운 좋게 문제의 원인을 찾아낸 걸 축하해야 할지, 그런 흔한 타자 실수로 값비싼 문제를 일으킨 데 좌절해야 할지 알지 못했다. 일부 린트(lint)[1] 프로그램으로 이 문제를 찾아낼 수 있지만, 이런 문제는 너무 쉬워서 자동 검증 프로그램을 사용하지 않았다.

이 책은 이와 같은 유익한 이야기를 한데 모았다. 경험 많은 프로그래머들의 지혜를 기록한 것이기에 여러분이 이러한 실수를 반복하지 않도록 도와줄 것이다. 그

1 (옮긴이) 소스 코드를 분석하여 프로그램 오류, 버그, 스타일 오류, 의심스러운 구조체 등을 알려 주는 프로그램을 총칭하는 용어로 린트 또는 린터(linter)라고도 부른다. 이 용어는 과거 C 언어 소스 코드를 검사하는 유닉스 유틸리티로부터 시작되었다.

리고 대체로 익숙해졌지만 여전히 탐험해 보지 못한 C 영역을 안내하는 역할을 한다. 또한 선언, 배열, 포인터 등 주요 주제에 대한 집중적인 토론과 함께 많은 힌트와 이를 잘 기억하는 방법도 제공한다. 안시 C 용어는 필요에 따라 일상 용어로 번역했다.

이 책에는 '프로그래밍 도전'과 '유용한 팁'이라는 코너가 있다.

> [프로그래밍 도전]
>
> '프로그래밍 도전' 코너에는 여러분이 작성해야 할 프로그래밍 과제를 제시할 것이다.

> [유용한 팁]
>
> '유용한 팁'에는 실무에서 통할 아이디어, 노하우, 지침을 적어 놓았다. 이 코너는 그저 여러분의 것으로 받아들이면 된다. 물론 더 선호하는 지침이 있다면 무시해도 좋다.

이 책에서 사용한 규칙

이 책에서는 변수 이름 규칙으로 과일과 채소를 사용한다(물론 실제 코드가 아닌 작은 코드 조각에 한정한다).

```
char pear[40];
double peach;
int mango = 13;
long melon = 2001;
```

이렇게 하면 C 예약어와 프로그래머가 제공하는 이름을 쉽게 구분할 수 있다. 어떤 사람들은 사과와 오렌지를 비교할 수 없다고 말하지만 상관없다. 어차피 둘 다 나무에서 자라고 손에 쥘 수 있는 동그란 형태를 띠고 있으며 먹을 수 있다. 익숙해지면 변수 이름으로 과일 이름을 사용하는 코드가 실제로 도움이 되는 것 같다. 규칙 하나가 더 있다. 때로는 강조하기 위해 핵심 사항을 반복한다. 다시 한번 말하는데 때로는 강조하기 위해 핵심 사항을 반복한다.

미식가의 요리책처럼 이 책은 여러분이 샘플로 활용할 수 있도록 각각의 맛있는 음식 모음으로 구성되어 있다. 즉, 모든 장은 관련이 있으면서도 그 자체로 독립적

인 절로 나뉜다. 그러므로 책을 처음부터 끝까지 순서대로 읽거나 읽고 싶은 주제를 먼저 부분적으로 읽어도 무방하다. 그리고 C 프로그래밍이 실전에서 어떻게 동작하는지 다양한 사례를 소개함으로써 기술적인 세부 사항까지 알 수 있게 했다. 유머는 새로운 내용을 익히는 중요한 기법이다. 이를 위해 장 마지막에는 재미있는 C 이야기 또는 소프트웨어 일화가 들어 있는 '쉬어 가기' 절을 넣어 읽는 속도를 조절했다.

여러분은 이 책을 아이디어의 원천으로 사용하거나 C 언어 팁이나 관용구 모음 정도로 사용할 수 있다. 또는 경험 많은 컴파일러 작성자에게서 안시 C를 더 배우는 용도로도 사용할 수 있다. 요약하자면 이 책은 안시 C 기법을 익히는 데 도움을 주는 유용한 아이디어 모음집으로 모든 정보와 힌트, 지침을 한곳에 모아 여러분에게 즐거움을 선사한다. 이제 가까운 데 있는 아무 종이나 가져와서 행운의 코딩 연필을 꺼내 쥐고 편안한 모니터 앞에 기대어 앉아 본격적으로 즐거움을 만끽해 보자!

