C Programming FAQs

번역 및 수정: 신성국 (Seong-Kook Shin)

September 18, 2015

This book is dedicated to C programmers everywhere.

모든 C 프로그래머들에게 이 책을 바칩니다.

Contents

Preface

Original Preface

At some point in 1979, I heard a lot of people talking about this relatively new language, C, and the book that had just come out about it. I bought a copy of K&R, otherwise known as *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, but it sat on my shelf for a while because I didn't have an immediate need for it (besides which I was busy being a college freshman at the time.) It proved in the end to be an auspicious purchase, though, because when I finally did take it up, I never put it down: I've been programming in C ever since.

In 1983, I came across the Usenet newsgroup net.lang.c, which was (and its successor comp.lang.c still is) an excellent place to learn a lot more about C, to find out what questions everyone else is having about C, and to discover that you may not know all there is to know about C after all. It seems that C, despite its apparent simplicity, has a number of decidedly nonobvious aspects, and certain questions come up over and over again. This book is a collection of some of those questions, with answers, based on the Frequently Asked Questions ("FAQ") list I began posting to comp.lang.c in May 1990.

I hasten to add, however, that this book is not intended as a critique or "hatched job" on the C language. It is all too easy to blame a language (or any tool) for the difficulties its users encounter with it or to claim that a properly designed tool "ought" to prevent its users from misusing it. It would therefore be easy to regard a book like this, with its long lists of misuses, as a litany of woes attempting to show that the language is hopelessly deficient. Nothing could be farther from the case.

I would never have learned enough about C to be able to write this book,

and I would not be attempting to make C more pleasant for others to use by writing this book now, if I did not think that C is a great language or if I did not enjoy programming in it. I *do* like C, and one of the reasons I teach classes in it and spend time participating in discussion about in on the Internet is that I would like to discover which aspects of C (or of programming in general) are difficult to learn or keep people from being able to program efficiently and effectively. This book represents some of what I've learned: These questions are certainly some of the ones people have the most trouble with, and the answers have been refined over several years in an attempt to ensure that people don't have too much trouble with *them*.

A reader will certainly have trouble if there are any errors in these answers, and although the reviewers and I have worked hard to eliminate them, it can be as difficult to eradicate the last error from a large manuscript as it is to stamp out the last bug in a program. I will appreciate any corrections or suggestions sent to me in care of the publisher or at the e-mail address given, and I would like to offer the customary $1.00 reward to the first finder of any error. If you have access to the Internet, you can check for an errata list (and a scorecard of the finders) at the ftp and http addresses mentioned in question 20.40.

As I hope I've made clear, this book is not a critique of the C programming langauge, nor is it a critique of the book from which I first learned C or of that book's authors. I didn't just learn C from K&R; I also learned a lot of programming. As I contemplate my own contribution to the C programming literature, my only regret is that the present book does not live up to a nice observation made in the second edition of K&R, namely, that "C is not a big language, and it is not well served by a big book." I hope that those who most deeply appreciate C's brevity and precision (and that of K&R) will not be too offended by the fact that this book says some things over and over and over or in three slightly different ways.

Although my name is on the cover, there are a lot of people behind this book, and it's difficult to know where to start handling out acknowledgements. In a sense, every one of comp.lang.c's readers (today estimated at 320,000) is a contributor: The FAQ list behind this book was written for comp.lang.c first, and this book retains the flavor of a good comp.lang.c discussion.

This book also retains, I hope, the philosophy of correct C programming that I began learning when I started reading net.lang.c. Therefore, I shall first ac-

knowledge the posters who stand out in my mind as having most clearly and consistently articulated that philosophy: Doug Gwyn, Guy Harris, Karl Heuer, Henry Spencer, and Chris Torek. These gentlemen have displayed remarkable patience over the years, answering endless questions with generocity and wisdom. I was the one who stuct his neck out and started writing the Frequent questions down, but I would hate to give the impression that the answers are somehow mine. I was once the student (I believe it was Guy who answered my posting asking essentially the present volume's question 5.10), and I owe a real debt to the masters who went before me. This book is theirs as much as mine, though I retain title to any inadequacies or mistakes I've made in the presentation.

The former on-line FAQ list grew by a factor of three in the process of becoming this book, and its growth was a bit rapid and awkward at times. Mark Brader, Vinit Carpenter, Stephen Clamage, Jutta Degener, Doug Gwyn, Karl Heuer, Joseph Kent, and George Leach read proposals or complete drafts and helped to exert some control over the process; I thank them for their many careful suggestions and corrections. Their efforts grew out of a shared wish to improve the overall understanding of C in the programming community. I appreciate their dedication.

Three of those reviewers have also been long-time contributors to the on-line FAQ list. I thank Jutta Degener and Karl Heuer for their help over the years, and I especially thank Mark Brader, who has been my most persistent critic ever since I first began posting the comp.lang.c FAQ list five years ago. I don't know how he has had the stamina to make as many suggestions and corrections as he has and to overcome my continuing stubborn refusal to agree with some of them, even though (as I eventually understood) they really *were* improvements. You can thank Mark for the form of many of this book's explanations and blame me for mangling any of them.

Additional assorted thanks: to Susan Cyr for the cover art; to Bob Dinse and Eskimo North for providing the network access that is particularly vital to a project like this; to Bob Holland for providing the computer on which I've done most of the writing; to Pete Keleher for the Alpha text editor; to the University of Washington Mathematics Research and Engineering libraries for access to their collections; and to the University of Washington Oceanoraphy department for letting me borrow their tape drivers to access my dusty old archives of Usenet postings.

Thanks to Tanmoy Bhattacharya for the example in question 11.10, to Arjan Kenter for the code in question 13.7, to Tomohiko Sakamoto for the code in question 20.31, and to Roger Miller for the line in question 11.35.

Thanks to all these people, all over the world, who have contributed to the FAQ list in various ways by offering suggestions, corrections, constructive criticism, or other support: Jamshid Afshar, David Anderson, Tanner Andrews, Sudheer Apte, Joseph Arceneaux, Randall Atkinson, Rick Beem, Peter Bennett, Wayne Berke, Dan Bernstein, John Bickers, Gary Blaine, Yuan Bo, Dave Boutcher, Michael Bresnahan, Vincent Broman, Stan Brown, Joe Buehler, Kimberley Burchett, Gordon Burditt, Burkhard Burow, Conor P. Cahill, D'Arcy J.M. Cain, Christopher Calabrese, Ian Cargill, Paul Carter, Mike Chambers, Billy Chambless, Franklin Chen, Jonathan Chen, Raymond Chen, Richard Cheung, Ken Corbin, Ian Cottam, Russ Cox, Jonathan Coxhead, Lee Crawford, Steve Dahmer, Andrew Daviel, James Davies, John E. Davis, Ken Delong, Norm Diamond, Jeff Dunlop, Ray Dunn, Stephen M. Dunn, Michael J. Eager, Scott Ehrlich, Arno Eigenwillig, Dave Eisen, Bjorn Engsig, David Evans, Clive D.W. Feather, Dominic Feeley, Simao Ferraz, Chris Flatters, Rod Flores, Alexander Forst, Steve Fosdick, Jeff Francis, Tom Gambill, Dave Gillespie, Samuel Goldstein, Tim Goodwin, Alasdair Grant, Ron Guilmette, ... I have tried to keep track of everyone whose suggestions I have used, but I fear I've probably overlooked a few; my apologies to anyone whose name should be on this list but isn't.

Finally, I'd like to thank my editor at Addison-Wesley, Debbie Lafferty, for tapping me on the electronic shoulder one day and asking if I might be interested in writing this book. I was, and you now hold it, and I hope that it may help to make C programming as pleasant for you as it is for me.

Seattle, Washington
July, 1995

Steve Summit
summit@aw.com

수는 없습니다.

특정 주제들에 대한 질문은 이 뉴스 그룹에 자주 올라옵니다. 이들은 매우 좋은 질문이며, 어떤 것들은 즉각 답변하기 매우 어려운 난해한 것들입니다. 하지만 이러한 질문들이 자주 올라온다면 네트워크에 과부하가 걸릴 수도 있고, 독자들은 매번 같은 답변을 읽느라 번거로울 것이며, 잘못된 답변에 대한 지겨운 정정 작업이 발생할지도 모릅니다.

이 문서는 이러한 일반적인 문제들에 대한 정확하고 간결한 답변을 제시하고자 매달 게시되며, 넷 토론(net discussion)이 이러한 기본적인 문제에 대한 지속적인 반복없이 좀 더 생산적인 질문에 전념하는데에 그 목적이 있습니다.

아무리 뉴스 그룹의 글이 좋다고 하더라도 언어 참고서(language reference manual)나 튜토리얼(tutorial)을 주의 깊게 읽는 것만 못합니다. C 언어에 관심이 있고, 이 뉴스 그룹에 참여하길 원한다면 당연히 이러한 책들을 반복하여 읽고 연구하는 것에도 관심을 가져야 할 것입니다. 어떤 C 책들과 컴파일러 매뉴얼들은 불행하게도 완전하지 않습니다. A few even perpetuate some of the myths which this article attempts to refute. 유명한 여러 책들에 대해서는 이 글의 '참고문헌란'에 나열되어 있습니다. 질문 18.9와 18.10을 참고하시기 바랍니다. 또한 많은 질문들과 답변들이 이러한 책들과 연결되어 있습니다. 관심있는 독자들은 참고하시기 바랍니다.

여러분이 이 문서에 나와 있지 않는, C 언어에 대한 질문이 있다면 넷 그룹에 게시하기 전에 이러한 참고 서적들에서 먼저 답을 찾아보거나, 주변의 동료들에게 도움을 청해보시기 바랍니다. 넷 그룹에는 답변하기를 즐기는 많은 사람들이 있지만, 증가되고 있는 질문 갯수와 같은 질문에 대한 반복적인 답변의 분량을 생각하면 그 사람들에게 너무나도 가혹한 일입니다. 또한 이 문서에 대한 질문이나 제안 사항이 있다면 전자 우편을 사용하여 알려주시기 바랍니다. 이 문서는 넷 트래픽을 감소시키기 위한 것이지, 증가시키기 위한 것이 아니기 때문입니다.

이 문서는 자주 물어보는 질문들만 나열한 것이 아니라 각 질문에 대한 답변들도 포함하고 있습니다. 여러분이 이들 답변에 대해 이미 알고 있더라도 한번 죽 훑어보는 것을 추천합니다. 그리고 이러한 질문이 뉴스 그룹에 게시되더라도 답변하기 위해 시간을 낭비할 필요는 없습니다. (그러나, 이 문서는 매우 방대하고, 무거운 주제를 다루기 때문에 뉴스그룹의 모든 사람들이 이 문서를 자세히 읽고 질문할 것이라고 기대하지는 마시기 바랍니다. 그리고 이러한 사람들에게 욕설을 한다거나 하는 행동은 자제해 주시기 바랍니다)

이 문서는 1999년 2월 7일에 마지막으로 수정되었습니다. 만약 여러분이 인쇄된 형식의 문서를 보거나, CD-ROM 또는 아카이브 사이트(archive site)에서 얻은 문서를 보신다면 먼저 최신의 문서를 다음 사이트에서 얻으시기 바랍니다.

`http://www.eskimo.com/~scs/C-faq/top.html`
`http://www.faqs.org/faqs/`

또는 질문 20.40에 언급된 FTP 사이트를 참고하시기 바랍니다. 그리고 이 목록은 자주 수정되기 때문에 질문 번호가 오래된 목록과 새 목록이 서로 다를 수 있습니다. 따라서 번호로 목록을 찾을 때 주의하시기 바랍니다.

이 문서는 자유로이 배포되는 것이므로 이 문서를 얻기 위해 어떠한 추가 비용을 지불할 필요는 없습니다.

이 문서의 다른 형식도 얻을 수 있습니다. 이전 문서와 새 문서의 차이점만을 명시한 짧은 형식도 준비되어 있으며, 하이퍼텍스트(hypertext) 버전도 위에 언급한 URL에 준비되어 있습니다. 마지막으로, 인쇄된 형식을 선호하는 사람들을 위해 책으로도 출판되어 있습니다. (책의 경우 좀 더 자세한 답변과 더 많은 질문과 답변이 수록되어 있습니다). 이 책은 Addison-Wesley사에서 출판되었으며 ISBN은 0-201-84519-9입니다.

이 문서는 계속 수정되고 있습니다. 여러분의 참여를 환영합니다. 여러분의 제안 사항을 `scs@eskimo.com`으로 보내주시기 바랍니다.

답변된 질문들은 다음 주제별로 나뉘어져 있습니다.

1. 선언과 초기화
2. 구조체, Unions, 열거형(enumerations)
3. 식(Expressions)
4. 포인터
5. 널(null) 포인터
6. 배열과 포인터
7. 메모리 할당(allocation)
8. 문자와 문자열
9. Boolean 식과 변수
10. C 전처리기(preprocessor)
11. ANSI/ISO C 표준(standard)
12. 표준 입출력(stdio)
13. 라이브러리 함수

14. 실수 (floating point)
15. 가변 인자 리스트(Variable-Length Argument Lists)
16. 이상한 문제들
17. 스타일
18. 툴과 기타 자료
19. 시스템 의존성
20. 기타(miscellaneous)
21. 참고문헌
22. Acknowledgements

각 단원의 질문 번호는 이 문서에서 연속적이지 않을 수 있습니다. 왜냐하면 이 질문들은 책에서 뽑은 것이기 때문에, 책과 같은 번호를 유지하고자 하기 때문입니다.)

About Translation

이 문서는 Steve Summit씨의 'C Programming FAQs'를 한국어로 번역한 것입니다. 이 한국어판 문서의 내용은 번역자에게 그 저작권이 있으며, 개인적인 목적으로는 자유로이 사용될 수 있지만 허가없이 출판되거나 홍보용, 상업용 등의 목적으로는 사용될 수 없습니다. 또한 (개인적인 목적을 포함) 어떠한 이유에서라도 이 문서의 일부만이 사용되는 것을 금합니다.

이 문서에 대한 조언, 비판, 정정은 언제든지 환영하며 이러한 사항은 아래 e-mail 주소로 보내시기 바랍니다:

```
cinsky at gmail.com
```

위의 주소에 보낼 수 없다면 다음 주소[1]로 보내시기 바랍니다:

```
tanarrian at yahoo dot co dot kr
```

물론 위에서 제공한 모든 e-mail 주소에서 'at'은 '@'로, 'dot'은 '.'로 바꾸어야 합니다. (스팸 메일을 방지하기 위한 조치이므로 양해바랍니다.) 번역자의 홈 페이지를 방문하면 항상 최신의 연락처를 얻을 수 있습니다:

```
http://www.cinsk.org/cfaqs/
```

Translation

이 문서는 Steve씨의 원문의 느낌을 그대로 전달하는 것을 목표로 하고 있습니다. 따라서 번역하기 곤란하거나, 우리말로 마땅한 단어가 없는 것은 영어를 그대로 쓰고 있습니다.

이 글에서 Unix, UNIX 등은 모두 같은 뜻이며 특정 UNIX 시스템이 아닌 일반적인(general) UNIX 시스템을 의미하는 단어입니다.

역자는 저자에게서 질문 번호를 바꾸지 않은 한, 새로운 질문/답변을 추가할 수 있다는 허락을 얻었습니다. 질문 번호가 숫자가 아닌, 알파벳으로 된 질문은 역자가 추가한 질문입니다.

Current Status – Very Important

오리지널 Steve씨의 원문은 현재, [C99] 표준이 완벽하게 제정되기 전의 상태에 쓰여졌습니다. 역자가 추가한 [C99] 표준에 관한 것은 현재 reference로 [C99]를 사용

[1] 첫 판에서 썼던 e-mail 주소인 `progman@pcrc.hongik.ac.kr`은 더이상 쓰지 않습니다.

합니다. Steve씨가 쓴 원문은 현재 [C9X]를 씁니다. [C9X]는 이 글이 개정되면서 [C99]로 바뀔 예정입니다.

오리지널 Steve씨의 출판된 책은 [ANSI]에 대한 reference가 많이 나와 있습니다만, 온라인 버전은 [ANSI] reference가 제공되지 않습니다. 왜냐하면 [ANSI]는 더 이상 공식 C 표준이 아니기 때문입니다. (질문 11.1을 참고하기 바랍니다.) 그러나 편의상 모든 질문에 책에서 얻은 [ANSI] reference도 추가합니다. 최신 정보를 얻으실 분들은 더 이상 [ANSI]나 [C89], [C9X], [ANSI Rationale], 그리고 [H&S]를 참고하면 안됩니다. (이들은 모두 구식의 C 표준을 따르고 있기 때문에, 최신 C 표준인 [C99]와 다른 부분이 있습니다.)

최신 C99에서 제공하는 여러가지 추가 기능에 대해서는 아직 이 글이 지원하지 않습니다. 여기에는 variable-length array, designator (structure) 등이 포함되며 곧 추가할 예정입니다. 또한 좀 더 자세한, 최신의 정보를 얻고 싶다면, 꼭 아래 site를 방문해서 최신의 정보를 얻기 바랍니다.

`http://www.cinsk.org/cfaqs/`

Acknowledgements

이 글을 만드는 데 도움을 주신 모든 분들께 감사를 드립니다. Steve Summit씨와 마찬가지로, 저도 Usenet의 많은 분들에게 도움을 받았습니다. 특히 `comp.lang.c`, `comp.unix.programmer` 뉴스 그룹의 모든 분들께 감사를 드립니다. 마찬가지로, 여러가지 정보를 얻을 수 있었던, KLDP BBS[2]의 모든 분들께도 감사드립니다.

문서 작성에 쓰인 LaTeX에 대한 도움을 주신 KTUG[3]의 모든 분께도 감사드립니다.

`lovewar`님과, Hyun-Kyung Lee님이 각각 여러 오타를 수정해 주셨습니다.

Yong-Uk Kim님, 김성훈(niky)님, 윤희수(cppig1995)님이 각각 본문에서 한 가지 문제점을 지적해 주셨습니다.

임성훈(wariua)님이 약 80여개의 오타를 지적해 주셨습니다.

아울러 혹시라도 제가 잊고 빠트린 분이 있다면 꼭 연락 주시기 바랍니다.

[2]Korean Linux Documentation Project, `http://bbs.kldp.org/`
[3]Korean TeX Users Group, `http://www.ktug.or.kr/`

Chapter 1

Declarations and Initializations

C 언어의 선언 문법은 (declaration syntax) 그 자체가 하나의 프로그래밍 언어라고 할 수 있습니다. 선언은 다음과 같이 여러 부분으로 구성되어 (꼭 모든 부분을 다 가져야 할 필요는 없습니다.) 있습니다: storage class, base type, type qualifier, 그리고 declarator (declarator는 initializer를 포함할 수 있습니다.) 각 declarator는 새 identifier를 선언하는 것 이외에, identifier가 배열인지, 포인터인지, 함수인지, 또는 어떤 복잡한 타입인지를 알려줍니다. 따라서 선언이 실제 identifier가 어떻게 쓰일 것인지를 (declaration mimics use) 알려 줍니다 (질문 1.21은 이 'declaration mimics use' 관계를 자세하게 다룹니다.)

1.1 Basic Type

프로그래머들은 종종 C 언어가 충분히 저수준 언어이면서도, C 언어의 type system이 상당한 수준으로 추상화되어 있다는 것에 놀라곤 합니다; 기본 타입들의 크기와 표현 방법이 언어 자체에 정확히 정의되어 있지 않습니다.

Q 1.1 어떤 타입의 정수를 쓸 것인지 어떻게 결정하죠?

Answer 큰 값 (32,767 이상이거나 -32,767 이하)이 필요한 경우, `long`을 쓰기 바랍니다. 차지하는 공간이 매우 중요하다면 (큰 배열이나 많은 구조체를 쓸 경우), `short`를 쓰기 바랍니다. 이런 경우가 아니라면 `int`를 쓰는 것이 가장 좋습니다. 오버플로우 문제가 중요시되고 음수가 필요하지 않는 경우라면, 또는

비트를 다룰 때 부호 확장[1]에서 문제가 발생하는 것을 원치 않는다면 적절한 크기를 가진 형의 `unsigned` 형을 쓰는 것도 좋습니다. (하지만 수식에서 `signed` 형과 `unsigned` 형을 섞어 쓰는 것은 좋지 않습니다.)

문자 타입도 (특히 `unsigned char`) "작은" 정수타입으로 쓰일 수 있지만, 예상치 못한 부호 확장이나 코드의 크기를 증가시킬 수 있기 때문에 바람직하지 않습니다. (`unsigned char` 타입을 쓰는 것이 도움이 될 경우도 있습니다; 관련된 문제는 질문 12.1를 참고하기 바랍니다.)

비슷한 크기/빠르기 문제가 `float`과 `double`에서 발생할 수 있습니다. 변수의 주소가 필요하고 어떤 특별한 타입이 필요한 경우라면 위의 규칙들은 모두 적용되지 않습니다.

C 언어에서 각각의 type들이 정확한 크기를 가지도록 정의되어 있다고 착각하기 쉬운데, 사실은 그렇지 않습니다. C 언어에서 정의하고 있는 것은 다음과 같습니다:

- `char` type은 127 이상을 저장할 수 있다;
- `short int` type과 `int` type은 32,767까지 저장할 수 있다;
- `long int`는 2,147,483,647까지 저장할 수 있다;
- 따라서 다음 규칙을 적용할 수 있다:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <=
  sizeof(long) <= sizeof(long long)
```

이 규칙은 `char`가 적어도 8 bit가 되어야 한다는 것과, `short int`와 `int`는 적어도 16 bit여야 한다는 것과, `long int`는 적어도 32 bit가 되어야 한다는 것을 뜻합니다. (각각 type의 signed와 unsigned version은 같은 크기를 가진다고 보장되어 있습니다.) ANSI C에서 특정 machine에서 각각 type의 최소값과 최대값은 `<limits.h>`에 정의되어 있으며, 요약하면 다음과 같습니다:

Base type	Min. size (bits)	Min. value (signed)	Max. value (signed)	Max. value (unsigned)
`char`	8	-127	127	255
`short`	16	-32,767	32,767	65,535
`int`	16	-32,767	32,767	65,535
`long`	32	-2,147,483,647	2,147,483,647	4,294,967,295

[1] sign-extension

이 표는 표준이 보장하는 최소값들을 보여줍니다. 많은 implementation이 이보다 더 큰 값을 제공하지만, portable한 프로그램을 만들고 싶다면 이러한 것에 의존해서는 안됩니다.

어떠한 이유에서 **정확한** 크기가 필요한 경우라면 — 이런 경우로는 외부 저장 배치(externally-imposed stroage layout)가 필요한 경우를 들 수 있습니다. 질문 20.5를 참고하기 바랍니다 — 적절한 `typedef`를 쓰시기 바랍니다. (아래 Note 참고)

References [K&R1] § 2.2 p. 34;
[K&R2] § 2.2 p. 36, § A4.2 pp. 195–6, § B11 p. 257
[C89] § 5.2.4.2.1, § 6.1.2.5
[H&S] § 5.1,5.2 pp. 110–114

Note C99 표준은 새 정수 타입인 `long long int`를 추가했습니다. 이 타입은 최소 64 bit이며, 다음과 같은 특징을 지닙니다.

Base type	long long
Min. size (bits)	64
Min. value (signed)	-9,223,372,036,854,775,807
Max. value (signed)	9,223,372,036,854,775,807
Max. value (unsigned)	18,446,744,073,709,551,615

또한, 정확한 크기를 요구하는 정수 타입을 선언하기 위해, C99 표준은 `<stdint.h>`를 통해 여러 가지 정수 타입을 제공합니다. 여기에 관한 것은 11.I를 참고하기 바랍니다.

References [C99] § 5.2.4.2.1 pp. 22–23, § 6.2.5

Q 1.2 왜 표준에서는 각 type의 크기를 정확히 정의하지 않나요?

Answer 다른 high-level 언어에 비해 C 언어가 확실히 저 수준 언어이기는 하지만, object의 정확한 크기는 implementation이 결정할 문제입니다. (C 언어에서 여러분이 bit의 갯수를 지정할 수 있는 유일한 곳은 structure 안에서 bitfield를 쓸 때입니다.; 질문 2.25와 2.26을 참고하기 바랍니다.) 대부분의 프로그램에서는 크기를 정확히 지정할 필요가 없습니다; 크기를 정확히 지정하는 많은 프로그램들은 지정하지 않도록 프로그램을 작성했을 때, 더 낫습니다.

`int` type은 컴퓨터의 가장 자연스러운 word size를 나타내는 것으로 알려져 있으며, 대부분의 정수를 저장할 때 가장 적당한 type입니다. 질문 1.1의 guideline을 보기 바랍니다; 덧붙여 질문 12.42, 20.5도 참고하시기 바랍니다.

Note C99에서 새로 제공하는 정수 타입들은 개발자가 원하는 크기를 만족시킬 수 있습니다. 덧붙여 질문 11.I도 참고하시기 바랍니다.

Q 1.3 C 언어가 size를 정확히 정의하지 않기 때문에, 저는 `int16`, `int32`와 같은 `typedef` 이름을 씁니다. 그래서 컴퓨터에 따라서 `int`, `short`, `long` 등을 써서 만듭니다. 이 방법이 모든 문제를 해결해 줄 수 있을 것 같은데, 맞습니까?

Answer 정확히 size를 제어할 필요가 있다면, 좋은 방법입니다. 그러나 다음과 같은 사항을 주의해야 합니다:

- 정확한 크기 제어가 불가능할 수도 있습니다 (예를 들면, 대부분의 36-bit machine)
- `int16`이나 `int32`를 쓰는 목적이 "적어도 이 정도의 크기를 보장"하는 것이라면 전혀 쓸모가 없습니다. 왜냐하면 `int`와 `long` type의 정의 자체가 각각 "적어도 16 bit", "적어도 32 bit"를 뜻하기 때문입니다.
- Typedef는 byte-order 문제를 해결해 주지 못합니다. (예를 들어 여러분이 data를 교환(interchange)하려 하거나, 외부적으로 고정된 저장 형식에 맞추려고[2] 하는 경우)

덧붙여 질문 10.16, 20.5도 참고하시기 바랍니다.

References [C89] § 7.18.1.1

Note C99에 따르면 `<inttype.h>`를 포함시켰을 때, 정확한 크기를 갖는 데이터 타입을 위한 `int16_t`, `int32_t`등을 쓸 수 있습니다. 자세한 것은 질문 11.I를 참고하기 바랍니다.

표준에 따라 정확히 말하면, `int_16_t`, `int32_t`등이 정의된 헤더 파일은 `<stdint.h>`에 정의되어 있습니다. 다시, 표준에 따르면, `<inttype.h>`는 `<stdint.h>`를 포함하게 되며, 부가적인 사항들을 제공합니다.

References [C99] § 7.18

Q 1.4 64-bit를 지원하는 컴퓨터에서 64-bit 타입을 쓸 수 있는 방법이 있나요?

Answer 골치아픈 문제입니다. 다음과 같이 세 가지 방식으로 이 문제를 생각할 수 있습니다.

[2] conforming to externally imposed storage layouts

- 현존하는 16, 32-bit 시스템에서 세 정수 타입에서 (`short`, `int`, `long`) 두 타입이 일반적으로 같은 크기를 가집니다. 64-bit 시스템에서는 세 타입이 모두 다른 크기를 가질 수 있습니다. 따라서 어떤 vendor들은 64-bit `long int`를 제공합니다.
- 아쉽게도, 현존하는 많은 코드들이 `int`와 `long`이 같다고 예상하고 쓰여져 있거나, 둘 중 하나가 32 bit라고 가정한 상태에서 쓰여져 있습니다. 이러한 코드를 그대로 유지시킬 수 있게 어떤 vendor들은 새로운, 비표준인 64-bit `long long` (`__longlong`, 또는 `__very long`등) 타입을 지원합니다.
- 64-bit 시스템에서 `int`가 64 bit이어야 한다는 논의도 있었습니다. 왜냐하면 보통 `int`는 시스템이 표현하는 가장 자연스러운 word 크기를 가지기 때문입니다. ("the machine's natual word size")

따라서, 64-bit 타입을 쓰고자 하는 개발자는 적절한 typedef를 써서 코드를 작성해야 합니다. (또 이식성이 높은 코드를 만들기 위해서, 16, 32-bit 시스템을 위해 64-bit를 일일히 수동으로 처리할 수 있는 코드도 만들어야 할 것입니다.) Vendor들은 또, "정확히 64 bit인 타입"보다 (현재 C 표준에 존재하지 않는) "적어도 64 bit 이상인 타입"을 소개해야 합니다.

Note [ANSI] § F.5.6
[C89] § G.5.6

Answer 다가오는 C 표준(C9X)에서는 `long long` 타입이 적어도 64 비트 이상이 될 것을 명시하고 있습니다. 그리고 이 타입은 이미 여러 컴파일러에 의해 지원되고 있습니다 (어떠한 컴파일러는 `__longlong` 타입으로 지원합니다.) 또 대부분의 컴파일러들은 `short int`를 16 비트로, `int`를 32 비트로, `long int`를 64 비트로 지원하고 있으며, 이와 다르게 할 이유가 없습니다.

질문 18.15d를 참고하시기 바랍니다.

References [C9X] § 5.2.4.2.1, § 6.1.2.5.

Note 최신 C 표준, C99에서는 적어도 64 bit 이상인, `long long` 타입을 지원합니다. 질문 1.1, 1.3을 참고하기 바랍니다.

References [C99] § 5.2.4.2.1 pp. 22–23, § 6.2.5

1.2 Pointer Declarations

포인터에 관한 것은 Chapter 4부터 Chapter 7까지 설명되어 있지만, 여기에서는 선언에 관련된 것만 다룹니다.

Q 1.5 다음 선언에서 무엇이 잘못되었나요?

```
char *p1, p2;
```

p2를 쓰려고 할 때 에러가 납니다.

Answer 위의 선언에서 잘못된 점은 없습니다 — 여러분이 원하는 작업을 해 주지 못한다는 것을 제외하면 말입니다. Pointer 선언에서 *는 'base type'의 일부분이 아닙니다; *는 선언할 이름으로 구성된 *declarator*의 일부분입니다 (질문 1.21 참고). 선언을 쓸 경우에 공백 문자의 구분은 의미가 없습니다. 따라서 첫번째 declarator는 "* p1"이며, *를 포함하고 있기 때문에, p1은 'a pointer to char', 즉 char를 가리키는 pointer가 됩니다. 그러나 p2의 declarator는 p2 이외에 다른 것을 가지고 있지 않으므로, (여러분이 원하는 바가 아니겠지만) 간단히 char type이 됩니다. 한 선언에서 두 개의 pointer를 선언하려면, 다음과 같이 해야 합니다:

```
char *p1, *p2;
```

*가 declarator의 일부분이기 때문에, 위에 쓴 것처럼 공백 문자를 쓰는 것이 좋습니다; char*와 같이 쓰는 것은 실수를 내기 쉬우며, 혼동을 가져올 수 있습니다.

덧붙여 질문 1.13도 참고하시기 바랍니다.

Q 1.6 Pointer를 선언하고 그 pointer가 어떤 공간을 할당하려고 했으나, 제대로 동작하지 않습니다. 아래 코드에 어떤 문제가 있습니까?

```
char *p;
*p = malloc(10);
```

Answer 여러분이 선언한 pointer는 p이지 *p가 아닙니다. 질문 4.2를 참고하기 바랍니다.

1.3 Declaration Style

변수나 함수를 선언하는 것은 단순히 컴파일러를 기쁘게? 하기 위한 것이 아닙니다; it also injects useful order into a programming project. When declarations are arranged appropriately within a project, mismatches and other difficulties can be more easily avoided, and the compiler can more accurately catch any error that do occur.

Q 1.7 전역(global) 함수와 변수를 선언 또는 정의하는 가장 좋은 방법이 무엇일까요?

Answer 먼저 한 "global" 변수는 (여러 translation unit에서) 여러 개의 선언을 가질 수 있지만, 반드시 하나의 정의를 가져야 합니다. Global 변수에서 정의는 공간을 할당하고, 필요하다면 초기값을 지정하는 일종의 선언입니다. 함수에서 선언은 function body를 제공하는 일종의 선언입니다. 아래는 선언의 예입니다:

```
extern int i;

extern int f();
```

(`extern` keyword는 함수 선언에서 option입니다; 질문 1.11을 참고하기 바랍니다.)

아래는 정의의 예입니다:

```
int i = 0;

int f()
{
  return 1;
}
```

여러 소스 파일에서 변수나 함수를 공유할 필요가 있다면, 당연히 여러분은 모든 변수와 함수를 일관되게(consistent) 만들어야 합니다. 가장 좋은 방법은 각 정의를 관련된 `.c` 파일에 저장하고, external 선언을 헤더 파일("`.h`")에 두는 것입니다. 그리고 선언이 필요한 곳에서는 `#include`를 써서 포함시키면 됩니다. 정의를 포함하는 `.c` 파일에도 같은 헤더 파일을 포함시켜야 컴파일러가 선언과 정의가 일치하는지 검사해 줍니다.

이 규칙은 매우 portability가 높은 방법입니다; 이는 ANSI C 표준의 요구에도 부합하며, ANSI 이전의 컴파일러와 링커에서도 잘 동작합니다 (UNIX 컴파일러와 링커는 최대 하나가 초기화된다는 조건 아래에 여러 개의 선언을 가능케 하는 "common model"을 지원합니다; 이는 ANSI 표준에 의해 "common extension"으로 언급되어 있지만, 'pun[3]'은 아닙니다. 어떤 이상한 시스템에서는 외부 선언과 정의를 구별하기 위해서 반드시 초기값을 필요로 하기도 합니다.)

[3]발음은 같으나 뜻이 다른 말

한 헤더 파일에 하나의 선언만 나오도록 하기 위해 다음과 같은 전처리기 트릭을 쓸 수도 있습니다:

```
DEFINE(int, i);
```

그리고 어떤 매크로의 설정에 따라 이 줄이 선언이나 정의가 되도록 할 수 있지만 이는 문제를 유발할 가능성이 많으므로 추천하지 않습니다.

컴파일러가 선언이 불일치하는지 검사하기 위해서는 반드시 전역 선언을 헤더 파일에 넣는 것이 중요합니다. 특히, external 함수의 prototype을 `.c` 파일에 넣지 않도록 하기 바랍니다. 이는 정의와 일치하는 지 검사해 주지도 않으며, 만약 정의와 일치하지 않는다면 오히려 쓰지 않는 것보다 못합니다.

질문 10.6과 18.8을 참고하기 바랍니다.

References [K&R1] § 4.5 pp. 76–7
[K&R2] § 4.4 pp. 80–1
[C89] § 6.1.2.2, § 6.7, § 6.7.2, § G.5.11
[ANSI Rationale] § 3.1.2.2
[H&S] § 4.8 pp. 101–104, § 9.2.3 p. 267
[CT&P] § 4.2 pp. 54–56

Q 1.8 C 언어에서 추상화된 data type을 만드는 가장 좋은 방법이 무엇일까요?

Answer 질문 2.4를 참고하기 바랍니다.

Q 1.9 "semiglobal" 변수, 즉, 몇 소스 파일의 함수들은 볼 수 있으나, 다른 소스 파일에서는 볼 수 없는 그러한 '절반 정도의' 전역 변수를 만들 수 있을까요?

Answer C 언어에서 그런 일은 할 수 없습니다. 모든 함수를 같은 source 파일에 넣는 것이 불편하거나 불가능하다면, 아래 중 한가지 방법을 쓰시기 바랍니다:

- 한 라이브러리나 패키지의 모든 함수와 변수에 적당한 prefix 이름을 붙이고, 사용자에게 이 prefix를 쓰는 이름을 만들지 말라고 권장합니다. (즉, 이 prefix로 시작하는 모든 이름은 reserved된 private한 것이라고 알립니다.)
- 일반적인 코드에서 쓰이지 않는, 밑줄('_')로 시작하는 이름을 사용합니다. (질문 1.29를 참고해서, 사용자와 시스템에서 제공하는 이름들과 충돌나지 않게 하기 바랍니다.)

어떤, 특별한 linker를 써서, 기존의 이름들의 scope를 제한해서 충돌을 미리 방지하는 방법도 있을 수 있지만, C 언어의 범위를 넘어가는 내용이므로 여기에서 다루지 않습니다.

1.4 Storage Class

우리는 선언의 두 가지 부분, base type과 declarator를 이미 다루었습니다. 다음 몇 질문에서는 storage class에 대한 것을 다룹니다. Storage class는 선언된 object나 함수의 visibility와 lifetime을 (각각 "scope"와 "duration"이라고 부르기도 합니다.) 다룹니다.

Q 1.10 `static` 함수나 변수를 선언할 때, 항상 `static`이라고 써 주어야 하나요?

Answer 언어 표준에서는 항상 써 줄 것을 요구하지는 않습니다. (가장 중요한 것은 첫 선언에 `static`을 써 주는 것입니다.) 하지만 규칙이 조금 복잡하게 얽혀 있고, 함수냐 변수냐에 따라 조금씩 다릅니다. (게다가 이 문제에 있어서 현존하는 코드도 너무나도 다양합니다.) 따라서 가장 안전한 방법은, 모든 정의와 선언에서 항상 `static`을 써 주는 것입니다.

References [ANSI] § 3.1.2.2
[C89] § 6.1.2.2
[ANSI Rationale] § 3.1.2.2
[H&S] § 4.3 p. 75

Q 1.11 함수 선언에서 `extern`이 의미하는 게 무엇인가요?

Answer 이 함수의 정의가 다른 소스 파일에 있을 수 있다는 것을 알려주는 단순한 스타일적인 문제입니다. 따라서 다음 두 줄의 차이는 없습니다:

```
extern int f();
int f();
```

덧붙여 질문 1.10도 참고하시기 바랍니다.

References [C89] § 6.1.2.2, § 6.5.1
[ANSI Rationale] § 3.1.2.2
[H&S] § 4.3, 4.3.1 pp. 75–6

Q 1.12 `auto` 키워드는 어디에 쓰이나요?

Answer 전혀 쓰이지 않습니다. 이는 오래된(archaic) 문법에 쓰이는 것으로 현재에는 쓰이지 않습니다. 질문 20.37을 참고하기 바랍니다.

References [K&R1] § A8.1 p. 193
[C89] § 6.1.2.4, § 6.5.1
[H&S] § 4.3 p. 75, § 4.3.1 p. 76

1.5 Typedefs

`typedef`가 문법적으로는 storage class이지만, 이 키워드는 이름이 알려주듯, 새로운 type 이름을 정의하는 데에 쓰입니다.

Q 1.13 새 타입을 만들때, typedef를 쓰는 것하고, 매크로를 쓰는 것하고 무슨 차이가 있죠?

Answer 일반적으로, typedef를 쓰는 것이 더 좋습니다. 왜냐하면 포인터 타입을 쓸 때 그 차이가 드러납니다:

```
typedef char *String_t;
#define String_d char *
String_t s1, s2;
String_d s3, s4;
```

위의 예에서 보면, `s1`, `s2`, `s3`는 모두 `char *` 타입이지만, `s4`는 `char` 타입입니다. 물론 이 결과는 개발자가 원한 것이 아닙니다. (덧붙여 질문 1.5도 참고하시기 바랍니다.)

매크로가 좋은 이유는, `#ifdef`를 쓸 수 있기 때문입니다. (덧붙여 질문 10.15도 참고하시기 바랍니다.) 반면에 typedef는 또 스코프 규칙을 잘 따른다는 장점이 있습니다. (즉, 함수나 블럭의 안에서 선언되어, 그 안에서만 영향을 줄 수 있습니다.)

덧붙여 질문 1.17, 2.22, 11.11, 15.11도 참고하시기 바랍니다.

References [K&R1] § 6.9 p. 141
[K&R2] § 6.7 pp. 146–7
[CT&P] § 6.4 pp. 83–4

Q 1.14 Linked list 정의가 안됩니다. 다음과 같이 했는데 컴파일러는 계속 에러 메시지만 출력합니다. C 언어에서 구조체는 자신에 대한 포인터를 포함할 수 없는 것인가요?

```
typedef struct {
  char *item;
  NODEPTR next;
} *NODEPTR;
```

Answer C 언어에서 구조체는 자신에 대한 포인터를 포함할 수 있습니다. 이 질문에서 문제는 `NODEPTR`이 `typedef` 이름이고, 이 이름을 사용한 시점에서 이 이름의 정의가 끝나지 않았다는 것입니다. 이 코드를 고치기 위해서, 먼저 구조체에 태그(tag, `struct node`)를 만들고 "`next`" 필드를 "`struct node *`" 타입으로 선언합니다. 또는 `typedef` 선언과 구조체 정의를 분리시켜도 됩니다. 다음 코드가 해결 방법 중 하나입니다.

```
struct node {
  char *item;
  struct node *next;
};
typedef struct node *NODEPTR;
```

이 문제를 해결하기 위해 적어도 세 가지의 다른 방법이 있습니다.

서로를 포함하는 한 쌍의 `typedef`된 구조체를 정의할 때도 비슷한 문제가 발생할 수 있으며, 위와 같은 방식으로 해결할 수 있습니다.

질문 2.1을 참고하시기 바랍니다.

References [K&R1] § 6.5 p. 101
[K&R2] § 6.5 p. 139
[C89] § 6.5.2, § 6.5.2.3
[H&S] § 5.6.1 pp. 132–3

Q 1.15 서로 참조하는 구조체를 만들 수 있을까요? 아래처럼 코드를 만들었는데 컴파일러가 `BPTR`이 없다고 에러를 출력합니다.

```
typedef struct {
  int afield;
  BPTR bpointer;
```

```
} * APTR;

typedef struct {
  int bfield;
  APTR apointer;
} * BPTR;
```

Answer 질문 1.14처럼, 이 문제는 구조체나 포인터에 있는 것이 아니라, typedef에 관한 것입니다. 먼저, (typedef 없이) 두 구조체 tag 이름을 써서 다음과 같이 만듭니다:

```
struct a {
  int afield;
  struct b *bpointer;
};

struct b {
  int bfield;
  struct a *apointer;
};
```

그러면, 컴파일러는, 아직 (자세히 말해, 아직 'incomplete'한 구조체인) `struct b`에 대해서 잘 모르지만 `struct a` 안에서 `struct b` 타입을 가리키는 `struct b *bpointer`를 쓰는 것을 허락합니다. 때때로 다음과 같이 먼저 선언해 주는 것이 필요합니다:

```
struct b;
```

This empty declaration masks the pair of structure declarations (if in an inner scope) from a different `struct b` in an outer scope. 위와 같이 두 구조체를 tag 이름과 같이 선언한 다음, 다음과 같이 따로 typedef를 만들 수 있습니다:

```
typedef struct a *APTR;
typedef struct b *BPTR;
```

멤버 포인터에 typedef 이름을 쓰기 위해, 다음과 같이 typedef 이름을 먼저 선언할 수도 있습니다:

```
typedef struct a *APTR;
```

```
typedef struct b *BPTR;
struct a {
  int afield;
  BPTR bpointer;
};

struct b {
  int bfield;
  APTR apointer;
};
```

덧붙여 질문 1.14도 참고하시기 바랍니다..

References [K&R2] § 6.5 p. 140
[ANSI] § 3.5.2.3
[C89] § 6.5.2.3
[H&S] § 5.6.1 p. 132

Q 1.16 다음 두 선언은 무엇이 서로 다른가요?

```
struct x1 { ... };
typedef struct { ... };
```

Answer 질문 2.1을 보기 바랍니다.

Q 1.17 "`typedef int (*funcptr)();`"은 무슨 뜻인가요?

Answer typedef 이름인 `funcptr`을 정의합니다. `funcptr`은 여기에서, 임의의 갯수의 인자를 받고 `int`를 리턴하는 함수에 대한 포인터입니다. 예를 들어 다음과 같이 하나 이상의 함수에 대한 포인터를 선언할 수 있습니다:

```
funcptr pf1, pf2;
```

위 선언은 아래와 완전히 같은 뜻이지만, 좀 더 보기 쉽습니다:

```
int (*pf1)(), (*pf2)();
```

덧붙여 질문 1.21, 4.12, 15.11도 참고하시기 바랍니다.

References [K&R1] § 6.9 p. 141
[K&R2] § 6.7 p. 147

1.6 The const qualifier

C 언어 선언에서는 type qualifier라는 게 있으며, 이는 ANSI C에서 새로 추가된 것입니다. Qualifier에 관한 질문은 Chapter 11에서 다룹니다.

Q 1.18 I've got the declarations

```
typedef char *charp;
const charp p;
```

Why is `p` turning out `const` instead of the characters pointed to?

Answer 질문 11.11을 보기 바랍니다.

Q 1.19 아래와 같이 배열을 초기화하는데 왜 `const` 값을 쓸 수 없는 것인가요?

```
const int n = 5;
int a[n];
```

Answer 질문 11.8을 보기 바랍니다.

Q 1.20 `const char *p`, `char const *p`, `char * const p`가 서로 어떻게 다른가요?

Answer 질문 11.9, 1.21을 보기 바랍니다.

1.7 Complex Declarations

C 언어의 선언은 엄청나게 복잡해질 수 있습니다. 일단 이 암호?를 풀어낼 방법을 배우면, 물론 이런 복잡한 선언은 거의 쓰이지 않지만, 어떤 선언이 나오더라도 두려워하지 않게 됩니다.

여러분이 프로그램을 `*(*(*a[N])())()`와 같은 declarator로 암호화하려는 취미가 없다면, 질문 1.21의 두번째와 같이 `typedef`를 써서 간단하게 만들 수 있습니다.

Q 1.21 문자를 가리키는 포인터를 리턴하는 함수에 대한 포인터를 리턴하는 함수에 대한 포인터 N 개로 이루어진 배열(array)을 어떻게 선언하죠?

Note 참고로 원문은 다음과 같습니다:

How do I declare an array of N pointers to functions returning pointers to functions returning pointers to characters?

Answer 다음과 같이 세 가지로 답할 수 있습니다:

- `char *(*(*a[N])())();`
- `typedef`를 써서 차례대로 만들어 가면 됩니다:

```
typedef char *pc;      /* pointer to char */
typedef pc fpc();      /* function returning pointer
                          to char */
typedef fpc *pfpc;     /* pointer to above */
typedef pfpc fpfpc();  /* function returning...  */
typedef fpfpc *pfpfpc; /* pointer to...  */
pfpfpc a[N];           /* array of...  */
```

- `cdecl` 프로그램을 쓰면 영어를 C 언어로, 또는 C 언어를 영어로 변환할 수 있습니다.

```
cdecl> declare a as array of pointer to function
    returning pointer to function returning pointer
    to char
char *(*(*a[])())()
```

`cdecl`은 복잡한 선언이나 캐스팅에 대해서도 많은 도움을 줍니다. 질문 18.1을 참고하시기 바랍니다.

C 언어를 설명하는 좋은 책이라면 이러한 복잡한 선언에 대한 내용이 있기 마련이니 이 부분을 꼭 읽어보시기 바랍니다.

위의 예에서 쓰인 함수 포인터(pointer-to-function) 선언은 파라메터 타입에 대한 정보를 포함하지 않았습니다. 만약 파라메터가 복잡한 함수라면 전체 선언은 매우 읽기 어렵습니다 (최신 버전의 `cdecl`은 이 경우에도 도움이 됩니다.)

References [K&R2] § 5.12 p. 122
[C89] § 6.5ff (esp. § 6.5.4)
[H&S] § 4.5 pp. 85–92, § 5.10.1 pp. 149–50

Q 1.22 같은 타입의 함수에 대한 포인터를 리턴하는 함수를 선언할 수 있나요? 저는 상태 머신(state machine)을 만들고, 각 함수가 하나의 상태를 나타내게 한

다음, 이 함수가 다음 상태를 나타내는 함수 포인터를 리턴하게 하려고 합니다. 그러나 이런 함수를 선언할 방법을 찾지 못하고 있습니다.

Answer 직접 할 수는 없습니다. 한가지 방법은 함수가 일반적인(generic) 함수 포인터를 리턴하게 한 다음, 이를 원하는 타입으로 캐스팅하는 것과 또 다른 방법으론 함수가 어떤 구조체를 리턴하게 하고, 그 구조체에 이 타입의 함수에 대한 포인터를 저장하는 방식을 쓸 수 있습니다.

1.8 Array Sizes

Q 1.23 지역 변수나 파라메터로 배열을 선언할 때, 그 크기를 따로 인자로 받아서 선언할 수 있을까요?

Answer 간단히 말해, 불가능합니다. 질문 6.15, 6.19를 참고하기 바랍니다.

Q 1.24 다음과 같이, 한 파일에 `extern` 배열을 만들고, 다른 파일이 그 배열을 쓰려고 합니다. `file2.c`에서 왜 `sizeof` 연산이 안되나요?

```
file1.c:
  int array[] = { 1, 2, 3 };

file2.c:
  extern int array[];
```

Answer 크기가 알려지지 않은, `extern` 배열 선언은 'incomplete type'입니다. 따러서 `sizeof`가 제대로 동작할 수 없습니다. 왜냐하면, 컴파일할 때, 다른 파일에 정의되어 있는 배열의 크기를 알 수 없기 때문입니다. 이 때, 사용자는 다음과 같이 세 가지 방법을 쓸 수 있습니다:

(a) 변수 하나를 따로 두어서, 그 배열의 크기로 초기화한 다음 씁니다:

```
file1.c:
  int array[] = { 1, 2, 3};
  int arraysz = sizeof(array);

file2.c:
  extern int array[];
  extern int arraysz;
```

덧붙여 질문 6.23도 참고하시기 바랍니다.

(b) 배열의 크기를 매크로 상수로 정의해 두고 이를 정의와 선언에 함께 사용합니다:

```
file1.h:
  #define ARRAYSZ       3

file1.c:
  #include "file1.h"
  int array[ARRAYSZ];

file2.c:
  #include "file1.h"
  extern int array[ARRAYSZ];
```

(c) 배열의 마지막 요소 값으로, 어떤 특정한 값을 (sentinel value, 일반적으로 0, -1, 또는 NULL) 써서, (나중에 필요하면 그 값이 나올때까지 세어서 배열의 크기를 알 수 있게) 특별히 크기에 대한 것을 지정하지 않도록 합니다.

```
file1.c:
  int array[] = { 1, 2, 3};

file2.c:
  extern int array[];
```

물론, 배열을 정의할 때, 초기값이 이미 정해져 있느냐에 따라 선택이 달라질 수 있습니다. 만약에 초기값이 위와 같이 정해져 있는 경우에는, 두번째 방식은 별로 좋지 않습니다. 덧붙여 질문 6.21도 참고하시기 바랍니다.

References [H&S] § 7.5.2 p. 195

1.9 Declaraction Problems

Sometimes the compiler insists on complaining about your declarations no matter how carefully constructed you thought they were. These questions uncover some of the reasons why. (Chapter 16 is a similar collection of baffling run-time problems.) 때때로 컴파일러는 여러분이 얼마나 주의깊게 선언을 만들었는지에 상관없이 불평을 하기도 합니다. 이 section의 질문들은 왜 그런지 그 이유를 설명합니다 (Chapter 16에서는 실행 시간에 발생할 수 있는 이상한 문제들에 대해 설명합니다.)

Q 1.25 제 컴파일러는 함수가 중복되어 정의되어 있다고 에러 메시지를 출력하지만 저는 한번만 정의했고, 한 번만 호출했습니다. 무엇이 잘못되었는지 모르겠군요.

Answer 함수의 선언이, 이 함수를 호출할 때까지 나타나지 않으면, 컴파일러는 이 함수가 `int`를 리턴한다고 가정합니다. 그 다음 나중에 실제 선언이나 정의가 나오고, 가정과 일치하지 않을 경우 그러한 에러가 발생합니다. 즉, `int`가 아닌 다른 타입을 리턴하는 함수는 호출하기 전에 반드시 선언해야 합니다.

또는 어떤 헤더 파일에 선언되어 있는 함수 이름을 (정말로) 중복해서 다른 함수로 만들었을 경우에 이 문제가 발생할 수 있습니다.

질문 11.3과 질문 15.1을 참고하기 바랍니다.

References [K&R1] § 4.2 p. 70
[K&R2] § 4.2 p. 72
[C89] § 6.3.2.2
[H&S] § 4.7 p. 101

Q 1.25b `main()`을 선언하는 정확한 방법이 궁금합니다. `void main()`으로 해도 좋은가요?

Answer 질문 11.12a와 11.15를 참고하기 바랍니다. (어쨌든 `void main()`은 틀린 선언입니다.)

Q 1.26 제가 보기에는 제대로 되어 있는데, 컴파일러가 함수 prototype이 서로 다르다고 경고합니다. 왜 그럴까요?

Answer 질문 11.3을 보기 바랍니다.

Q 1.27 파일 가장 첫 부분에서 이상한 문법 에러가 발생합니다. 왜 그럴까요?

Answer 질문 10.9를 보기 바랍니다.

Q 1.28 다음과 같이, 큰 배열을 만들었는데, 컴파일러가 에러를 냅니다. 왜 그럴까요?

```
double array[256][256];
```

Answer 질문 19.23, 7.16을 보기 바랍니다.

1.10 Namespace

이름을 짓는 것이 어려워 보이지 않을 지 몰라도, 사실 매우 어려운 문제입니다. 물론 함수나 변수의 이름을 짓는 것이 책이나 건물, 아기들의 이름을 짓는 것보다는 쉽습니다 — 여러분은 많은 사람들이 여러분 코드의 이름들을 싫어할 지 모른다고 고민할 필요는 없습니다 — 적어도 같은 이름이 이미 쓰이고 있는지만 조사하면 됩니다.

Q 1.29 어떤 이름이 이미 예약되어 있고(reserved), 어떤 이름을 안전하게 쓸 수 있는지 알 수 있는 방법이 있을까요?

Answer Namespace management는 항상 큰 이슈가 됩니다. 이 문제는 — 항상 확실하게 문제를 정의하는 것은 불가능하지만 — 여러분이 선택한 이름이 시스템이 쓰고 있는 이름과 충돌나지 않게 해서 "multiply defined"라는 에러가 발생하지 않게 하거나, (좀 더 나쁜 상황이라 할 수 있는) 충돌이 나서, 기존 시스템에서 정의한 내용을 바꾸어 엉망이 될 수 있는 상황을 방지하자는 취지입니다. 또한, 지금은 충돌이 나지 않지만, 나중에 개발될, 시스템 라이브러리가 제공하는 이름과 충돌나지 않는다는 어떤 보증이 필요할 경우도 있습니다.[4] (실제로, 새 컴파일러에서 재컴파일했을 경우에 namespace와 다른 문제들 때문에 build 실패하는 경우가 많습니다.) 그래서, ANSI/ISO C 표준은 사용자와 시스템 사이의 독립된 namespace에 대한 정의에 신경을 많이 쓰고 있습니다.

ANSI 규칙에 따라, 우리는 어떤 identifier가 예약(reserved)되어 있는지 알 수 있습니다. 먼저, 우리는 identifier에 대한 특징에 대해 알아야 합니다: 이 특징에는 scope, namespace, linkage가 있습니다.

- C 언어는 네 개의 scope가 있습니다. (Scope는 idenitifer의 선언이 어떤 영역에서 영향을 미치는가를 나타냅니다.): function, file, block, 그리고 prototype으로 구분합니다. (마지막 prototype scope는 함수 prototype 선언에서 파라메터 리스트 안에서 적용되는 것입니다. 질문 11.5를 참고하기 바랍니다.)
- C 언어는 네 개의 namespace가 있습니다: (`goto`에서 대상을 지정할 때 쓰이는) label, (structure, union, enumeration에서 쓰이며, 이들은 이론상 충분히 각각 다른 namespace를 가질 수 있지만, 사실 하나의 namespace를 가지는) tag, (각각 structure, union마다 독립적인) structure/union member, 그리고 이 외의 다른 데에서 쓰이는 보통의

[4] 이러한 염려는, public symbol 뿐만 아니라, 시스템이 내부적으로 사용하는 internal, private symbol에도 해당합니다.

identifier, 즉 "ordinary identifier"를 (예를 들어 함수, 변수, typedef 이름, enumeration constant 등) 위한 namespace. 또, 표준에 "namespace"라고 지정되어 있지는 않지만, preprocessor macro를 위한 이름들의 집합(A set of names)이 있습니다; 이 것은 컴파일러에게 전달되기 전, preprocessor에 의해 다 처리되기 때문에, 컴파일러 에게는 네 개의 namespace가 주어집니다.

- 표준은 세 개의 "linkage"를 정의합니다: external, internal, 그리고 none이 있습니다. (이 문제의 답변을 위해 말하면) External linkage에 해당하는 것은 (모든 source file에 걸쳐) global, non-static 변수와 함수입니다. Internal linkage에 해당하는 것은 file scope를 가지는 static 함수와 변수입니다. 그리고 none, 즉 "no linkage"에 해당하는 것은, enumeration constant나 typedef 이름입니다.

[ANSI] § 4.1.2.1 ([C89] § 7.1.3)에 따르면 다음과 같은 규칙이 있습니다:

Rule 1 밑줄 문자로 시작하고, 두 번째 문자가 밑줄이거나 대문자인 모든 이름은 (모든 scope와 모든 namespace에서) 항상 reserved 상태입니다.

Rule 2 밑줄로 시작하는 모든 이름은 file scope에서 ordinary identifier(즉, 함수, 변수, typedef, enumeration constant 등)를 위해 reserve되어 있습니다.

Rule 3 어떤 표준 헤더 파일을 포함했을 때, 그 헤더 파일에서 제공하는 모든 매크로 이름은 reserve되어 있습니다.

Rule 4 (함수 이름처럼) External linkage를 가지는 모든 표준 라이브러리 identifier들은 external linkage를 위한 identifier로 reserve되어 있습니다.

Rule 5 표준 헤더 파일에 정의되어 있는, file scope를 가지는 typedef와 tag 이름은, 그 헤더 파일을 포함시켰을 경우, (같은 namespace를 지니는) file scope에서 모두 reserve되어 있습니다. (표준은 실제로 "each identifier with file scope,"라고 말하지만, 네번째 규칙에 적용되지 않는 이름은 typedef와 tag 이름밖에 없습니다.)

게다가, 여러 매크로 이름과 표준 라이브러리 identifier가 미래 표준을 위해 예약되어 있기 때문에, 규칙 3, 4를 더 복잡하게 만듭니다. 즉, 어떤 패턴을 가지는 이름들이 미래에 나올 표준에 쓰일 수 있으며, 이 패턴은 아래 표에 나와 있습니다:

Header	Future directions patterns
<ctype.h>	is[a-z]*, to[a-z]* (function)
<errno.h>	E[0-9]*, E[A-Z]* (macros)
<locale.h>	LC_[A-Z]* (macros)
<math.h>	cosf, sinf, sqrtf, etc.
	cosl, sinl, sqrtl, etc. (all functions)
<signal.h>	SIG[A-Z]*, SIG_[A-Z]* (macros)
<stdlib.h>	str[a-z]* (functions)
<string.h>	mem[a-z]*, str[a-z]*, wcs[a-z]* (functions)

여기에서 [A-Z]는 "아무 대문자"를 뜻하며, [a-z]는 "아무 소문자"를 뜻하며, [0-9]는 "아무 숫자"를 뜻합니다. *는 "아무것이나 다"를 뜻합니다. 예를 들어, 여러분이 <stdlib.h>를 포함시켰다면, str로 시작하고, 그 다음 글자가 소문자로 이루어진 모든 external identifier는 예약되어 있습니다.

이 다섯개의 규칙이 어렵다면, 다음 advice를 따르면 됩니다:

1, 2 밑줄로 시작하는 이름을 쓰지 말기 바랍니다.

3 (위 표에 나온 모든 이름을 포함해서) 표준 매크로와 같은 이름을 쓰지 말기 바랍니다.

4 표준 라이브러리에 있는 모든 변수 및 함수 이름과, 위 표에 나온 이름을 쓰지 말기 바랍니다. (엄밀하게 말해서, 이름 비교는 첫 여섯 글자만 가지고 이루어 집니다. 질문 11.27 참고)

5 표준에서 제공하는 typedef나 tag 이름을 새로 정의하는 이름에 쓰지 말기 바랍니다.

사실 위 advice는 너무 고압적입니다. 원한다면 다음 예외 사항을 기억하기 바랍니다:

1, 2 밑줄로 시작하고, 두 번째 문자가 숫자나 소문자인 이름을, label 또는 structure/union 이름에 쓸 수 있습니다. 또 function, block, prototype scope에서 쓸 수 있습니다.

3 어떤 매크로를 정의하는 표준 헤더 파일을 포함시키지 않는다면, 그 (매크로) 이름을 쓸 수 있습니다.

4 표준 라이브러리 함수 이름을, static 또는 local 변수에 쓸 수 있습니다. (정확히 말해서, internal 또는 no linkage를 가지는 이름으로 쓸 수 있습니다.)

5 어떤 typedef나 tag 이름을 정의하는 표준 헤더 파일을 포함시키지 않았다면, 그 이름을 쓸 수 있습니다.

그러나, 위 예외 사항을 쓰기 전에, 몇 개는 위험할 수 있다는 것을 꼭 기억하기 바랍니다. (특히, 세번째와 다섯번째 예외 사항은 조심해야 합니다. 세번째와 다섯번째에 해당하는 이름을 쓴 후, 관련된 헤더 파일을 나중에 포함시키는 등의 실수가 우려됩니다.) 그리고 예외 1, 2는 이른바 시스템과 사용자 namespace 사이의 무인도(no man's land)에 해당하는 namespace를 다루고 있습니다.

이러한 예외 사항을 제공하는 이유는, 여러 add-in 라이브러리 제작자가, 자신만의 internal, hidden identifier를 선언할 수 있게 하기 위해서 입니다. 즉, 위 예외 사항에 해당하는 identifier를 썼을 경우, 시스템이 제공하는 identifier와 충돌할 가능성은 없습니다. 그러나 third-party가 제공하는 library의 identifier와 충돌할 가능성은 여전히 존재합니다. (또, 이러한 라이브러리 제작자라면, 필요할 경우에 시스템이 쓸 수 있는 identifier를 쓰는 것도 괜찮습니다. 단 매우 주의해야 합니다.)

일반적으로, 네번째 예외 사항을 써서, 함수 파라메터나 local 변수 이름에 표준 라이브러리 함수 이름이나, 앞 표에 나온 "future directions" 패턴에 해당하는 이름을 쓰는 것은 괜찮습니다. 예를 들어, "`string`"은 파라메터 이름이나 local 변수 이름으로 흔히 쓰이는 — 그리고 합법적인 — 이름입니다.

References [ANSI] § 3.1.2.1, § 3.1.2.2, § 3.1.2.3, § 4.1.2.1, § 4.13
[C89] § 6.1.2.1, § 6.1.2.2, § 6.1.2.3, § 7.1.3, § 7.13
[ANSI Rationale] § 4.1.2.1
[H&S] § 2.5 pp. 21–3, § 4.2.1 p. 67, § 4.2.4 pp. 69–70, § 4.2.7 p. 78, § 10.1 p. 284

Note 현재 C 표준은 위에서 말한 것처럼 여섯 글자 제한을 쓰지 않습니다. 자세한 것은 질문 11.27을 참고하기 바랍니다.

1.11 Initialization

변수의 선언은, 물론 그 변수에 대한 초기값을 포함할 수 있습니다. 만약 초기값을 주지 않으면, 기본적인 초기화(default initialization)가 수행될 수 있습니다.

Q 1.30 초기화되지 않은 변수의 값을 미리 예상할 수 있습니까? 전역 변수는 초기화되지 않은 경우 0을 가진다고 하는데, 이것을 널 포인터나 실수 0.0으로 해석해도 괜찮은가요?

Answer "static" 속성을 가진 초기화되지 않은 (전역 변수이거나 `static`으로 선언된 변수) 변수는 0의 값을 가집니다. 즉 프로그래머가 "`= 0`"으로 써 준 것과 똑같다는 말입니다. 따라서 널 포인터일 경우에도 올바른 값을 가집니다 (Chapter 5 참고). 마찬가지로 실수일 경우 0.0으로 해석됩니다.

"automatic" 속성을 가진 (`static`으로 선언되지 않은 지역 변수) 변수는 초기화되지 않을 경우 쓰레기 값을 가집니다. (이 경우에 쓰레기 값이 무엇인지는 아무도 모르며, 쓸 이유가 없습니다.)

`malloc()`이나 `realloc()`으로, 동적으로 할당된 메모리도 처음에 쓰레기 값을 가집니다. 따라서 프로그램에서 적당히 초기화시켜 주어야 합니다. `calloc()`으로 할당받은 메모리는 비트 단위로 0을 가지게 되지만, 이 것이 포인터나 실수 타입에서 0을 의미한다고 말할 수는 없습니다 (질문 7.31과 Chapter 5 참고).

References [K&R1] § 4.9 pp. 82–4
[K&R2] § 4.9 pp. 85–86
[C89] § 6.5.7, § 7.10.3.1, § 7.10.5.3
[H&S] § 4.2.8 pp. 72–3, § 4.6 pp. 92–3, § 4.6.2 pp. 94–5, § 4.6.3 p. 96, § 16.1 p. 386

Q 1.31 이 코드는 어떤 책에서 나온 것인데 컴파일되지 않습니다.

```
int f()
{
  char a[] = "hello, world!";
}
```

Answer 아마도 쓰고 있는 컴파일러가 ANSI 이전의 컴파일러일 것이라고 추측됩니다. ANSI 이전의 컴파일러는 "automatic aggregates" (예를 들어, `static`이 아닌 지역 배열, 구조체, `union`)의 초기화를 지원하지 않습니다.

(그리고 변수 `a`가 어떻게 쓰일 지에 따라 다르겠지만, 이 변수를 전역 (global) 또는 static으로, 또는 포인터로 바꾸어서 해결할 수 있으며, 또는 `strcpy()` 등을 써서 대입시켜주는 방법도 있습니다.)

질문 11.29을 참고하기 바랍니다.

Q 1.31b 이 초기화에서 잘못된 것이 무엇인가요?

```
char *p = malloc(10);
```

제 컴파일러는 "invalid initializer"라는 에러 메시지를 출력합니다.

Answer 아마 위의 선언이 정적(static)이거나 전역(non-local) 변수일 것입니다. 초기화에서 함수 호출을 쓰는 것은 자동(automatic) 변수[5]에서만 가능합니다.

Q 1.32 다음 두 선언에 차이점이 있나요?

```
char a[] = "string literal";
char *p  = "string literal";
```

Answer 문자열은 크게 두 가지 방법으로 쓰일 수 있습니다. 하나는 배열의 초기값 (위에서는 `char a[]`에 해당)으로 쓰이는 것입니다. 이는 배열의 각 요소들인 문자들에 대입되는 초기값을 나타냅니다. 이 경우가 아니라면 문자열이 이름이 없는 정적(static)인 배열에 저장되고 — 대개 이 배열은 읽기 전용의 속성을 가집니다 — 수식(expression)에서 쓰일 때에는 이 배열의 첫 요소를 가리키는 포인터로서 쓰이게 됩니다. 따라서 위의 선언 중 두번째 것은 실제 문자열이 읽기전용 배열에 저장되기 때문에 포인터 `p`를 가지고 문자열을 수정할 수 없습니다.

(오래된 C 언어 코드의 경우, `p`와 같은 포인터로 문자열의 내용을 변경하려고 시도하는 경우도 있습니다. 이러한 경우를 해결하기 위해 어떤 컴파일러는 문자열을 쓰기 가능한 메모리에 저장하도록 하는 옵션을 가지고 있습니다.)

질문 1.31, 6.1, 6.2, 6.8을 참고하기 바랍니다.

References [K&R2] § 5.5 p. 104
[C89] § 6.1.4, § 6.5.7
[ANSI Rationale] § 3.1.4
[H&S] § 2.7.4 pp. 31–2

Q 1.33 `char a[3] = "abc";`가 맞는 표현인가요?

Answer 맞습니다. 질문 11.22를 보기 바랍니다.

Q 1.34 함수 포인터를 선언하는 문법은 알겠는데, 초기화시키는 방법을 모르겠습니다.

[5]즉, static이 아닌 변수를 의미함.

Answer 다음과 같이 하면 됩니다:

```
extern int func();
int (*fp)() = func;
```

위와 같이 함수 이름이 수식에서 쓰인 경우, 이 이름은 — 이 함수의 시작 주소를 나타내는 — 포인터로 변경(decay)됩니다. 배열 이름이 단독으로 쓰이는 경우와 비슷합니다.

일반적으로 명백히 함수의 선언을 미리 적어주게 됩니다. 왜냐하면 이 경우, 자동으로 외부 함수 선언[6]을 만들어주지 않기 때문입니다 (왜냐하면 초기화에서 쓰이는 함수 이름은 함수 호출이 아니기 때문입니다).

질문 1.25와 4.12를 참고하기 바랍니다.

Q 1.35 `union`을 초기화할 수 있나요?

Answer 질문 2.20을 보기 바랍니다.

Q 1.A 구조체 멤버를 초기화하려는데 "Initializer element not computable at load time"라는 에러가 발생합니다.

Answer 질문 11.J를 보기 바랍니다.

[6] implicit external function declaration

Chapter 2

Structures, Unions, and Enumerations

Structure, union, enumeration은 여러분에게 새로운 data type을 정의할 수 있게 해준다는 공통점을 가집니다. 먼저, structure나 union은 여러분이 member나 field를 선언하여 새 data type을 정의할 수 있고, enumeration의 경우 상수를 (constant) 선언하여 새 data type을 정의할 수 있습니다. 동시에 여러분은 새로운 data type에 *tag* name을 줄 수 있습니다. 일단 새 type을 정의했다면, 정의와 동시에 또는 나중에 새 type의 instance(변수)를 선언할 수 있습니다.

복잡하게도, 기본 타입과 마찬가지로 user-defined type에도 `typedef`를 써서 새 이름을 줄 수 있습니다. 이렇게 했을 때, 여러분은 `typedef` 이름이 (만약 tag 이름이 존재할 경우) tag 이름과는 전혀 상관없다는 것을 아셔야 합니다.

이 chapter의 질문들은 다음과 같이 정리되어 있습니다: 질문 2.1부터 2.18은 structure에 대하여, 질문 2.19부터 2.20까지는 union에 대하여, 질문 2.25부터 2.26까지는 bitfield에 대해 다룹니다.

2.1 Structure Declaration

Q 2.1 다음 두 선언의 차이점이 무엇인가요?

```
struct x1 { ...  };
typedef struct { ...  } x2;
```

Answer 첫번째 선언은 "구조체 태그(structure tag)"를 선언한 것입니다; 두번째 선

언은 "typedef"를 선언한 것입니다. 주 차이점은 첫번째 경우의 타입 이름은 `struct x1`이며, 두번째 경우의 타입 이름은 간단히 `x2`라는 것입니다. 즉 두번째 것이 조금 더 추상화된 타입이라 할 수 있습니다. — 즉 사용자들이 `struct`이라는 키워드를 쓰지 않게되어 이 타입이 구조체인지 아닌지 알 필요가 없습니다:

```
x2 b;
```

tag를 써서 정의된 구조체는 다음과 같이 선언해야 합니다.[1]

```
struct x1 a;
```

(두 가지 방식 모두 적용해서 다음과 같이 정의할 수도 있습니다:

```
typedef struct x3 { ... } x3;
```

조금 혼동스럽기는 하지만, tag 이름과 typedef 이름이 중복되는 것은, 서로 다른 namespace에 속하기 때문에, 전혀 상관없습니다. 질문 1.29를 보기 바랍니다.)

Q 2.2 다음 코드가 왜 동작하지 않을까요?

```
struct x { ...  };
x thestruct;
```

Answer C 언어는 C++이 아닙니다. 즉 구조체 태그(tag) 이름에 대해 자동으로 typedef 이름이 만들어지지 않습니다. 실제 구조체는 `struct` 키워드를 써서 정의합니다:

```
struct x thestruct;
```

원한다면, 구조체를 선언할 때, typedef 이름을 같이 선언하고, 이 typedef 이름으로 실제 구조체 변수 또는 상수를 만들 때 쓸 수 있습니다:

```
typedef struct { ... } tx;

tx thestruct;
```

[1] C++ 언어에서는 이 두가지 방식이 차이가 없습니다. 몇몇 C++ 컴파일러는 C와 호환을 위해 C 언어처럼 동작합니다. C++에서는 구조체 tag는 자동으로 typedef 이름으로 선언됩니다.

덧붙여 질문 2.1도 참고하시기 바랍니다.

Q 2.3 구조체가 자신에 대한 포인터를 포함할 수 있나요?

Answer 당연히 포함할 수 있습니다. typedef와 함께 쓸 때 조금 문제가 될 수 있습니다. 이 점에 대해서는 질문 1.14, 1.15를 보기 바랍니다.

Q 2.4 C 언어에서 추상화된 데이터 타입을 구현하는 가장 좋은 방법을 알려주세요.

Answer 한 가지 방법은 사용자들이 구조체 포인터를 쓰게 하는 것입니다 — 이 때, typedef 이름을 쓰는 것이 좋습니다. 이 포인터는 어떤 구조체를 가리키는 것이며, 이 구조체의 세부 사항은 사용자에게 알려줄 필요가 없습니다. 다시 말해서, client는 이 구조체가 어떠한 멤버를 포함하고 있는 지, 전혀 알 필요없이, 이 구조체 포인터를 사용합니다. (함수를 부를때, 이 포인터를 넘겨주며, 이 구조체 포인터를 리턴하는 등) 구조체에 대한 자세한 사항을 몰라도 될 경우에 — `->` 연산자나 `sizeof` 연산자가 쓰이지 않는다면 — C 언어로 완전하지 않은, incomplete type에 대한 포인터를 쓸 수 있습니다. 단지, 이 구조체를 실제 다루는 함수가 들어있는 파일안에서 완전한 정의를 제공하면 됩니다. 덧붙여 질문 11.5도 참고하시기 바랍니다.

Note 표준 입출력 함수들이 `FILE *` 타입의 인자를 받는 것을 생각하면 쉽습니다.

Q 2.5 아래와 같이 선언했더니, 이상한 경고 메시지("struct x introduced in prototype scope" 또는 "struct x declared inside parameter list")가 발생합니다:

```
extern f(struct x *p);
```

Answer 질문 11.5를 보기 바랍니다.

Q 2.6 구조체를 다음과 같이 정의하는 코드를 봤습니다:

```
struct name {
  int namelen;
  char namestr[1];
};
```

그리고나서 namestr에 공간을 할당하고 (allocation), 배열 namestr이 여러 element를 가진 것처럼 쓰는 것을 봤습니다. 이게 안전한 방법인가요?

Answer 합법적인지(legal), 이식성이 뛰어난 지는 확실하지 않으나, 매우 인기있는 방법입니다. 보통 다음과 같은 코드를 써서 위 구조체를 사용합니다:

```
#include <stdlib.h>
#include <string.h>

struct name *makename(char *newname)
{
  struct name *ret =
    malloc(sizeof(struct name) - 1 +
           strlen(newname) + 1);
      /* -1 for initial [1]; +1 for \0 */
  if (ret != NULL) {
    ret->namelen = strlen(newname);
    strcpy(ret->namestr, newname);
  }
  return ret;
}
```

이 함수는 name 구조체가, 그 멤버인 namestr이 주어진 문자열을 충분히 포함할 수 있도록 (단순히 크기 1인 배열이 아닌) 공간을 할당합니다.

이것은 인기있는 테크닉 중의 하나이지만 Dennis Ritchie씨는 이 방법을 "unwarranted chumminess with the C implementation"이라고 부릅니다. 이 방법은 공식적인 C 표준에 정확히 부합하지는 않지만, (이것이 표준에 부합하는지 아닌지에 대한 열렬한 논쟁이 있지만, 이 책의 범위를 벗어나므로 생략합니다.) 현존하는 거의 모든 컴파일러에서 동작합니다. (배열의 경계를 검사해주는 컴파일러에서는 경고를 출력할 수도 있습니다.)

또 한가지 방법으로는 가변적인 요소를 매우 작게 잡든 대신 아주 크게 잡는 것입니다; 위의 예에 적용하자면:

```
#include <stdlib.h>
#include <string.h>

#define MAX 100
```

```
struct name {
  int namelen;
  char namestr[MAX];
};

struct name *makename(char *newname)
{
  struct name *ret =
    malloc(sizeof(struct name) - MAX +
           strlen(newname) + 1);
      /* -1 for initial [1]; +1 for \0 */
  if (ret != NULL) {
    ret->namelen = strlen(newname);
    strcpy(ret->namestr, newname);
  }
  return ret;
}
```

이 때 MAX는 예상되는 저장될 문자열보다 크게 잡습니다. 그러나 이 방법도 표준에 정확히 부합하는 방법도 아닙니다. 게다가 이런 구조체를 쓸 때에는 매우 주의를 기울여야 합니다. 왜냐하면 이런 경우에는 컴파일러보다 프로그래머가 그 크기에 대해 더욱 잘 알고 있기 때문에 (특히, 이런 경우 pointer로만 작업을 할 수 있습니다.) 컴파일러가 알려주는 정보(경고나 에러)가 의미없게 됩니다.

가장 올바른 방법은, 다음과 같이, 배열 대신에 문자 포인터를 쓰는 것입니다:

```
#include <stdlib.h>
#include <string.h>

struct name {
  int namelen;
  char *namep;
};

struct name *makename(char *newname)
{
  struct name *ret = malloc(sizeof(struct name));
  if (ret != NULL) {
```

```
    ret->namelen = strlen(newname);
    ret->namep = malloc(ret->namelen + 1);
    if (ret->namep == NULL) {
      free(ret);
      return NULL;
    }
    strcpy(ret->namestr, newname);
  }
  return ret;
}
```

위와 같이 하면, 문자열의 길이와 실제 문자열을 하나의 메모리 블럭에 저장한다는 "편리함"이 사라집니다. 또한 위와 같이 할당한 메모리를 돌려주기 위해서는 `free`를 두 번 불러야 합니다; 질문 7.23을 참고하기 바랍니다.

만약에, 위와 같이, 저장할 데이터 타입이 문자(character)라면, `malloc`을 두 번 부르는 것을 다시 한 번으로 줄여서, 연속성을 보장할 수 있는 방법이 있습니다. (따라서 `free`를 한 번만 불러도 됩니다):

```
struct name *makename(char *newname)
{
  char *buf = malloc(sizeof(struct name) +
                     strlen(newname) + 1);
  struct name *ret = (struct name *)buf;
  ret->namelen = strlen(newname);
  ret->namep = buf + sizeof(struct name);
  strcpy(ret->namep, newname);

  return ret;
}
```

그러나, 위와 같이, `malloc`을 한 번 불러서, 두번째 영역까지 할당하는 것은, 두 번째 영역이 `char` 배열로 취급될 경우에만 이식성이 있습니다. 다른, 더 큰 데이터 타입을 쓴다면, alignment (질문 2.12, 16.7 참고) 문제가 발생할 가능성이 높습니다.

[C9X]에서는 "flexible array member"라는 개념을 소개하고 있고, 이는 배열이 구조체의 마지막 멤버로써 쓰일 때에는 배열의 크기 지정을 생략할 수 있도록 해 줍니다.

References [ANSI Rationale] § 3.5.4.2
[C9X] § 6.5.2.1

Note C99 표준에 소개된, "flexible array member"를 쓰면, 이 문제를 깔끔하게 해결할 수 있습니다. 반드시 질문 11.G를 읽기 바랍니다.

2.2 Structure Operations

Q 2.7 Structure가 변수에 대입되고 함수 인자로 전달될 수 있다고 들었지만, [K&R1]에서는 그렇지 않다고 하는군요.

Answer [K&R1]에서도, 앞으로 만들어질 컴파일러에서는 이러한 제한들이 없을 것이라고 씌여 있습니다. 그리고 실제로도, Ritchie가 만든 컴파일러에서도 ([K&R1] 출판 이후) 구조체를 대입하고, 함수 인자로 전달하고, 구조체를 리턴하는 모든 연산들이 허용되었습니다. 아주 오래된 몇몇 C 컴파일러의 경우, 이 제한이 남아있지만 대부분의 컴파일러들은 구조체 연산을 지원합니다. 그리고 이 연산들은 이제 ANSI C 표준의 일부가 되었습니다. 따라서 전혀 거리낌없이 쓰셔도 좋습니다.[2]

(구조체가 복사, 전달, 리턴되는 경우, 복사 작업은 통채로(monolithically) 이루어지기 때문에 구조체 안의 포인터 멤버 필드가 가리키는 데이터는 복사되지 않는다는 것에 주의하시기 바랍니다.)

질문 14.11의 코드를 참고하기 바랍니다.

References [K&R1] § 6.2 p. 121
[K&R2] § 6.2 p. 129
[ANSI] § 3.1.2.5, § 3.2.2.1, § 3.3.16
[C89] § 6.1.2.5, § 6.2.2.1, § 6.3.16
[H&S] § 5.6.2 p. 133

Q 2.8 왜 ==나 !=를 써서 구조체를 비교할 수 없나요?

Answer C 언어의 저수준성을 고려해서, 구조체 비교를 구현하기 위한 좋은 방법은, 컴파일러 입장에서, 존재하지 않습니다. 간단히 바이트 단위로 비교하는 것은, 구조체 필드 사이에 있을지도 모르는 "hole"들을 생각할 때 좋지 않습니

[2] 그러나, 큰 구조체를 함수의 인자로 전달하는 것은 효율성이 매우 떨어집니다. (질문 2.9 참고) 따라서 이 구조체에 대한 포인터를 쓰는 것이 (물론 pass-by-value의 의미가 필요없다는 전제 아래에서) 훨씬 바람직합니다.

다. (이러한 'hole'은 각 필드가 주어진 정렬(alignment) 방법에 맞도록 위치시키기 위해 필요합니다. 질문 2.12를 참고하시기 바랍니다.) 또한 구조체 안에 많은 필드가 있을 때, 모든 것을 필드 단위로 비교하는 것은, 반복되는 코드가 많아서 적당하지 않을 수도 있습니다. 또 컴파일러가 자동으로 만들어 낸 비교 코드가 모든 경우에, 구조체 포인터 멤버들을 제대로 비교할 수 있는 것도 어렵습니다; 예를 들어, 대부분의 경우에, `char *` 타입의 필드는 `==`로 비교하는 것보다 `strcmp`로 비교하는 것이 더 바람직합니다. (질문 8.2 참고)

여러분이 두 개의 구조체를 비교하길 원한다면, 필드 단위로 구조체를 비교하는 함수를 직접 만들어야 합니다.

References [K&R2] § 6.2 p. 129
[ANSI] 4.11.4.1 footnote 136
[ANSI Rationale] § 3.3.9
[H&S] § 5.6.2 p. 133

Q 2.9 구조체를 전달하거나, 리턴할 때 쓰면, 실제로 어떻게 구현되는 것인가요?

Answer 함수의 인자로 구조체가 전달되면, 구조체의 모든 값이 스택에 저장됩니다. (대부분의 프로그래머가 이 복사에 해당하는 overhead를 줄이기 위해서 구조체를 직접 전달하지 않고, 대신 구조체의 포인터를 씁니다.) 어떤 컴파일러들은 자동으로 구조체를 가리키는 포인터를 전달하기도 하지만, 때때로 pass-by-value 개념을 위해, 따로 복사본을 만들어야 할 필요가 발생합니다.

함수의 리턴 타입으로 구조체가 쓰이면, 대부분은 따로 컴파일러가 마련한, 보이지 않은 곳, 일반적으로 함수의 인자 형태로 저장됩니다. 어떤 오래된 컴파일러는 구조체를 리턴할 때 쓸 목적으로 특별한, static 공간을 마련합니다. 이런 경우, 구조체를 리턴하는 함수가 다시 재진입(reentrant)할 수 없기 때문에, ANSI C에 부합하지 않습니다.

References [ANSI] § 2.2.3
[C89] § 5.2.3

Q 2.10 구조체 인자를 받아들이는 함수에 상수값을 (constant value) 전달할 수 있나요?

Answer 이 글을 쓸 당시 C 언어에는 이름 없는(anonymous) 구조체를 만들 방법이 없었습니다. 따라서 임시 구조체 변수를 만들거나 구조체를 리턴하는 함수를

써야 합니다. 질문 14.11을 보기 바랍니다. (GNU C 컴파일러는 구조체 상수를, 추가 기능으로 제공하며, 이후 C 표준에 채택될 가능성이 높습니다.) 질문 4.10을 참고하기 바랍니다.

드디어, [C9X] 표준은 "compound literal"이라는 개념을 소개합니다; 'compound literal'의 한가지 형태는 구조체 상수를 쓸 수 있게 해 줍니다. 예를 들어, `struct point` 타입의 인자를 받는 `plotpoint()`에 상수 구조체를 전달하려면 다음과 같이 할 수 있습니다:

```
plotpoint((struct point){1, 2});
```

(또다른 [C9X] 표준인) "designated initializer"라는 개념을 함께 쓰면, 각각의 멤버 이름을 지정할 수도 있습니다:

```
plotpoint((struct point){.x = 1, .y = 2});
```

References [C9X] § 6.3.2.5, § 6.5.8

Q 2.11 구조체를 화일에서 읽거나 쓰는 방법은?

Answer 구조체를 화일에 쓸 경우는 대개 `fwrite()` 함수를 씁니다:

```
fwrite(&somestruct, sizeof somestruct, 1, fp);
```

그리고, 이렇게 쓴 데이터는 `fread()`를 써서 읽을 수 있습니다. 그러면 `fwrite`가 구조체를 가리키고 있는 포인터를 써서, 주어진 구조체가 저장되어 있는 메모리의 내용을 파일에 기록합니다. (`fread`의 경우에는 파일에서 읽어옵니다.) 이 때 `sizeof` 연산자는 복사할 byte 수를 지정합니다.

ANSI 호환의 컴파일러를 쓰고, 함수 선언이 되어 있는 헤더 파일을 (대개 `<stdio.h>`) 포함했으면 위와 같이 쓰는 것이 좋습니다. 만약 ANSI 이전의 컴파일러를 쓰고 있다면, 첫번째 인자를 다음과 같이 캐스팅해주어야 합니다:

```
fwrite((char *)&somestruct, sizeof somestruct, 1, fp);
```

여기서 중요한 것은, `fwrite`가 구조체에 대한 포인터가 아니라, 바이트를 가리키는 포인터를 받는다는 것입니다.

위와 같이 쓰여진 데이터 화일은 (특히 구조체가 포인터나 실수(floating point)를 포함하고 있을 때) 이식성이 없습니다. (질문 2.12와 20.5를 참고하기 바랍니다). 구조체가 저장되는 메모리 이미지는 컴퓨터와 컴파일러에 매우 의존적입니다. 각각 다른 컴파일러는 각각 다른 크기의 padding을 사용할 수

있으며, 바이트 크기와 순서(endian)가 다를 수 있습니다. 한 시스템에서 구조체를 어떤 파일에 쓰고(write), 다른 시스템에서 이 파일의 내용을 올바르게 읽는 것이 중요하다면, 질문 2.12와 20.5를 참고하기 바랍니다.

만약 구조체가 포인터를 (`char *` 타입의 문자열이나 다른 구조체를 가리키고 있는 포인터) 포함하고 있었다면, 단지 포인터 값만 기록되기 때문에, 화일에서 이 데이터를 읽을 경우, 의미없는 값이 됩니다. 또한 데이터 파일의 이식성을 높이기 위해서는 `fopen()`을 호출할 때, "`b`" flag을 써야 합니다; 질문 12.38을 참고하시기 바랍니다.

좀 더 이식성이 높은 방법은, 구조체를 필드 단위로 읽고 쓰는 함수를 만들어 쓰는 것입니다.

References [H&S] § 15.13 p. 381

2.3 Structure Padding

Q 2.12 제 컴파일러는 구조체 안에 'hole'을 만들어 넣어서 공간을 낭비하고 외부 데이터 화일에 "binary" I/O를 불가능하게 합니다. 이 'padding' 기능을 끄거나 구조체 필드의 alignment를 조정할 수 있을까요?

Answer 대부분의 컴퓨터는 데이터들이 적절하게 align되어 있을 때, 메모리를 가장 효과적으로 읽고 쓸 수 있습니다. 예를 들어 byte-address machine에서 `short int`나 크기가 2인 데이터는 짝수로 된 주소에 위치해 있는 것이 가장 효과적이며, `long int`나 크기가 4인 데이터는 4의 배수로 된 주소에 위치하는 것이 좋습니다. 어떤 머신에서는, 위와 달리 제대로 위치해 있지 않은 데이터는 아예 읽고 쓸 수 없을 수 있습니다.

다음과 같은 구조체가 있다고 가정해 봅시다:

```
struct {
  char c;
  int i;
};
```

대부분의 컴파일러에서 위와 같은 구조체를 만들 때, `char`와 `int` 사이에 어떤 'hole'을 만들어서 `int` 값이 제대로 align될 수 있도록 해 줍니다. (이렇게 두번째 필드를 첫번째 필드를 기준으로 하여, 증가하는 방식의 정렬을 (incremental alignment) 쓰는 것은, 이 구조체 자체가 올바르게 정렬되어 있다는 것을 가정한 것입니다. 결국 컴파일러는 `malloc`이 하는 것처럼, 구조체를 할당할 때 올바른 align을 보장해 주어야 합니다.)

아마도 컴파일러에서 이런 ('padding'이나 'hole'을 어떻게 쓰는지) 제어를 할 수 있는 방법을 제공할 것입니다. (`#pragma`를 써서 할 수 있습니다; 질문 11.20을 참고하기 바랍니다), 그러나 이런 제어를 위한 표준 방법은 없다는 것을 아셔야 합니다.

이러한 padding으로 발생하는, 낭비되는 공간이 염려된다면, 구조체의 멤버를, 큰 크기에서 작은 크기 순으로 정의하면, 낭비되는 공간을 줄일 수 있습니다. bit field를 사용하면 더욱 많은 공간을 절약할 수 있으나, bit field 나름대로의 단점도 존재합니다. (질문 2.26을 참고하기 바랍니다.)

덧붙여 질문 16.7, 20.5도 참고하시기 바랍니다.

References [K&R2] § 6.4 p. 138
[H&S] § 5.6.4 p. 135

Q 2.13 구조체 타입에 `sizeof` 연산자를 썼더니, 제가 예상한 것보다 훨씬 큰 값을 리턴합니다. 왜 그러죠?

Answer 구조체는 필요한 경우 이러한 'padding' 공간을 포함할 수 있습니다. 이는 구조체가 배열로 만들어질 때, 정렬(alignment) 속성이 보존되도록 하기 위한 것입니다. 또 배열로 쓰이지 않을 경우에도 이러한 여분의 'padding'이 남아 있을 수 있습니다. 이는 `sizeof`가 일관된 크기를 리턴하게 하기 위한 것입니다. 질문 2.12를 참고하기 바랍니다.

References [H&S] § 5.6.7 pp. 139–40

2.4 Accessing Members

Q 2.14 구조체 안에서 각각의 필드에 대한 byte offset을 얻을 수 있나요?

Answer ANSI C는 `offsetof()` 매크로를 정의합니다. `<stddef.h>`를 보시기 바랍니다. 만약에 이 매크로가 없다면 다음과 같이 만들 수 있습니다:

```
#define offsetof(type, mem) ((size_t) \
  ((char *)&((type *)0)->mem - (char *)(type *)0))
```

이 방법은 100% 이식성이 뛰어난 것이 아닙니다. 어떤 컴파일러에서는 이 방법을 쓸 수 없을 수 있습니다.

(복잡하기 때문에 조금 더 설명하면, 잘 캐스팅된 널 포인터를 빼는 것은 널 포인터가 내부적으로, 0이 아닌 다른 값이더라도 offset 값을 얻을 수 있게 보장해 줍니다. `(char *)`로 캐스팅하는 것은 offset이 byte offset 단위로 결과를 얻을 수 있게 하기 위한 것입니다. 호환성이 없는 부분이 하나 있는데, 주소 계산을 할 때, `type` 오브젝트가 주소 0번지에 있다고 속이는 부분에 있습니다만, 실제로 참조되지 않기 때문에 access violation이 일어날 가능성은 없습니다.) 쓰는 방법을 알기 위해, 질문 2.15를 참고하기 바랍니다.

References [ANSI] § 4.1.5
[C89] § 7.1.6
[ANSI Rationale] § 3.5.4.2
[H&S] § 11.1 pp. 292–3

Q 2.15 실행 시간(run-time)에 구조체 필드를 이름으로 access할 수 있습니까?

Answer 각각의 필드 이름과 offset을 `offsetof()` 매크로를 써서 테이블에 저장해 두면 됩니다. `struct a`에서 필드 '`b`'의 오프셋은 다음과 같이 얻을 수 있습니다:

```
offsetb = offsetof(struct a, b)
```

만약 이 구조체 변수를 가리키는 포인터 `structp`가 있고, 필드 '`b`'가 `int`일 때, 위에서 계산한 `offsetb`를 쓰면 `b`의 값을 다음과 같이 설정할 수 있습니다:

```
*(int *)((char *)structp + offsetb) = value;
```

Q 2.16 C 언어에, Pascal의 `with` 문장과 같은 기능이 있습니까?

Answer 질문 20.23을 보기 바랍니다.

2.5 Miscellaneous Structure Questions

Q 2.17 배열의 이름이, 배열의 첫번째 요소를 가리키는 포인터처럼 동작한다면, 왜 구조체 이름은 비슷한 방식으로 동작하지 않을까요?

Answer The rule (see question 6.3) that causes array references to "decay" into pointers is a special case that applies only to arrays and that reflects their "second-class" status in C. (An analogous rule applies to functions.) Structures, however, are first-class objects: When you mention a structure, you get the entire structure.

Q 2.18 이 프로그램은 정상적으로 동작하지만 프로그램이 끝났을 때, `core` 파일을 만들어 냅니다. 왜 그런가요?

```
struct list {
  char *item;
  struct list *next;
}

  /* Here is the main program.  */

  main(argc, argv)
  { ...  }
```

Answer 구조체를 정의할 때 세미콜론(';')을 빠뜨렸기 때문에 `main()`이 구조체를 리턴하는 함수로 정의되어 버렸습니다. (중간에 들어간 주석(comment) 때문에 이 버그를 찾아내기가 더욱 힘들 것입니다) 대개 구조체를 리턴하는 함수들은 숨겨진 리턴 포인터(hidden return pointer)를 추가하는 식으로 구현되기 때문에, `main()`이 세개의 인자를 받는 것처럼 만들어집니다. 원래 C start-up code는 `main()`이 두 개의 인자를 받는 것으로 짜여져 있으므로, 이 경우 정상적으로 동작할 수 없습니다. 질문 10.9와 16.4를 참고하기 바랍니다.

References [CT&P] § 2.3 pp. 21–2

2.6 Unions

Q 2.19 structure와 union은 어떻게 서로 다른가요?

Answer union은 모든 필드가 서로 겹쳐서 존재한다는 것을 제외하고는, structure와 같습니다; 따라서 union에서는 한 번에 하나의 필드만을 쓸 수 있습니다. (물론 하나의 필드에 쓰고(write), 다른 필드에서 읽어서, 어떤 타입의 비트 패턴을 조사하거나 다른 식으로 해석하기 위해서 쓸 수도 있지만, 이는 상당히

machine dependent한 것입니다.) union의 크기는, 각각의 멤버들 중 가장 큰 멤버의 사이즈가 됩니다. 반면에 structure의 크기는 각각의 멤버들의 크기를 다 더한 값입니다. (두 가지 경우 모두, 적절한 padding 값이 더해질 수 있습니다; 질문 2.12와 2.13을 참고하기 바랍니다.)

References [ANSI] § 3.5.2.1
[C89] § 6.5.2.1
[H&S] § 5.7 pp. 140–5 esp. § 5.7.4

Q 2.20 `union`을 초기화(initialization)할 수 있습니까?

Answer 현재 C 표준은 `union` 안에, 이름이 있는 첫번째 멤버의 초기화만 인정합니다. (ANSI 이전의 컴파일러에서는 union을 초기화할 수 있는 방법이 전혀 없습니다.)

union을 초기화하기 위해, 여러 제안이 있었지만 아직 채택된 것은 없습니다. (GNU C 컴파일러는 어떤 멤버라도 초기화할 수 있는 확장 기능을 제공하며, 곧 이 기능이 표준으로 채택될 가능성이 높습니다.) If you're really desperate, you can sometimes define several variant copies of a union, with the members in different orders, so that you can declare and initialize the one having the appropriate first member. (These variants are guaranteed to be implemented compatibly, so it's okay to "pun" them by initializing one and then using the other.)

[C9X]는 "designated initializer"를 소개하고 있으며, 어떠한 멤버의 초기값도 쓸 수 있도록 하고 있습니다.

Note [C9X]에 소개된 예는 다음과 같습니다:

```
union { /* ... */ } u = { .any_member = 42 };
```

즉, 위의 예는 union `u`의 멤버인 "`any_member`"를 42로 초기화하고 있습니다.

References [K&R2] § 6.8 pp. 148–9
[C89] § 6.5.7
[C9X] § 6.7.8
[H&S] § 4.6.7 p. 100

Q 2.21 union의 어떤 필드가 현재 쓰여지고 있는지 알 수 있는 방법이 있을까요?

Answer 없습니다. 여러분이 직접 어떤 필드가 쓰여지고 있는지 기록해 두어야 합니다.

```
struct taggedunion {
  enum { UNKNOWN, INT, LONG, DOUBLE, POINTER } code;
  union {
    int i;
    long l;
    double d;
    void *p;
  } u;
};
```

위와 같이 만들고, union에 값을 쓸(write) 때마다 `code`를 올바른 값으로 설정하면 됩니다; 컴파일러가 자동으로 이런 것을 해 주지는 않습니다. (C 언어에서 union은 Pascal의 variant record와는 다릅니다.)

References [H&S] § 5.7.3 pp. 143

2.7 Enumerations

Q 2.22 enumeration(열거형)을 쓰는 것과 `#define`으로 정의한 매크로를 쓰는 것과 차이가 있습니까?

Answer 현재 이 둘에는 약간의 차이가 있습니다. C 표준은 'enumeration이 에러없이 정수형 타입으로 쓰일 수 있다'고 정의할 수 있습니다. (달리 표현하면, 이렇게 명백하게 캐스트를 하지 않고 섞어쓸 수 없었다면 현명하게 enumeration을 썼을 때에만 프로그래밍 에러를 잡아 줄 수 있었을 겁니다. If such intermixing were disallowed without explicit casts, judicious use of enumerations could catch certain programming errors.)

enumeration을 썼을 때에 좋은 점은, 수치 값이 자동적으로 대입되기 때문에, 디버거(debugger)가 열거형 변수를 검사할 때 심볼(symbol) 값으로 보여줄 수 있다는 점입니다. (enumeration과 정수형을 섞어쓰는 것이 오류는 아니지만, 좋은 스타일이 아니기 때문에 어떤 컴파일러는 가벼운 경고를 출력하기도 합니다.) enumeration을 쓸 때의 단점은 이러한 사소한 경고를 프로그래머가 처리해 줘야 한다는 것입니다; 어떤 프로그래머들은 enumeration 변수의 크기를 제어할 수 없다는 것에 불평하기도 합니다.

References [K&R2] § 2.3 p. 39, § A4.2 p. 196
[ANSI] § 3.1.2.5, § 3.5.2, § 3.5.2.2, Appendix E
[C89] § 6.1.2.5, § 6.5.2, § 6.5.2.2, Annex F
[H&S] § 5.5 pp. 127–9, § 5.11.2 p. 153

Note 최신의 GCC에 적절한 디버깅 옵션을 주면, 디버거 GDB에서, 매크로도 값이 아닌, 이름(symbol 값)으로 보여줍니다. 따라서 이 경우, enum이 가지는 장점 하나는 없어진다고 말할 수 있습니다.

Q 2.23 enumeration은 이식성이 있나요?

Answer Enumeration은 그리 오래지 않아 C 언어에 포함되었습니다. (K&R1에는 없었습니다.) 그렇지만 확실하게 C 언어의 일부가 되었으며, 따라서 모든 현대 컴파일러들은 이를 지원합니다. They're quite portable, although historical uncertainty about their precise definition led to their specification in the Standard being rather weak. (질문 2.22를 참고하기 바랍니다.)

Q 2.24 열거형 값을 심볼로 출력할 수 있는 간단한 방법이 있을까요?

Answer 없습니다. 열거형 값을 문자열로 매핑(mapping)시켜주는 함수를 직접 만들어야 합니다. (디버깅 목적으로, 성능 좋은 디버거들은 자동으로 열거형 값을 심볼로 보여주기도 합니다.)

2.8 Bitfields

Q 2.25 아래와 같이 구조체 선언에 콜론(:)을 쓰는게 뭐죠?

```
struct record {
  char *name;
  int refcount : 4;
  unsigned dirty : 1;
};
```

Answer 비트 필드(bitfield)라고 합니다; 콜론(:) 다음에 오는 것은, 콜론 앞 필드의 정확한 크기를 비트 단위로 나타냅니다. (C 언어를 자세히 다루는 여러 책에서 잘 다루고 있습니다.) 여러 비트 단위의 flag이나 작은 값들을 저장하는 데 bitfield를 사용하면 공간을 절약할 수 있습니다. 어떤, 알려진 저장 방식에 정확히 일치하는 구조체를 만드는데 쓰기도 합니다. (Their success at

the latter task is mitigated by the fact that bitfields are assigned left to right on some machines and right to left on others.)

콜론을 써서 비트 필드의 크기를 정하는 것은 구조체나 union에서만 쓸 수 있습니다. 다른 변수 타입의 크기를 직접 정하려고 쓸 수 없습니다. (질문 1.2와 1.3 참고)

References [K&R1] § 6.7 pp. 136–8
[K&R2] § 6.9 pp. 149–50
[ANSI] § 3.5.2.1
[C89] § 6.5.2.1
[H&S] § 5.6.5 pp. 136–8

Q 2.26 Why do people use explicit masks and bit-twiddling code so much instead of declaring bitfields?

Answer Bitfields are thought to be nonportable, although they are no less portable than other parts of the language. You don't know how large they can be, but that's equally true for values of type `int`. You don't know by default whether they're signed, but that's equally true of type `char`. You don't know whether they're laid out from left to right or right to left in memory, but that's equally true of the bytes of *all* types and matters only if you're trying to conform to externally imposed storage layout. (Doing so is always nonportable; see also question 2.12 and 20.5.)

Bitfields are inconvenient when you also want to be able to manipulate some collection of bits as a whole (perhaps to copy a set of flags). You can't have arrays of bitfields; see also question 20.8. Many programmers suspect that the compiler won't generate good code for bitfields; historically, this was sometimes true.

Straightforward code using bitfields is certainly clearer than the equivalent explicit masking instructions; it's too bad that bitfields can't be used more often.

Chapter 3

Expressions

C 언어 디자인 목표 중의 하나는 효과성을 강조합니다 — 즉 C 컴파일러가 상대적으로 작고 만들기 쉽게 하자는 것과, (기계어) 코드를 쉽게 생성할 수 있도록 하자는 것입니다. 이 두가지 목표는 C 언어 specification에 큰 영향을 미쳤습니다. 비록 C 언어가 좀 더 tight하게 정의되었으면 하는 사용자들과 C 언어가 지원하는 것보다 좀 더 많은 것을 (예를 들어 사용자의 실수를 미리 방지하는 기능) 요구하는 사용자들에게는 반가운 내용은 아니었지만 말입니다.

3.1 Evaluation Order

복잡한 expression (수식) 안에서 subexpression을 (부분식) 평가하는 순서는 완전히 컴파일러 마음대로입니다; 이 순서는 여러분이 생각하는 operator precedence와는 (연산자 우선 순위) 별 상관이 없습니다. 여러개의 보이는 부작용이 (multiple visible side effects) 없거나, 한 변수에 여러 개의 side effect가 평행하게 (parallel) 작용하지 않는 한 컴파일러의 평가 순서는 생각할 필요가 없습니다. 그렇지 않다면 이러한 경우, 컴파일러의 행동은 정의되어 있지 않을 수 있습니다. (The behavior may be undefined.)

Q 3.1 이 코드가 왜 동작하지 않을까요?

```
a[i] = i++;
```

Answer 부분식인 '`i++`'은 부작용을 일으킬 수 있습니다 — 즉 `i`의 값이 변경됩니다 — 수식의 다른 부분에 `i`가 또 쓰이기 때문에, 이것이 변경되기 이전의 값일지, 변경된 다음의 값일지 알 수가 없습니다. (K&R에서는 이런 수식이 어

떠한 행동을 취할 지 명세되지 않았다고(unspecified) 하지만, C 표준에서는 이런 수식의 결과에 대해 정의되지 않았다고(undefined) **강력하게** 말하고 있습니다 — 질문 11.33을 참고하기 바랍니다.)

References [K&R1] § 2.12
[K&R2] § 2.12
[C89] § 6.3
[H&S] § 7.12 pp. 227–9

Q 3.2 제 컴파일러로 다음과 같은 코드를 실행하면:

```
int i = 7;
printf("%d\n", i++ * i++);
```

'49'를 출력합니다. 평가 순서(order of evaluation)에 상관없이, '56'을 출력해야 하지 않을까요?

Answer 증가(++), 감소(--) 연산자가 뒤에 쓰일 때에는 먼저 기존의 값을 계산한 다음, 증가/감소하게 됩니다. "after"라는 말이 쓰이긴 하지만 잘못 이해하고 있는 것입니다.

즉 기존의 값을 만든 다음 바로 증가/감소를 할지, 아니면 다른 부분식을 평가하고 난 다음에 할 지는 보장할 수 없습니다. 보장되는 것은 증가/감소 연산이 전체 수식이 끝나기 전에 (ANSI C의 표현을 빌리자면 뒤따르는 "sequence point"로 넘어가기 전에; 질문 3.8 참고) 이루어진다는 것 뿐입니다. 위의 코드에서는 컴파일러가 기존의 값으로 곱한 다음, 증가시키기 때문에 그런 결과가 나오는 것입니다.

부작용이 예상되는 것을 동시에 같은 식에서 쓰면 그 행동양식은 정의되어 있지 않습니다. (대충 말하면 ++, --, =, +=, -=등이 한 수식에서 쓰여서 같은 오브젝트(변수)가 두 번 이상 변경될 경우를 의미합니다; 정확한 정의는 질문 3.8을 참고하기 바라며 "정의되어 있지 않다(undefined)"라는 용어에 대해서는 질문 11.33을 참고하기 바랍니다.) 이러한 상황에서 여러분의 컴파일러가 어떻게 동작할 지 알려고 할 필요가 없습니다 (많은 C 교과서에서 잘못된 설명을 하고 있습니다); K&R에서 언급했던 것처럼, "다양한 컴퓨터에서 어떻게 동작하는 지를 모른다면, 그것을 아예 모르는 것이 낫습니다. (원문: If you don't know how they are done on various machine, that innocence may help to protect you.)

Note A 란의 "after"란 단어는 원문을 보시면 더 빨리 이해하실 수 있습니다:

> Although the postincrement and postdecrement operators ++ and -- perform their operations after yielding the former value, the implication of "after" is often misunderstood.

References [K&R1] § 2.12 p. 50
[K&R2] § 2.12 p. 54
[C89] § 6.3
[H&S] § 7.12 pp. 227–9
[CT&P] § 3.7 p. 47
[PCS] § 9.5 pp. 120–1

Q 3.3 다음과 같은 코드를 여러 컴파일러에서 실행해 보았습니다:

```
int i = 3;
i = i++;
```

어떤 컴파일러는 i가 3이라고 하며, 또 4를 출력하는 컴파일러도 있었습니다. 어떤 컴파일러가 맞는 것인가요?

Answer 여기에는 올바른 답이 없습니다; 위와 같은 수식은 행동 양식이 정의되어 있지 않습니다. 질문 3.1, 3.8, 3.9, 11.33을 참고하기 바랍니다. (i++나 ++i는 둘 다 i + 1과 같지 않습니다. 원하는 것이 단순히 i 값을 증가시키는 것이라면 i=i+1, i+=1, i++, ++i 중 하나를 쓰시기 바랍니다. 질문 3.12를 참고하기 바랍니다)

Q 3.3b 다음과 같은 테크닉은 a와 b를 임시 변수없이 바꾸어 준다고 합니다:

```
a ^= b ^= a ^= b
```

이 코드가 올바른가요?

Answer 이식성(portability)이 없을 뿐만 아니라, 제대로 동작하지도 않는 코드입니다. 위의 코드는 한 'sequence point'에서 변수 a의 값을 두번이나 변경하려 하기 때문에, 행동 양식이 정의되어 있지 않습니다.

예를 들어 다음과 같은 코드를 SCO 최적화 C 컴파일러(icc)에서 돌렸을 경우, b를 123으로 설정하고 a를 0으로 설정한다고 보고되었습니다:

```
int a = 123, b = 7654;
a ^= b ^= a ^= b;
```

질문 3.1, 3.8, 10.3, 22.15c를 참고하기 바랍니다.

Q 3.4 괄호(parentheses)를 써서 평가 순서를 제가 원하는 대로 바꿀 수 있을까요? 만약에 괄호를 쓰지 않더라도 알아서 되지 않나요?

Answer 항상 그런 것은 아닙니다.

연산자 우선 순위와 괄호는 수식 평가의 일부분만을 변경할 수 있습니다. 다음과 같은 수식에서:

```
f() + g() * h()
```

우리는 곱셈이 덧셈보다 먼저 일어난다는 것을 알고 있습니다. 그러나, 세개의 함수 중 어떤 함수가 먼저 호출될지는 알 수 없습니다. 즉, 우선 순위는, 평가에서 일부분 영향을 미치는 것이며, 각 피연산자(operand)의 평가 순서에 영향을 주지는 못합니다.

괄호로 둘러 싸는 것은 어떤 피연산자(operand)가 어떤 연산자(operator)와 연결될 것인지를 결정하지만, 마찬가지로, 평가의 모든 부분에 영향을 줄 수 없습니다. 다음과 같이 괄호를 쓰더라도:

```
f() + (g() + h())
```

함수 실행 순서에 영향을 주지 못합니다. 비슷하게, 질문 3.2에서 나온 식을 괄호로 둘러 싸는 것도 아무 영향을 주지 못합니다. 왜냐하면 `++`는 원래 `*`보다 우선 순위가 높기 때문입니다:

```
(i++) * (i++)     /* WRONG */
```

위 수식은 괄호가 있던 없던 상관없이 'undefined behavior'에 해당합니다.

부분식(subexpression)의 평가 순서가 중요할 때에는, 임시 변수를 만들고 각각 다른 문장(statement)으로 나눠 쓰는 것이 좋습니다.

References [K&R1] § 2.12 p. 49, § A.7 p. 185
[K&R2] § 2.12 pp. 52–3, § A.7 p. 200

Q 3.5 그렇다면 `&&`, `||` 연산자에서는 어떤가요? 다음과 같은 코드를 본 기억이 있거든요.

```
while ((c = getchar()) != EOF && c != '\n')
  ....
```

Answer 위 코드는 이른바 'short-circuit' 예외(exception)라고 하는 예외 사항입니다. 즉, 연산자의 왼쪽의 결과만 가지고도 전체 결과를 알 수 있을 때에는 오른쪽은 평가되지 않습니다 (즉, || 연산자에서 왼쪽이 참이거나, && 연산자에서 왼쪽이 거짓인 경우). 그러므로, 콤마(comma) 연산자와 마찬가지로 이 연산자들은 왼쪽에서 오른쪽으로 평가된다는 것을 보장할 수 있습니다. 게다가 이 연산자들은 모두 (?: 연산자 포함) 추가로 내부적인 'sequence point'를 가지고 있습니다. (질문 3.6, 3.8 참고)

References [K&R1] § 2.6 p. 38, § A7.11–12 pp. 190–1
[K&R2] § 2.6 p. 41, Secs. A7.14–15 pp. 207–8
[ANSI] § 3.3.13, § 3.3.14, § 3.3.15
[C89] § 6.3.13, § 6.3.14, § 6.3.15
[H&S] § 7.7 pp. 217–8, § 7.8 pp. 218–20, § 7.12.1 p. 229
[CT&P] § 3.7 pp. 46–7

Q 3.6 조건에 따라서 &&, || 연산자의 오른쪽이 평가되지 않는다고 보장할 수 있나요?

Answer 보장합니다.

```
if (d != 0 && n / d > 0) {
  /* average is greater than 0 */
}
```

이나,

```
if (p == NULL || *p == '\0') {
  /* no string */
}
```

는 C 코드에서 매우 자주 볼 수 있는 것입니다. 이는 이른바 'short circuit'이라고 합니다. 만약 이 'short circuit'이 없다면, 첫번째 예제의 &&의 오른쪽에서, d가 0일 경우, 0으로 나누는, 'divide by 0' 에러가 발생합니다. 두번째 예제에서는, 만약 p가 널 포인터일 경우, 존재하지 않는 메모리 공간을 참조하는 에러가 발생할 것입니다.

References [ANSI] § 3.3.13, § 3.3.14
[C89] § 6.3.13, § 6.3.14
[H&S] § 7.7 pp. 217–8

Q 3.7 아래 코드에서 왜 `f2`가 먼저 호출되나요? 제 생각으로는 콤마(,) 연산자는 왼쪽에서 오른쪽으로 평가하는 (evaluation) 것으로 알고 있는데요.

```
printf("%d %d", f1(), f2());
```

Answer 콤마(,) 연산자는 왼쪽에서 오른쪽으로 평가하는 것이 보장되어 있습니다. 그러나 함수 호출에서 각 인자를 구별하기 위해 사용하는 콤마(,)는 콤마 연산자가 아닙니다.[1] 함수 호출에서 각 인자의 평가 순서는 정해지지 않았습니다. (*unspecified*) (질문 11.33을 참고하기 바랍니다.)

References [K&R1] § 3.5 p. 59
[K&R2] § 3.5 p. 63
[ANSI] § 3.3.2.2
[C89] § 6.3.2.2
[H&S] § 7.10 p. 224

Q 3.8 이러한 복잡한 규칙을 다 알아야 하나요? 또 'sequence point'라는 것은 무엇인가요?

Answer 'sequence point'라는 것은 어떤 시간대 (전체 수식의 평가가 끝난 시점, 또는 `||`. `&&`, `?:`, 또는 콤마(comma) 연산자, 또는 함수 호출 바로 이전)의 위치를 의미하는 것으로, 모든 부작용이 일어나지 않는다고 보장하는 시점입니다. 표준에서 'sequence point'라고 하는 것은 다음과 같은 상황을 말합니다:

- at the end of a *full expression* (an expression statement or any other expression that is not a subexpression within any larger expression);
- at the `||`, `&&`, `?:`, and comma operators; and
- at a function call (after the evaluation of all the arguments, just before the actual call).

ANSI/ISO C 표준에서는 다음과 같이 정의하고 있습니다:

[1] 만약 함수 호출에서 쓰이는 콤마가 콤마 연산자라면, 하나 이상의 인자를 받는 함수를 만드는 게 불가능해집니다.

> Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.

위에서 두번째 문장이 어려울 수도 있습니다. 즉, 어떤 오브젝트에 값을 쓰는 (write) 경우, 전체 수식은 이 오브젝트에 저장할 값을 계산하기 위한 목적으로 쓰여야 한다는 것을 의미합니다. This rule effectively constrains legal expressions to those in which the accesses demonstrably precede the modification.

질문 3.9를 참고하기 바랍니다.

References [C89] § 5.1.2.3, § 6.3, § 6.6, Annex C
[ANSI Rationale] § 2.1.2.3
[H&S] § 7.12.1 pp. 228–9

Q 3.9 다음 코드에서 배열의 몇번째 요소에 값을 쓸 지는 모르지만, `i`는 단 한번만 증가되는 것으로 생각할 수 있나요?

```
a[i] = i++;
```

Answer 아닙니다! 먼저, 몇번째 요소에 값을 쓰는 지 상관없다면, 왜 저런 식으로 코드를 만들었나요? 그리고, 일단 수식이나 프로그램의 행동 양식이 정의되어 있지 않다라고 (undefined) 했으면, 모든 면에서 어떻게 동작할 지 모릅니다. 만약 'undefined expression'이 두 개의 해석이 가능하다고 해도, 컴파일러가 그 두 개의 해석 중 하나를 선택할 것이라고 가정해서는 안됩니다. 표준은 이런 경우에, 컴파일러가 어떤 선택을 해야 한다고 말하지 않습니다. 그리고, 실제로 어떤 컴파일러는 전혀 다른 방식을 쓰기도 합니다. 위 코드의 경우, 우리는 `a[i]` 또는 `a[i + 1]`에 값이 들어갈지 모를 뿐더러, 전혀 다른 요소에 (또는 아예 이 배열과는 전혀 상관없는 곳에) 쓸 수 있습니다. 그리고 이 문장 실행 뒤에, `i`가 어떤 값을 갖고 있는지 전혀 예측할 수 없습니다. 질문 3.2, 3.3, 11.33, 11.35를 참고하기 바랍니다.

Q 3.10 사람들이 `i = i++`가 'undefined behavior'를 낳는다고 계속 말하지만, 제가 ANSI 표준 호환 컴파일러에서 실험한 결과, 예상했던 값을 얻었습니다.

Answer 질문 11.35를 보기 바랍니다.

Q 3.11 이 모든 복잡한 규칙을 모르고, 평가 순서에서 발생할 지도 모르는 'undefined behavior'를 피할 수 있을까요?

Answer 가장 쉬운 방법은, 여러가지 방식으로 해석되지 않는, 단일한 의미를 가지는 expression을 쓰면, 'undefined behavior'를 피할 수 있습니다. (물론 여기에서 '여러가지 방식으로 해석되지 않는'이라는 말이 사람마다 다를 수도 있지만, `a[i] = i++`나 `i = i++`이 여러가지 방식으로 해석될 수 있다고 생각하는 사람이면, 문제없을 것 같습니다.)

좀 더 자세히 알아보면, 다음과 같은 규칙을 지키면, 컴파일러나 동료 개발자들에게 확실한 의미를 전달하는데 도움이 됩니다. (표준보다 좀 더 보수적인 규칙일 수 있습니다.)

(a) 각 expression이 최대 하나의 object만을 변경할 (modification) 수 있게 합니다: 예를 들어, 간단한 변수, 배열의 한 요소, 포인터가 가리키는 대상 (e.g. `*p`). 여기서 modification이란, `=` 연산자를 쓴 간단한 대입이나, `+=`, `-=`, `*=`와 같은 연산자를 (compound assignment) 쓴 대입, `++`나 `--`를 쓴 증가 또는 감소를 쓴 것을 뜻합니다.

(b) 한 expression에서 어떤 object가 한 번 이상 나오며, 이 object의 값이 변경되는 경우에, 저장될 새로운 값을 계산하기 위해, 기존 값을 쓰는 형태가 되도록 합니다. 이 규칙에 따라서 `i = i + 1`과 같은 수식을 쓸 수 있습니다. 왜냐하면, 비록 `i`가 두 번 쓰였지만, `i`의 새 값을 계산하기 위하여, 기존 `i` 값을 썼기 때문입니다.

(c) 첫번째 규칙을 어긋나는 경우가 꼭 필요하다면, 변경되는 여러 object들이 서로 구별되어 계산되는 것이어야 합니다. 또한, 많아야 두개 또는 세개 정도의 변경이 이루어지도록 하며, 가능하면 지금 소개하는 예의 한 꼴이 되도록 합니다. (이 경우에도 두번째 규칙이 지켜지도록 해야 합니다.)

이 규칙에 따라서 우리는 `c = *p++`와 같은 연산을 쓸 수 있습니다. 왜냐하면, 각 object들이 (`c`와 `p`) 서로 다르게 계산되기 때문입니다. `*p++ = c`도 쓸 수 있습니다, because `p` and `*p` (i.e., `p` itself and what it points to) are both modified but are almost certainly distinct. Similarly, both `c = a[i++]` and `a[i++] = c` are allowed, because `c`, `i`, and `a[i]` are presumably all distinct. Finally, expressions in which three or more things are modified—e.g., `p`, `q`, and `*p` in `*p++ = *q++`, and `i`, `j`, and `a[i]` in `a[i++] = b[j++]`—are allowed *if* all three objects are distinct, i.e., only if two *different* pointers `p` and `q` or two *different* array indices `i` and `j` are used.

(d) You may also break the first rules if you interpose a defined sequence point operator between the two modifications or between the modification and the access. This expression (commonly seen in a `while` loop while reading a line) is legal because the second access of variable `c` occurs after the sequence point implied by `&&`.

```
(c = getchar()) != EOF && c != '\n'
```

Without the sequence point, the expression would be illegal because the access of `c` while comparing it to `'\n'` on the right does not "determine the value to be stored" on the left.

3.2 Other Expression Questions

C 언어는 expression(수식)에서 각각 다른 타입의 operand(피연산자)를 변경하는 적당한 규칙을 가지고 있습니다. 보통 이런 규칙은 매우 간단합니다. 그러나 예상할 수 없는 결과가 나올 수 있으며, 질문 3.14와 3.15가 그러한 상황에 대해 설명해 줍니다. 덧붙여, 이 section의 질문들은 autoincrement operator와 conditional `?:` (또는 "ternary"라고도 하는) operator에 대한 것도 다룹니다.

Q 3.12 만약 수식의 값을 쓰지 않는다면, 변수의 값을 증가시키기 위해, `i++`을 써야 하나요, `++i`를 써야 하나요?

Answer 두 expression 모두, 그 expression의 값이 다른 expression의 내부에 쓰일 경우 (containing expression), 어떻게 값이 해석되느냐에 차이가 있는 것이므로, 간단히 변수의 값을 증가시키기 위한 목적으로 쓴다면, 아무런 차이가 없습니다. (만약 containing expression이 없는 경우에는 full expression이라고 합니다.)

또한 full expression이라는 전제 아래에서는 (`i++`이 `++i`와 같은 것과 비슷하게) `i += 1`과 `i = i + 1`이 완전히 같습니다. (그러나, C++에서는 `++i`의 형식을 더 선호합니다.) 덧붙여 질문 3.3도 참고하시기 바랍니다.

References [K&R1] § 2.8 p. 43
[K&R2] § 2.8 p. 47
[C89] § 6.3.2.4, § 6.3.3.1
[H&S] § 7.4.4 pp. 192–3, § 7.5.8 pp. 199–200

Q 3.13 어떤 숫자 값이 주어진 범위 안에 들어 있는지 검사해야 합니다. 다음과 같이 했는데 왜 제대로 동작하지 않을까요?

```
if (a < b < c)
  ...
```

Answer '<'와 같은 관계 연산자는 (relational operator) 모두 binary operator입니다; 즉, 두 개의 피연산자를 받아서 처리 결과를 참(1), 또는 거짓(0)으로 알려줍니다. 따라서 `a < b < c`는 먼저 `a < b`를 검사하고, 그 결과 0 또는 1을 돌려줍니다. 그래서 결국 평가하는 것은 `0 < c` 또는 `1 < c`가 됩니다. (좀 더 확실히 알기 위해서, `a < b < c`를 `(a < b) < c`로 생각하면 쉽습니다. 왜냐하면 컴파일러가 해석하는 순서와 같기 때문입니다.) 한 수치가 어떤 범위에 포함되는지 알고 싶으면, 다음과 같은 코드를 써야 합니다:

```
if (a < b && b < c)
```

References [K&R1] § 2.6 p. 38
[K&R2] § 2.6 pp. 41–2
[ANSI] § 3.3.8, § 3.3.9
[C89] § 6.3.8, § 6.3.9
[H&S] § 7.6.4, § 7.6.5 pp. 207–10

Q 3.14 이 코드는 왜 동작하지 않을까요?

```
int a = 1000, b = 1000;
long int c = a * b;
```

Answer C 언어의 'integral promotion' 규칙에 의해 위의 곱셈은 'int' 타입의 곱셈으로 계산됩니다. 따라서 그 결과가 오버플로우(overflow) 되거나, 또는 'promotion'하기 전에 잘려나갈(truncate) 수 있습니다. 따라서 'long' 타입의 곱셈을 수행하라고 (강제로) 다음과 같이 알려주어야 합니다:

```
long int c = (long int)a * b;
```

또는 다음과 같이 합니다:

```
long int c = (long int)a * (long int)b;
```

'`(long int)(a * b)`'와 같이 하는 것은 질문의 코드와 똑같은 결과를 만드므로 바람직하지 않습니다. 이런 식으로 캐스팅을 하는 것은 (즉, 곱셈이 끝난 결과를 캐스팅하는 것) 결과 값을 `long int` 타입에 대입할 때, 어차피 자동적으로 변환되는 것이기 때문에 (implicit conversion), 쓰나마나 한 것이 되어 버립니다.

결과값이 실수 타입인 경우, 나눗셈을 할 경우에도 비슷한 문제가 발생할 수 있습니다. 해결 방법은 위와 같습니다.

덧붙여 질문 3.15도 참고하시기 바랍니다.

References [K&R1] § 2.7 p. 41
[K&R2] § 2.7 p. 44
[C89] § 6.2.1.5
[H&S] § 6.3.4 p. 176
[CT&P] § 3.9 pp. 49–50

Q 3.15 왜 아래 코드는 계속 0이 나올까요?

```
double degC, degF;
degC = 5 / 9 * (degF - 32);
```

Answer 어떤 binary operator의 두 operand가 integer인 경우, expression의 나머지 부분이 어떤 타입인지에 상관없이, 정수로 계산합니다. 위 계산에서, 나눗셈 부분은 양쪽이 모두 정수이기 때문에 정수값으로 계산하면, `5 / 9 = 0`이 나옵니다. (부분 식에서 예상하지 못한 방법으로 계산되는 것은 꼭 `int`나 나눗셈에서만 발생하는 것은 아닙니다.) 상수를 `int`가 아닌 `float`이나 `double`로 쓰면, 이 문제는 해결되며, 또는 적절하게 `float`이나 `double`로 캐스팅해도 해결됩니다:

```
degC = (double)5 / 9 * (degF - 32);
```

또는,

```
degC = 5.0 / 9 * (degF - 32);
```

캐스팅할 때, 반드시 하나 또는 두 연산자에 캐스팅이 이루어져야 합니다. 아래와 같이 계산이 끝난 다음에 캐스팅하는 것은 아무런 도움이 되지 못합니다:

```
degC = (double)(5 / 9) * (degF - 32);
```

덧붙여 질문 3.14도 참고하시기 바랍니다.

References [K&R1] § 1.2 p. 10, § 2.7 p. 41
[K&R2] § 1.2 p. 10, § 2.7 p. 44
[ANSI] § 3.2.1.5
[C89] § 6.2.1.5
[H&S] § 6.3.4 p. 176

Q 3.16 어떤 조건에 따라 서로 다른 변수에 복잡한 계산 결과를 대입하려고 합니다. 다음과 같은 코드를 써도 좋습니까?

```
((condition) ? a : b) = complicated_expression;
```

Answer 안됩니다. `?:` 연산자는 대부분 연산자들과 같이 '값(value)'을 만들어 내고, 따라서 이 값에 다른 값을 대입할 수 없습니다. (다른 말로, `?:`는 'lvalue'를 만들어내지 않습니다.) 정말 이런 식의 코드를 써야 한다면, 다음과 같이 할 수 있습니다:

```
*((condition) ? &a : &b) = complicated_expression;
```

(그러나 일반적으로 이런 식의 코드는 지저분해 보이기 때문에 잘 쓰이지 않습니다.)

References [ANSI] § 3.3.15 esp. footnote 50
[C89] § 6.3.15
[H&S] § 7.1 pp. 179–180

Q 3.17 어떤 코드에서 다음과 같은 문장을 봤습니다:

```
a ? b = c : d
```

어떤 컴파일러에서는 위 문장이 동작하는데, 어떤 컴파일러에서는 컴파일되지 않습니다. 왜 그런 것인가요?

Answer C 언어 정의에 따르면, `=`는 `?:`보다 우선 순위가 낮습니다. 따라서 어떤 오래된 컴파일러에서는 위 문장을 다음과 같이 해석하기도 합니다:

```
(a ? b) = (c : d)
```

위 코드는 아무런 의미가 없는 잘못된 코드이므로, 현대 컴파일러들은 원래 문장을 다음과 같이 해석합니다:

```
a ? (b = c) : d
```

여기에서, =의 왼쪽은 단순히 b입니다. 사실 ANSI/ISO C 표준은 컴파일러가 두번째로 해석해야 한다고 씌여 있습니다. (The grammar in the standard is not precedence based and says that any expression may appear between the ? and : symbols.)

따라서, 물어보신 문장은 ANSI 호환 컴파일러에서 당연히 옳은 문장입니다. 그러나 만약 아주 오래된 컴파일러를 쓰고 있다면, 적당히 괄호를 써 주어야 합니다.

References [K&R1] § 2.12 p. 49
[ANSI] § 3.3.15
[C89] § 6.3.15
[ANSI Rationale] § 3.3.15

3.3 Preserving Rules

앞 section에서 말한 "expression(수식)에서 각각 다른 타입의 operand(피연산자)를 변경하는 적당한 규칙"의 의미는 classic C와 ANSI/ISO C에서 약간 바뀌었습니다; 이 section에서는 그 차이에 대하여 설명합니다.

Q 3.18 "sematics of '>' change in ANSI C"라는 문장을 봤는데, 이게 무슨 뜻이죠?

Answer 어떤 민감한 컴파일러들은, 여러분이 쓴 어떤 코드가 ANSI 이전의 "unsigned preserving" 규칙과, ANSI의 "value preserving" 규칙에 따라 다른 식으로 해석될 수 있기 때문에 이러한 경고를 보여줍니다.

약간 혼동스러울 수 있는데, 이 메시지는 > 연산자 때문에 발생한 것이 아닙니다. (사실 이러한 메시지는 거의 모든 C 연산자에서 발생할 수 있습니다.) 이 메시지가 발생한 까닭은 두 개의 서로 다른 타입이 binary operator의 양쪽에 쓰였거나, 작은 타입의 정수 타입이 promote되어야 할 경우에 발생합니다. (만약에 여러분이 생각할 때, 코드에서 unsigned 타입을 쓴 적이 없다고 생각되면, 대부분은 strlen 때문에 이 메시지가 나왔을 것입니다. 표준 C에 따르면, strlen은 size_t 타입을 리턴하며, 이 것은 unsigned 타입입니다.

질문 3.19를 보기 바랍니다.

Q 3.19 "unsigned preserving"과 "value preserving"이란 말이 무슨 뜻이고, 어떤 차이가 있죠?

Answer 이 두 방식은 작은 unsigned 타입이, '큰' 타입으로 promote될 때 어떤 식으로 되느냐에 따라 차이가 있습니다. 간단히 말해서, 부호가 있는, 큰 타입으로 promote될 것인지, 큰 unsigned 타입으로 promote될 것인지가 다릅니다. (여기서 말한 '큰' 타입이 정말로 크냐에 따라 약간 달라질 수 있습니다.)

"unsigned preserving" (또는 signed preserving이라고도 합니다) 규칙에서는, 항상 unsigned 타입으로 promote됩니다. 이 규칙은 매우 간단하다는 장점이 있지만, 가끔 예상치 못한 결과를 내기도 합니다. (아래 예를 보기 바랍니다.)

"value preserving" 규칙에서는, promote되기 전의 타입과 후의 타입이 실제로 크기가 다르냐에 따라 달라집니다—다시 말하면, promote된 후의 타입이 그 전의 타입의 모든 표현 가능한 unsigned 값을, signed 값으로 다 표현할 수 있느냐에 달려 있습니다—실제로 크다면, promote된 후의 타입은 signed입니다. 만약 두 타입이 실제로 크기가 같다면 promote된 후의 타입은 unsigned입니다. (후자의 경우 "unsigned preserving"과 똑같이 동작합니다.)

실제 타입의 크기가 이 결정에 중요한 역할을 하므로, 이 결과는 시스템에 따라 달라질 수 있습니다. 어떤 시스템에서는 `short int`가 `int`보다 작지만, 어떤 시스템에서는 두 타입이 실제로 크기가 같습니다. 또 어떤 시스템에서는 `int`가 `long int`보다 작지만, 어떤 시스템에서는 두 타입이 실제로 크기가 같습니다.

실제로 이 규칙이 적용되는 경우는, binary operator의 한 operand가 `int`이고 다른 한 쪽이 (규칙에 따라 달라질 수 있지만) `int`이거나 `unsigned int`일 경우입니다. 만약에 한 operand가 `unsigned int`인 경우, 다른 한 쪽이 `unsigned`로 변경됩니다—당연하게 한 쪽의 값이 음수일 경우에는 예상치 못한 결과가 발생합니다. (뒤따르는 코드를 보기 바랍니다.) ANSI C 위원회가 처음 만들어졌을 때, 예상치 못한 변환을 최소화하기 위해서 "value preserving" 규칙이 채택되었습니다.

Chapter 4

Pointer

Pointer는 C 언어에서 제공하는 가장 강력하고 인기있는 기능이지만, 초보 programmer들에게는 악몽과도 같은 것입니다. Pointer가 가리켜야 할 것을 가리키고 있지 않을 때 발생하는 혼란은 끝이 없습니다. (사실 포인터에서 발생하는 이러한 문제들은 대개 메모리 할당과 관련이 있습니다; Chapter 7 참고하기 바랍니다.)

4.1 Basic Pointer Use

Q 4.1 도대체, 포인터를 써서 좋은 점이 있나요?

Answer 다음과 같은 장점을 포함하여 여러가지 좋은 점이 있습니다:

- dynamically allocated arrays (질문 6.14, 6.16 참고)
- generic access to several similar variables
- (simulated) by-reference function parameters (질문 4.8, 20.1 참고)
- dynamically allocated structures of all kinds, especially trees and linked lists
- walking over arrays (for example, while parsing strings)
- efficient, by-reference "copies" of arrays and structures, especially as function parameters

(물론, 위의 목록이 전부는 아닙니다.)

덧붙여 질문 6.8도 참고하시기 바랍니다.

Q 4.2 포인터를 선언하고 메모리를 할당하려고 합니다. 그런데 제대로 동작하지 않습니다. 이 코드에 잘못된 부분이 있나요?

```
char *p;
*p = malloc(10);
```

Answer 질문하신 분이 선언한 pointer는 p이지, *p가 아닙니다. pointer가 어떤 곳을 가리키게 하려면 다음과 같이 그냥 pointer의 이름을 써야 합니다:

```
p = malloc(10);
```

그리고, 이 pointer가 가리키는 메모리 공간을 쓰기 위해서는 다음과 같이 'indirection' operator(연산자)인 *를 사용합니다:

```
*p = 'H';
```

질문하신 것같은 실수는 자주 볼 수 있습니다. 왜냐하면 malloc을 지역 변수 선언의 초기값으로 썼다면 다음과 같이 하기 때문입니다:

```
char *p = malloc(10);
```

이 코드를 선언과 지정(assignment)으로 분리시키려면 *를 제거해야 한다는 것을 기억하기 바랍니다.

요약하면, expression에서 p는 pointer이고, *p는 포인터가 가리키는 곳이라는 (이 경우 하나의 char) 것입니다.

덧붙여 질문 1.21, 7.1, 8.3도 참고하시기 바랍니다.

References [CT&P] § 3.1 p. 28

Q 4.3 '*p++'은 'p'를 증가시키는 것인가요, 아니면, 'p'가 가리키는 것을 증가시키는 것인가요?

Answer *, ++, --와 같은, 피연산자가 하나인 연산자는 (unary operator라고 합니다.) 항상 오른쪽에서 왼쪽으로 결합합니다. 따라서 *p++는 *(p++)와 같습니다; 일단 p를 증가시킨 다음, p가 증가하기 전에 가리키던 곳의 값을 리턴합니다. p가 가리키는 것을 증가시키려면 (*p)++을 (또는, 부작용이 일어나도 상관없는 경우, ++*p를 써도 됩니다) 쓰면 됩니다.

References [K&R1] § 5.1 p. 91
[K&R2] § 5.1 p. 95
[ANSI] § 3.3.2, § 3.3.3
[C89] § 6.3.2, § 6.3.3
[H&S] § 7.4.4 pp. 192–3, § 7.5 p. 193, § 7.5.7, 7.5.8 pp. 199–200

4.2 Pointer Manipulations

Q 4.4 `int` 배열에 포인터를 써서 작업하려 합니다. 아래 코드에서 어디가 잘못된 것일까요?

```
int array[5], i, *ip;
for (i = 0; i < 5; i++) array[i] = i;
ip = array;
printf("%d\n", *(ip + 3 * sizeof(int)));
```

Answer 해야될 것보다 더 많은 일을 손수 해버린 셈입니다. C 언어에서 포인터 연산은 항상 포인터가 가리키고 있는 대상의 크기만큼 scale됩니다. 따라서 올바른 코드는 (간단하게) 다음과 같습니다:

```
printf("%d\n", *(ip + 3));
        /* 또는 ip[3] - 질문 6.3 참고 */
```

위 코드는 배열의 세번째 요소를 출력합니다. 이러한 코드에서 여러분이 직접 포인터에, 그 포인터가 가리키고 있는 대상의 크기만큼 곱해주지 않는 것이 정답입니다. 직접 곱해줄 경우, (위 코드에서는 `sizeof(int)`에 따라 다르지만 대개 `array[6]` 또는 `array[12]`) 존재하지 않는 범위에 접근하려고 할 것입니다.

References [K&R1] § 5.3 p. 94
[K&R2] § 5.4 p. 103
[ANSI] § 3.3.6
[C89] § 6.3.6
[H&S] § 7.6.2 p. 204

Q 4.5 문자 포인터로 (`char *` pointer) 어떤 int를 가리키고 있고, 현재, 이 포인터를 써서 다음 int를 가리키게 하려고 합니다. 왜 이 코드가 동작하지 않을까요?

```
((int *)p)++;
```

Answer C 언어에서 캐스팅(casting)은 "실제 비트들이 다른 타입인 것처럼 흉내내는 것"이 아닙니다; 캐스팅은 변환(conversion) 연산이며, 정의에 의해 (대입되거나, ++를 써서 증가될 수 없는) *rvalue*를 만들어 냅니다. (위와 같은 문장을 실수로 만들어냈던, 의도적으로 한 것이든, 또 설령 어떤 컴파일러가—오래된 컴파일러나, gcc의 확장 기능—위와 같은 코드를 원하는 대로 코드를 만들어낸다고 해도 이것은 표준 기능이 아닙니다. 따라서 다음과 같이 하는 것이 옳습니다:

```
p = (char *)((int *)p + 1);
```

(p가 char *이기 때문에) 또는 간단히 다음과 같이 할 수 있습니다:

```
p += sizeof(int);
```

또는, (좀 더 복잡하게 직접 캐스팅을 하는) 다음과 같이 할 수도 있습니다.

```
int *ip = (int *)p;
p = (char *)(ip + 1);
```

될 수 있으면 캐스팅 대신 올바른 타입의 포인터를 쓰는 것이 바람직하다는 것을 염두에 두시기 바랍니다.

덧붙여 질문 16.7도 참고하시기 바랍니다.

References [K&R2] § A7.5 p. 205
[ANSI] § 3.3.4 (esp. footnote 14)
[C89] § 6.3.4
[ANSI Rationale] § 3.3.2.4
[H&S] § 7.1 pp. 179–80

Q 4.6 왜 void *에 사칙연산을 쓸 수 없나요?

Answer 질문 11.24를 보기 바랍니다.

Q 4.7 외부에서 얻은 구조체를 분석하는 코드를 얻었는데, 실행하면 "unaligned address"라는 메시지를 출력하고 끝납니다. 이게 무슨 뜻인가요?

Answer 질문 16.7을 보기 바랍니다.

4.3 Pointers as Function Parameters

Q 4.8 포인터를 인자로 받는 함수를 만들고, 그 함수에서 이 포인터를 초기화하는 함수를 만들었습니다:

```
void f(ip)
int *ip;
{
  static int dummy = 5;
  ip = &dummy;
}
```

그러나 다음과 같이 호출했을 때, 포인터의 값은 변경되지 않는군요.

```
int *ip;
f(ip);
```

Answer C 언어는 인자를 전달할 때, 'call by value' 방식을 쓴다는 것을 기억하기 바랍니다. 위의 함수에서는 단지 복사된 포인터를 변경하기 때문에 실제 인자로 전달한 ip는 변경되지 않습니다. 따라서 이 문제를 해결하려면 함수가 포인터의 주소를 받을 수 있도록 (즉, 인자가 포인터를 가리키는 포인터(pointer-to-pointer)) 만들거나, 함수가 포인터를 리턴하도록 해야 합니다.

```
void f(ipp)
int *ipp;
{
  static int dummy = 5;
  *ipp = &dummy;
}

...

  int *ip;
  f(&ip);
```

이 경우, 우리는 'pass by reference'를 흉내내는 효과를 얻습니다.

또 다른 방법은 포인터를 리턴하는 함수를 만드는 것입니다:

```
int *f()
{
  static int dummy = 5;
  return &dummy;
}
```

질문 4.9와 4.11을 참고하기 바랍니다.

Q 4.9 함수가 범용 포인터(generic pointer)를 레퍼런스(reference)로 받게 하기 위해 다음과 같이 `void **` 포인터를 써도 되나요?

```
void f(void **);
double *dp;
f((void **)&dp);
```

Answer 이식성이 없습니다. 위와 같은 코드는 동작할 수도 있고, 때때로 좋은 코드로 여겨지기도 하지만, 이 코드는 "모든 포인터 타입이 실제로 같은 방식으로 표현된다"를 전제로 하고 있으며, 이 전제는 대부분의 시스템에서 참이지만 그렇지 않은 경우도 있습니다. 질문 5.17을 참고하기 바랍니다.

C 언어에서 범용의 포인터를 가리키는 포인터 (pointer-to-pointer)는 존재하지 않습니다. `void *`가 범용의 포인터로 쓰이는 이유는, 단지 다른 포인터 타입의 값에 대입하거나, 대입될 때, 자동으로 변환(conversion)이 일어나기 때문입니다; 이러한 변환은 `void **`에, `void *`가 아닌 다른 포인터 타입을 대입하거나, 대입될 때는 일어나지 않습니다. 여러분이 `void **`를 쓸 때 (예를 들어 `void **`에 `*` 연산을 써서 원래의 `void *`를 얻으려고 할 때), 컴파일러는 `void *`가, 다른 포인터 타입에서 변환되어서 온 것인지를 알 지 못합니다. 그래서 컴파일러는 말 그대로 `void *`로 인식합니다; 다른 형태의 implicit conversion을 전혀 수행할 수 없습니다.

즉, 여러분이 작업하고자 하는 `void **`는 반드시 어딘가에 실제로 있는, `void *` 값을 가리키고 있는 포인터이어야 합니다. `(void **)&dp`와 같은 코드는, 강제로 캐스팅하므로 컴파일러가 경고를 출력하지 않지만, 이식성이 떨어지며, 때때로 원하는 결과를 얻지 못할 수 있습니다; 질문 13.9 참고. 만약 `void **`가 가리키고 있는 포인터가 `void *`가 아닐 경우, 그리고 그 포인터가 `void *`와는 다른 크기나 내부 표현을 (internal representation) 쓰고 있을 경우, 컴파일러는 그 값을 올바른 방법으로 얻지 못합니다.

질문에 주어진 코드를 동작하게 만들려면, `void *` 타입의 임시 변수를 만드는 것이 바람직합니다:

```
double *dp;
void *vp = dp;
f(&vp);
dp = vp;
```

변수 vp에 할당하고, 얻어내는 방식을 쓰면, 컴파일러가 (필요한 경우) 적당하게 conversion을 수행할 수 있습니다.

다시 말하지만, 이 모든 논쟁은, 포인터가 타입별로 크기나 표현 방식이 다르다는 가정 아래에서 이뤄진 것이며, 최근의 시스템에서는 이런 경우가 드뭅니다. void **에서 발생할 수 있는 문제를 좀 더 쉽게 알기 위해, 다음과 같은 상황을 생각해 봅시다. 일단 int와 double의 크기와 내부 표현 방식이 다르다고 하고, 다음과 같은 함수를 보기 바랍니다.

```
void *incme(double *p)
{
  *p += 1;
}
```

그리고 나서 다음과 같은 코드를 실행합니다:

```
int i = 1;
double d = i;
incme(&d);
i = d;
```

당연히, i는 1만큼 증가합니다. (이 것은 앞에서 vp라는 임시 변수를 써서 설명했던 void ** 코드와 같은 방식을 쓴 것입니다.) 이와 달리, 아래와 같은 코드를 실행했다고 생각해 보기 바랍니다:

```
int i = 1;
incme((double *)&i)  /* WRONG */
```

이 것은, 앞에서 동작하지 않는다고 말했던 (질문에 나온) 코드와 일치합니다.

Q 4.10 다음과 같은 함수가 있을 때:

```
extern int f(int *);
```

상수(constant)의 레퍼런스를 전달할 수 있나요? 다음과 같이 해 봤지만 잘 되질 않습니다:

```
f(&5);
```

Answer 직접적으로 할 수는 없습니다. 일단 임시 변수를 선언하고, 이 변수의 주소를 함수에 전달해야 합니다:

```
int five = 5;
f(&five);
```

일반적으로, 어떤 간단한 값이 아닌, 어떤 값을 가리키는 포인터를 인자로 받는 함수들은 대개, 그 값을 변경하기 위해서 포인터를 인자로 받습니다. 따라서 상수에 대한 포인터를 전달하는 것은 좋은 방식이라고 말하기 곤란합니다. 만약 `f`가 정말로 `int *`를 인자로 받는 함수로 선언되어 있었다면, `const int`에 대한 포인터를 넘겨 주기 전에 먼저 그 함수가 어떤 일을 하는지 분석해 보아야 합니다. (주어진 함수가 `int *`가 아닌 `const int *`를 받는 것으로 선언되어 있었다면, 그 함수 내부에서 인자를 변경하지 않는다는 것이 보장되므로, 안전합니다.)

질문 2.10, 4.8, 20.1을 참고하기 바랍니다.

Q 4.11 C 언어에 "pass by reference"가 존재하나요?

Answer 존재하지 않는다고 할 수 있습니다. 엄밀히 말해 C 언어는 무조건 'call by value'만 사용합니다. 단, 원하는 타입의 포인터 타입을 인자로 받고, `&` 연산자를 써서 주소를 전달함으로써, 'pass by reference'를 흉내낼 수는 있습니다. 또한 배열을 인자로 전달할 경우에는 컴파일러가 자동으로 이 작업을 해 줍니다 (배열 대신 포인터를 사용하는 것은 질문 6.4를 참고하기 바랍니다). 어쨌든, C 언어는 진정한 'pass by reference'나 C++의 레퍼런스 파라메터(parameter)와 같은 것은 없습니다. (그러나 매크로 함수들은 "pass by name"을 제공합니다.) 덧붙여 질문 4.8, 7.9, 12.27, 20.1도 참고하시기 바랍니다.

References [K&R1] § 1.8 pp. 24–5, § 5.2 pp. 91–3
[K&R2] § 1.8 pp. 27–8, § 5.2 pp. 95–7
[ANSI] § 3.3.2.2, esp. footnote 39
[C89] § 6.3.2.2
[H&S] § 9.5 pp. 273–4

4.4 Miscellaneous Pointer Use

Q 4.12 함수를 호출할 때, 포인터를 써서 호출하는 방식을 봤습니다. 왜 이런 일을 하는 거죠?

Answer 원래, 함수를 가리키는 포인터는 * 연산자를 쓸 경우 (그리고 우선 순위를 위해 괄호를 같이 쓸 경우) 진짜(real) 함수 호출로 변경됩니다:

```
int r, func(), (*fp)() = func;
r = (*fp)();
```

위 코드의 마지막 줄의 의미는 다음과 같습니다: `fp`는 함수를 가리키는 포인터이고, `*fp`는 이 함수를 뜻합니다. 그리고 뒤따르는 `()`는 함수 호출에서 쓰이는 인자를 받는 인자 list입니다. 이때 연산자 우선 순위를 고려해서 `*fp`를 괄호로 둘러쌉니다. 이러면 완벽한 함수 호출이 됩니다.

다음과 같은 이론도 있습니다: 함수는 항상 포인터를 써서 불려지고, 수식에서 "진짜" 함수 이름은, 초기화에서 이루어지는 것처럼, 항상 함축적으로 포인터로 변환됩니다 (decay): 질문 1.34 참고. 이 이유는 실제로 `pcc`에서부터 널리 퍼졌고, ANSI 표준에 채택되었습니다. 따라서 `fp`가 위와 같이, 함수를 가리키는 포인터일 때, `r = fp();`는 올바른 표현입니다. (이렇게 쓰는 것은 전혀 혼동스러워 할 필요가 없습니다. 왜냐하면, 함수 이름 뒤에 인자 리스트가 따라올 때, 함수 호출 이외에 다르게 쓰이는 경우가 없기 때문입니다.) 직접 *을 써서 함수 호출하는 것은, 오래된 컴파일러로 이식이 필요한 경우에 많이 쓰입니다.

덧붙여 질문 1.34도 참고하시기 바랍니다.

References [K&R1] § 5.12 p. 116
[K&R2] § 5.11 p. 120
[ANSI] § 3.3.2.2
[C89] § 6.3.2.2
[ANSI Rationale] § 3.3.2.2
[H&S] § 5.8 p. 147, § 7.4.3 p. 190

Q 4.13 완전하게 'generic'한 포인터 타입이 있나요? 제가 쓰는 컴파일러는 함수 포인터를 `void *`로 변환하려고 하니 경고가 발생합니다.

Answer 완벽하게, 아무때나 쓸 수 있는 'total generic'한 포인터 타입은 존재하지 않습니다. `void *`는 아무 오브젝트(또는 데이터) 포인터를 저장할 수 있도록

보장된 포인터입니다; 따라서 함수 포인터를 `void *`로 변환하는 것은 이식성이 떨어집니다. (어떤 시스템들에서, 함수에 대한 주소값은 매우 큰 값입니다—모든 데이터 포인터가 가질 수 있는 값보다 훨씬 큰)

그렇지만, 모든 함수 포인터들은, 쓰이기 전에 원래의 타입으로 다시 변환된다는 가정 아래에서, 서로 다른 함수 포인터를 가리킬 수 있도록 보장되어 있습니다. 따라서 아무런 함수 포인터 타입을 하나 정해서 (일반적으로 `int (*)()`나 `void (*)()`. 즉 `int`나 `void`를 리턴하는 함수 포인터), 'generic'한 함수 포인터로 쓸 수 있습니다.

만약, 데이터와 함수를 구별하지 않는 포인터가 필요하다면, 그리고 이식성이 중요하다면, union을 써서, 한 멤버는 `void *`로, 다른 하나는 (앞에서 말한) 함수 포인터로 만들어 씁니다.

덧붙여 질문 1.22, 5.8도 참고하시기 바랍니다.

References [ANSI] § 3.1.2.5, § 3.2.2.3, § 3.3.4
[C89] § 6.1.2.5, § 6.2.2.3, § 6.3.4
[ANSI Rationale] § 3.2.2.3
[H&S] § 5.3.3 p. 123

Q 4.14 정수값을 포인터로 변환하거나, 그 반대로 해도 상관없나요? 일시적으로 정수값을 포인터에 저장하거나, 포인터를 정수 타입의 변수에 저장해도 되나요?

Answer 예전에는, 포인터가 (비록 `int`인지 `long`인지는 알 수 없으나) 정수로 변환될 수 있다는 것이 보장되어 있었습니다. 즉, 정수가 포인터 값으로 변경되고, 다시 나중에 이 포인터 값을 (충분히 큰) 정수로 바꾸는 등의 변환이 이루어졌습니다. (물론 이런 변환은, 시스템의 주소 구조를 잘 알고 있는 사람들이 써 왔습니다.)

다시 말하면, 이러한 정수/포인터 변환은 선례가 많고 예전부터 지원되어 왔으나, 항상 시스템 의존적인 부분이므로, 이식성이 없습니다. 그리고 이런 변환에는 (비록 오래된 컴파일러들은 경고를 보여주지 않을 수 있지만) 항상 직접 캐스팅해 주어야 합니다.

ANSI/ISO C 표준은, C 언어가 널리 구현되기를 바라는 목적에서, 이러한 정수/포인터 변환 보장을 약화시켰습니다. 따라서 포인터를 정수로 변환하거나, 정수를 포인터로 변환하는 것은 항상 'implementation-defined'입니다. (질문 11.33 참고), 그리고 값의 변환없이 이 변환이 이루어진다는 것도 보장하지 않습니다. (즉, 변환 도중, 값이 바뀔 수도 있다는 말입니다.)

따라서 포인터를 정수로 변환하거나, 정수를 포인터로 변환하는 것은 좋은 습관이 아닙니다. 만약, 정수 또는 포인터를 저장해야 하는 타입이 필요하다면 union을 쓰기 바랍니다.

덧붙여 질문 5.18, 19.25도 참고하시기 바랍니다.

References [K&R1] § A14.4 p. 210
[K&R2] § A6.6 p. 199
[ANSI] § 3.3.4
[C89] § 6.3.4
[ANSI Rationale] § 3.3.4
[H&S] § 6.2.3 p. 170, § 6.2.7 pp. 171–2

Chapter 5

Null Pointers

For each pointer type, C defines a special pointer value, the null pointer, that is guaranteed not to point to any object or function of that type. (The null pointer is analogous to the nil pointer in Pascal and LISP.) C programmers are often confused about the proper use of null pointers and about their internal representation (even though the internal representation should not matter to most programmers). The *null pointer constant* used for representing null pointers in source code involves the integer 0, and many machines represent null pointers internally as a word with all bits zero, but the second fact is *not* guaranteed by the language.

Because confusion about null pointers is so common, this chapter discusses them rather exhaustively. (Question 5.13–5.17 are a retrospective on the confusion itself.) If you are fortunate enough not to share the many misunderstandings covered or find the discussion too exhausting, you can skip to question 5.15 for a quick summary.

5.1 Null Pointers and Null Pointer Constants

These first three questions cover the fundamental definitions of null pointers in the language.

Q 5.1 악명높은 '널 포인터'란 게 도대체 뭔가요?

Answer 언어 정의에 의하면 각각의 포인터 타입에 대해, 특별한 값이 — 널(null) 포인터 — 있어서, 다른 포인터 값들과는 구별되며, 어떤 오브젝트나 함수를

가리키는 포인터와는 항상 구별되는 포인터를 말합니다. 즉, 주소를 리턴하는 `&` 연산자는 절대로 널 포인터를 만들어 낼 수 없으며, 실패하지 않는 한 `malloc()` 함수도 널 포인터를 리턴하지 않습니다 (`malloc()`은 실패할 경우, 널 포인터를 리턴합니다. 그리고 이 것이 널 포인터의 쓰임새 — "할당되지 않은" 또는 "어떠한 것도 가리키지 않는"을 의미하는 특별한 포인터로 쓰이는 것 — 중 하나입니다.)

널 포인터와 초기화되지 않은 포인터[1]와는 개념상 완전히 다릅니다. 널 포인터는 어떠한 오브젝트나 함수도 가리키지 않는 포인터이고, 초기화되지 않는 포인터는 어떤 값을 가지는 지 모르므로, 아무 오브젝트나 가리킬 수 있는 포인터입니다. 질문 1.30, 7.1, 7.31을 참고하기 바랍니다.

위에서 설명한 것처럼, C 언어는 각각의 포인터 타입에 따라 널 포인터가 존재합니다. 그리고 널 포인터의 실제 값은 각 타입에 따라 서로 다를 수 있습니다. 컴파일러가 각 타입에 따른 실제 값으로 변경해 주기 때문에 프로그래머들은 각 타입에 따라 서로 다른 널 포인터의 내부적인 값을 알 필요가 전혀 없습니다 (질문 5.2, 5.5, 5.6을 참고).

References [K&R1] § 5.4 pp. 97–8
[K&R2] § 5.4 p. 102
[C89] § 6.2.2.3
[ANSI Rationale] § 3.2.2.3
[H&S] § 5.3.2 pp. 121–3

Q 5.2 프로그램에서 어떻게 널 포인터를 쓰나요?

Answer 널 포인터 상수를 (*null pointer constant*) 이용합니다. 언어 정의에 따라, 포인터가 쓰일 곳(context)에 상수 0을 — 좀 더 정확히 말해서, 0을 가지는 정수 상수 수식[2]을 — 쓰면 컴파일할 때 자동으로 널 포인터로 변경됩니다. 즉, 초기화나, 대입, 비교할 때, 한쪽이 포인터 타입의 변수나 수식일 경우, 다른 쪽의 0은 컴파일러가 자동으로 널 포인터로 바꾸어 준다는 뜻입니다. 컴파일러는 이 상수 0을 실제 널 포인터 값으로 바꾸어 줍니다. 따라서 다음과 같은 코드는 전혀 문제될 것이 없습니다 (질문 5.3 참고):

```
char *p = 0;
if (p != 0)
```

덧붙여 질문 5.3도 참고하시기 바랍니다.

[1] uninitialized pointer
[2] integral constant expression with the value 0

그러나, 함수의 인자로 포인터를 전달할 경우, 포인터가 쓰일 곳(pointer context)으로 인식하지 못하고, 단순히 정수 0으로 인식할 가능성이 있습니다. 이럴 때에는 널 포인터라는 것을 강제적으로 캐스팅을 써서 알려 주어야 합니다. 예를 들어, UNIX 시스템 콜인 `execl`은 가변 인자 리스트[3]를 받습니다. 이 함수는 인자의 끝을 알리기 위해서 널 포인터를 마지막으로 전달해야 합니다. 즉:

```
execl("/bin/sh", "sh", "-c", "date", (char *)0);
```

마지막 인자의 `(char *)` 캐스팅이 생략될 경우, 컴파일러는 이를 널 포인터로 인식하지 못하고 단순히 정수 0으로 인식합니다. (대부분의 UNIX 매뉴얼은 이 부분을 잘못 설명하고 있으니 주의해야 합니다. 덧붙여 질문 5.11도 참고하시기 바랍니다.)

함수의 프로토타입(prototype)이 있을 경우, 인자 전달은 대입(assignment) 연산으로 인식되기 때문에, 캐스팅을 할 필요가 없습니다. 왜냐하면 함수 프로토타입이 컴파일러에게 적절한 타입이 무엇이라는 것을 알려주기 때문입니다. 따라서 단순히 0만 전달해도, 컴파일러가 알아서 널 포인터로 바꾸어 줍니다. 그러나 가변 인자 리스트를 쓰는 함수의 인자는 프로토타입을 알더라도, 각각의 인자에 대한 타입을 알 수 없으므로 이런 함수의 인자로 쓰인 널 포인터에는 반드시 캐스팅을 써 주어야 합니다. (질문 15.3을 참고하시기 바랍니다.) varargs 함수에 쓰일 것을 대비하고, 함수 프로토타입이 없을 경우도 대비하고, ANSI 호환이 아닌 컴파일러에 쓰일 것을 대비하기 위해 널 포인터 상수 0에 항상 캐스팅을 하는 것이 혼동되지 않고 안전할 수 있습니다.

아래 표는 널 포인터 상수(0)를 그대로 써도 좋은 경우와, 그렇지 않는 경우에 대한 상황을 알려줍니다:

[3] variable length argument list

그냥 0을 써도 좋은 경우:	캐스팅이 반드시 필요한 경우:
초기화(initialization) 대입(assignment) 비교(comparison)	
함수 호출, 프로토타입 있음 (prototype in scope)	함수 호출, 프로토타입 없음 (no prototype in scope)
고정된 인자	함수 호출에서 가변 인자 (variable argument) 사용

References [K&R1] § A7.7 p. 190, § A7.14 p. 192
[K&R2] § A7.10 p. 207, § A7.17 p. 209
[ANSI] § 3.2.2.3
[C89] § 6.2.2.3
[H&S] § 4.6.3 p. 95, § 6.2.7 p. 171

Q 5.3 포인터가 널 포인터인지 비교하기 위해 "`if (p)`"라고 쓰는 것이 안전한가요? 만약 널 포인터의 실제 값이 0이 아닐 경우에는 어떻게 되는 건가요?

Answer 항상 안전합니다. C 언어에서 불리언(boolean) 값이 필요할 때 (예를 들어, `if`, `while`, `for` 그리고 `do`와 같은 문장이나, `&&`, `||`, `!` 그리고 `?:`와 같은 연산자에서), 거짓은 0을 의미하며, 참(true)은 0이 아닌 값을 의미하게 됩니다. 따라서 다음과 같이 쓰게 되면:

```
if (expr)
```

실제로 '`expr`'이 무엇이든지, 컴파일러는 위의 코드를 다음의 코드와 같은 것으로 봅니다.