쉬어 가기: 파일 시스템 튜닝

C와 유닉스를 보면 유쾌할 때가 많다. 엉뚱한 발상이지만 괜찮다. IBM·모토로라·애플이 함께 만든 PowerPC 아키텍처에는 I/O를 순차적으로 수행하는 'E.I.E.I.O.'라는 CPU 명령[2]이 있는데, 'Enforce In-order Execution of I/O'를 뜻한다. 이와 비슷한 분위기의 tunefs라는 유닉스 명령이 있는데, 이 명령은 숙련된 시스템 관리자가 파일 시스템의 동적 파라미터(parameter)를 변경하고 디스크의 블록 레이아웃을 개선하는 데 사용한다.

모든 BSD(Berkeley Software Distribution) 유닉스 명령과 마찬가지로 원래 tunefs의 온라인 매뉴얼 페이지는 다음과 같은 '버그' 섹션으로 끝난다.

> 버그:
> 이 프로그램은 마운트된 활성 파일 시스템에서 작동해야 하지만 실제로는 그렇지 못하다. 슈퍼 블록은 버퍼 캐시에 보관되지 않기 때문에 이 프로그램은 마운트가 해제된 파일 시스템에서만 제대로 동작한다. 루트(root) 파일 시스템에서

2 (옮긴이) 〈Old McDonald had a farm〉이라는 동요의 가사 첫 줄이 'Old McDonald had a farm, E-i-e-i-o'로 시작하는데 이 책의 지은이는 CPU 명령어와 동요 가사 발음의 유사성을 유쾌한 발상이라고 여기고 있다. 이 동요는 한국에서 〈그래 그래서〉라는 제목으로 번안되었다.

실행할 경우 시스템을 재시동해야 한다. 파일 시스템(file system)은 튜닝할 수 있지만 물고기(fish)는 튜닝할 수 없다.

심지어 버그 섹션 초안에는 버그 섹션을 지우려는 사람을 겁주기 위한 경고성 문구까지 있었다.

이 내용을 지우는 사람에게는 유닉스 악마(Demon)[3]가 `time_t` 값이 한 바퀴 돌 때까지 졸졸 따라다닐 것이다.

이후 썬이 다른 회사들과 함께 SVR4(System V Release 4) 유닉스로 바꾸면서[4] 우리는 값진 보석을 잃게 되는데, SVR4 매뉴얼 페이지에서 '버그' 섹션이 '노트(Notes)' 섹션으로 대체되었다(이렇게 한다고 누가 속을까?). '참치(tuna fish)' 문구가 사라졌는데[5] 그 죗값으로 아마 유닉스 악마가 지금까지 쫓아다니고 있을 것이다. 그 악마는 어쩌면 lpd(line printer daemon)[6]인지도 모른다.

[프로그래밍 도전] 컴퓨터 시간

`time_t`는 언제 꽉 찰까? 이 문제를 해결하는 프로그램을 작성해 보자.

1. `time_t`의 정의(definition)를 찾는다. 정의 부분은 /usr/include/time.h 파일에 있다.
2. `time_t` 타입 변수에 가장 큰 값을 넣고, 이 변수를 `ctime()`에 전달하여 아스키(ASCII) 문자열로 변환하는 프로그램을 작성한다. 그리고 문자열을 출력한다. `ctime`은 C 언어와 아무 관련이 없다. 단지 '변환 시간'을 의미한다.

과연 주석을 제거한 익명의 기술자는 앞으로 몇 년간 유닉스 데몬을 걱정해야 할까? 그것을 확인할 수 있도록 프로그램을 수정해 보자.

1. `time()`을 호출하여 현재 시각을 얻는다.

3 (옮긴이) demon(악마)은 유닉스 계열 운영 체제에서 백그라운드로 실행되는 프로세스를 일컫는 용어인 daemon(원뜻은 그리스 신화에 나오는 반신반인의 존재)과 발음이 같다. 또한 BSD 계열 운영 체제인 FreeBSD에서는 악마 모양(*https://en.wikipedia.org/wiki/BSD_Daemon*) 마스코트(*https://www.freebsd.org/art/*)를 사용하고 있다.