```
if ((expr) != 0)
```

따라서 '`expr`'을 주어진 '`p`'로 바꾸면, '`if (p)`'가 '`if (p != 0)`'이 됩니다. 그리고 이 수식은 비교를 하는 문맥(comparison context)이기 때문에, 컴파일러는 0이 널 포인터 상수라는 것을 알고, 실제 널 포인터 값으로 변경해줍니다. 크게 특별한 기술을 사용한 것도 아니고, 컴파일러는 두 경우 모두 같은 코드를 만들어 냅니다. 여기에서 실제 널 포인터의 값이 0인지 아닌지는 전혀 문제되지 않습니다.

다음과 같이 불리언 부정(not) 연산자인 `!`를 쓰는 것은:

```
!expr
```

다음과 같이 쓰는 것과 완전히 같습니다:

```
(expr) ? 0 : 1
```

또는 다음과 같이 쓸 수 있습니다:

```
((expr) == 0)
```

따라서, 다음과 같이 쓰는 것은:

```
if (!p)
```

다음과 같이 쓰는 것과 같습니다:

```
if (p == 0)
```

줄여서 (abbreviation) `if (p)`로 쓰는 것은 전혀 문제될 것이 없습니다. 그러나 어떤 사람들은 이런 식으로 코딩하는 것이 나쁜 습관이라고 말합니다 (물론 어떤 사람들은 좋은 습관이라고 말합니다. 질문 17.10을 참고하기 바랍니다).

덧붙여 질문 9.2도 참고하시기 바랍니다.

References [K&R2] § A7.4.7 p. 204
[ANSI] § 3.3.3.3 § 3.3.9, § 3.3.13, § 3.3.14, § 3.3.15, § 3.6.4.1, § 3.6.5
[C89] § 6.3.3.3, § 6.3.9, § 6.3.13, § 6.3.14, § 6.3.15, § 6.6.4.1, § 6.6.5
[H&S] § 5.3.2 p. 122

5.2 The `NULL` Macro

So that a program's use of null pointers can be made a bit more explicit, a standard preprocessor macro, `NULL`, is defined, having as its value a null pointer constant. Unfortunately, although it is supposed to clarify things, this extra level of abstraction sometimes introduces extra level of confusion.

Q 5.4 그럼 NULL은 무엇이고 어떻게 정의되어(`#define`) 있나요?

Answer 스타일에 관한 문제이지만, 대부분의 프로그래머들은 프로그램에서 0을 직접 쓰지 않는 (왜냐하면 0 자체가 수치를 뜻하기도 하고, 널 포인터를 뜻하기도 하기 때문에) 경향이 있습니다. 대신에, 전처리기(preprocessor) 매크로인 `NULL`을 씁니다. 이 매크로는 `<stdio.h>`와 `<stddef.h>`를 포함한 여러 헤더 파일에 정의되어 있으며, 실제로 0으로 정의되어 있으며, 대개는 `(void *)`로 캐스팅되어 있습니다 (질문 5.6 참고). 따라서 정수 0과 널 포인터 상수인 0을 쉽게 구별하기 위해, 널 포인터가 오는 곳에 `NULL`을 사용합니다.

`NULL`을 쓰는 것은 단순히 스타일적인 문제입니다; 전처리기가 `NULL`을 0으로 바꾸어주므로, 컴파일러가 볼 때에는 모두 0으로 보게 됩니다. 따라서 함수 인자로 사용할 경우에는 0을 사용할 때와 마찬가지로 NULL도 캐스팅을 해줘야 할 필요가 있습니다.

질문 5.2의 표에서 0 대신에 NULL을 그대로 쓸 수 있습니다 (캐스팅하지 않는 NULL은 캐스팅하지 않는 0과 같기 때문입니다).

그러나 NULL은 반드시 포인터가 쓰이는 문맥에서만 쓰여야 합니다. 질문 5.9를 참고하기 바랍니다.

References [K&R1] § 5.4 pp. 97–8
[K&R2] § 5.4 p. 102
[ANSI] § 4.1.5, § 3.2.2.3
[C89] § 7.1.6, § 6.2.2.3
[ANSI Rationale] § 4.1.5
[H&S] § 5.3.2 p. 122, § 11.1 p. 292

Q 5.5 널 포인터 값으로 0이 아닌 비트를 포함하는 값을 내부적으로 사용하는 시스템에서는 `NULL`이 어떻게 정의되어 있나요?

Answer 다른 시스템과 똑같습니다. NULL은 시스템이나 컴파일러에 상관없이, 항상 0 또는 `((void *)0)`으로 정의되어 있습니다 (질문 5.4를 참고하기 바랍니다).

프로그래머가 널 포인터를 쓸 경우, 0을 쓰던지 NULL을 쓰던지에 상관없이, 컴파일러가 실제 컴퓨터의 내부적인 널 포인터 값으로 만들어 줍니다. (다시 말하지만, 컴파일러는 0이 포인터가 쓰일 곳에 쓰인 경우, 알아서 널 포인터로 바꾸어 줍니다. 질문 5.2 참고) 그렇기 때문에, 실제 널 포인터가 0이 아닌 다른 값을 갖는 시스템에서 `NULL`을 0으로 정의한 것은 당연합니다. 컴파일러는 0이 포인터가 쓰일 곳에 쓰인 경우, 항상 그 시스템에 맞는, 올바른 널 포인터 값을 만들어 내야 합니다. 상수 0은 널 포인터 상수이며, `NULL`은 단순히 같은 것을 의미하는 또다른 이름일 뿐입니다. (질문 5.13 참고)

C 표준 4.1.5장을 보면, NULL에 대해서 "expands to an implementation-defined null pointer constant,"라고 표현한 문장이 있습니다. 즉, 어떤 형태의 0을 쓰던지, void * 캐스트를 쓸 것인지는 컴파일러가 결정합니다; 질문 5.6, 5.7 참고. 여기에서 "implementation-defined"란 용어가 NULL이 0이 아닌, 내부적으로 쓰이는 널 포인터 값으로 쓰인다는 것을 뜻하지는 않습니다.

덧붙여 질문 5.2, 5.10, 5.17도 참고하시기 바랍니다.

References [ANSI] § 4.1.5
[C89] § 7.1.6
[ANSI Rationale] § 4.1.5

Q 5.6 NULL이 다음과 같이 정의되어 있다면:

```
#define NULL        ((char *)0)
```

함수 인자로 NULL을 전달할 때, 캐스팅하지 않아도 되지 않을까요?

Answer 일반적으로, 꼭 캐스팅해야 합니다. 어떤 컴퓨터들은 포인터의 타입에 따라, 포인터의 내부 표현 방식이 다릅니다. 따라서 문자를 가리키는 포인터가 필요한 곳에 NULL을 그냥 쓰는 것은 문제가 없으나 (왜냐하면 위에서 char에 대한 포인터로 캐스팅을 했기 때문), 다른 타입을 가리키는 포인터가 필요한 곳에 그냥 쓰는 것은 문제가 발생할 수 있습니다. 따라서, FILE *fp = NULL; 이 제대로 동작하지 않을 수도 있습니다. 그렇기 때문에, 반드시 적당한 타입의 포인터로 캐스팅해 주어야 합니다.

그러나 ANSI C는 NULL을 다음과 같이 정의하는 것을 허락하고 있습니다[4]:

```
#define NULL ((void *)0)
```

그러나 NULL을 위와 같이 정의하는 것은 NULL을 잘못쓰는 문제를 어느 정도 (모든 포인터가 같은 내부 표현 방식을 가진 경우에만) 해결해 줄 수 있습니다. (ASCII NUL 문자가 필요한 경우, 질문 5.9 참고) 덧붙여 질문 5.7도 참고하시기 바랍니다.

최근에 나온 "flat" 메모리 구조를 가진 시스템에 익숙해져 있는 프로그래머라면 이러한 "타입에 따라 서로 표현 방식이 다른 포인터"라는 개념이 낯설 것입니다. 질문 5.17을 참고하기 바랍니다.

[4] Because of the special assignment properties of void * pointers, the initialization FILE *fp = NULL; *is* valud if NULL is defined as ((void *)0).

References [ANSI Rationale] § 4.1.5

Q 5.7 제가 쓰는 시스템에서 NULL은 0L로 정의되어 있습니다. 왜 그럴까요?

Answer 어떤 프로그래머들은 포인터가 쓰일 곳이 아닌 곳에 널 포인터를 쓰거나, 또는 캐스팅 없이 쓰는, 부주의한 실수를 저지릅니다. (이런 경우, 항상 동작한다고 보장할 수 없습니다. 질문 5.2, 5.11 참고) 정수보다 큰 크기를 갖는 포인터를 쓰는 시스템에서는 (예를 들어 PC 호환 "large" model을 쓰는 경우; 질문 5.17 참고) NULL을 0L로 정의하는 것이, 몇가지 에러를 잡는데 도움을 줍니다. (어쨌든, 0L도 완벽한 NULL의 정의가 될 수 있습니다. 왜냐하면 "integral constant expression with value 0"에 속하기 때문입니다.) Whether it is wise to coddle incorrect programs is debatable; 질문 5.6과 Chapter 17 참고.

References [ANSI Rationale] § 4.1.5
[H&S] § 5.3.2 pp. 121–2

Q 5.8 함수 포인터 값으로 NULL을 쓰는 것은 괜찮나요?

Answer 좋습니다. (그러나 질문 4.13을 참고하기 바랍니다.)

Q 5.9 NULL과 0이 널 포인터 상수로서 완전히 같다면 도대체 어떤 것을 써야하는 거죠?

Answer 대부분의 프로그래머들은 포인터가 쓰일 곳에서는 반드시 NULL을 써야하는 것으로 믿고 있습니다. 다른 사람들은 NULL이라는 매크로로 0을 가리는 것이 오히려 더 혼동을 가져온다고 생각하고 무조건 0을 쓰는 것을 선호하기도 합니다. 그렇지만 이 질문에는 어떠한 것도 완전한 해답이 되지 못합니다. (질문 9.2와 17.10을 참고하기 바람) C 프로그래머라면 포인터 문맥에서 NULL과 0을 마음대로 쓸 수 있다는 것을 알아야 합니다. 그리고 단지 0만을 쓰는 것도 완벽하다는 것도 알아야 합니다. (0하고는 달리) 포인터가 올 수 있는 곳이면, NULL을 쓰는 것은 좋습니다. 그러나 프로그래머가 (NULL을 0과 다른 것으로 취급하거나 컴파일러에서 특별하게 취급한다고 생각하는 등) 포인터 0과 정수 0을 구별하는데 NULL을 썼느냐 안 썼느냐로 판단하는 것은 좋지 않습니다.

포인터가 쓰일 곳에서만, NULL과 0이 같다는 것을 잊어서는 안됩니다. 포인터가 쓰이지 않는 곳에서 NULL을 쓰는 것은, 만약 제대로 동작한다 할지라도,

쓰면 안됩니다. 왜냐하면 그럴 경우, 잘못된 스타일에 관한 메시지가 발생하기 때문입니다. (게다가 ANSI는 NULL을 `((void *)0)`으로 정의할 수 있도록 하고 있으므로), 포인터가 쓰일 수 없는 곳에 NULL을 쓸 수 없는 시스템도 있습니다. 특히 ASCII null 문자 (NUL)이 쓰일 곳에 NULL을 쓰면 안됩니다. 꼭 매크로를 써야 한다면 다음과 같이 따로 정의해서 쓰는 것이 더 낫습니다:

```
#define NUL '\0'
```

References [K&R1] § 5.4 pp. 97–8
[K&R2] § 5.4 p. 102

Q 5.10 그렇지만 NULL의 값이 (0이 아닌 다른 값으로) 변경될 때를 (내부적으로 0이 아닌 다른 값을 쓰는 컴퓨터) 대비해서 0 대신에 NULL을 쓰는 것이 더 좋지 않을까요?

Answer 아닙니다. (NULL을 쓰는 것이 바람직할 수 있지만, 이런 이유에서가 아닙니다.) 일반적으로 심볼릭(symbolic) 상수가 쓰이는 것은, 실제 값이 변경될 경우를 대비해서 쓰는 경우가 많지만, 0의 자리에 NULL이 쓰이는 것은 이런 이유가 아닙니다. 다시 말하지만 언어 자체가 (포인터 문맥에서) 0이 널 포인터를 만들어 낸다고 정의하고 있습니다. NULL을 쓰는 것은 단순히 스타일에 관한 문제입니다. 질문 5.5, 9.2를 참고하기 바랍니다.

Q 5.11 `NULL`을 쓰지 않은 경우, 아예 동작하지 않는 프로그램을 만들어 내는 컴파일러를 쓴 적이 있습니다.

Answer 이식성이 없게 코드를 작성한 것이 아니라면, 컴파일러가 고장난 것입니다. 아마 질문 5.2와 같이 이식성이 없는 코드를 다음과 같이 썼는지 확인해 보기 바랍니다:

```
execl("/bin/sh", "sh", "-c", "date", NULL);  /* WRONG */
```

`NULL`을 `((void *)0)`으로 정의하고 있는 (질문 5.6 참고) 컴파일러에서 이 코드는 동작합니다[5]. 그러나, 포인터와 정수가 다른 크기나 표현 방식을 쓰는 시스템이라면, 아래와 같은 코드는 (똑같이 잘못된 것이면서) 동작하지 않을 수 있습니다:

[5] Using `(void *)0`, in the guise of `NULL`, instead of `(char *)0` happens to work only because of a special guarantee about the representations of `void *` and `char *` pointers.

```
execl("/bin/sh", "sh", "-c", "date", 0);  /* WRONG */
```

이식성이 뛰어난, 좋은 코드는 다음과 같습니다:

```
execl("/bin/sh", "sh", "-c", "date", (char *)NULL);
```

캐스트를 써서, 위 코드는, 시스템에서 포인터와 정수가 크기가 다르거나, 내부 표현 방식이 다르더라도 동작하며, NULL의 정의가 어떤 식으로 되어 있느냐에 상관없이 동작합니다. (질문 5.2에서 NULL 대신에 0을 쓴 코드는 같은 이유로 올바른 코드입니다; 덧붙여 질문 5.9도 참고하시기 바랍니다.)

Q 5.12 다음과 같은 매크로를 써서 널 포인터가 적절한 타입이 되도록 하고 있습니다:

```
#define Nullptr(type) (type *)0
```

이게 좋은 습관일까요?

Answer 이 트릭은 매우 인기있고 매력적이긴 하지만, 크게 도움이 되지 않습니다. 이 방법은 대입이나 비교에서는 필요 없습니다. (질문 5.2 참고) 게다가 추가적으로 타이핑하는 수고가 필요합니다. 또, 이 코드의 개발자가 널 포인터에 대하여 잘 이해 못하고 있다는 것을 알려주며, 다른 개발자는 위 매크로를 정의한 부분, 사용한 부분, 그리고 포인터를 쓰는 모든 코드를 다시 검사해야 안전할 것입니다. 질문 9.1과 10.2도 보시기 바랍니다.

5.3 Retrospective

In some circles, misunderstandings about null pointers run rampant. These five questions explore some of the reasons why.

Q 5.13 약간 헷갈립니다. NULL은 0인 것이 보장되어 있고, 널 포인터는 아니라고 한 것 같은데 맞습니까?

Answer 일반적으로 “null”과 “NULL”이 혼용되어 쓰이긴 하지만 다음 사항은 알아두셔야 합니다:

(a) 널(null) 포인터에 대한 개념은 질문 5.1에 정의되어 있습니다.

(b) 널 포인터가 실제로 갖게 되는 내부적인 값은 각각의 타입에 따라 다르며 특정 비트가 0이 아닌 값일 수도 있습니다. 실제 널 포인터의 값은 컴파일러 제작자나 필요한 것이지 C 프로그래머는 널 포인터가 실제 어떠한 값인지 전혀 알 필요가 없습니다.

(c) 널 포인터 상수는 정수 상수 0[6]입니다 (질문 5.4 참고).

(d) NULL 매크로는 0으로 정의(`#define`)되어 있습니다. (질문 5.4 참고).

(e) ASCII 널 문자 (NUL)는 모든 비트가 0인 값으로 널 포인터와는 이름만 같을 뿐, 전혀 상관이 없습니다.

(f) 널 문자열(null string)은 빈 문자열 (`""`)을 나타내는 다른 말로, C 언어에서 '널 문자열'이라는 용어를 쓰는 것은 혼동을 가져옵니다. 비어있는 문자열은 널 문자('`\0`')를 말하는 것이지, 널 포인터와는 상관 없습니다.

In other words, to paraphrase the White Knight's description of his song in Through the Looking-Glass, the name of the null pointer is "0", but the name of the null pointer is called "`NULL`" (and we're not sure what the null pointer *is*).

이 글에서는 "널 포인터(null pointer)"라는 용어를 위의 1번의 목적으로 사용합니다. 3 번의 목적으로는 0이나 "널 포인터 상수"라는 표현을 쓰며, 4 번의 목적으로 "NULL"을 사용합니다[7].

References [H&S] § 1.3 p. 325
Through the Looking-Glass, chapter VIII.

Q 5.14 왜 이토록 널 포인터에 대한 논쟁이 많은 것인가요? 왜 이런 질문이 자주 나오죠?

Answer C 프로그래머들은 전통적으로 저수준 기계의 구현 방식(machine implementation)에 대해 좀 더 많은 것을 알기 원하는 경향이 있습니다. 문제는 널 포인터가 소스 코드와 기계 자체에 다 쓰이는 개념이지만, 실제 기계에서는 0이 아닌 다른 값으로 표현될 수 있다는 데에 있습니다. 전처리기 매크로인 `NULL`을 쓰는 것이 나중에 변경될 소지가 있는 것처럼 보이는 것도 문제가 됩니다. 또 "`if (p == 0)`"에서, 사실은 0을 널 포인터로 생각하고 비교하는 것이지만 `p`를 정수형으로 바꾸고 비교하는 것처럼 보일 수도 있습니다. 마지막으로

[6] More precisely, a null pointer constant is an integer constant expression with the value 0, possibly cast to `void *`.

[7] To be very, very precise, the word "null" as a noun means only sense 5, and "`NULL`" means only sense 4; the other usages all use "null" as an adjective, as does the (unrelated) term "null statement." These are admittedly fine points

여러 의미를 가지는 (질문 5.13 참고) "null"이라는 용어를 건성으로 보는 경향이 있습니다.

C 언어에서 이런 혼동을 없애기 위한 방법으로, 널 포인터 용으로 (Pascal의`nil`과 같은) 키워드(keyword)를 만들어 썼다면 좋은 효과를 얻었을 것이라고 생각합니다. 그러면 컴파일러는 "`nil`"을 적절한 널 포인터로 바꾸어 줄 수 있을 것이며, 널 포인터가 올 수 없는 곳에서는 경고를 만들어 낼 수 있을 것입니다. 그러나 현재 C 언어에서 널 포인터를 나타내는 키워드는 "`nil`"이 아니라 "0"입니다. 그리고 널 포인터가 올 수 없는 곳에 0이 쓰이면, 에러가 발생하는 것이 아니라 정수 0으로 해석되며, 널 포인터가 와야 할 자리에 캐스팅하지 않는 0이 오게 되면, 동작하지 않을 수도 있다는 것이, 우리의 이상과는 다릅니다.

Q 5.15 매우 헷갈리는 군요. 널 포인터에 관한 이 모든 사항을 쉽게 알 수 없나요?

Answer 이럴 때 취할 수 있는 방법은 다음 두가지의 간단한 규칙을 따르는 것입니다:

(a) 소스 코드에서 널 포인터 상수가 필요할 경우, 0이나 "NULL"을 씁니다.

(b) 0 또는 NULL이 함수 인자로 쓰일 경우에는, 그 함수 인자 타입에 맞는 포인터로 캐스팅해서 사용합니다.

이 chapter의 나머지 질문들은, 대부분 사람들이 널 포인터가 실제로 갖는 내부적인 값에 대해 잘못 이해하고 있는 (꼭 이것을 이해할 필요는 없습니다.), 부분에 대한 것과 ANSI C에서 개정된 부분에 관한 것입니다. 질문 5.1, 5.2, 5.4를 잘 이해하고 있고, 5.3, 5.9, 5.13, 5.14를 생각해 보았다면, 충분합니다.

Q 5.16 이런 혼동을 없애기 위해, 단순히 널 포인터가 내부적으로 0으로 나타내어 진다고 아예 못 박아 놓는 것이 좋지 않을까요?

Answer 다른 이유없이 그렇게 하는 것은 좋지 않은 생각입니다. 왜냐하면 어떤 기계에서는 널 포인터를 쓸 경우, 자동적으로 하드웨어 트랩(trap)이 발생하도록 해 놓았기 때문에 실제로 널 포인터가 0이 아닌 다른 값으로 쓰일 수 있기 때문입니다.

게다가 널 포인터에 대해 잘 이해하기 위해, 실제로 내부적으로 표현되는 널 포인터의 값을 (0인지 아닌지에 대해) 알 필요가 전혀 없습니다. 단순히 널 포인터가 내부적으로 0으로 표현된다고 생각한다고 해서, 코드를 작성하기 쉬워지는 것도 아닙니다. (잘못된 `calloc()`에 대한 설명을 질문 7.31에서 참고하기 바랍니다.)

그리고 널 포인터가 0이라고 해도 포인터의 크기가 타입에 따라 달라질 수 있기 때문에 여전히 함수 호출에서 캐스팅을 해야 합니다. (만약 질문 5.14에서 말한 것처럼 "`nil`"이 널 포인터로 쓰일 수 있다면 널 포인터가 0인지 아닌지에 대한 논쟁 자체가 의미없는 것이 될 것입니다.)

Q 5.17 (심각하게) 정말로, 0이 아닌 비트 패턴을 사용하는 기계나 각각의 타입에 따라 다른 형태의 포인터를 쓰는 컴퓨터가 있나요?

Answer Prime 50 시리즈는 적어도 PL/I 언어에서 널 포인터를 나타내기 위해 세그먼트 07777, 옵셋 0을 사용합니다. 최근의 모델에서는 `TCNP` (Test C Null Pointer) 명령을 써서 (C 언어에서) 세그먼트 0, 옵셋 0을 사용합니다. 또 오래된 워드 주소를 쓰는(word-addressed) Prime 기계는 바이트 포인터 (`char *`)보다 워드 포인터 (`int *`)가 크기가 더 작습니다.

Data General사의 Eclipse MV 시리즈는 기계 수준에서 세 가지의 포인터 타입을 제공합니다 (워드, 바이트, 비트 포인터). C 언어에서는 두 가지 형태를 사용하며 `char *`와 `void *`는 바이트 포인터로, 나머지 포인터는 워드 포인터로 구현됩니다.

어떤 Honeywell-Bell 메인프레임에서는 널 포인터 값으로 06000을 사용합니다.

CDC Cyber 180 시리즈는 링(ring), 세그먼트, 옵셋 부분으로 이루어진 48 비트 포인터를 사용하며, (링 11의) 대부분의 사용자는 널 포인터로 0xB00000000000를 사용합니다. 오래된 CDC는 "1의 보수(one's complement)" 방식을 사용하며, 잘못된 주소를 포함한 모든 데이터의 예외 상황에 모든 비트가 1인 수치를 사용합니다.

오래된 HP3000 시리즈는 위에서 소개한 다른 시스템처럼, `char *`, `void *` 타입에 대한 포인터와 나머지 포인터들을 바이트 어드레싱과 워드 어드레싱을 써서 구현하며, 두 어드레싱이 서로 다른 방식을 사용합니다.

Symbolics Lisp 컴퓨터에서는, (tagged architecture), 아예 수치로 표현되는 포인터를 제공하지 않습니다. C 널 포인터는 `<NIL, 0>`으로 구현됩니다. (기본적으로 `<object, offset>`을 사용함.)

8086 계열의 프로세서 (PC 호환) 에서는 '메모리 모델'에 따라 16 비트 데이터 포인터와 32 비트 함수 포인터를 쓸 수 있습니다. 또는 32 비트 데이터 포인터와 16 비트 함수 포인터를 쓸 수 있습니다.

어떤 64 비트 Cray 컴퓨터에서는 `int *`를 한 워드의 하위 48 비트로 표현하며, `char *`는 나머지 상위 16 비트를 옵셋으로 써서 표현합니다.

References [K&R1] § A14.4 p. 211

5.4 What's Really At Address 0?

A null pointer should not be thought of as pointing at address 0, but if you find yourself accessing address 0 (either accidentally or deliberately), null pointers may seem to be involved.

Q 5.18 Run-time에 정수 0을 포인터로 캐스팅했을 때, 널 포인터 값으로 쓴다면 괜찮을까요?

Answer 아닙니다. 오직 정수 상수 수식 0만이 (*constant* integral expressions with value 0) 널 포인터가 되는 것이 보장되어 있습니다. 질문 4.14, 5.2, 5.19를 보기 바랍니다.

Q 5.19 시스템 주소 0에 위치해 있는 interrupt vector에 어떻게 접근할 수 있을까요? 포인터에 0을 대입한다면, 컴파일러가 이 것을 널 포인터로 해석해버릴 텐데요.

Answer 주소 0에 실제로 어떤 데이터가 있는지는 순전히 시스템에 의존적인 문제입니다. 따라서 시스템이 제공하는 여러 가지 기법을 쓰면 해결될 수 있을 것입니다. 시스템이 제공하는 문서를 참고하기 바랍니다. (그리고 Chapter 19도 참고하기 바랍니다.) 주소 0에 접근하는 것이 당연한 시스템이라면, 당연히 접근하기 쉬운 방법을 제공했을 것입니다. 몇가지 가능성을 생각해 보면:

(a) 단순히 포인터에 0을 대입한다. (동작한다는 보장은 없지만, 주소 0에 접근하는 것이 의미있는 시스템이라면, 동작할 것입니다.)

(b) 수치 0을 `int` 변수에 대입하고, 이 `int`를 포인터로 변환합니다. (역시, 보장할 수는 없습니다.)

(c) union을 써서, 포인터의 모든 비트를 0으로 합니다:

```
union {
  int *u_p;
  int u_i;  /* assumes sizeof(int) >= sizeof(int *) */
} p;
p.u_i = 0;
```

(d) `memset`을 써서 포인터의 모든 비트를 0으로 합니다:

```
memset((void *)&p, 0, sizeof(p));
```

(e) extern 변수나 배열을 선언하고:

```
extern int location0;
```

어셈블리나 링커의 특별한 명령을 써서 이 심볼이 주소 0을 가리키도록 합니다.

덧붙여 질문 4.14, 19.25도 참고하시기 바랍니다..

References [K&R1] § A14.4 p. 210
[K&R2] § A6.6 p. 199
[ANSI] § 3.3.4
[C89] § 6.3.4
[ANSI Rationale] § 3.3.4
[H&S] § 6.2.7 pp. 171–2

Q 5.20 run-time에 "null pointer assignment"라는 에러가 발생합니다. 이게 무엇을 의미하나요? 또 어떻게 해결할 수 있죠?

Answer 이 메세지는 MS-DOS 용 컴파일러가 자주 발생시키는 전형적인 포인터 에러 메시지입니다. 즉 널 (또는 초기화되지 않은) 포인터를 써서 잘못된 위치 (대개 디폴트 데이터 세그먼트의 옵셋 0)에 어떤 데이터를 쓰려할 때 발생합니다.

어떤 디버거들은 데이터 와치포인트(watchpoint)를 주소 0에 설정할 수 있도록 해 줍니다. 또는 아예 주소 0 근처의 약 20 바이트 정도를 다른 곳에 복사해두고 주기적으로 비교해서 변경되었는지를 검사할 수도 있습니다. 질문 16.8을 참고하기 바랍니다.

Chapter 6

Arrays and Pointers

A strength of C is its unification of arrays and pointers. Pointers can be conveniently used to access arrays and to simulate dynamically allocated arrays. The so-called equivalence between arrays and pointers is so close, however, that programmers sometimes lose sight of the remaining essential differences between the two, imagining either that they are identical or that various nonsensical similarities of identities can be assumed between them.

The cornerstone of the "equivalence" of arrays and pointers in C is the fact that most array references *decay* into pointers to the array's first element, as described in question 6.3. Therefore, arrays are "second-class citizens" in C: You can never manipulate an array in its entirely (i.e., to copy it or pass it to a function), because whenever you mention its name, you're left with a pointer rather than the entire array. Because arrays deacy to pointers, the array subscripting operator `[]` always find itself, deep down, operating on a pointer. In fact, the subscripting expression `a[i]` is defined in terms of the equivalent pointer expression `*((a) + (i))`.

Parts of this chapter (especially the "retrospective" questions in 6.8–6.10) may seem redundant — people have many confusing ways of thinking about arrays and pointers, and this chapter tries to clear them up as best it can. If you find yourself bored by the repetition, stop reading and move on. But if you're confused or if things don't make sense, keep reading until they fall into place.

6.1 Basic Relationship of Arrays and Pointers

Q 6.1 소스 파일에 `char a[6]`이라고 정의하고 `extern char *a`라고 선언해 두었는데 왜 동작하지 않을까요?

Answer 소스 파일에 정의한 것은 문자(`char`)로 이루어진 배열입니다. 그리고 선언한 것은 문자를 가리키는 포인터입니다. 따라서 선언과 정의가 일치하지 않는 경우입니다. 일반적으로, T 타입을 가리키는 포인터(pointer to type T)의 타입은 T 타입의 배열(array of type T)과 다릅니다. 대신 `extern char a[]`을 사용하기 바랍니다.

References [ANSI] § 3.5.4.2
[C89] § 6.5.4.2
[CT&P] § 3.3 pp. 33–4, § 4.5 pp. 64–5

Q 6.2 그러나 `char a[]`는 `char *a`와 같지 않나요?

Answer 전혀 다릅니다. (여러분이 생각한 것은 아마도 함수의 formal parameter에 관한 것일 겁니다. 질문 6.4를 참고하기 바랍니다.) 비록 서로 연관되어 있기는 하지만 (질문 6.3 참고), 또 비슷하게 쓰이기도 하지만 (질문 4.1, 6.8, 6.10, 6.14 참고), 배열은 포인터가 아닙니다.

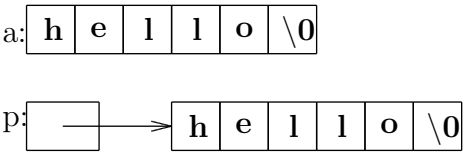
`char a[6]`과 같은 선언은 문자 여섯 개를 저장할 수 있는 공간을 요청하고, 그 결과 "`a`"라는 이름이 이 공간을 대표하게 됩니다. 반면 포인터 선언 `char *p`는 포인터를 저장할 수 있는 공간을 요구하고 이 위치는 "`p`"라고 이름지어집니다. 이 포인터는 이제 어떤 곳도 가리킬 수 있습니다: 정확히 말해 어떤 문자나, 문자로 이루어진 배열의 한 요소를 가리킬 수도 있으며, 아무것도 가리키지 않을 수도[1] 있습니다 (질문 5.1과 1.30을 참고).

그림으로 보는 것이 훨씬 더 나을 것입니다. 다음 두 선언은:

```
char a[] = "hello";
char *p = "world";
```

다음과 같이 데이터를 초기화합니다:

[1] Don't interpret "anywhere" and "nowhere" too broadly. To be valid, a pointer must point to properly allocated memory (see questions 7.1, 7.2, and 7.3); to point definitively nowhere, a pointer must be a null pointer (see question 5.1).



x[3]과 같은 reference가 x가 배열이냐 포인터이냐에 따라 서로 다른 코드를 생성한다는 것을 알아야 합니다. 위에 선언한 코드를 예로 들면, 컴파일러는 a[3]을 봤을 때, 주소 'a'에서 세 개만큼 지나서 그 위치의 문자를 가져옵니다. 그리고 컴파일러가 p[3]과 같은 코드를 봤을 때, 주소 'p'에 셋을 더한 다음, 그 위치의 문자를 가져옵니다. 다시 말하면, a[3]은 'a'라고 이름붙은(named) 곳에서 3만큼 지난 곳을 의미하며, p[3]은 'p'가 가리키는(pointed) 곳에서 3만큼 지난 곳을 의미합니다. 위의 예에서는 a[3]이나 p[3]이 모두 문자 'l'을 가지고 있지만, 컴파일러는 이 곳을 찾기 위해서 다른 코드를 사용합니다. (가장 큰 차이는 'a'와 같은 배열과 'p'와 같은 포인터의 값을 계산할 때 컴파일러가 서로 다른 방법을 써서 계산한다는 것입니다. 이 계산은 subscript 연산자 []를 쓰는지의 여부와 관계없습니다. 다음 질문을 참고하기 바랍니다.)

References [K&R2] § 5.5 p. 104
[CT&P] § 4.5 pp. 64–5

Q 6.3 그럼 "포인터와 배열은 같다(equivalent)"라는 말은 어디에서 나온것일까요?

Answer C 언어에서 가장 혼동을 가져오는 부분은 바로 이 문장을 잘못 이해한 것에서 비롯됩니다. 배열과 포인터가 같다는 (equivalent) 의미는 서로 같다는 (identical) 말도 아니고, 서로 마음대로 바꿔쓸 수 있다는 (interchangeable) 뜻도 아닙니다. 이 말이 의미하는 바는 다음과 같은 정의를 의미합니다:

> T 타입의 배열의 'lvalue'가 수식에서 나타날 때에는 배열의 첫번째 요소를 가리키는 포인터로 변경(decay)됩니다. 여기에는 세 가지의 예외가 있습니다. 그리고 변경된 포인터의 타입은 T 타입을 가리키는 포인터입니다. (이 때 예외 사항은 다음과 같습니다: 배열이 sizeof의 피연산자로 쓰일때, 배열이 & 연산자의 피연산자로 쓰일때, 문자 배열에서, 초기값인 문자열로 쓰일 때[2])

이 정의에 따라서, 내부적으로 배열이 포인터와는 매우 다르지만, 배열이냐 포인터냐에 상관없이 [] 연산자를 쓸 수 있습니다.[3] 배열 a와 포인터 p가 있다고 가정하고, a[i]는 위 정의에 따라서, 배열이 포인터로 퇴화하게(decay)

[2]By "string literal initializer for a character array," we include also wide string literals for arrays of wchar_t.

[3]Strictly speaking, the [] operator is always applied to a pointer; see question 6.10 item 2

되므로, 포인터로 변경되고, 이는 포인터 변수를 쓴 `p[i]`와 의미가 같아집니다. (물론 질문 6.2에서 설명한 것처럼, 실제 메모리 접근은 다릅니다.) 배열의 주소를 다음과 같이 포인터에 대입하면:

```
p = a;
```

`p[3]`은 `a[3]`과 같은 곳을 가리키게 됩니다.

이렇기 때문에, 포인터를 써서 배열에 접근할 수 있는 것이고, 함수 parameter로 쓰일 수 (질문 6.4 참고) 있으며, dynamic array를 흉내낼 수 (질문 6.14 참고) 있습니다.

덧붙여 질문 6.8, 6.10도 참고하시기 바랍니다.

Note 위의 정의의 원문은 다음과 같습니다:

> An lvalue of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T. (The exceptions are when the array is the operand of a `sizeof` or an `&` operator or is a string literal initializer for a character array.)

References [K&R1] § 5.3 pp. 93–6
[K&R2] § 5.3 p. 99
[C89] § 6.2.2.1, § 6.3.2.1, § 6.3.6
[H&S] § 5.4.1 p. 124

Q 6.4 그럼 왜 함수의 formal parameter로 배열과 포인터 선언을 마음대로 바꿔 쓸 수 있다는 것일까요?

Answer 편의상 그런 것입니다.

배열 이름은 즉시 포인터로 바뀌기 때문에[4], 배열은 함수로 전달되지 않습니다. 포인터 파라메터를 선언할 때 배열처럼 쓸 수 있는 것은, 파라메터가 그 함수 내부에서 배열처럼 쓰일 수 있기 때문입니다. 특히, 다음과 같이 배열처럼 선언된 파라메터는:

```
void f(char a[])
{ ... }
```

[4]Since arrays decay immediately into pointers,

컴파일러에 의해 포인터로 인식됩니다. 즉, 다음과 같습니다:

```
void f(char *a)
{ ... }
```

그래서, 함수가 배열에 대한 어떤 작업을 하거나, 파라메터가 함수 내부에서 배열처럼 취급될 때, 함수가 배열을 받는다고 말하는 것은 나쁘지 않습니다.

이런 사항들은 함수의 formal parameter 선언에만 적용되며, 다른 곳에서는 적용될 수 없습니다. 만약 배열을 받는 것처럼 선언한 함수가 신경쓰인다면, 안 쓰면 됩니다; 비록 몇몇은 함수의 선언이나, 함수 내부에서 배열을 받아 쓰는 것처럼 보이게 하는 것이 이득이라고 하지만, 많은 부분에서 혼동을 가져오는 것은 사실입니다. (이러한 변환은 오직 한 번만 일어납니다; `char a2[][]`와 같은 표현은 쓸 수 없습니다. 질문 6.18, 6.19를 보기 바랍니다.)

덧붙여 질문 6.21도 참고하시기 바랍니다.

References [K&R1] § 5.3 p. 95, § A10.1 p. 205
[K&R2] § 5.3 p. 100, § A8.6.3 p. 218, § A10.1 p. 226
[ANSI] § 3.5.4.3, § 3.7.1, § 3.9.6
[C89] § 6.5.4.3, § 6.7.1, § 6.9.6
[H&S] § 9.3 p. 271
[CT&P] § 3.3 pp. 33–4

6.2 Arrays Can't Be Assigned

If an array appears on the right-hand side of an assignment, only the pointer it decays to is copied, not the entire array. Furthermore, an array may not appear on the left-hand side of an assignment (in part because, by the previous sentence, there would never be an entire array for it to receive).

Q 6.5 왜 다음과 같이 할 수 없을까요?

```
extern char *getpass();
char str[10];
str = getpass("Enter password: ");
```

Answer C 언어에서 배열은 "second-class citizens"입니다; 그 결과 배열에 대입 연산을 쓸 수 없습니다 (질문 6.7도 참고하기 바랍니다). 한 배열의 내용을 다른 곳으로 복사하려면, 직접 해야 합니다. `char` 타입의 배열인 경우에는, 대개 `strcpy`를 써서 복사할 수 있습니다:

```
strcpy(str, getpass("Enter password: "));
```

복사하는 대신, 단순히 전해주고 싶다면, 포인터와 대입 연산을 쓸 수 있습니다. 덧붙여 질문 4.1, 8.2도 참고하시기 바랍니다.

References [ANSI] § 3.2.2.1
[C89] § 6.2.2.1
[H&S] § 7.9.1 pp. 221–2

Q 6.6 배열에 대입 연산을 쓸 수 없다면, 다음 코드는 왜 동작할까요?

```
int f(char str[])
{
  if (str[0] == '\0')
    str = "none";
  ...
}
```

Answer 위 코드에서 `str`은 함수 파라메터입니다. 따라서 컴파일러는 위 코드를 질문 6.4에서 다룬 것처럼 바꿉니다. 다시 말하는데, 이 때 `str`은 (타입이 `char *` 인) 포인터입니다. 그리고 포인터에는 대입 연산을 쓸 수 있습니다.

Q 6.7 배열 자체에 값을 대입할 수 없다면 어떻게 배열이 'lvalue'가 될 수 있나요?

Answer *lvalue*라는 단어가 "대입할 수 있는 것"을 뜻하지는 않습니다; "(메모리에) 어떤 위치를 가지고 있는 것"이 더 나은 표현입니다.[5] ANSI C 표준은 "modifiable lvalue"을 정의했고, 배열은 여기에 포함되지 않습니다. 덧붙여 질문 6.5도 참고하시기 바랍니다.

References [C89] § 6.2.2.1
[ANSI Rationale] § 3.2.2.1
[H&S] § 7.1 p. 179.

Note 여기에서 배열이라 함은 subscript 연산자인 `[]`를 쓰지 않은 배열 이름 자체를 의미합니다.

[5]The original definition of "lvalue" did have to do with the left-hand side of assignment statements.

6.3 Retrospective

Because the basic relationship between arrays and pointers occasionally generates so much confusion, here are a few questions about that confusion.

Q 6.8 실제로 배열과 포인터의 차이는 무엇인가요?

Answer 배열은 같은 타입의 연속적인 요소들이 미리 할당된 공간이며, 크기와 위치가 고정되어 있습니다. (An array is a single, preallocated chunk of contiguous elements (all of the same type), fixed in size and location) 포인터는 다른 방법으로 할당된 메모리 공간을 가리켜야 하지만, 다른 곳을 가리키도록 그 값을 바꿀 수 있습니다. (만약 가리키는 공간이 `malloc`을 써서 얻은 것이라면, 크기도 바꿀 수 있습니다.) 포인터는 배열을 가리킬 수 있으며, (`malloc`과 함께 써서) 동적으로 할당한 배열처럼 (흉내내어) 쓸 수 있습니다. 그리고 포인터가 배열보다 더 일반화된 자료 구조입니다 (much more general data structure) (질문 4.1 참고).

배열과 포인터가 같다는 (so-called equivalence) 말 때문에 (질문 6.3 참고), 배열과 포인터는 자주 섞여 쓰입니다 (interchangeable). 특히 `malloc`으로 할당된 메모리에 접근할 때 쓰는 것은 포인터이지만, 배열처럼(`[]`를 써서) 쓰는 경우가 많습니다. 질문 6.14와 6.16을 참고하기 바랍니다. (그러나 `sizeof` 연산자를 쓸 때에는 배열과 포인터가 서로 다르니 주의하기 바랍니다. 질문 7.28 참고)

덧붙여 질문 1.32, 6.10, 20.14도 참고하시기 바랍니다.

Q 6.9 누군가 제게 배열은 단지 포인터 상수라고 (constant pointer) 말한 것이 기억나는 군요. 맞는 말인가요?

Answer 지나치게 간소화한 말입니다 (oversimplification). 배열의 이름은 어떤 값을 대입시킬 수 없는 상수(constant)이지만, 질문 6.2에서 설명하고 보여준 것처럼 배열 자체가 포인터는 아닙니다. 덧붙여 질문 6.3, 6.8, 6.10도 참고하시기 바랍니다.

Q 6.10 아직도 잘 모르겠습니다. 포인터가 배열의 일종인가요, 아니면 배열이 어떤 포인터의 한 종류인가요?

Answer 배열은 포인터가 아니며, 포인터도 배열이 아닙니다. 배열 reference(즉, 값이 쓰이는 곳에 배열을 쓰는 것)는 포인터로 변경됩니다 (질문 6.2, 6.3 참고).

(a) Pointers can simulate arrays (though that's not all; see question 4.1)

(b) There's hardly such a thing as an array (it is, after all, a "second-class citizen"); the subscripting operator `[]` is in fact a pointer operator.

(c) At a higher level of abstraction, a pointer to a block of memory is effectively the same as an array (although this says nothing about other uses of pointers).

But to reiterate, here are two ways *not* to think about it:

(d) "They're completely the same." (False; see question 6.2.)

(e) "Arrays are constant pointers." (False; see question 6.9.)

덧붙여 질문 6.8도 참고하시기 바랍니다.

Q 6.11 `5["abcdef"]`와 같은 이상한 표현을 봤습니다. 이것이 C 언어에서 쓸 수 있는 표현인가요?

Answer 쓸 수 있습니다. Subscript 연산자인 `[]`에는 교환 법칙 (commutative law)이 성립합니다.[6] `a[e]`는 어떤 expression `a`와 `e`에 대해, 하나가 포인터 expression이고, 하나가 정수 수식이란 전제 아래에서, `*((a) + (e))`와 완전히 같습니다 (identical). 이 증명은 다음과 같습니다:

`a[e]`	
`*((a) + (e))`	(*by definition*)
`*((e) + (a))`	(*by commutativity of addition*)
`e[a]`	(*by definition*)

어떤 C 책에서는 이런 것을 자랑삼아 보여주기는 하지만, '혼동스러운 C 컨테스트 (Obfuscated C Contest)'에 쓰이지 않는한, 따로 특별히 쓸모 있는 표현이 아닙니다 (질문 20.36을 참고하기 바랍니다).

C 언어에서 문자열은 `char` 타입 배열이기 때문에, `"abcdef"[5]`란 표현은 틀린 표현이 아니며, `'f'`로 평가됩니다. 이 것을 다음 코드의 줄인 표현으로 생각할 수 있습니다:

```
char *tmpptr = "abcdef";

... tmpptr[5] ...
```

[6] The commutativity is of the array-subscripting operator [] itself; obviously, `a[i][j]` is in general different from `a[j][i]`.

실제 이 코드가 쓰이는 예는 질문 20.10에서 볼 수 있습니다.

References [ANSI Rationale] § 3.3.2.1
[H&S] § 5.4.1 p. 124, § 7.4.1 pp. 186–7

6.4 Pointers to Arrays

Since arrays usually decay into pointers, it's particularly easy to get confused when dealing with the occasional pointer to an entire array (as opposed to a pointer to its first element).

Q 6.12 배열 참조가 포인터로 변환된다면[7], `arr`이 배열일때, `arr`과 `&arr`의 차이는 무엇인가요?

Answer 타입이 서로 다릅니다. 표준 C 언어에서, `&arr`은 포인터를 만들어 내며, 이 포인터의 타입은 배열 T 전체를 가리키는 포인터(pointer to array of T)입니다 (ANSI C 이전의 오래된 C 언어에서는 `&arr`에 쓰인 `&`에서 경고를 발생시키며, 무시됩니다.). 모든 C 언어 컴파일러에서 (`&`를 사용하지 않는) 간단한 배열 reference는 포인터를 만들어내며, 이 타입은 T를 가리키는 포인터(pointer to T)이며, 배열의 첫 요소를 가리킵니다.

다음과 같은 간단한 배열에서:

```
int a[10];
```

`a`에 대한 reference는 "pointer to int"란 타입을 가지며, `&a`에 대한 reference는 "pointer to array of 10 `int`s"란 타입을 가집니다. 다음과 같은 이차원 배열에서는:

```
int array[NROWS][NCOLUMNS];
```

`array` 타입은 "pointer to array of `NCOLUMNS` `int`s"이며, `&array` 타입은 "pointer to array of `NROWS` array of `NCOLUMNS` `int`s"입니다.

(질문 6.3, 6.13, 6.18을 참고하기 바랍니다.)

References [ANSI] § 3.2.2.1, § 3.3.3.2
[C89] § 6.2.2.1, § 6.3.3.2
[ANSI Rationale] § 3.3.3.2
[H&S] § 7.5.6 p. 198

[7] Since array references decay into pointers

Q 6.13 배열을 가리키는 포인터(pointer to an array)는 어떻게 선언하죠?

Answer 보통 이런 작업은 필요하지 않습니다. 사람들이 대개 배열을 가리키는 포인터라고 말하는 것은 배열의 첫번째 요소를 가리키는 포인터를 말하는 경우가 많습니다.

배열 자체를 가리키는 포인터 (a pointer to an array) 대신, 배열의 요소를 가리키는 포인터를 (a pointer to one of the array's elements)생각해보시기 바랍니다. 타입 T의 배열은 (편리하게도) 타입 T의 포인터로 변환됩니다 (decay) (질문 6.3을 참고). 이 포인터에 `[]` (subscript) 연산자를 쓸 수 있고, 또는 증가/감소시켜서 배열의 요소에 접근할 수 있습니다. 정말로 배열 자체를 가리키는 포인터에 (pointers to arrays), subscript 연산자를 쓰거나, 증가시키면, 배열 전체를 건너뛰게 되므로, 배열을 요소로 가진 배열을 (arrays of arrays[8]) 대상으로 할 때에만 의미가 있습니다. (질문 6.18을 참고하기 바랍니다.)

그래도 배열 전체에 대한 포인터가 필요하다면, `int (*ap)[N];`과 같이 선언할 수 있으며, 이 때 `N`은 배열의 크기입니다. (질문 1.21을 참고.) 만약 배열의 크기를 모른다면, `N`은 생략될 수 있습니다. 그러나 이 경우 "크기를 모르는 배열에 대한 포인터"가 되기 때문에 전혀 쓸모가 없습니다.

아래는 간단한 포인터와, 배열에 대한 포인터의 차이에 관한 예입니다. 다음과 같은 선언이 있을 때:

```
int a1[3] = { 0, 1, 2 };
int a2[2][3] = { { 3, 4, 5 }, { 6, 7, 8 } };
int *ip;      /* pointer to int */
int (*ap)[3]; /* pointer to array [3] of int */
```

일차원 배열 `a1`의 요소를 다루기 위해서, `int`에 대한 포인터인, `ip`를 다음과 같이 쓸 수 있습니다:

```
ip = a1;
printf("%d ", *ip);
ip++;
printf("%d\n", *ip);
```

그 결과, 다음과 같이 출력됩니다:

```
0 1
```

[8]This discussion also applies to three- or more dimensional arrays.

만약 a1에 ap를 쓰려 하면:

```
ap = &a1;
printf("%d\n", **ap);
ap++;                   /* WRONG */
printf("%d\n", **ap); /* undefined */
```

처음 printf에서는 0을 출력하고, 그 다음에서는 어떻게 동작할 지 알 수 없습니다 (프로그램이 깨질 수도 있습니다). 배열 자체에 대한 포인터는 a2와 같은, 배열에 대한 배열에서만 의미가 있습니다:

```
ap = a2;
printf("%d %d\n", (*ap)[0], (*ap)[1]);
ap++;                   /* steps over entire (sub)array */
printf("%d %d\n", (*ap)[0], (*ap)[1]);
```

그 결과 다음과 같은 출력을 얻을 수 있습니다:

```
3 4
6 7
```

덧붙여 질문 6.12도 참고하시기 바랍니다.

References [ANSI] § 3.2.2.1
[C89] § 6.2.2.1

6.5 Dynamic Array Allocation

The close relationship between arrays and pointers makes it easy to use a pointer to dynamically allocated memory to simulate an array, of size determined at run time.

Q 6.14 배열의 크기를 실행 시간에 변경할 수 있나요? 고정 크기의 배열 말고 다른 배열이 있나요?

Answer 배열과 포인터의 같은 점 (equivalence) (질문 6.3 참고) 때문에 malloc으로 할당한 메모리를 포인터가 가리키게 한 다음, 배열처럼 쓸 수 있습니다. 다음 코드를 실행한 다음:

```
#include <stdlib.h>
int *dynarray;
dynarray = malloc(10 * sizeof(int));
```

(malloc이 성공했다는 가정 아래에서) dynarray[i]처럼 (i는 0에서 9까지) 쓸 (reference) 수 있습니다. 즉 dynarray를 int a[10]과 같이 선언된 것처럼 쓸 수 있습니다. 정적(static)으로 선언된 배열과 동적으로 할당한 메모리를 가리키는 포인터의 차이는 sizeof 연산자를 쓸 때입니다. 덧붙여 질문 1.31b, 6.16, 7.28, 7.7, 7.29도 참고하시기 바랍니다.

Q 6.15 (함수 인자로) 전달된 크기를 가지는 배열을 선언할 수 있나요?

Answer 현재는 그렇게 할 수 없습니다. C 언어에서 배열의 크기는 컴파일 시간에 정해지는 상수입니다. 일반적으로 이러한 문제는 malloc과 free를 써서 해결합니다.

[C9X]에서는 크기를 변경할 수 있는 'variable-length array(VLA)'을 소개하고 있으며, 이 VLA을 써서 이 문제를 해결할 수 있습니다; 지역 배열은 변수나 다른 수식(대개 함수 파라메터로 전달된 값)을 써서 크기를 지정할 수 있습니다. (GCC는 파라메터화된(parameterized) 배열이라는 확장 기능을 제공합니다.) [C9X]나 GCC를 쓸 수 없다면 malloc()을 쓰는 수 밖에 없습니다. 단, 배열을 다 사용했다면 반드시 free()를 써서 썼던 메모리를 돌려주어야 합니다. 질문 6.14, 6.16, 6.19, 7.22, 또 7.32를 참고하기 바랍니다.

References [ANSI] § 3.4, § 3.5.4.2
[C89] § 6.4, § 6.5.4.2
[C9X] § 6.5.5.2

Q 6.16 다차원(multidimensional) 배열을 동적으로 할당할 수 있나요?

Answer 가장 널리 쓰이는 방법은 포인터의 배열[9]을 할당하고, 각 포인터가 동적으로 할당한 "열(row)"을 가리키게 하는 것입니다. 다음 코드는 2 차원 배열을 동적으로 할당한 것입니다:

```
#include <stdlib.h>
```

[9]Strictly speaking, these aren't arrays but rather objects to be *used* like arrays; see also question 6.14

```
int **array1 = malloc(nrows * sizeof(int *));
for (i = 0; i < nrows; i++)
  array1[i] = malloc(ncolumns * sizeof(int));
```

실제 코드를 쓸 때에는 malloc의 리턴 값을 검사해 주어야 합니다.

배열의 내용을 연속적으로 만들려면, 다음과 같이 약간의 포인터 계산을 해야 합니다 (이 경우, 나중에 각 열을 재배치(reallocation)하기 매우 힘듭니다.)

```
int **array2 = malloc(nrows * sizeof(int *));
array2[0] = malloc(nrows * ncolumns * sizeof(int));
for (i = 1; i < nrows; i++)
  array2[i] = array2[0] + i * ncolumns;
```

둘 (array1 또는 array2) 중 어떤 것이라도, 동적으로 할당한 배열의 각 요소는 일반적인 배열 subscript 연산자인 []를 써서 다룰 수 있습니다: arrayx[i][j] (이 때 0 <= i < nrows와 0 <= j < ncolumns를 만족해야 함).

위와 같이 두번 간접적으로 (double indirection) 메모리에 접근하는 것이 어떤 이유로 인하여 불가능하다면[10], 다음과 같이 메모리를 한 번만 할당할 수도 있습니다. 즉 일차원 배열을 다차원 배열로 흉내내는 것입니다:

```
int *array3 = malloc(nrows * ncolumns * sizeof(int));
```

그러나, 위와 같은 식으로 만들었다면, 각각의 element에 접근하기 위해 조금 더 계산이 필요합니다. 즉, (i, j) 번째 element에 접근하기 위해, array3[i * ncolumns + j] 라고[11] 해야 합니다. 그리고 이 배열은 다차원 배열을 받는 함수에 전달할 수 없습니다. 덧붙여 질문 6.19도 참고하시기 바랍니다.

대신, 배열에 대한 포인터를 (pointers to arrays) 쓸 수 있습니다:

```
int (*array4)[NCOLUMNS] =
  (int (*)[NCOLUMNS])malloc(nrows * sizeof(*array4));
```

또는,

```
int (*array5)[NROWS][NCOLUMNS] =
  (int (*)[NROWS][NCOLUMNS])malloc(sizeof(*array5));
```

[10] Note, however, that double indirection is not necessarily any less efficient than multiplicative indexing

[11] A macro such as `#define Arrayaccess(a, i, j) ((a)[(i) * ncolumns + (j)])` could hide the explicit calculation. Invoking that macro, however, would require parentheses and commas that wouldn't look exactly like conventional C multidimensional array syntax, and the macro would need access to at least one of the dimensions, as well.

도 가능합니다.

그러나 이런 방식은 매우 복잡한 문법을 써야 합니다. (배열 array5에 접근하기 위해서는 (*array5)[i][j]와 같이 씀) 그리고 최대, 한 차원은 실행 시간에 정해져야 합니다.

이 모든 테크닉과 함께, 이렇게 할당한 배열이 더 이상 필요없을 때에는, free를 써서 돌려 주어야 합니다; array1과 array2의 경우 여러 단계를 거쳐야 합니다 (질문 7.23 참고):

```
for (i = 0; i < nrows; i++)
  free((void *)array1[i]);
free((void *)array1);
free((void *)array2[0]);
free((void *)array2);
```

그리고, 이런 배열과 원래 C 언어에서 제공하던, 정적으로 할당된 배열과 섞어 쓸 수 없습니다 (질문 6.20, 6.18 참고).

또, 여기에 소개했던 기술들은 삼차원 또는 그 이상의 고차원 배열에서도 쓸 수 있습니다. 첫번째 기술을 써서 삼차원 배열을 다룬 예입니다:

```
int ***a3d = (int ***)malloc(xdim * sizeof(int **));
for (i = 0; i < xdim; i++) {
  a3d[i] = (int **)malloc(ydim * sizeof(int *));
  for (j = 0; j < ydim; j++)
    a3d[i][j] = (int *)malloc(zdim * sizeof(int));
}
```

덧붙여 질문 20.2도 참고하시기 바랍니다.

References [C9X] § 6.5.5.2

Q 6.17 만약 다음과 같이 코드를 작성하면:

```
int realarray[10];
int *array = &realarray[-1];
```

배열의 첫 요소가 1에서부터 시작하는 것처럼 흉내낼 수 있습니다. 이것이 안전할까요?

Answer 이 테크닉이 매우 매력적으로 보이긴 하지만 (이는 "Numerical Recipes in C"라는 책의 예전판에서 쓴 방식입니다.) 엄밀히 말해 C 표준을 만족하는 방법이 아닙니다. 포인터 연산은 포인터가 할당된 메모리 범위 안을 가리킬 때에나, 배열의 마지막 요소 바로 다음 가상의(imanary) 요소를 가리킬 때에만 정의될 수 있습니다 (Pointer arithmetic is defined only as long as the pointer points within the same allocated block of memory or to the imaginary "terminating" element one past it); 다른 경우에는 포인터 연산을 쓰면 dereference하지 않는다 하더라도, "undefined behavior"를 낳습니다. 따라서 위의 코드에서 뺄셈을 하는 곳에서 잘못된 주소값이 만들어질 수 있습니다 (아마도 주소가 어떤 메모리 세그먼트의 시작점에서 "wrap around" 하기 때문일 수도 있습니다).

References [K&R2] § 5.3 p. 100, § 5.4 pp. 102–3, § A7.7 pp. 205–6
[ANSI] § 3.3.6
[C89] § 6.3.6
[ANSI Rationale] § 3.2.2.3

6.6 Functions and Multidimensional Arrays

It's difficult to pass multidimensional arrays to functions with full generality. The rewriting of array parameters as pointers (as discussed in question 6.4) means that functions that accept simple arrays seem to accept arrays of arbitrary length, which is convenient. However, the parameter rewriting occurs only on the "outermost" array, so the "width" and higher dimensions of multidimensional arrays cannot be similarly variable. This problem arises in past because in standard C, array dimensions must always be compile-time constants; they cannot, for instance, be specified by other parameters of a function.

Q 6.18 이차원 배열을 '포인터를 가리키는 포인터[12]'를 인자로 받는 함수에 전달하면, 제 컴파일러는 경고를 발생합니다.

Answer 배열이 포인터로 변경된다는 규칙(질문 6.3)은 재귀적으로(recursively) 적용되는 규칙이 아닙니다. 배열의 배열(arrays of arrays, 즉 이차원 배열)은 배열을 가리키는 포인터로 (a pointer to an array) 변경되지 (decay), 포인터를 가리키는 포인터로 (a pointer to a pointer) 변경되지 않습니다. 배열을 가리키는 포인터는 어렵고, 또 매우 조심스럽게 다루어야 합니다; 질문 6.13 참고.

[12] a pointer to a pointer

(더욱 혼란스러운 것은, 잘못된 컴파일러들이 존재한다는 것입니다. 오래된 버전의 pcc나 이 컴파일러를 기초로 한 어떤 lint들은 multilevel pointer에 다차원 배열을 대입하는 것을 허락해버립니다)

함수에 이차원 배열을 전달한다면:

```
int array[NROWS][NCOLUMNS];
f(array);
```

함수의 선언은 다음과 같아야 합니다:

```
void f(int a[][NCOLUMNS])
{ ... }
```

또는 다음과 같아야 합니다.:

```
void f(int (*ap)[NCOLUMNS])
  /* ap is a pointer to an array */
{ ... }
```

첫번째 선언에서 컴파일러는 "배열에 대한 배열"을 "배열을 가리키는 포인터"로 바꾸어 줍니다 (질문 6.3과 6.4를 참고); 두번째 선언은 좀 더 정확하게 선언한 것입니다. 이 함수가 배열을 위해 메모리를 추가적으로 할당하지 않기 때문에, 전체 배열의 크기나, 배열의 행(row, 여기에서는 NROWS)의 갯수를 생략할 수 있습니다. 그러나 이 배열의 형태(shape)는 그래도 중요합니다. 따라서 배열의 폭(column)은 꼭 적어 주어야 합니다 (삼차원 이상의 배열에서는 첫번째 열의 수를 제외하고는 다 유지되어야 함).

함수가 이미 '포인터를 가리키는 포인터'를 받도록 선언되어 있다면 여기에 이차원 배열을 직접 전달하는 것은 무의미합니다. 가끔 중간에 임시 포인터 변수를 두어 이차원 배열을 전달하려 하는 코드를 볼 수 있지만:

```
extern g(int **ipp);


int *ip = &array[0][0];
g(&ip);           /* PROBABLY WRONG */
```

이렇게 쓰는 것은 틀렸습니다. 왜냐하면 배열의 형태가 망가졌기('flatten', its shape has been lost) 때문입니다.

덧붙여 질문 6.12, 6.15도 참고하시기 바랍니다.

References [K&R1] § 5.10 p. 110
[K&R2] § 5.9 p. 113
[H&S] § 5.4.3 p. 126

Q 6.19 폭(width)을 컴파일 시간에 알 수 없는, 이차원 배열을 인자로 받는 함수를 만들 수는 없을까요?

Answer 쉬운 일이 아닙니다. 한가지 방법은 `[0][0]` 요소의 주소를 전달하고, 배열의 subscript를 흉내내는 것입니다:

```
void f2(int *aryp, int nrows, int ncolumns)
{ ...  array[i][j]에 접근하기 위해
       aryp[i * ncolumns + j]를 씀 ...  }
```

수동으로 접근하기 위해, 열의 갯수가 (the number of rows) 아닌, 각 열의 폭을 (the "width" of each row) 뜻하는 `ncolunms`을 쓰는 것에 주의하기 바랍니다; 바꿔 쓰는 실수가 흔합니다.

이 함수는 질문 6.18에 나온 배열을 다음과 같이 전달받을 수 있습니다:

```
f2(&array[0][0], NROWS, NCOLUMNS);
```

예전에도 말했지만, 이런 식으로 다차원 배열의 subscript를 직접 하는 것은 ANSI C 표준에 정확히 부합하지 않습니다; 공식 설명에 따르면 `(&array[0][0])[x]`와 같은 표현은 `x >= NCOLUMNS`일 경우, "undefined behavior"에 속합니다.

[C9X]는 가변 크기 배열을 제공하며, [C9X] 확장 기능을 제공하는 컴파일러가 널리 퍼지게 되면, 이 방법이 가장 바람직한 방법이 될 수 있을 것입니다. (GCC는 이미 가변 크기 배열을 제공합니다.)

함수가 다양한 크기를 가지는 다차원 배열을 전달받을 수 있게 하는 한 방법으로, 질문 6.16에 나오는, 배열을 동적으로 시뮬레이션하는 방법이 있습니다.

질문 6.18, 6.20, 6.15를 참고하기 바랍니다.

References [ANSI] § 3.3.6
[C89] § 6.3.6
[C9X] § 6.5.5.2

Q 6.20 정적 또는 동적으로 할당된 다차원 배열을 함수에 전달할때, 서로 구별하지 않고 쓸 수 있는 방법이 있을까요?

Answer 완벽한 방법은 없습니다. 다음 선언이 있다고 할 때:

```
int array[NROWS][NCOLUMNS];
int **array1;                /* ragged */
int **array2;                /* contiguous */
int *array3;                 /* "flattened" */
int (*array4)[NCOLUMNS];
int (*array5)[NROWS][NCOLUMNS]
```

포인터들은 질문 6.16에 나온 것처럼 초기화되어 있다고 가정하고 함수 선언은 다음과 같다고 가정합니다:

```
void f1a(int a[][NCOLUMNS], int nrows, int ncolumns);
void f1b(int (*a)[NCOLUMNS], int nrows, int ncolumns);
void f2(int *aryp, int nrows, int ncolumns);
void f3(int **pp, int nrows, int ncolumns);
```

이때, `f1a()`와 `f1b()`는 일반적인 이차원 배열을 인자로 받으며, `f2()`는 펼쳐진(flattened) 이차원 배열을 인자로 받으며, `f3()`은 '포인터를 가리키는 포인터'를 인자로 받습니다 (질문 6.18과 6.19를 참고). 따라서 다음과 같이 호출할 수 있습니다:

```
f1a(array, NROWS, NCOLUMNS);
f1b(array, NROWS, NCOLUMNS);
f1a(array4, nrows, NCOLUMNS);
f1b(array4, nrows, NCOLUMNS);
f1a(*array5, NROWS, NCOLUMNS);
f1b(*array5, NROWS, NCOLUMNS);
f2(&array[0][0], NROWS, NCOLUMNS);
f2(*array, NROWS, NCOLUMNS);
f2(*array2, nrows, ncolumns);
f2(array3, nrows, ncolumns);
f2(*array4, nrows, NCOLUMNS);
f2(**array5, NROWS, NCOLUMNS);
f3(array1, nrows, ncolumns);
f3(array2, nrows, ncolumns);
```

대부분의 컴퓨터에서 다음의 호출들도 동작할 것이나, 복잡한 캐스트가 필요하고, 동적인 `ncolumns`가 (대개 변수) 정적인 NCOLUMNS와 (대개 매크로 상수) 일치할 경우에만 쓸 수 있습니다:

```
f1a((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1a((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1b((int (*)[NCOLUMNS])array3, nrows, ncolumns);
f1b((int (*)[NCOLUMNS])array3, nrows, ncolumns);
```

동적 또는 정적으로 할당한 배열 모두에 편하게 쓸 수 있는 함수는 `f2`입니다. 그러나 "ragged" 형태로 만든 배열인 `array1`에는 동작하지 않을 수도 있습니다. 그리고 `&array[0][0]`을 (또는 `*array`) `f2()`에 전달하는 것은 표준에 정확히 맞지 않습니다; 질문 6.19를 참고하기 바랍니다.

만약 위에 나열한 호출이 어떻게 동작하고 어떻게 작성되는지 이해한다면, 그리고 왜 다른 조합이 동작하지 않는지 않다면, 여러분은 C 언어에서 배열과 포인터에 대해 매우 잘 이해하고 있다고 말할 수 있습니다.

위에서 어떤 것을 쓸 것인지 결정하는데 고민이 된다면, 질문 6.16에 나온 것처럼, 다양한 크기를 가지는 다차원 배열을 모두 동적으로 만드는 것이 한 방법이 될 수 있습니다 — 만약 모든 배열이 질문 6.16의 `array1`, `array2`처럼 만들어 진다면 — 모든 함수는 `f3()`처럼 만들어야 합니다.

6.7 Sizes of Arrays

The `sizeof` operator will tell you the size of an array if it can, but it's not able to if the size is not known or if the array has already decayed to a pointer.

Q 6.21 배열이 함수의 파라메터로 전달될 경우 왜 `sizeof` 연산자는 제대로 동작하지 않을까요? 아래와 같이 테스트 함수를 만들었더니, 10이 아닌 4를 출력합니다.

```
f(char a[10])
{
  int i = sizeof(a);
  printf("%d\n", i);
}
```

Answer 컴파일러는 배열 파라메터를 포인터로 선언된 것으로 간주합니다 (즉, 위 함수에서는 파라메터를 `char *a`로 여깁니다. 질문 6.4 참고). 그래서 `sizeof` 연산자가 배열의 크기가 아닌, 포인터의 크기를 리턴합니다.

덧붙여 질문 1.24, 7.28도 참고하시기 바랍니다.

References [H&S] § 7.5.2 p. 195

Q 6.22 한 파일에 배열이 `extern`으로 선언되어 있을 때 (즉, 실제 크기와 함께 다른 파일에 정의되어 있음), 배열의 크기를 알 수 있을 까요? `sizeof` 연산자로는 안되는 것 같습니다.

Answer 질문 1.24를 보기 바랍니다.

Q 6.23 배열 안에 몇 개의 요소가 있는 지 알고 싶으면 어떻게 하나요? `sizeof` 연산자는 배열의 크기를 byte 단위로 돌려주는데요.

Answer 전체 배열의 크기를, 한 요소의 크기로 나누면 됩니다.

```
int array[] = { 1, 2, 3 };
int narray = sizeof(array) / sizeof(array[0]);
```

References [ANSI] § 3.3.3.4
[C89] § 6.3.3.4

Chapter 7

Memory Allocation

많은 사람들이 C 언어에서 pointer가 가장 배우기 힘들다고 말합니다. 그러나 사실 이 말은 pointer보다 pointer가 가리키는 메모리를 관리하기가 힘들다는 뜻입니다. C 언어는 상대적으로 저수준의 언어이기 때문에, 프로그래머가 메모리를 직접 할당해야 합니다. 제대로 할당되지 않은 메모리를 가리키는 포인터를 쓰는 것은 심각한 버그를 발생하는 가장 주된 원인이 됩니다.

7.1 Basic Allocation Problems

여러분이 직접 `malloc`을 부르지 않는다해도, 여러분이 쓰려고 하는 메모리가 제대로 초기화되었는지 검사해야 합니다.

Q 7.1 이 코드는 왜 동작하지 않을까요?

```
char *answer;
printf("Type something:\n");
gets(answer);
printf("You typed \"%s\"\n", answer);
```

Answer `gets()`에 전달된 포인터 변수인 `answer`는 문자열이 저장될 어떤 공간을 가리키고 있어야 합니다. 위의 코드에서는 `answer`가 어디를 가리키는 지 알 수 없습니다. 즉, 초기화되지 않는 변수이기 때문에 다음 코드와 같이 잘못된 코드입니다:

```
int i;
printf("i = %d\n", i);
```

따라서 주어진 코드의 첫 줄에서 우리는 answer가 어디를 가리키는지 알 수 없습니다. 즉, 바로 위의 코드에서 i가 무슨 값을 가지는지 알 수 없는 것과 같은 이유입니다. (초기화되지 않은 지역 변수[1]는 일반적으로 쓰레기 값을 가지게 됩니다. 이는 널 포인터와는 다릅니다. 질문 1.30, 5.1을 참고하기 바랍니다)

위와 같이, 간단히 질문하고 답변받는 프로그램에서는 포인터 대신 지역 배열을 쓰는 것이 훨씬 간단합니다:

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

char answer[100], *p;
printf("Type something:\n");
fgets(answer, sizeof answer, stdin);
if ((p = strchr(answer, '\n')) != NULL)
  *p = '\0';
printf("You typed \"%s\"\n", answer);
```

이 예제는 gets() 대신 fgets()를 썼습니다. 따라서 배열의 끝이 덮어쓰여질 염려가 없습니다. (질문 12.23을 참고하기 바랍니다. 아쉽게도 이 예에서 쓴 fgets()는 gets()와 달리 마지막 \n을 지우지 못합니다.) answer에 malloc()으로 메모리를 할당하고, 버퍼 크기를 다음과 같이 파라메터로 넘기는 방법도 있습니다.

```
#define ANSWERSIZE    100
```

Q 7.2 strcat()이 동작하지 않습니다. 코드는 다음과 같으며:

```
char *s1 = "Hello, ";
char *s2 = "world!";
char *s3 = strcat(s1, s2);
```

이상한 결과가 발생합니다.

Answer 질문 7.1에서 언급한 것처럼, 주 원인은 연결한 문자열을 저장할만한 공간이 없다는 것입니다. C 언어는 동적으로 문자열을 처리해주지 않습니다. C 컴파일러는 소스에서 요구한 만큼만 메모리를 할당합니다 (문자열의 경우엔 문

[1] uninitialized local variable

자 배열과 문자열 상수를 포함합니다). 프로그래머는 문자열 연결(concatenation)과 같은 실행 시간 연산(run-time operation)에 필요한 적절한 공간을 배열이나 `malloc()`을 불러서 직접 만들어 주어야 합니다.

`strcat()`은 어떠한 공간도 할당해주지 않습니다; 두번째 문자열은 단순히 첫번째 문자열에 붙어서 연결됩니다. 그러므로 첫번째 문자열을 저장하는 곳이 충분한 공간을 가지고 있어야 하며, 쓸 수 있어야(writable) 합니다. 따라서 다음과 같이 배열로 선언하면 쉽습니다:

```
char s1[20] = "Hello, ";
```

물론, 실전에 쓰기 위해서는 위와 같이 20이라는 상수를 쓰는 것은 좋지 않습니다. 우리는 적절한 공간을 보장할 수 있는 어떠한 메커니즘을 써서 공간을 할당해야 합니다.

`strcat()`이 첫번째 문자열을 가리키는 (질문에서는 `s1`) 포인터를 리턴하므로, 변수 `s3`는 불필요합니다; `strcat()`을 부른 다음에 결과는 `s1`이 가리키고 있습니다.

질문에서 `strcat()`을 부른 코드는 크게 두가지 문제를 가지고 있습니다: 먼저 `s1`이 충분한 공간을 가지고 있지 않다는 것과, 이 문자열이 쓸 수 없는 (read-only) 문자열이라는 것입니다. 질문 1.32를 참고하기 바랍니다.

References [CT&P] § 3.2 p. 32

Q 7.3 그러나 매뉴얼(man) 페이지는 `strcat()`이 인자로 두 개의 `char *`를 받는다고 씌여 있습니다. 어떻게 이 것만 가지고 필요한 메모리 공간을 만들어 주어야 한다는 것을 알 수 있죠?

Answer 일반적으로 포인터를 사용할 때에는 항상 메모리 공간을 잡아 주어야 한다고 생각해야 합니다. 만약 라이브러리 함수의 설명에 메모리 할당에 대한 언급이 없다면, 대부분은 호출하는 쪽에서 알아서 해 주어야 합니다.

UNIX 스타일의 매뉴얼(man) 페이지의 첫 부분에 있는 'Synopsis' 섹션, 또는 ANSI C 표준에 나온 이러한 부분은 잘못 이해하기 쉽습니다. 이들 문서에 나온 코드는 호출하는 입장이 아닌 함수를 만드는 입장에 더 가깝기 때문입니다. 특히 포인터를 인자로 받는 (구조체나 문자열) 대부분의 함수들은 포인터가, 어떤 오브젝트를 (구조체나 배열 — 질문 6.3, 6.4 참고) 가리켜야 하는 경우가 많으며, 이 오브젝트는 부르는 입장에서 할당해 주어야 하는 경우가 대부분입니다. 다른 예로는 `time()` 함수와 (질문 13.12 참고) `stat()` 함수를 들 수 있습니다.

Q 7.3b 다음과 같은 코드를 실행했는데:

```
char *p;
strcpy(p, "abc");
```

동작을 합니다. 왜 그러죠? 제 예상대로라면 프로그램이 망가져야(crash) 하는데요.

Answer 추측컨대 아주 재수가 좋은 모양입니다. 초기화되지 않은 포인터 p에 어떤 쓰레기 값이 들어갔고, 그 값이 여러분이 쓸 수 있는 메모리 공간을 가리키고 있었고, 그 공간이 어떤 중요한 목적으로 쓰이고 있지 않았기 때문입니다.

Q 7.3c 포인터 변수는 얼마나 큰 메모리를 할당할까요?

Answer 아주 잘못된 질문입니다. 포인터 변수를 다음과 같이 선언했다고 할 때:

```
char *p;
```

여러분은 (좀더 정확히 말해서, 컴파일러는) 포인터 자체만 저장할 수 있는 공간을 할당한 것입니다; 즉, 이 경우 `sizeof(char *)` 바이트만큼의 메모리가 할당된 것입니다. 그러나 이 포인터는 아직 어떠한 메모리도 가리키고 있지 않습니다. 질문 7.1과 7.2를 참고하기 바랍니다.

Q 7.4 다음과 같이 파일에서 줄 단위로 읽는 코드를 만들었습니다:

```
char linebuf[80];
char *lines[100];
int i;

for (i = 0; i < 100; i++) {
  char *p = fgets(linebuf, 80, fp);
  if (p == NULL) break;
  lines[i] = p;
}
```

그런데 이 코드를 실행하면, 모든 줄에 마지막 줄의 내용이 들어가 있습니다.

Answer 여러분이 선언한 `linebuf`는 단지 한 줄만을 저장할 수 있는 버퍼입니다. `fgets`를 부를 때마다, 기존의 줄은 덮어써져 버립니다. `fgets`는 내부적으로 메모리를 할당해 주지 않습니다. 에러가 났거나 `EOF`를 만나지 않는다면, `fgets`가 리턴하는 값은 여러분이 첫번째 인자로 전해 준 포인터와 같습니다. (이 경우 `linebuf`를 가리키는 포인터)

이런 식으로 코드를 작성하려 한다면, 여러분이 각각의 줄을 저장할 공간을 일일이 할당해 주어야 합니다. 질문 20.2의 코드를 참고하기 바랍니다.

References [K&R1] § 7.8 p. 155
[K&R2] § 7.7 pp. 164–5 [ANSI] § 4.9.7.2
[C89] § 7.9.7.2
[H&S] § 15.7 p. 356

Q 7.5a 문자열을 리턴하는 함수를 만들었는데, 리턴한 문자열이 쓰레기로 채워진 것 같습니다. 왜 그럴까요?

Answer 포인터가 적절한 메모리 공간을 가리키고 있는지 잘 검사해보시기 바랍니다. 함수가 리턴하는 포인터는 정적으로 할당된 버퍼이거나, 이 함수를 부른 위쪽에서 전달해 준 버퍼이거나, 또는 `malloc`으로 할당한 버퍼 등을 가리키고 있어야 합니다. 그러나 절대로 이 함수 안에서 할당한 지역 변수면 (automatic array) 안됩니다. 예를 들어 절대로 다음과 같이 하지 말기 바랍니다:

```
#include <stdio.h>

char *itoa(int n)
{
  char retbuf[20];          /* WRONG */
  sprintf(retbuf, "%d", n);
  return retbuf;            /* WRONG */
}
```

함수가 끝나면, 함수 안에서 만들어진 automatic, local 변수는 없어집니다. 따라서 위 경우는 올바른 코드가 아닙니다 (즉, 리턴되는 포인터가 존재하지 않는 곳을 가리키고 있습니다).

한 가지 방법은 (완전한 것은 아닙니다. 특히 이 함수가 재귀적으로 호출되거나 동시에 이 함수를 여러번 부르고, 그 값을 사용하려 할 때에는 쓸 수 없습니다.) 리턴할 버퍼를 다음과 같이 만드는 것입니다:

```
static char retbuf[20];
```

다른 방법은, 이 함수를 부르는 함수가, 버퍼를 제공하도록 고치는 것입니다:

```
char *itoa(int n, char *retbuf)
{
  sprintf(retbuf, "%d", n);
  return retbuf;
}

...

  char str[20];
  itoa(123, str);
```

또, malloc을 쓰는 방법도 있습니다:

```
#include <stdlib.h>

char *itoa(int n)
{
  char *retbuf = malloc(20);
  if (retbuf != NULL)
    sprintf(retbuf, "%d", n);
  return retbuf;
}

...

  char *str = itoa(123);
```

이 경우, 이 함수를 부르는 쪽에서 더 이상 이 함수가 리턴한 값이 필요없을 경우, 리턴한 포인터가 가리키고 있는 메모리를 free해 주어야 합니다.

덧붙여 질문 7.5b, 12.21, 20.1도 참고하시기 바랍니다.

References [ANSI] § 3.1.2.4
[C89] § 6.1.2.4

Q 7.5b 그럼 문자열이나 기타 이런 것들을 리턴하려면 어떻게 해야 하죠?

Answer 포인터를 리턴할 때에는 이 포인터가 정적으로 할당된 (statically-allcated) 공간을 가리키고 있어야 합니다. 또는 이 버퍼가 이 함수에 전달된 메모리를 가리키고 있거나, 이 버퍼가 `malloc()`으로 할당된 버퍼이어야 합니다. 절대로 지역(local, automatic) 배열을 가리키고 있어서는 안됩니다.

질문 20.1을 참고하기 바랍니다.

7.2 Calling `malloc`

고정된 메모리 블럭을 넘어서, 좀 더 많은 유연성을 필요로 한다면, 메모리를 동적으로 할당하는 방식을 쓸 단계입니다. 대개는 `malloc` 함수를 써서 이 작업을 수행합니다. 이 section에서는 `malloc`에 대한 기본적인 사항에 대하여 다루며, 다음 section에서는 `malloc`이 실패했을 경우에 대한 것을 배웁니다.

Q 7.6 `malloc()`을 호출할 때 왜 "warning: assignment of pointer from integer lacks a cast"라는 경고가 발생할까요?

Answer `<stdlib.h>`를 포함하거나, `malloc`을 쓰기 위해 올바른 선언을 제공했는지 확인하기 바랍니다. 만약 이런 일을 안했다면, 컴파일러는 `int`를 리턴하는 것으로 (질문 1.25 참고) 가정합니다. 물론 이 가정은 잘못된 것입니다 (같은 이유로, 질문한 경고 메시지가 `calloc`이나 `realloc`에서 발생할 수 있습니다. 덧붙여 질문 7.15도 참고하시기 바랍니다.

References [H&S] § 4.7 p. 101

Q 7.7 어떤 코드를 보면 `malloc()`이 리턴한 포인터를 대입할 포인터의 타입으로 캐스팅한 것을 볼 수 있는 데, 왜 그럴까요?

Answer ANSI/ISO C 표준에서 `void *`를 소개되기 전에는, 대개 포인터 변환에 관계된 경고를 없애기 위해, 또는 불필요한 변환을 줄이기 위해 이러한 캐스팅을 사용했습니다.

ANSI/ISO C 표준에서는 이러한 캐스팅이 전혀 필요없습니다. 그리고 현재 이런 캐스팅을 사용하는 것은 나쁜 프로그래밍 스타일로 간주되기도 합니다. 왜냐하면 `malloc()`이 선언되지 않았을 때 발생할 수 있는 유용한 경고 메시지를 발생시키지 않기 때문입니다; 질문 7.6을 참고하기 바랍니다. (그러나 이런 캐스팅은 여전히 자주 쓰이고 있습니다. 왜냐하면 C++에서는 이러한 캐스팅이 반드시 필요하기 때문에, 호환성을 유지하기 위해서입니다.)

References [H&S] § 16.1 pp. 386–7

Q 7.8 다음과 같은 코드를 본 적이 있습니다:

```
char *p = malloc(strlen(s) + 1);
strcpy(p, s);
```

제 생각에는 `malloc((strlen(s) + 1) * sizeof(char))`로 되어야 할 것 같은데요.

Answer `sizeof(char)`로 곱하는 것은 전혀 필요 없습니다. 왜냐하면 정의에 의해, `sizeof(char)`는 항상 1이기 때문입니다. 다른 말로 하면 `sizeof(char)`를 곱하는 것은 전혀 문제가 되지 않습니다. 1을 곱하는 것은 아무런 영향을 끼치지 않기 때문입니다. 추가적으로 `size_t` 타입을 쓰는 것이 도움이 될 때도 있습니다. (질문 7.15 참고) 덧붙여 질문 8.9, 8.10도 참고하시기 바랍니다.

References [ANSI] § 3.3.3.4
[C89] § 6.3.3.4
[H&S] § 7.5.2 p. 195

Q 7.9 다음과 같이 `malloc()`의 wrapper를 만들었습니다. 그런데 왜 동작하지 않을까요?

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

mymalloc(void *retp, size_t size)
{
  retp = malloc(size);
  if (retp == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(EXIT_FAILURE);
  }
}
```

Answer 질문 4.8을 참고하기 바랍니다. (이 경우, 여러분은 `mymalloc`이 할당된 공간을 가리키는 포인터를 돌려주도록 해야 합니다.)

Q 7.10 포인터를 선언하고 여기에 공간을 할당하려고 합니다. 그런데 제대로 동작하지 않는군요. 다음 코드에서 뭐가 잘못되었나요?

```
char *p;
*p = malloc(10);
```

Answer 질문 4.2를 보기 바랍니다.

Q 7.11 배열을 동적으로 할당할 수 있을까요?

Answer 질문 6.14와 6.16을 보기 바랍니다.

Q 7.12 쓸 수 있는 메모리 공간이 얼마나 남아 있는지 알 수 있을까요?

Answer 질문 19.22를 보기 바랍니다.

Q 7.13 `malloc(0)`은 어떤 값을 리턴하는 거죠? 널 포인터를 리턴하나요, 0 바이트를 가리키는 포인터를 리턴하나요?

Answer 질문 11.26을 보기 바랍니다.

Q 7.14 어떤 시스템에서는 `malloc()`으로 메모리를 할당해도 프로그램에서 이 메모리에 접근하기 전에는 운영체제가 메모리를 할당하지 않는다고 들었습니다. 그래도 상관없을까요?

Answer 꽤 말하기 어려운 문제입니다. 표준에서는 이런 식으로 동작하는 시스템이 있다고 말한 바가 없습니다. 하지만, 반대로 그러한 시스템이 없다고말한 바도 없습니다. (이와 같은 "deferred failure" implementation은 표준이 요구하는 것과 상관없어 보입니다.)

실제 메모리에 쓰려고(write) 할 때, 남아 있는 메모리가 없는 경우가 문제가 되는데, C 언어 semantic에는 recourse가 없으므로, 이 경우, 대개 OS가 해당 프로그램을 죽이게(kill) 됩니다. (당연히, `malloc`은 메모리가 없을 경우, 널 포인터를 리턴하게 되므로, `malloc`의 리턴 값을 확인했다면, 할당한 메모리가 아닌 다른 메모리에 접근할 까닭이 없습니다.)

이와 같이 "lazy allocation" 방식을 쓰는 시스템에서는 대개 메모리가 부족하다는 것을 알려주는 시그널을 제공합니다. 그러나 프로그램에서 이 시그널을

처리하려면 이식성이 떨어지는 프로그램이 될 가능성이 높습니다. 이 방식을 제공하는 시스템 중에는 사용자 또는 프로세스 단위로 이 기능을 끄는(off) 기능을 제공하는 (즉, 전형적인 `malloc` semantic을 쓰는) 것도 있습니다만, 시스템마다 매우 달라집니다.

References [ANSI] § 4.10.3
[C89] § 7.10.3

7.3 Problems with `malloc`

Q 7.15 왜 `malloc`이 이상한 값을 리턴할까요? 질문 7.6을 읽고나서 `malloc`을 부르기 전에 `extern void *malloc();` 선언을 포함시켰는데도 이상한 쓰레기 값을 줍니다.

Answer `malloc`의 인자 타입은 `size_t`이며, 아마 `unsigned long` 타입일 가능성이 높습니다. 만약에 `int` 타입(또는 `unsigned int`)을 전달했다면, `malloc`이 (깨진) 쓰레기 값을 받았을 가능성이 있습니다. (또는 `long` 타입을 전달했는데 `size_t`가 `int`일 경우에도 이럴 가능성이 있습니다.)

일반적으로, 표준 라이브러리 함수의 선언을 `extern`으로 직접 써 주는 것보다 알맞는 헤더 파일을 포함시키는 것이 훨씬 더 안전합니다. 덧붙여 질문 7.16도 참고하시기 바랍니다.

비슷하면서 잘 알려진 또 다른 문제는, `size_t` 타입의 값을 (`sizeof`의 값 포함) `printf`의 `%d`를 써서 출력하는 것입니다. 이식성이 높은 코드를 얻기 위해서, 주어진 값을 직접 (`unsigned long`으로) 캐스팅한 다음 `%lu` 포맷으로 출력하는 것이 좋습니다.

```
printf("%lu\n", (unsigned long)sizeof(int));
```

덧붙여 질문 15.3도 참고하시기 바랍니다..

References [ANSI] § 4.1.5, § 4.1.6
[C89] § 7.1.6, § 7.1.7

Note C99 표준은 `size_t`를 쉽게 출력하기 위해, `printf`, `scanf` 함수들에 `z` modifier를 추가했습니다. 질문 13.1을 참고하기 바랍니다.

Q 7.16 어떤 수학 계산을 하기 위해 꽤 큰 배열이 필요해서 다음과 같이 코드를 작성했습니다:

```
double *array = malloc(256 * 256 * sizeof(double));
```

이 때 `malloc()`은 널 포인터를 리턴하지 않았지만 프로그램이 매우 이상하게 동작합니다. 제 생각에는 메모리를 겹쳐 쓰는(overwrite) 현상이 발생한 것 같거나 `malloc()`이 원하는 만큼의 큰 메모리를 할당한 것 같지가 않습니다.

Answer 256 곱하기 256은 65,536이며, `int`가 16-bit인 경우, 저장될 수 없는, 큰 값입니다. 게다가 이 수치는 다시 `sizeof(double)`로 곱해야 하기 때문에 상당히 큰 크기입니다. 이런 큰 배열이 꼭 필요하다면 꽤 주의를 기울여야 합니다. 만약 여러분의 시스템에서 `size_t` 타입이 (`malloc()`이 인자로 받는 타입) 32 비트라면, 그리고 `int`가 16 비트라면, 위 코드에서 `malloc`의 인자로 `256 * (256 * sizeof(double))`를 써서 해결할 가능성도 있습니다. (질문 3.14 참고). 그렇지 않다면 여러분은 이 메모리를 작은 크기로 쪼개어 쓰거나, 또는 32 비트 컴퓨터나 컴파일러를 쓰거나, 비표준으로 제공되는 메모리 할당 함수를 써야 할 것입니다. 덧붙여 질문 7.15, 19.23도 참고하시기 바랍니다.

Note 이 질문은 16-bit 시스템 및 컴파일러가 대부분일 시절에 만들어진 것이기 때문에, 현실과 거리가 멀 수도 있지만, 비슷한 이유로 32-bit 시스템에서도 문제가 발생할 수 있으므로 그냥 두겠습니다.

Q 7.17 제 PC는 8 메가 바이트의 메모리를 보유하고 있습니다. 그런데 왜 640K의 크기 밖에 쓸 수 없는 것일까요?

Answer PC 호환의 세그먼트(segment) 구조를 사용하는 컴퓨터에서 640K 이상의 메모리를 사용하는 것은 (특히 MS-DOS에서) 매우 어렵습니다. 질문 19.23을 참고하기 바랍니다.

Q 7.18 제 프로그램은 어떤 노드(어떤 자료 구조)를 동적으로 할당하는 작업을 매우 많이 수행합니다. 테스트 결과 `malloc/free`에서 발생하는 오버헤드가 bottleneck이 되고 있는데, 좋은 방법이 있을까요?

Answer 만약 주어진 노드의 크기가 모두 같다면, `free`를 부르는 대신에 개발자가 직접 쓰이지 않는 노드로 이루어진 리스트를 관리하는 것이 좋을 수 있습니다. (이 방식은, 프로그램에서 가장 많은 메모리를 소비하는 것이 바로 이 노드일 때 좋은 해결책이 됩니다. 그렇지 않다면 이 노드 리스트에 묶여 있는 메모리가 많아서 다른 곳에서 메모리를 쓸 수 없게 되어 버립니다.)

Note 사용자가 직접 죽은(dead) 노드들을 리스트에 저장해 두고, 필요할 때, `malloc`을 부르는 대신 이 죽은 노드를 활용하는 것을, 'object caching'이라고 부르기도 합니다.

예를 들어, 리스트의 노드가 다음과 같을 경우:

```
struct node {
  struct node *next;
  ...
};
```

그리고, 새 노드를 만드는 함수와 필요없는 노드를 삭제하는 함수가 다음과 같다고 가정합시다:

```
struct node *new_node(void)
{
  struct node *p;
  p = malloc(sizeof(*p));
  /* check if malloc() returns a null pointer */
  ...
  return p;
}

void delete_node(struct node *p)
{
   ...
   free(p);
}
```

이 경우, 다음과 같은 전역 변수를 두고,

```
static struct node *grave = 0;
```

위에서 만든 두 함수를 "object caching" 기법을 쓰도록 바꿀 수 있습니다:

```
struct node *new_node(void)
{
  struct node *p;
  if (grave) {
    p = grave;
```

```
    grave = p->next;
  }
  else {
    p = malloc(sizeof(*p));
    /* check if malloc() returns a null pointer */
  }
  ...
  return p;
}

void delete_node(struct node *p)
{
   ...
   p->next = grave;
   grave = p;
}
```

물론 수정된 두 함수가 완벽하지는 않습니다. `delete_node()`는 `free`를 부르지 않고, 필요없는 모든 `node`를 `grave` 포인터에 리스트로 보관하고, 필요한 경우, (즉 `new_node()`를 불렀을 경우) 리스트에 보관된 노드를 다시 씁니다. 최적화를 위해서, 일정한 갯수만 `grave` 리스트에 보관하고, 나머지는 `free`를 불러서 해제하는 방식으로도 만들 수 있습니다.

이런 "object caching"이 모든 문제를 해결해 주지는 않습니다. 상황에 따라, 이 방식이 도움이 될 경우가 있으므로, 여기에서 소개한 것이지, 항상 좋다는 것은 아니라는 것을 기억하시기 바랍니다.

Q 7.19 제 프로그램은 `malloc()` 내부에서 오류가 발생한 것 같습니다. 그런데 제가 보기에는 제 프로그램에 잘못된 부분이 없는 것 같습니다. `malloc()` 내부에 버그가 있는 게 아닐까요?

Answer 불행히도, 프로그래머가 `malloc()`이 내부적으로 유지하는 데이터 구조를 망가뜨리는 경우가 자주 발생합니다. 또 그 결과 생기는 문제는 매우 다루기가 어렵습니다. 가장 흔히 발생하는 실수는, `malloc()`으로 할당한 메모리보다 더 많은 데이터를 쓰려고(write) 하는 경우입니다; 특히 문자열을 저장하기 위해 `strlen(s) + 1`을 쓰지 않고 `malloc(strlen(s))`를 쓰는 경우가 흔합니다.[2] 다른 문제로 이미 `free()`를 불러서 반환한 메모리 블럭에 쓰려고

[2]또 자주 발생하는 다른 버그로, `malloc(strlen(s + 1))`로 쓰는 경우와 `p = malloc(sizeof(p))`를 쓰는 경우가 있습니다.

(write) 하거나 (질문 7.30 참고), `free()`를 같은 포인터에 대해 두 번 호출하는 경우도 있습니다. (이 중 몇가지는 표준에 의해 인정되고 있습니다. ANSI 호환 시스템에서 크기가 0인 메모리 블럭을 `malloc`, `realloc` 등으로 할당하거나, 널 포인터를 `free()`로 반환하는 것은 아무런 문제도 일으키지 않습니다. 물론 오래된 시스템에서는 문제가 됩니다.) 대개 이러한 실수는 금방 발견되지 않고, 오랫동안 숨어 있거나, 또는 전혀 상관없는 코드 부분에서 에러가 발생한 것처럼 보이기도 해서 수정하기가 매우 까다롭습니다.

대부분 `malloc` implementation은 중요한 정보를 할당한 메모리 바로 옆에 보관하기 때문에, 사용자가 포인터를 조금이라도 잘못 쓰게 되면, 심각한 문제가 발생합니다.

덧붙여 질문 7.15, 7.26, 16.8, 18.2도 참고하시기 바랍니다.

7.4 Freeing Memory

Memory allocated with `malloc` can persist as long as you need it. It is never deallocated automatically (except when your program exits; see question 7.24). When your program uses memory on a transient basis, it can—and should—recycle it by calling `free`.

Q 7.20 동적으로 할당한 메모리를 해제한 다음에는 다시 쓸 수 있나요?

Answer 쓸 수 없습니다. 어떤 오래된 `malloc()`에 대한 문서는 해제된 메모리는 변경되지 않는 상태로 남아 있다고 (left undisturbed) 씌여 있지만 이는 잘못된 것이며, C 표준에도 언급되지 않은 사항입니다.

대부분의 프로그래머들이 반환된 메모리의 내용을 일부러 다시 쓰지는 않습니다만, 실수로 반환한 메모리 공간을 쓰기도 합니다. 다음에 나온 singly-linked 리스트를 반환하는 코드를 보시기 바랍니다:

```
struct list *listp, *nextp;
for (listp = base; listp != NULL; listp = nextp) {
  nextp = listp->next;
  free(listp);
}
```

만약 위 코드가 임시 변수 `nextp`를 쓰지 않고, `listp = listp->next`를 썼다면, 이미 반환한 메모리를 `listp->next`로 다시 접근하려 한다는 것을 알 수 있습니다.

References [K&R2] § 7.8.5 p. 167
[ANSI] § 4.10.3
[C89] § 7.10.3
[ANSI Rationale] § 4.10.3.2
[H&S] § 16.2 p. 387
[CT&P] § 7.10 p. 95

Q 7.21 왜 `free()`를 부른 다음에 포인터가 널이 되지 않는 걸까요?

Answer `free()`를 부르면 이 함수에 전달된 포인터가 가리키고 있던 메모리가 해제됩니다. 그러나 이 포인터 자체의 값은 변경되지 않고 남아있습니다. 왜냐하면 C 언어는 인자를 전달할 때, 'pass-by-value' 개념을 쓰기 때문입니다. 따라서 함수가 (이 경우 `free()`) 인자로 전달된 변수의 값을 변경할 수 없습니다. (질문 4.8을 참고하기 바랍니다.)

일단 해제된 포인터 값은 엄밀히 말해서, 유효하지 않습니다(invalid). 그리고 (dereference가 아니더라도) 어떠한 목적으로 (심지어, 단순히 대입하거나 비교하는 것도) 이 값을 쓰는 것은, 물론 구현 방법에 따라 다르긴 하지만, 이론상 문제를 발생할 수 있습니다. (물론, 대부분의 시스템이 문제가 없어 보이는 invalid 포인터 쓰임새를 너그럽게 봐 주지만, 표준은 확실하게, 어떤 것도 보장될 수 없다고 말합니다. 또한 어떤 시스템은 구조상 이런 exception 상황이 흔히 발생합니다.)

References [ANSI] § 4.10.3
[C89] § 7.10.3
[ANSI Rationale] § 3.2.2.3

Q 7.22 함수에 속한 지역(local) 포인터에 `malloc()`으로 메모리를 할당했을 때에도 반드시 `free()`를 해주어야 하나요?

Answer 당연합니다. 포인터와 포인터가 가리키는 메모리는 서로 다른 것이라는 것을 기억해야 합니다. 함수가 끝났을 때, 지역 변수로 선언한 포인터는 자동으로 해제되지만, 포인터 자체가 해제된다는 뜻이지, 포인터가 가리키는 메모리 블럭이 해제된다는 것은 아닙니다. `malloc()`으로 할당한 메모리는 `free()`를 써서 해제하기 전에는 메모리에 남아 있습니다. 일반적으로 모든 `malloc()`에는 각각의 호출에 해당하는 `free()`를 만들어 주어야 합니다.

Q 7.23 동적으로 할당한 어떤 오브젝트를 가리키는 포인터를 포함하는 구조체를 할당했습니다. 이 구조체를 해제할 때, 각각의 포인터 멤버가 가리키는 메모리도 따로 해제해 주어야 하나요?

Answer 그렇습니다. `malloc()`으로 할당한 메모리는 각각, 정확히 딱 한번, `free()` 해주어야 합니다. 프로그램에서 각각 `malloc()`으로 할당한 메모리는 모두 `free()`시키는 것이 좋은 습관입니다.

질문 7.24를 참고하기 바랍니다.

Q 7.24 프로그램이 끝나기 전에 동적으로 할당한 메모리는 반드시 해제시켜야 하나요?

Answer 그럴 필요는 없습니다. 일반적으로 운영체제는 프로그램이 끝났을 때, 프로그램이 할당한 모든 메모리를 해제시켜 줍니다. 그럼에도 불구하고, 어떤 PC에서는 메모리를 제대로 복원시키지 못한다고 알려져 있습니다. ANSI/[C89] C 표준에서는 이런한 상황을 '구현 수준에 따른 상황(quality of implementation issue)'이라고 말하고 있습니다.

References [C89] § 7.10.3.2

Q 7.25 제 프로그램은 메모리 블럭을 `malloc()`으로 할당하고, 나중에 `free()`시키는 구조를 가집니다. 그런데, 프로그램이 끝난 다음에도, 운영 체제의 메모리 상태를 살펴보면 전혀 줄어들지 않는 것을 볼 수 있습니다.

Answer 대부분의 `malloc/free`의 구현 방법들은 메모리를 운영체제에 즉시 반환하는 것이 아니며, 단지 그 프로그램에서 나중에 나올 `malloc()`이 그 메모리를 다시 쓸 수 있도록 해 주도록 되어 있습니다.

7.5 Sizes of Allocated Blocks

Q 7.26 `free()` 함수는 어느 정도 크기의 메모리를 해제할 지 어떻게 알 수 있는 것일까요?

Answer `malloc/free` 구현은 동적으로 할당한 각각의 메모리 블럭의 크기를 내부적으로 유지하고 있습니다. 따라서 `free()`를 쓸 때에 그 크기를 지정하지 않아도 자동으로 그 크기를 알 수 있습니다.

Q 7.27 그렇다면, 할당한 블럭의 크기가 어느 정도인지 `malloc` 패키지를 써서 알 수 있을까요?

Answer 불행하게도, 동적 블럭의 크기를 알 수 있는 표준 또는 호환성있는 방법은 제공되지 않습니다.

Q 7.28 어떤 포인터가 메모리를 가리키고 있을 때, 왜 `sizeof`는 그 메모리 블럭의 크기를 알려주지 않나요?

Answer `sizeof` 연산자는 포인터가 가리키는 값이 `malloc`으로 할당한 메모리를 가리키고 있는지 알지 못합니다. `sizeof`가 알려주는 것은 포인터 자체가 차지하고 있는 크기입니다. `malloc`을 불러서 얻은 메모리 블럭의 크기를 알아내는 (이식성이 높은) 방법은 없습니다.

7.6 Other Allocation Functions

Q 7.29 (질문 6.14를 따라서) 배열을 동적으로 할당한 다음, 이 배열의 크기를 바꿀 수 있을까요?

Answer 물론입니다. `realloc`이 바로 그 역할을 해 주는 함수입니다. (예를 들어, 질문 6.14에 나온 것처럼 `dynarray`) 동적으로 할당된 배열의 크기를 바꾸려면 다음과 같은 코드를 씁니다:

```
dynarray = (int *)realloc((void *)dynarray,
                          20 * sizeof(int));
```

`realloc`이 항상 메모리 블럭의 크기를 늘리는데[3] 쓰이는 것은 아닙니다. `realloc`은, 가능하면, 전달받은 인자와 같은 포인터 값을 돌려주지만, 요청한 크기에 맞게 메모리 블럭을 다시 찾을 경우에는, 인자로 전달된 포인터 값과는 다른 값을 돌려줍니다. 이 경우, 인자로 전달된 포인터 값은 더 이상 쓸 수 없습니다.

만약 `realloc`이 요청한 메모리 공간을 찾지 못했다면, 널 포인터를 리턴합니다. 이 때, 인자로 전달되었던 메모리는 (`realloc`을 부르기 바로 전 상태로) 그대로 유지됩니다.[4]

[3] 또는 항상 크기를 줄이는데에만 쓰이는 것도 아닙니다.

[4] 그러나 어떤, ANSI 이전의 컴파일러에서는 `realloc`이 실패했을 경우, 인자로 전달되었던 메모리 블럭의 보존 여부를 보장하지 않습니다.

realloc을 써서 메모리의 크기를 변경했을 경우에, 다른 포인터가 이 메모리 공간을 가리키고 있었는지 ("alias"라는 용어를 씁니다) 주의해야 합니다: 만약에 realloc이 다른 곳에 메모리 블럭을 할당했다면, 다른 포인터들도 값이 올바르게 바뀌어야 합니다. (malloc의 실패 여부를 확인하지 않았다는 단점이 있긴 하지만) 누군가가 제공한 코드입니다:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *p, *p2, *newp;
int tmpoffset;

p = malloc(10);
strcpy(p, "Hello,");    /* p is a string */
p2 = strchr(p, ',');    /* p2 points into that string */

tmpoffset = p2 - p;
newp = realloc(p, 20);
if (newp != NULL) {
  p = newp;                   /* p may have moved */
  p2 = p + tmpoffset;         /* relocate p2 as well */
  strcpy(p2, ", world");
}

printf("%s\n", p);
```

위와 같이 기본값(base)을 기초로 해서 포인터 값을 다시 계산하는 것이 바람직합니다. 다른 방법은—두 값의 차(difference)인 newp - p의 값을 기초로, realloc을 부르기 전, 후의 베이스 포인터의 값을 쓰는 방식– 동작한다고 보장할 수 없습니다. 왜냐하면, 포인터 뺄셈은 같은 오브젝트를 가리키는 포인터 사이에서만 의미가 있기 때문입니다. 덧붙여 질문 7.12, 7.30도 참고하시기 바랍니다..

References [K&R2] § B5 p. 252
[ANSI] § 4.10.3.4
[C89] § 7.10.3.4
[H&S] § 16.3 pp. 387–8

Q 7.30 `realloc` 함수의 첫번째 인자로 널 포인터를 전달하는 것이 안전한가요? 그리고 왜 그런 일을 하는 것이죠?

Answer ANSI C 표준은 그런 식으로 써도 (그리고 `realloc(..., 0)`처럼 써서 메모리를 해제하는 것) 괜찮다고 말합니다. 그러나 대부분의 오래된 구현 방법에서는 이러한 사용법을 제공하지 않습니다. 따라서 표준이기는 하지만 완전히 호환성을 갖춘 방법이 아닙니다. `realloc`의 첫 인자로 널 포인터를 전달하면 메모리를 증가적으로 동적으로 할당하는 알고리즘을 (self-starting incremental allocation algorithm) 만들기 쉽습니다.

References [C89] § 7.10.3.4
[H&S] § 16.3 p. 388

Q 7.31 `calloc()`과 `malloc()`의 차이는 무엇인가요? `calloc()`이 메모리를 0으로 만드는 것을 믿고 쓸 수 있나요? 그리고, `calloc()`으로 할당한 메모리를 `free()`를 써서 해제할 수 있나요? 아니면 `cfree()`와 같은 것을 사용하나요?

Answer `calloc()`은 다음의 코드를 수행하는 것과 같습니다:

```
p = malloc(m * n);
memset(p, 0, m * n);
```

0으로 채운다는 것은 할당한 메모리의 모든 비트를 0으로 채운다는 뜻입니다. 따라서 이 값이 널 포인터가 아닐 수도 있으며(여기에 관한 것은 5 절을 참고하기 바랍니다.) 실수(floating-point)로 0이 아닐 수도 있습니다. 그리고 `calloc()`으로 할당한 메모리를 해제할 때에도 `free()`를 씁니다.

References [C89] § 7.10.3부터 7.10.3.2
[H&S] § 16.1 p. 386, § 16.2 p. 386
[PCS] § 11 pp. 141–142

Q 7.32 `alloca()` 함수는 어떤 함수이며, 왜 쓰지 말라고 하죠?

Answer `alloca()`는 이 함수를 호출한 함수가 끝날 때, 자동으로 할당한 메모리를 해제해 주는 함수입니다. 즉, `alloca`로 할당한 메모리는 특정 함수의 "스택 프레임(stack frame)"이나 문맥(context)에 종속적입니다.

`alloca()` 함수는 이식성있게 만들 수가 없습니다. 그리고 일반적인 스택을 사용하지 않는 컴퓨터에서는 매우 만들기 어려운 함수입니다. 특히 `alloca`

로 할당한 메모리를 리턴하는 경우, 심각한 문제가 발생할 수 있습니다. 예를 들면:

```
fgets(alloca(100), 100, stdin);
```

이런 이유에서 `alloca` 함수는 표준 함수가 아니며, 높은 이식성이 요구되는 프로그램에서는 (매우 쓸모 있기는 하지만) 쓸 수 없습니다.

질문 7.22를 참고하기 바랍니다.

References [ANSI Rationale] § 4.10.3

Chapter 8

Characters and Strings

C has no built-in string type; by convention, a string is represented as an array of characters terminated with '\0'. Furthermore, C hardly has a character type; a character is represented by its integer value in the machine's character set. Because these representations are laid bare and are visible to C programs, programs have a tremendous amount of control over how characters and strings are manipulated. The downside is that to some extent, programs *have* to exert this control: The programmer must remember whether a small integer is being interpreted as a numeric value or as a character (see question 8.6) and must remember to maintain arrays (and allocated blocks of memory) containing strings correctly.

See also question 13.1 through 13.7, which cover library functions for string handling.

Q 8.1 왜 다음 코드는 실행이 안되는 것일까요?

```
strcat(string, '!');
```

Answer 문자열과 문자는 큰 차이가 있습니다. `strcat()` 함수는 문자열을 이어주는 함수입니다.

`'!'`와 같은 문자 상수(character constant)는 한 문자를 나타냅니다. 문자열(string literal)은 쌍따옴표로 둘러싼, 대개 여러 글자의 문자로 구성됩니다. `"!"`와 같은 문자열은 하나의 문자를 나타내는 것처럼 보이지만, 실제로 ! 문자와, 문자열 끝을 나타내는 `\0`, 두 개의 문자로 구성됩니다.

C 언어에서 문자는 문자 집합(character set)에서 그 문자가 나타내는 작은

정수 값으로 표현됩니다 (질문 8.6을 참고하기 바랍니다). 문자열은 문자들의 배열로서 표현됩니다; 문자열을 다룰 때에는 보통 문자 배열의 첫 요소를 가리키는 포인터를 써서 합니다. It is never correct to use one when the other is expected. 문자열에 `!`를 이어 붙이려면 다음과 같이 해야 합니다.

```
strcat(string, "!");
```

질문 1.32, 7.2, 16.6을 참고하기 바랍니다.

References [CT&P] § 1.5 pp. 9–10

Q 8.2 문자열이 어떤 특정한 값과 같은지 검사하려고 합니다. 다음과 같이 코드를 만들었는데 왜 동작하지 않을까요?

```
char *string;
...
if (string == "value") {
  /* string matches "value" */
  ...
}
```

Answer C 언어는 문자열을 문자의 배열로 처리합니다. 그리고 C 언어에서는 배열 전체에 대해 어떤 연산을 (대입, 비교 등) 직접 할 수 있는 방법은 없습니다. 위의 코드에서 `==` 연산은 피연산자인 포인터의 값을 비교합니다 — 즉 변수 `string`의 포인터 값과 문자열 `"value"`를 가리키는 포인터 값을 비교합니다 — 따라서 두 개의 포인터가 같은 곳을 가리키는지를 비교합니다. 대개의 경우 이 값이 같게 될 경우는 거의 없으므로, 이 비교는 거의 항상 같지 않다고 나옵니다.

두 문자열을 비교하는 방법으로 라이브러리 함수인 `strcmp()`를 쓰는 것이 가장 좋습니다:

```
if (strcmp(string, "value") == 0) {
  /* string matches "value" */
  ...
}
```

Q 8.3 다음과 같이 할 수 있다면:

```
char a[] = "Hello, world!";
```

왜 이렇게는 할 수 없을까요?

```
char a[14];
a = "Hello, world!";
```

Answer 문자열은 배열입니다. 그리고 배열에는 직접 대입 연산을 쓸 수 없습니다. `strcpy()` 함수를 쓰기 바랍니다:

```
strcpy(a, "Hello, world!");
```

질문 1.32, 4.2, 6.5, 7.2를 참고하기 바랍니다.

Q 8.4 왜 `strcat`이 동작하지 않을까요? 아래 코드처럼 했는데, 잘 안됩니다:

```
char *s1 = "Hello, ";
char *s2 = "world!";
char *s3 = strcat(s1, s2);
```

Answer 질문 7.2를 보기 바랍니다.

Q 8.5 아래 두 초기화 문장의 차이가 있나요?

```
char a[] = "string literal";
char *p = "string literal";
```

제 프로그램에서 `p[i]`에 어떤 값을 대입하려 하면 프로그램이 비정상적으로 끝나버립니다.

Answer 질문 1.32를 보기 바랍니다.

Q 8.6 문자에 해당하는 수치 (ASCII 또는 다른 문자 set code)) 값을 어떻게 얻을 수 있죠? 또 그 반대로 수치 값에서 그 값에 해당하는 문자를 어떻게 얻을 수 있죠?

Answer C 언어에서 문자는 (컴퓨터의 문자 셋의) 문자 코드 번호로 작은 정수로서 표현됩니다. 따라서 질문한 것과 같은 변환이 필요없습니다; 즉 문자를 가지고 있다면, 그 자체가 값이 됩니다. 예를 들어, 다음 코드는:

```
int c1 = 'A', c2 = 65;
printf("%c %d %c %d\n", c1, c1, c2, c2);
```

ASCII를 쓰는 시스템에서, 다음과 같은 출력을 만들어 냅니다:

```
A 65 A 65
```

덧붙여 질문 8.9, 20.10도 참고하시기 바랍니다.

Q 8.7 C 언어에서, 다른 언어에서 제공하는 것처럼, 문자열의 일부를 뽑아내는, "substr"와 같은 기능이 있나요?

Answer 질문 13.3를 보기 바랍니다.

Q 8.8 사용자가 입력한 문자열을 읽어서 배열에 저장한 다음, 나중에 출력하려고 합니다. 사용자가 `\n`와 같은 문자를 입력한 경우, 왜 제대로 처리되지 않을까요?

Answer `\n`와 같은 문자 시퀀스(character sequence)들은 컴파일할 때 해석됩니다. 문자 상수나 문자열에 백슬래시가 나오고 바로 뒤에 `n`이 나오면, 한 글자, newline 문자로 해석됩니다. (다른 character escape sequence도 비슷한 방법으로 처리됩니다.) 사용자나 파일에서 문자열을 읽을 때는 이와 같은 해석이 적용되지 않습니다: 즉, 백슬래시는 다른 문자와 전혀 다를게 없이 취급되며, 하나의 문자로 간주됩니다. (run-time I/O가 일어날 때, newline 문자를 위해 어떤 번역 작업이 이뤄질 수 있지만, 이 것은 전혀 다른 문제입니다. 질문 12.40을 보기 바랍니다.) 덧붙여 질문 12.6도 참고하시기 바랍니다.

Q 8.9 제 컴파일러에 버그가 있습니다. `sizeof('a')`의 값이 `sizeof(char)`인 1로 나오지 않고, 2가 나옵니다.

Answer 놀랍게도, C 언어에서 문자 상수(character constant)의 타입은 `int`입니다. 따라서 `sizeof('a')`는 `sizeof(int)`와 같습니다. (C++에서는 조금 다릅니다.) 덧붙여 질문 7.8도 참고하시기 바랍니다.

Note 참고로 C++에서 문자 상수의 타입은 `char`입니다. 즉, `sizeof('a')`는 `sizeof(char)`와 같습니다.

References [ANSI] § 3.1.3.4
[C89] § 6.1.3.4
[H&S] § 2.7.3 p. 29

Q 8.10 I'm starting to think about multinational character sets. Should I worry about the implication of making `sizeof(char)` be 2 so that 16-bit character sets can be represented?

Answer 만약 `char`가 16 bit가 된다고 해도, `sizeof(char)`는 여전히 1이 됩니다. 대신 `<limits.h>`에 정의된, `CHAR_BIT`이 16이 됩니다. 이 경우에는 한, 8-bit 오브젝트를 선언하는 것은 (또, `malloc`으로 할당하는 것도) 불가능합니다.

전통적으로, 한 바이트가 꼭 8 bit일 필요는 없습니다. 단지, 한 글자를 저장하기에 충분한, 작은 크기의 메모리 공간이면 됩니다. C 표준에서도 이 방식을 따르며, 따라서 `malloc`이나 `sizeof`에서 쓰이는 바이트가 8 bit 이상일 수도 있습니다.[1] (대신 표준에서, 바이트가 8 bit 이상되어야 한다고 정해 놓았습니다.)

Multinational character set을 처리하기 위해서는 `char` 이상의 어떤 타입이 필요하고, ANSI/ISO C 표준에서는 이를 위해, 더 큰 범위를 포함할 수 있다는 뜻의 "wide" 문자 타입인, `wchar_t`를 제공합니다. 또한 이 타입으로 처리하는 wide 문자 상수, wide 문자열, 또 wide 문자 및 문자열을 처리할 수 있는 함수들을 제공합니다.

덧붙여 질문 7.8도 참고하시기 바랍니다.

References [ANSI] § 2.2.1.2, § 3.1.3.4, § 3.1.4, § 4.1.5, § 4.10.7, § 4.10.8
[C89] § 5.2.1.2, § 6.1.3.4, § 6.1.4, § 7.1.6, § 7.10.7, § 7.10.8
[ANSI Rationale] § 2.2.1.2
[H&S] § 2.7.3 pp. 29–30, § 2.7.4 p. 33, § 11.1 p. 293, §§ 11.7, 11.8 pp. 303–10

[1] 전문 용어로, 8 bit로 표현되는 바이트를 "octet"이라고 합니다.

Chapter 9

Boolean Expressions and Variables

C provides no formal, built-in Boolean type. Boolean values are just integers (though with greatly reduced range!), so they can be held in any integral type. C interprets a zero value as "false" and *any* nonzero value as "true." The relational and logical operators, such as `==`, `!=`, `<`, `>=`, `&&`, and `||`, return the value 1 for "true," so 1 is slightly more distinguished as a truth value than the other nonzero values (but see question 9.2).

C 언어는 C99 표준 다음부터 이제 boolean 타입을 지원하지만, 문제는 boolean 타입의 필요성에 따라, 표준으로 제정되기 이전부터 개발자들이 임의로 boolean 타입을 만들어 써 왔다는 것입니다. 따라서, 기존 프로그램들과 호환성을 위해서, boolean 타입의 이름은 (이상하지만) `_Bool`입니다. 좀 더 자세한 것은 질문 9.1을 보기 바랍니다.

Q 9.1 C 언어에서 불리언(boolean) 값으로 쓸 수 있는 괜찮은 타입이 있을까요? 그리고 왜 그런 타입이 표준으로 지정되어 있지 않는 거죠? 참(true)과 거짓(false)을 나타내기 위해 `#define`으로 정의를 하는 게 좋을까요, 아니면 `enum`을 쓰는 것이 좋을까요?

Answer C 언어는 표준으로 제공하는 불리언(boolean) 타입이 없습니다. 왜냐하면, 불리언 타입은 항상 공간/처리시간의 'tradeoff' 문제를 가지기 때문입니다. 따라서 이는 프로그래머가 알아서 결정하도록 해 놓은 것입니다. (`int`를 쓰면 처리가 빠를 수 있고, `char`를 쓰면 공간을 절약할 가능성이 있습니다.[1] 그리

[1] Bitfield를 쓰면 더 공간을 절약할 가능성이 있습니다. 질문 2.26 참고. 이 경우, unsigned 타입의

고, 공간을 절약하기 위해 작은 데이터 타입을 썼다면, 나중에 int 타입으로 변환하는 코드가 많이 필요할 경우, 오히려 코드의 크기가 커지고 느려질 수 있다는 단점이 있습니다.)

#define과 enum 중 어느 것을 쓸 것인지는 그리 큰 문제가 되지 않습니다 (질문 2.22와 17.10을 참고하기 바랍니다.) 다음 둘 중 아무 것이나 써도 좋으며, 0과 1을 직접 써도 좋습니다:

```
#define TRUE  1             #define YES 1
#define FALSE 0             #define NO  0

enum bool {false, true};    enum bool {no, yes};
```

그러나 일단 어떤 것을 쓰겠다고 정했다면, 한 프로그램이나 프로젝트 내에서는 계속 그것을 쓰는 것이 좋습니다. (그러나 디버거에서 변수의 값을 검사할때, 수치가 아닌 이름으로 보여 줄 수 있으므로 enum이 더 좋을 수도 있습니다. (**Note** 질문 2.22 참고)

다음과 같이 typedef를 쓸 수도 있습니다:

```
typedef int bool;
```

또는

```
typedef char bool;
```

또는

```
typedef enum { false, true } bool;
```

어떤 사람들은 다음과 같이 쓰는 것을 선호합니다:

```
#define TRUE (1==1)
#define FALSE (!TRUE)
```

또는 다음과 같이 "도우미(helper)" 매크로를 만듭니다:

```
#define Istrue(e) ((e) != 0)
```

그렇지만 이런 방식이 특별히 좋다는 것은 아닙니다. (질문 9.2를 참고하기 바랍니다; 질문 5.12와 10.2도 보시기 바랍니다).

---

bitfield가 필요합니다. 1-bit로 된 signed 타입 bitfield는 호환성을 보장하면서 +1의 값을 저장할 수 없습니다.

Note C99에서 `_Bool` 타입을 제공하기 때문에, 위 설명은 이제 올바른 답변이 아닙니다. C 언어는 boolean 타입을 제공하며, 타입 이름은 `_Bool`입니다. 깔끔한 이름이 아닌, 이렇게 이상한 이름을 쓰는 이유는, 기존의 많은 C 프로그램들이, 위에서 설명한 것처럼 나름대로 boolean 타입을 이미 쓰고 있기 때문에, 이들 프로그램들과 호환을 위해서 이러한 이름을 쓰게 되었습니다. 표준 헤더 파일 `<stdbool.h>`를 포함시키면, 매크로 `bool`[2] 이 `_Bool`로 정의되어 있기 때문에, 다음과 같이 쓸 수 있습니다:

```
#include <stdbool.h>

_Bool a;    /* okay */
bool  b;    /* okay, using macro 'bool' */
```

또, `<stdbool.h>`는 매크로 `true`, `false`를 정의하고 있기 때문에 각각 1, 0 대신으로 쓸 수 있습니다.

Boolean 타입이 올 자리에 다른 정수 타입을 쓰게되면, 자동으로 변환되며, 아시다시피, 0이 아닌 모든 값은 1로, 0은 0으로 바뀝니다.

또한, C 언어가 boolean 타입을 지원한다고 해서, C++ 처럼, 비교 연산자들의 비교 결과가 `_Bool` 타입인 것은 아닙니다. 비교 연산자의 비교 결과는 여전히 `int`인 것을 주의하기 바랍니다. 여기에서 비교 연산자라고 하는 것은, 표준에서 relational operator와 equality operator를 포함한 것입니다. 즉, `<`, `<=`, `>`, `>=`, `==`, `!=`가 여기에 해당합니다.

References [C89] § 6.2.5 p. 33, § 6.3.1.2 p. 43, § 6.5.8 p. 86, § 6.5.9 p. 86 § 7.16 p. 252
[H&S2002] § 5.1.2 p. 128, § 5.1.5 p. 132 § 7.6.4 p. 233

Q 9.2 `#define`을 써서 `TRUE`를 1로 만드는 것은 위험하지 않을까요? C 언어에서는 0이 아닌 비트가 하나라도 있으면 참(true)으로 인식될 테니까요. 만약에 어떤 함수나 연산자가 1이 아닌 다른 값을 리턴할 경우 어떻게 되는 거죠?

Answer 맞습니다. 사실 C 언어에서는 0이 아닌 모든 값이 참(true)입니다. 그러나 이것은 단지 "입력"일 경우에만 의미를 가집니다; 다시 말하면 어떤 불리언 값이 예상되는 경우에만 그 의미를 가집니다. 만약 불리언 값이 내장된(built-in) 연산자에 의해 만들어진 것이라면 분명히 0 또는 1의 값을 가지게 됩니다. 따라서 다음과 같이 코드를 만들어도 안전합니다 (`TRUE`가 1이라고 가정할 때):

[2] 이미 C++ 언어는 boolean 타입으로 `bool`이라는 키워드를 제공하기 때문에, 주의가 필요합니다.

```
if ((a == b) == TRUE)
```

그러나 위와 같이 직접 TRUE나 FALSE와 같은 지 비교하는 것은 무의미할 뿐더러, 매우 위험합니다. 왜냐하면 어떤 라이브러리 함수들은 (특히 isupper(), isalpha(), 등등) 참(true)을 나타내기 위해 '0이 아닌 값'을 씁니다. 즉, 이때 '0이 아닌 값'은 꼭 1일 필요가 없습니다.

(또, 만약 여러분이 아래 코드보다

```
if (a == b)
```

다음 코드가 더 향상된 것이라고 믿는다면,

```
if ((a == b) == TRUE)
```

왜 다음과 같이 쓰지 않나요?

```
if (((a == b) == TRUE) == TRUE)
```

아예 다음과 같이 쓰는게 더 좋은 것이라고 믿는 것인가요?

```
if ((((a == b) == TRUE) == TRUE) == TRUE)
```

Lewis Caroll의 에세이, "What the Tortoise Said to Achilles."를 읽어보시기 바랍니다.)

if (a == b)가 완벽하게 올바른 것과 마찬가지로, 아래 코드도 완벽히 올바른 코드입니다:

```
#include <ctype.h>
...
if (isupper(c)) {
  ...
}
```

왜냐하면, isupper는, 참/거짓을, zero/nonzero로 알려주기 때문입니다. 비슷하게, 아래 코드는 좋은 코드입니다:

```
int isvegetable;   /* really a bool */
...
if (isvegetable) {
  ...
}
```

마찬가지로, 아래 코드도 좋습니다:

```
extern int fileexists(char *);  /* returns true/false */
...
if (fileexists(outfile)) {
  ...
}
```

위 두 예에서 isvegetable과 fileexists()는 개념상 boolean 타입입니다. 따라서 다음과 같이 쓰는 것은 전혀 개선이라고 할 수 없습니다:

```
if (isvegetable == TRUE)
```

또는,

```
if (fileexists(outfile) == YES)
```

(이러한 스타일이 좀 더 "안전한" 또는 "좋은 스타일"이라고 생각한다면, 정말로 위험하고 나쁜 스타일을 믿고 있다고 말할 수 있습니다. 가독성도 오히려 떨어집니다. 질문 17.10을 참고하기 바랍니다.)

여기에서 배울 가장 중요한 규칙은, 값을 대입할 때, 함수의 인자로 전달할 때, 또는 불리언 값을 리턴할 때에만 TRUE나 FALSE를 쓴다입니다. 즉, 비교할 때에 이런 것을 쓰는 것은 좋지 않습니다. (질문 5.3 참고)

TRUE나 FALSE와 같은 (또는 YES와 NO) 매크로를 쓰는 것은 어떻게 보면 좋은 것 같지만 C 언어에서 불리언 값이라는 개념이 헷갈리기 때문에, 어떤 프로그래머들은 0과 1이라는 수치를 직접 쓰는 것을 선호하기도 합니다. (질문 5.9를 참고하기 바랍니다.)

References [K&R1] § 2.6 p. 39, § 2.7 p. 41
[K&R2] § 2.6 p. 42, § 2.7 p. 44, § A7.4.7 p. 204, § A7.9 p. 206
[C89] § 6.3.3.3, § 6.3.8, § 6.3.9, § 6.3.13, § 6.3.14, § 6.3.15, § 6.6.4.1, § 6.6.5
[H&S] § 7.5.4 pp. 196–7, § 7.6.4 pp. 207–8, § 7.6.5 pp. 208–9,
[H&S] § 7.7 pp. 217–8, § 7.8 pp. 218–9, § 8.5 pp. 238–9, § 8.6 pp. 241–4
Carroll, "What the Tortoise Said to Achilles"

Note C99는 표준 헤더 파일, <stdbool.h>을 통해서 매크로 true와 false를 제공하며, 각각 0과 1로 정의되어 있습니다. 질문 9.1에서 새 boolean type인 _Bool을 참고하기 바랍니다.

Q 9.3 p가 포인터일때, `if (p)`와 같은 코드를 쓰는 것은 나쁜가요?

Answer 아닙니다. 좋습니다. 질문 5.3을 참고하기 바랍니다.

Q 9.4 `TRUE`나 `FALSE`와 같은 심볼을 쓰는 것이 좋을까요, 아니면, 1과 0을 직접 쓰는 것이 좋을까요?

Answer 그것은 여러분이 결정할 문제입니다. Preprocessor 매크로인 `TRUE`, `FALSE`는 (물론 `NULL` 포함) 단지 코드를 읽기 쉽게 하기 위해 쓰는 것이지, 이들 매크로가 나타내는 값이 바뀔 가능성이 있기 때문은 아닙니다. (질문 5.10, 17.10 참고) 즉 이것은 스타일에 관한 문제이지 옳고 그른 문제가 아닙니다.

어떤 사람들은 `TRUE`와 `FALSE`같은 심볼 이름을 쓰는 것은, 코드를 읽는 사람에게 boolean 값이 쓰인다는 것을 알려주는 역할을 하기 때문에, 좋다고 합니다. 또 어떤 사람들은 어차피 C 언어에서 boolean 값과 정의가 혼동스럽기 때문에, `TRUE`와 `FALSE`같은 매크로를 쓰는 것은 오히려 더 복잡하게 만든다고 생각합니다. (질문 5.9 참고)

Q 9.5 다른 곳에서 라이브러리를 받아 왔는데, 그 라이브러리에서 이미 `TRUE`와 `FALSE`를 다르게 정의하고 쓰고 있어서, 제 코드와 충돌이 일어납니다.

Answer 질문 10.10을 보기 바랍니다.

Chapter 10

C Preprocessor

C's preprocessor provides reasonable solutions to a number of software engineering and configuration management problems, although its syntax is unlike the rest of C in several respects. As its name suggests, the preprocessor operates *before* formal parsing and compilation begin. Because it doesn't know about the structure of the code as seen by the rest of the compiler, it cannot do any processing that relates to declared types or function structure.

The first part of this chapter is arranged around the major preprocessor directives: `#define` (questions 10.1 through 10.5), `#include` (questions 10.6 through 10.11), and `#if` (questions 10.12 through 10.19). Questions 10.20 through 10.25 cover fancier macro replacement, and finally questions 10.26 and 10.27 cover a particular set of problems relating to the preprocessor's lack of support for variable-length macro argument lists.

Q 10.1 다음과 같이 간단한, 함수와 비슷한 매크로를 만들려고 합니다.

```
#define square(x)   x * x
```

그런데, 가끔씩 제대로 동작하지 않습니다. 왜 그럴까요?

Answer 매크로 확장(expansion)은 순수하게 텍스트로 이루어집니다. 즉, 쓰여진 그대로 확장된다는 뜻이며, 때때로 (위와 같이) 개발자가 원하지 않은 결과를 가져올 수 있습니다. 함수같은 매크로를 정의할 때에는 다음과 같은 세가지 규칙을 지켜야 합니다:

(a) 매크로를 포함한 expression에서 연산자 우선 순위 때문에 원하지 않은 결과를 얻을 수 있기 때문에, 항상 정의 전체를 괄호로 둘러 싸야 합니

다. 예를 들어 질문에서 제공한 `square()` 매크로를 다음과 같이 썼다고 생각해 봅시다:

```
i / square(n)
```

그러면, 위 expression은 다음과 같이 확장됩니다:

```
i / n * n
```

따라서, `(i / n) * n`으로 평가됩니다. 물론 여러분이 의미한 것은 다음과 같습니다:

```
i / (n * n)
```

이 경우는, 우선 순위(precedence)라기 보다는, 결합성(associativity)에 관계된 문제입니다만, 결과는 같습니다.

(b) 매크로 정의 안에서, 모든 파라메터들은 괄호로 둘러 싸서 원치 않는 우선순위 문제를 제거합니다. 다시 한번, 질문에서 준 매크로를 다음과 같이 썼다고 가정합시다:

```
square(n + 1)
```

그러면, 위 expression은 다음과 같이 확장됩니다:

```
n + 1 * n + 1
```

즉, 다음과 같이, 우리가 원한 것과는 다른 결과가 나옵니다:

```
(n + 1) * (n + 1)
```

(c) 만약에, 매크로가 확장될 때 어떤 파라메터가 여러 번 쓰였다면, 그리고 매크로에 전달한 actual argument가 side effect를 가지고 있는 것이라면, 원하는 결과를 얻을 수 없을 수도 있습니다. 예를 들어 다음과 같이 매크로를 쓰면:

```
square(i++)
```

다음과 같이 확장되기 때문에, undefined behavior를 발생시킵니다. (질문 3.2 참고)

```
i++ * i++
```

즉, 규칙 1과 2에 따라서, `square()` 매크로는 다음과 같이 정의해야 합니다.

```
#define square(x) ((x) * (x))
```

규칙 3을 지키는 것은 좀 어렵습니다. 때때로, `&&`나 `||`, `?:` 연산자의 short-circuit 방식을 쓰면 (질문 3.6 참고), 파라메터가 딱 한번만 평가되도록 할 수 있습니다. 또는 매크로가 안전하지 않으니, 부를 때, side effect가 발생하지 않도록 조심하라고 문서에 써 두는 것이 좋습니다. 또는, 안전하게 만들 수 없는 것이면, 매크로로 만들지 않는 편이 좋을 때도 있습니다.

일반적으로, (스타일에 관계되어) 매크로 이름에는 대문자를 쓰거나, 또는 전부 대문자로 써서, 매크로라는 것을 암시해 줍니다. 만약, 완전하게 함수와 같이 동작하는 것이라면, 소문자로만 이름을 만드는 것도 괜찮습니다. 그러나 이 경우, 위 세가지 규칙을 준수하는지 확인하기 바랍니다. 질문에서 제공한 매크로는 위의 세가지 규칙을 지킬 수 없으므로, 다음과 같이 만드는 것이 낫습니다:

```
#define Square(x)  ((x) * (x))    /* UNSAFE */
```

References [K&R1] § 4.11 p. 87
[K&R2] § 4.11.2 p. 90
[H&S] §§ 3.3.6, 3.3.7 pp. 49–50
[CT&P] § 6.2 pp. 78–80

Q 10.2 다음과 같은 매크로를 만들면, C 코드를 Pascal처럼 쓸 수 있습니다.

```
#define begin    {
#define end      }
```

Answer 이런 식으로 매크로를 쓰는 것은, 때때로 아주 매력적으로 보이기는 하지만, 권장되지 않는 방법입니다; 대부분의 경우, 이렇게 쓰는 것을 "preprocessor abuse"라고 (남용) 부릅니다. Your predilections are unlikely to be shared by later readers or maintainers of the code, and any simulation of another language is most unlikely to be perfect (so any alleged convenience or utility will probably be outweighed by the nuisance of remembering the imperfections).

일반적으로, 프리프로세서를 써서 만드는 매크로는 C 언어 형식을 따르는 것이 좋습니다. 인자가 없는 매크로는 변수나 다른 identifier처럼 보이게 만들어야 하며, 인자가 있는 매크로는 함수처럼 만들어야 합니다. "만약 프리프로세서를 거치지 않고, 이 코드를 직접 컴파일러에 주었을 때, 문법 에러가 (syntax error) 몇 개나 발생할까?"라고 자신에게 묻는 습관을 가지면 좋습니다. (물론 undefined symbol, non-constant array dimension에 관한 많은 에러가 발생하겠지만, 이들은 문법 에러가 아닙니다.) 즉, C 코드는, 매크로 호출을 거치기 전에도, C 코드처럼 보여야 합니다. `begin`, `end`, 또는

CTRL(D)와 (질문 10.21 참고) 같은, nonsyntactic macro는 C 코드를 gobbledygook[1]처럼 보이게 만듭니다. 물론 이런 것은 스타일에 관한 것입니다. (Chapter 17 참고)

Q 10.3 두 변수의 값을 바꾸기 위한 일반적인 매크로를 만들 수 있을까요?

Answer 이 질문에 대한 좋은 답변은 없습니다. 변수가 정수형일때에는 잘 알려진 exclusive OR를 쓰는 방법이 있긴 합니다만 타입이 포인터나 실수(floating-point)일 때에는 동작하지 않으며, 만약 두 값이 같은 변수에 저장되어 있을 때에도 동작하지 않습니다. (또, 정수 타입에 쓸 수 있는, 아주 압축된 코드인 `a ^= b ^= a ^= b`는 중복된 side effect를 발생시키므로 나쁜 코드입니다. 질문 3.2, 3.3b, 20.15c 참고) 만약, 모든 타입에 대해 쓸 수 있는 매크로를 만든다고 하면, 임시 변수가 반드시 필요하며, 분명 문제가 있는 코드가 됩니다. 왜냐하면:

- 다른 이름과 절대로 충돌하지 않는다고 보장할 수 있는 임시 변수를 만들 수 없습니다. 즉, 어떤 이름을 선택하더라도, 결국 이 이름과 같은 변수가 이미 존재할 확률이 있으며, 따라서 그 변수의 값이 바뀌어 버리는 경우가 생길 수 있습니다. 만약 `##` 연산자를, 두 인자에 써서, 이름을 만들어 낸다고 해도, 합쳐서 만든 이름이 31 글자를 넘는다면[2], 다른 이름과 겹치지 않는다고 보장할 수 없습니다. 또한 간단한 이름이 아닌, 예를 들어 `a[i]`와 같은 인자가 들어왔을 때에는 제대로 동작하지 않을 것입니다. 질문 1.29에서 다룬 것처럼, `__tmp`와 같은 이름을 써서, 사용자나 컴파일러 namespace에서 (완벽히 보장할 수는 없지만) 쓰이지 않는 이름을 만들 수도 있습니다.
- 또, 임시 변수를 정확한 타입으로 선언할 방법이 없습니다. (C 표준에서는 `typeof`와 같은 연산자를 제공하지 않습니다.) 혹은, 적당한 크기의 메모리 공간을 만들고 그 곳에 `memcpy`와 `sizeof` 연산자를 써서 바이트 단위로 복사하는 방식을 쓸 수도 있으나, 만약 매크로에 전달된 인자가 `register`로 선언되었다면 이 방법도 불가능합니다.

가장 좋은 해결책은, 두 변수의 타입을 매크로 인자로 전달하지 않는 한, 위와 같은 매크로로 처리하겠다는 생각을 접는 것입니다. (또, 만약에 어떤 structure나 배열을 바꾸겠다는 생각을 했다면, 단순히 이들을 가리키는 포인터의 값을 바꾸는 것으로 해결할 수 있습니다.)

[1] 은어, 특수어, 전문용어, 특수한 계층의 사람들만 쓰는 용어 (경멸적으로 쓰임)

[2] C 표준은, 컴파일러가 이름을 비교할 때, 31글자 다음의 글자들을 비교해야 한다고 강요하지 않습니다.

다시 말하지만, 두 변수의 내용을 매크로를 써서 바꾸겠다는 생각에 사로잡혀 있다면, 재고하기 바랍니다. 당신의 에너지를 소비할 가치가 있는 다른 일들도 많이 널려 있습니다.

Q 10.4 여러 문장으로 이루어진 매크로를 만드는 좋은 방법 좀 알려 주세요.

Answer 이런 매크로를 만드는 일반적인 방법은 매크로 자체를 일반 함수처럼 쓸 수 있도록 하는 것입니다. 즉 호출하는 쪽에서 마지막 ';'을 직접 써 주게 하고 매크로의 몸통에서는 ';'을 따로 써 주지 않는 식으로 쓰는 것입니다. 예를 들면:

```
MACRO(arg1, arg2);
```

그러므로 매크로 정의는 단순히 여러 statement를 중괄호로 둘러싼 'compound statement' 형식으로 만들 수 없습니다. 왜냐하면 이 매크로가 호출될 때 세미콜론이 추가적으로 붙는다면 `if`나 `if/else` 문장에서 에러가 발생할 수 있기 때문입니다. 다음 코드를 보기 바랍니다:

```
if (cond)
  MACRO(arg1, arg2);
else  /* some other code */
```

만약, 단순히 중괄호로 둘러싼 것으로 매크로가 확장된다면, 위 코드는 다음과 같이 해석되어, 마지막 세미콜론 때문에 syntax error가 발생합니다:

```
if (cond)
  { stmt1; stmt2; };
else  /* some other code */
```

전통적으로, 이 경우에는 다음과 같은 방법을 씁니다:

```
#define MACRO(arg1, arg2) do { \
    /* declarations */         \
    stmt1;                     \
    stmt2;                     \
    /* ...  */                 \
  } while(0) /* (no trailing ; ) */
```

매크로를 부르는 쪽이 세미콜론을 붙이면, 매크로가 하나의 문장으로 해석됩니다. (좋은 최적화를 제공하는 컴파일러라면 필요없는 테스트[3]나, 조건문에서 상수 0을 검사하는 일[4]같은 것은 알아서 없애 줍니다. 그러나 lint는 불평할 수 있습니다.)

또 다른 방법으로 다음과 같이 할 수도 있습니다:

```
#define MACRO(arg1, arg2) if (1) \
    stmt1;                       \
    stmt2;                       \
  } else
```

그러나, 이 경우, 만약에 부르는 쪽에서 세미콜론을 빼먹을 경우, 전체 코드가 엉망이 되 버릴 경우가 있어서 좋은 방법이 아닙니다.

매크로의 모든 문장이 매우 간단한 expression이라면, 즉 선언이나 루프가 없다면, 콤마(‘,’) 연산자를 써서 한 문장으로 만들 수 있습니다:

```
#define FUNC(arg1, arg2)   (expr1, expr2, expr3)
```

예를 들어 질문 10.26의 `DEBUG()` 매크로를 보기 바랍니다. 이 테크닉은 또 매크로가 어떤 ‘값(value)’을 리턴할 수 있게 해 줍니다.

`gcc`와 같은 컴파일러는 이런 간단한 기능을 하는 함수가 매크로처럼 원하는 경우 확장(expand)할 수 있는 기능을, 비표준인, “`inline`” 키워드나 기타 방법을 써서 제공하기도 합니다.

Note 무슨 소리냐 하면 매크로를 정의할 때, 마지막에 세미콜론을 붙여 버린다면 아래와 같은 상황에서 에러가 발생할 수도 있다는 뜻입니다:

```
#define MULTI_STATEMNT_MACRO(x) do { \
    stmt1; \
    stmt2; \
  } while(0);

if (some_condition)
  MULTI_STATEMNT_MACRO(a);
else {
  /* ...  */
}
```

[3] “dead” test

[4] branches on the constant condition 0

매크로를 확장해보면 세미콜론이 두 번 만들어지고, 따라서 `else` 부분에서 에러가 발생합니다.

References [H&S] § 3.3.2 p. 45
[CT&P] § 6.3 pp. 82–3

Note "`inline`"은 C99에서 표준이 되었습니다.

References

Q 10.5 새로 만든 user-defined 타입을 typedef로 만드는 것이 좋을까요, 아니면, preprocessor macro로 만드는 것이 좋을까요?

Answer 질문 1.13을 보기 바랍니다.

Q 10.6 여러 개의 소스 파일로 이루어진 프로그램을 만들었습니다. 그런데, 어떤 것들을 `.c` 파일에 두어야 하고, 어떤 것들을 `.h` 파일에 두어야 하는지를 모르겠습니다. (또 "`.h`"는 어디에 쓰이나요?)

Answer 보통 헤더 (`.h`) 파일에 넣는 것들은 다음과 같습니다:

- 매크로 정의 (`#define`)
- structure, `union`, `enum` 선언
- `typedef` 선언
- 외부(external) 함수 선언 (질문 1.11 참고)
- 전역(global) 변수 선언

특히 여러 파일에서 공통적으로 쓰이는 선언이나 정의는 꼭 헤더 파일에 넣는 것이 중요합니다. 공통적으로 쓰이는 이름을 여러 소스 파일에 중복해서 선언하거나 정의하지 말기 바랍니다; 이런 부분들은 헤더 파일에 집어 넣고, `#include`를 써서 포함해야 합니다. 단순히 타이핑하는 수고를 덜기 위해 이러한 규칙을 정한 것이 아닙니다. 만약 공통적인 부분을 나중에 고쳐야 한다면, 한 파일을 고쳐서, 이 변경 사항이 모든 소스 파일에 반영되도록 해야 하기 때문입니다. (특히, 절대로 외부 함수 prototype을 `.c`에 넣어서는 안됩니다. 질문 1.7을 참고하기 바랍니다.)

또, 정의나 선언이 하나의 `.c` 파일에서만 쓰인다면, 그 파일에 두어도 좋습니다. (그리고 이러한 private file-scope 함수나 변수들은 `static`으로 선언되어야 합니다. 덧붙여 질문 2.4도 참고하시기 바랍니다.)

마지막으로, 실제 코드(actual code)나 (예를 들어, 함수의 몸체), 전역 변수 정의를 (정의 또는 초기화하는 코드) 헤더 파일에 두면 안됩니다. 또, 여러 소스 파일에서부터 프로젝트를 설계했다면, 각각의 소스 파일을 따로 컴파일하고 (대개 컴파일러 옵션을 써서 컴파일만 하게 할 수 있습니다) 마지막으로 모든 오브젝트 파일을 링크해야 합니다. (일반적으로 통합 환경(IDE, integrated development environment)에서는 이 모든 작업이 자동으로 진행됩니다.) Don't try to "link" all of your source files together with `#include`; the `#include` directive should be used to pull in header files, not other .c files.

덧붙여 질문 1.7, 10.7, 17.2도 참고하시기 바랍니다.

References [K&R2] § 4.5 pp. 81–2
[H&S] § 9.2.3 p. 267
[CT&P] § 4.6 pp. 66–7

Q 10.7 헤더 파일에서 다른 파일을 `#include` 하는 것은 괜찮나요?

Answer 스타일에 관한 질문이군요. 따라서 상당한 논란의 여지가 있습니다.

많은 사람들이 "중첩된(nested) `#include` 파일"을 쓰지 않는 것이 좋다고 말합니다: 권위있는 Indian Hill Style Guide에서도 (질문 17.9 참고) 이런 쓰임새를 피하라고 씌여있습니다; 관련된 정의를 찾기가 훨씬 더 어렵기 때문입니다; 또한 두번 `#include`하는 경우, 중복된 정의 에러가 (multiple-definition error) 발생할 가능성이 높습니다; 또 수동으로 Makefile을 만들 경우, 상당히 복잡해질 가능성이 있습니다.

그러나 헤더 파일을 중첩하여 포함할 경우, 각각의 헤더 파일을 모듈화해서 (modular way), 헤더 파일에서 필요한 다른 헤더 파일을 `#include`함으로써 수고를 덜어줄 수 있다는 장점도 있습니다; `grep`과 같은 툴을 (또는 tags 파일) 사용하면 정의가 어떤 파일에 되어 있느냐에 상관없이 쉽게 찾을 수 있습니다. 다음과 같은 트릭을 쓰면, 헤더 파일이 여러 곳에서 포함되었느냐에 상관없이, 딱 한번만 포함되게(idempotent) 할 수 있습니다:

```
#ifndef HFILENAME_USED
#define HFILENAME_USED
...header file contents...
#endif
```

(이 때, 각각의 헤더 파일에 각각 다른 매크로 이름을 사용합니다) 이 방식은 헤더 파일이 꼭 한 번만 포함되도록 해 주므로 여러 번 `#include`하더라도

문제가 발생하지 않습니다; 또한 자동으로 Makefile을 관리해주는 툴을 (큰 프로젝트를 관리할 때에는 꼭 필요합니다, 질문 18.1 참고) 사용할 경우, 중첩된 `#include`를 처리해 주므로 좀 더 편합니다. 질문 17.10을 참고하기 바랍니다.

Note GNU autoconf와 GNU automake는 Makefile을 관리해주는 아주 좋은 툴입니다.

References [ANSI Rationale] § 4.1.2

Q 10.8a `#include <>`와 `#include ""`에 차이가 있나요?

Answer `<>` 형식은 헤더 파일이 시스템에서 제공한 것이거나 표준 헤더 파일일 경우에 사용하며, `""`는 프로그래머가 제작한 헤더 파일에 사용합니다.

Note 아래 질문 10.8b를 참고하기 바랍니다.

Q 10.8b 헤더 파일을 찾는 알고리즘을 알고 싶어요.

Answer 정확한 알고리즘은 구현 방법에 따라 다릅니다 (implementation-defined). (즉, 이 방법에 대해 문서화되어 있을 가능성이 높습니다. 질문 11.33을 참고하기 바랍니다).

일반적으로 `<>` 형식으로 포함된 헤더 파일들은 하나 이상의 표준으로 지정된 디렉토리에서 찾게 됩니다. (게다가, `<>`로 포함한 헤더들은 꼭 파일 형식으로 저장되어 있을 필요도 없습니다.) `""` 형식으로 포함된 헤더 파일은 우선 "현재 디렉토리"에서 찾은 다음, 없을 경우, 표준으로 지정된 디렉토리에서 찾게 됩니다.

전형적으로 (특히 UNIX 컴파일러), 현재 디렉토리란 `#include` 를 쓴 파일이 있는 디렉토리를 말합니다. 다른 컴파일러에서는 현재 디렉토리가 컴파일러가 실행된 그 디렉토리를 의미하기도 합니다. (물론, "현재 디렉토리"라는 개념이 없는 시스템이나, 디렉토리 자체가 없는 시스템에서는 다른 규칙이 있을 것입니다)

일반적으로, 시스템 헤더 파일이 위치할 디렉토리 목록에 사용자가 원하는 디렉토리를 추가할 수 있는 방법이 (보통 컴파일러를 실행할 때 command-line option 'I'를 쓰거나, 환경 변수를 지정하는 방법으로) 제공됩니다. 좀 더 자세한 것은 컴파일러의 매뉴얼을 참고하기 바랍니다.

References [K&R2] § A12.4 p. 231
[ANSI] § 3.8.2
[C89] § 6.8.2
[H&S] § 3.4 p. 55

Q 10.9 잘못된 것이 없어 보이는 데에도 코드의 첫번째 선언문에서 'syntax error'가 발생합니다.

Answer 아마도 마지막으로 `#include`한 헤더 파일의 내용에서, 마지막 선언 부분에서 세미콜론(semicolon)이 빠져 있을 가능성이 높습니다. 질문 2.18, 11.29, 16.1b를 참고하기 바랍니다.

Q 10.10 다른 곳에서 제작한 라이브러리 두 개를 쓰는 헤더 파일을 만들었습니다. 그런데 이 라이브러리들이, 도움을 준답시고, 여러 매크로들을 정의했습니다. 예를 들면, `TRUE`, `FALSE`, `Min()`, `Max()` 등. 그래서 컴파일할 때, 제가 만든 헤더 파일에서 매크로가 여러번 정의되어 있다고 에러가 발생합니다. 어떻게 하면 되죠?

Answer 매우 짜증나는 상황 중 하나입니다. 전형적인 namespace problem이라 할 수 있으며, 질문 1.9, 1.29를 참고하기 바랍니다. 이상적으로(ideally), 라이브러리 제작자는 symbol을 (예를 들어, 매크로, 전역 변수 및 함수) 정의할 때, 이름이 서로 충돌하지 않도록, 매우 조심스럽게 해야 합니다. 가장 좋은 방법은, 라이브러리 제작자에게 연락해서, 코드를 고치는 것입니다. 힘들다면, 임시적으로, 매크로 정의를 없애고(undefine), 다시 정의하는(redefine) 것을 반복해서 `#include`에서 일어나는 충돌을 막을 수 있습니다.

Note 서로 충돌하는 매크로의 실제 정의 부분이, 다음과 같이 서로 같다면:

```
file1.h:
  #include TRUE 1
  ...
file2.h:
  #include TRUE 1
  ...
```

다음과 같이 두 파일의 내용을 고쳐서 해결할 수 있습니다:

```
#ifndef TRUE
#define TRUE 1
#endif
```

그러나 이 방법은 두 매크로의 내용이 완전히 같을 때 쓸 수 있습니다. 서로 내용이 다른 매크로라면, 코드가 안전하다고 보장할 수 없습니다.

Q 10.10b 라이브러리 함수 정의를 포함하는 헤더 파일을 제대로 `#include`시켰는데도 링커(linker)는 정의되어 있지 않다고 에러를 발생합니다.

Answer 질문 13.25를 참고하기 바랍니다.

Q 10.11 시스템 헤더 파일인 `<sgtty.h>`가 없습니다. 제게 복사본을 주실 수는 없을까요?

Answer 표준 헤더 파일들은 컴파일러, 운영체제, 프로세서에 따라 서로 다른 정의로 이루어져 있습니다. 따라서 다른 사람의 헤더 파일을 가져온다 하더라도 제대로 동작하지 않을 것입니다. 물론 그 사람이 여러분과 완전히 같은 시스템을 사용하고 있다면 동작할 수도 있습니다. 컴파일러 vendor에게, 왜 그 헤더 파일이 제공되지 않는지 물어보고, 그 파일을 달라고 요청해 보시기 바랍니다.

표준이 아닌 헤더 파일인 경우에는 조금 더 까다롭습니다. 어떤 헤더 파일들은 (예를 들어 `<dos.h>`), 완전하게 컴파일러 또는 OS에 종속되어 제공됩니다. (즉, 다른 시스템에서는 원래 제공되지 않을 수 있습니다.) 이와 달리, 인기있는, 라이브러리에서 제공하는 헤더 파일이 없는 경우에는, 이 라이브러리를 얻은 곳에서 찾아보기 바랍니다. 그 헤더 파일을 쓰는 소스는 있는데, 그 헤더 파일이 없는 경우, 관련 라이브러리 자체가 설치되어 있지 않을 수도 있습니다. (질문 13.25 참고) 때때로 archie가 여러분이 찾는 것에 대해 도움을 줄 수도 있습니다; 질문 18.16 참고.

Note ARCHIE는 1990년에 Alan Emtage, Bill Heelan, Peter J. Deutsch가 만든 FTP 검색 프로그램이며, 현재에는 거의 쓰이지 않습니다.

10.1 Conditional Compilation

Q 10.12 전처리기(preprocessor) `#if` 수식에서 문자열을 비교할 수 있을까요?

Answer 직접 비교할 수 없습니다; `#if`는 정수 연산만 지원합니다. 따라서, 여러 상황을 정수 상수로 정의하고 다음과 같이 할 수 있습니다:

```
#define RED        1
#define BLUE       2
#define GREEN      3

#if COLOR == RED
/* red case */
#else
#if COLOR == BLUE
/* blue case */
#else
#if COLOR == GREEN
/* green case */
#else
/* default case */
#endif
#endif
#endif
```

(C 표준에서 소개된 `#elif`를 쓰면 위 코드를 조금 더 깔끔하게 만들 수 있습니다.)

질문 20.17을 참고하기 바랍니다.

Note `#elif`를 쓰면 위 코드를 다음과 같이 쓸 수 있습니다.

```
#define RED        1
#define BLUE       2
#define GREEN      3

#if COLOR == RED
/* red case */
#elif COLOR == BLUE
/* blue case */
#elif COLOR == GREEN
/* green case */
#else
/* default case */
```

```
#endif
```

References [K&R2] § 4.11.3 p. 91
[ANSI] § 3.8.1
[C89] § 6.8.1
[H&S] § 7.11.1 p. 225

Q 10.13 `#if` 지시어(directive)에서 `sizeof` 연산을 쓸 수 있을까요?