4 (옮긴이) 1987년 썬은 AT&T와 합작 프로젝트로 SVR4를 발표한다. 이후 썬은 SunOS 코드를 BSD에서 SVR4 기반으로 변경하고 운영 체제 이름을 솔라리스로 바꾼다.

5 (옮긴이) 원문의 'tune a fish'와 발음이 비슷한 점을 이용한 말장난이다. FreeBSD tunefs 매뉴얼 페이지에는 아직 '참치' 문구가 남아 있다.

6 (옮긴이) 유닉스 계열 운영 체제에서 쓰이는 프린트 서비스다.

2. difftime()을 호출하여 현재 시각과 time_t의 최댓값이 몇 초 차이가 나는지 계산한다.
3. 계산한 값을 연, 월, 주, 일, 시간, 분으로 서식을 지정한 후 출력한다.

여러분이 예상한 시간보다 더 오래 걸리는가?

[프로그래밍 해답] 컴퓨터 시간

이 연습 문제의 결과는 PC와 유닉스 시스템에서 서로 다르며, time_t가 저장되는 방식에 따라서도 또 달라진다. 썬 시스템에서는 typedef long time_t다. 첫 번째 답은 다음과 같다.

```
#include <stdio.h>
#include <time.h>

int main() {
  time_t biggest = 0x7FFFFFFF;

  printf("biggest = %s \n", ctime(&biggest) );
  return 0;
}
```

결과는 다음과 같다.

```
biggest = Mon Jan 18 19:14:07 2038
```

그러나 이것은 정답이 아니다. 함수 ctime()이 반환하는 것은 현지 시각일 뿐 협정 세계시(co-ordinated universal time, 그리니치 표준시라고도 함)와는 다르다. 이 책을 쓰고 있는 캘리포니아주(州)는 런던보다 8시간이 늦다.

가장 큰 협정 세계시 시간값을 얻기 위해서는 gmtime() 함수를 사용해야 한다. 이 함수는 출력 가능한 문자열을 반환하지 않으므로 asctime() 함수를 사용하여 출력 가능한 문자열로 변환한다. 이 두 함수를 묶어 사용해 수정한 프로그램은 다음과 같다.

```
#include <stdio.h>
#include <time.h>

int main() {
  time_t biggest = 0x7FFFFFFF;

  printf("biggest = %s \n", asctime(gmtime(&biggest)) );
  return 0;
}
```

결과는 다음과 같다.

```
biggest = Tue Jan 19 03:14:07 2038
```

이렇게 앞의 결과에 8시간을 더 짜냈다!

하지만 아직 끝나지 않았다. 뉴질랜드 시각을 사용하고 2038년에 일광 절약 시간을 적용한다고 가정하면 13시간을 더 얻을 수 있다. 뉴질랜드는 남반구에 있기 때문에 1월이 일광 절약 시간에 해당한다. 뉴질랜드는 시간대와 관련하여 가장 동쪽에 위치하기 때문에 특정 날짜로 유발된 버그가 발생하는 첫 번째 국가라는 불편한 진실이 있다.

간단하게 보이는 것도 놀랍게 변형시키는 것이 소프트웨어다. 그러므로 날짜 프로그램을 한 번에 짤 수 있으리라 생각했어도 막상 프로그램을 작성하다 보면 쉽게 해결하지 못하는 자신을 발견했을 것이다.

1장

Expert C Programming

C가 지나온 길

C는 별난 데다 결함투성이지만 엄청난 성공을 거두었다.

— 데니스 리치(Dennis Ritchie)

C의 초창기

C 이야기는 역설적이게도 실패에서 시작된다. 1969년 운영 체제 구축을 위해 제너럴 일렉트릭, MIT, 벨 연구소가 만든 합작 회사의 멀틱스(Multics, Multiplexed Information and Computing Service) 프로젝트가 어려움에 빠졌다. 당초 계획은 빠르고 편리한 온라인 시스템을 제공하는 것이었는데, 목표 달성은커녕 사용할 가치가 없는 제품이 되어 버렸다. 우여곡절 끝에 개발 팀이 멀틱스를 어찌어찌 개발하기는 했지만, IBM OS/360과 똑같은 늪에 빠져 버렸다. 즉, 작은 하드웨어에서 돌리기에는 너무 큰 운영 체제를 만들어 버린 것이다. 멀틱스는 엔지니어링 문제의 해답으로 가득한 보물 창고였지만 결국 작은 것이 아름답다는 것을 보여 줄 기회를 C 언어에 제공했다.