Answer 쓸 수 없습니다. '전처리(preprocessing)'는 말 그대로 (타입 이름을 parsing 하기 전에) 컴파일 초기 단계 이전에 이루어지기 때문입니다. `sizeof`를 쓰는 대신 ANSI 표준인 `<limits.h>`에 정의되어 있는 상수를 쓰는 방법으로 바꾸기 바랍니다. 가능하다면, "configure" 스크립트(script)를 쓰는 것도 좋습니다. (프로그램을 타입의 크기에 독립적으로 작성하는 게 더 바람직합니다; 질문 1.1을 참고하기 바랍니다.)

References [ANSI] § 2.1.1.2, § 3.8.1 footnote 83
[C89] § 5.1.1.2, § 6.8.1
[H&S] § 7.11.1 p. 225

Note GNU autoconf 패키지는 시스템에서 어떤 기능을 제공하는지, 어떤 타입의 크기가 얼마인지 알아내는 'configure' 스크립트를 자동으로 만들어 줍니다. 이 'configure'를 실행하면 Makefile이 만들어지므로, 사용자가 소스에서 실행 파일을 만드는 데 필요한 수고를 많이 덜어 줍니다. 이 패키지에 관한 것은 아래 URL을 참고하기 바랍니다:

```
http://www.gnu.org/software/autoconf/
ftp://ftp.gnu.org/pub/gnu/autoconf/
```

Q 10.14 `#define` 줄에서 `#ifdef`를 써서 다음과 같이 각각 다른 방식으로 정의하게 할 수 있을까요?

```
#define a b \
#ifdef whatever
        c d
#else
        e f g
#endif
```

Answer 안됩니다. 프리프로세서를 자신에게 또 실행하는 (run the preprocessor on itself) 것은 불가능합니다. 대신 `#ifdef`를 써서 두 개의 `#define` 문장으로 만드는 방식을 쓰기 바랍니다:

```
#ifdef whatever
#define a b c d
#else
#define a b e f g
#endif
```

References [ANSI] § 3.8.3, § 3.8.3.4
[C89] § 6.8.3, § 6.8.3.4
[H&S] § 3.2 pp. 40–1

Q 10.15 `typedef` 이름을 `#ifdef`와 같이 테스트할 수 있는 방법이 있을까요?

Answer 불행하게도 그런 방법은 존재하지 않습니다. (왜냐하면, `typedef`와 타입은 프리프로세싱할 때 준비되어 있지 않기 때문입니다.) 질문 1.13, 10.13을 참고하기 바랍니다.

대신 이런 방법을 생각할 수 있습니다: 몇가지 매크로를 정의해서 (예: `MY_TYPE_DEFINED`) 어떤 `typedef`들이 선언되어 있는지 매크로를 써서 검사할 수 있습니다 (물론, 완전하지 않습니다).

References [ANSI] § 2.1.1.2, § 3.8.1 footnote 83
[C89] § 5.1.1.2, § 6.8.1
[H&S] § 7.11.1 p. 225

Q 10.16 컴퓨터가 'big-endian' 방식인지, 'little-endian' 방식인지 `#if`를 써서 검사할 수 있을까요?

Answer 거의 불가능합니다. Endian을 확인하기 위해서는 대개 포인터와 `char` 배열 또는 `union`을 써서 확인하는데, 프리프로세서 연산은 단지 `long` 정수만 쓰며, 주소(addressing)에 대한 개념이 프리프로세서에 존재하지 않습니다.

정말로 컴퓨터의 endian을 알아야 할 필요가 있는지 잘 생각해보기 바랍니다. 될 수 있으면, 이런 것과 무관하게 코드를 작성하는 것이 바람직합니다. (예를 들어 질문 12.42에 나온 코드를 보기 바랍니다.) 덧붙여 질문 20.9도 참고하시기 바랍니다.

References [ANSI] § 3.8.1
[C89] § 6.8.1
[H&S] § 7.11.1 p. 225

Note GNU autoconf를 쓰면 프로그램을 컴파일하기 전 간단한 인디언 테스트용 프로그램을 실행해서, 자동으로 어떤 매크로를 정의하게 만들고 (예를 들어 `BIG_ENDIAN_MACHINE`) 프로그래머가 이 매크로의 정의 여부에 따라 프로그램을 작성할 수 있습니다. (질문 10.13 참고).

Q 10.17 분명히 `#ifdef`를 써서 포함되지 않도록 한 부분에서 이상한 syntax error가 생깁니다.

Answer 질문 11.19를 보기 바랍니다.

Q 10.18 어떤 코드를 분석하려 하는데, 너무나도 많은 `#ifdef` 때문에 어렵습니다. 어떤 조건부 컴파일에 관한 것만 남겨두고 나머지 부분만 'preprocssing'할 수 있는 방법이 있을까요? (물론 `#include`나 `#define`은 제외한다는 조건하에서)

Answer 이러한 일을 처리해 주는 `unifdef`, `rmifdef`, `scpp` ("selective C preprocessor") 등의 프로그램이 있습니다. 질문 18.16을 참고하기 바랍니다.

Q 10.19 미리 정의된(predefined) 모든 매크로들을 뽑아낼 수 있을까요?

Answer 자주 언급되는 사항임에도 이런 일을 할 수 있는 표준 방법은 없습니다. 컴파일러 문서에 만족할만한 정보가 없다면, 가장 적당한 방법으로, 컴파일러 또는 프리프로세서 실행 파일에서, 파일에서 문자열을 뽑아내는 UNIX `strings`와 같은 프로그램을 쓰면 됩니다. `gcc`는 `-E`와 함께 사용할 수 있는 `-dM` 옵션을 제공합니다. 다른 컴파일러도 비슷한 기능을 제공할 것입니다. 많은 오래된 (traditional) 시스템 전용의 predefined identifier들이 (예를 들어 "unix") 비표준이므로 (왜냐하면, user namespace와 충돌하기 때문), 조심하기 바랍니다. 이런 비표준인 이름들은 대개 (표준에서) 곧 없어질 예정이거나 다른 이름으로 바뀔 예정인 것들이 많습니다. (어떤 경우라도 조건부 컴파일을 최소화하는 것이 좋은 방법이라는 것을 잊지 말기 바랍니다.)

10.2 Fancier Processing

Macro replacement can get fairly complicated—sometimes *too* complicated. For two somewhat popular tricks that used to work (if at all) by accident, namely, "token pasting" and replacement inside string literals, ANSI C introduces defined, supported mechanisms.

Q 10.20 다음과 같이 'identifier'를 생성하는 매크로를 본 적이 있습니다:

```
#define Paste(a, b) a/**/b
```

그러나 동작하질 않는군요.

Answer (특히 John Reiser의) 오래된 전처리기에서는, 주석(comment)을 봤을 때, 주석 자체를 없애기 때문에, 주석 양 옆의 토큰(token)을 서로 붙이는데 (pasting) 씁니다. 그러나, ANSI 표준에서는 ([K&R1] 포함), Comment가 공백으로 대체된다고 씌여있기 때문에, 더 이상 쓸 수 없는 기능입니다. 따라서 위 `Paste()` 매크로는 쓸 수 없습니다. 대신, 토큰을 서로 붙이는 기능 자체는 필요하기 때문에, ANSI는 이 역할을 하는 연산자인 `##`를 제공합니다. 따라서 위 매크로는 다음과 같이 만들 수 있습니다:

```
#define Paste(a, b) a##b
```

참고로 ANSI 이전의 컴파일러에서 쓸 수 있는 또다른 방법은 다음과 같습니다:

```
#define XPaste(s)   s
#define Paste(a, b) XPaste(a)b
```

질문 11.17을 참고하기 바랍니다.

References [ANSI] § 3.8.3.3
[C89] § 6.8.3.3
[ANSI Rationale] § 3.8.3.3
[H&S] § 3.3.9 p. 52

Q 10.21 어떤 코드에는 다음과 같은 매크로를 쓰는데:

```
#define CTRL(c)   ('c' & 037)
```

동작하지 않습니다. 왜 그럴까요?

Answer 위 매크로는 보통 다음과 같이 쓰기 위해 만들어졌습니다:

```
tchars.t_eofc = CTRL(D);
```

다음과 같이 확장될 것을 예상하고 만든 것입니다:

```
tchars.t_eofc = ('D' & 037);
```

결국, 이 매크로는 파라메터 c가 문자 상수에 쓰이는 따옴표 안에서도 확장될 것을 예상하고 만든 것인데, 프리프로세서는 이런 경우에 확장하지 않습니다; 이 코드가 제대로 동작했다면, 그 자체가 잘못된 것입니다. ANSI C는 이런 목적으로 쓸 수 있는, 문자열로 만들어주는(stringizing) 연산자를 제공합니다 (질문 11.17 참고). 그러나 문자로 만들어주는 (charizing) 연산자는 제공하지 않습니다.

이 매크로를 해결하는 가장 좋은 방법은 다음과 같이 매크로를 만들고:

```
#define CTRL(c)   (c & 037)
```

다음과 같이 호출하는 것입니다:

```
CTRL('D')
```

이렇게 하면, 매크로를 C 언어 형식으로 만들어주는 ("syntactic") 효과도 있습니다; 질문 10.2 참고.

Stringizing 연산자와 약간의 indirection을 써서 만들 수도 있습니다:

```
#define CTRL(c)    (*#c & 037)
```

또는

```
#define CTRL(c)    (#c[0] & 037)
```

위 두가지 방법 모두 완전하다고 볼 수 없습니다. 위 두 매크로 모두, case 레이블이나, 전역 변수를 초기화할 때 쓰일 수 없습니다. (전연 변수 초기화에 쓰이거나, case 레이블로 쓰이려면, 여러가지 형태의 상수 expression이 필요하며, 문자열이나 indirection이 허용되지 않습니다.)

덧붙여 질문 11.18도 참고하시기 바랍니다.

Note "Stringizing"이란 말 대신, "Stringification"이란 말을 쓰기도 합니다만, 아시다시피 둘 다 올바른 영어 단어는 아닙니다.

References [ANSI] § 3.8.3 footnote 87
[C89] § 6.8.3
[H&S] §§ 7.11.2, 7.11.3 pp. 226–7

Q 10.22 다음 매크로를 쓰면 "macro replacement within a string literal?" 이라는 경고가 발생합니다:

```
#define TRACE(n) printf("TRACE: %d\n", n)
```

제가 생각하기에는 `TRACE(count);`를 다음과 같이 확장하는 것 같습니다:

```
printf("TRACE: %d\count", count);
```

Answer 질문 11.18을 참고하기 바랍니다.

Q 10.23 매크로를 확장할 때, 문자열 안에까지 매크로 인자가 확장되게 할 수 있을까요?

Answer 질문 11.18을 보기 바랍니다.

Q 10.24 ANSI "stringing" 프리프로세싱 연산자인 '`#`'를 써서 매크로 값을 메시지에 넣을려고 했는데, 매크로의 값이 아닌, 매크로의 이름을 문자열로 만들어 버립니다. 왜 그럴까요?

Answer 질문 11.17과 11.18을 참고하기 바랍니다.

Q 10.25 복잡한 전처리를 해야 하는데, 방법을 모르겠습니다.

Answer C 언어의 전처리기(preprocessor)는 범용의(general purpose) 툴이 아닙니다. (또한 전처리기가 독립적인 프로그램으로 제공되는지도 알 수 없습니다.) 이를 복잡한 형식으로 사용하기에 앞서, 어떤 특정한 목적의 작은 프리프로세싱 툴을 직접 만드는 것이 나을 지 생각해 보기 바랍니다. `make`과 같은 툴을 사용하면, 이러한 프리프로세싱 툴을 자동으로 실행하도록 만들 수 있습니다.

C 언어 대신 다른 것을 'preprocessing' 하려고 한다면 여러가지 목적으로 쓸 수 있는 프리프로세서를 쓰기 바랍니다. (오래 전부터, 대부분의 UNIX 시스템에서는 '`m4`'라는 프리프로세서를 제공합니다.)

10.3 Macros with Variable-Length Argument Lists

There are sometimes good reasons for a function to accept a variable number of arguments (the canonical example is `printf`; see also Chapter 15). For the same sorts of reasons, it's sometimes wished that a function-like macro could accept a variable number of arguments; a particularly common wish is for a way to write a general-purpose `printf`-like `DEBUG()` macro.

Q 10.26 가변 인자를 받는 매크로를 만들 수 있습니까?

Answer 각각의 인자를 하나의 괄호로 둘러싸서 하나의 인자처럼 전달하는 것은 매우 인기있는 트릭입니다. 다음과 같이 할 수 있습니다:

```
#define DEBUG(args) (printf("DEBUG: "), printf args)

if (n != 0) DEBUG(("n is %d\n", n));
```

문제는 이런 매크로 함수를 호출할 때, 항상 괄호를 두 쌍을 써 줘야 한다는 것을 기억해야 합니다. 또 다른 문제는, 추가적으로 다른 인자를 줄 수 없다는 것입니다. (즉, `DEBUG()`는 `fprintf(debugfd, ...)`처럼 확장될 수 없습니다.)

GCC는 함수처럼 가변 인자를 받을 수 있는 확장 매크로를 지원하지만 이 기능은 표준이 아닙니다. 생각해 볼 수 있는 다른 방법으로, 다음과 같은 방법들이 있습니다:

- 인자의 갯수에 따라 다른 매크로를 (`DEBUG1`, `DEBUG2` 같이) 제공합니다.
- 콤마(comma)를 별도의 매크로를 써서 제공하는 방법이 있습니다:

  ```
  #define DEBUG(args) (printf("DEBUG: "), printf(args))
  #define _ ,

  DEBUG("i = %d" _ i)
  ```

- 위험하지만, 다음과 같이, 쌍이 맞지 않는 괄호를 쓸 수 있습니다:

  ```
  #define DEBUG       fprintf(stderr,

  DEBUG "%d", x);
  ```

여기에 나온 모든 방식이, 조심해서 써야 하며, 보기에 좋지 않는 방법입니다.

보통은 매크로가 아닌, 진짜 함수를 써서 가변 인자를 처리하는 것이 낫습니다. 질문 15.4, 15.5를 보기 바랍니다.

디버그 관련 메시지가 출력되지 않게 하려면, 따로 DEBUG 매크로의 다른 버전을 다음과 같이 만듭니다:

```
#define DEBUG(args)  /* empty */
```

또는, 진짜 함수를 쓴다면, 인자는 그대로 두고, 함수 이름만 없애버릴 수 있습니다:

```
#define DEBUG       (void)
```

또는,

```
#define DEBUG       if (1) {} else printf
```

또는,

```
#define DEBUG       1 ? 0 : (void)
```

이 모든 방법은, 좋은 optimizer가 있어서, 의미없는 "`printf`" 호출을 제거하고, void 타입으로 캐스팅 된, 컴마 연산자의 expression을 코드로 만들지 않는다는 것을 가정하고 만든 것입니다. 질문 10.14를 참고하기 바랍니다.

C99는 가변 인자를 처리할 수 있는 함수 매크로를 소개하고 있습니다. `...` 형식을 매크로의 끝에 사용하고 (`varargs` 함수처럼), 매크로 함수 안에서 `__VA_ARGS__` pseudo 매크로를 써서 가변 인자를 처리할 수 있습니다.

References [C89] § 6.8.3, § 6.8.3.1

Note C99 표준을 써서, 정말 좋은 형태의 `DEBUG` 매크로를 다음과 같이 만들 수 있습니다:

```
#define DEBUG(...) fprintf(debugfd, __VA_ARGS__)

DEBUG("age = %d, name = %s\n", age, namestr);
```

가변 인자를 받는 함수와 달리, 고정 인자가 하나 이상 필요하다는 규칙은 적용되지 않습니다.

References

Q 10.27 __FILE__과 __LINE__ 매크로를, 디버그용으로, 메시지 출력하는 매크로에 쓰고 싶습니다. 어떻게 하면 될까요?

Answer 질문 10.26을 간단히 하기 위해, 따로 만든 질문입니다. 한가지 방법은, 가변 인자를 받는 함수를 만들고 (질문 15.4, 15.5 참고), __FILE__과 __LINE__을 받아서 그 함수에 전달해 주는 함수를 따로 만듭니다. 예를 들면:

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

void debug(char *fmt, ...);
void dbginfo(int, char *);
#define DEBUG       dbginfo(__LINE__, __FILE__), debug

static char *dbgfile;
static int dbgline;

void dbginfo(int line, char *file)
{
  dbgfile = file;
  dbgline = line;
}

void debug(char *fmt, ...)
{
  va_list argp;
  fprintf(stderr, "DEBUG: \"%s\", line %d: ",
          dbgfile, dbgline);
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  fprintf(stderr, "\n");
}
```

이렇게 만들어 두고, 다음과 같이 부르면:

```
DEBUG("i is %d", i);
```

다음과 같이 확장됩니다:

```
dbginfo(__LINE__, __FILE__), debug("i is %d", i);
```

그리고 아래 문장을 출력합니다:

```
DEBUG: "x.c", line 10: i is 42
```

영리한 사람이면 다음과 같이 위 코드를 개선할 수 있습니다:

```
void debug(char *fmt, ...);
void (*dbginfo(int, char *))(char *, ...);
#define DEBUG  (*dbginfo(__LINE__, __FILE__))

void (*dbginfo(int line, char *file))(char *, ...)
{
  dbgfile = file;
  dbgline = line;
  return debug;
}
```

이 정의에 따라서 `DEBUG("i is %d", i);`는 다음과 같이 확장됩니다:

```
(*dbginfo(__LINE__, __FILE__))("i is %d", i);
```

또 다른, 그리고 간단한 방법으로 아래처럼 만들 수 있습니다:

```
#define DEBUG   printf("DEBUG: \"%s\", line %d: ", \
                       __FILE__, __LINE__), printf
```

그러면, `DEBUG("i is %d", i);`는 다음과 같이 확장됩니다:

```
printf("DEBUG: \"%s\", line %d: ",
       __FILE__, __LINE__), printf("i is %d", i);
```

Note 다행스럽게도, C99에서 가변 인자를 받을 수 있는 매크로를 지원하기 때문에 (질문 10.26 참고), `DEBUG()` 매크로를 다음과 같이 만들 수 있습니다.

```
#define DEBUG(...) debug(__FILE__, __LINE__, __VA_ARGS__)

void debug(const char *file, int line,
           const char *fmt, ...)
{
```

```
  va_list argp;
  fprintf(stderr, "DEBUG: \"%s\", line %d: ",
          file, line);
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  fprintf(stderr, "\n");
}
```

위 매크로에 따르면, `DEBUG("i is %d", i);`는 다음과 같이 확장됩니다:

```
debug(__FILE__, __LINE__, "i is %d", i);
```

C99에서는 미리 정의된(predefined) identifier인 `__func__`를 제공합니다. (이것은 `__FILE__`, `__LINE__`과는 달리 미리 정의된 매크로가 아닙니다. 자세한 것은 ???를 참고하기 바랍니다.) `__func__`는 현재 함수의 이름을 가지고 있는 문자열 상수라고 생각하시면 됩니다. 따라서 위 매크로와 함수를 다음과 같이 만들면 더욱 좋습니다:

```
#define DEBUG(...) debug(__FILE__, __LINE__, __func__, \
                         __VA_ARGS__)

void debug(const char *file, int line, const char *func,
           const char *fmt, ...)
{
  va_list argp;
  fprintf(stderr,
          "DEBUG: \"%s\", line %d, function \"%s\": ",
          file, line, func);
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  fprintf(stderr, "\n");
}
```

위 매크로에 따르면, `DEBUG("i is %d", i);`는 다음과 같이 확장됩니다:

```
debug(__FILE__, __LINE__, __func__, "i is %d", i);
```

References [C89] ???

Chapter 11

ANSI/ISO Standard C

The release of the ANSI C Standard (X3.159-1989) in 1990 (now superseded by ISO 9899:1990) and its ongoing revisions) marked a major step in C's acceptance as a stable language. The standard clarified many existing ambiguities in the language, but it introduced a few new features and definitions that are occasionally troublesome. Misunderstandings also arise when an ambiguity was resolved contrary to someone's experience or when people with pre-ANSI compilers try to use code written since the standard became widely adopted.

Standard C can be referred to in several ways. It was originally written by a committee (X3J11) under the auspices of the American National Standards Institute, so it's often called "ANSI C." The ANSI C Standard was adopted internationally by the International Organization for Standardization, so it's sometimes called "ISO C." ANSI eventually adopted the ISO version (superseding the original), so it's now often called "ANSI/ISO C." Unless you're making a distinction about the wording of the original ANSI standard before ISO's modifications, there's no important difference among these terms, and it's correct to simply refer to "the C Standard" or "standard C." (When the subject of C is implicit in the discussion, it's also common to use the word "standard" by itself, often capitalized.)

C 언어 표준은 항상 진행중입니다. 현재 C99까지 나왔으며, 앞으로도 다음 표준이 나올 수 있습니다. 여기에서는 [C99 Rationale]에 나온 C89 위원회의 원칙에 대해 몇가지 간단히 소개하겠습니다:

- Existing code is important, existing implementations are not. 현존하는 C 코드의 중요성을 인식하고, 가능하면 기존의 코드가 변화없이 동작할 수

있도록 합니다.

- C code can be portable. C 언어는 비록 PDP-11에서 동작하는 UNIX 시스템에서 태어났지만, 현존하는 모든 시스템에서 표준 C 언어를 (언어 및 라이브러리) 구현 가능하도록 합니다.
- C code can be non-portable. C 표준이 프로그래머가 이식성이 높은 프로그램을 작성할 수 있도록 해 주지만, 그렇다고 프로그래머가 고수준 어셈블러처럼 C 언어를 쓰는 것을 제한하지 않습니다. 시스템 의존적인 코드를 만들 수 있는 것도 C 언어의 장점 중 하나이기 때문입니다.
- Avoid "quite changes." 현존하는 코드의 의미를 바꾸는 변화는 문제가 될 수 있습니다. 적어도 이런 문제가 있다면 항상 쉽게 알 수 있는 방법을 제공해야 하며, 가능한 이런 은근슬쩍 의미를 바꾸지 않도록 할 것이며, 이런 변화가 있다면 표준과 Rationale에 "QUITE CHANGE"로 명시할 것입니다.
- A standard is a treaty between implementor and programmer. 시스템(컴파일러) vendor와 개발자들이 좀 더 자세히 이해할 수 있도록, 여러 수치 제한이 추가됩니다. 하지만, 이런 제한은 표준보다 우수하거나, 표준의 제한을 벗어나는 시스템을 개발하는 것을 막지 않습니다. 이런 제한들은 (*minimum maxima*)최고값이 제공할 수 있는 최하위 값의 형태로 제공됩니다.
- Keep the spirit of C. 기존 C 언어가 가지고 있는 틀을 그대로 유지합니다. 즉,
  - Trust the programmer.
  - Don't prevent the programmer from doing what needs to be done.
  - Keep the language small and simple.
  - Provide only one way to do an operation.
  - Make it fast, even if it is not guaranteed to be portable.
- Support international programming.
- Codify existing practice to address evident deficiencies.
- Minimize incompatibilities with C89.
- Minimize incompatibilities with C++.
- Maintain conceptual simplicity.

좀 더 자세한 것은 [C99 Rationale]를 참고하기 바랍니다.

11.1 The Standard

Q 11.1 "ANSI C 표준"이란 무엇인가요?

Answer 1983년, 미국 규격 협회(American National Standards Institute, ANSI)는 C 언어의 표준을 제정하기 위해 X3J11이라는 위원회를 열었습니다. 매우 긴 기간동안 토론한 끝에 이 위원회의 보고서는 1989년 12월 14일 ANSX3.159-1989라는 이름으로 비준받아서, 1990년에 출판되었습니다. 대부분의 내용은 기존의 C 언어에서 가져온 것이며, 몇몇은 C++에서 (대부분 함수 prototype에 대한 것) 가져온 것입니다. 그리고 (논쟁의 여지가 있던 3중 음자(trigraph) 문자 시퀀스를 포함한) 다국적 문자 세트를 지원하는 기능도 포함시켰습니다. ANSI C 표준은 C run-time 라이브러리도 표준화시켰습니다.

그 후에 국제 표준 기구인 ISO[1]는 미국 표준인 X3.159를 국제 표준인 ISO/IEC 9899:1990으로 바꿔서 국제 표준으로 만들었습니다. 이 표준에서는 ANSI의 표준을 정정하고 보충한 것이 대부분이었기 때문에 흔히 'ANSI/ISO 9899-1990' [1992] 라고 부릅니다.

1994년 'Technical Corrigendum 1(TC1)'은 표준에서 약 40 가지를 수정하였습니다. 대부분 수정은 부분적으로 명확한 설명이 필요한 것에 보충 설명을 단 것입니다. 그리고 'Normative Addendum 1(NA1)'은 약 50 페이지 분량의 새로운 내용을 추가했으며, 대부분이 국제화(internationalization[2])에 관한 함수 설명입니다. 1995년 TC2는 몇가지 정정 사항을 추가했습니다.

이 글을 쓸 때, 표준의 완전한 개정판은 이제 막바지 작업에 들어 갔습니다. 새로운 표준은 현재 "[C9X]"라고 이름이 붙었고, 1999년 말에 완성될 거라는 뜻을 나타냅니다. (이 글의 많은 부분도 새로운 [C9X]를 반영하려고 수정되었습니다.)

오리지널 ANSI 표준은 많은 부분에서 결정한 부분에 대한 설명과, 작은 문제들에 대한 논의를 포함한 "[ANSI Rationale] (이론적 해석)"을 포함하고 있습니다. 몇 가지는 이 글에 이미 포함되었습니다. ([ANSI Rationale] 자체는 ANSI 표준 X3.159-1989에 포함된 부분이 아니지만 정보 제공 목적으로만 제공되는 것이며, ISO 표준에 포함되는 내용도 아닙니다. [C9X] 용으로 새 판이 준비되고 있는 상황입니다.)

Note [H&S2002] 소개 부분에 나온 내용을 인용 및 요약하면 크게, C 언어는 네번의 변화를 거쳤습니다.

[1] International Organization for Standardization

[2] 이 단어가 너무 길기 때문에 보통 i18n으로 부르기도 합니다. 여기에서 18은 이 단어의 총 글자 수입니다.

첫째, Brian Kernighan씨와 Dennis Ritchie씨가 1978년에 쓴 *The C Programming Language*에서 소개한 C 언어입니다. 보통 "K&R"이라고 부르며, 1980년대에 쓰였던 모든 C 언어를 "traditional C"라고 합니다.

둘째, 표준을 정해 놓는 것이, C 언어가 널리 퍼지게 하는데 도움이 될 것이라는 믿음에서, 앞에서 소개한 ANSI 표준이 만들어졌습니다. (지금은 NCITS J11이 된) X3J11 위원회는 1989년에 C 언어 표준과 런타임 라이브러리를 "*American National Standard X3.159-1989*"로 공식화 했으며, 보통 "ANSI C"라고 부릅니다. 이는 곧 국제 표준으로 받아들여져, 국제 표준인 ISO/IEC 9899:1990으로 등록되었습니다. ANSI C와 이 국제 C 언어 표준과 차이점은 거의 없습니다. 그리고 이 국제 표준을 보통 "Standard C"라고 부릅니다. 그러나 이 표준은 또 변했기 때문에, 이 것을 "Standard C (1989)"로 부르거나, 줄여서 "C89"라고 합니다.

셋째, WG14 그룹은 C89에 대한 두 문서를 만들었는데 하나는 'Technical Corrigenda'이며 (버그 수정 문서), 또 하나는 'Amendment (확장)'입니다. 이들이 반영된 표준은, "C89 with Amendment 1" 또는 "C95"라고 부릅니다.

넷째, WG14는 계속 작업을 거듭했고, 그 추가 및 확장 사항들이 다시 반영되어, 1999년에 표준으로 제정되었습니다. 이는 "*ISO/IEC 9899:1999*" 또는, 줄여서 "C99"라고 부르며, 이 것이 가장 최신의 표준입니다.

참고로, Bjarne Stroustrup씨가 1980년 초반에 디자인한 C++은 현재 가장 인기있는 언어 중의 하나이며, 이 언어는 C 언어를 기초로 만들어 졌습니다. 이 C++ 언어도 역시 표준화가 (1998년) 이루어졌으며, 그 결과, *ISO/IEC 14882:1998* 또는 간단히 "Standard C++"이라고 부릅니다.

표준화가 이루어진 년도를 주의깊게 보았다면, C 언어 최신 표준이 C++ 언어의 표준이 제정된 다음에 개정된 것을 알 수 있습니다. 물론 단순히 이 사실 만으로 알 수 있는 것은 아니지만, 사실 표준 C++ 언어가, 표준 C 언어의 모든 점을 포함하고 있지 않습니다. 따라서 정확히 말해서, "C 언어는 C++ 언어의 부분집합이다" 또는 "C++ 언어는 C 언어를 포함한다"라는 말은 모두 틀린 말입니다. 물론 C++ 언어가 C 언어의 대부분을 포함하고 있지만, 전부는 아닙니다.

따라서, C 언어 표준을 완벽하게 지원하도록 만든 코드가 C++ 언어 표준에 맞지 않을 가능성이 있습니다. C 언어와 C++ 언어에서 완벽하게 동작하는 코드를 (어떤 사람들은) "Clean C"라고 부릅니다.

앞 답변에서 "C9X"라고 부르는 것은, 1990년대에 아직, 새 C 표준이 정해지지 않았을 때, (정확한 제정 연도를 몰랐기 때문에) 붙여진 이름입니다. 이제는 1999년에 제정된 것을 알기 때문에 "C99"라고 부르는 것이 올바른 표

현입니다.

References [H&S2002] § 1.1 pp. 4–5
[C99 Rationale] § 0

Q 11.2 표준 문서를 어디서 얻을 수 있죠?

Answer 미국이라면 다음 주소에서 사본을 신청할 수 있습니다:

```
American National Standards Institute
11 W.  42nd St., 13th floor
New York, NY  10036  USA
(+1) 212 642 4900
```

또는 다음 주소도 가능합니다:

```
Global Engineering Documents
15 Inverness Way E
Englewood, CO  80112  USA
(+1) 303 397 2715
(800) 854 7179  (U.S.  & Canada)
```

다른 나라들에서는 제네바(Geneva)에 있는 ISO에 주문하거나 각 국의 표준 위원회에 연락하시기 바랍니다:

```
ISO Sales
Case Postale 56
CH-1211 Geneve 20
Switzerland
```

(또는 `http://www.iso.ch`를 방문하거나, `comp.std.internat` FAQ 리스트에서 Standards.Faq를 참고하기 바랍니다).

저자가 마지막으로 검사했을 때, ANSI에서 주문하려면 130.00$가 필요했으며, GLobal에서 주문할 때에는 400.50$가 필요했습니다. ([ANSI Rationale]을 포함한) 오리지널 X3.159는 ANSI에서 205.00$, Global에서는 162.50$가 필요했습니다. ANSI에서는 표준 문서를 판매한 수익금으로 운영하기 때문에 전자 출판 형식으로는 제공해주지 않습니다.

미국이라면 ([ANSI Rationale]를 포함한) 오리지널 ANSI X3.159의 사본을 "FIPS PUIB 160"으로 다음 주소에서 주문할 수 있을 것입니다:

```
National Technical Information Service (NTIS)
U.S.  Department of Commerce
Springfield, VA  22161
703 487 4650
```

Herbert Schildt씨가 해설한(annotated) "Annotated ANSI C Standard"는 ANSI가 아닌 ISO 9899를 설명하고 있습니다; Osborne/McGraw-Hill에서 출판되었으며, ISBN 0-07-881952-0이며, 대략 $40 선에서 판매되고 있습니다. 표준과 이 책의 가격 차이는 대부분 저자가 단 해설(annotation) 가격입니다: 그러나 많은 에러와 너무 많은 생략으로 평판이 좋지 않습니다. net의 대부분 사람들은 아예 이 책을 무시합니다. Clive Feather씨는 이 책에 대한 서평을 썼고 아래의 URL에서 볼 수 있습니다:

```
http://www.lysator.liu.se/c/schildt.html
```

[ANSI Rationale] 문서는 (완전한 표준은 아님) anonymous FTP로 `ftp.uu.net`의 (질문 18.16 참고) `doc/standards/ansi/X3.159-1989` 디렉토리에서 얻을 수 있습니다. 또 `http://www.lysator.liu.se/c/rat/title.html`에서 볼 수도 있습니다. Silicon Press에서 출판되기도 했습니다. ISBN 0-929306-07-4입니다.

ISO/IEC C9X의 진행판은 JTC1/SC22/WG14 사이트인 아래에서 얻을 수 있습니다:

```
http://www.dkuung.dk/JTC1/SC22/WG14/
```

질문 11.2b를 참고하기 바랍니다.

Note 최신 C 표준인 C99의 PDF 버전 (ISO web site에서) 공식 가격은 (2005년 1월 24일자로) 340 CHF입니다. 이 날짜로, 원화 가격은 약 30만원입니다. ANSI를 비롯, 다른 web site에서도 판매하고 있습니다. 구입할 때, 무료로 배포된는 Corrigenda (버그 수정) 문서도 꼭 함께 받으시기 바랍니다. 현재 다음 두 가지가 나와 있습니다:

- ISO/IEC 9899:1999/Cor 1:2001
- ISO/IEC 9899:1999/Cor 2:2004

검색 엔진을 쓸 때, "9899:1999"로 검색하시면 빨리 찾을 수 있습니다.

Q 11.2b 개정된 표준에 대한 정보는 어디에서 얻을 수 있나요?

Answer ([C9X] 진행판을 포함한) 관련된 정보는 다음 웹 사이트에서 찾을 수 있습니다:

```
http://www.lysator.liu.se/c/index.html
http://www.dkuug.dk/JTC1/SC22/WG14/
http://www.dmk.com/
```

Note `http://www.dkuug.dk/JTC1/SC22/WG14/`

위 사이트의 새 주소는 다음과 같습니다:

```
http://www.open-std.org/jtc1/sc22/wg14/
```

11.2 Function Prototypes

The most significant introduction in ANSI C is the *function prototype* (borrowed from C++), which allows a function's argument types to be declared. To preserve backward compatibility, nonprototype declarations are still acceptable, which makes the rules for prototypes somewhat more complicated.

Q 11.3 제 ANSI 컴파일러는 다음과 같은 코드를 봤을 때, 함수가 일치하지 않는다고 에러를 발생합니다:

```
extern int func(float);

int func(x)
float x;
{ ...
```

Answer 그 이유는 새 (스타일) 프로토타입(prototype) 선언인 "`extern int func(float)`"을 오래된 (스타일) 정의인 "`int func(x) float x;`"와 섞어 썼기 때문에 발생합니다. 보통 이 두 스타일을 섞어 쓰는 것이 가능하지만(질문 11.4 참고) 이 경우에는 안됩니다.

Traditional C (프로토 타입과 가변 인자 리스트를 제공하지 않는 ANSI C; 질문 15.2 참고) 언어에서는 함수에 전달되는 어떤 인자들을 "확장(widen)" 시킵니다. 즉 `float`은 `double`로, `char`나 `short int`는 `int`로 확장시킵니다. (구 스타일로 정의한 함수에서는, 이렇게 확장되어 전달된 인자가, 함수의

몸체 부분에 들어갈 때, 다시 원래의 크기로 변환됩니다) 따라서 위의 오래된 스타일로 만든 정의는 사실상 `func` 함수를 `double` 타입 인자를 받도록 만든 것입니다. (물론 함수 내부에서 이 인자는 다시 `float` 타입으로 바뀝니다)

이 문제는 함수 정의에서 새 스타일의 문법을 써서 고칠 수 있습니다:

```
int func(float x) { ...  }
```

또는 새 형식의 프로토타입 선언을 구 형식의 정의와 일치하도록 다음과 같이 만들어 주면 됩니다:

```
extern int func(double);
```

(이 경우, 가능하다면 구 형식의 정의에서 `double`을 쓰도록 바꿔주는 것이 더 깨끗합니다.[3])

"narrow" 효과를 피하기 위해, "narrow" 효과를 발생하는 타입들을 (예를 들어 `char`, `short int`, `float` 등) 함수 인자나 리턴 타입으로 쓰지 않는 편이 안전할 수 있습니다.

질문 1.25를 참고하기 바랍니다.

References [K&R1] § A7.1 p. 186
[K&R2] § A7.3.2 p. 202
[C89] § 6.3.2.2, § 6.5.4.3
[ANSI Rationale] § 3.3.2.2, § 3.5.4.3
[H&S] § 9.2 pp. 265–7, § 9.4 pp. 272–3

Note 함수 호출시, prototype이 확인되지 않다면, 컴파일러는 모든 인자에 대해 default argument promotion을 실시합니다. 즉, `int`보다 작은 정수 타입들은 모두 `int`로, `float`은 `dobule`로 변환하며, 나머지 타입들은 그대로 전달됩니다. 만약, 인자(argument)가 파라메터와 갯수가 다를 경우, undefined behavior가 발생하며, 만약 나중에 함수 prototype이 발견되었는데, 이 함수 prototype이 가변 인자를 받는 것으로 선언되어 있거나, 파라메터들의 타입이, default argument promotion 후의 인자 타입과 서로 다를때에도 undefined behavior가 발생합니다. (마지막 문장은 함수 인자/파라메터 뿐만 아니라, 함수의 리턴 타입에도 적용됩니다.)

이 규칙은 생각보다 좀 더 까다롭습니다. 아래 Reference를 꼭 읽어 보기 바랍니다.

[3] Changing a parameter's type may require additional changes if the address of that parameter is taken and must have aparticular type

[C99] § 6.5.2.2, § 6.7.5.3
[H&S2002] § 9.2

Q 11.4 함수 구문에서 오래된 형식과 새 형식을 섞어 쓸 수 있나요?

Answer 섞어 쓸 수 있긴 하지만 매우 주의해야 합니다. (질문 11.3을 꼭 보시기 바랍니다). 현재에는 prototype 형식이 선언과 정의에 모두 사용되고 있습니다. (오래된 형식은 쓸모 없이 되어가고 있기 때문에, 언젠가 공식적으로 제거될 것입니다.)

References [ANSI] § 3.7.1, § 3.9.5
[C89] § 6.7.1, § 6.9.5
[H&S] § 9.2.2 pp. 265–7, § 9.2.5 pp. 269–70

[C99] § 6.5.2.2 § 6.9.1

Note 일반적으로, 오래된 형식과 섞어 쓰기 위해서는 다음 두 가지 규칙을 지켜야 합니다:

- 함수의 리턴 타입과 모든 파라메터 타입은 default argument promotion 후의 타입과 일치하도록 한다. 즉, `int`보다 작은 정수 타입이나 `float`을 쓰지 않는다.
- 가변 인자를 받는 함수는 오래된 형식으로 만들지 않는다.

References [H&S2002] § 9.2.2, pp. 291–292 § 9.2.5 pp. 294–295
GNU Coding Standard § 3.4

Q 11.5 컴파일러가 다음 선언을 만나면:

```
extern int f(struct x *p);
```

"struct x introduced in prototype scope"라는 이상한 경고를 발생시킵니다.

Answer C 언어의 블럭 스코프(scope) 규칙에 따르면 함수의 프로토타입에만 선언된 structure는 같은 소스의 다른 구조체와 호환성이 없습니다 이 structure와 tag는 함수의 프로토타입 선언이 끝날 때 스코프를 벗어납니다; 질문 1.29를 참고하기 바랍니다.

함수 prototype 앞에 structure 선언을 두어 이 문제를 해결할 수 있습니다. (보통, prototype과 structure 선언은 같은 헤더 파일에 존재하며, 이렇기 때문에 한쪽이 다른 한쪽을 참조할 수 있습니다.) 만약 prototype에 아직 선언되지 않은 structure를 꼭 쓸 필요가 있다면, prototype 앞에 다음과 같이 써줍니다:

```
struct x;
```

아무것도 아닌 것 같은 이 선언은 `struct x` 이렇게 하면, 이 구조체의 (incomplete) 선언이 파일 스코프를 가지게 되어, 이후에 나올 선언에서 `struct x`를 사용할 때, 같은 `struct x`를 가리키도록 할 수 있습니다.

References [ANSI] § 3.1.2.1, § 3.1.2.6, § 3.5.2.3
[C89] § 6.1.2.1, § 6.1.2.6, § 6.5.2.3.

Q 11.6 다음 코드가 이상하게 동작합니다:

```
printf("%d", n);
```

위에서 `n`은 `long int` 타입입니다. ANSI 함수 prototype이 이런식으로 인자와 파라메터 타입이 서로 일치하지 않을때, (conversion 등으로) 보호해 주지 않나요?

Answer 질문 15.3을 보기 바랍니다.

Q 11.7 `printf`를 쓰기 전에 `<stdio.h>`를 include해야 한다고 들었습니다. 왜 그런가요?

Answer 질문 15.1을 참고하기 바랍니다.

11.3 The `const` Qualifier

Another introduction from C++ is an additional dimension to the type system: type qualifiers. Type qualifiers can modify pointer types in several ways (affecting either the pointer or the object pointed to), so qualified pointer declarations can be tricky. (The questions in this section refer to `const`, but most of the issues apply to the other qualifiers, `volatile`, as well.)

Q 11.8 배열의 크기를 지정할 때, 다음과 같이 상수 값을 쓰면 안되는 이유가 뭔가요?

```
const int n = 5;
int a[n];
```

Answer `const` qualifier는 "읽기 전용"인 것을 의미합니다; 즉 지정한 오브젝트는 실행할 때 (일반적으로) 변경할 수 없는 run-time 오브젝트입니다. 따라서 이러한 'const' 오브젝트는 상수 수식(constant expression)이 아니기 때문에, 배열의 크기 지정, case label등에 쓰일 수 없습니다. (이 부분에서 C 언어와 C++이 다릅니다.) 정말로 compile-time 상수가 필요하다면 `#define`으로 (또는 `enum`으로) 상수를 정의하기 바랍니다.

References [ANSI] § 3.4
[C89] § 6.4
[H&S] § 7.11.2,7.11.3 pp. 226–7

Q 11.9 '`const char *p`'와 '`char * const p`'의 차이는 무엇인가요?

Answer '`const char *p`'는 ('`char const *p`'라고 쓸 수 있음) 상수 문자에 대한 포인터를 선언한 것입니다 (가리키는 문자를 바꿀 수 없는 포인터); '`char * const p`'는 문자에 대한 상수 포인터를 선언한 것입니다 (문자를 변경할 수는 있지만 포인터를 변경할 수는 없습니다).

해설을 잘 음미해보시기 바랍니다; 질문 1.21도 참고하시기 바랍니다.

References [ANSI] § 3.5.4.1 examples
[C89] § 6.5.4.1
[ANSI Rationale] § 3.5.4.1
[H&S] § 4.4.4 p. 81

Note 아래 예에서, `const_pointer`는 포인터가 상수인 것을 나타냅니다. 그리고, `pointer_to_const`는 상수를 가리키는 포인터를 나타냅니다:

```
int * const const_pointer;
const int *pointer_to_const;
```

즉, `const_pointer`는 포인터가 가리키는 대상을 변경할 수 있으나, 다른 대상을 가리키도록 할 수는 없습니다. 그리고, `pointer_to_const`는 다른 대상을 가리키도록 할 수 있지만, 포인터가 가리키는 대상을 변경할 수 없습니다.

References [H&S2002] § 4.4.4 pp. 89–91

Q 11.10 `const char **`를 인자로 받는 함수에 `char **`를 전달하면 안되나요?

Answer (어떤 T 타입에 대해) const T가 와야 하는 곳에, T를 가리키는 포인터를 쓸 수 있습니다. 이 규칙은 포인터 타입이 qualified 부분이 서로 약간 다를 경우에도 쓸 수 있다는 것을 의미하며, 여기에는 한 가지 예외 사항이 있는데, 이 규칙은 계속 재귀적으로(recursively) 적용되지 않고, 단지 top-level에만 적용 된다입니다. (즉, `const char **`는 const char에 대한 포인터를 가리키는 포인터이므로, 이 예외 사항에 해당되지 않습니다.)

`const char **`가 필요한 곳에 `char **`를 쓸 수 없는 이유는 조금 불명확합니다. `const`가 붙어 있기 때문에, 컴파일러는 여러분이 `const` 값을 변경하지 않는다고 한 약속을 지킬 수 있도록 도와주려 합니다. 그렇기 때문에 `const char *`가 필요한 곳에 `char *` 타입을 쓸 수 있으며, 반대 경우에는 쓸 수 없습니다: 간단한 포인터 타입에 `const`를 붙이는 것은, 프로그램이 매우 안전하게 동작할 수 있도록 도와줍니다. 그러나, 반대로 `const`를 제거하는 것은 때때로 위험할 수 있습니다. 아래처럼 조금 복잡한 대입 연산들을 생각해보기 바랍니다:

```
const char c = 'x'; /* 1 */
char *p1;               /* 2 */
const char **p2 = &p1;  /* 3 */
*p2 = &c;               /* 4 */
*p1 = 'X';              /* 5 */
```

세번째 줄에서 우리는 `const char **`가 필요한 곳에, `char **`를 대입했습니다. (컴파일러는 이 부분에서 경고를 발생시킵니다.) 네번째 줄에서, 우리는 `const char *`를 `const char *`가 필요한 곳에 대입했습니다; 이 것은 아무런 문제가 없습니다. 다섯번째 줄에서 우리는 `char *` 포인터가 가리키는 것을 변경시켰습니다. 이 것은 아무런 문제가 없어보이지만, `p1`은 사실 `c`를 가리키고 있고, 이 것은 `const`이기 때문에 문제가 됩니다. 다시 잘 분석하면, 이 것은 네번째 줄에서 `*p2`가 실제로 `p1`을 가리키고 있었던 것이 문제입니다. `*p2`가 `p1`을 가리키도록 만든 것은 바로 세번째 줄에서 했던 것이고, 이 것은 허용되지 않는 대입 연산이며, 이런 이유 때문에, 허용되지 않습니다.[4]

[4] C++ 언어는 `const`가 붙은 포인터에 대해 좀 더 복잡한 규칙을 쓰며, 이 규칙은 좀 더 다양한 꼴의 대입을 허용하지만, 이와 같이 `const`가 붙은 오브젝트를 변경하는 것은 막아 줍니다. C++에서도 `const char **`가 올 곳에 `char **`를 대입하는 것은 허용되지 않지만, `const char * const *`가 올 곳에 `char **`를 쓰는 것은 허용됩니다.

(앞 예제 세번째 줄에서처럼) `const char **`에 `char **`를 대입하는 것은 그 자체가 위험한 것은 아닙니다. 그러나 위에서 `p2`가 약속하고 있는 것, 즉 궁극적으로 가리키고 있는 값을 변경하지 않는다를 깨뜨릴 수 있는 실마리를 제공합니다. 그래도 이러한 연산이 필요하다면, 다시 말해, 가장 최상위 수준이 아닌 곳에서 qualifier가 서로 달라서 대입이 되지 않는 것을 대입시키려면, 직접 캐스트해서 (즉, 이 경우에는 `(const char) **`로) 쓸 수 있습니다.

References [ANSI] § 3.1.2.6, § 3.3.16.1, § 3.5.3
[C89] § 6.1.2.6, § 6.3.16.1, § 6.5.3
[H&S] § 7.9.1 pp. 221–2

Q 11.11 다음과 같은 선언에서:

```
typedef char *charp;
const charp p;
```

왜 `p`가 가리키는 `char`가 `const`가 되지 않고, `p` 자체가 `const`가 되는 것일까요?

Answer Typedef로 치환한 것은 순수하게 textual 치환이 아닙니다. (이 것은 typedef를 쓰는 한가지 장점이기도 합니다; 질문 1.13 참고) 다음과 같은 선언에서:

```
const charp p;
```

`const int i`가 `i`를 `const`로 만드는 것과 같은 원리에서, `p`는 `const`가 됩니다. `p`에 대한 선언은, 포인터가 관련이 되어있는지 typedef 안까지 쫓아가서 확인하지 않습니다.

References [H&S] § 4.4.4 pp. 81–2

Note 아래와 같은 선언이 있다고 가정하고 (질문 11.9 참고):

```
int * const const_pointer;
```

위에서 `const_pointer`는 `typedef`를 써서 다음과 같이 쓸 수 있습니다:

```
typedef int *int_pointer;
const int_pointer const_pointer;
```

이 때, `const_pointer`는 상수 `int`를 가리키는 포인터처럼 보이지만, 실제로는 (상수가 아닌) `int`를 가리키는 const 포인터입니다. 또, 타입 specifier와 타입 qualifier의 순서는 중요하지 않기 때문에 (순서가 바뀔 수 있기 때문에), 다음과 같이 쓸 수도 있습니다:

```
int_pointer const const_pointer;
```

References [H&S2002] § 4.4.4 p. 90

11.4 Using `main()`

Although every C program must by definition supply a function named `main`, the declaration of `main` is unique because it has two acceptable argument lists, and the rest of the declaration (in particular, the return type) is dictated by a factor outside of the program's control, namely, the startup code that will actually call `main`.

Q 11.12a `main()`의 정확한 선언 방법을 알고 싶습니다.

Answer `main()`의 선언은 다음 중에서 골라 써야 합니다:

```
int main(void);
int main(int argc, char *argv[]);
```

`argv`를, `char **argv`로 선언할 수도 있습니다. (질문 6.4를 참고하기 바랍니다.) (물론 이때 '`argv`'와 '`argc`'라는 이름은 얼마든지 바꿀 수 있습니다.) 또 오래된 스타일을 써서 다음과 같이 할 수도 있습니다:

```
int main()


int main(argc, argv)
int argc;
char **argv;
```

질문 11.12b부터 11.15까지 참고하기 바랍니다.

References [C89] § 5.1.2.2.1, § G.5.1
[H&S] § 20.1 p. 416
[CT&P] § 3.10 pp. 50–51

[C99] § 5.1.2.2.1, § J.1.1, § J.2.1, § J.3.2.1, § J.5.1
[H&S2002] § 9.9, § 9.11.4, § 16.5

Note 원래 C 언어에서, 함수의 정의나 변수 선언에서 타입을 (type specifier) 생략했을 때, 디폴트로 `int` 타입이 됩니다. 그러나 이런 식으로 생략하는 것은 그 동안 나쁜 프로그래밍 스타일로 여겨졌으며, 현재 C99 표준에서는 에러입니다.

[H&S2002] § 4.4.1

Q 11.12b "main returns no value"라는 경고를 피하기 위해, `main()`을 `void` 타입으로 선언해도 괜찮을까요?

Answer 안됩니다. `main()`은 반드시 `int` 타입을 리턴하도록 선언되어야 하며, 인자는 0개 또는 2개이어야 합니다. 종료할 때, `exit()`를 썼는데도 계속 이러한 경고가 발생한다면, 마지막에 쓸데없는 여분의 (redundant) 리턴 문장을 써 주어 경고가 발생하는 것을 막을 수 있습니다. (가능하다면 컴파일러가 제공하는 "not reached"를 의미하는 지시어(directive)를 써 주어도 됩니다).

단순히 경고를 없애려고 함수를 `void` 타입으로 선언하는 것은 매우 좋지 않습니다; 왜냐하면, 내부적인 함수 호출/리턴 시컨스가 함수를 호출하는 쪽과 (`main()`의 경우, C run-time startup code) 서로 다를 수 있기 때문입니다.

(Note that this discussion of main() pertains only to "hosted" implementations; none of it applies to "freestanding" implementations, which may not even have main(). However, freestanding implementations are comparatively rare, and if you're using one, you probably know it. If you've never heard of the distinction, you're probably using a hosted implementation, and the above rules apply.)

References [C89] § 5.1.2.2.1, § G.5.1
[H&S] § 20.1 p. 416
[CT&P] § 3.10 pp. 50–51

Q 11.13 `main`의 세 번째 인자인 `envp`가 있다고 들었습니다.

Answer 세 번째 인자는 자주 쓰이기는 하지만 표준 인자는 아닙니다. 이는 환경(environment) 변수에 접근하기 위한 것으로, 이 목적으로 쓰이는 표준 함수 `getenv()`가 제공되고, 전역 변수로 `environ`이 제공되기 때문에, 이 비표준 세번째 인자를 쓸 이유가 없습니다 (하지만, 전역 변수 `environ`도 비표준이기는 마찬가지입니다).

References [C89] § G.5.1
[H&S] § 20.1 pp. 416–7

Note 전역 변수 environ은 POSIX.1 표준입니다. (물론 POSIX.1은 C 표준과 다르기 때문에, C 언어 관점에서는 비표준이라 할 수 있습니다.)

```
#include <unistd.h>

extern char **environ;
```

원래, 전통적으로 UNIX 시스템에서는 main이 세번째 인자를 받을 수 있게 선언할 수 있었습니다. 즉 다음과 같습니다:

```
int main(int argc, char *argv[], char *envp[]);
```

그러나, 이 것은 (C 표준이 아닌 것은 물론) POSIX.1 표준도 아닙니다. POSIX.1을 따르면, ISO C 표준을 존중하기 위해, main의 세번째 인자인 envp와 같은 방식으로 동작하는, 전역 변수 environ을 써야 한다고 씌여 있습니다.

한 환경 변수의 값을 얻기 위해서는, 표준 함수인 getenv를 쓰는 것을 권장합니다. 하지만, 모든 환경 변수의 이름과 값을 얻기 위한 표준 방법은 존재하지 않습니다. 모든 환경 변수의 이름과 값을 꼭 얻어야 겠다면, (C 표준은 아니지만) POSIX.1 표준인 전역 변수 environ을 쓸 것을 권장합니다.

질문 19.33을 참고하기 바랍니다.

References [H&S2002] § 9.9 p. 304
[SUS] environ

Q 11.14 main()을 void 타입으로 선언한다 하더라도, exit() 함수를 써서 종료한다면 문제될 게 전혀 없지 않나요? 게다가 제가 쓰고 있는 운영 체제는 프로그램의 종료/리턴 코드를 아예 무시한답니다.

Answer main()의 리턴 값이 쓰이냐, 쓰이지 않느냐는 중요한 문제가 아닙니다; 문제는 main()을 void 타입으로 선언함으로 인하여, main()을 호출하는 부분이 (런-타임 시작(startup) 코드) main()을 제대로 호출하지 못할 수 있다는 것입니다 (이는 calling convension 문제입니다; 질문 11.12b를 참고하기 바랍니다).

Borland C++ 4.5에서 void main()을 썼을 때, 프로그램이 망가질 수 있다는 것이 이미 보고되었습니다. 그리고 어떤 컴파일러들은 (DEC C V4.1과 gcc) main을 void 타입으로 선언했을 때, 경고를 발생합니다.

여러분의 운영 체제가 종료 상태(exit status)를 무시할 수도 있고, `void main()`이 동작할 수도 있지만, 이는 이식성이 없을 뿐만 아니라, 올바른 것도 아닙니다.

여러 시스템에서 `void main()`으로 써도 동작한다는 것은 사실입니다. 만약 이식성을 전혀 고려할 생각이 없고, 이 방식이 더 편하다고 생각하면, 아무도 말릴 사람은 없습니다.

Note C 표준은, 질문 11.12a에 나온 꼴의 `main` 함수를 지원할 것을, 컴파일러에게 요구합니다. 또한 컴파일러는 C 표준에 나와 있지 않는 형태의 `main`을 제공하는 것에는 관여하지 않습니다. 그러나, 다른 비표준 형태를 쓰는 것은 (표준 관점에서 봤을 때) 'undefined behavior'를 낳습니다.

Q 11.15 제가 보고 있는 책 "C Programming for the Compleat Idiot"에서는 항상 `void main()`을 사용합니다.

Answer 아마도 그 책의 저자는 자신도 그 범주(complete idiot)에 계산한 모양입니다. 많은 책들이 `void main()`을 쓰고 있지만 이는 잘못된 것입니다.

Q 11.16 `main()`에서 어떤 값을 리턴하는 것과 `exit()`를 쓰는 것과 완전히 같나요?

Answer 그렇다고, 또는 그렇지 않다고 할 수 있습니다. C 표준에서는 완전히 같다고 언급하고 있지만, 어떤 오래된 시스템에서는 (특히 C 표준을 준수하지 않는), (리턴하는 것과 `exit`를 쓰는 것 중) 어느 한 쪽이 정상적으로 동작하지 않을 수 있습니다.

`main`에 local 데이터가 'cleanup' 과정에서 필요할 경우, `main`에서 리턴하는 방법은 제대로 동작하지 않을 수 있습니다; 질문 16.4를 참고하기 바랍니다. 그리고 아주 오래된 (표준을 지원하지 않는) 몇몇 시스템에서는 두 가지 형식 중 하나가 제대로 동작하지 않을 수 있습니다.

(마지막으로, 이 두가지 형태는 `main()`을 재귀적으로 호출할 경우, 다른 코드를 생성합니다.)

References [K&R2] § 7.6 pp. 163–4
[C89] § 5.1.2.2.3

Note C++ 언어와는 달리, C 언어 표준에서는 `main()`이 재귀적으로 (즉, recursive하게, `main`이 다시 `main`을 부르는 경우) 호출되는 것을 막지 않았습니다.[5] 즉, 원한다면 `main`에서 다시 `main`을 부를 수 있습니다. 그러나, IOCCC

[5] C++ 언어에서는 `main`을 재귀적으로 부를 수 없으며, `main`의 주소를 얻는 것도 금지되어 있습니다.

에 출품할 것이 아니라면, 그런 코드를 만들 이유가 없습니다. (IOCCC에 대한 것은 질문 20.36을 참고하기 바랍니다.)

C99 표준에 따르면, (표준에 부합하는 형태, 즉 `int`를 리턴하는) `main`에서 어떤 값을 `return` 하는 것은, `exit` 함수를 그 값으로 부르는 것과 같다고 씌여 있습니다. 그리고 이 효과가 일어나는 것은, 프로그램 시작점으로 쓰인 `main`에서만 일어납니다. 즉, 재귀적으로 불려진 `main`에서 `return`한다고 `exit`가 자동으로 호출되지 않습니다.

또한 `exit`나 `return` 문장 없이 `main`이 끝나버리면, `return 0`을 쓴 것과 같은 효과를 얻을 수 있다고 씌여 있습니다. (이 것은 C99에 새로 추가된 내용입니다. 그 전 표준인 ANSI, C89 등에서는 해당되지 않습니다.)

`return`하는 것과 `exit`를 부르는 것에 미세한 차이가 있을 수 있는데, 만약 `atexit`로 exit handler를 등록시켜 놓았고, 그 handler가 `main`에서 만든 어떤 automatic (static이 아닌) 변수에 접근한다면, `return`하는 것은 이 변수에 접근할 수 없습니다. 즉, `return`을 써서 자동으로 `exit`를 부르게 하는 것은, 이미 `main` 함수의 블럭을 벗어났기 때문에, `main`에서 선언한 automatic 변수는 존재하지 않습니다.

References [C99] § 5.1.2.2.3, § 7.19.3.5

11.5 Preprocessor Features

ANSI C introduces a few new features into the C preprocessor, including the "stringizing" and "token pasting" operators and the `#pragma` directive.

Q 11.17 ANSI "stringizing" 전처리기 연산자인 `#`를 써서 심볼릭 상수의 값을 문자열에 집어 넣으려고 합니다, 그런데, 그 결과, 상수의 값이 들어가는 대신, 상수의 이름이 들어가는군요.

Answer `#`의 정의에 따르면, 이 것은 매크로 인자를 (인자가 또 다른 매크로 이름이더라도 더 이상 확장하지 않고) 바로 문자열로 만듭니다. 매크로가 원래 지닌 뜻으로 확장되길 원한다면 다음과 갈이 두 단계를 거쳐서 쓸 수 있습니다:

```
#define Str(x) #x
#define Xstr(x) Str(x)
#define OP plus
char *opname = Xstr(OP);
```

이 코드는 opname을 "OP"로 설정하지 않고, "plus"로 설정합니다. (즉, Xstr() 매크로가 인자를 확장하고 Str() 매크로가 문자열로 만듭니다.)

비슷한 상황이 "token-pasting" 연산자인 ##를 쓸 때, 두 매크로의 값을 연결하려 할 때 발생할 수 있습니다.

#나 ##는 일반 소스 코드에서는 쓰일 수 없으며, 다만 매크로 정의 부분에서만 쓸 수 있는 연산인 것을 꼭 기억하기 바랍니다.

References [ANSI] § 3.8.3.2, § 3.8.3.5 example
[C89] § 6.8.3.2, § 6.8.3.5.

Q 11.18 메시지 "warning: macro replacement within a string literal"은 무슨 뜻이죠?

Answer : ANSI 이전의 어떤 컴파일러/전처리기는 매크로 정의를 다음과 같이 정의할 경우:

```
#define TRACE(var, fmt) printf("TRACE: var = fmt\n", var)
```

다음과 같은 식으로 호출하게 되면:

```
TRACE(i, %d);
```

다음과 같이 확장하게 됩니다:

```
printf("TRACE: i = %d\n", i);
```

즉, 매크로 인자로 나온 이름이 문자열 안에 있는 경우라도 확장시켜 버립니다. (물론 이러한 버그가 위와 같이 유용하게 쓰일 수도 있지만, 이 것은 대개 초창기 컴파일러를 만들 때 잘 못 만든 것입니다.)

이러한 식의 매크로 확장은 K&R에 언급된 것도 아니며, 표준 C 언어에서 언급된 것도 아닙니다. (매우 위험한 방법이며, 코드가 어려워집니다. 질문 10.22를 참고 바랍니다.) 매크로 인자 자체가 문자열이 되기를 원한다면 전처리기 연산자인 #를 쓰거나, 문자열 연결 (concatenation) 기능을 쓰면 됩니다 (이는 ANSI 표준의 새로운 기능입니다.):

```
#define TRACE(var, fmt) \
  printf("TRACE: " #var " = " #fmt "\n", var)
```

질문 11.17을 참고하기 바랍니다.

References [H&S] § 3.3.8 p. 51

Q 11.19 `#ifdef`를 써서 컴파일하지 말라고 한 곳에서 매우 이상한 구문(syntax) 에러가 납니다.

Answer ANSI C에서, `#if`, `#ifdef`, `#ifndef`에 쓴 텍스트는 전처리기가 처리할 수 있는 유효한 것이어야 (valid preprocssing token) 합니다. 즉, C 언어에서처럼 "나 '는 각각이 쌍을 이루어서 나와야 하며, 이처럼 둘러싼 문자열의 안에 newline 문자가 나와서는 안되며, 주석이 끝나지 않고 열려 있어서도 안됩니다. 서로 다른 따옴표가 엇갈려서 있어도 안됩니다. (특히, 영어의 생략형(contracted word)에 쓰이는, 역 따옴표(apostophe, ‘)는 문자 상수의 시작처럼 보일 수 있다는 것에 주의하기 바랍니다.) 따라서 긴 주석이나 pseudo code를 쓰는 것이 목적이라면 `#ifdef`를 써서 빼라고 지정을 했더라도, 공식적(offical)인 주석(comment)인 `/* ... */`을 써야 합니다.

References [C89] § 5.1.1.2, § 6.1
[H&S] § 3.2 p. 40

Note C99 표준에 따라서, C++ 언어에서 쓰는 것처럼 `//`로 시작하는 주석도 쓸 수 있습니다.

Q 11.20 `#pragma`는 어디에 쓰나요?

Answer `#pragma`는 모든 종류의 (이식성이 떨어지는) 모든 구현 방법에 따른 기능을 제어하고, 확장 기능을 제공합니다; 여기에는 소스 리스팅 제어, 구조체 압축(packing), 그리고 경고 출력 수준(`lint`의 오래된 주석 형태인 `/* NOTREACHED */`와 같이) 등이 포함됩니다.

References [ANSI] § 3.8.6
[C89] § 6.8.6
[H&S] § 3.7 p. 61

Note 예전에는 `#pragma` 뒤에 나오는 것들(간단히 pragma라고 부르기도 함)은 모두 시스템에 의존적인 사항이었으나, C99에서는 몇가지 표준 pragma를 만들었습니다. 표준 pragma는 `#pragma` 바로 다음에 `STD`가 나오며, 그 뒤에 나오는 정보는 매크로 확장이 되지 않습니다.

```
#pragma STD FENV_ACCESS       ON
#pragma STD FP_CONTRACT       ON
#pragma STD CX_LIMITEED_RANGE ON
```

표준 C 언어에서 공식적으로 제공하는 pragma는 위 세 가지이며, 위에서 `ON` 대신에 `OFF`나 `DEFAULT`를 쓸 수 있습니다.

References [H&S2002] § 3.7 pp. 67–69

Q 11.21 "`#pragma once`"가 의미하는 것이 뭐죠?

Answer 이는 어떤 전처리기들이 제공하는 기능으로 헤더 파일이 단 한번씩만 포함되도록 하는, 질문 10.7에 소개된 `#ifndef` 트릭과 같은 역할을 합니다. 단 이식성이 떨어집니다.

11.6 Other ANSI C Issues

Q 11.22 `char a[3] = "abc";`가 올바른 표현인가요?

Answer ANSI C (그리고 ANSI 이전의 몇몇 시스템에서)는 이러한 것을 올바른 표현이라고 말하지만 쓰이는 곳은 거의 없습니다. 이 코드는 정확히 세개의 요소를 갖는 배열을 선언하고 각각을 '`a`', '`b`', '`c`'로 초기화합니다. 즉, 문자열의 끝을 나타내는 `\0`은 들어가지 않습니다. 따라서 이 배열은 C 언어의 문자열이라고 말하기가 곤란합니다. 그래서 `strcpy`나 `printf`와 같은 함수에 인자로 전달될 수 없습니다.

대개, 배열의 크기를 지정하지 않고, 컴파일러가 배열의 크기를 알아서 지정하도록 (즉 위의 경우에서, 크기를 지정하지 않으면, 배열의 크기는 4가 됨) 하는 것이 일반적입니다.

References [C89] § 6.5.7
[H&S] § 4.6.4 p. 98

Q 11.24 왜 `void *` 타입의 포인터에는 산술(arithmetic) 계산을 할 수 없을까요?

Answer 포인터가 가리키는 오브젝트의 크기를 알 수 없기 때입니다. 따라서 연산을 하기 전에 포인터를 `char *`나 처리하고자 하는 포인터 타입으로 변환해야 합니다 (질문 4.5를 꼭 참고하기 바랍니다).

References [C89] § 6.1.2.5, § 6.3.6
[H&S] § 7.6.2 p. 204

Q 11.25 `memcpy()`와 `memmove()`은 하는 일이 같지 않나요?

Answer `memmove()`는 원본과 대상이 겹칠 경우에도 안전하게 동작한다는 것을 보장합니다. `memcpy()`는 이러한 보증을 하지 않으므로 좀 더 빨리 동작할 수 있습니다.. 의심이 간다면 `memmove()`를 쓰는 것이 더 안전합니다.

References [K&R2] § B3 p. 250
[C89] § 7.11.2.1, § 7.11.2.2
[ANSI Rationale] § 4.11.2
[H&S] § 14.3 pp. 341–2
[PCS] § 11 pp. 165–6

Q 11.26 `malloc(0)`은 무엇을 의미하죠? 이 때 널 포인터가 리턴되는 것인가요, 아니면 0 바이트를 가리키는 포인터가 리턴되는 것인가요?

Answer ANSI/ISO 표준은 둘 중 하나일 수 있다고 말하고 있습니다; 그 결과는 구현 방법에 의존적[6]입니다. (질문 11.33을 참고하기 바랍니다.

References [C89] § 7.10.3
[PCS] § 16.1 p. 386

Q 11.27 외부 이름(external identifier)을 쓸 때, 왜 ANSI 표준은 여섯 글자 이상인 이름의 유일성을 보장할 수 없다고 할까요?

Answer 오래된 링커(linker)의 경우, ANSI/ISO C나, C 컴파일러 개발자와 상관없이 시스템에 의존적인 경우가 많다는 것이 문제입니다. 이 제한은 이름의 첫 여섯 글자만을 유일하다고 보장하기 때문에, 첫 여섯 글자가 같은 이름들은, 전체가 같은 이름으로 취급합니다. 이 제한은 이미 쓸모없어져 가고 있으므로 (obsolescent), ISO 표준에서는 없어질 예정입니다.

References [C89] § 6.1.2, § 6.9.1
[ANSI Rationale] § 3.1.2
[C9X] § 6.1.2
[H&S] § 2.5 pp. 22–3

Note 즉, 여섯 글자의 제한은 "`AAAAAAB`"와 "`AAAAAAC`"가 같은 이름으로 취급될 수 있다는 말입니다.

[6] implementation-defined

그러나, 현 ISO/IEC C 표준은 internal identifier나 매크로 이름으로 적어도 63글자, external idendifier로 31글자가 유효하다고 말하고 있습니다. 또한 이 제한 사항도 다음 C 표준에서는 사라질 예정입니다.

References [C99] § 5.2.4.1, § 6.11.3

11.7 Old or Nonstandard Compilers

Although ANSI C largely standardized existing practice, it introduced sufficient new functionality that ANSI code is not necessarily acceptable to older compilers. Furthermore, any compiler may provide nonstandard extensions or may accept (and therefore seem to condone) code that the standard says is suspect.

Q 11.29 제 컴파일러는 간단한 테스트 프로그램조차 컴파일하지 못합니다. 수 많은 구문 에러를 (syntax error) 출력합니다.

Answer 아마도 ANSI 이전의 컴파일러인 것 같습니다. 그러한 컴파일러들은 함수 원형(prototype)과 같은 것을 처리하지 못합니다.

질문 1.31, 10.9, 11.30, 16.1b를 참고하기 바랍니다.

Q 11.30 제가 쓰는 컴파일러는 ANSI 컴파일러인데도 어떤 ANSI/ISO 표준 함수들이 정의되어 있지 않다고 하는군요.

Answer 컴파일러가 ANSI의 구문을 썼다고 하더라도, ANSI 호환의 헤더 파일이나, 런-타임 라이브러리를 갖지 않을 수 있습니다. (사실 gcc와 같이, 시스템 벤더(vendor)가 제공하지 않는 컴파일러에서는 종종 있는 일입니다.) 질문 11.29, 13.25, 13.26을 참고하기 바랍니다.

Q 11.31 구 스타일로 써진 C 프로그램을 ANSI C로 바꿔주거나, 또는 그와 반대 작업을 해주는 프로그램이 있을까요?

Answer 프로토타입을 쓰는 새 방식과, 구 방식을 바꿔주는 'protoize'와 'unprotoize'라는 프로그램이 있습니다. (이 프로그램이 구 스타일의 C와 ANSI C와의 100% 완전한 변환을 보장하지는 않습니다.) 이 프로그램들은 FSF의 GNU C compiler 배포판에 포함되어 있습니다; 질문 18.3을 참고하기 바랍니다.

'unproto' 프로그램은 (`ftp.win.tue.nl`의 `/pub/unix/unproto5.shar.Z`) 전처리기와 컴파일러 사이에서 변환을 담당하는 일종의 '필터(filter)'입니다. 그리고 ANSI C 스타일과 구 스타일의 변환을 거의 완벽하게 해 줍니다.

GNU Ghostscript 패키지에는 간단한 ansi2knr이라는 프로그램이 포함되어 있습니다.

그러나, ANSI C 스타일을 구 스타일로 바꿀 때, 이러한 변환이 모두 자동으로 안전하게 변환되는 것은 아닙니다. ANSI C에서는 K&R C에는 없는 새로운 기능과 복잡성을 내포하고 있으므로, 프로토타입이 있는 함수를 호출할 때에 주의해야 합니다; 아마도 캐스팅이 필요할 지도 모릅니다. 질문 11.3과 11.29를 참고하기 바랍니다.

변형된 'lint' 같은 프로그램은 prototype을 만들어 내주기도 합니다. 1992년 3월에 `comp.sources.misc`에 게시된 CPROTO 프로그램도 이런 기능을 합니다. 또 "cextract"라는 프로그램도 있습니다. 또 이러한 프로그램들이 컴파일러와 함께 제공되기도 합니다. 질문 18.16을 참고하기 바랍니다. (그러나 작은 (narrow) 인자를 갖는 구 스타일 함수를 프로토타입 스타일로 변경할 때, 주의해야 합니다; 질문 11.3을 참고하기 바랍니다.)

Q 11.32 Froozz Magic C 컴파일러는 ANSI 호환이라고 되어 있는데도, 왜 제 코드를 컴파일하지 못할까요? 제 코드는 gcc에서 동작하는 것을 보니, ANSI 호환인 데요.

Answer 많은 컴파일러들이 비표준 확장 기능을 제공합니다. gcc는 특히 더 많은 확장 기능을 제공하는 것으로 알려져 있습니다. 아마도 여러분이 만든 그 코드가 이러한 확장 기능을 사용하는 것 같습니다. 특히, 어떤 컴파일러를 가지고 한 언어의 성질을 검사한다는 것은 매우 위험합니다; 어떤 표준은 구현 방법에 따라 여러 가지 방법을 선택할 수 있도록 하고 있을 수 있으며, 컴파일러가 잘못된 경우도 많기 때문입니다. 질문 11.35를 참고하기 바랍니다.

Q 11.J 제 컴파일러는 다음과 같은 코드를 실행하면 에러가 납니다. 무엇이 잘못된 것일까요?

```
int a[3] = { 1, 2, 3 };
int b[3] = { 0, 9, 8 };
int *c[2] = { a, b };

...
```

Answer 아마도 C99 표준을 지원하지 못하는 (ANSI 표준만을 지원하는) 컴파일러를 쓰신 것 같습니다. ANSI 표준, 즉 C89 표준에 따르면 초기화에 쓸 수 있는 수식에 다음과 같은 제한이 있습니다:

> § 6.5.7 All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

즉, 정적 변수나 배열/구조체/union등의 초기값은 반드시 상수식(constant expression)이어야 합니다. 주어진 코드에서 배열 `c`의 초기값으로 쓰인 주소값 `a`, `b`는 자동 변수에서 얻은 값이기 때문에 상수식이 아닙니다.

C99 표준에 따르면, 다음과 같은 제한 사항이 있습니다:

> All the expressions in an initializer for an object that has static storage duration shall be constant expressions or string literals.

즉, 정적 변수일 경우에만 상수식 또는 상수 문자열을 써야 한다고 제한합니다. 배열/구조체/union에 대한 제한 사항은 사라진 것을 알 수 있습니다. 결국 C99 표준에 따르면, 주어진 코드는 완전히 합법적입니다.

두 가지 해결책이 있는데, 첫째, C99를 지원하는 컴파일러로 바꾸기 바랍니다. 둘째, 초기값으로 쓰일 수식을 상수식으로 바꾸면 됩니다. 예를 들어, 주어진 코드에서 변수 `a`와 `b`를 `static`으로 선언하면 됩니다.

References [C89] § 6.5.7
[C99] § 6.7.8

11.8 Compliance

Obviously, the whole point of having a standard is so that programs and compilers can be compatible with it (and therefore with each other). Compatibility is not a simple black-or-white issue, however: There are degrees of compliance, and the scope of the standard's definitions is not always as comprehensive as might be expected. In keeping with the "spirit of C," several issues are not precisely specified; portable programs must simple avoid depending on these issues.

Q 11.33 "구현 방법에 따라 정의됨 (implementation defined)", "나타나지 않은 (unspecified)", "정의되어 있지 않은 행동 (undefined behavior)"이라는 용어가 있는데, 서로 차이가 있는 것인가요?

Answer 간단하게 말해서, "implementation defined"는 각각의 구현 방법에 따라, 어떤 행동 방식(behavior)이 결정될 수 있고, 그에 따라 문서화되어 있다는 뜻입니다. "unspecified"는 구현 방법에 따라 다른 행동 방식이 결정될 수는 있지만, 문서화될 필요가 없다는 뜻입니다. "undefined"는 어떤 일이라도 일어날 수 있다는 뜻입니다. 어떤 경우에도 표준은 필요 조건을 강요하지 않습니다[7]; 때때로 처음 두 가지 경우에 대해서는 여러 가지의 상태가 제안되기도 합니다.

표준에서 'undefined behavior'라고 정의한 부분은 말 그대로 입니다. 컴파일러는 어떤! 일이라도 할 수 있습니다. 특히, 프로그램의 나머지 부분이 정상적으로 동작한다는 보장이 없습니다. 따라서 이런 부분이 여러분의 프로그램에 포함된다면 매우 위험합니다; 질문 3.2에서 간단한 예를 볼 수 있습니다.

이식성이 뛰어난 프로그램을 만들고자 한다면, 이러한 것을 다 무시하고, 위 세가지에 의존하는 어떠한 것도 만들어서는 안될 것입니다.

질문 3.9와 11.34를 참고하기 바랍니다.

References [C89] § 3.10, § 3.16, § 3.17
[ANSI Rationale] § 1.6

Q 11.34 ANSI 표준에는 많은 것들이 정의되지 않은 채 남아 있습니다. 이러한 것들에 대해서도 표준을 정해야 하지 않을까요?

Answer C 언어의 한가지 특징은, 어떠한 것들은 컴퓨터와 컴파일러에 따라 각각 다른 행동 방식이 나올 수 있다는 것입니다. 이 것은, 컴파일러가 일반적인 작업을 좀더 효과적으로 수행할 수 있도록 하기 위해서입니다. 따라서 표준은 사소한 것까지 정의하지 않고 단순히 관습을 따르는 것입니다.

프로그래밍 언어 표준은 언어 사용자와 컴파일러 개발자 사이에 위치한 일종의 계약으로 생각할 수 있습니다. 절반은 컴파일러 개발자가 제공하려 하는 것과, 사용자가 '이런 것들은 제공될 것이다'하고 생각하는 것들로 이루어지고, 나머지 절반은 사용자가 따라야 하는 규칙과 개발자가 따를 것으로 생각되는 규칙으로 이루어 집니다. 따라서 어느 한 쪽이 이런 규약을 어긴다면, 어떤 일이 발생할 지 아무도 보장할 수 없습니다.

덧붙여 질문 11.35도 참고하시기 바랍니다.

[7] In no case does the Standard impose requirements

References [ANSI Rationale] § 1.1

Q 11.35 `i = i++`과 같은 코드의 행동 방식은 정의되어 있지 않다고 들었습니다. 그런데, 이 코드를 ANSI를 준수하는 컴파일러에서 실행하면 제가 추측한 그 결과가 나옵니다.

Answer 컴파일러는 행동 방식이 정의되어 있지 않는(undefined behavior) 코드에 대해서는 여러분이 예상한 방식을 포함한, 어떠한 일도 할 수 있습니다. 이런 것에 의존하는 것은 매우 나쁩니다. 질문 11.32와 11.33, 11.34를 참고하기 바랍니다.

11.9 `volatile` qualifier

TODO: 'volatile'에 대해 소개하기. `volatile`은 `const`와 더불어 C89에서 소개된 것입니다.

Q 11.A 자주 쓰이지는 않다고 알고 있는데, `volatile`이 정확히 어떤 의미를 가지는 것인가요?

Answer `volatile` 타입 qualifier는 주어진 오브젝트가 컴파일러가 의도하지 않은 방식으로 변경될 수 있다는 것을 나타냅니다. 따라서, 컴파일러는 이 오브젝트를 최적화(optimization) 과정에서 제외시킵니다. 좀더 정확히 말해서, 이 오브젝트에 대한 참조(reference)나 변경(modification)은 sequence point를 넘어다니며 최적화되지 않습니다. 단, sequence point 안에서 최적화될 수 있습니다. (sequence point에 관한 것은 질문 3.8을 참고하기 바랍니다.)

일반적으로, `volatile`이 쓰이는 곳은 크게 두 가지로 나누어 생각할 수 있습니다.

- 첫째, (대부분 디바이스 드라이버라고 하는) 하드웨어를 직접 제어하는 코드
- 둘째, `setjmp`와 `longjmp` 함수를 써서 non-local goto를 사용하는 코드
- 셋째, (인터럽크 관련) signal handler에서 (보통 전역) 변수의 값을 설정할 때.

예를 들어, 어떤 시스템은 세 개의 특정 메모리 주소를 제공하고, 이 중 두 개는 하드웨어의 정보를 알려 주는 데에 쓰이며, 나머지 하나는 하드웨어에 직접 데이터를 쓰기 위한 목적으로 사용한다고 가정해 봅시다. 읽는 목적으로

쓰는 주소는 각각 in1, in2라는 포인터가 가리키고 있고, 쓰기 위한 주소는 out이라는 포인터에 저장되어 있다고 가정합시다. 이 경우 다음과 같은 코드를 예상할 수 있습니다:

```
volatile unsigned char *out;
volatile unsigned char *in1, *in2;
int i;
...
for (i = 0; i < N; ++i)
  *out = a[i] & (*in1 + *in2);
```

이 코드는 out이 가리키는 곳에, *in1과 *in2를 더해서, a[i]의 값과 AND 한 결과를 쓰게 됩니다. (위 코드에서 volatile이 없다고 가정하면) 단순한 시스템일 경우, 루프를 매번 돌 때마다, *in1 + *in2를 수행해서, 그 결과를 a[i]와 더해, *out에 쓰게 되지만, 최적화를 수행한다면, 매번 *in1 + *in2 덧셈을 수행할 이유가 없습니다. 그래서 컴파일러는 보통 더한 결과를 특정 레지스터에 저장해 두고, 이 것을 루프를 반복할 때마다 a[i]와 더하는 코드를 만들어 냅니다. 그러나 *in1과 *in2는 하드웨어가 직접 건드리는 값이 들어 있으므로, 루프를 돌 때, 매번 같다는 보장을 할 수 없습니다. 따라서 최적화를 수행한 코드와 그렇지 않은 코드가 서로 실행 결과가 다르거나, 예상하지 못한 결과를 가져올 수 있습니다.

이 때, 관련된 변수인 in1, in2, out를 volatile로 선언함으로써, 이 변수들이 컴파일러의 의도와 상관없이 변경될 수 있다는 것을 알려주면, 컴파일러는 이 변수가 관계된 코드는 최적화 고려 대상에서 제외시킵니다.

또한 non-local goto 역할을 수행하는 함수 setjmp와 longjmp를 쓸 때, volatile을 유용하게 쓸 수 있습니다. 자세한 것은 질문 20.A를 참고하기 바랍니다.

Signal handler에 대한 것은 질문 19.A를 참고하기 바랍니다.

References [H&S2002] § 4.4.5 [C99 Rationale] § 5.1.2.3, § 5.2.3, § 6.7.3 [C99] § 5.1.2.3, § 6.7.3 pp. 108–109, § 7.13.2.1, § 7.14

11.10 `restrict` qualifier

The type qualifier `restrict` is new in C99. It may only be used to qualify pointers to object or incomplete types, and in serves as a “no alias” hint to the C compiler...

TODO: ‘restrict’ qualifier에 대하여 소개하기.

11.11 Flexible Array Members

질문 2.6의 질문에 있는 코드는 매우 자주 쓰이지만, 엄밀하게 말해, C89 표준까지 "undefined behavior"로 간주되어 왔습니다. 즉, 동작은 하지만, C 표준에는 위배되는 코드였습니다.

C99 표준은 새로 "flexible array member"라는 개념을 도입해서 이런 문제를 쉽고 깔끔하게 해결할 수 있도록 도와줍니다.

Q 11.G C99에 "flexible array member"가 소개된 것으로 아는데, 정확히 무엇을 뜻하는 것인가요?

Answer 먼저 질문 2.6을 보기 바랍니다. 그 질문에 나온 structure와 답변 첫 번째로 주어진 `makename` 함수를 새로 만들 수 있습니다.