벨 연구소 직원들은 멀틱스 프로젝트에서 철수하면서 다른 작업을 둘러보게 되는데, 그중 한 연구원인 켄 톰슨(Ken Thompson)이 다른 운영 체제를 연구하기 위해 벨 경영진에게 몇 가지 제안을 했다. 결과적으로 당시 제안했던 내용은 모두 거

절되었지만, 어쨌든 제안 결과를 기다리는 동안 톰슨과 동료 데니스 리치는 톰슨이 개발한 '우주여행(Space Travel: 우주선을 조종해 태양계를 여행하고 행성에 착륙하는 그래픽 시뮬레이션 프로그램)' 소프트웨어를 당시 대중적이지 않았던 시스템인 PDP-7에 이식하는 것을 즐기고 있었다. 동시에 톰슨은 PDP-7에 제공할 새로운 운영 체제에 대한 기초 연구를 진행했는데, 멀틱스보다 훨씬 간단하고 가볍게 만드는 데 집중했다. 모든 내용은 어셈블리 언어로 작성되었다. 브라이언 커니핸(Brian Kernighan)은 여기에다 그동안 멀틱스로부터 얻은 선택과 집중이라는 교훈을 추가하여 1970년에 비로소 '유닉스'라는 이름을 붙였다. 그림 1-1은 초기 C, 유닉스 및 관련 하드웨어를 보여 준다.

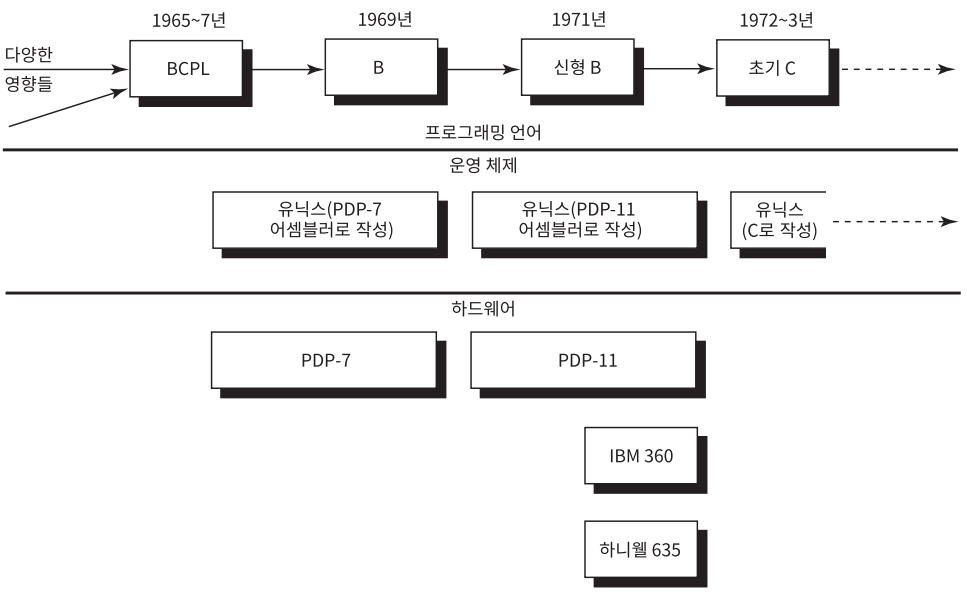

그림 1-1 초기 C, 유닉스 및 관련 하드웨어

'닭이 먼저냐? 달걀이 먼저냐?'로 빠질 수 있는 상황인데, 확실하게 유닉스는 C보다 먼저 나타났다(유닉스 시스템 시간이 1970년 1월 1일 이후부터 초 단위로 측정되는 이유는 유닉스가 이때 시작했기 때문이다). 어쨌든 가금류 논쟁은 그만하고 프로그래밍 이야기로 돌아가자. 유닉스를 어셈블러로 작성하는 작업은 대단히 불편했다. 자료 구조를 만드는 데도 시간이 많이 걸리고, 디버그하고 이해하기도 대단히 힘들었다. 그래서 톰슨은 고수준으로 구현된 프로그래밍 언어의 장점을 추가하

는 반면, 멀틱스에서 경험했던 PL/I[1]의 성능 및 복잡성 문제는 덜어 내고자 했다. 먼저 포트란(Fortran)으로 잠깐 시도했으나 실패로 끝나고, 다시 연구용 프로그래밍 언어 BCPL[2]을 단순화한 프로그래밍 언어 B를 만들어 시도했다. 따라서 B의 인터프리터는 PDP-7의 8KB 워드(word) 메모리에 맞춰졌을 것이다. 하지만 B 언어는 절대 성공할 수 없었다. 하드웨어 메모리는 인터프리터를 돌리기에는 충분했지만 컴파일러를 사용하기에는 부족했기 때문이다. 결과적으로 B 언어의 성능은 좋지 않았고, 유닉스 시스템 프로그래밍에 사용되지 못했다.

[프로그래밍 토막 지식] 컴파일러 제작자의 황금률: 성능이 (거의) 전부다

성능은 컴파일러의 거의 모든 것이다. 물론 중요한 오류 메시지, 충실한 설명, 제품 지원 등 다른 요소도 있지만 이러한 요소들은 사용자가 컴파일 속도를 중시하는 것에 비하면 새 발의 피다. 컴파일러 성능에는 런타임 성능(코드 실행 속도)과 컴파일 타임 성능(코드를 생성하는 데 걸리는 시간)이라는 두 측면이 있는데, 개발 중이거나 연구 목적으로 사용되는 것을 제외하고는 런타임 성능을 더 우선시한다.

컴파일러를 다양하게 최적화하면 컴파일 타임은 길어지지만, 그 덕분에 컴파일된 코드의 실행 시간은 훨씬 단축된다. 죽은(필요 없는) 코드 제거, 런타임 검사 생략 등 다른 최적화 기능을 사용하면 컴파일 타임과 실행 시간이 빨라지고 메모리 사용량도 줄어든다. 하지만 공격적인 최적화의 단점은 잘못된 결과를 사전에 알려 주지 못하는 위험 요소를 안고 있다는 점이다. 그러므로 최적화 도구는 안전한 변환만 수행하도록 극도로 주의를 기울이지만, 프로그래머가 잘못된 코드를 작성하면 엉뚱한 결과를 유발할 수 있다(예: 필요한 변수가 배열에 인접해 있다는 것을

1 PL/I을 배우고 사용하고 구현하는 것이 얼마나 어려웠는지는 어떤 프로그래머가 남긴 다음 구절에 잘 나타나 있다.

IBM에 PL/I이 있었네.
그 구문은 JOSS보다 나빴다네.
하지만 모든 곳에 사용되었으니
결국 완전히 망할 수밖에.

JOSS(JOHNNIAC Open Shop System)는 PL/I보다 앞서 나온 언어로 여기에서는 따로 설명하지 않겠다.
(옮긴이) JOSS 언어는 초기 대화형 프로그래밍 언어로 랜드(RAND)에서 만든 폰 노이만 아키텍처 기반 초기 컴퓨터인 조니악 시스템에서 사용하기 위해 1963년에 베타 버전이 개발되었고, 1964년에 조니악 시스템으로 상용화되었다. 나중에 베이식 등에 영향을 주었다.

2 "BCPL: A Tool for Compiler Writing and System Programming," Martin Richards, Proc. AFIPS Spring Joint Computer Conference, 34(1969), pp. 557-566. BCPL은 Before C Programming Language의 약자가 아니다. 그저 재미있는 우연의 일치일 뿐이다. 원래는 Basic Combined Programming Language의 약자로 여기에서 사용된 'basic'이 '단순함'을 지향하고 있다. BCPL은 영국의 런던 대학과 캠브리지 대학의 연구자들이 협업으로 개발했다. 멀틱스에서 BCPL이 사용되었다.