```
struct name {
  int namelen;
  char namestr[];  /* flexible array member */
};
```

즉, 배열의 크기를 지정하지 않으면 됩니다. 쓰는 방법은 다음과 같습니다:

```
#include <stdlib.h>
#include <string.h>

struct name *makename(char *newname)
{
  struct name *ret =
    malloc(sizeof(struct name) +
           strlen(newname) + 1);
      /* No need to add -1; +1 for \0 */
  if (ret != NULL) {
    ret->namelen = strlen(newname);
    strcpy(ret->namestr, newname);
  }
  return ret;
}
```

단, 다음 두 가지를 조심해야 합니다. 첫째, flexible array member를 쓰기 위해서는, structure에 다른 멤버가 하나 이상 나와야 합니다. 즉 아래와 같은 코드는 잘못된 것입니다:

```
struct name {
  char namestr[];  /* flexible array member */
};
```

둘째, flexible array member는 항상 structure의 마지막 member로 나와야 합니다.

References [C99] § 6.7.2.1
[C99 Rationale] § 6.7.2.1

Q 11.H 질문 11.G에 나온 것처럼, flexible array member를 가진 structure의 크기는 어떻게 계산되나요? 즉, `sizeof` 연산자를 쓰면, 어떤 값을 돌려주나요?

Answer Flexible array member를 가진 structure에 `sizeof` 연산자를 쓰면, 그 flexible array member를 뺀 크기를 알려줍니다. 다음과 같은 structure를 생각해 봅시다:

```
struct foo {
  char ch;
  int i[];
};
```

그리고 쓰고 있는 시스템은 `sizeof(int)`가 4이고, `int`는 항상 4 byte의 배수인 주소에 align된다고 가정합시다. 위 structure에서 `sizeof(ch)`는 정의에 의해 1입니다. 그리고 `i`는 flexible array member이므로 무시됩니다. 단, `i`는 4 byte alignment를 지켜야 하므로, `ch`와 `i` 사이에는 3 byte의 padding이 존재하게 됩니다. 이 padding은 structure의 크기에 포함되므로 결국 이 경우, `sizeof(struct foo)`는 4입니다.

단, 위 단락은 이해를 돕기 위해 쓴 글입니다. 특이한 방식의 alignment를 쓰는 시스템에서는 다르게 나올 수 있습니다. 중요한 것은 다음 문장입니다. 꼭 기억해 두기 바랍니다.

> `sizeof` applied to the structure that includes a flexible array member ignores the array but counts any padding before it.

References [C99] § 6.7.2.1
[C99 Rationale] § 6.7.2.1

11.12 Types

Q 11.I 원하는 크기에 정확히 맞는 정수 타입을 쓰고 싶습니다. 새로운 C 표준에서 이러한 타입을 제공한다고 들었는데, 맞습니까?

Answer C99 표준에 따라, `<stdint.h>` 또는 `<inttypes.h>`를 포함시킬 경우, 다음과 같은 타입을 쓸 수 있습니다. 아래 표에서 N은 8, 24와 같은 십진수를 뜻합니다. 또한 두세번째 열에서 나온 최대/최소값을 위한 매크로는 `<stdint.h>`를 포함하기 전, `__STDC_LIMIT_MACROS`를 정의해야 쓸 수 있습니다.

```
#define __STDC_LIMIT_MACROS
#include <stdint.h>
```

(경고: 이 타입들은 시스템에 따라서 제공되지 않을 수도 있습니다.[8]):

int$N$_t	INTN_MIN $-(2^{N-1})$	INTN_MAX $2^{N-1}-1$
uint$N$_t	0	UINTN_MAX $2^{N-1}-1$
int_least$N$_t	INT_LEASTN_MIN $-(2^{N-1}-1)$	INT_LEASTN_MAX $2^{N-1}-1$
uint_least$N$_t	0	UINTN_MAX $2^{N}-1$
int_fast$N$_t	INT_FASTN_MIN $-(2^{n-1}-1)$	INT_FASTN_MAX $2^{N-1}-1$
uint_fast$N$_t	0	UINT_FASTN_MAX $2^{N}-1$
intptr_t	INTPTR_MIN $-(2^{15}-1)$	INTPTR_MAX $2^{15}-1$
uintptr_t	0	UINTPTR_MAX $2^{16}-1$
intmax_t	INTMAX_MAX $-(2^{63}-1)$	INTMAX_MAX $2^{63}-1$
uintmax_t	0	UINTMAX_MAX $2^{64}-1$

[8]However, if an implementation provides integer types with width of 8, 16, 32, or 64 bits, it shall define the corresponding typedef names.

(a) `int`*N*`_t`는 크기 *N*인, 부호있는 정수를 위한 typedef 이름입니다. (시스템이 8, 16, 32, 64 비트를 가진 정수 타입을 지원한다면, 해당하는 `int`*N*`_t`도 정의되어야 합니다. 나머지는 optional입니다.)

(b) `int`*N*`_t`는 부호없는 정수를 위한 이름이며, 제한 사항은 위와 같습니다.

(c) `int_least`*N*`_t`는 부호있는, 적어도 *N*보다 큰 크기를 가진 정수를 위한 typedef 이름입니다. (*N*이 8, 16, 32, 64인 경우는 필수로 제공되며, 나머지는 optional)

(d) `int_least`*N*`_t`는 부호없는, 적어도 *N*보다 큰 크기를 가진 정수를 위한 typedef 이름입니다. 제한 사항은 위와 같습니다.

(e) `int_fast`*N*`_t`는 적어도 *N* 크기를 가지는, 빠르게 동작할 수 있는[9], 부호있는 정수를 위한 typedef 이름입니다. (*N*이 8, 16, 32, 64인 경우는 필수로 제공되며, 나머지는 optional입니다.)

(f) `uint_fast`*N*`_t`는 부호없는 정수를 위한 것이며, 나머지는 위 타입과 같습니다.

(g) `intptr_t`는 void 포인터의 값을 저장하고, 다시 void 포인터로 변환했을 때 변화없음을 보장하는, 부호있는 정수를 위한 typedef 이름입니다. 즉, 포인터를 안전하게 저장할 수 있는 정수 타입이라고 생각하시면 됩니다. (이 타입은 optional입니다.)

(h) `uintptr_t`는 `intptr_t`와 같으며, 부호없는 정수를 위한 typedef 이름입니다. (이 타입은 optional입니다.)

(i) `intmax_t`는 부호있는 정수 중 가장 큰 값을 표현할 수 있는 타입입니다. (이 타입은 필수로 제공됩니다.)

(j) `uintmax_t`는 부호없는 정수 중 가장 큰 값을 표현할 수 있는 타입입니다. (이 타입은 필수입니다.)

표준에 따라 정확히 말하면, `int_16_t`, `int32_t`등이 정의된 헤더 파일은 `<stdint.h>`에 정의되어 있고, `<inttype.h>`는 `<stdint.h>`를 포함하게 되며, 부가적인 사항들을 제공합니다. 이 부가적인 사항들은 질문 12.A와 12.B를 참고하기 바랍니다.

덧붙여 질문 1.1도 참고하시기 바랍니다.

[9] 표준에 따라, 모든 경우에 가장 빠르게 동작하는 것을 보장하지 않습니다. 만약 시스템이 어느 경우 가장 빠르게 동작하는 타입인 것을 선택할 수 없다면, 단순히 부호있음/없음과 크기만을 만족하는 타입을 쓸 수도 있다는 것을 명심하기 바랍니다.

Chapter 12

The Standard I/O Library

프로그램에게 할 일을 말할 수 없거나, 프로그램이 자기가 처리한 일을 사용자에게 알려줄 수 없다면, 쓸모있는 프로그램이라고 말할 수 없을 것입니다. 따라서 거의 모든 프로그램은 입출력(I/O)을 처리한다고 말할 수 있습니다. C 언어에서 I/O는 라이브러리 함수를 통해서 제공됩니다 — 표준 I/O 또는 "stdio" 라이브러리[1] — 그리고 이 함수들은 가장 많이 쓰이는 C 라이브러리 중의 하나입니다.

C 언어의 minimalist philosophy에 동조하기 위해, stdio 함수들은 간단한, 그러나 꼭 필요한(least-common-denominator) I/O 모델입니다. 여러분은 파일을 열고(open), 읽고(read), 쓸(write) 수 있습니다. 파일은 연속적인 문자의 스트림(sequential character stream)으로 취급되며, 찾기(seeking[2])가 가능합니다. 필요하다면 파일을 텍스트(text)와 이진(binary) 로 구별하여 처리할 수 있으며, 파일에 문자열을 써서 이름을 지어줄 수 있습니다; 파일 이름에 관한 자세한 것은 OS에 따라 달라집니다. 파일 이름을 지을 때를 제외하고 디렉토리에 관한 개념은 C 언어에 없습니다. 즉, 파일 이름을 지정할 때 디렉토리 이름을 쓸 수 있지만, 디렉토리 자체를 취급하는 일, 디렉토리를 만들거나, 디렉토리를 나열하는 일을 처리하는 표준 방법은 없습니다 (19 절을 참고하기 바랍니다). 모든 프로그램은 처음 시작할 때 자동적으로 세 개의 predefined I/O 스트림이 열려지게 됩니다: 여러분은 대개 키보드로 연결된 `stdin`에서 읽을(read) 수 있고, 대개 스크린으로 연결된 `stdout` 이나 `stderr`를 써서 쓸(write) 수 있습니다. 그렇지만 키보드나 스크린과 관련된 기능은 매우 적습니다 (다시 19 절을 참고하기 바랍니다).

이 절의 많은 부분이 `printf` (질문 12.6에서 12.11까지)와 `scanf` (질문 12.12에서 12.20까지)에 연관되어 있습니다. 질문 12.21에서 12.26은 다른 stdio 함수들을

[1] 우리가 "stdio library"라고 말할 때는 "C 런타임 라이브러리안에 있는 stdio 함수들"이나 `<stdio.h>`에 설명된 함수들을 의미합니다."

[2] 원하는 내용을 읽거나 쓸 수 있도록 file position을 옮긴다는 뜻

설명합니다. 특정 파일을 access할 필요가 있다면, `fopen` (질문 12.27부터 12.32까지)을 써서 직접 열어서(open) 쓰거나(use), 표준 stream을 redirect해서 (질문 12.33에서 12.36까지) 쓸 수 있습니다. text I/O를 하려는 마음이 없다면, "binary" stream을 쓸 수 (질문 12.37에서 12.42까지) 있습니다. 이 모든 것을 자세히 알아보기 전에 먼저 간단한 것부터 알고 넘어갑시다. 다음 질문들을 읽어보기 바랍니다.

12.1 Basic I/O

Q 12.1 이 코드에서 잘못된 부분이 있나요?

```
char c;
while ((c = getchar()) != EOF) ...
```

Answer 일단, `getchar`의 리턴 값을 저장하는 변수는 반드시 `int`이어야 합니다. `getchar()`는 어떠한 문자 값이나, EOF를 리턴할 수 있습니다. EOF는 `int` 타입이기 때문에 이 리턴 값을 `char`에 저장하는 것은 EOF를 잘못 해석하게 할 소지가 있습니다 (특히 `char`의 타입이 `unsigned`인 경우 문제가 심각합니다).

위의 코드처럼 `getchar()`의 리턴값을 `char`에 담을 경우, 두 가지 결과를 예상할 수 있습니다.

- `char`의 타입이 signed인 경우, 그리고 `EOF`가 -1로 정의된 경우, 문자값이 부호 확장(sign extension)되어, 문자값 255가 (C 언어 표현으로 `'\377'` 또는 `'\xff'`) `EOF`와 같아집니다. 따라서 입력 도중에 입력 처리가 끝나버릴 수 있습니다.
- `char`의 타입이 unsigned인 경우, EOF 값이 (상위 비트들이 잘라져 대개 255나 `0xff`의 값으로) 잘라져서, EOF로 인식이 되지 않아 버립니다. 따라서 이 경우, 무한 루프에 빠져 버립니다.[3]

이 버그는 `char`가 signed이고, 입력이 모두 7 비트 문자인 경우, 발견하기가 매우 힘듭니다 (`char`가 signed인지 unsigned인지는 implementation-defined 입니다.)

References [K&R1] § 1.5 p. 14
[K&R2] § 1.5.1 p. 16
[C89] § 6.1.2.5, § 7.9.1, § 7.9.7.5
[H&S] § 5.1.3 p. 116, § 15.1, § 15.6

[3]앞 절에서 쓴 것처럼, 255는 `char`가 8 비트라는 것을 가정한 것입니다. `char`가 8 비트보다 더 큰 시스템에서는, 비슷하게 처리가 끝나버릴 수 있습니다.

[CT&P] § 5.1 p. 70
[PCS] § 11 p. 157

Q 12.2 이 코드는 마지막 문장을 두번 복사하는데 왜 그럴까요?

```
while (!feof(infp)) {
  fgets(buf, MAXLINE, infp);
  fputs(buf, outfp);
}
```

Answer C 언어에서 end-of-file은 단지 입력 루틴이 읽으려 했으나 실패했다는 것을 의미합니다. (다시 말해 C의 I/O는 Pascal과는 다릅니다.) 일반적으로 다음과 같이 입력 루틴의 리턴 값을 검사해야 합니다:

```
while (fgets(buf, MAXLINE, infp) != NULL)
  fputs(buf, outfp);
```

대개 `feof()`를 직접 쓸 이유는 없습니다 (stdio 함수가 `EOF`나 `NULL`을 리턴한 경우, `feof()`나 `ferror()`를 써서 정말로 파일의 끝인지, 읽기(read) 에러인지 검사할 필요가 있긴 합니다.)

References [K&R2] § 7.6 p. 164
[ANSI] § 4.9.3, § 4.9.7.1, § 4.9.10.2
[C89] § 7.9.3, § 7.9.7.1, § 7.9.10.2
[H&S] § 15.14 p. 382

Note 예를 들어 `fread()`나 `fwrite()`, `fgets()`, `fputs()`와 같은 함수들은 에러가 발생하거나 파일 끝에 다다랐을 때, `EOF`를 리턴합니다. 이런 함수들이 정말 파일 끝에 다다랐는지 확인할려면 `feof()`나 `ferror()`를 써야 하는 것입니다.

이런 경우를 제외하고, feof()를 직접 쓰는 경우는 매우 드뭅니다. 따라서 일반적으로 다음과 같은 코드보다:

```
while (!feof(fp))
  fgets(line, max, fp);
```

다음과 같은 코드를 쓰기 바랍니다:

```
while (fgets(line, max, fp) != NULL)
  ...;
```

즉, feof()는, end-of-file인지 아닌지 검사하기 위해 쓰는 함수가 아니라, end-of-file과 error인 상황을 구별하기 위해 쓰는 함수라고 기억하는 것이 좋습니다. (물론 이 목적으로, ferror()를 대신 쓸 수 있습니다.)

Q 12.3 `fgets()`를 써서, 파일에서 한 줄씩 읽어 그 내용을 포인터의 배열에 저장하려 합니다. 왜 모든 줄의 내용이 마지막 줄로 되어 있을까요?

Answer 질문 7.4를 보기 바랍니다.

Q 12.4 제 프로그램의 프롬프트와 중간 단계까지의 출력은 (특히 파이프를 써서 출력을 다른 프로그램에 넘길 때), 때때로 스크린에 출력되지 않습니다.

Answer 출력이 당장 스크린에 반영되기를 원한다면[4] (특히 텍스트가 `\n`으로 끝나지 않는 경우), `fflush(stdout)`을 불러 주는 것이 좋습니다. 대개의 메커니즘이 적절한 시기에 자동으로 `fflush()`를 불러주지만, 이 메커니즘들은 `stdout`이 interactive한 터미널로 연결되어 있다고 가정한 것이기 때문에, 가끔 제대로 동작하지 않을 수 있습니다. (덧붙여 질문 12.24도 참고하시기 바랍니다.)

References [ANSI] § 4.9.5.2
[C89] § 7.9.5.2

Note 질문 21.11의 예를 보면, `fflush`를 부르는 예를 볼 수 있습니다.

Q 12.5 RETURN 키를 누르지 않고 한 글자를 즉시 입력받을 수 있는 방법이 있을까요?

Answer 질문 19.1을 참고하기 바랍니다.

12.2 `printf` Formats

Q 12.6 `printf`로 '`\%`'를 출력할 수 있을까요? `\%`를 써 보았지만 출력되지 않더군요.

[4] 다른 방법으로, `setbuf()`나 `setvbuf()`를 써서 버퍼링을 쓰지 않은 방법이 있지만, 버퍼링을 사용하지 않으면 성능이 떨어질 가능성이 높으므로 그리 좋다고 할 수 없습니다.

Answer %%를 쓰면 됩니다. % 문자를 출력하는 것이 까다로운 이유는, printf()가 % 문자를 escape 문자로 쓰기 때문입니다. 아시다시피, printf는 % 문자를 보면, 다음 글자를 읽어서 어떤 값을 출력해야 하는지 조사하게 됩니다. 이때, 다음 글자로 %를 쓰면 — 즉 %%를 쓰면 — '%'를 출력하게 됩니다.

\%는 동작하지 않습니다. 이 방법이 왜 동작하지 않는지 이해하려면, 먼저 일단 백 슬래시('\')는 컴파일러의 escape 문자이며, 컴파일러가 여러분의 코드를 이해하는 길을 보여주는 목적으로 쓰인다는 것을 아셔야 합니다. 컴파일러 입장에서는 \%는 정의되지 않은 escape 문자이며, 대개 '%' 한 글자를 의미하게 됩니다. 따라서 만약 printf()가 '\'를 특별하게 처리할 수 있도록 만들어져 있다 하더라도, 컴파일러가 이를 먼저 처리해 버리게 되므로, '%'를 출력할 수 없게 됩니다.

덧붙여 질문 8.8, 19.17도 참고하시기 바랍니다.

References [K&R1] § 7.3 p. 147
[K&R2] § 7.2 p. 154
[C89] § 7.9.6.1

Q 12.7 다음 코드는 왜 동작하지 않나요?

```
long int n = 123456;
printf("%d\n", n);
```

Answer long int를 출력할 때는, 반드시 l (L의 소문자) modifier를 printf 포맷에 써야 (예를 들어 %ld) 합니다. 왜냐하면 printf()는 여러분이 전달한 인자의 (여기서는 변수 n의 값) 타입을 알 수 없기 때문입니다. 그러므로 반드시 올바른 포맷 (format specifier)을 써야 됩니다.

Q 12.8 Aren't ANSI function prototypes supposed to guard against argument type mismatches?

Answer 질문 15.3을 참고하기 바랍니다.

Q 12.9 printf()에서 %lf를 쓰지 말라고 한 것을 들었습니다. double 타입을 쓸 때, scanf()는 %lf를 쓰면서 왜 printf()는 %f를 쓰는 것인가요?

Answer `printf`의 %f specifier는 `float`과 `double` 두 타입에 모두 쓰는 것은 사실입니다.[5] "default argument promotion" 때문에 (`printf`와 같이 가변 인자[6]를 사용하거나, 프로토타입이 없는 함수에게 적용됨) `float` 타입의 값은 인자로 전달될 때, `double` 타입으로 바뀌게 됩니다. 따라서 `printf`는 결국 무조건 `double` 타입만 전달받는 셈입니다. 덧붙여 질문 15.2도 참고하시기 바랍니다. (참고로 `printf`에 `long double` 타입을 쓸 경우에는 %Lf를 사용합니다.)

`scanf`는 인자로 포인터를 받기 때문에 그러한 promotion이 일어나지 않습니다. 따라서 완전히 다른 상황입니다. (포인터를 써서) `float`에 값을 저장하는 것은 `double`에 값을 저장하는 것과 다릅니다. 따라서 `scanf`는 `%f`와 `%lf`로서 구별합니다.

아래 표에서는 `printf`와 `scanf`에서 쓰이는, 각각의 타입에 따른 format specifier를 보여줍니다.

Format	`printf`	`scanf`
`%c`	`int`	`char *`
`%d`, `%i`	`int`	`int *`
`%o`, `%u`, `%x`	`unsigned int`	`unsigned int *`
`%ld`, `%li`	`long int`	`long int *`
`%lo`, `%lu`, `%lx`	`unsigned long int`	`unsigned long int *`
`%hd`, `%hi`	`int`	`short int *`
`%ho`, `%hu`, `%hx`	`unsigned int`	`unsigned short int *`
`%e`, `%f`, `%g`	`double`	`float *`
`%le`, `%lf`, `%lg`	n/a	`double *`
`%s`	`char *`	`char *`
`%[...]`	n/a	`char *`
`%p`	`void *`	`void **`
`%n`	`int *`	`int *`
`%%`	none	none

(엄격하게 말해서, 대부분의 시스템이 받기는 하지만, `printf`에서 `%lf`는 정의되어 있지 않습니다. 이식성을 위해서 항상 `%f`를 쓰기 바랍니다.)

덧붙여 질문 12.13, 15.2도 참고하시기 바랍니다.

References [K&R1] § 7.3 pp. 145–47, § 7.4 pp. 147–50

[K&R2] § 7.2 pp. 153–44, § 7.4 pp. 157–59

[5] `printf`의 `%e`와 `%g`, 그리고 각각 같은 뜻을 가지는 `scanf`의 `%le`와 `%lg`에 대해서도 같습니다.

[6] Default argument promotion은 가변 인자 리스트에서 가변 인자 부분에만 적용됩니다. Chapter 15 참고 바랍니다.

[C89] § 7.9.6.1, § 7.9.6.2
[H&S] § 15.8 pp. 357–64, § 15.11 pp. 366–78
[CT&P] § A.1 pp. 121–33

Q 12.9b `printf`로 `size_t`와 같은, 즉, 그 크기가 `long`인지, 다른 것인지 알 수 없는 `typedef` 타입을 출력할 때에는 어떤 포맷을 써야 하나요?

Answer 주어진 타입을 알려진, 적절한 타입으로 캐스팅해서 출력하면 됩니다. 예를 들어, 어떤 타입의 크기를 출력할 때에는 다음과 같이 할 수 있습니다:

```
printf("%lu", (unsigned long)sizeof(thetype));
```

Q 12.10 `printf()`에서 가변 폭(field width)을 지정하려 합니다. 예를 들어, %8d를 쓰지 말고 실행 시간에 이 폭을 지정하려 합니다.

Answer `printf("%*d", width, x)`를 쓰면 됩니다. Format specifier에서 별표는 필드 폭을 지정하기 위해 `int` 값이 인자로 들어온다는 것을 뜻합니다. (인자 목록에서 폭은 출력할 값보다 먼저 지정합니다.) 덧붙여 질문 12.15도 참고하시기 바랍니다.

References [K&R1] § 7.3
[K&R2] § 7.2
[C89] § 7.9.6.1
[H&S] § 15.11.6
[CT&P] § A.1

Q 12.11 천 단위로 수치에 콤마(comma)를 찍어서 출력하고 싶습니다. 화폐 단위같은 수치를 출력할 수 있는 방법이 있나요?

Answer `<locale.h>`는 이러한 연산을 수행하기 위한 간단한 기능을 제공합니다. 그러나 이런 일을 하기 위한 표준 루틴은 없습니다. (`printf()`가 할 수 있는 일은, locale 설정에 따라 소수점을 찍는 것 뿐입니다.[7].

Locale 설정에 따른, 콤마로 구분되는 수치를 출력하기 위한, 간단한 함수를 제공합니다:

[7] The only thing `printf()` does in response to a custom locale setting is to change its decimal-point character

```
#include <locale.h>

char *
commaprint(unsigned long n)
{
  static int comma = '\0';
  static char retbuf[30];
  char *p = &retbuf[sizeof(retbuf) - 1];
  int i = 0;

  if (comma == '\0') {
    struct lconv *lcp = localeconv();
    if (lcp != NULL) {
      if (lcp->thousands_sep != NULL &&
          *lcp->thousands_sep != '\0')
        comma = *lcp->thousands_sep;
      else
        comma = '.';
    }
  }
  *p = '\0';
  do {
    if (i % 3 == 0 && i != 0)
      *-p = comma;
    *-p = '0' + n % 10;
    n /= 10;
    i++;
  } while (n != 0);
  return p;
}
```

좀 더 나은 방법은 세개의 숫자로 구별된다고 가정하지 말고, `lconv` 구조체의 `grouping` 필드를 쓰는 것입니다. `retbuf`의 적당한 크기는 다음과 같습니다:

```
4 * (sizeof(long) * CHAR_BIT + 2) / 3 / 3 + 1
```

질문 12.21을 참고하기 바랍니다.

References [C89] § 7.4
[H&S] § 11.6 pp. 301–4

Q 12.A int16_t과 같은 타입을 출력하기 위해서, 어떤 포맷을 써야 하나요?

Answer 헤더 파일 <inttypes.h>를 포함시키면 다음과 같은 매크로를 쓸 수 있습니다. 모든 매크로는 문자열 상수(character string literal)로 치환됩니다:

int*N*_t	int_least*N*_t	int_fast*N*_t	intmax_t	intptr_t
PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR
uint*N*_t	**uint_least*N*_t**	**uint_fast*N*_t**	**uintmax_t**	**uintptr_t**
PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR

모든 매크로는 PRI로 시작하며, printf() 계열의 함수에서 쓸 수 있는, %d, %i, %o, %u, %x, %X 포맷을 의미하는 글자인 d, i, o, u, x, X가 나온 다음, 각각의 타입을 대문자로 나타낸 이름이 나옵니다.

즉, uint_fast16_t를 썼고, %o를 쓴 효과를 얻으려면, PRIoFAST16을 쓰면 됩니다. 마찬가지로, 가장 큰 unsigned 타입의 정수값을 저장하고 싶다면 uintmax_t 타입의 오브젝트를 쓰면 됩니다. 예를 들면 다음과 같습니다:

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

int
main(void)
{
  uint_fast16_t foo;
  uintmax_t i = UINTMAX_MAX;
  ...
  fprintf(stderr, "foo is %" PRIoFAST16 ".\n", foo);

  printf("The largest integer value is %020" PRIxMAX "\n",
         i);

  return 0;
}
```

덧붙여 질문 1.1, 11.I, 12.B도 참고하시기 바랍니다.

References [C99] § 7.8.1, § 7.18.1

12.3 scanf Formats

Q 12.12 왜 `scanf("%d", i)`는 동작하지 않나요?

Answer `scanf()`에 전달하는 인자는 항상 포인터이어야 합니다: 각각의 값이 변환될 때마다 `scanf`는 여러분이 전달한 포인터가 가리키는 위치에 그 값을 저장합니다. (덧붙여 질문 20.1도 참고하시기 바랍니다.) 그렇기 때문에, `scanf("%d", &i)`로 써야 합니다.

Q 12.13 다음 코드는 왜 동작하지 않나요?

```
double d;
scanf("%f", &d);
```

Answer `printf()`와는 달리, `scanf()`는 `double` 타입을 쓸 때에는 %lf를, `float` 타입을 쓸 때에는 %f를 씁니다.[8] `scanf`는 `%f` format을 보면 `float`을 가리키는 포인터를 받게 됩니다. 따라서 여러분이 준 `double`을 가리키는 포인터는 잘못된 것입니다. `%lf`을 쓰거나 변수를 `float`으로 선언하기 바랍니다. 덧붙여 질문 12.9도 참고하시기 바랍니다.

Q 12.14 이 코드는 왜 동작하지 않나요?

```
short int s;
scanf("%d", &s);
```

Answer `%d`를 쓰면, `scanf`는 `int`를 가리키는 포인터를 받습니다. `short int`를 가리키는 포인터를 쓰려면 `%hd`를 쓰기 바랍니다. (질문 12.9의 표를 참고하기 바랍니다.)

Q 12.15 `scanf` 포맷 문자열에 가변 폭을 지정할 수 있을까요?

[8] 여기에서 말한 것은 `%e`와 `%f`, 비슷한 format인 `%le`와 `%lg`에도 적용됩니다.

Answer 안됩니다. `scanf()`에서 별표(`*`)는 대입을 생략하기 위한 것입니다. ANSI 문자열 관련 함수를 사용하거나, 원하는 폭 크기를 나타내는 preprocessor macro를 써서 `scanf` format string을 만들어야 합니다:

```
#define WIDTH          3

#define Str(x)         #x
#define Xstr(x)        Str(x)    /* 질문 11.17 참고 */

scanf("%" Xstr(WIDTH) "d", &n);
```

폭의 크기가 실행 시간에만 알 수 있다면, format string을 실행 시간에 만들어야 할 것입니다:

```
char fmt[10];
sprintf(fmt, "%%%dd", width);
scanf(fmt, &n);
```

이렇게 만든 `scanf` format은 표준 입력에서 읽을 때보다는, `fscanf`나 `sscanf`에서 쓸모있을 것입니다. 덧붙여 질문 11.17, 12.10도 참고하시기 바랍니다.

Q 12.16 파일에서 데이터를 미리 지정한 형식으로 읽으려면 어떻게 해야 할까요? `scanf()`에서 `%f`를 10번 쓰는 방법 말고, 어떻게 10개의 `float`을 파일에서 읽을 수 있을까요? How can I read an arbitrary number of fields from a line into an array?

Answer 일반적으로 데이터를 parsing(파싱)하는 방법은 크게 세 가지로 나눌 수 있습니다:

(a) 적당한 format string과 함께 `fscanf`나 `sscanf`를 씁니다. 이 chapter에서 지적한 제한 사항에도 불구하고 (질문 12.20 참고), `scanf` 계열의 함수는 매우 강력합니다. 공백으로 구별된 필드가 가장 다루기 쉽지만, `scanf` format string은 좀 더 compact, column-oriented인 FORTRAN-style의 데이터를 처리하는 데에도 쓰입니다. 예를 들어, 아래와 같은 줄은:

```
1234ABC5.678
```

`"%d%3s%f"`를 써서 읽을 수 있습니다. (질문 12.19의 마지막 예를 참고하기 바랍니다.)

(b) 데이터를 공백으로 (또는 다른 delimeter를 써서) 구별되는 필드로 쪼개고, `strtok` 함수나 이와 같은 기능을 가진 함수를 (질문 13.6 참고) 쓴 다음, `atoi`나 `atof`와 같은 함수를 쓰면 됩니다. (일단 이렇게 쪼개고 나면, 각각의 field를 다루는 코드는 `main()`에서 `argv` 배열을 다루는 것과 거의 같아집니다; 질문 20.3 참고) 이 방법은 특히 데이터가 몇 개의 필드로 이루어졌는지 모를 때 (실제 데이터를 읽기 전까지) 특히 쓸모가 있습니다.

아래는 10개의 실수를 배열로 복사하는 간단한 예제입니다:

```
#include <stdlib.h>
#define MAXARGS 10

char *av[MAXARGS];
int ac, i;
double array[MAXARGS];

ac = makeargv(line, av, MAXARGS);
for (i = 0; i < ac; i++)
  array[i] = atof(av[i]);
```

(`makeargv`의 정의는 질문 13.6을 참고하기 바랍니다.)

(c) 특별한 방법으로 (in ad hoc way) 데이터를 분석할 수 있는 아무런 포인터 연산이나 라이브러리 함수를 쓰면 됩니다. (ANSI `strtol`과 `strtod` 함수는 어디서 분석을 멈추었는가를 포인터로 알려주기 때문에 특히 이런 스타일의 parsing에 쓸모 있습니다.) 이 방법은 물론 가장 일반적인 방법이나, 가장 에러를 발생하기도 쉽고 어려운 방법입니다: The thorniest parts of many C programs are those that use lots of tricky little pointers to pick apart strings.

가능하다면 데이터 파일의 형식과 입력 형식을 복잡한 파싱 과정이 필요없게 디자인하고, 첫번째 또는 두번째 방법을 써서 분석할 수 있도록 만드는 것이 좋습니다.

Q 12.B int16_t와 같은 타입을 입력받기 위해서, 어떤 포맷을 써야 하나요?

Answer 헤더 파일 `<inttypes.h>`를 포함시키면 다음과 같은 매크로를 쓸 수 있습니다. 모든 매크로는 문자열 상수(character string literal)로 치환됩니다:

int*N*_t	int_least*N*_t	int_fast*N*_t	intmax_t	intptr_t
SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR
uint*N*_t	**uint_least*N*_t**	**uint_fast*N*_t**	**uintmax_t**	**uintptr_t**
SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR
SCNX*N*	SCNXLEAST*N*	SCNXFAST*N*	SCNXMAX	SCNXPTR

위 매크로들은 질문 12.A에서 설명한 것과 동일한 규칙을 가지며, PRI 대신 SCN으로 시작한다는 것만 다릅니다.

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

main(void)
{
  uint_fast16_t foo;
  uintmax_t i = UINTMAX_MAX;

  sscanf("123", "%" SCNuFAST16, &foo);
  printf("foo is %" PRIoFAST16 ".\n", foo);

  return 0;
}
```

덧붙여 질문 1.1, 11.I, 12.A도 참고하시기 바랍니다.

References [C99] § 7.8.1, § 7.18.1

12.4 `scanf` Problems

Though it seems to be an obvious complement to `printf`, `scanf` has a number of fundamental limitations that lead some programmers to recommend avoiding it entirely.

Q 12.17 `scanf("%d\n", ...)`을 썼더니, 추가적인 줄을 입력하기 전까지는 멈춰버립니다. 왜 그럴까요?

Answer 놀랍게도 `scanf` 포맷 문자열의 `\n`은 newline에 해당하는 것이 아니고, 공백(whitespace) 문자가 나오는 동안, 입력을 읽어서 무시하라는 뜻입니다. (사실, `scanf` format string에 나오는 어떠한(any) 공백 문자라도, 공백 문자를 읽어서 무시하라는 뜻입니다. 게다가 `%d`와 같은 format도 역시 공백 문자를 무시합니다. 따라서 일반적으로 여러분이 공백 문자를 `scanf` format string에 쓸 이유는 없습니다.

따라서 `"%d\n"`의 `\n`은 `scanf`가 공백 문자가 아닌 문자가 나올 때 까지 읽으라는 뜻이 되고, 따라서 공백 문자가 아닌 문자를 찾기 위해 다음 줄까지 읽어버리는 상황이 된 것입니다. 이 경우 단순히 `"%d"`만 쓰면 해결됩니다. (물론 여러분의 프로그램은 수치를 입력받은 다음에 나오는 `\n`을 처리해야 합니다. 질문 12.18을 참고하기 바랍니다.)

`scanf` 함수는 free-format input을 처리하기 위해 디자인되었습니다. 따라서 키보드에서 사용자에게 입력받기 위한 목적으로 쓰이는 경우는 거의 없습니다. "Free format"은 `scanf`가 `\n`를 다른 공백문자와 똑같이 취급한다는 것을 뜻합니다. 따라서 `"%d %d %d"` format은 입력이

```
1 2 3
```

인 경우나

```
1
2
3
```

인 경우 모두 쓸 수 있습니다.

(다른 언어들과 비교해 보면, C, Pascal, 그리고 LISP이 free-format이며, traditional BASIC과 FORTRAN이 free-format이 아닙니다.)

여러분이 꼭 `scanf`가 newline을 다르게 처리하게 만들고 싶다면, "scanset" directive를 쓰면 됩니다:

```
scanf("%d%*[\n]", &n);
```

Scanset은 강력하기는 하지만, `scanf`의 모든 문제를 해결해 주지는 않습니다. 덧붙여 질문 12.20도 참고하시기 바랍니다.

References [K&R2] § B1.3 pp. 245–6
[C89] § 7.9.6.2
[H&S] § 15.8 pp. 357–64

Q 12.18 `scanf("%d", ...)`를 써서 여러 수치를 읽은 다음, `gets()`를 써서 문자열을 읽으려 합니다:

```
int n;
char str[80];

printf("enter a number: ");
scanf("%d", &n);
printf("enter a string: ");
gets(str);
printf("You typed %d and \"%s\"\n", n, str);
```

그런데 컴파일러는 `gets()` 호출을 무시합니다! 왜 그런가요?

Answer 여러분이 위의 프로그램에 데이터를 다음과 같이 입력했다고 가정해 봅시다:

```
42
a string
```

이 때 `scanf`는 42를 읽지만, 그 뒤에 나오는 newline을 읽지 않습니다. 이 newline은 input stream에 남아 있고, `gets()`가 바로 이 newline을 읽게 됩니다. `gets()`는 newline이 나올 때까지의 입력을 문자열로 리턴해 주므로, 바로 빈 줄을 리턴하게 됩니다. 따라서 입력의 두번째 줄, "a string"은 아예 읽히지 않습니다. 만약 여러분이 다음과 같이 입력을 주었다면:

```
42 a string
```

프로그램이 예상한 대로 동작할 것입니다.

일반적으로 `scanf()` 다음에 바로 `gets()`과 같은 다른 입력 루틴을 쓰는 것은 바람직하지 않습니다. `scanf()`의 이런 이상한(peculiar) 방식은 항상 문제를 일으킬 소지가 있기 때문입니다. 따라서 전체 입력을 모두 `scanf`로 받거나 아예 `scanf`를 쓰지 않는 것이 좋습니다.

덧붙여 질문 12.20, 12.23도 참고하시기 바랍니다.

References [C89] § 7.9.6.2
[H&S] § 15.8 pp. 357–64

Q 12.19 `scanf()`의 리턴 값을 조사하면 사용자가 실제로 수치 값을 입력했는지 체크할 수 있기에 좀 더 안전하다고 배웠습니다:

```
int n;

while (1) {
  printf("enter a number: ");
  if (scanf("%d", &n) == 1)
    break;
  printf("try again: ");
}
printf("You typed %d\n", n);
```

그런데, 때때로 무한 루프에 빠지는 경우가 있더군요.[9] 왜 그럴까요?

Answer `scanf()`가 수치 값을 처리할 경우, 수치가 아닌 문자가 나오게 되면 변환을 중지하고 그 문자 데이터를 입력 스트림에 그대로 놔 둡니다. 만약 이 상태에서 또 수치 값을 입력받으려고 `scanf()`를 부르게 되면, 문자 데이터가 있기 때문에 바로 중단합니다. 따라서 수치 값을 처리하는 `scanf`를 루프 안에 둘 경우, 잘못된 입력이 들어가게 되면 무한 루프에 빠지는 경우가 가끔 발생합니다. 예를 들어 사용자가 'x'와 같은 글자를 입력했고, `scanf`가 이를 %d나 %f와 같은 포맷으로 처리할 경우, 바로 중단하고, 다음 `scanf` 호출에선 또 'x'를 같은 방식으로 처리하게 되는 것입니다.

왜 `scanf`가 처리할 수 없는 문자들을 input stream에 그대로 두는지 궁금하게 생각하실 것입니다. 여러분이 공백으로 구분되지 않은, 다음과 같은 입력을 처리해야 한다고 생각해보시기 바랍니다:

```
123CODE
```

이 데이터를 `scanf`를 써서 처리하려 한다면, `"%d%s"`를 쓸 것입니다. 이 때 `scanf`가 매치되지 않은 문자를 input stream에 놓아 두지 않고 제거한다면, 여러분이 받은 데이터는 `"CODE"`가 아닌 `"ODE"`가 될 것입니다. (The problem is a standard one in lexical analysis: When scanning an arbitrary-length numeric constant or alphanumeric identifier, you never know where it ends until you've read "too far." This is one reason that `ungetc` exists.)

덧붙여 질문 12.20도 참고하시기 바랍니다.

References [C89] § 7.9.6.2

[H&S] § 15.8 pp. 357–64

[9] 여러분이 control-C키를 쓸 수 있거나 리부팅할 의사가 있을 때에만 이 코드를 실행해보기 바랍니다.

Q 12.20 왜 대부분의 사람들은 `scanf()`를 쓰지 말라고 할까요? 또 `scanf()`를 쓰지 않는다면 대신 어떤 함수를 써야 하죠?

Answer `scanf()`는 많은 문제를 가지고 있습니다 — 질문 12.17, 12.18, 12.19를 참고하기 바랍니다. 또 `gets()`가 가지고 있는 문제가 %s 포맷을 사용할 때에도 발생합니다 (질문 12.23 참고) — 즉 입력받을 버퍼가 오버플로우가 일어나지 않는다고[10] 보장할 수 없습니다.

일반적으로, `scanf()`는 아주 형식화된(relatively structured, formatted) input을 위해 design된 것입니다. (이름부터 "scan formatted"의 약자입니다). 따라서 주의깊게 사용하면 작업이 실패했는지의 여부는 알려 주지만, 왜, 어떻게 작업이 실패했는지는 알려주지 않습니다. 따라서 `scanf()`에서 에러가 발생했다면, 이를 처리하기가 거의 불가능합니다.

User input은 가장 구조화되지 않은 input이라 할 수 있습니다. 잘 디자인된 user interface는 사용자가 어떤 문자라도 입력할 수 있다는 것을 염두에 두어야 합니다 — 숫자가 필요한 경우에 문자나 구두점을 입력하는 단순한 경우를 포함해서, 사용자가 그냥 Return 키만 친다거나, 바로 EOF를 입력한다거나 하는 모든 경우를 말합니다. 이런 모든 경우를 `scanf`로 처리한 다는 것은 거의 불가능합니다; (`fgets()`와 같은 함수를 써서) 줄 전체를 입력 받은 다음, 이를 `sscanf`와 같은 함수를 써서 원하는 형태로 끊어내는(`strtol`, `strtok`, `atoi`와 같은 함수를 써서) 방법을 쓰는 것이 좋습니다; 덧붙여 질문 12.16, 13.6도 참고하시기 바랍니다.) `sscanf`를 쓴다면, 리턴 값을 검사해서 원하는 만큼 입력을 처리했는지 확인하는 것이 중요합니다.

`scanf`의 이런 단점은 `scanf`에만 적용되는 것이지, `fscanf`나 `sscanf`와는 상관없다는 점을 아셔야 합니다. 문제가 되는 것은 표준 입력이 키보드로 연결되어 있고, `scanf`는 표준 입력을 처리하게 되어 있어서, least constrained인 사용자 입력을 `scanf`가 처리해버린다는 것입니다. 데이터 파일이 formatted인 경우, `fscanf`를 쓰는 것은 좋습니다. `sscanf`를 써서 줄을 parsing하는 것은 (리턴값을 검사하는 경우에) 매우 좋습니다. 왜냐하면, 입력을 취소하거나, 다시 입력을 검사하는 등 여러가지 작업을 할 수 있기 때문입니다.[11]

Note User input은 제멋대로 들어올 수 있기 때문에, user input을 바로 `atoi`를 써서, 수치로 변경하는 것은 좋지 않습니다. 질문 21.11를 참고하기 바랍니다.

References [K&R2] § 7.4 p. 159

[10] 여러분이 `%20s`와 같이 폭을 지정한다면 문제가 해결될 수도 있습니다. 질문 12.15를 참고하기 바랍니다.

[11] It's so easy to regain control, restart the scan, discard the input if it didn't match, etc.

12.5 Other stdio Functions

Q 12.21 sprintf 호출에서 필요한 버퍼의 크기를 미리 계산할 수 있는 방법이 있을까요? 즉, 어떻게 하면 sprintf()에서 버퍼 오버플로우를 피할 수 있을까요?

Answer sprintf에 사용한 포맷이 상대적으로 간단하면 여러분은 필요한 버퍼의 크기를 미리 계산할 수 있습니다. 포맷에 하나 이상의 '%s'가 있다면 포맷 문자열에 사용한 고정된 문자들의 갯수를 직접 세어서 (또는 sizeof 연산자를 써서) 계산해서 그 결과를 집어넣을 문자열에 strlen을 취한 값과 더해서 계산할 수 있습니다. 예를 들면:

```
sprintf (buf, "You typed \"%s\"", answer);
```

이 때의 버퍼 크기는 다음과 같이 계산합니다:

```
int bufsize = 13 + strlen (answer);
```

정수에 대해서는 %d의 출력 결과에 필요한 문자 갯수가 다음 식을 넘지 않습니다:

```
((sizeof (int) * CHAR_BIT + 2) / 3 + 1)   /* +1 for '-' */
```

(CHAR_BIT는 <limits.h>에 정의되어 있음) 그러나 이런 식으로 매번 계산하는 것은 지나치게 조심스러운 것[12]입니다. (위의 식은 수치를 8 진수로 표기한다고 가정했을 때의 계산 결과입니다. 10 진법을 쓴다면 위의 계산 결과와 같거나, 그보다 작은 공간을 차지할 수 있습니다.)

포맷 문자열이 매우 복잡하거나 런타임이 되기 전에는 알 수 없다면 버퍼 크기를 예측하는 것은 sprinf()를 새로 만드는 것처럼 어려운 일입니다. 그리고 복잡한 만큼 에러가 발생하기 쉽습니다 (그래서 권장하지 않습니다). 마지막 방법은 일단 그 포맷과 같은 포맷을 fprintf()를 써서 임시 파일에 저장한 다음, fprintf()의 리턴값을 받거나 그 파일의 크기를 조사하는 방법입니다 (그러나 질문 19.12를 꼭 읽기 바랍니다. 그리고 출력 에러를 생각해야 합니다).

주어진 버퍼가 크지 않을 경우, 오버플로우가 일어나지 않는다는 확실한 보장없이는 sprintf()를 부르기 꺼릴 것입니다. 포맷 스트링을 알 수 있다면 %s를 쓰는 대신 %.Ns(이 때 N은 어떤 수치)를 쓰거나 %.*s를 (덧붙여 질문 12.10도 참고하시기 바랍니다.) 쓸 수 있습니다.

가장 확실한 오버플로우 방지책은 길이를 제한하는 snprintf()를 쓰는 것입니다. 실행 방법은 다음과 같습니다:

[12] over-conservative

```
snprintf(buf, bufsize, "You typed \"%s\"", answer);
```

snprintf()는 많은 stdio 라이브러리에서 (GNU와 4.4BSD 포함) 제공되고 있습니다. 그리고 이 함수는 [C9X] 표준에서 채택될 예정입니다.

[C9X] snprintf()를 쓸 수 있다면 배열의 크기를 미리 계산할 수 있습니다. [C9X] snprintf()는 버퍼에 기록한 문자의 수를 리턴하는 것이 아니라, 버퍼에 기록할 문자의 수를 리턴합니다. 게다가 이때 버퍼의 크기에 0, 또는 버퍼에 널 포인터를 전달할 수 있습니다. 그러므로 다음과 같은 호출을 사용하면:

```
nch = snprintf (NULL, 0, fmtstring, /* 다른 인자들 */);
```

주어진 포맷 스트링(fmtstring)에 필요한 버퍼의 크기를 계산할 수 있습니다.

References [C9X] § 7.13.6.6

Q 12.22 sprintf의 리턴 값이 왜 그리 문제인가요? int 타입을 리턴하나요, 아니면 char * 타입을 리턴하나요?

Answer 표준은 (printf나 fprintf와 같이 write한 글자 수) int를 리턴한다고 말하고 있습니다. 그러나 예전의 어떤 라이브러리는 sprintf가 첫 번째 인자로 들어온 char *를 리턴합니다 (i.e. strcpy의 리턴값과 같은 목적).

References [ANSI] § 4.9.6.5
[C89] § 7.9.6.5
[PCS] § p. 175

Q 12.23 왜 모두들 gets()를 쓰지 말라고 할까요?

Answer fgets()와는 달리 gets()는 버퍼의 크기를 지정하지 않습니다. 따라서 버퍼 오버플로우를 방지할 길이 없습니다 — 머피의 법칙에 의하면 예상되는 입력보다 큰 값이 언젠가는 들어오게 됩니다.[13] 일반적으로는 항상 fgets()를 써야 합니다. (물론 입력은 어떤 특정한 크기 이상 들어올 수 없지만, 실수할 경우를 방지하기 위해[14], 항상 fgets를 쓰는 것이 좋습니다.)

[13] gets()의 단점을 논할때, 1988년 "Internet worm"에 대한 것을 빼 놓을 수 없습니다 — 이 'worm'은 UNIX finger daemon이 gets()를 쓴다는 것을 이용, gets()의 버퍼를 overflow시켜서, 함수의 return address를 바꾸어 코드의 흐름이 다른 곳으로 가도록 만들어 버립니다.

[14] OS가 한번에 들어오는 키보드 입력의 최대 크기를 제한한다고 생각할 수도 있지만, 표준 입력이 redirect된 파일에서 들어올 수도 있습니다.

fgets와 gets의 다른 점 하나는, fgets는 문자열 마지막에 \n을 제거하지 않는다는 것입니다. 그러나 이 newline을 없애기는 아주 쉽습니다. 질문 7.1을 보면 gets 대신 fgets()를 사용한 코드를 볼 수 있습니다.

Note 머피의 법칙(Murphy's Law)에 관한 것은 [Raymond]를 참고하기 바랍니다.

줄의 길이에 상관없이 한 줄을 완벽하게 읽을 수 있는 getline 함수는 질문 21.6을 참고하기 바랍니다.

References [ANSI Rationale] § 4.9.7.2
[H&S] § 15.7 p. 356

Q 12.24 printf를 호출한 다음에 errno를 검사해서 printf가 실패했는지 검사했다고 생각합니다:

```
errno = 0;
printf("This\n");
printf("is\n");
printf("a\n");
printf("test.\n");
if (errno != 0)
  fprintf(stderr, "printf failed: %s\n", strerror(errno));
```

그런데 왜 "printf failed: Not a typewriter"라는 이상한 메시지를 출력할까요?

Answer 많은 stdio 패키지들은 stdout이 터미널일 경우는 특별하게 취급합니다. 따라서 stdout이 터미널인지 검사하기 위해서, 내부적으로 어떠한 연산을 하게 되며, 그 결과 stdout이 터미널이 아닌 경우에는 errno를 ENOTTY로 설정합니다. 따라서 printf가 성공적으로 수행되었다 하더라도, errno에 설정한 값은 그대로 남게 됩니다. 이러한 일은 조금 혼동스럽기는 하지만, 틀렸다고 말할 수는 없습니다. 왜냐하면 errno의 값은 에러가 난 다음에만 의미를 가지기 때문입니다. (좀 더 정확히 말한다면, errno는, errno를 셋팅하는 라이브러리 함수가 에러 코드를 리턴한 경우에만 의미가 있습니다 (meaningful).) 일반적으로 일단 함수의 리턴 값을 검사해야 합니다. 여러 stdio 함수를 호출한 다음 그 동안 연산의 에러를 알고 싶다면 ferror 함수를 쓰면 됩니다. 덧붙여 질문 12.2, 20.4도 참고하시기 바랍니다.

Note 에러가 발생하지 않았을 때에는 errno가 0이라는 것을 보장할 수 없습니다.

References [C89] § 7.1.4, § 7.9.10.3
[CT&P] § 5.4 p. 73
[PCS] § 14 p. 254

Q 12.25 `fgetpos`/`fsetpos`와 `ftell`/`fseek`이 차이가 있나요? `fgetpos`/`fsetpos`는 어디에 쓰는 거죠?

Answer 새로운 `fgetpos`/`fsetpos` 함수는 파일 offset을 나타내기 위해, `typedef`인 `fpos_t`를 사용합니다. 따라서 올바르게 만들어졌다면 아무리 큰 파일이더라도 offset을 표현할 수 있습니다. 이와는 달리 `ftell`과 `fseek`는 `long int`를 쓰기 때문에, offset의 범위가 `long int`의 범위로 제한됩니다. (`long int` type은 $2^{31}-1$보다 큰 수를 가진다고 보장할 수 없습니다. 따라서 파일의 옵셋이 20억 바이트 ($2^{31}-1$)로 제한됩니다. 또한 `fgetpos`/`fsetpos`는 다중바이트(multibyte) 스트림에도 쓸 수 있습니다. 덧붙여 질문 1.4도 참고하시기 바랍니다.

References [K&R2] § B1.6 p. 248
[C89] § 7.9.1, § 7.9.9.1, 7.9.9.3
[H&S] § 15.5 p. 252

Q 12.26 입력을 방출(flush)시킬 방법이 있을까요? 즉 사용자가 미리 입력한 입력을 무시해서, 다음 프롬프트를 출력할 때 이 입력이 출력되지 않도록 하고 싶습니다. `fflush(stdin)`을 쓰면 될까요?

Answer C 언어 표준에 따르면, `fflush()`는 output stream에 대해서만 동작합니다. "flush"라는 정의가 buffering된 문자들을 쓰는 것을 완료시킨다는[15] 것이므로 문자를 취소시키는 것(discard)과는 아무런 관계가 없습니다. 따라서 입력 스트림의 입력을 무시하는 기능과는 전혀 상관이 없습니다.

아직 읽지 않은 문자(unread characters)들을 stdio input stream에서 무시하는(discard) 표준 방법은 존재하지 않습니다. 어떤 컴파일러 회사들은 `fflush(stdin)`이 읽지 않은 문자들을 취소할 수 있도록 라이브러리를 제공하기도 하지만, 이식성이 뛰어난 프로그램을 만들려면 절대로 써서는 안되는 기능입니다. (어떤 stdio 라이브러리는 `fpurge`나 `fabort`를 같은 기능으로 제공하기도 하지만, 마찬가지로 표준은 아닙니다.) 또한 input buffer를 flush하는 것이 꼭 해결책이라고 할 수도 없습니다. 왜냐하면 일반적으로 아직 읽히지 않은 입력은 OS-level input buffer에 존재하기 때문입니다.

[15] To complete the writing of buffered characters

Input을 flush할 방법이 필요하다면 (질문 19.1과 19.2에 나온 것처럼) 시스템 의존적인 방법을 찾아야 할 것입니다. 게다가 사용자가 매우 빠른 속도로 입력하고 있고, 여러분의 프로그램이 그 입력을 무시해버릴 수 있다는 것을 꼭 염두에 두어야 합니다.

또는 `\n`이 나오기 전까지 문자를 읽어서 무시하거나, curses에서 제공하는 루틴인 `flushinp()`를 쓰면 됩니다. 덧붙여 질문 19.1, 19.2도 참고하시기 바랍니다.

References [C89] § 7.9.5.2
[H&S] § 15.2

12.6 Opening and Manipulating Files

Q 12.27 다음과 같이 파일을 여는(open) 함수를 만들었습니다:

```
myfopen(char *filename, FILE *fp)
{
  fp = fopen(filename, "r");
}
```

그런데 다음과 같이 호출하고 나면:

```
FILE *infp;
myfopen("filename.dat", infp);
```

`infp` 변수가 제대로 설정(setting)되지 않습니다. 왜 그럴까요?

Answer C 언어에서 함수들은 항상 자신에게 전달된 인자의 사본을 받습니다. 따라서 함수는 절대로 자신에게 전달된 인자에 값을 지정(assignment)하는 것으로 값을 리턴할 수 없습니다. 질문 4.8을 참고하기 바랍니다.

주어진 질문을 고치려면 다음과 같이 `myfopen` 함수가 `FILE *`를 리턴하도록 고치면 됩니다:

```
FILE *
myfopen(char *filename)
{
  FILE *fp = fopen(filename, "r");
  return fp;
}
```

그 다음 다음과 같이 씁니다.

```
FILE *infp;
infp = myfopen("filename.dat");
```

또는 다음과 같이 myfopen이 FILE *에 대한 포인터[16]를 인자로 받게하면 됩니다:

```
myfopen(char *filename, FILE **fpp)
{
  FILE *fp = fopen(filename, "r");
  *fpp = fp;
}
```

그리고 다음과 같이 씁니다:

```
FILE *infp;
myfopen("filename.dat", &infp);
```

Q 12.28 아주 간단한 fopen 조차도 동작하지 않습니다. 아래 코드에서 잘못된 점이 있나요?

```
FILE *fp = fopen(filename, 'r');
```

Answer 문제는 fopen의 두 번째 인자인 mode는 "r"과 같은 string이어야지, 'r'과 같은 문자가 아니라는 것입니다. 덧붙여 질문 8.1도 참고하시기 바랍니다.

Q 12.29 왜 절대 경로를 써서 파일을 열면 항상 실패할까요? 다음 코드는 동작하지 않더군요:

```
fopen("c:\newdir\file.dat", "r");
```

Answer 두 개의 backslash 문자를 써야 할 것입니다. 질문 19.17을 참고하기 바랍니다.

[16] a pointer to pointer to FILE

Q 12.30 파일의 내용을 고치려고 합니다. 일단 `"r+"` 모드로 파일을 연 다음, 원하는 문자열을 읽고 다시 이 문자를 쓰려고 했으나 제대로 동작하지 않습니다.

Answer 쓰기 전에 `fseek` 함수를 불렀는지 검사해보기 바랍니다. 읽은 다음에, 그 위치에 쓰기 위해서는, 읽은 만큼 파일 위치를 앞쪽으로 옮겨 주어야 합니다. 즉, read/write `"+"` 모드로 연 파일에서 읽기(reading)나 쓰기(writing)을 한 다음에는 반드시 `fseek`나 `fflush`를 써 주어야 합니다. 또 덮어쓰기 위해서는 기존의 문자열과 덮어 쓸 문자열의 길이가 같아야 합니다; 특정 위치에 문자들을 추가하거나(insert) 지우는(delete) 방법은 존재하지 않습니다. 텍스트 파일에서 덮어쓰는 작업을 하면, 파일을 덮어쓴 위치까지 잘려버릴 수도 있습니다[17]. 덧붙여 질문 19.14도 참고하시기 바랍니다.

References [ANSI] § 4.9.5.3
[C89] § 7.9.5.3

Q 12.31 파일 중간에 한 줄을 (또는 레코드) 추가하거나 지우는 방법이 있나요?

Answer 질문 19.14를 참고하기 바랍니다.

Q 12.32 열려진 stream에서 파일 이름을 다시 얻어낼 수 있습니까?

Answer 질문 19.15를 참고하기 바랍니다.

12.7 Redirecting `stdin` and `stdout`

Q 12.33 프로그램 안에서 `stdin`이나 `stdout`을 파일로 'redirect'할 수 있을까요?

Answer `freopen()` 함수를 쓰기 바랍니다. `f()`라는 함수가 있고, 이 함수는 보통 `stdout`으로 데이터를 출력한다고 가정해 봅시다. 이런 경우에는 단순히 `freopen` 함수를 쓰면 됩니다:

```
freopen(file, "w", stdout);
f();
```

(그러나 질문 12.34를 꼭 읽어보시기 바랍니다).

[17] overwriting in text mode may truncate the file at that point

References [ANSI] § 4.9.5.4
[C89] § 7.9.5.4
[H&S] § 15.2. pp. 347–8

Q 12.34 일단 `freopen()`을 쓴 다음, 다시 원래의 `stdout`으로 (또는 `stdin`으로) 복귀할 수는 없나요?

Answer 좋은 해결책이 없습니다. 다시 돌아올 필요가 있을 때, `freopen()`을 쓰는 것은 별로 좋은 방법이 아닙니다. 차라리 따로 output (또는 input) stream 변수를 만들어서 작업해서 `stdout`을 (또는 `stdin`) 건드리지 않는 것이 좋습니다. 예를 들어 다음과 같이 전역 변수를 만들고:

```
FILE *ofp;
```

`printf(...)`를 전부 `fprintf(ofp, ...)`로 바꿉니다. (물론, `putchar`나 `puts`를 썼다면 이 함수들도 바꾸어야 할 것입니다.) 그런 다음, `ofp`가 `stdout`를 가리키도록 하는 등의 작업을 하면 됩니다.

혹시 다음과 같은 코드로 `freopen`을 대신할 수 있다고 생각하실지도 모르겠습니다:

```
FILE *savestdout = stdout;
stdout = fopen(file, "w");    /* WRONG */
```

그 다음, 아래 코드로 `stdout`을 복원할 수 있다고 생각하실지도 모르겠습니다:

```
stdout = savestdout;          /* WRONG */
```

이러한 코드는 동작한다고 말하기 어렵습니다. 왜냐하면 `stdout`은 (또 `stdin`, `stderr`) 보통 상수(constant)이기 때문에 다른 값으로 assign할 수 없기 때문입니다. (그렇기 때문에 `freopen`이라는 함수가 제공되는 것입니다.)

원래의 스트림에 관한 정보를 `freopen()`을 쓰기 전에 복사해 두었다가 복원시키는 방법도 있지만, 이는 시스템 의존적인, `dup()`과 같은 함수를 쓰기 때문에 이식성이 (portability) 떨어지며, 비안정적입니다 (unreliable.)

어떤 시스템에서는 controlling terminal을 (질문 12.36 참고) 직접 open하는 방법을 제공하기도 하지만, 이는 여러분이 원하는 것이 아닐 것입니다. 왜냐하면 원래의 input 또는 output이 (즉 `freopen`을 부르기 전의 `stdin`이나 `stdout`의 값) command line에서 redirect되어 들어올 수 있기 때문입니다.

어떤 subprogram을 실행시키고 그 결과를 capture하고 싶다면, `freopen`은 제대로 동작할 수 없습니다; 질문 19.30을 대신 참고하기 바랍니다.

Q 12.35 프로그램 안에서 표준 입력이나 표준 출력이 redirect되었는지 확인할 수 있을까요? 다시 말해 command line에서 "<"나 ">"을 썼는지 확인할 수 있을까요?

Answer 먼저 무엇 때문에 그러한 방법이 필요한 것인지 잘 생각해 보시기 바랍니다. 만약 프로그램에 전달한 입력 파일이 없을 때, `stdin`으로 데이터를 받고자 한다면 간단히 `argv`를 써서 할 수 있습니다 (질문 20.3 참고). 또는 파일 이름을 쓰는 대신 "-"와 같은 placeholder를 쓰게 만들 수 있습니다. 입력이 interactive terminal에서 온 경우가 아닌 경우, prompt를 출력하게 않도록 만들고 싶다면, 시스템에 따라서 (예를 들어 UNIX나 MS-DOS에서) `isatty(0)`이나 `isatty(fileno(stdin))`을 써서 terminal인지 아닌지 알아낼 수 있습니다.

Q 12.36 "more"와 같은 프로그램을 만들려고 합니다. `stdin`이 redirect되었다면 어떻게 keyboard를 다시 쓸 수 있을까요?

Answer 이런 일을 할 수 있는 portable한 방법은 없습니다. UNIX에서는 special file인 `/dev/tty`를 열면 됩니다. 또, MS-DOS에서는, 파일 `CON`을 열거나, `getch`처럼 BIOS call을 이용하는 루틴을 쓰면 input이 redirect되었는지와는 상관없이 keyboard를 쓸 수 있습니다.

Q 12.36b 출력을 동시에 두곳으로, 예를 들면 스크린과 파일로 보낼 수 있는 방법이 있을까요?

Answer 없습니다. 대신 여러분이 모든 것을 두 번 반복하는 `printf` 스타일의 함수를 만들어 쓰면 됩니다. 질문 15.5를 참고하기 바랍니다.

12.8 "Binary" I/O

보통의 stream은 출력할 수 있는 text로 이루어져 있고, 저수준의 운영체제에서 쓰는 어떤 convention에 맞도록 변환(translation)을 할 수 있는 것으로 여겨지고 있습니다. 이러한 translation없이, 정확히 바이트 단위로 read/write를 할려면 "binary" I/O를 써야 합니다.

Q 12.37 메모리와 파일에서, `fprintf`나 `fscanf`가 쓰는 formatted string이 아닌, 한 번에 한 바이트씩 읽거나 쓰려고(write) 합니다. 어떻게 하면 될까요?

Answer 이런 일을 대개 "binary" I/O라고 부릅니다. 먼저 `fopen`을 부를때 (`"rb"`나 `"wb"`와 같은) `"b"` modifier를 썼는지 (질문 12.38 참고) 확인하기 바랍니다.

그 다음, `&`와 `sizeof` operator를 써서 몇 바이트를 transfer할 것인지 확인하기 바랍니다. 보통 `fread`나 `fwrite`같은 함수를 써서 작업할 수 있습니다; 예제는 질문 2.11을 보기 바랍니다.

`fread`나 `fwrite` 자체가 binary I/O를 수행하는 함수는 아닙니다. 일단 binary mode로 파일을 열었다면, 어떤 I/O 함수라도 binary I/O를 할 수 있습니다 (질문 12.42의 예제 참고); Text mode로 파일을 열었다해도 원한다면 `fread`나 `fwrite`를 쓸 수 있습니다.

마지막으로, binary data file은 portable하지 않습니다; 질문 20.5 참고. 덧붙여 질문 12.40도 참고하시기 바랍니다.

Q 12.38 바이너리 데이터 파일을 적절하게 읽을 수 있는 방법을 알려주세요. 때때로 0x0a와 0x0d 값이 멋대로 들어오고 데이터가 0x1a을 포함할 때 EOF가 리턴되는 경우가 있습니다.

Answer Binary data를 읽어 들이려면 `fopen()`을 호출할 때 `"rb"` mode를 써서 텍스트 파일 변환(text file translation)이 일어나지 않도록 해야 합니다. 비슷하게, 바이너리 데이터를 쓰기 위해서는 `"wb"`를 써야 합니다. (UNIX와 같이 text와 binary 파일을 구별하지 않는 운영체제에서는 `"b"`를 쓸 필요는 없지만, 썼다고 하더라도 문제될 것은 없습니다.)

텍스트/바이너리의 차이는 파일을 열 때에만 적용됩니다. 일단 파일이 열린 다음에는 어떤 I/O 루틴을 써도 상관없습니다. 덧붙여 질문 12.40, 12.42, 20.5도 참고하시기 바랍니다.

References [ANSI] § 4.9.5.3
[C89] § 7.9.5.3
[H&S] § 15.2.1 p. 348

Q 12.39 Binary file에 쓰기 위한 일종의 "filter"를 만들고 있습니다. 그런데 `stdin`과 `stdout`은 이미 text stream으로 열려 있는 상태라서 문제가 됩니다. 어떻게 하면 이 stream을 binary mode로 바꿀 수 있을까요?

Answer 이런 일을 하기 위한 표준 방법은 없습니다. UNIX와 같은 OS는 text와 binary를 따로 구별하지 않으므로, mode를 바꿀 필요가 없습니다. 어떤 MS-DOS 컴파일러는 `setmode`라는 함수를 제공하기도 합니다. 다른 OS의 경우에는 여러분 스스로 찾아야 할 것입니다.

Q 12.40 Text I/O와 binary I/O의 차이점이 뭔가요?

Answer Text mode의 파일은 출력 가능한(printable) 문자로 (탭 같은 문자 포함) 이루어져 있는 것이 상식입니다. Stdio 라이브러리에 있는 함수는 (`getc`, `putc`와 같은 모든 함수 포함) OS에서 쓰는 end-of-line 문자 표현 방법을 하나의 `\n` 문자로 바꾸어서 처리합니다. 따라서 C 프로그램에서 OS에서 사용하는 end-of-line 문자 표현 방식을 신경쓸 필요가 없습니다: C 프로그램에서 `'\n'` 문자를 쓰면(write), stdio 라이브러리는 이 문자를 OS에서 사용하는 end-of-line 문자 표현 방식으로 바꾸어 쓰며(write), 읽을 때에는 OS가 사용하는 end-of-line 문자 표현 방식을 보면 단순히 `\n`을 프로그램에 돌려줍니다.[18]

이와는 달리 binary mode에서는 프로그램에서 읽고 쓰는(write) 바이트와 파일 저장 방식과 완전히 같습니다. 즉 어떠한 변환도 이루어지지 않습니다. (MS-DOS 시스템에서는 control-Z를 end-of-file 문자로 해석하는 것도 이루어지지 않습니다.)

Text mode translation은 파일의 크기에도 영향을 끼칩니다. 왜냐하면 프로그램에서 처리하는 문자가 file에 저장될 문자와 1:1로 match될 필요가 없기 때문입니다. 같은 이유에서 `fseek`나 `ftell`이 파일에서 정확한 byte offset을 나타낼 이유도 없습니다. (엄격히 말해서 text mode에서는 `fseek`나 `ftell`이 리턴하는 값을 프로그래머가 직접 해석하려고 하면 안됩니다: `ftell`이 리턴하는 값은 `fseek`의 인자로만 써야 하며, 또한 `ftell`이 리턴하는 값만이 `fseek`의 인자로 쓰여야 합니다.)

Binary mode에서는 `fseek`와 `ftell`이 byte offset을 나타냅니다. 그러나 어떤 시스템에서는 완전한 레코드를 만들기 위해 여러 개의 null byte를 파일 뒤에 추가할 수도 있습니다.

덧붙여 질문 12.37, 19.12도 참고하시기 바랍니다.

References [ANSI] § 4.9.2; [C89] § 7.9.2
[ANSI Rationale] § 4.9.2
[H&S] § 15 p. 344, § 15.2.1 p. 348

[18] 어떤 시스템은 text file을 space로 구분되는(space-padded) record로 저장합니다. 이러한 시스템에서 text mode로 한 줄을 읽을 때에는 뒤따라는 공백 문자(trailing space)들을 없앨 필요가 있습니다. 따라서 프로그램에서 여러 공백 문자를 썼다고 해도, 기록되지 않을 수도 있습니다.

Q 12.41 Structure를 어떻게 파일에서 읽거나(read) 쓸(write) 수 있을까요?

Answer 질문 2.11을 참고하기 바랍니다.

Q 12.42 오래된 binary data file에서 data를 읽어오려면 어떻게 해야 할까요?

Answer Word size와 byte-order 차이, floating-point format, structure padding 때문에 어렵습니다. 이런 것들을 극복하려면 아마도 바이트 단위로 data를 읽거나 써야(write) 할지도 (shuffling 또는 rearranging한 다음) 모릅니다. (이런 작업이 항상 나쁜 것은 아닙니다. 왜냐하면 코드의 portability를 높여주며, 여러분이 확실히 제어할 수 있기 때문입니다.)

예를 들어 여러분이 문자와 32-bit integer, 16-bit integer로 이루어진 structure를 주어진 `fp`에서 다음 structure 안으로 읽을 필요가 있다고 가정해 봅시다:

```
struct mystruct {
  char c;
  long int i32;
  int i16;
}
```

다음과 같은 코드를 쓰면 해결될 것입니다:

```
s.c = getc(fp);

s.i32 = (long)getc(fp) << 24;
s.i32 |= (long)getc(fp) << 16;
s.i32 |= (unsigned)(getc(fp) << 8);
s.i32 |= getc(fp);

s.i16 = getc(fp) << 8;
s.i16 |= getc(fp);
```

이 코드는 `getc`가 8-bit 문자를 읽고 데이터가 최상위 바이트가 우선하여 (most significant byte first, "big endian") 저장되어 있다고 가정했습니다. `(long)` 타입으로 casting한 것은 16-bit 또는 24-bit shift 연산이 `long` 값에 확실히 동작하도록 (질문 3.14 참고) 보장해주며, `(unsigned)`로 casting한 것은 부호 확장(sign extension)을 막기 위한 것입니다. (일반적으로 이러한

코드를 작성할 때에는 모든 data type을 `unsigned`로 하는 것이 좋습니다. 그러나 먼저 질문 3.19를 참고하기 바랍니다.)

Structure의 내용을 파일에 쓰는(write) 코드는 다음과 같습니다:

```
putc(s.c, fp);

putc((unsigned)((s.i32 >> 24) & 0xff), fp);
putc((unsinged)((s.i32 >> 16) & 0xff), fp);
putc((unsigned)((s.i32 >> 8) & 0xff), fp);
putc((unsigned)(s.i32 & 0xff), fp);

putc((s.i16 >> 8) & 0xff, fp);
putc(s.i16 & 0xff, fp);
```

덧붙여 질문 2.12, 12.38, 16.7, 20.5도 참고하시기 바랍니다.

Chapter 13

Library Functions

예전에는, 특정 run-time library는 C 언어의 표준의 한 부분이 아니었습니다. ANSI/ISO Standard C의 출현으로 대부분의 전통적인 (traditional) run-time library는 (Chapter 12의 stdio 함수들을 포함) 표준이 되었습니다.

특별히 중요한 라이브러리 함수들은 각각 다른 section에 설명되었습니다; 메모리 할당에 관련된 `malloc`과 `free`에 관한 것은 7 장에 설명되었으며, `<stdio.h>`에 나온 "standard I/O" 함수들은 Chapter 12에 설명되었습니다. 이 chapter는 다음과 같이 나누어집니다:

String Function: 13.1–13.7

Sorting: 13.8–13.11

Date and Time: 13.12–13.14

Random Numbers: 13.15–13.21

Other Library Functions: 13.22-13.28

마지막 몇 개의 질문들은 (13.25부터 13.28까지) link시 문제가 되는 (예를 들면, "undefined external" 에러) 것을 다루었습니다.

13.1 String Functions

Q 13.1 수치를 문자로 바꾸고 싶은데 (`atoi` 함수의 반대 기능), `itoa()`라는 함수가 있나요?

Answer 그냥 `sprintf()`를 쓰시면 됩니다. (`sprintf`가 너무 복잡하고, 큰 비용(시간이나 크기 면에서)이 든다고 생각할 지 모르나, 사실 효과적으로 동작합니다.) 질문 7.5a에 대한 답변에 나온 예를 보기 바랍니다. 질문 12.21을 참고하기 바랍니다.

`long`이나 실수도 `sprintf()`를 써서 바꿀 수 있습니다 (각각 `%ld`와 `%f`를 쓰면 됨); 즉, `sprintf`를 `atol`이나 `atof`의 반대 역할을 하는 함수라고 생각할 수 있습니다. 게다가, 어떤 식으로 출력할 것인지도 제어할 수 있습니다. (그렇기 때문에, C 언어는 `itoa` 식의 함수보다, `sprintf`를 제공하는 것입니다.)

꼭 `itoa`를 직접 만들어야 한다면, 다음과 같은 사항들을 고려해야 합니다:

- K&R은 단순한 버전의 `itoa`를 제공합니다.
- 돌려줄 버퍼를 어떻게 할당할 것인지 생각해 봐야 합니다; 질문 7.5를 참고 바랍니다.
- 단순히 만들었다면, 보통 가장 적은 음수(`INT_MIN`, 보통 -32,768 또는 -2,147,483,648)를 올바르게 처리하지 못할 수도 있습니다.

질문 12.21과 20.10도 참고 바랍니다.

References [K&R1] § 3.6 p. 60
[K&R2] § 3.6 p. 64

Note C99 표준에 새로 추가된 `long long` 타입을 쓰려면, `ll` length modifier를 써야 합니다. 즉 `long`을 출력하기 위해 `%ld`를 쓴 것처럼, `%lld`를 씁니다.

마찬가지로, `size_t` 타입의 값을 쉽게 출력할 수 있도록 `printf`, `scanf` 형태의 함수에 `z` modifier를 추가했습니다. 정수 출력에 쓰이는 conversion인, `d`, `i`, `o`, `u`, `x`, `X`에 `z`를 붙여서 `size_t`를 쉽게 출력할 수 있습니다. 예를 들면:

```
size_t sz;
...
printf("%zu\n", sz);
```

References [C99] § 7.19.6.1, § 7.19.6.2
[H&S2002] § 15.11.6

Q 13.2 왜 `strncpy()`는 대상 문자열에 항상 `\0`을 써 주지 않는 것일까요?

Answer strncpy()의 원래 개발 목적은 고정 크기를 갖고, \0으로 끝나지 않는 "문자열"[1]도 처리할 수 있게 하는 것이 그 목적이었습니다. 따라서, 다른 목적으로 strncpy()를 쓸 때에는 좀 번거로운 작업이 필요합니다; (strncpy()의 한 가지 단점은 여분의 공간에 0을 계속 기록한다는 것입니다.) 복사한 문자열이 저장된 버퍼의 끝에 수동으로 '\0'을 채워주어야 할 필요가 있습니다. 그래서 그리 자주 쓰이는 함수가 아닙니다. 그래서, strncpy() 대신 strncat을 쓰기도 합니다: 대상 버퍼가 비어있는 경우, strncat은 여러분이 strncpy에서 기대한 작업을 수행해 줍니다:

```
*dest = '\0';
strncat(dest, source, n);
```

위 코드는 최대 n개의 문자를 복사하며, 항상 마지막에
0을 붙여 줍니다.

또 다른 방법은 다음과 같습니다:

```
sprintf (dest, "\%.*s", n, source);
```

정확하게는 위 코드는 n이 509 이하일때에만 동작합니다.

문자열이 아닌 어떤 크기의 바이트를 복사하고자 한다면 strncpy() 대신 memcpy()를 쓰는 것이 더 효과적입니다.

Q 13.3 다른 언어에서 제공하는 "substr" (문자열에서 부분 문자열을 빼내는) 함수가 있나요?

Answer 없습니다. (그 이유 중 하나는, C 언어에는 문자열을 취급하는 전용의 데이터 타입이 없기 때문입니다. 질문 7.2와 Chapter 8 참고하기 바랍니다.)

원본 문자열의 POS번째에서 시작해서 길이가 LEN인 부분 문자열을 꺼내기 위해서 다음과 같이 할 수 있습니다:

```
char dest[LEN + 1];
strncpy(dest, &source[POS], LEN);
dest[LEN] = '\0';           /* ensure \0 termination */
```

[1] 예를 들어, 초창기 C 컴파일러와 링커는 내부적으로 처리하는 심볼 테이블에 8개의 글자를 갖는 고정된 문자열을 썼습니다. 지금도 어떤 버전의 UNIX는 14개의 글자를 갖는 고정된 문자열을 쓰기도 합니다. strncpy는 이런 문자열을 위해, 짧은 문자열을 복사할 때, 남아있는 공간을 '
0'으로 채워줍니다. 따라서 이런 형태의 문자열을 비교할 때, 좀 더 효과적으로 처리할 수 있습니다. 왜냐하면, '
0'을 찾은 계산 대신, 아무 생각 없이 n개의 바이트를 비교하면 되기 때문입니다.

또, 질문 13.2에서 다룬 것처럼 다음과 같이 할 수 있습니다:

```
char dest[LEN + 1] = "";
strncat(dest, &source[POS], LEN);
```

다음과 같이 포인터 형태로 할 수도 있습니다:

```
strncat(dest, source + POS, LEN);
```

(정의에 의해, `&source[POS]`는 `source + POS`와 완벽히 같은 표현입니다; Chapter 6 참고)

Q 13.4 모든 문자열을 대문자로 또는 소문자로 바꾸려면 어떻게 하죠?

Answer 어떤 라이브러리들은 이 역할을 위해, `strupr`와 `strlwr`, 또는 `strupper`와 `strlower`를 제공합니다만, 이식성도 없고, 표준도 아닙니다. 헤더 파일 `<ctype.h>`에 정의되어 있는 매크로인 `toupper`와 `tolower`를 써서 이 기능을 수행하는 함수를 만들면 됩니다; 질문 13.5를 참고 바랍니다. (매우 간단하지만, 만들 함수가 전달받은 문자열을 고칠 것이냐, 아니면 따로 공간을 할당하고 변경된 문자열을 그 곳에 저장할 것이냐는 미리 생각해 두어야 합니다; 질문 7.5를 참고 바랍니다.)

(다국적(multinatinoal) 문자 셋을 쓸 때에는 문자 또는 문자열을 대문자 또는 소문자로 바꾸는 것은 더 복잡합니다.)

Q 13.5 어떤 버전의 `toupper()`에, 대문자를 인자로 줄 경우, 이상하게 동작합니다. 이런 것을 예상해서인지 `toupper()`를 부르기 전에 `islower()`를 부르는 코드도 보았습니다.

Answer 오래된 버전의 `toupper()`와 `tolower()`의 경우, 변환이 필요없는 값이 인자로 전달되면, 제대로 동작하지 않습니다. (즉 알파벳이 아닌 문자나, 또는 이미 변환이 된 알파벳인 경우). 그래서 오래된 (또는 이식성이 아주 높은) 코드는 `toupper` 또는 `tolower`를 부르기 전에, `islower`와 `isupper`를 부르는 경향이 있습니다.

그러나, C 표준은 이러한 함수들이 어떤 값이 전달되더라도 안전하게 동작한다고 말하고 있습니다. 즉, 변경이 필요없는 문자들은 변경되지 않습니다.

References [ANSI] § 4.3.2
[C89] § 7.3.2
[H&S] § 12.9 pp. 320–1
[PCS] p. 182

Q 13.6 문자열 내용을 공백 문자를 기준으로 구별된 단어들로 끊어내고 싶습니다. `main()`이 `argc`와 `argv`를 처리하는 것처럼 할려면 어떻게 해야 할까요?

Answer 단어 ("token") 단위로, 문자열을 끊어주는 표준 함수 `strtok()`이 있습니다. 물론, 이 함수가 이러한 모든 일을 다 처리할 수 있는 것은 아닙니다. (예를 들어, 인용(quoting)된 문자열을 제대로 처리하기 힘듭니다.) 아래에 각각 필드를 얻어서 출력하는 예를 보입니다:

```
#include <string.h>
char string[] = "this is a test";
  /* not char *; see Q16.6 */
char *p;
for (p = strtok(string, " \t\n"); p != NULL;
                    p = strtok(NULL, " \t\n"))
  printf("\"%s\"\n", p);
```

또, 아래는 `argv`를 한꺼번에 만들어 주는 함수입니다:

```
#include <ctype.h>

int
makeargv(char *string, char *argv[], int argvsize)
{
  char *p = string;
  int i;
  int argc = 0;

  for (i = 0; i < argvsize; i++) {
    /* skip leading whitespace */
    while (isspace(*p))
      p++;
    if (*p != '\0')
      argv[argc++] = p;
```

```
      else {
        argv[argc] = 0;
        break;
      }

      /* scan over arg */
      while (*p != '\0' && !isspace(*p))
        p++;
      /* terminate arg: */
      if (*p != '\0' && i < argvsize - 1)
        *p++ = '\0';
    }
    return argc;
  }
```

makeargv는 다음과 같이 호출합니다:

```
  char *av[10];
  int i, ac = makeargv(string, av, 10);
  for (i = 0; i < ac; i++)
    printf("\"%s\"\n", av[i]);
```

만약 구분(separator) 문자가 중요하다면 — 즉, 두 개의 탭이 동시에 나올 때, 한 필드가 생략되었다는 것을 알리고 싶으면 — strchr 함수를 쓰는 것이 더 편할 수 있습니다:

```
  #include <string.h>

  char *p = string;

  while (1) {     /* break in middle */
    char *p2 = strchr(p, '\t');
    if (p2 != NULL)
      *p2 = '\0';
    printf("\"%s\"\n", p);
    if (p2 == NULL)
      break;
    p = p2 + 1;
  }
```

여기에 나온 모든 코드는 원본 문자열에, 각 필드가 끝날 때마다
0을 넣어서 각 필드를 구분할 수 있게 만듭니다. 따라서 원본 문자열이 나중에 다시 필요하다면, 이 곳에 있는 코드를 실행하기 전에 먼저 복사해 두어야 합니다.

References [K&R2] § B3 p. 250
[ANSI] § 4.11.5.8
[C89] § 7.11.5.8
[H&S] § 13.7 pp. 333–4
[PCS] p. 178

Q 13.7 와일드 카드(wildcard)와 정규식(regular expression)을 처리하려고 합니다.

Answer 다음과 같은 차이점을 먼저 알아두기 바랍니다:

- 우선 regular expression은 `ed`나 `grep`과 같은 UNIX 유틸리티에서 자주 쓰입니다. Regular expression에서, 마침표(`.`)는 아무런 글자 하나에 매치(match)되며, `.*`는 길이에 상관없이 여러 문자열에 매치됩니다. (물론, regular expression에는 이보다 훨씬 더 많은 기능이 있습니다.)
- Filename wildcard는 대부분 OS에서 자주 쓰이며, 여러가지 버젼이 존재합니다. 그러나 보통 `?` 문자는 아무런 글자 하나에 매치되며, `*`는 길이에 상관없이 여러 문자열에 매치됩니다.

정규식 매칭에 사용되는 패키지는 많습니다. 그리고 대부분의 패키지들은 두 가지 함수를 사용합니다: 하나는 정규식을 "컴파일(compile)"하기 위한 것이고, 다른 하나는 이 "컴파일된" 정규식과 문자열을 비교해주는(execute) 함수입니다. 먼저 헤더 파일 `<regex.h>`나 `<regexp.h>`가 있는지, 그리고 `regcmp/regex`, `regcomp/regexec`, `re_comp/re_exec` 함수가 존재하는 지 (이 함수들은 대개 각각의 독립적인 라이브러리 형태로 제공됩니다.) 체크해보시기 바랍니다. 인기있고, 공개인 Henry Spencer씨의 regexp 패키지를 `ftp.cs.toronto.edu`에서 `pub/regexp.shar.Z`로 얻을 수 있습니다. GNU 프로젝트에서는 rx라고 하는 패키지를 제공합니다. 덧붙여 질문 18.16도 참고하시기 바랍니다.

Filename wildcard matching은 (때때로 "globbing"이라고 함) 각각의 시스템에 따라 다르게 이루어집니다. UNIX에서는 와일드 카드를 shell이 처리해 주므로, 각각의 프로그램들은 이 와일드 카드에 대해 전혀 고려하지 않아도 됩니다. 그러나 MS-DOS에서는 shell이라 할 수 있는 command interpreter가 이를 처리해 주지 않으므로, (컴파일러가 제공하는) 와일드 카드를 처리해주는 특별한 오브젝트 파일과 함께 링크해야 합니다. (이 오브젝트 파일은

agrv를, 와일드 카드 처리한 다음의 상태로 만들어 줍니다) 그리고 (MS-DOS와 VMS를 포함한) 대부분의 시스템에서는 와일드카드로 파일을 리스팅해주거나, 파일을 open하는 시스템 서비스들을 제공합니다. 먼저 컴파일러/라이브러리의 문서를 참고하기 바랍니다. 덧붙여 질문 19.20, 20.3도 참고하시기 바랍니다.

아래는 Arjan Kenter씨가 제공한 간단한 형태의 wildcard를 처리하는 함수입니다:

```
int match(char *pat, char *str)
{
  switch (*pat) {
  case '\0':  return !*str;
  case '*':   return match(pat + 1, str) ||
                     *str && match(pat, str + 1);
  case '?':   return *str && match(pat + 1, str + 1);
  default:    return *pat == *src &&
                     match(pat + 1, str + 1);
}
```

이 함수를 `match("a*b.c", "aplomb.c")`로 부르면, 1을 리턴합니다.

References Schumacher, ed., *Software Solutions in C* § 3 pp. 35–71

13.2 Sorting

Q 13.8 `qsort()` 함수를 써서 문자열의 배열을 (array of strings) 정렬하려고 합니다. `strcmp()` 함수를 비교 함수로 사용했는데 동작하지 않습니다.

Answer 질문하신 "문자열의 배열 (array of strings)"은 아마도 "문자에 대한 포인터의 배열 (array of pointers to char)"를 의미한 것일 것입니다. `qsort`에 전달되는 비교 함수는 정렬하고자 하는 오브젝트에 대한 포인터를 인자로 받아야 합니다. 이 경우, 문자에 대한 포인터를 가리키는 포인터를 입력 받아야 합니다. 그런데 `strcmp()`는 단순히 문자를 가리키는 포인터를 입력 받습니다. 그러므로 `strcmp()`를 직접 비교 함수로 사용할 수는 없습니다. 다음과 같이 wrapper 함수를 만들어야 합니다:

```
/* compare strings via pointers */
int pstrcmp(const void *p1, const void *p2)
```

```
{
  return strcmp(*(char * const *)p1, *(char * const *)p2);
}
```

비교 함수의 인자는 "범용 포인터"인 `const void *`이어야 합니다. 그리고 함수 내부에서 이를 원하는 형태로 (예를 들면 문자를 가리키는 포인터) 바꾸어 씁니다. 즉 이 타입의 인자는 함수 내부에서 원래의 타입인 `char **`로 바뀌어지고, 다시 실제로 가리키는 타입인 `char *`를 얻어서 `strcmp` 함수에 전달됩니다. (ANSI 이전 컴파일러에서 쓸 때에는 `void *` 대신에 `char *`를 쓰고, 모든 `const`를 빼면 됩니다.)

실제 `qsort`는 다음과 같이 부릅니다:

```
#include <stdlib.h>
char *strings[NSTRINGS];
int nstrings;

/* nstrings cells of strings[] are to be sorted */
qsort(strings, nstrings, sizeof(char *), pstrcmp);
```

([K&R2] § 5.11 pp. 119–20에 나온 것을 혼동하면 안됩니다. 그것은 표준 함수인 `qsort`와는 관계없으며, `char *`와 `void *`가 서로 같다는 불필요한 암묵적인 가정을 하고 쓴 것입니다.)

`qsort` 비교 함수에 대한 더 자세한 정보는 — 어떻게 선언되고 어떻게 불리워지는가에 대한 — 질문 13.9를 보기 바랍니다.

References [ANSI] § 4.10.5.2
[C89] § 7.10.5.2
[H&S] § 20.5 p. 419

Q 13.9 Structure에 대한 배열을 `qsort()`로 정렬하려 합니다. 제가 만든 비교 함수는 이 structure에 대한 포인터를 받도록 했지만, 컴파일러는 `qsort`에 잘못된 타입의 비교 함수가 전달되었다고 불평합니다. 어떻게 함수 포인터를 캐스팅해야 컴파일러가 불평하지 않을까요?

Answer 질문 13.8에서 다룬 것처럼, 캐스팅은 비교 함수 안에서 이루어져야 합니다. 즉 비교 함수는 "generic pointer"인 `const void *` 타입을 인자로 받도록 만들어야 합니다. 예를 들어 아래처럼 structure를 만들었다고 합시다:

```
struct mystruct {
  int year, month, day;
}
```

이 때, 비교 함수는 다음과 같이 만들어야 합니다:

```
int mystructcmp(const void *p1, const void *p2)
{
  const struct mystruct *sp1 = p1;
  const struct mystruct *sp2 = p2;

  if (sp1->year < sp2->year) return -1;
  else if (sp1->year > sp2->year) return 1;
  else if (sp1->month < sp2->month) return -1;
  else if (sp1->month > sp2->month) return 1;
  else if (sp1->day < sp2->day) return -1;
  else if (sp1->day > sp2->day) return 1;
  else return 0;
}
```

(Generic 포인터를 `struct mystruct` 포인터로 변경하는 것은, 초기화할 때인 `sp1 = p1`, `sp2 = p2`에서 이루어졌습니다; `p1`과 `p2`가 `void` 포인터이기 때문에, 컴파일러는 이 conversion을 자동으로 해 줍니다. ANSI 이전의 컴파일러에서 쓰려면 강제로 캐스팅하고, `char *` 포인터를 쓰면 됩니다. 질문 7.7 참고)

이런 형태의 `mystructcmp`를 쓰기 위해, 다음과 같이 `qsort`를 부릅니다:

```
#include <stdlib.h>
struct mystruct dates[NDATES];
int ndates;
/* ndates cells of dates[] are to be sorted */
qsort(dates, ndates, sizeof(struct mystruct),
      mystructcmp);
```

만약, structure를 가리키는 포인터를 정렬하고 싶다면, 질문 13.8에 나온 것처럼 indirection이 필요합니다. 이 때 비교 함수는 다음과 같이 시작합니다:

```
int myptrstructcmp(const void *p1, const void *p2)
{
```

```
    struct mystruct *p1 = *(struct mystruct * const *)p1;
    struct mystruct *p2 = *(struct mystruct * const *)p2;
```

그리고, 다음과 같이 부릅니다:

```
  struct mystruct *dateptrs[NDATES];
  qsort(dateptrs, ndates, sizeof(struct mystruct *),
                          myptrstructcmp);
```

qsort 함수에서 쓰이는 비교 함수에 왜 이런 포인터 변환이 필요한지 이해하기 위해서 (그리고 왜 qsort를 부를때 함수 포인터를 캐스팅하는 것이 불가능한지) qsort가 어떤 식으로 동작하는지 알 필요가 있습니다: qsort는 정렬할 대상이 어떤 타입인지, 어떤 식으로 표현되어 있는지 전혀 알지 못합니다. qsort는 단지, 전달된 조그만 메모리 블럭(little chunks of memory)들을 섞어 줄 뿐입니다. (이 메모리 블럭에 대해 알고 있는 것은, qsort의 세번째 인자로 받은 블럭의 크기입니다.) 두 개의 블럭을 교환(swap)해야 할 경우, qsort는 여러분이 만든 비교 함수를 부릅니다. (실제로 교환하는 것은 memcpy가 하는 것과 같은 방식으로 이루어집니다.)

qsort는 어떤 타입인지 알지 못하는, 일반적인 타입에 대한 정렬을 하므로, 정렬 대상을 가리키기 위해, generic 포인터를 (void *) 씁니다. qsort가 비교 함수를 부를때, 여러분이 만든 비교 함수는 void *를 인자로 받도록 만들어져 있어야 하며, 실제 정렬 대상에 대해 어떤 작업을 (예를 들어 실제 비교를 하기 위해) 수행하기 위해서, 인자로 받은 포인터를 원하는 타입으로 다시 변경해야 합니다. 어떤 시스템에서는 void 포인터가 structure 포인터와 서로 다른 표현 방식을 씁니다; 예를 들어, 실제 포인터의 길이가 다를 수 있으며, 내부 표현 방식이 다를 수 있습니다. (그렇기 때문에 캐스팅이 필요합니다.)

어떤 Structure에 대한 배열을 정렬한다고 가정해봅시다. 그리고 비교 함수가 이 structure에 대한 포인터를 받는다고 하면 다음과 같습니다:

```
  int mywrongstructcmp(struct mystruct *,
                       struct mystruct *);
```

만약 다음과 같이 qsort를 부르게 되면:

```
  qsort(dates, ndates, sizeof(struct mystruct),
        (int (*)(const void *,
                 const void*)mywrongstructcmp);
        /* WRONG */
```

이 것은, 컴파일러가 "qsort에 전달된 비교 함수가 제대로 동작하지 않을 수 있다"고 경고하는 것만을 없애 줄 수 있을 뿐입니다. 여기서 사용한 캐스팅은 실제 qsort가 여러분의 비교 함수를 부를 때에는 이미 알 수 없는 것입니다: 즉, qsort는 const void * 타입의 인자를 만들 것이고, 여러분이 만든 비교 함수는 이 인자를 반드시 받을 수 있어야 합니다. 어떠한 메카니즘도, qsort가 mywrongstructcmp를 부를 때, void 포인터를 struct mystruct 포인터로 바꿔 줄 수 없습니다.

따라서, 일반적으로 위와 같이 단순히 컴파일러 경고를 없애기 위한 캐스팅은 매우 나쁜 방법입니다. 컴파일러 경고는 보통 여러분에게 어떤 사항을 알려줄려고 나온 것이기 때문에, 여러분이 어떤 식으로 동작하는지 잘 모르고 이 경고를 무시한다면, 그 결과 매우 위험한 상태가 될 수 있습니다. 덧붙여 질문 4.9도 참고하시기 바랍니다..

References [ANSI] § 4.10.5.2
[C89] § 7.10.5.2
[H&S] § 20.5 p. 419

Q 13.10 'linked list'를 정렬하는 방법은?

Answer 때때로, 리스트를 만들 때, 정렬하며 만드는 방법이 더 쉬울 때가 많습니다. (또는 리스트 대신에 트리(tree)를 쓰는 것이 더 편할 수 있습니다.) 삽입 정렬(insertion sort)이나 병합 정렬(merge sort) 같은 알고리즘은 리스트를 쓸 때, 매우 효과적입니다. 만약 표준 라이브러리 함수를 써서 정렬하고 싶다면, 먼저 이들 리스트를 가리키는 포인터의 배열을 (임시로) 만들고, qsort()를 쓰면 됩니다. 그리고, 이 정렬된 포인터를 이용하여, 리스트를 정리하면 됩니다.

References [Knuth] § 5.2.1 pp. 80–102, § 5.2.4 pp. 159–168
[Robert] § 8 pp. 98–100, § 12 pp. 163–175

Q 13.11 메모리의 크기보다 더 큰 데이터를 정렬하고 싶은데, 어떻게 하죠?

Answer 외부 정렬을 (external sort) 써야 합니다. 여기에 대한 것은 [Knuth]에 잘 나와 있습니다. 기본적인 아이디어는, 데이터를 (메모리에 불러올 수 있을 정도의) 작은 조각으로 나누고, 이 데이터를 정렬하고 난 다음, 이를 임시 파일에 저장하고, 다 반복하였으면, 이 임시 파일들을 합치는 (merge) 것입니다.

운영 체제에서 이러한 정렬 프로그램을 제공할 수도 있으니 프로그램 안에서 이러한 프로그램을 실행시키는 것을 생각할 수도 있습니다; 질문 19.27과 19.30, 그리고 질문 19.28의 예를 참고하기 바랍니다.

References [Knuth] § 5.4 pp. 247–378
[Robert] § 13 pp. 177–187

13.3 Date and Time

Q 13.12 현재 날짜와, 요일을 알아내려면 어떻게 하죠?

Answer `time()`, `ctime()`, `localtime()`, 또는 `strftime()`을 쓰면 됩니다. (이 함수들은 매우 오래 전부터 제공되었으며, 현재 ANSI 표준입니다.) 예를 들면 다음과 같습니다[2]:

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

int main()
{
  time_t now;
  time(&now);
  printf("It's %.24s.\n", ctime(&now));
  return 0;
}
```

`localtime`과 `strftime`은 다음과 같이 씁니다:

```
struct tm *tmp = localtime(&now);
char fmtbuf[30];
printf("It's %d:%02d:%02d\n",
       tmp->tm_hour, tmp->tm_min, tmp->tm_sec);
strftime(fmtbuf, sizeof fmtbuf, "%A, %B %d, %Y", tmp);
printf("on %s\n", fmtbuf);
```

(이 함수들이 주어진 값을 고치지 않는데도 `time_t` 변수에 대한 포인터를 인자로 받는 것에 주의하기 바랍니다.[3]

[2] ANSI 표준에 따르면 `time` 함수는 실패할 수 있으며, 이 때 `(time_t)(-1)`을 리턴합니다

[3] 이런 식의 인터페이스를 쓰는 이유는, 초창기 C 언어에서 `long` 데이터 타입이 나오기 전에 두 개의 `int`로 이루어진 배열을 써서 time 값을 저장했습니다.

References [K&R2] § B10 pp. 255–7
[ANSI] § 4.12
[C89] § 7.12
[H&S] § 18

Q 13.13 라이브러리 함수 `localtime()`은 `time_t`를 `struct tm` 형태로 바꾸어 줍니다. 그리고 `ctime()`은 `time_t`를 문자열로 바꾸어 줍니다. 거꾸로 변환하고 싶으면 어떻게 해야 할까요? 즉, 문자열이나 `struct tm`을 `time_t` 타입으로 변경하고 싶습니다.

Answer ANSI C에서는 `struct tm` 타입을 `time_t`로 바꾸어 주는 `mktime()` 함수를 제공합니다.

문자열을 `time_t`로 바꾸는 것은 조금 힘듭니다. 왜냐하면, 각 나라 혹은 지역별로 사용하는 날짜와 시간 형식이 각각 다르기 때문입니다. 어떤 시스템은 `strptime()`이라는 함수를 제공하며, 그 기능은 `strftime()`의 반대입니다. 또 (RCS 패키지와 함께 제공되는) `partime()`이라는 함수와 (최신의 C news 배포판에서 제공하는) `getdate()`라는 함수도 자주 쓰입니다. 질문 18.16을 보기 바랍니다.

References [K&R2] § B10 p. 256
[ANSI] § 4.12.2.3
[C89] § 7.12.2.3
[H&S] § 18.4 pp. 401–2

Q 13.14 주어진 날짜에 (`N` days) 날짜를 더하는 기능이 있나요? 또 주어진 두 날짜를 비교하는 함수가 있을까요?

Answer 이러한 문제를 해결하기 위해서, ANSI/ISO C 표준은 `mktime`과 `difftime` 함수를 제공합니다. `mktime()`은 정규화되지 않은 (non-normailized) 날짜를 `struct tm` 타입으로 입력받아 이것을 정규화시켜 줍니다. 즉, `struct tm`의 변수에 기준 날짜 값을 넣고, `tm_mday`에서 원하는 날짜만큼 더하거나 뺀 다음, `mktime`을 부르면 normalize된 년, 월, 일을 얻을 수 있습니다. (추가적으로 이 값을 `time_t` 타입으로 변경시켜 줍니다.) `difftime()`은 두 개의 `time_t` 값을 입력받아 초 단위로 비교해 줍니다; 이 때 필요한 `time_t` 값은 `mktime()`을 써서 얻습니다.

그러나 이 해결책들은 주어진 날짜 값이 `time_t` 범위 안에 있을 때만 동작합니다. `tm_days` 필드는 `int`이기 때문에 일(day) 수가 `int` 값을 넘는 값일 경우 (예를 들어 32,736 이상), 오버플로우가 발생할 수 있습니다. 또한

daylight saving time(써머 타임)을 고려하면 하루는 24 시간이 아닐 수도 있습니다 (따라서 86400 seconds/day로 나누면 해결될 것이라는 생각을 하면 안됩니다).

다음은 October 24, 1994에서 90일을 빼는 코드입니다:

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

tm1.tm_mon = 10 - 1;
tm1.tm_mday = 24;
tm1.tm_year = 1994 - 1900;
tm1.tm_hour = tm1.tm_min = tm1_tm_sec = 0;
tm1.tm_isdst = -1;

tm1.tm_mday -= 90;

if (mktime(&tm1) == -1)
  fprintf(stderr, "mktime failed.\n");
else
  printf("%d/%d/%d\n",
         tm1.tm_mon + 1, tm1.tm_mday, tm1.tm_year + 1900);
```

(`tm_isdst`를 -1로 설정하거나 `tm_hour`를 12로 설정하면 daylight saving time 문제로부터 보호받는데 도움을 줍니다.)

아래 코드는 2000년 2월 28일과 2000년 3월 1일 사이의 날짜 차이를 구하는 코드입니다:

```
struct tm tm1, tm2;
time_t t1, t2;

tm1.tm_mon = 2 - 1;
tm1.tm_mday = 28;
tm1.tm_year = 2000 - 1900;
tm1.tm_hour = tm1.tm_min = tm1.tm_sec = 0;
tm1.tm_isdst = -1;

tm2.tm_mon = 3 - 1;
tm2.tm_mday = 1;
```

```
tm2.tm_year = 2000 - 1900;
tm2.tm_hour = tm2.tm_min = tm2.tm_sec = 0;

t1 = mktime(&tm1);
t2 = mktime(&tm2);

if (t1 == -1 || t2 == -1)
  fprintf(stderr, "mktime failed\n");
else {
  long d = (difftime(t2, t1) + 86400L/2) / 86400L;
  printf("%ld\n", d);
}
```

(위에서 추가적으로 쓴 `86400L / 2`는 계산 결과를 원하는 근사값으로 만들어 줍니다; 질문 14.6 참고)

"Julian day number"를 (또는 4013 BC[4] 1월 1일) 사용하는 방법도 있습니다. 아래는 Julian day로 바꾸는 함수의 선언입니다:

```
/* returns Julian for month, day, year */
long ToJul(int month, int day, int year);

/* returns month, day, year for jul */
void FromJul(long jul,
             int *monthp, int *dayp, int *yearp);
```

이 때, `n`일(day)을 더하는 것은 다음과 같이 합니다:

```
int n = 90;
int month, day, year;
FromJul(ToJul(10, 24, 1994) + n, &month, &day, &year);
```

또, 두 날짜의 차이(일 수 단위로)는 다음과 같이 구합니다:

```
ToJul(3, 1, 2000) - ToJul(2, 28, 2000)
```

Julian day를 다루는 함수들은 Snippets 콜렉션에 (질문 18.15c 참고) 있습니다. 그리고 이 콜렉션은 Simtel/Oakland 아카이브(archive)에서 (파일

[4]좀 더 정확히 말해, 그 날짜 GMT 정오부터입니다. 여기서 말하는 Julian day number는 데이터 프로세싱에 쓰이는 "Julian dates"와 아무런 상관이 없습니다. 그리고 이 둘 다 모두 "Julian calendar"와 아무 상관없습니다.

JULCAL10.ZIP, 질문 18.16 참고) 얻을 수 있습니다. 아래의 "Date conversions" 기사도 참고하기 바랍니다.

덧붙여 질문 13.13, 20.31, 20.32도 참고하시기 바랍니다.

References [K&R2] § B10 p. 256
[ANSI] § 4.12.2.2, § 4.12.2.3
[C89] § 7.12.2.2, 7.12.2.3
[H&S] § 18.4,18.5 pp. 401–2
[Burki]

Q 13.14b C 언어는 2000년 문제에 어떤 문제가 있을까요?

Answer 없습니다. 단지, 허술하게 작성된 C 프로그램에 문제가 있을 뿐입니다.

`struct tm`의 `tm_year` 필드는 1900년대에서 시작한 년도를 가지고 있습니다. 따라서 2000년일 경우에 이 값은 100이 됩니다. `tm_year`를 제대로 사용하는 코드라면 (변환할 때, 1900을 더하거나 빼는) 전혀 문제될 것이 없습니다. 그러나 이 값을 두 자리 숫자로 바로 사용한다거나 다음과 같은 코드를 썼다면:

```
tm.tm_year = yyyy % 100;        /* WRONG */
```

또는, 다음과 같이 했다면:

```
printf("19%d", tm.tm_year);     /* WRONG */
```

2000년 문제가 발생합니다. 덧붙여 질문 20.32도 참고하시기 바랍니다.

References [K&R2] § B10 p. 255
[C89] § 7.12.1
[H&S] § 18.4 p. 401

13.4 Random Numbers

Q 13.15 난수(random number) 발생기가 필요합니다.

Answer 표준 C 라이브러리에는 `rand()`라는 함수가 있습니다. 물론 각각의 시스템에 따라 이 함수의 구현(implementation)은 완전하지 않을 수 있습니다. 그러나, 이 함수보다 더 좋은 성능을 가진 함수를 직접 만들기란 쉬운 일이 아닙니다.

직접 난수 발생기를 만들려고 한다면, 읽어야 할 책들이 꽤 많습니다; 이 글 뒷부분에 있는 Reference를 보시기 바랍니다. 또한 인터넷 상에 많은 패키지가 이미 올라와 있습니다; r250, RANLIB, FSULTRA로 검색해 보기 바랍니다. (질문 18.16도 참고.)

아래는 Park씨와 Miller씨에 의해 제안된, "minimal standard" generator 입니다.

```
#define a 16807
#define m 2147483647
#define q (m / a)
#define r (m % a)

static long int seed = -1;

long int PMrand()
{
  long int hi = seed / q;
  long int lo = seed % q;
  long int test = a * lo - r * hi;
  if (test > 0)
    seed = test;
  else
    seed = test + m;
  return seed;
}
```

(The "minimal standard" is adequately good; it is something "against which all others should be judged" and is recommended for use "unless one has access to a random number generator *known* to be better.")

이 코드는 `a = 16807`이고, `m = 2147483647`(이 값은 $2^{31} - 1$입니다)이고, `c = 0`인, 다음 generator를 구현합니다:

$$X \leftarrow (aX + c) \bmod m$$

나머지(modulus) 연산이 소수(prime) 연산이기 때문에, 이 generator는 질문 13.18에 나온 문제가 발생하지 않습니다. 위에서 사용한 곱하기는 Schrage씨에 의해 소개된 기법을 써서, 곱한 결과인 `aX`가 overflow를 발생시키지 않습니다. 위 코드는 값의 범위가 `[1, 2147483647]`를 만족하는 `long int` 값을

돌려주며, 이 범위는 C 언어의 `rand` 함수가 `RAND_MAX`보다 작은 값을 돌려주는 것과 비슷한 방식입니다. (단지 위 코드가 만들어 내는 값에는 0이 들어가지 않는다는 것만 다릅니다.) 위 코드를 (Park씨와 Miller씨의 논문에 나온 것처럼) 범위 `(0, 1)` 사이의 실수를 만들도록 고치려면, 위 함수의 선언을 다음과 같이 고치고:

```
double PMrand(void);
```

마지막 줄을 다음과 같이 고칩니다:

```
return (double)seed / m;
```

통계적으로 좀 더 좋은 결과를 얻으려면, (Park씨와 Miller씨의 추천에 따르면) `a = 48271`을 쓰기 바랍니다.

References [K&R2] § 2.7 p. 46, § 7.8.7 p. 168
[ANSI] § 4.10.2.1
[C89] § 7.10.2.1
[H&S] § 17.7 p. 393
[PCS] § 11 p. 172
[Knuth] Vol. 2 Chap. 3 pp. 1–177
[Park]

Q 13.16 어떤 범위를 갖는 난수를 발생시키고 싶습니다.

Answer 다음과 같이 하는 것은 (0에서 N-1까지의 수치를 리턴함) 매우 서투른 방법입니다:

```
rand() % N          /* POOR */
```

왜냐하면, 대부분의 난수 발생기에서 하위(low-order) 비트들은 그리 랜덤하지 않기 때문입니다. (질문 13.18을 보기 바랍니다.) 따라서, 다음과 같이 하는 것이 좋습니다:

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

실수를 쓰기 싫다면, 다음과 같이 해도 좋습니다:

```
rand() / (RAND_MAX / N + 1)
```

두 방법 모두 (ANSI 표준에 의해 <stdlib.h>에 정의되어 있는) RAND_MAX를 사용하고, N이 RAND_MAX보다 아주 작은 값이라는 것을 가정한 방법입니다.

만약 N이 RAND_MAX에 가깝고, random number generator의 범위가 N의 배수가 아니라면 (즉, RAND_MAX + 1 % N != 0인 경우) 위 모든 방법이 나쁜 방법이 됩니다: 어떤 범위의 수치들이 다른 범위보다 훨씬 더 많이 나타나게 됩니다. (실수를 쓴다고 해결될 문제가 아닙니다. 문제는 rand 함수가 서로 다른 RAND_MAX + 1가지의 값을 리턴하고, 이 것을 N개의 묶음으로 공평하게 나눌 수 없기 때문에 발생하는 것입니다.) 만약 이것이 문제가 된다면, rand를 여러 번 불러서, 특정 값들을 무시하면 됩니다:

```
unsigned int x = (RAND_MAX + 1u) / N;
unsigned int y = x * N;
unsigned int r;

do {
  r = rand();
} while (r >= y);
return r / x;
```

또, [M, N]의 범위를 가지는 random number를 구하고 싶다면, 위 결과를 shift해서 얻을 수 있습니다:

```
M + rand() / (RAND_MAX / (N - M + 1) + 1)
```

(어쨋든, RAND_MAX는 rand()가 리턴할 수 있는 최대 수치를 나타내는 상수라는 것을 알아 두시기 바랍니다. RAND_MAX에 여러분이 다른 수치를 대입할 수는 없으며, rand()가 다른 범위의 수치를 리턴하도록 해 주는 방법은 존재하지 않습니다.)

만약에 여러분이 (질문 13.15의 마지막 PMrand 함수 또는 질문 13.21의 drand48 함수처럼) 0과 1 사이의 범위를 가지는 난수 발생기를 가지고 있다면, 그리고 0과 N-1 사이의 범위를 가지는 정수 난수를 만들고 싶다면, 단순히 그 난수 발생기의 수치에 N을 곱하면 됩니다:

```
(int)(drand48() * N)
```

References [K&R2] § 7.8.7 p. 168
[PCS] § 11 p. 172

Q 13.17 프로그램을 실행시킬 때마다, `rand()`는 일정한 순서로 난수를 (random number) 발생시킵니다. 왜 그러죠?

Answer 대부분 가상(pseudo) random generator의 특징은 (C 라이브러리의 `rand`도 마찬가지), 항상 같은 번호에서 시작해서 똑같은 sequence로 값이 나온다는 것입니다. (Among other things, a bit of predictability can make debugging much easier.) 예상 불가능한 난수를 발생시키고 싶으면, `srand` 함수를 불러서, 이 pseudo-random generator의 초기값(seed)으로, 진짜 random 값을 하나 주면 됩니다. 보통 이 seed 값으로, 현재 시간을 쓰거나, 사용자가 키를 누르기까지의 시간 간격 등을 쓰게 됩니다. (물론 키를 누르기까지의 경과된 시간을 얻는 것은 이식성 있는 범위 안에서 만들기가 힘듭니다. 질문 19.37 참고) 아래는 현재 시간을 써서 seed 값을 설정하는 코드입니다.

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

srand((unsigned int)time((time_t *)NULL));
```

(일반적으로 프로그램 내에서는 `srand()` 함수를 한번만 불러주어도 충분합니다. `rand()`를 부를 때마다, `srand()`를 부른다면, random number를 얻을 수 없습니다.)

References [K&R2] § 7.8.7 p. 168
[ANSI] § 4.10.2.2
[C89] § 7.10.2.2
[H&S] § 17.7 p. 393

Q 13.18 참/거짓을 나타내는 random number(난수)가 필요합니다. `rand() % 2`를 썼는데, 계속 0, 1, 0, 1, 0 만을 반복합니다.

Answer (불행하게도 어떤 시스템에서 제공하는 것과 같은) 엉성하게 만들어진 pseudo-random number generator는 하위 (low-order) 비트들에 대해서는 그렇게 랜덤하지 않습니다. (In fact, for a pure linear congruential random number generator with period 2^e, and this tends to be how random number generates for e-bit machines are written, the low-order n bits repeat with period 2^n.) 이러한 이유에서, 상위 (high-order) 비트를 쓰는 것이 더 바람직합니다: 질문 13.16을 보기 바랍니다.

References [Knuth] § 3.2.1.1 pp. 12–14

Q 13.19 반복되지 않는, random number의 sequence를 얻으려면 어떻게 해야 할까요?

Answer 여러분이 찾고 있는 것은 보통, "random permutation(순열)", 또는 "shuffle"이라고 부릅니다. 한가지 방법으로, 뒤섞이기 원하는 값들을 배열에 넣고, 이 배열의 임의의 두 element의 값을 서로 바꾸는 작업을 반복해서 얻을 수 있습니다:

```
int a[10], i, nvalues = 10;

for (i = 0; i < nvalues; i++)
  a[i] = i + 1;

for (i = 0; i < nvalues - 1; i++) {
  int c = randrange(nvalues - i);
  int t = a[i]; a[i] = a[i + c]; a[i + c] = t;  /* swap */
}
```

여기에서 `randrange(N)`은 `rand() / (RAND_MAX / (N) + 1)` 또는 질문 13.16에 나온 코드를 씁니다.

References [Knuth] § 3.4.2 pp. 137–8

Q 13.20 How can I generate random numbers with a normal or Gaussian distribution?

Answer 적어도 다음과 같은 세 가지 방법이 있습니다:

- Exploit the Central Limit Theorem ("law of large numbers") and add up several uniformly distributed random numbers:

```
#include <stdlib.h>
#include <math.h>

#define NSUM 25

double gaussrand()
{
  double x = 0;
  int i;
```

```
  for (i = 0; i < NSUM; i++)
    x += (double)rand() / RAND_MAX;

  x -= NSUM / 2.0;
  x /= sqrt(NSUM / 12.0);

  return x;
}
```

(Don't overlook the sqrt(NSUM / 12.) correction, although it's easy to do so accidentally, especially when `NSUM` is 12.)

- Use a method described by Abramowitz and Stegun:

```
#include <stdlib.h>
#include <math.h>

#define PI 3.141592654

double gaussrand()
{
  static double U, V;
  static int phase = 0;
  double z;

  if (phase == 0) {
    U = (rand() + 1.) / (RAND_MAX + 2.);
    V = rand() / (RAND_MAX + 1.);
    Z = sqrt(-2 * log(U)) * sin(2 * PI * V);
  }
  else
    Z = sqrt(-2 * log(U)) * cos(2 * PI * V);

  phase = 1 - phase;

  return Z;
}
```

- Use a method described by Box and Muller and discussed in Knuth:

```
#include <stdlib.h>
```

```
#include <math.h>

double gaussrand()
{
  static double V1, V2, S;
  static int phase = 0;
  double X;

  if (phase == 0) {
    do {
      double U1 = (double)rand() / RAND_MAX;
      double U2 = (double)rand() / RAND_MAX;

      V1 = 2 * U1 - 1;
      V2 = 2 * U2 - 1;
      S = V1 * V1 + V2 * V2;
    } while (S >= 1 || S == 0);

    X = V1 * sqrt(-2 * log(S) / S);
  }
  else
    X = V2 * sqrt(-2 * log(S) / S);

    phase = 1 - phase;

    return X;
}
```

These methods all generate numbers with mean 0 and standard deviation 1. (To adjust to another distribution, multiply by the standard deviation and add the mean.) Method 1 is poor "in the tails" (especially if NSUM is small), but methods 2 and 3 perform quite well. See the references for more information.

References

[Knuth] Vol.2 § 3.4.1 p. 117

Box and Muller, "A Note on the Generation of Random Normal Deviates"

Marsaglia and Bray, "A Convenient Method for Generating Normal Vari-

ables"
Abramowitz and Stegun, *Handbook of Mathematical Functions*
Press *et al.*, *Numerical Recipes in C* § 7.2 pp. 288-90.

Q 13.21 어떤 프로그램을 포팅하고 있는데, 이 프로그램이 `drand48`이란 함수를 씁니다. 그리고 이 함수는 제 시스템에서 제공하지 않습니다. 어떻게 이 함수를 만들 수 있을까요?

Answer UNIX System V에서 제공하는 `drand48()` 함수는 (추측하건대, 48 비트의 precision을 갖는) 범위 `[0, 1)` 사이의 floating-point(실수) random number를 리턴합니다. (이 함수의 seed 값을 설정하는 함수는 `srand48`입니다; 두 함수 모두 C 표준이 아닙니다.) 정확성이 낮은 (low precision) 대체 함수는 다음과 같습니다:

```
#include <stdlib.h>

double drand48(void)
{
  return rand() / (RAND_MAX + 1.);
}
```

좀 더 정확한 값을 (`drand48`처럼 48 bit의 precision을 갖는) 돌려주는 함수는 다음과 같습니다:

```
#define PRECISION 2.82e14      /* 2**48, rounded up */

double drand48(void)
{
  double x = 0;
  double denom = RAND_MAX + 1.;
  double need;

  for (need = PRECISION; need > 1;
       need /= (RAND_MAX + 1.)) {
    x += rand() / denom;
    denom *= RAND_MAX + 1.;
  }
  return x;
}
```

Before using code like this, though, beware that it is numerically suspect, particularly if (as is usually the case) the period of `rand` is on the order of `RAND_MAX`. (If you have a longer-period random number generator available, such as BSD `random`, definitely use it when simulating `drand48`.)

References [PCS] § 11 p. 149

13.5 Other Library Functions

Q 13.22 `exit(status)`가 정말로 `main`에서 `status`를 리턴하는 것과 같나요?

Answer 질문 11.16을 보기 바랍니다.

Q 13.23 `memcpy`와 `memmove`가 다른 점이 있나요?

Answer 질문 11.25를 보기 바랍니다.

Q 13.24 오래된 프로그램을 포팅하려고 합니다. 그런데 `index`, `rindex`, `bcopy`, `bcmp`, `bzero` 함수가 정의되어 있지 않은 것 같습니다. "undefined external"이라는 에러가 뜨는 군요.

Answer 안 쓰는 구식의 함수들입니다. 다음 함수들을 쓰기 바랍니다.

`index`	`strchr` 함수를 쓰기 바랍니다.
`rindex`	`strrchr` 함수를 쓰기 바랍니다.
`bcopy`	첫 번째 인자와 두 번째 인자를 바꾸어서 `memmove` 함수를 쓰기 바랍니다. 덧붙여 질문 11.25도 참고하시기 바랍니다.
`bcmp`	`memcmp` 함수를 쓰기 바랍니다.
`bzero`	두 번째 인자를 0으로 해서 `memset` 함수를 쓰기 바랍니다.

만약, 오래된 시스템을 쓰고 있고, 위 표의 오른쪽에 있는 함수들을 쓰는 프로그램을 포팅해야 한다면, 왼쪽에 있는 함수로 바꾸거나, 아니면 새 함수들을 만들 수 있습니다. 덧붙여 질문 12.22, 13.21도 참고하시기 바랍니다.

References [PCS] § 11

Note `index`, `rindex`, `bcopy`, `bcmp`, `bzero` 모두 4.3 BSD 시스템에서 제공하는 함수로서, C 표준이 아닙니다. 이들 함수는 특히, 오래된 network 관련 프로그램 소스에서 많이 쓰며, 1:1 대응되는 표준 함수들이 (위에서 소개한 것처럼) 있으니, 표준 함수들을 쓰시기 바랍니다. 기존 소스를 수정할 수 없는 상황이라면, 이들 함수를 매크로로 표준 함수로 바꿔서 만들 수 있습니다. Richard Stevens씨는 *UNIX Network Programming* 책에서 모든 코드가 4.3 BSD 함수를 쓰도록 만들고 헤더 파일 `<unp.h>`에서 표준 함수로 (매크로를 써서) 예전 함수를 만들어 두었습니다.

References [Stevens98]

Q 13.25 필요한 헤더 파일들을 모두 `#include`시켰는데도 라이브러리 함수가 정의되어 있지 않다고 에러가 발생합니다.

Answer 일반적으로, 헤더 파일은 선언(declaration)만을 포함하고 있습니다. 어떤 경우에는 (특히, 비표준 함수인 경우) 실제 정의를 포함시키기 위해서는 특별한 라이브러리를 같이 링크(link)하라고 명령을 주어야 합니다. (`#include`하는 것만으로는 불충분합니다.) 덧붙여 질문 11.30, 13.26, 14.3도 참고하시기 바랍니다.

Q 13.26 올바른 라이브러리를 포함시키라고 했는데도 라이브러리 함수가 정의되어 있지 않다고 에러가 발생합니다.

Answer 대부분 linker들은, 주어진 오브젝트 파일과 라이브러리 파일들을 단 한번씩만 조사하고, 함수 호출에 필요한 정의가 발견되면 함께 링크합니다. 만약 주어진 파일 목록에서 (한번 시도해서) 함수 정의를 찾지 못한다면, 심볼이 정의되어 있지 않다고 에러가 생깁니다. 따라서, linker에게 전달하는 오브젝트 파일들과 라이브러리 파일들의 순서가 매우 중요합니다; 보통의 경우, 라이브러리 파일을 마지막에 써 두는 것이 좋습니다.

예를 들어, 대부분 UNIX 시스템에서 다음과 같이 명령을 실행하면, 보통 동작하지 않습니다:

```
cc -lm myproc.c
```

대신, `-l` 옵션을 마지막에 주면 동작합니다:

```
cc myproc.c -lm
```

라이브러리 파일을 먼저 주면, linker는 이 라이브러리에서 제공하는 정의들이 나중에 쓰일지 쓰이지 않을 지 알 수가 없습니다. (아직 이 라이브러리에 있는 함수를 쓰는 오브젝트 파일을 발견하지 못했기 때문) 질문 13.28을 보기 바랍니다.

Q 13.27 제 프로그램은 단순히 "Hello, world"를 출력하는 일만 하는데, 실행 파일의 크기가 (수백 KB 정도로) 매우 큽니다. 왜 그럴까요? Include하는 헤더 파일을 줄여야 하나요?

Answer 그 문제는, 현재 라이브러리 디자인이 아주 서툰 수준이기 때문에 일어납니다. (특히 graphic user interface를 포함하는) run-time 라이브러리들은, 가능한 많은 기능을 라이브러리에 포함시키려하는 경향이 있습니다. 한 라이브러리가 어떤 일을 하기 위해 다른 라이브러리를 부른다면 (이런 방식은 매우 좋은 현상이며, 라이브러리가 왜 존재하는지 그 이유에 해당합니다.), (현실적으로) 그 라이브러리에 있는 거의 모든 내용을 한꺼번에 포함시키려 할 수 있습니다. 따라서 실행 파일의 크기가 커질수 있습니다.

포함하는 헤더 파일을 줄인다고 해결될 문제는 아닙니다. 왜냐하면, 선언된 함수를 줄인다고 해서 (보통 헤더 파일을 선언하고 그 헤더 파일에서 제공하는 함수를 쓰지 않는 경우), 여러분의 실행 파일에 포함되는 함수 정의와는 관계가 없기 (왜냐하면 쓰이지 않았기 때문에) 때문입니다. 덧붙여 질문 13.25도 참고하시기 바랍니다.

힘든 방법이지만, 쓰이지 않는 함수들의 의존성을 검사한 다음, 필요없는 라이브러리를 link하지 않는 방법이 있습니다. 또는, 라이브러리 제작자에게 라이브러리를 좀 더 깔끔하게 만들어달라고 요구할 수 있습니다.

Note Shared library (dynamic linked library)를 쓰면 이 문제를 해결할 수 있습니다. 즉 실행 파일이 필요한 함수들의 정의를 포함하지 않고, 단순히 그 함수를 부르는 stub code만이 들어가며, 실제 라이브러리는 필요에 따라서 시스템이 메모리에 load하는 경우에는, 실행 파일이 커지지 않습니다. 시스템에 따라서 shared library 기능이 없을 수 있으므로 확인 바랍니다.

보통 라이브러리는 여러 개의 오브젝트 파일을 묶은 형태로 존재합니다. 기능이 떨어지는 linker를 쓴다면, 위에서 설명한 것처럼 라이브러리 내용을 모두 실행 파일에 포함시키겠지만, 기능이 뛰어난 linker는 (라이브러리 안에 존재하는) 오브젝트 파일 단위로 link합니다. 따라서 의존성이 여러 오브젝트 파일에 거쳐서 존재하지 않은 한, 위와 같은 현상은 드물게 발생합니다. 물론 나중에는 정말로 뛰어난 linker가 나와서 이런 문제가 전혀 없을 수 있지만, 아직은 그런 때가 아닌 것 같습니다.

비슷한 이유에서 GNU C Library(glibc) 소스를 보면, (오브젝트 파일 단위로 링크되는 것을 가정하고, 최소한의 내용만 link될 수 있게 하기 위해서) 소스 파일 하나가 한 두개의 함수 정의만 포함하는 것을 알 수 있습니다.

References GNU C Library – `http://www.gnu.org/software/libc/`

Q 13.28 `_end`가 정의되어 있지 않다고 링커가 에러를 내는데 이게 무엇을 의미하나요?

Answer 오래된 UNIX 링커에서 흔히 발생하는 것입니다. 대개 `_end`가 정의되어 있지 않다고 에러가 나는 것은 다른 심볼(symbol)들도 정의가 되어 있지 않다는 것을 의미합니다. 정의되어 있지 않은 다른 심볼들이 있나 검사해 보기 바랍니다. 그러면 `_end`가 정의되어 있지 않다고 하는 에러는 사라집니다. (덧붙여 질문 13.25, 13.26도 참고하시기 바랍니다.)

Chapter 14

Floating Point

Floating point(실수) 계산은 때때로 문제가 되며, 신비스럽게 보이기도 합니다. 게다가 C 언어는 원래 floating point를 주로 쓸 목적이 아니었기 때문에 더욱 문제가 됩니다.

Q 14.1 `float` 타입의 변수에 3.1을 넣고 `printf`로 출력해보니 3.09999999가 나옵니다. 왜 그럴까요?

Answer 대부분의 컴퓨터는 실수를 표현할 때, 내부적으로 2 진수로 기록합니다. 2 진수에서는 0.1은 무리수로 (0.0001100110011...) 표현됩니다. 따라서 3.1과 같은 수는 (10 진수로 볼 때에는 유리수이지만) 2 진수에서는 유한한 수치로 표현될 수 없습니다. 여러분의 시스템에서 10 진수를 2 진수로 변환하는 루틴이 어떻게 작성되었느냐에 따라 다르겠지만, (특히 정밀도가 낮은 `float`을 쓸 경우) 대입한 수치와 출력 결과가 약간 다를 수 있습니다. 덧붙여 질문 14.6도 참고하시기 바랍니다.

Q 14.2 제곱근을 (square root) 구하려 하는데, 이상한 값만 나옵니다.

Answer 먼저 `<math.h>`를 포함시켰는지 검사해 보고, 함수들이 `double`을 리턴하도록 선언되었는지 체크해보기 바랍니다. (`atof()` 함수는 `<stdlib.h>`에 선언되어 있음을 주의하기 바랍니다.) 덧붙여 질문 14.3도 참고하시기 바랍니다.

References [CT&P] § 4.5 pp. 65–6

Q 14.3 삼각 함수를 쓰려고 하는데 `<math.h>`를 포함시켰는데도 컴파일러는 "undefined: sin"이라는 컴파일 에러를 출력합니다.

Answer 수학 라이브러리를 포함시켰는지 체크하기 바랍니다. 예를 들어 UNIX 시스템에서는 '-lm' 옵션을 컴파일할 때 마지막 인자로 주어야 합니다. 덧붙여 질문 13.25, 13.26, 14.2도 참고하시기 바랍니다.

Q 14.4 제가 만든 실수 계산 루틴은 매우 이상한 결과를 출력하고, 또 컴퓨터가 다르면 다른 결과가 나옵니다.

Answer 먼저 질문 14.2를 읽어보기 바랍니다.

문제가 간단하지 않겠지만, 디지털 컴퓨터에서 실수 처리는 정확하지 않을 수도 있다는 것을 알아두셔야 합니다. 언더플로우(underflow)가 일어날 수도 있으며, 오차가 점점 누적될 수도 있습니다.

따라서 실수 연산이 정확하지 않다는 것을 기억하기 바라며, 같은 이유로 두 실수가 같은지 비교하는 것은 좋지 않습니다. (Don't throw haphazard "fuzz factors" in, either; 질문 14.5를 참고하기 바랍니다.)

이런 문제는 꼭 C 언어에만 국한된 것이 아니고, 모든 프로그래밍 언어에서 발생할 수 있습니다. 실수에 대한 어떤 부분들은 대개 "프로세서가 어떻게 처리하느냐"에 따라 다릅니다 (덧붙여 질문 11.34도 참고하시기 바랍니다.), 그렇지 않는 경우라면 적절한 기능을 수행할 수 있게 하기 위해 댓가가 큰 에뮬레이션을 사용합니다.

안타깝게도 이 문서는 이런 실수 연산에 대한 단점이나 해결책을 위한 것이 아닙니다. 좋은 수치 프로그래밍에 (numerical programming) 대한 책들이 여러분을 도와줄 겁니다. 아래 참고 문헌에 나온 책들을 보시면 좋습니다.

References [1] § 6 pp. 115–8
[Knuth] Vol. 2 Chap. 4
[Goldberg]

Q 14.5 그럼 "아주 가까운" 두 실수를 비교하는 방법이 있을까요?

Answer 실수의 절대값이 정의에 의하면 'magnitude'에 따라 달라질 수 있으므로, 두 실수를 비교하려면 다음과 같이, 오차가 상대적으로 무시할 수 있는 작은 값인지를 검사하는 것이 좋습니다. 즉 다음과 같이 할 것이 아니라:

```
double a, b;
...
if (a == b) /* WRONG */
```

이렇게 합니다:

```
#include <math.h>

if (fabs(a - b) <= epsilon * fabs(a))
```

... TODO ...

References [Knuth] § 4.2.2 pp. 217–8

Q 14.6 수치를 반올림하는 (round) 방법을 알려주세요.

Answer 가장 간단하고 직관적인 방법은 다음과 같은 코드를 쓰는 것입니다:

```
(int)(x + 0.5)
```

단, 이 기법은 음수에는 쓸 수 없습니다. 음수에는 다음과 같은 코드를 써야 할 것입니다:

```
(int)(x < 0 ? x - 0.5 : x + 0.5)
```

Q 14.7 왜 C 언어에는 지수를 (exponentiation) 계산하는 연산자가 없을까요?

Answer 대부분 프로세서에는 지수를 계산하는 명령(instruction)이 없기 때문입니다. 대신 C 언어는 `pow()`라는 함수를 제공합니다. 이 함수는 `<math.h>`에 선언되어 있습니다. 크기가 작고 0보다 큰 정수를 곱하는 것이 이 함수를 쓰는 것보다 더 효과적일 수도 있습니다.

References [C89] § 7.5.5.1
[H&S] § 17.6 p. 393

Q 14.8 제 시스템의 `<math.h>`에는 매크로 `M_PI`가 정의되어 있지 않습니다.

Answer 이 매크로 상수는 (이 매크로의 값은 pi이며, 정밀도는 각 시스템에 따라 다릅니다) 표준이 아닙니다. 'pi' 값이 필요하다면, 여러분이 직접 정의하거나 `4 * atan(1.0)`으로 계산해야 합니다.

References [PCS] § 13 p. 237

Q 14.9 IEEE NaN과 같은 특수한 값을 검사하려면 어떻게 하면 되죠?

Answer 고품질의 IEEE 실수 구현 방법을 제공하는 대부분의 시스템에서는 (미리 정의된 상수와 `isnan()`과 같은 함수를 제공하며, 이들은 표준이 아닙니다. 또 `<math.h>`나 `<ieee.h>`, `<nan.h>`에 선언되어 있을 수 있습니다.) 이런 값들을 다루기 위한 함수들을 제공합니다. 그리고 이런 기능들을 지금 표준화하려고 하고 있답니다. NaN을 검사하는 가장 거칠고(crude) 간단한 방법은 다음과 같습니다.

```
#define isnan(x) ((x) != (x))
```

IEEE를 생각하지 않은 컴파일러는 이런 테스트를 최적화 단계에서 없애버릴 수도 있습니다[1].

[C9X]는 `isnan()`, `fpclassify()` 등과 다른 종류 함수들을 제공합니다.

또 하나의 방법은 이 실수를 `sprintf()`와 같은 루틴을 써서 포맷화시켜 보는 것입니다. 많은 시스템이 이런 경우 "NaN"이나 "Inf"와 같은 문자열을 만들어 줄 겁니다.

덧붙여 질문 19.39도 참고하시기 바랍니다.

References [C9X] § 7.7.3

Q 14.11 복소수[2]를 C 언어에서 구현하려면 어떤 방법이 제일 좋을까요?

Answer 가장 쉬운 방법은 간단한 구조체를 만들고 이 구조체에 대해 연산할 수 있는 함수를 만드는 것입니다. [C9X]는 '`complex`'라는 표준 데이터 타입을 지원하려 합니다. 덧붙여 질문 2.7, 2.10, 14.12도 참고하시기 바랍니다.

References [C9X] § 6.1.2.5, § 7.8

[1] although non-IEEE-aware compilers may optimize the test away
[2] complex number

Q 14.12 I'm looking for some code to do: Fast Fourier Transforms (FFT's), matrix arithmetic (multiplication, inversion, etc.), complex arithmetic?

Answer Ajay Shah has prepared a nice index of free numerical software which has been archived pretty widely; one URL is
`ftp://ftp.math.psu.edu/pub/FAQ/numcomp-free-c`.

덧붙여 질문 18.13,18.15c, 18.16도 참고하시기 바랍니다.

Q 14.13 Turbo C를 사용하고 있습니다. 그런데 프로그램을 실행하면 "floating point format not linked"라는 메시지를 출력하고 종료해버립니다.

Answer Borland사의 컴파일러를 포함한 (Ritchie씨의 오리지널 PDP-11 컴파일러도) 규모가 작은 시스템의 어떤 컴파일러들은 실수 처리 부분이 필요없다고 판단되면 실수 처리 루틴을 포함시키지 않습니다. 특히 실수를 처리하지 않는 (즉 `%f`, `%e`를 쓰지 않는) `printf()`나 `scanf()`는 많은 공간을 줄일 수 있습니다. 아마도 여러분의 컴파일러에서는 실수 처리 부분이 필요없다고 생각되어 빠진 것 같습니다. 이럴 때에는 간단한 dummy call을 사용하면 됩니다. 즉, `sqrt()`와 같은 함수를 한 번 불러주면, 실수 처리 루틴이 포함될 것입니다. (자세한 것은 `comp.os.msdos.programmer`의 FAQ를 읽어 보시기 바랍니다.)

References [H&S2002] § 6.2.5 p. 40,

Chapter 15

Variable-Length Argument Lists

널리 알려지지는 않았지만, C 언어는 함수가 가변 인자를 (즉 인자의 갯수가 정해지지 않은) 받을 수 있는 기능을 제공합니다. Variable-length argument list(가변 인자 리스트)는, 드물기는 하지만, `printf`와 같은 함수들에게는 꼭 필요한 것입니다. (variable-length argument list는 ANSI C 표준에서 공식적으로 지원하지만, ANSI C 표준 이전에는 엄격히 말해서 정의되어 있지 않습니다.)

Variable-length argument list를 처리하는 방법은 상당히 기묘하기까지 합니다. 정식으로 varaible-length argument list는 fixed part(고정된 부분)와 variable-length part(가변 길이)의 두 부분으로 나누어져 있습니다. 우리는 "variable-length argument list의 variable-length part"라는 과장된 용어를 쓰고 있다는 사실을 발견했지만, 어쩔 수 없습니다 (혹시 여러분이 "variadic"이나 "varargs"라는 용어가 쓰이는 것을 보신 적이 있을지도 모르겠습니다: 두 가지 용어 모두 "having a variable number of arguments.[1]"을 뜻합니다. 따라서 "vararg function" 또는 "varargs argument"라고 말할 수도 있습니다.)

Variable-length argument list를 쓰는 방법은 크게 세 단계로 이루어 집니다. 먼저 `va_list` 타입의 특별한 pointer type을 써서 선언하고, 그 다음 `va_start`를 불러서 초기화합니다.

[1] 정해지지 않은 갯수의 인자를 가지는

15.1 Calling Varargs Functions

Q 15.1 `printf()`를 부르기 전에 `#include <stdio.h>`를 쓰라고 하더군요. 꼭 그럴 필요가 있을까요?

Answer 적절한 프로토타입(prototype)을 현재 영역(scope) 안에 포함시키기 위해서입니다.

어떤 컴파일러에서는 가변 인자 리스트를 쓰는 함수에는 일반 방식과는 다른 호출 순서(calling sequence)를 사용합니다. (It might do so if calls using variable-length argument lists were less efficient than those using fixed-length.) 그러므로 함수의 프로토타입이 (즉 "`...`"를 포함한 선언) 영역 안에 있어야 컴파일러가 가변 인자 리스트 처리 메커니즘을 사용할 수 있습니다.

References [ANSI] § 3.3.2.2, § 4.1.6
[C89] § 6.3.2.2, § 7.1.7
[ANSI Rationale] § 3.3.2.2, § 4.1.6
[H&S] § 9.2.4 pp. 268–9, § 9.6 pp. 275–6

Q 15.2 `printf()`에서 `%f`가 `float`과 `double` 인자 모두에 쓰일 수 있는 이유는 무엇인가요?

Answer 가변 인자 리스트에서 가변 인자 부분에는 "default argument promotion"이 적용됩니다: 즉, `char`와 `short int` 타입은 `int`로 변경되며(promotion), `float` 타입은 `double` 타입으로 변경됩니다. (이 것은 함수의 프로토타입이 없거나 구 방식(old style)으로 선언된 함수에서 일어나는 'promotion'과 같은 것입니다; 질문 11.3을 참고하기 바랍니다.) 그러므로 `printf`의 `%f` 포맷은 항상 `double` 타입만 받아들이는 셈이 됩니다. (비슷한 이유에서 `%c`, `%hd` 포맷은 항상 `int`만을 받아 들이게 됩니다.) 덧붙여 질문 12.9, 12.13도 참고하시기 바랍니다.

References [ANSI] § 3.3.2.2
[C89] § 6.3.2.2
[H&S] § 6.3.5 p. 177, § 9.4 pp. 272–3

Q 15.3 `n`이 `long int`일 경우, 다음 코드에서 문제가 발생합니다:

```
printf("%d", n);
```

그렇지만 ANSI 함수 프로토타입을 제공했으니, 자동으로 형 변환이 적용될 거라고 생각합니다. 무엇이 잘못되었나요?

Answer 함수가 가변인자를 받는다면, 프로토타입이 있더라도, 가변 인자에 대해서는 어떠한 정보도 알지 못하며, 또한 알 수도 없습니다. 따라서, 일반적으로, 가변 인자 리스트에서 가변 인자에 해당하는 부분은 그러한 보호를 (type conversion) 받지 못합니다. 컴파일러는 가변인자에 대해서는 implicit type conversion을 수행하지 않으며, type이 서로 맞지 않는 경우 경고 해 줄 수 있습니다. 따라서 프로그래머는 가변 인자로 들어간 인자가 예상하는 타입과 같은지 확인해야 하며, 다를 경우에는 반드시 캐스팅을 (explicit cast) 해 주어야 합니다:

어떤 컴파일러들과 (gcc 포함), 어떤 `lint` 프로그램은, format string이 문자열 상수(string literal)인 경우에 한해서, `printf`와 같은 함수들에 전달된 가변 인자가 정말로 올바른 타입으로 전달되었는지 검사해 주기도 합니다.

덧붙여 질문 5.2, 11.3, 12.9, 15.2도 참고하시기 바랍니다.

Note `gcc`의 경우, function attribute란 것으로 `printf` 또는 `scanf`와 같은 함수의 가변 인자 부분을 검사해 줄 수 있습니다. 자세한 것은 `gcc` 매뉴얼을 참고 바랍니다.

15.2 Implementing Varargs Functions

Q 15.4 가변 인자를 받는 함수를 어떻게 만들 수 있을까요?

Answer `<stdarg.h>` 헤더 파일에 있는 기능을 사용합니다.

아래의 함수는 주어진 여러 개의 문자열을 붙여서 `malloc()`으로 할당한 메모리에 저장해서 리턴하는 함수입니다:

```
#include <stdlib.h>     /* for malloc, NULL, size_t */
#include <stdarg.h>     /* for va_ stuff */
#include <string.h>     /* for strcat et al.  */

char *vstrcat(char *first, ...)
{
  size_t len;
  char *retbuf;
  va_list argp;
```

```
    char *p;

    if (first == NULL)
      return NULL;

    len = strlen(first);

    va_start(argp, first);

    while ((p = va_arg(argp, char *)) != NULL)
      len += strlen(p);

    va_end(argp);

    retbuf = malloc(len + 1);   /* +1 for trailing \0 */

    if (retbuf == NULL)
      return NULL;            /* error */

    (void)strcpy(retbuf, first);

    va_start(argp, first);      /* restart; 2nd scan */

    while ((p = va_arg(argp, char *)) != NULL)
      (void)strcat(retbuf, p);

    va_end(argp);

    return retbuf;
  }
```

(인자 리스트를 처음부터 다시 검색할 때에는 위에서처럼 `va_start`를 다시 불러 주어야 합니다. 또한 `va_end`가 실제로 하는 일이 없을 수는 있지만, 이식성을 위해서 반드시 불러주어야 합니다.)

사용법은 다음과 같습니다:

```
  char *str = vstrcat("Hello, ", "world!", (char *)NULL);
```

마지막 인자를 캐스팅한 것을 꼭 주의깊게 보시기 바랍니다; 질문 5.2, 15.3을 참고하기 바랍니다. (또한 이 함수를 부른 함수는 이 함수가 리턴한 문자열을 free시켜 주어야 할 책임이 있습니다.)

위의 예는 모든 가변 인자의 타입이 char *입니다. 이제, 가변 인자의 갯수와 타입이 정해지지 않는 함수를 만들어 보겠습니다; 이 함수는 아주 간단한 기능만 지원하는 printf입니다. 특히 va_arg()에서 원하는 타입을 미리 지정한 부분에 대해 주의깊게 보시기 바랍니다.

(아래 miniprintf 함수는 질문 20.10에서 쓴 baseconv를 씁니다. 또한 이것은 INT_MIN과 같은, 가장 작은 정수 값을 올바르게 출력하기에 적당하지 않습니다.)

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

extern char *baseconv(unsigned int, int);

void
miniprintf(char *fmt, ...)
{
  char *p;
  int i;
  unsigned u;
  char *s;
  va_list argp;

  va_start(argp, fmt);

  for (p = fmt; *p != '\0'; p++) {
    if (*p != '%') {
      putchar(*p);
      continue;
    }

    switch (*++p) {
    case 'c':
      i = va_arg(argp, int);
      /* not va_arg(argp, char); 질문 15.10 참고 */
```

```
    putchar(i);
    break;

  case 'd':
    i = va_arg(argp, int);
    if (i < 0) {
      /* XXX won't handle INT_MIN */
      i = -i;
      putchar('-');
    }
    fputs(baseconv(i, 10), stdout);
    break;

  case 'o':
    u = va_arg(argp, unsigned int);
    fputs(baseconv(u, 8), stdout);
    break;

  case 's':
    s = va_arg(argp, char *);
    fputs(s, stdout);
    break;

  case 'u':
    s = va_arg(argp, unsigned int);
    fputs(baseconv(u, 10), stdout);
    break;

  case 'x':
    u = va_arg(argp, unsigned int);
    fputs(baseconv(u, 16), sttout);
    break;

  case '%':
    putchar('%');
    break;
  }
```

```
    va_end(argp);
    }
}
```

덧붙여 질문 15.7도 참고하시기 바랍니다.

References [K&R2] § 7.3 p. 155, § B7 p. 254
[ANSI] § 4.8
[C89] § 7.8
[ANSI Rationale] § 4.8
[H&S] § 11.4 pp. 296–9
[CT&P] § A.3 pp. 139–141
[PCS] § 11 pp. 184–5, § 13 p. 242

Q 15.5 `printf()`와 같이 포맷 문자열을 받아들여 처리하는 함수를 만들어 그 처리를 `printf()`에게 맡기고 싶습니다.

Answer `vprintf()`, `vfprintf()`, `vsprintf()` 함수를 쓰면 됩니다. 이 함수들은, 각각 `printf()`, `fprintf()`, `sprintf()`과 같은 일을 하며, 단지 마지막 인자가 가변 인자인 대신 `va_list` 타입에 대한 포인터를 받습니다.

예를 들어, 아래의 `error()` 함수는 에러 메시지를 받아들여 그 메시지 앞에 "error: "를 덧붙이고 newline을 붙여서 출력해주는 함수입니다:

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

void error(char *fmt, ...)
{
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

덧붙여 질문 15.7도 참고하시기 바랍니다.

References [K&R2] § 8.3 p. 174, § B1.2 p. 245
[C89] § 7.9.6.7, 7.9.6.8, 7.9.6.9
[H&S] § 15.12 pp. 379–80
[PCS] § 11 pp. 186–7

Q 15.6 `scanf()`와 같은 기능을 하는 함수를 만들고 싶습니다.

Answer [C9X]는 `vscanf()`와 `vfscanf()`, `vsscanf()`를 지원할 것입니다. (물론 당장 하기 위해서는 여러분 스스로가 그러한 함수를 만들어야 합니다.)

References [C9X] § 7.3.6.12–14

Note C99 표준은 `vscanf()`, `vfscanf()`, `vsscanf()`를 지원합니다. 따라서 질문 15.5와 같은 방법으로 만들면 됩니다.

References [H&S2002] § 15.12 p. 401 [C99] § 7.19.1, § 7.19.6.9, § 7.19.6.11, § 7.19.6.14

Q 15.7 ANSI 이전의 컴파일러를 사용하고 있습니다. `<stdarg.h>`가 없는데 어떻게 하죠?

Answer `<stdarg.h>`에 해당하는 오래된 헤더파일인 `<varargs.h>`를 쓰면 됩니다. 예를 들어, 질문 15.4에서 만든 `vstrcat`을, `<varargs.h>`를 쓰도록 고치면 다음과 같습니다:

```
#include <stdio.h>
#include <varargs.h>
#include <string.h>

extern char *malloc();

char *vstrcat(va_alist)
va_dcl                          /* no semicolon */
{
  int len = 0;
  char *retbuf;
  va_list argp;
  char *p;
```

```
    va_start(argp);

    while ((p = va_arg(argp, char *)) != NULL)
      len += strlen(p);

    va_end(argp);

    retbuf = malloc(len + 1);  /* +1 for trailing '\0' */

    if (retbuf == NULL)
      return NULL;             /* error */

    retbuf[0] = '\0';

    va_start(argp);            /* restart for second scan */

    while ((p = va_arg(argp, char *)) != NULL)
      strcat(retbuf, p);

    va_end(argp);

    return retbuf;
  }
```

(va_dcl뒤에 세미콜론(';')이 없는 것에 주의하기 바랍니다. 그리고, 이 경우, 첫번째 인자에 대한 특별한 처리가 필요없습니다.) 또, <string.h>을 포함시키는 대신, 직접 문자열 처리 관련 함수들을 선언해 주어야 할 지도 모릅니다.

만약, 시스템이 vfprintf를 제공하며, <stdarg.h>를 제공하지 않을 경우, 아래에 (질문 15.5에서 쓴) <varargs.h>를 쓴 error 함수를 보입니다:

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

  void error(va_alist)
  va_dcl                  /* no semicolon */
  {
    char *fmt;
```

```
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp);
    fmt = va_arg(argp, char *);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

(<stdarg.h>에서와는 달리, <varargs.h>에서는 모든 인자가 가변인자입니다. 따라서 fmt 인자도, va_arg를 써서 얻어야 합니다.)

References [H&S] § 11.4 pp. 296–9
[CT&P] § A.2 pp. 134–139
[PCS] § 11 pp. 184–5, § 13 p. 250

15.3 Extracting Variable-Length Arguments

Q 15.8 함수에 몇 개의 인자가 전달되었는지를 정확히 알 수 있는 방법이 있나요?

Answer 호환성있는 방법은 존재하지 않습니다. 어떤 오래된 시스템에서는 비표준 함수인 nargs()를 제공하기도 합니다. 그러나 이 함수는 인자의 갯수를 리턴하는 게 아니라 전달된 word 갯수를 리턴합니다. (구조체나 long int, 실수는 여러 개의 word로 이루어져 있는 경우가 대부분입니다.)

가변 인자를 받아 처리하는 함수는 그 자체만으로 인자의 갯수를 파악할 수 있어야 합니다. printf 계열의 함수들은 포맷 문자열에서 (%d와 같은) 포맷 specifier를 보고 그 갯수를 파악합니다. (그렇기 때문에 printf()에 전달된 인자의 갯수가 포맷 문자열과 맞지 않을 경우에 오류를 일으킵니다.) 또 다른 방법으로, 가변 인자가 모두 같은 타입일 경우, 마지막 인자를 (0, -1, 또는 적절한 널 포인터와 같은) 어떤 특정한 값으로 설정해서 인자의 갯수를 파악합니다 (질문 5.2, 15.4에서 execl()과 vstrcat() 함수의 사용법을 참고하시기 바랍니다). 마지막으로, 인자의 갯수를 미리 파악할 수 있다면, 전체 인자의 갯수를 인자로 전달하는 것도 좋은 방법입니다. (although it's usually a nuisance for the caller to supply).

References [PCS] § 11 pp. 167–8

Q 15.9 제 컴파일러는 다음과 같은 함수를 정의하면 에러를 냅니다.