'안다'고 배열 범위를 넘어서는 영역을 참조하는 것 등).

이게 바로 성능이 전부가 아니라 '거의' 전부인 이유다. 정확한 결과를 얻지 못한다면 결과를 얼마나 빨리 얻는지는 중요하지 않다. 컴파일러 제작자는 일반적으로 컴파일러 옵션을 제공하므로 프로그래머는 필요한 최적화 옵션을 선택할 수 있다. 데니스 리치가 B 언어의 성능을 높인 '신형 B' 컴파일러를 만들 때까지 B 언어가 성공하지 못했다는 사실은 컴파일러 작성자를 위한 황금률이 얼마나 중요한지 잘 보여 준다.

B는 BCPL을 간소화하기 위해 일부 기능(중첩된 프로시저 및 일부 반복 구조)을 생략하고, 배열을 참조할 때는 포인터+오프셋으로 '분해'하는 아이디어를 적용했다. B 언어도 BCPL처럼 데이터 타입이 없는 형태를 유지했는데, 대상 기계의 워드[3]가 유일한 데이터 타입이었다. 톰슨은 여기에 ++와 -- 연산자를 고안하여 PDP-7의 B 컴파일러에 추가했다. PDP-11의 자동 증가/감소 주소 지정 모드 때문에 C 언어에서 ++와 -- 연산자가 추가됐다고 많이들 오해하고 있지만 사실이 아니다. 자동 증가와 감소는 PDP-11 하드웨어보다 먼저 나타났다. 그런데도 이런 오해가 생긴 이유는 문자열의 문자를 복사하는 C 코드 `*p++ = *s++;`를 PDP-11 어셈블러 코드 `movb (r0)+,(r1)+`로 효율적으로 컴파일할 수 있기 때문이다. 즉, 이 PDP-11 코드를 위해 문자열 복사 C 코드가 특별히 만들어졌다고 오해한 것이다.

1970년에 새롭게 도입된 PDP-11 시스템으로 개발 환경이 바뀌었을 때 데이터 타입이 없는 언어는 사용할 수 없다는 결론이 나왔다. PDP-11 프로세서는 여러 가지 크기의 데이터 타입을 하드웨어 차원에서 지원하는 것이 특징이었는데 B 언어로는 이를 표현할 방법이 없었다. 톰슨이 PDP-11 어셈블리 언어로 운영 체제를 다시 구현하는 데 문제가 생기자 데니스 리치는 PDP-11에 걸맞은 강력한 언어를 만드는 데 집중했으며, 마침내 여러 종류의 데이터 타입과 성능, 두 마리 토끼를 모두 잡을 수 있는 '신형 B'를 만들었다. '신형 B(이 이름은 빠르게 발전하여 'C'가 되었다)'는 인터프리트 방식이 아닌 컴파일 방식을 채용했고, 데이터 타입 시스템을 도입하여 사용하기 전에 변수 타입을 설명하도록 했다.

3 (옮긴이) 워드는 하나의 기계어 명령어나 연산을 통해 저장된 장치로부터 레지스터에 옮겨 놓을 수 있는 데이터 단위다. 메모리에서 레지스터로 데이터를 옮기거나 ALU(arithmetic logic unit)를 통해 데이터를 조작하거나 할 때, 하나의 명령어로 실행될 수 있는 데이터 처리 단위다. 32비트 CPU라면 워드는 32비트가 된다(출처: 위키백과).

C 언어와 함께했던 초창기 경험

데이터 타입 시스템은 새로운 PDP-11 하드웨어에서 사용하는 워드로부터 컴파일러 제작자가 우선 부동 소수점 수(`float`, `double`), 문자 등을 구분하도록 지원하기 위해 도입되었다. 이것은 파스칼(Pascal)과 같은 언어와 대조되는데, 원래 데이터 타입 시스템의 목적은 데이터 항목에 대한 유효한 연산을 제한하여 프로그래머를 보호하려는 것이다. 파스칼과 철학이 달랐던 C는 엄격한 데이터 타입 지정을 거부하고, 프로그래머가 원할 경우 다른 타입 객체 간에도 대입 연산이 가능하게 했다. 데이터 타입 시스템은 거의 사후 검토 방식으로 진행되었으며, 엄격하게 평가하거나 사용 가능성을 광범위하게 테스트하지 않았다. 한동안 C 프로그래머들은 '강한 타이핑(strong typing)'을 키보드를 조금 더 세게 치는 것으로 알고 있었다.

데이터 타입 시스템 외에도 C 컴파일러 제작자를 돕기 위해 C에 많은 기능이 추가되었다(당연히 처음 몇 년간은 C 컴파일러 제작자가 주요 고객이었기 때문이다). 컴파일러 제작자와 함께 발전한 것으로 보이는 C 언어의 특징은 다음과 같다.

- **배열은 1이 아닌 0부터 시작한다.** 사람들은 대부분 0이 아닌 1부터 세기 시작한다. 하지만 컴파일러 제작자는 오프셋 관점에서 생각하기 때문에 0부터 시작한다. 컴파일러 제작자가 아닌 사람에게 이러한 개념은 어렵게 느껴진다. 배열을 `a[100]`으로 정의했다면 `a[0]`에서 `a[99]`까지만 데이터를 저장할 수 있고 `a[100]`에는 저장할 수 없다.
- **C 언어의 기본 데이터 타입은 대상 하드웨어에 직접적인 영향을 받는다.** 예를 들어 포트란에서 사용하는 복소수 타입 같은 데이터 타입은 C 언어에 내장되지 않았다. 컴파일러 제작자는 하드웨어가 직접적으로 제공하지 않는 의미 체계(semantics)를 지원하기 위해 어떠한 노력도 기울일 필요가 없다. 그래서 C 언어는 하드웨어에서 제공될 때까지 부동 소수점 타입을 지원하지 않았다.
- **`auto` 키워드는 분명히 쓸모가 없다.** `auto` 키워드는 심벌 테이블(symbol table)에 항목을 채워야 하는 컴파일러 제작자에게만 의미가 있다. `auto` 키워드는 전역 정적(static) 할당이나 힙(heap)의 동적 할당과 달리 '이 저장소는 해당 블록에 진입할 때 자동으로 할당된다는 것'을 뜻한다. `auto` 키워드는 기본으로 가져오기 때문에 프로그래머에게 영향을 미치지 않는다.
- **표현식의 배열 이름이 포인터로 '축약'된다.** 배열을 포인터로 취급함으로써 작업

이 간단해진다. 그 덕분에 배열을 복합 객체로 다루기 위해 복잡한 메커니즘을 사용하지 않아도 되고, 배열을 함수로 전달하기 위해 배열의 모든 내용을 복사하는 비효율적인 상황도 겪을 필요가 없다. 그러나 배열과 포인터가 항상 같다고 생각해서는 큰코다친다. 이에 대해서는 4장에서 자세히 설명한다.

- **부동 소수점 표현식은 모든 곳에서 배정밀도 길이로 확장되었다.** 안시 C에서는 더 이상 사실이 아니지만, 원래 실수 상수는 언제나 `double`이며 `float` 변수는 모든 표현식에서 `double`로 항상 변환되었다. 명확하게 문서로 정리된 것은 아니지만, 그 이유는 PDP-11 부동 소수점 하드웨어와 관련이 있다. 우선 PDP-11 또는 백스(VAX)[4]에서 `float`를 `double`로 변환하는 것은 정말 저렴하다. 그저 한 워드만큼 0을 추가하면 된다. 다시 `float`로 변환하려면 `double`의 두 번째 워드를 버리면 된다. 다음으로 일부 PDP-11 부동 소수점 하드웨어에는 모드 비트가 있어서 모드 비트에 따라 단정밀도(single-precision) 또는 배정밀도(double-precision) 방식으로 부동 소수점 연산이 이루어졌다. 초기 유닉스 프로그램은 대부분 부동 소수점에 집중하지 않았기 때문에 컴파일러 제작자가 일일이 모드 비트를 추적하기보다는 그냥 모드 비트를 배정밀도로 두는 것이 더 쉬웠다.
- **중첩된 함수(다른 함수 안에 포함된 함수)를 사용하지 않는다.** 이렇게 하면 컴파일러가 단순해지고 C 프로그램의 런타임 구성이 약간 빨라진다. 정확한 메커니즘은 6장에서 설명한다.
- **`register` 키워드** 이 키워드는 '핫'한(자주 참조되는) 변수가 무엇인지에 대한 단서를 컴파일러 제작자에게 제공하여 자주 참조되는 변수를 레지스터에 보관하도록 만들었지만 결국 실수였다. 컴파일러 입장에서는 변수가 사용될 때마다 레지스터를 할당해 주는 것이, 변수 선언 후 존속 기간 내내 레지스터를 할당하는 것보다 더 좋은 코드를 만들어 낼 수 있다. `register` 키워드로 레지스터 할당에 대한 부담을 프로그래머에게 전가함으로써 컴파일러를 좀 더 단순하게 만든 것이다.

C 컴파일러 제작자의 편의를 위해 발명된 C 기능은 이 밖에도 수두룩하다. 하지만 이것을 꼭 나쁘게만 볼 것은 아니다. 이러한 기능들 덕에 언어가 대단히 단순해졌는데, 에이다(Ada)의 제네릭(generic)이나 태스킹, PL/I의 문자열 처리, C++의 템플

4 (옮긴이) virtual address extension의 약자로 DEC가 1970년대 중반에 개발한 명령어 집합 아키텍처

릿(template) 또는 다중 상속(inheritance)과 같은 복잡한 개념이 없어서 C 언어를 배우고 구현하기가 한결 쉬워졌고, 더 빠른 성능을 제공할 수 있었다.

다른 프로그래밍 언어와 달리 C 언어는 오랫동안 발전했고, 현재 형태에 도달하기까지 다양한 중간 과정을 거치며 성장했다. 즉, 다년간 실제 사용을 통해 검증된 언어로 발전했다. 최초의 C 컴파일러는 거의 반세기 전인 1972년에 등장했으며, 유닉스 시스템의 인기에 힘입어 C 언어 역시 함께 성장했다. 특히 C 언어는 하드웨어에서 직접 지원되는 저수준 처리에 중점을 두면서 속도와 이식성이 향상되어 유닉스가 널리 사용되는 데 결정적인 공헌을 했다.

표준 I/O 라이브러리 및 C 전처리기

C 컴파일러에서 제외된 기능이라도 어딘가에서는 필요하다. C 언어에서는 이러한 기능들을 런타임에서, 즉 애플리케이션 코드 또는 런타임 라이브러리에서 볼 수 있다. 다른 프로그래밍 언어의 컴파일러는 암암리에 런타임 지원을 호출하는 코드를 작성해 주기 때문에 프로그래머가 런타임 코드에 대해 신경 쓸 필요가 없지만, C 언어는 C 라이브러리의 거의 모든 루틴을 명시적으로 호출해야 한다. 즉, 동적 메모리 사용, 가변 크기 배열 프로그래밍, 배열 범위 테스트 등 필요한 모든 것을 프로그래머가 직접 수행해야 한다.

비슷한 이유로 I/O 역시 초기 C에서는 정의하지 않았고 그 대신 별도의 라이브러리 루틴을 제공했는데 훗날 실질적인 표준 기능으로 자리 잡았다. 마이크 레스크(Mike Lesk)[5]가 이식 가능한 I/O 라이브러리를 작성했고 1972년 기존 하드웨어 플랫폼 세 곳에 처음 탑재됐다. 하지만 실제 환경에서 막상 사용해 보니 성능이 기대에 미치지 못했고, 라이브러리를 가다듬고 최적화하는 과정을 반복한 후 결국 표준 I/O 라이브러리로 거듭났다.

비슷한 시기에 앨런 스나이더(Alan Snyder)의 제안에 따라 도입된 C 전처리기는 세 가지 주요 목적을 달성했다.

5 마이크는 나중에 다음과 같이 익살스럽고 역설적인 법칙을 제시했다. "매뉴얼을 가능한 한 짧게 만들 수 있도록 시스템을 설계해서 학습에 들어가는 노력을 최소화하라."(Datamation, November 1981, p.146) 하지만 '학습에 들어가는 노력을 최소화하라' 다음에 '푸하하!' 소리가 들리는 건 내 착각일까?