```
int f(...)
{
}
```

Answer 표준 C에서는 `va_start()`를 쓰려면 적어도 하나의 고정된 인자가 있어야 한다고 말하고 있습니다. (어쨌든, 가변 인자의 갯수 또는 타입을 알려면, 하나 이상의 인자가 필요합니다.) 덧붙여 질문 15.10도 참고하시기 바랍니다.

References [ANSI] § 3.5.4, § 3.5.4.3, § 4.8.1.1
[C89] § 6.5.4, § 6.5.4.3, § 7.8.1.1
[H&S] § 9.2 p. 263

Q 15.10 가변 인자를 처리하는 함수에서 `float` 인자를 처리하지 못합니다. 왜 `va_arg(argp, float)`을 쓰면 동작하지 않을까요?

Answer 가변 인자 리스트에서 가변 인자 부분은 항상 "default argument promotion"이 적용됩니다: 즉 `float` 타입의 인자들은 항상 `double`로 변환되며, `char`나 `short int`의 경우 항상 `int`로 변환됩니다.

따라서, `va_arg(argp, float)`은 잘못된 코드이며, 대신 `va_arg(argp, double)`을 써야 합니다. 비슷한 이유로 `char`, `short`, `int`를 받기 위해서는 `va_arg(argp, int)`를 써야 합니다. 덧붙여 질문 11.3, 15.2도 참고하시기 바랍니다.

References [C89] § 6.3.2.2
[ANSI Rationale] § 4.8.1.2
[H&S] § 11.4 p. 297

Q 15.11 함수 포인터를 `va_arg()`로 얻을려고 하는데, 잘 안됩니다.

Answer `typedef`를 써서 함수 포인터를 새 타입으로 만들고 해 보기 바랍니다.

`va_arg()`와 같은 매크로는 함수 포인터와 같은 복잡한 타입을 사용할 때, 곤란을 겪기도 합니다 (be stymied). 이 문제를 이해하기 위해, 아래에 간단한 `va_arg()` 구현 예를 보입니다:

```
#define va_arg(argp, type) \
  (*(type *)(((argp) += sizeof(type)) - sizeof(type)))
```

위에서, argp는 (즉, va_list 타입) char *입니다. 이 때, 만약 다음과 같이 불렀다면:

```
va_arg(argp, int (*)())
```

va_arg는 다음과 같이 확장됩니다:

```
(*(int (*)() *)(((argp) += sizeof(int (*)())) -
    sizeof(int (*)())))
```

위 결과를 자세히 보면 알겠지만, syntax error입니다. (첫번째 (int (*)() *)로 캐스트하는 것은 의미가 없습니다.)[2]

만약, 함수 포인터를 다른 이름으로 typedef했다면 모든 문제가 해결됩니다. 주어진 함수 포인터를 다음과 같이 typedef를 만들었다면:

```
typedef int (*funcptr)();
```

그리고 아래와 같이 불렀다면:

```
va_arg(argp, funcptr)
```

다음과 같이 확장됩니다:

```
(*(funcptr *)(((argp) += sizeof(funcptr)) -
    sizeof(funcptr)))
```

이 경우, 정상적으로 동작합니다.

덧붙여 질문 1.13, 1.17 1.21도 참고하시기 바랍니다.

References [ANSI] § 4.8.1.2
[C89] § 7.8.1.2
[ANSI Rationale] § 4.8.1.2

15.4 Harder Problems

You can pick apart variable-length argument lists at run time, as we've seen. But you can *create* them only at compile time. (We might say that strictly

[2]이 확장이 올바르려면 다음과 같아야 합니다:
`(*(int (**)())(((argp) += sizeof(int (*)())) - sizeof(int (*)())))`

speaking, there are no truly variable-length argument lists; every actual argument list has some fixed number of arguments. A vararg function merely has the capability of accepting a different length of argument list with each call.) If you want to call a function with a list of arguments created on the fly at run time, you can't do so portably.

Q 15.12 가변 인자를 받아서 다시 가변 인자를 처리하는 함수에 넘겨 줄 수 있을까요?

Answer 일반적으로, 불가능합니다. 이상적으로, 이 경우, `va_list`를 받는 함수를 만들어야 합니다.

예를 들어, 치명적인(fatal) 에러 메시지를 출력하고 프로그램을 끝내는 함수인 `faterror`를 만든다고 가정해 봅시다. 여러분은 질문 15.5에서 쓴 `error` 함수를 쓰길 원한다고 합시다:

```
void faterror(char *fmt, ...)
{
  error(fmt, ????);
  exit(EXIT_FAILURE);
}
```

위에서 보면 알겠지만, `faterror`가 받은 가변 인자를 `error`에 전달할 방법이 없습니다.

이 문제를 처리하기 위해서, 첫째, `error` 함수를 분해해서 가변 인자 대신 하나의 `va_list`를 받는 `verror` 함수를 만듭니다. (별로 어렵지 않습니다. 왜냐하면 `verror`의 대부분은 `error`와 같으며, 일단 `verror`를 만들고 나면, `error` 함수는 `verror`를 써서 아주 쉽게 만들 수 있습니다.)

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

void verror(char *fmt, va_list argp)
{
  fprintf(stderr, "error: ");
  vfprintf(stderr, fmt, argp);
  fpritnf(stderr, "\n");
}

void error(char *fmt, ...)
```

```
{
  va_list argp;
  va_start(argp, fmt);
  verror(fmt, argp);
  va_end(argp);
}
```

위와 같이 만들었으면, 이제 faterror를 다음과 같이 만들 수 있습니다:

```
void faterror(char *fmt, ...)
{
  va_list argp;
  va_start(argp, fmt);
  verror(fmt, argp);
  va_end(argp);
  exit(EXIT_FAILURE);
}
```

자세히 보면 error와 verror의 관계는 print와 vprintf의 관계와 같습니다. Chris Torek씨가 제안한 것에 따르면, 여러분이 가변 인자를 처리하는 함수를 만들때마다, 두 가지 버전을 제공하는 것이 좋습니다: 하나는 (verror와 같이) va_list를 받아서 처리하는 함수와, (error와 같이) 앞 함수의 간단한 wrapper 역할을 하는 함수, 두 가지입니다. 이 방식의 한가지 단점은, verror와 같은 함수는 가변 인자를 단 한번씩만 scan할 수 있다는 것입니다; va_start를 다시 부를 방법은 없습니다.

만약, (faterror와 같은) 가변 인자를 받아서 이 것을 가변 인자를 받는 다른 함수에 전달하려고 하는데, (error와 같이) va_list를 받는 저수준 함수를 다시 만들 수 없다면, 이 문제를 해결하기 위한 portable한 방법은 존재하지 않습니다. (이 문제를 시스템의 어셈블리 언어를 써서 해결할 가능성은 있습니다. 덧붙여 질문 15.13도 참고하시기 바랍니다.)

다음과 같은 방법은 동작하지 않습니다:

```
void faterror(char *fmt, ...)
{
  va_list argp;
  va_start(argp, fmt);
  error(fmt, argp);      /* WRONG */
  va_end(argp);
```

```
    exit(EXIT_FAILURE);
}
```

`va_list` 자체는 가변 인자 리스트가 아닙니다; 내부적으로 이 것은 가변 인자 리스트를 가리키는 포인터입니다. 따라서 `va_list`를 받는 함수는 가변 인자 리스트를 받는 함수가 아니며, 그 역도 성립하지 않습니다.

Another kludge that is sometimes used and that sometimes works even though it is grossly nonportable is to use a lot of `int` arguments, hoping that there are enough of them and that they can somehow pass through pointer, floating-point, and other arguments as well:

```
void faterror(fmt, a1, a2, a3, a4, a5, a6)
char *fmt;
int a1, a2, a3, a4, a5, a6;
{
    error(fmt, a1, a2, a3, a4, a5, a6); /* VERY WRONG */
    exit(EXIT_FAILURE);
}
```

위 예는, 이런 식으로 하지 말라는 것을 보여주기 위해 만든 것입니다. 이 글에서 봤다는 이유로 위 코드를 쓰려고 하지 말기 바랍니다.

Q 15.13 런타임에 인자 리스트를 만들어서 함수를 부를 수 있는 방법은 없을까요?

Answer 호환성있는 방법도 없으며, 그런 일을 할 수 있다고 보장할 수도 없습니다.

실제 인자 리스트를 처리하는 대신 generic (`void *`) 포인터의 배열을 넘겨주는 방법을 생각할 수 있습니다. 그리고 불려진 함수는 (`main()`이 `argv` 인자를 처리하듯이) 이 배열 요소를 하나씩 조사해서 원하는 정보를 얻을 수 있습니다. (물론, 이 방식은 여러분이 불려진 모든 함수를 직접 제어할 수 있을 때에만 의미가 있습니다.)

덧붙여 질문 19.36도 참고하시기 바랍니다.

Chapter 16

Strange Problems

It's not even worth asking the rhetorical question, Have you ever had a baffling bug that you just couldn't track down? Of course you have; everyone has. C has a number of splendid "gotcha!s" lurking in wait for the unwary; this chapter discusses a few of them. (In fact, any language powerful enough to be popular probably has its share of surprises like these.)

Q 16.1 왜 아래 loop는 항상 한 번만 실행될까요?

```
for (i = start; i < end; i++);
{
  printf("%d\n", i);
}
```

Answer `for`를 쓴 문장의 마지막 뒤에 실수로 붙인 세미콜론 때문에, `for`가 반복할 문장(statement)은 "null statement"가 됩니다. 중괄호로 둘러싼 블럭은, 얼핏 보기에는 `for`가 반복할 코드로 보이지만, 사실은 `for`와 무관한, 그냥 다음 statement일 뿐입니다.

덧붙여 질문 2.18도 참고하시기 바랍니다.

Note 위 코드를 다음과 같이 띄어쓰면 쉽게 이해할 수 있습니다:

```
for (i = start; i < end; i++)
  ;

{
```

```
    printf("%d\n", i);
}
```

Q 16.1b 문법 오류(syntax error)가 엄청나게 많이 발생했는데 도저히 그 이유를 모르겠습니다.

Answer 먼저 헤더 파일과 소스 파일 전체에서 `#if` / `#ifdef` / `#ifndef` / `#else` / `#endif`의 쌍이 맞지 않는지 검사해보기 바랍니다. (덧붙여 질문 2.18, 10.9, 11.29도 참고하시기 바랍니다.)

Q 16.1c 왜 제 프로시져를 호출하는 부분이 동작하지 않을까요? 아무리 봐도 컴파일러가 프로시져 호출 부분을 건너 뛰어 버리는 것 같습니다.

Answer 혹시 다음과 같이 코드를 작성하지 않았나요?

```
myprocedure;
```

C 언어는 함수(function)만을 지원합니다. 그리고 함수에 전달되는 인자가 하나도 없더라도 괄호로 둘러싼 'argument list'를 써 주어야 합니다. 따라서 다음과 같이 써야 합니다:

```
myprocedure();
```

Q 16.2 파일의 첫 부분에서 이상한 syntax error가 납니다. 그런데 제가 보기에는 아무런 문제가 없습니다.

Answer 질문 10.9를 보기 바랍니다.

Q 16.3 동작하기도 전에 프로그램이 깨져버립니다. (debugger로 한 스텝씩 따라하면 `main()`에 들어가기도 전에 죽어 버립니다.)

Answer 아마도 하나 이상의 (1KB 이상의) 매우 큰 local 배열을 썼을 것입니다. 대부분의 시스템은 고정된 크기의 스택을 가지고 있고, 동적으로 스택을 할당하는 (e.g. Unix) 시스템이더라도 스택이 갑자기 단 한번에 엄청나게 커지려고 하면 혼란을 겪게 됩니다. 매우 큰 배열이 필요하다면 정적(static)으로 할당하는게 좋습니다 (함수를 호출할 때마다 재귀 호출과 같은 이유로, 새로운 배열이 필요하다면 `malloc()`을 써서 할당하는 게 좋습니다. 덧붙여 질문 1.31도 참고하시기 바랍니다.).

또, 프로그램을 잘못 link했을 확률도 있습니다 (각각 다른 컴파일러 옵션을 써서 만든 object module을 합쳤거나, 잘못된 dynamic library를 link했을 경우). 또는 run-time dynamic library linking이 어떤 이유로 실패했거나, `main`을 잘못 선언했을 수도 있습니다.

(덧붙여 질문 11.12b, 16.4, 16.5, 18.4도 참고하시기 바랍니다.)

Note 물론, 위 답변에서 쓴 1KB의 크기는 시스템에 따라 매우 달라질 수 있습니다. 이 글이 써진 시점이 조금 오래되었다는 것을 알기 바랍니다. 그러나, 어떤 시스템이든 상관없이, 그 시스템에서 상대적으로 큰 크기의 스택을 쓰려 했을 경우, 같은 현상이 일어날 수 있다는 것을 기억하기 바랍니다.

Q 16.4 제 프로그램은 정상적으로 동작하는데, 끝나기 직전에 박살납니다. 즉 `main()`의 마지막 문장을 실행한 다음에 깨지는 거죠. 왜 그럴까요?

Answer 적어도 다음 세 가지 사항을 검사해보기 바랍니다:

(a) `main` 바로 위에 선언한 선언에서 세미콜론(';')이 빠져 있으면, `main` 함수의 리턴 타입이 잘못되어 structure를 리턴하도록 선언되었을 가능성이 있습니다. 이 경우 run-time startup 코드와 충돌하여, 이런 현상이 일어날 수 있습니다. 질문 2.18, 10.9 참고

(b) `setbuf`, `setvbuf`에 전달한 버퍼가 `main`에서 선언한 local (그리고 automatic) 변수라면, `main`이 끝나고 stdio 라이브러리가 적절한 cleanup 코드를 수행할 때, 이 버퍼가 존재하지 않으므로 이러한 문제가 발생할 수 있습니다.

(c) `atexit`로 등록한 함수가 에러가 나서 이런 문제가 생길 수도 있습니다. 아마도 `main` 또는 다른 함수에 속해 있는 (local), 그래서 더 이상 존재하지 않는 데이터에 접근하려 했을 수 있습니다.

(두 번째와 세 번째 문제는 질문 7.5와 밀접한 관계가 있습니다; 덧붙여 질문 11.16도 참고하시기 바랍니다.)

References [CT&P] § 5.3 pp. 72–3

Note 또다른 경우, `main`에서 선언한 local이며 automatic인 변수를 (대개는 배열) 다룰 때, 이 변수의 범위를 벗어난 곳에 접근했을 (특히 write) 때, 이러한 현상이 나타납니다. 물론 범위를 벗어난 곳에 접근하려 하는 것은 그 자체가 "undefined behavior"를 낳습니다.

Q 16.5 이 프로그램은 어떤 컴퓨터에서는 완벽하게 돌아가지만, 다른 컴퓨터에서 돌리면 이상한 결과를 만들어 냅니다. 더욱 이상한 것은, 가끔 debugging 목적으로 코드의 중간 중간에 출력하는 문장을 넣으면 때때로 동작하기도 합니다. 왜 그럴까요?

Answer 이런 증상을 만들어 내는 이유는 많습니다. 아래에 일반적으로 문제를 일으키는 경우가 있으니, 참고하시기 바랍니다:

- (uninitialized local variables[1]) 초기화되지 않은 변수 (덧붙여 질문 7.1도 참고하시기 바랍니다.)
- integer overflow, 특히 16-bit 컴퓨터에서, 특히 `a * b / c`와 같은 연산을 할 때, 계산 도중 overflow가 일어날 때. (덧붙여 질문 3.14도 참고하시기 바랍니다.)
- (undefined evaluation order) 평가 순서가 정의되지 않은 경우 (질문 3.1에서 3.4까지 참고)
- external 함수의 선언이 생략되었을 경우. 특히 `int`가 아닌 다른 타입을 리턴하거나 "narrow" 또는 가변 인자를 가진 함수의 경우 (질문 1.25, 11.3, 14.2, 15.1 참고)
- 널 포인터를 dereference한 경우 (Chapter 5 참고)
- `malloc`/`free`를 잘 못 쓴 경우, 특히 `malloc`으로 할당한 메모리가 0으로 초기화되었다고 가정한 경우나, 이미 `free`한 메모리를 계속 쓴 경우, 두 번 `free`한 경우, `malloc`이 할당한 부분을 벗어나서 작업한 경우 (덧붙여 질문 7.19, 7.20도 참고하시기 바랍니다.)
- 일반적인 포인터 문제들 (덧붙여 질문 16.7 16.8도 참고하시기 바랍니다.)
- `printf()` 포맷과 인자가 서로 맞지 않은 경우, 특히 `%d`를 써서 `long int`를 출력하려 한 경우 (질문 12.7, 12.9 참고)
- `unsigned int`가 표현할 수 있는 범위 밖의 크기의 메모리를 할당하려 한 경우, 특히 메모리에 제한 사항이 많은 컴퓨터에서. 예를 들어 `malloc(256 * 256 * sizeof(double))`과 같이. (덧붙여 질문 7.16, 19.23도 참고하시기 바랍니다.)

[1] 특히, stack-based 시스템에서는, 초기화되지 않는 변수의 실제 값은, 그 당시 stack에 어떠한 값이 있었느냐에 따라 달라집니다. 그렇기 때문에, 코드의 중간 중간 디버깅을 위해 출력 문장을 넣었을 때, 동작할 수도 있습니다. 즉, `printf`와 같은, 코드의 덩치가 큰 함수를 실행하면, 스택에 있는 값들이 매우 달라질 수 있기 때문입니다. **Note** 여기에서 말한 스택에 있는 값이란, 스택에 올바른 연산으로 쌓여 있는 값이 아니라, 빈 스택에 있는 쓰레기 값을 의미합니다.

- array overflow 문제들. 특히 작은 임시 버퍼를 쓸 때나 `sprintf()`를 써서 문자열을 만들 때. (덧붙여 질문 7.1, 12.21, 19.28도 참고하시기 바랍니다.)
- `typedef`된 타입을 특정 타입으로 잘못 예상하고 코딩한 경우, 특히 `size_t`에 대해. (질문 7.15 참고)
- 실수(floating point) 처리 문제 (덧붙여 질문 14.1, 14.4도 참고하시기 바랍니다.)
- 특정 시스템에서만 동작하게 되어 있는 교묘한(clever) 코드를 사용한 경우

올바른 함수 prototype을 사용한다면 이런 종류의 많은 문제를 미리 잡아낼 수 있습니다; `lint`를 쓰는 것도 좋은 방법입니다. 덧붙여 질문 16.3, 16.4, 18.4도 참고하시기 바랍니다.

Note 위에서 `printf`에 특히, `long int`를 잘못 전달했을 때, 그러한 문제가 발생할 수 있다고 쓴 것은, (대부분의 16-bit 시스템처럼) `int`와 `long int`가 실제 크기가 다른 시스템을 예상하고 말한 것입니다. (대부분의 32-bit 시스템처럼) 만약 `int`와 `long int`의 크기가 같은 경우라면 이러한 문제가 자주 발생하지는 않습니다만, 어차피 잘못된 코드이기 때문에 고쳐야 할 부분입니다. C99는 `long long int`를 지원하므로, `long int`와 `long long int`를 잘못 썼을 때 이런 문제가 발생할 수 있는 확률이 높다고 할 수 있습니다.

Q 16.6 왜 이 코드가 동작하지 않을까요?:

```
char *p = "hello, world!";
p[0] = 'H';
```

Answer 문자열 상수(string constant)도 상수입니다. 즉, 컴파일러는 문자열 상수를 쓰기가 금지된 곳에 배치시킬 수 있으므로, 이것을 변경하려는 시도는 위험합니다. 쓸(write) 수 있는 문자열이 필요하다면 쓸 수 있는 메모리를 배열을 선언하거나 `malloc`으로 할당해 주어야 합니다. 다음과 같이 해 보기 바랍니다:

```
char a[] = "hello, world!";
```

같은 이유에서, 다음과 같이, 전통적으로 쓰는, UNIX `mktemp` 쓰임새는 잘못된 것입니다:

```
char *tmpfile = mktemp("/tmp/tmpXXXXXX");
```

올바른 방법은 다음과 같습니다:

```
char tmpfile[] = "/tmp/tmpXXXXXX";
mktemp(tmpfile);
```

덧붙여 질문 1.32도 참고하시기 바랍니다.

References [ANSI] § 3.1.4
[C89] § 6.1.4
[H&S] § 2.7.4 pp. 31–2

Q 16.7 External structure를 푸는 코드를 받았습니다만, 실행하면 항상 "unaligned access"라는 에러가 납니다. 이게 무슨 뜻인가요? 실제 코드는 다음과 같습니다:

```
struct mystruct {
  char c;
  long int i32;
  int i16;
};

char buf[7], *p;
fread(buf, 7, 1, fp);
p = buf;
s.c = *p++;
s.i32 = *(long int *)p;
p += 4;
s.i16 = *(int *)p;
```

Answer The problem is that you're playing too fast and loose with your pointers. 어떤 시스템들은 데이터 값들이 정확히 align되어 있어야만 그 값을 access할 수 있습니다. 예를 들어, 2-byte `short int`는 2의 배수 형태를 띄는 주소값에서만 읽을 수 있으며, 4-byte `long int`는 4의 배수 형태를 띄어야만 access가 가능합니다. (질문 2.12 참고). (아무 곳이나 가리킬 수 있는) `char *` 타입의 포인터를 강제로 `int *`나 `long int *` 포인터로 바꿔 쓰면, 프로세서에게 multitype 값을 제대로 align되지 않은 곳에 access하게 만들게 됩니다.

주어진 역할을 좀 더 올바르게 수행할 수 있는 방법은 다음과 같습니다:

```
unsigned char *p = buf;

s.c = *p++;

s.i32 = (long *)p++ << 24;
s.i32 |= (long *)p++ << 16;
s.i32 |= (unsigned)(*p++ << 8);
s.i32 |= *p++;

s.i16 = *p++ << 8;
s.i16 |= *p++;
```

This code also gives you control over byte order. (This example, though, assumes that a `char` is 8 bits and the `long int` and `int` being unpacked from the "external structure" are 32 and 16 bits, respectively.) (비슷한 코드를 보여주는) 질문 12.42를 보기 바랍니다.

덧붙여 질문 4.5도 참고하시기 바랍니다.

References [ANSI] § 3.3.3.2, § 3.3.4
[C89] § 6.3.3.2, § 6.3.4
[H&S] § 6.1.3 pp. 164–5

Q 16.8 "Segmentation violation"이나 "Bus error"가 뭘까요? "core dump"는 또 뭘까요?

Answer 일반적으로 이런 메시지의 (또 memory-access violation 또는 protection fault와 비슷한 모든 메시지) 뜻은, 여러분의 프로그램이 access할 수 없는 메모리 공간을 (잘못된 포인터 사용으로) access하려 했다는 것을 의미합니다. 이런 메시지를 발생시킬 수 있는 몇가지 예는:

- 널 포인터를 잘못 썼을 때 (질문 5.2, 5.20 참고)
- 초기화하지 않았거나, 잘못 aligne된 주소, 또 할당을 올바르지 않게 한 포인터를 썼을 때 (질문 7.1, 7.2, 16.7 참고)
- 다른 곳으로 배치된(relocated) 곳을 가리켜야 하지만, 그렇지 않는 포인터를 (stale alias) 썼을 때 (질문 7.29 참고)
- `malloc`이 유지하는 내부 arena가 망가졌을 때 (질문 7.19 참고)
- (`const`로 선언되었거나, 문자열 상수처럼 — 질문 1.32 참고) 읽기 전용인 값을 변경하려 할 때.

- 잘못 전달된 함수 인자, 특히 포인터 관련; 두 가지 예를 들면, `scanf`를 잘 못 쓰거나 (질문 12.12), `fprintf`를 잘 못 썼을 때 (첫번째 인자가 `FILE *`라는 것을 꼭 검사하기 바람)

UNIX에서, 이런 모든 행위는 "core dump"가 납니다; 즉, `core`[2]라는 파일이 현재 디렉토리에 만들어 지며, 박살난 프로세스를 디버깅할 수 있도록 그 프로세스의 메모리 이미지가 저장되어 있습니다.

"bus error"와 "segmentation violation"의 차이는 중요하지 않습니다; 각각 다른 상황에서 각각 다른 버전의 UNIX는 이러한 시그널들을 만들어 냅니다. 간단히 말해, segmentation violation은, 아예 존재하지도 않는 메모리에 access하려 했다는 것이고, bus error는 메모리를 잘못된 방법으로 access하려 (대개 잘못된 align을 가지는 포인터, 질문 16.7 참고) 했다는 것을 뜻합니다.

덧붙여 질문 16.3, 16.4도 참고하시기 바랍니다.

[2]Yes, the name "core" derives ultimately from old ferrite core memories.

Chapter 17

Style

Q 17.1 C 언어 코딩을 할 때, 가장 좋은 스타일은 무엇일까요?

Answer K&R은 다음과 같은 단점을 가지고 있음에도, 가장 널리 쓰이는 스타일입니다:

> 대부분 사람들이 굳은 믿음을 가지고 있음에도, 중괄호의 ({}) 위치는 중요하지 않습니다. 그래서 우리는 인기있는 스타일 중 하나를 선택했습니다. 여러분에게 맞는 스타일을 찾은 다음, 그것을 일관성있게 사용하시기 바랍니다.

완전한 스타일을 찾는 것보다는, 하나의 스타일을 일관성있게 사용하는 것이 더 중요합니다. 만약 여러분이 처한 환경이 (예를 들어, 어떤 지역적인 관습이나, 회사 정책 등) 이러한 스타일을 정의하지 않았고, 또, 여러분이 새 스타일을 만들려는 마음이 없다면, 그냥 K&R을 쓰시기 바랍니다. (물론 중괄호의 위치나 들여쓰기의 간격에 대한 많은 논쟁이 있을 수 있지만, 여기에서 그런 논쟁을 하지는 않겠습니다. Indian Hill Style Guide를 참고하기 바랍니다.)

'좋은 스타일'이란 (good style) 단순하게 코드를 어디에 배치하느냐보다 훨씬 많은 것을 내포하고 있습니다; don't spend time on formatting to the exclusion of more substantive code quality issues.

덧붙여 질문 10.6도 참고하시기 바랍니다.

References [K&R1] § 1.2 p. 10
[K&R2] § 1.2 p. 10

Q 17.3 두 문자열이 같은지 비교하기 위해 다음과 같은 코드를 썼습니다:

```
if(!strcmp(s1, s2))
```

이것이 좋은 스타일일까요?

Answer 흔히들 그런 식으로 코드를 작성하기는 하지만 특별히 좋은 스타일이라고 말하기는 어렵습니다. 위의 코드는 두 문자열이 같은 경우를 검사한다는 점에서는 전혀 문제가 될 것이 없지만, ! 연산자는 대개 'not'을 의미하는 곳에 쓰이므로 혼동을 가져올 수 있기 때문입니다.

다음과 같은 매크로를 쓰는 것이 도움이 될 경우도 있습니다:

```
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
```

덧붙여 질문 17.10도 참고하시기 바랍니다.

Q 17.4 어떤 사람들은 `if (x == 0)`을 쓰지 않고 `if (0 == x)`와 같은 코드를 쓰는데, 그 이유는 무엇인가요?

Answer 다음과 같은 실수를 하는 것을 막기 위함입니다:

```
if (x = 0)
```

상수를 == 연산자의 왼쪽에 쓰는 습관을 들이면, 다음과 같은 실수를 했을 때, 컴파일러가 에러를 발생합니다:

```
if (0 = x)
```

어떤 사람들은 이런 방식을 쓰는 것을 외우는 것보다 두 개의 =를 쓰는 것을 외우는 것이 쉽기 때문에, 이런 방식을 좋아하지 않습니다. (물론, 이러한 스타일은 한 오퍼랜드(operand)가 상수일 경우에만 도움을 줄 수 있습니다.)

References [H&S] § 7.6.5 pp. 209–10

Q 17.5 제가 본 코드에는 `printf()`를 부를 때, 항상 `(void)`로 캐스팅하고 있는데요, 왜 그럴까요?

Answer `printf()`는 어떤 수치를 리턴하지만, 대개의 경우 쓰이지 않습니다. 어떤 컴파일러는 (특히 lint와 같은 프로그램도) 함수가 어떤 값을 리턴하는데, 이 값을 쓰지 않으면, 경고를 출력합니다. 따라서 `(void)`로 캐스팅해줌으로써, 이 리턴 값을 무시한다는 것을 컴파일러에게 알려, 경고를 출력하지 않게 할 수 있습니다.

`strcpy()`나 `strcat()`의 경우에도 같은 이유로 캐스팅이 사용되곤 합니다.

References [K&R2] § A6.7 p. 199
[ANSI Rationale] § 3.3.4
[H&S] § 6.2.9 p. 172, § 7.13 pp. 229–30

Q 17.8 "헝가리안 표기법"이란 (Hungarian Notation) 무엇인가요? 그리고 그것이 쓸만한 가치가 있나요?

Answer 헝가리안 표기법은 Charles Simonyi씨가 만든 (변수의 이름에 변수의 타입에 대한 정보를 포함하는) 이름 짓는 방법입니다. 어떤 사람들에게는 매우 사랑받는 표기법이며, 어떤 사람들에게는 매우 배척받는 표기법입니다. 이 표기법의 가장 큰 장점은 변수의 이름만 보고도, 변수의 타입과, 쓰임새를 알 수 있는 것입니다; 가장 큰 단점은 이름을 짓는 데 타입에 관한 정보가 별로 중요하지 않다는 것입니다.

References Simonyi and Heller, "The Hungarian Revolution".

Note 헝가리안 표기법은 특히 Microsoft사의 Win32 API를 쓰는 프로그래머들에게 사랑받고 있습니다. (API 자체도 이 표기법을 사용하고 있습니다.) 대개의 UNIX 프로그래머들은 이를 배척하고 있습니다.

Q 17.9 "Indian Hill Style Guide"나 이와 유사한 코딩 스타일에 관한 표준을 얻을 수 있는 곳이 있나요?

Answer Anonymous FTP를 써서 여러 곳에서 얻을 수 있습니다.

Site:	File or directory
`ftp.cs.washington.edu`	`pub/cstyle.tar.Z` (최신의 Indian Hill Guide)
`ftp.cs.toronto.edu`	`doc/programming` (Henry Spencer씨의 "10 Commandments for C Programmers" 포함)
`ftp.cs.umd.edu`	`pub/style-guide`

"The Elements of Programming Style", "Plum Hall Programming Guidelines", "C Style: Standards and Guidelines"라는 책이 도움이 될 수 있습니다: '참고문헌'란을 보기 바랍니다.

덧붙여 질문 18.9도 참고하시기 바랍니다.

Q 17.10 어떤 사람들은 `goto`가 악마와도 같다고 말하면서 절대로 쓰지 말라고 하는데, 너무 과장된 말이 아닐까요?

Answer 프로그래밍 스타일이란, 글을 쓰는 스타일과 같이 예술과도 같은 것이기에, 어떠한 강경한 규칙으로 제정될 수 없는 것입니다, although discussions about style often seem to center exclusively around such rules.

`goto` 문장은 남용되면 유지/보수/관리하기 힘들 정도의 복잡한, 이른바 스파게티 코드를 (spaghetti code) 만들어 낼 수 있습니다. 그렇다고, 아무런 생각없이 `goto` 문장을 무조건 금지하는 것이 즉각 좋은 코드를 만들어 내는 것도 아닙니다: 물론 `goto`를 전혀 쓰지 않고도 (이상해 보이는 중첩된 루프나, 불리언 제어 변수를 써서) 코드를 만들어 낼 수도 있습니다.

대부분 프로그래밍 스타일에 관한 "규칙(rule)"은 규칙이란 단어보다는 "안내지침(guideline)"이란 단어로 보는 게 더 바람직합니다. 또한 이러한 규칙들이 왜 만들어졌는 가를 이해하는 것이 더 중요합니다. 이러한 규칙에 대한 이해없이, 무조건 어떤 구조를 배척하는 것은 원래 규칙이 의미하고자 한 것과 오히려 반대되는 결과를 만들어낼 수 있습니다.

게다가 프로그래밍 스타일에 대한 많은 의견은 단지 의견일 뿐입니다. 따라서 이러한 "스타일 전쟁"에 참여하는 것은 쓸데없는 짓입니다. (질문 9.2, 5.3, 5.9, 10.7에 이러한 내용이 있습니다.) 여러분의 의견에 반대하는 사람은 결코 여러분의 의견에 동의하지 않을 것이며, 여러분의 의견에 동조하는 사람은 결코 반대하지 않을 것이므로, 이런 논쟁을 할 필요가 없습니다.

Chapter 18

Tools and Resources

Q 18.1 도움이 될만한 프로그램이 필요합니다.

Answer 아래의 테이블을 참고하기 바랍니다 (덧붙여 질문 18.16도 참고하시기 바랍니다.):

cross-reference generator	cflow, cxref, calls, cscope, xscope, ixfw
C beautifier/pretty printer	cb, indent, GNU indent, vgrind
버전/설정 관리	CVS, RCS, SCCS
C source obfuscator	obfus, shroud, opqcp
"make" dependency generator	makedepend, `cc -M`, `cpp -M`
compute code metrics	ccount, Metre, lcount, csize,
C lines-of-source counter	wc, `grep -c ";"`
C declaration aid	`comp.sources.unix`의 Volumn 14. [K&R2] 참고
tracking down `malloc` problem	질문 18.2 참고
"selective" C preprocessor	질문 10.18 참고
language translation tool	질문 11.31 참고
C verifier (lint)	질문 18.7 참고
C compiler	질문 18.3 참고

(당연히 이 테이블이 모든 것을 나타내 주는 것은 아닙니다; 만약 여기에 나온 것 이외에 더 많은 것을 알고 있다면 관리자에게 연락하기 바랍니다.)

Usenet `comp.compilers`와 `comp.software-eng`에서 여러 툴에 대한 다른 목록과, 논쟁을 찾아 볼 수 있습니다.

덧붙여 질문 18.3, 18.16도 참고하시기 바랍니다.

Note 위 테이블의 code metric에 관한 것은 다음 URL을 참고하시기 바랍니다:

`http://www.qucis.queensu.ca/Software-Engineering/Cmetrics.html`

GNU indent에 관한 것은 아래 URL을 참고하시기 바랍니다:

`http://www.gnu.org/software/indent/`

Q 18.2 `malloc`에서 문제가 발생한 것 같은데 검사를 쉽게 해주는 툴은 없나요?

Answer `malloc` 문제를 도와주는 여러가지 디버깅 패키지들이 있습니다. 인기있는 것 중의 하나는 conor p. Cahill씨의 "dbmalloc"이며, 1992년 `comp.sources.misc`의 volumn 32에서 구할 수 있습니다. 또 `comp.sources.unix` volumn 27에서 얻을 수 있는 "leak"도 좋습니다; JMalloc.c and JMalloc.h in the "Snippets" collection; and MEMDEBUG from ftp.crpht.lu in pub/sources/memdebug. See also question 18.16.

A number of commercial debugging tools exist, and can be invaluable in tracking down malloc-related and other stubborn problems:

- Bounds-Checker for DOS, from Nu-Mega Technologies, P.O. Box 7780, Nashua, NH 03060-7780, USA, 603-889-2386.
- CodeCenter (formerly Saber-C) from Centerline Software, 10 Fawcett Street, Cambridge, MA 02138, USA, 617-498-3000.
- Insight, from ParaSoft Corporation, 2500 E. Foothill Blvd., Pasadena, CA 91107, USA, 818-792-9941, insight@parasoft.com .
- Purify, from Pure Software, 1309 S. Mary Ave., Sunnyvale, CA 94087, USA, 800-224-7873, http://www.pure.com , info-home@pure.com . (I believe Pure was recently acquired by Rational.)
- Final Exam Memory Advisor, from PLATINUM Technology (formerly Sentinel from AIB Software), 1815 South Meyers Rd., Oakbrook Terrace, IL 60181, USA, 630-620-5000, 800-442-6861, info@platinum.com, www.platinum.com .
- ZeroFault, from The Kernel Group, 1250 Capital of Texas Highway South, Building Three, Suite 601, Austin, TX 78746, 512-433-3333, http://www.tkg.com, zf@tkg.com .

Note 요즈음에는 다음과 같은 툴들이 인기가 많습니다:

- Electric Fence (또는 줄여서 efence) — C 프로그램에서 buffer overrun과 underrun을 검사해 줍니다.

- Valgrind — 다양한 메모리 에러를 검사해 주며, 코드를 다시 컴파일할 필요가 없습니다. 다만 현재 x86 프로세스를 쓰는 Linux 시스템에서만 쓸 수 있습니다.

Q 18.3 저렴하거나 공짜로 구할 수 있는 컴파일러가 있을까요?

Answer 인기있고, 공짜로 구할 수 있으며, 고 품질인 FSF의 GNU C 컴파일러 (또는 gcc라고 불리움)를 쓰면 됩니다. 이는 `prep.ai.mit.edu`의 `pub/gnu` 디렉토리나 기타 GNU 아카이브 사이트에서 구할 수 있습니다. MS-DOS 용으로 포팅한 djgpp도 있으며, `http://www.delorie.com/djgpp/`에서 얻을 수 있습니다.

PCC라는 쉐어웨어(shareware) 컴파일러도 있으며, 이름은 `PCC12C.ZIP`입니다.

MS-DOS용 컴파일러로 매우 싼 가격에 구할 수 있는 Power C는 Mix Software에서 만들었고, 주소는 1132 Commerce Drive, Richardson, TX 75801, USA, 전화번호는 214-783-6001입니다.

최근에 개발된 컴파일러로는 lcc가 있습니다. Anonymous FTP로 `ftp.cs.princeton.edu`의 `pub/lcc`에서 구할 수 있습니다.

`ftp.hitech.com.au`의 `hitech/pacific`에서 쉐어웨어 MS-DOS용 컴파일러를 구할 수 있습니다. 사고파는 목적이 아니라면 등록은 할 필요가 없습니다.

매킨토시 용으로 구할 수 있는 쉐어웨어 컴파일러는 알려진 것이 없습니다.

`comp.compilers` archive에는 compiler, interpreter, grammer등에 대한 굉장히 많은 정보가 들어 있습니다. (FAQ 리스트를 포함한) `comp.compilers` archive는 iecc.com에서 근무하는 moderator인 John R. Levine씨에 의해 관리되고 있습니다. 사용 가능한 컴파일러들과 이에 관련된 자료는 Mark Hopkins, Steven Robenalt, 그리고 David Muir Sharnoff씨에 의해 관리되며 `ftp.idiom.com`의 `pub/compilers-list/`에 있습니다. (`rtfm.mit.edu` 또는 `ftp.uu.net`의 `comp.compilers` 디렉토리에서 `news.answers`에 관한 아카이브도 참고하기 바랍니다; 질문 20.40도 참고하기 바랍니다.)

Note 위 답변은 상대적으로 오래된 것입니다. GCC는 원래 "GNU C Compiler"의 약자이었지만, C 언어 뿐만 아니라 C++, java와 같은 언어도 지원하기 때문에 이름이 "GNU Compiler Collection"으로 이름이 바뀌었습니다. GCC와 더불어 GNU software를 구하시려면 `ftp.gnu.org`의 `/pub/`에서 구할 수 있으며, 국내에서는 `ftp.bora.net`의 `/pub/gnu/`에서 download 받는 것이

빠릅니다. 자세한 것은 GCC의 홈페이지인 아래 사이트를 방문하시면 얻을 수 있습니다:

```
http://www.gnu.org/software/gcc/
```

lcc는 표준 C 언어로 씌여진, 이식성이 매우 우수한(retargetable) C 컴파일러이며, ALPHA, SPARC, MIPS R3000, Intel x86용 코드를 만들어 냅니다. lcc는 소스가 다른 컴파일러에 비해 상대적으로 짧고 읽기가 쉬워서 컴파일러 교육용으로 많이 쓰입니다. 아래 사이트를 방문하시면 좀 더 많은 정보를 얻을 수 있습니다:

```
http://www.cs.princeton.edu/software/lcc/
```

덧붙여 질문 18.16도 참고하시기 바랍니다.

Q 18.4 프로그램을 동작시켰는데, 매우 이상하게 동작합니다. 잘못된 점을 어떻게 찾을 수 있죠?

Answer 먼저 lint를 돌려보시기 바랍니다. (아마도 `-a`, `-c`, `-h`, `-p` 등의 옵션을 함께 써야 할 것입니다.) 대부분의 C 컴파일러는 완벽한 컴파일러가 아닙니다. 다른 말로, 만약 프로그래머가, 할 일을 명확하게 지시하지 않았거나, 지시한 사항이 동작한다고 보장할 수 없는 경우에 무시하고 넘어가는 경우가 많습니다. (그런 경우에도, 컴파일러가 제공하는 경고 메시지를 최대로 출력하도록 해 보기 바랍니다.)

덧붙여 질문 16.5, 16.8, 18.7도 참고하시기 바랍니다.

References [Darwin]

Q 18.5 'lint'를 실행시키면 `malloc()`을 부를 때마다 "warning: possible pointer alignment problem"라는 경고가 발생하는데, 어떻게 이 경고좀 발생하지 않게 할 수 없을까요?

Answer 오래된 lint의 경우, `malloc()`이 "어떠한 타입의 object도 저장할 수 있는 공간을 가리키는 포인터를 리턴한다"라는 사실을 모르기 때문에 그런 경고가 나옵니다. 한가지 방법은 `#ifdef lint` 안에 `#define`을 써서 `malloc()`에 대한 'pseudo-implementation'을 만들어 두는 것입니다. 그렇지만 정말로 중요한 경고 메시지도 보여주지 않을 가능성이 있기 때문에 좋은 방법이라 할

수는 없습니다. 차라리 'grep -v'를 써서 그런 메시지를 없애버리는 것이 훨씬 더 간단하고 안전할 것입니다. (그러나 'lint'가 보여주는 많은 메시지들을 무시하는 습관을 들이는 것은 매우 위험합니다. 어느날 중요한 메시지를 못 보고 넘어가는 경우도 생길지 모릅니다.)

Q 18.7 ANSI 호환의 lint를 구할 수 있을까요?

Answer 대부분 시스템에서 쓸 수 있는 'PC-Lint'와 'FlexeLint'가 있으며, 아래 주소에서 구할 수 있습니다:

```
Gimpel Software
3207 Hogarth Lane
Collegeville, PA  19426  USA
(+1) 610 584 4261
gimpel@netaxs.com
```

Unix System V release 4의 lint는 ANSI 호환입니다. 그리고 UNIX Support Labs나 System V 판매업체에 의해 각각 다른 패키지 형태 (다른 C tool들이랑 같이 번들 형태)로 제공됩니다.

고수준으로 문법을 검사해주는 ANSI 호환의 다른 lint로는 'LCLint'가 있으며, `larch.lcs.mit.edu`의 `pub/Larch/lclint`에서 구할 수 있습니다 (via anonymous ftp).

그리고 최근에 제공되는 많은 컴파일러들은 lint가 제공하는 것처럼 자세히 소스를 분석하고 알려주는 기능이 있습니다. ('gcc'의 경우는 '-Wall -pedantic' 옵션을 쓰면 됩니다.)

Note lclint는 GPL[1]로 라이센스되어 있으며, 홈페이지는 다음과 같습니다:

```
http://lclint.cs.virginia.edu/
```

2002년 이후, lclint는 splint로 이름을 바꾸었으며, 홈페이지는 다음과 같습니다:

```
http://www.splint.org/
```

Q 18.8 ANSI function prototype이 있으니깐 이제 lint는 필요없지 않나요?

[1] GNU General Public License

Answer 아닙니다. 무엇보다도 prototype은 존재하고 있고, 올바로 주어질 때에만 동작합니다; 잘못된 prototype은 아예 없는 것이 낫습니다. 두번째로 lint는 여러 소스 파일들을 읽고 일관성을 검사해 줄 수 있고, 함수뿐만 아니라 데이터 선언까지 검사해 줍니다. 게다가 lint와 같이 시스템에 독립적인 프로그램을 사용하면 좀더 정확한 분석을 받을 수 있기 때문에, 컴파일러의 특별한 기능이나, 특정 시스템 기능에 의존하지 않는, 이식성이 높은 프로그램을 작성하는데 큰 도움이 됩니다.

'lint'를 쓰는 대신 function prototype을 쓰길 원한다면, 먼저 여러 파일들이 같은 prototype을 지녔는지 검사하고, 정확한 prototype을 썼는지 검사해야 합니다. 질문 1.7과 10.6을 참고하기 바랍니다.

Q 18.9 인터넷상에서 구할 수 있는 C 언어 안내서나 기타 자료는 없나요?

Answer 물론 많습니다:

- Tom Torfs씨는 매우 좋은 튜토리얼을 가지고 있습니다. `http://members.xoom.com/tomtorfs/cintro.html`.
- Christopher Sawtell씨가 만든 "Notes for C programmers"는 아래 주소에서 구할 수 있습니다:
  - `svr-ftp.eng.cam.ac.uk`의 `misc/sawtell_C.shar`
  - `garbo.uwasa.fi`의 `/pc/c-lang/c-lesson.zip`
- Tim Love씨의 "C for Programmers"는 `svr-ftp.eng.cam.ac.uk`의 `misc` 디렉토리에서 구할 수 있습니다. HTML 버전은 아래에서 구할 수 있습니다:

  `http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/`
- Coronado Enterprises C tutorial은 Simtel 미러 사이트의 `pub/msdos/c` 또는 `http://www.swcp.com/ dodrill`에서 구할 수 있습니다.
- Rick Rowe씨의 tutorial은 `ftp.netcom.com`의 `pub/rowe/tutorde.zip` 또는 `ftp.wustl.edu`의 `pub/MSDOS_UPLOADS/programming/c_language/ctutorde.zip` 에서 구할 수 있습니다.
- Web용으로 만들어진 코스로는:

  `http://www.strath.ac.uk/CC/Courses/CCourse/CCourse.html`
- Martin Brown씨는 C 코스 자료를 Web에 게시했습니다:

  `http://www-isis.ecs.soton.ac.uk/computing/c/Welcome.html`

- 어떤 Unix 시스템에서는 `learn c`라고 shell prompt에서 입력하면 간단하게 가르쳐 줍니다 (그러나 상당히 오래된 내용일 것입니다).
- 마지막으로 이 FAQ 목록의 저자도 C 언어 교육을 하고 있으며, 그 노트를 web에서 구할 수 있습니다;

  `http://www.eskimo.com/~scs/cclass/cclass.html`

  [Disclaimer: 여기에 있는 많은 tutorial들은 저자가 다 검토해 본 것은 아닙니다. 또한 몇 가지 에러가 있는 것으로 압니다. 이 글들에 저자의 이름이 들어 있기는 하지만 어떠한 것도 보증할 수 없습니다. 또한 이런 종류의 글들은 대개 시간이 지날 수록 값어치가 떨어지기 때문에 여러분이 읽으려고 할 때에는 이미 너무 구식의 글일지도 모릅니다.]
- 이러한 tutorial들과 C 언어에 관한 여러가지 정보는 아래 URL에서 구할 수 있습니다:

  `http://www.lysator.liu.se/c/index.html`
- Vinit Carpenter씨는 C와 C++을 배울때 쓸모있는 여러가지 자료에 대한 목록을 유지하고 있습니다. 이 목록은 newsgroup `comp.lang.c`와 `comp.lang.c++`에 게시되며 이 FAQ list가 있는 곳에 (질문 20.40 참고), 같이 보관되어 있으며, 아래에서 구할 수 있습니다.

  `http://www.cyberdiem.com/vin/learn.html`

덧붙여 질문 18.10, 18.15c도 참고하시기 바랍니다.

Note 원 저자의 홈페이지의 다음 링크에서 관련된 자료를 찾아볼 수 있습니다:

`http://www.eskimo.com/~scs/C.html`

Q 18.10 C 언어를 배우려고 하는데 좋은 책좀 추천해 주세요.

Answer C 언어에 관한 책들은 이 글에서 모두 다루기에는 너무나도 많습니다; 따라서 이들 책 모두를 평가해본다는 것은 너무나도 힘이 듭니다. 많은 사람들이 가장 처음 만들어진 책이 가장 좋다고 말하며 *The C Programming Language*를 추천합니다. 이 책은 Kernighan씨와 Ritchie씨가 썼습니다 ("K&R"이라고도 하며, 현재 두번째 판이 나와 있습니다). 그러나 K&R이 초보자가 보기에는 적당하지 않다는 의견도 있습니다; 그렇지만 우리들 대부분이 그 책으로 C를 배웠으며, 아주 잘 배웠다고 생각합니다; 그러나 책의 내용이 너무나 딱딱하기 때문에 프로그래밍에 대한 배경 지식이 전혀 없는 사람에게는 무리일 것입니다. 이 책에 대한 여러 보충 설명이나 정정 목록은 아래에서 구할 수 있습니다:

- http://www.csd.uwo.ca/ jamie/.Refs/.Footnotes/C-annotes.html
- http://www.eskimo.com/ scs/cclass/cclass.html
- http://www.lysator.liu.se/c/c-errata.html#main

comp.lang.c의 많은 사람들은 K.N. King씨가 쓴 *C: A Modern approach*를 추천합니다.

가장 좋은 참고 서적은 *C: A Reference Manual*이며, Samuel p. Harbison씨와 Guy. L. Steele씨가 썼습니다. 현재 네번째 판이 나와 있습니다.

C 언어를 처음부터 배우기에는 적당하지 않지만 이 FAQ 목록도 출판되어 책으로 나와 있습니다. 이 글의 마지막 "저서 목록"의 [Summit]를 참고하시기 바랍니다.

Mitch Wright씨는 C와 UNIX에 관한 많은 참고서에 대한 목록을 가지고 있으며, ftp.rahul.net의 pub/mitch/YABL에서 구할 수 있습니다.

Scott McMahon씨는 여러가지 비평에 대한 글을 가지고 있습니다:

```
http://www.skwc.com/essent/cyberreviews.html
```

'Association of C and C++ Users (ACCU)'는 C/C++에 대한 많은 책들에 대한 논평을 모아 놓았습니다:

```
http://bach.cis.temple.edu/accu/bookcase
http://www.accu.org/accu
```

이 FAQ 목록의 편집자도 권장하는 많은 책에 대한 정보를 가지고 있습니다; 필요하면 보내드리고 있습니다. 덧붙여 질문 18.9도 참고하시기 바랍니다.

Note *C: A Reference Manual*은 현재 다섯번째 판이 나와 있습니다. 자세한 것은 다음 사이트를 참고하기 바랍니다:

```
http://www.careferencemanual.com/
```

Q 18.13 표준 C 라이브러리에 대한 소스를 구할 수 있을까요?

Answer (public domain은 아니지만) P.J. Pluager씨의 *The Standard C Library*가 있습니다. 모든 C 라이브러리에 대한 소스 코드는 NetBSD나 GNU (물론 Linux도) project의 일부분으로 제공됩니다. 덧붙여 질문 18.15c, 18.16도 참고하시기 바랍니다.

Note GNU C 라이브러리의 홈페이지는 아래와 같습니다:

```
http://www.gnu.org/software/glibc/
```

Q 18.13b 온라인 C reference manual이 있을까요?

Answer 두 개가 있습니다:

- `http://www.cs.man.ac.uk/standard_c/_index.html`
- `http://www.dinkumware.com/htm_cl/index.html`.

Q 18.13c ANSI/[C89] C 표준을 구할 수 있을까요?

Answer 질문 11.2를 보기 바랍니다.

Q 18.14 수식(expression)을 파싱(parsing)하고 평가하는 코드가 필요합니다.

Answer 두 개의 패키지가 각각 1993년 12월에 `comp.sources.misc`에 1994년 1월에 `alt.sources`에 게시되었지만, 지금은 없고, 대신 `sunsite.unc.edu`의 다음 위치에서 얻을 수 있습니다:

```
pub/packages/development/libraries/defunc-1.3.tar.Z
```

또, `lamont.ldgo.columbia.edu`에서 "parse" 패키지도 찾아보기 바랍니다. 여러가지 옵션들이 포함되어 있는 S-Lang 인터프리터는 anonymous ftp로 `amy.tch.harvard.edu`의 `/pub/slang`에서 구할 수 있습니다. 또한 Cmm ("C-minus-minus")이라는 쉐어웨어도 있습니다. 덧붙여 질문 18.16, 20.6도 참고하시기 바랍니다.

*Software Solutions in C*의 Chapter 12, 페이지 235–55에서 파싱/평가하는 코드를 찾아볼 수 있습니다.

Q 18.15 C 언어에 대한 BNF 또는 YACC 문법을 구할 수 있을까요?

Answer 완벽한 문법은 물론 ANSI 표준에 들어 있습니다; 질문 11.2를 보기 바랍니다. 다른 문법으로는 (C++과 관련된) Jim Roskind씨가 만든 것인데 `ics.uci.edu`의 `pub/c++grammar1.1.tar.Z`로 얻을 수 있습니다. 아마도 `ftp.ics.uci.edu`, `OLD/pub/c++grammar1.1.tar.Z`일지도 모릅니다. 또는 `ftp.eskimo.com`의 `u/s/scs/roskind_grammar.Z`로 구할 수 있습니다. 작업중인 ANSI 문법은

(Jeff Lee씨가 작업하는) ftp.uu.net의 (질문 18.16 참고) usenet/net.sources/ansi.c.grammer.Z에서 구할 수 있으며, lexer가 포함되어 있습니다. [K&R2]의 부록이나 FSF의 GNU C compiler에서도 문법을 얻을 수 있습니다.

comp.compilers 아카이브는 문법에 대한 많은 정보를 가지고 있습니다; 질문 18.3을 보기 바랍니다.

References [K&R1] § A18 pp. 214–219
[K&R2] § A13 pp. 234–239
[C89] § B.2
[H&S] pp. 423–435 Appendix B

Q 18.15b C 컴파일러를 테스트해 볼 수 있는 tool들은 없을까요?

Answer A: Plum Hall (formerly in Cardiff, NJ; now in Hawaii) sells one; other packages are Ronald Guilmette's RoadTest(tm) Compiler Test Suites (ftp to netcom.com, pub/rfg/roadtest/announce.txt for information) and Nullstone's Automated Compiler Performance Analysis Tool.

```
http://www.nullstone.com/
```

The FSF's GNU C (gcc) distribution includes a c-torture-test which checks a number of common problems with compilers. Kahan's paranoia test, found in netlib/paranoia on netlib.att.com, strenuously tests a C implementation's floating point capabilities.

Q 18.15c 쓸모 있는 코드나 예제를 모아놓은 것은 없을까요?

Answer Bob Stout씨의 "SNIPPETS"는 매우 인기있으며, ftp.brokersys.com의 pub/snippets나 http://www.brokersys.com/snippets/에 있습니다.

Lars Wirzenius씨의 'public' 라이브러리는 ftp.funet.fi의 다음 위치에 있습니다:

```
pub/language/C/Publib/
```

덧붙여 질문 14.12, 18.9, 18.13, 18.16도 참고하시기 바랍니다.

Note SNIPPETS의 홈페이지는 아래로 변경되었습니다:

```
http://www.snippets.org/
```

Q 18.15d "multiple precison arithmetic"을 위한 코드가 필요합니다.

Answer BSD Unix libc 소스에 있는 "quad" 함수들은 매우 인기있는 패키지입니다. (`ftp.uu.net`의 다음 위치에 있습니다:

```
/systems/unix/bsd-sources/ .... /src/lib/libc/quad/*
```

GNU MP 라이브러리, 또 MIRACL 패키지 (`http://indigo.ie/ mscott/` 참고), 그리고 오래된 Unix에서 제공되는 libmp.a를 참고 바랍니다. 덧붙여 질문 14.12, 18.16도 참고하시기 바랍니다.

References [Dale] § 17 pp. 343–454

Q 18.16 이러한 공개용 프로그램들을 어떻게 얻을 수 있나요?

Answer 프로그램 숫자도 갈수록 증가하고, 이런 프로그램들을 정리해 놓은 보관(archive) 사이트들도 갈수록 증가하고 있고, 이러한 프로그램들을 쓰려는 사람들도 덩달아 많아지고 있기 때문에 이 질문은 매우 대답하기 쉬우면서도 어렵습니다.

널리 알려진, 공개를 취지로 한 archive 사이트들을 예로 들면, `ftp.uu.net`, `archive.wustl.edu`, `oak.oakland.edu`, `sumex-aim.stanford.edu`, 그리고 `wuachive.wustl.edu`가 있습니다. 이 사이트들은 모두 많은 소프트웨어를 가지고 있으며, 많은 정보들이 공개되어 있습니다. FSF의 GNU 프로젝트의 주 사이트는 `prep.ai.mit.edu`입니다. 이 사이트는 정말로 바쁘기 때문에 접속이 거부당할 수도 있지만, 그만큼 많은 미러 사이트가 있으므로 참고하기 바랍니다.

인터넷을 통하여 이런 사이트에서 파일을 받을 수 있는 가장 일반적인 방법은 anonymous ftp를 쓰는 것입니다. ftp를 쓸 수 없다면, mail로 ftp를 쓸 수 있게 해주는 ftp-by-mail 서버가 많이 있으므로 이것을 쓰면 됩니다. 게다가 world-wide web (WWW)는 많은 데이터 파일들을 전송하고 인덱싱하고 알리는 데 쓰이고 있으며, 데이터를 얻는 가장 새로운 방법입니다.

그리고 이 질문에 대한 답은 쉬운 편도 있지만, 어려운 것은 세부적인 내용에 대한 것입니다. 즉 쓸 수 있는 archive 사이트들을 모두 알려주거나 이 사이트에서 데이터를 받아보는 방법을 알려 주는 것은 불가능합니다. 인터넷에 접속할 수 있다면 여러분은 이 글보다 훨씬 새롭고 알찬 사이트 및 접근 방법을 이미 알고 있을 것입니다.

또 여러분이 찾고자 하는 것이 어디에 있는 지 알려주기도 쉬운 일이 아닙니다. 이미 이러한 일을 처리할 수 있는 새로운 인덱싱 서비스가 널려 있습니다. Alta Vista나 Excite, Yahoo가 그 예입니다.

유즈넷을 쓸 수 있다면 comp.sources.unix나 comp.sources.misc 뉴스 그룹에 정기적으로 게시되는, 보관 정책과 아카이브에 접속하는 방법을 읽어 보시기 바랍니다. 이 그룹의 글을 볼 수 있는 두 개의 사이트를 소개하면:

- ftp://gatekeeper.dec.com/pub/usenet/comp.sources.unix/
- ftp://ftp.uu.net/usenet/comp.sources.unix/

뉴스 그룹 comp.archives은 여러가지 주제에 대한 많은 글을 게시하고 있습니다. 마지막으로 뉴스 그룹 comp.sources.wanted에서는 소스를 찾고자 할 때 질문을 게시할 수 있는 더 적당한 곳입니다. 그러나 먼저 이 뉴스 그룹의 FAQ 목록을 읽고 어떻게 소스를 찾을 수 있는지 알고 나서 글을 게시하기 바랍니다.

덧붙여 질문 14.12, 18.13, 18.15c도 참고하시기 바랍니다.

Note FSF의 GNU 프로젝트의 공식 홈페이지는 다음과 같습니다:

http://www.gnu.org/

Chapter 19

System Dependencies

Q 19.1 키보드에서 RETURN 키를 누르지 않고 바로 한 글자를 읽으려면 어떻게 해야 하나요? 또 키를 누를때 스크린에 그 문자가 찍히지 않게 하려면 어떻게 하죠?

Answer 이 문제에 대한 표준, 또는 이식성이 뛰어난 방법은 없습니다. screen과 keyboard에 관한 것은 아예 C 표준에 한마디도 나와 있지 않습니다. 대신 간단한 I/O (문자로 이루어진) 스트림(stream)에 대한 것만 나와 있습니다.

Interactive한 키보드 입력은 대개 여러 입력이 모아져서 프로그램에 한 줄씩 전달되게 됩니다. 이런 까닭은 운영 체제가 입력된 줄을 편집할 (backspace/delete 등) 수 있는 일관된 방법을 (편집 기능을 일일히 코딩하지 않더라도) 제공하기 위해서 입니다. 즉 사용자가 RETURN 키를 눌렀을 때에 비로소, 한 줄이 프로그램에 전달됩니다. 프로그램이 한 문자씩 읽어들이게 짜여져 있다고 하더라도 한 문자를 읽어들이는 함수(예를 들면 `getchar()`)가 호출될 때, 프로그램이 중단되고, 사용자에게 한 줄을 입력받은 다음, 각각의 문자가 빠른 속도로 그 함수에게 전달됩니다.

질문한 내용처럼 한 문자가 읽혀지는 즉시, 프로그램에 전달되게 하려면. 한 줄씩 전달되게 하는 그 logic을 멈추게 해야 합니다. 어떤 시스템 (예를 들면 MS-DOS, VMS의 어떤 모드)에서는 프로그램이 OS 수준의 입력 함수를 불러서 해결하기도 하며, 어떤 시스템 (예를 들면 Unix, VMS의 어떤 모드)에서는 입력을 처리하는 (대개 "터미널 드라이버"라고 부르는) OS의 부분에다 줄 단위 처리 기능을 끄라고 알려야 합니다. 그리고 일반적인 입력 함수(예를 들면 `read()`, `getchar()`, 등)를 써서 한 문자씩 읽습니다. 또 어떤 시스템에서는 (특히 오래된 배치 프로세싱[1] 시스템) 입력 처리가 주변 장치에 의해

[1] batch-oriented mainframe

처리되며, 줄 단위 입력만 처리할 수 있어, 다른 방식을 쓸 수 없는 경우도 있습니다.

그러므로, 문자 단위로 처리하고 싶거나 (키보드 에코 기능을 끄고 싶다면), 여러분이 쓰고 있는 시스템에 의존적인 특수한 방법을 써야 합니다. `comp.lang.c`는 C 언어가 정의하고 있는 기능에 대한 것을 다루는 곳이므로 이 질문을 다루기에는 적당하지 않습니다. 시스템에 의존적인 다른 뉴스 그룹, 예를 들면 아래 뉴스 그룹에 묻는 것이 좋습니다:

- `comp.os.msdos.programmer`,
- `comp.unix.questions`

또는 그러한 그룹의 FAQ 목록을 보는 것도 좋은 방법입니다. 또 이런 방법은 시스템의 여러 종류마다 각각 차이가 있을 수 있으므로, 참을성을 가지고 기다려야 할 지도 모릅니다.

그러나 이러한 질문은 자주 이 곳에 게시되므로, 일반적인 경우에 처리할 수 있는 방법을 간단히 소개합니다.

어떤 버전의 curses에서는 `cbreak()`, `noecho()`, `getch()`와 같은, 질문의 목적에 맞는 함수를 제공합니다. 간단히, 짧은 암호를 입력할 수 있는 루틴이 필요하다면 `getpass()`를 쓰면 됩니다.

UNIX에서는 `ioctl()` 함수를 써서 터미널 드라이버 모드를 제어할 수 있습니다. ("classic" 버전에서는 `CBREAK`나 `RAW`를 쓰고, System V나 POSIX 시스템에서는 `ICANON`, `c_cc[VMIN]`, 그리고 `c_cc[VTIME]`을 쓰고, 대부분 시스템에서는 `ECHO`를 쓸 수 있습니다. 또 `system()`과 stty 명령도 쓸 수 있습니다. (좀 더 자세한 설명을 원한다면, "classic" 버전에서는 `<sgtty.h>`와 tty(4)를, System V에서는 `<termio.h>`와 termio(4)를, POSIX 시스템에서는 `<termios.h>`와 termios(4)를 찾아보시기 바랍니다.)

MS-DOS에서는 `getch()`나 `getche()`를 쓰거나, BIOS 인터럽트에 해당하는 함수를 쓰면 됩니다.

VMS에서는 스크린 관리(Screen Management) 루틴(SMG$)이나 curses, 또는 IO$_READVBLK 함수를 통해, 저수준 $QIO를 (또는 IO$M_NOECHO) 쓰면 됩니다. (VMS 터미널 드라이버에서는 한 번에 한 글자씩[2] 읽거나 "pass through" 모드를 쓸 수 있습니다.)

다른 운영체제에서 하는 방법은 직접 찾아보시기 바랍니다.

(덧붙이면, 단순히 `setbuf()`나 `setvbuf()`를 써서 stdin이 버퍼링되지 않게 하는 것은 한 번에 한 글자씩 받아 들이는 것과 별 상관이 없습니다.)

[2] character-at-a-time

이식성이 뛰어난 프로그램을 만들고자 할 때 가장 좋은 방법은 이러한 함수들을 정의하고 (1) 터미널 드라이버나 입력 시스템을 "한 번에 한 글자씩" 읽도록 설정하고, (2) 문자를 입력받고, (3) 프로그램을 끝낼 때 원래의 터미널 모드로 복원시키는 것입니다. (이론상, 이런 종류의 함수들은 언젠가 C 표준에 포함될 것입니다.) 이 FAQ 목록의 확장판(질문 20.40 참고)에서는 여러 시스템에서 이러한 함수를 만드는 법에 대한 예제가 나와 있습니다.

... TODO ... 덧붙여 질문 19.2도 참고하시기 바랍니다.

Note `ioctl()`을 쓰는 가장 주된 이유는 Terminal I/O를 제어하기 위한 것이며, [Lewine] 에 의하면 POSIX.1에서 Terminal I/O 인터페이스를 새로 만든 이유를 다음과 같이 설명하고 있습니다:

- `ioctl()` 함수는 두번째 인자에 따라 세번째 인자의 타입, 크기가 달라질 수 있으므로, 정확히 설명(specify)하기 어렵습니다.
- `ioctl()`의 정확한 semantics는 운영체제마다 각각 다릅니다.
- 여태 나온 implementation 중, 국제 환경(international environment)에 맞는 것이 없습니다.

즉, `ioctl()`은 권장되지 않는 함수입니다. 자세한 것은 [Lewine] 를 참고하기 바랍니다.

References [PCS] § 10 pp. 128–9, § 10.1 pp. 130–1
[POSIX] § 7.
[Lewine] Chap. 8 pp. 145

Q 19.2 문자를 읽어들이기 전에 얼마나 많은 문자가 대기하고 있는지 알아낼 방법이 있나요? 또 대기하고 있는 문자가 없을 경우, 바로 함수가 종료하게 만들 수 없을까요?

Answer 이 질문도, 마찬가지로 완전히 운영체제에 의존적인 부분입니다. 어떤 버전의 curses는 `nodelay()` 함수를 쓸 수 있게 해 줍니다. 시스템에 따라 다르지만 "nonblocking I/O"를 쓸 수 있는 시스템이 있습니다. 또는 "`select`"나 "`poll`"이라는 시스템 콜로 이 작업을 대신할 수 있습니다. 또 `FIONREAD` `ioctl`, `c_cc[VTIME]`, `kbhit()`, `rdchk()`, 그리고 `open()`이나 `fcntl()`에서 `O_NDELAY` 옵션이 이러한 작업을 해 줄 수 있습니다. 덧붙여 질문 19.1도 참고하시기 바랍니다.

Q 19.3 현재 어떤 일이 몇 퍼센트가 진행되었는지 계속 표시할 수 있는 방법이나 "twirling baton[3]"을 표시할 수 있는 방법을 알려주세요.

Answer 가장 간단하고, 이식성이 뛰어난 방법은 한 줄을 출력한 다음, 캐리지 리턴(carriage return) 문자인 '`\r`'을 출력하는 것입니다. 이 문자를 라인 피드(line feed) 문자 없이 혼자 쓰면 현재 줄을 다시 덮어 쓸 수 있습니다. 또는 백스페이스(backspace) 문자인 '`\b`'을 쓰면 커서를 한 칸 왼쪽으로 옮겨 한 글자를 덮어 쓸 수 있게 됩니다.

References [C89] § 5.2.2

Q 19.4 화면을 지우는 방법은? 색깔을 입혀 텍스트를 출력하는 방법은? 커서를 지정한 x, y 위치로 옮길 수 있는 방법은?

Answer 이러한 방법은 여러분이 쓰고 있는 터미널(또는 디스플레이)의 종류에 따라 다릅니다. 대개는 termcap, terminfo, curses와 같은 라이브러리를 써서 이런 작업을 처리합니다. MS-DOS 시스템에서는 `clrscr()`, `gotoxy()`를 찾아보기 바랍니다.

화면을 지우는 방법 중 가장 이식성이 높은 방법은 폼 피드(form-feed) 문자 ('`\f`')를 출력하는 것입니다. 이 문자를 출력하면 대다수 화면이 지워지게 됩니다. (조금 지저분하지만) 더 이식성이 높은 방법은 충분히 많은 newline 문자를 출력해서 한 화면 분의 여러 줄들을 넘겨 버리는 것입니다. 마지막 수단으로, `system()`을 써서 (질문 19.27 참고) 화면을 지우는 명령을 실행할 수도 있습니다.

References [PCS] § 5.1.4 pp. 54–60, § 5.1.5 pp. 60–62

Q 19.5 화살표 키를 읽으려면 어떻게 해야 하나요?

Answer Terminfo, 또 몇몇 termcap, 그리고 몇몇 curses는 ASCII가 아닌 키들을 입력받을 수 있습니다. 일반적으로 이러한 키들은 여러 개의 문자 입력으로 (대개 첫번째 문자는 ESC, '`\033`') 처리되며, 이 입력을 분석해 내는 것은 상당히 까다롭습니다. (`keypad()`를 부를 경우 curses가 이 작업을 대신 처리해 줍니다.)

MS-DOS에서는 만약 문자 값이 0인 것(문자 '0'이 아님!)이 들어온다는 것은, 다음 문자가 특별한 키 입력 값이라는 것을 나타내는 신호입니다. 이 것을 스캔 코드(scan code)라고 하는데, 여기에 관한 것은 여러 DOS programming

[3] 작업 진행 표시기 — `-`, `/`, `|`, `\`를 같은 위치에 번갈아 가며 출력.

guide를 찾아보시면 됩니다. (몇 개만 말해보면; up, left, right, down 화살표 키는 각각 72, 75, 77, 80이며, function key들은 59에서 68 사이의 값을 가집니다.)

References [PCS] § 5.1.4 pp. 56–7

Q 19.6 마우스 입력은 어떻게 처리하나요?

Answer 시스템 문서들을 읽어보거나 적당한 뉴스 그룹에 물어보시기 바랍니다. (뉴스 그룹에 물어볼 때에는 먼저 FAQ 목록을 본 다음에) 마우스 처리는 X 윈도우 시스템, MS-DOS, 매킨토시 등등의 시스템에 따라 매우 다릅니다.

References [PCS] § 5.5 pp. 78–80

Q 19.7 시리얼("comm") 포트 입/출력은 어떻게 처리하죠?

Answer 마찬가지로 시스템에 의존적인 문제입니다. UNIX에서는 `/dev`에 있는 적당한 장치 파일을 `open`, `read`, `write` 함수를, 터미널 드라이버가 제공하는 여러 기능들과 함께 쓰면 됩니다. MS-DOS에서는 미리 정의된 스트림인 stdaux를 쓰거나 "`COM1`"과 같은 이름으로 파일을 열거나 저수준 BIOS 인터럽트들을 쓰면 됩니다. 많은 네터[4]들이 Joe Campbell씨의 *C Programmer's Guide to Serial Communications*을 추천하고 있습니다.

Q 19.8 출력을 프린터로 보내는 방법을 알려주세요.

Answer Unix에서는 popen() (질문 19.30 참고) 함수를 써서 'lp'나 'lpr' 프로그램을 실행하거나, '`/dev/lp`'와 같은 특수 파일을 열어서 작업합니다. MS-DOS에서는 (비표준) 미리 정의된 스트림인 `stdprn`을 쓰거나 "`PRN`"이나 "`LPT1`"으로 파일을 열어서 작업하면 됩니다.

References [PCS] § 5.3 pp. 72–74

Q 19.9 Escape sequence를 터미널이나 비슷한 장치에 보내는 방법은?

Answer 문자를 장치에 보내는 방법을 알고 있다면 (질문 19.8 참고), Escape sequence를 보내는 방법은 매우 쉽습니다. ASCII로 ESC는 033, (10 진수로 27), 이므로:

[4] netter, 뉴스 그룹을 사용하는 사람. '네티즌(netizen)' 정도로 해석하면 됩니다.

```
fprintf(ofd, "\033[J");
```

을 쓰면 escape sequence, `ESC [ J`를 보낼 수 있습니다.

Q 19.10 그래픽을 처리하는 방법은?

Answer 옛날, Unix는, plot(3)와 plot(5)에서 설명하는, 장치 독립적인 plot 함수를 제공했습니다. GNU libplot 패키지는 같은 목적으로 좀 더 나은 기능을 제공합니다. 다음 URL을 참고하기 바랍니다:

```
http://www.gnu.org/software/plotutils/plotutils.html
```

MS-DOS에서 작업한다면 VESA나 BGI 표준에 따르는 라이브러리를 쓰기를 원할 것입니다.

플로터에 그리는 작업은 대개 특정 escape sequence로 이루어집니다; 플로터 제조자는 대개 C 언어로 된 라이브러리 패키지를 제공하므로, 이 것을 쓰던지 net을 뒤져보기 바랍니다.

윈도우 시스템(매킨토시나 X Window System, 또는 Microsoft Windows)을 쓰고 있다면, 윈도우 기능을 쓰고 싶어할지도 모릅니다; 이 경우 관련된 뉴스 그룹이나 FAQ 목록을 먼저 참고하기 바랍니다.

References [PCS] § 5.4 pp. 75–77

Q 19.11 저는 사용자에게 입력 파일이 없다는 경고를 출력하고 싶습니다. 파일이 존재하는 지 어떻게 하면 검사할 수 있을까요?

Answer 이런 간단한 문제도 표준에 맞게, 또는 호환성이 높게 처리할 방법이 없다는 것은 참으로 안타까울 뿐입니다. 검사하는 어떤 방법을 썼다고 하더라도 테스트 한 후, 파일을 열기 전에 (다른 프로세스에 의해) 그 파일이 새로 만들어지거나 지워질 수 있기 때문입니다.

이런 목적으로 쓸 수 있는 함수는 `stat()`, `access()`, `fopen()`이 있습니다. (`fopen()`을 쓴다면 파일을 읽기 모드로 열고 바로 닫으면 됩니다. `fopen()`이 실패한다고 해서 무조건 파일이 존재하지 않는다는 것은 아닙니다.) 물론 이 함수들 중에서는 `fopen()`이 가장 이식성이 뛰어납니다. UNIX의 set-UID 기능이 있다면 `access()` 함수는 주의깊게 써야 합니다.

단순히 파일이 성공적으로 열렸다고 가정하는 것보다 항상 리턴 값을 검사해서 실패했는지 조사하는 것이 바람직합니다.

References [PCS] § 12 pp. 189, 213
[POSIX] § 5.3.1, § 5.6.2, § 5.6.3

Q 19.12 파일을 읽기 전에 그 파일의 크기를 알 수 있을까요?

Answer C 언어에서 "파일의 크기"란 파일에서 읽을 수 있는 문자의 갯수를 말합니다. 정확히 그 크기를 아는 것은 매우 어렵거나 불가능합니다.

UNIX에서는 `stat()` 함수가 정확한 값을 알려 줄 수 있습니다. 대부분의 다른 시스템에서는 UNIX와 비슷한 `stat()` 함수를 제공하고, 비슷한 값(정확하지 않을 수도 있음)을 알려줍니다. `fseek()`를 써서 파일 위치를 맨 뒤로 옮긴 다음, `ftell()`을 써서 값을 얻어내거나 `fstat()`을 쓸 수도 있지만 이 두가지 방법은 같은 단점을 가집니다. 일단 `fstat()`은 이식성이 뛰어나지 않으며, `stat()`과 같은 정보를 알려줍니다. `ftell()`은 바이너리 파일이 아닐 경우 (즉 텍스트 파일), 정확한 바이트 갯수를 알려준다는 보장이 없습니다. 어떤 시스템은 `filesize()`나 `filelength()`와 같은 함수를 제공하지만, 역시 이식성이 뛰어나지 않습니다.

파일을 읽기 전에 그 파일의 크기를 아는 것이 꼭 필요한 지 먼저 생각해 보기 바랍니다. 왜냐하면 C 언어에서 파일 크기를 알아내는 가장 정확한 방법은 파일을 열어서 읽어보는 것이기 때문입니다. 파일을 읽어가며 파일 크기를 계산하는 것도 한가지 해결책이 될 수 있습니다.

References [C89] § 7.9.9.4
[H&S] § 15.5.1
[PCS] § 12 p. 213
[POSIX] § 5.6.2

Q 19.12b 파일 변경 날짜와 시간을 알아내려면 어떻게 하나요?

Answer Unix와 POSIX 함수인 `stat()`을 쓰면 됩니다. 대부분의 다른 시스템에서도 이 함수를 지원합니다. (덧붙여 질문 19.12도 참고하시기 바랍니다.)

Note TODO: stat()이 표준인지 아닌지 조사!

Q 19.13 파일을 완전히 지우거나 새로 만들지 않고 파일 크기를 줄이는 방법이 없을까요?

Answer BSD 시스템은 `ftruncate()` 함수를 제공하고, 몇몇 다른 시스템에서는 `chsize()` 함수를, 또 어떤 시스템에서는 `fcntl()` 함수에 (대개 문서화되지 않은) `F_FREESP` 옵션을 써서 파일 크기를 줄일 수 있습니다. MS-DOS에서는 가끔 `write(fd, "", 0)`을 써서 해결할 수 있지만 이들은 모두 이식성이 뛰어나지 않습니다. 마찬가지로 파일의 앞 부분을 잘라내는 것도 이식성이 뛰어난 방법은 존재하지 않습니다. 덧붙여 질문 19.14도 참고하시기 바랍니다.

Q 19.14 파일의 중간 쯤에 한 줄(또는 레코드)을 추가하거나 지울 수 있을까요?

Answer 파일을 다시 만들지 않는 한, 거의 불가능하다고 봐야 합니다. (한 레코드를 지우는 경우에는 간단히 지웠다고 표시(marking)하는 것이 간단한 해결책이 될 수 있습니다.) 또, 단순한 일반 파일이 아닌 데이터베이스를 쓰는 것도 한 가지 해결책이 될 수 있습니다. 덧붙여 질문 12.30, 19.13도 참고하시기 바랍니다.

Q 19.15 주어진 스트림(FILE *)이나, 파일 디스크립터(descriptor)를 써서 파일 이름을 다시 얻어낼 수 있는 방법이 있을까요?

Answer 이 문제는 일반적으로 해결할 수 없습니다. 예를 들어, Unix에서는 (어떤 특별한 권한으로) 디스크 전체를 다시 읽는 방법이 이론상 필요하고, 주어진 디스크립터가 파이프(pipe)나 지워진 파일을 가리키고 있다면 알 수 없으며, 또 파일이 여러 개의 링크를 가지고 있을 경우에는 틀린 이름을 알려 줄 수도 있습니다. 따라서 파일을 열 때, 파일 이름을 따로 저장하는 등의 방법(예를 들어, fopen()의 wrapper 함수를 만들어)을 쓰면 좋습니다.

Q 19.16 파일은 어떻게 지우나요?

Answer 표준 C 라이브러리 함수인 `remove()`를 쓰면 됩니다. (이 질문은 아마 이 단원에서 "시스템 의존적인 방법을 쓰지 않고" 답할 수 있는 몇되지 않은 질문입니다.) 오래된, ANSI Unix 이전의 시스템에서는 `remove()`를 제공하지 않을 수도 있습니다. 이 경우, `unlink()`를 쓰기 바랍니다.

Note `unlink()`는 표준 함수가 아닙니다.

References [K&R2] § B1.1 p. 242
[C89] § 7.9.4.1
[H&S] § 15.15 p. 382

[PCS] § 12 pp. 208, 220–221
[POSIX] § 5.5.1, § 8.2.4

Q 19.16b 파일을 복사하는 방법은?

Answer `system()`을 써서 운영 체제에서 제공하는 파일 복사 명령을 쓰거나 (질문 19.27 참고), 원본과 사본 파일을 (`fopen()`이나 저수준 파일 open 시스템 콜을 써서) 열어 문자 단위 또는 블럭 단위로 데이터를 복사하는 방법을 쓰면 됩니다.

References K&R § 1, § 7

Q 19.17 왜 절대 경로를 써서 파일을 열 수 없나요? 아래와 같이 호출하면 항상 실패합니다:

```
fopen("c:\newdir\file.dat", "r")
```

Answer 실제 요청한 파일 이름은 — 문자 `\n`과 `\f`가 쓰였으므로 — 아마 존재하지 않을 것입니다. 따라서 생각한 것처럼 파일이 열리지 않습니다.

문자 상수나 문자열에서 백슬래시, \는 이스케이프 문자로 해석되어, 뒤따르는 문자에게 특별한 의미를 주는 데에 쓰인다는 것을 기억하기 바랍니다. 백슬래시가 파일 이름[5]에 쓰이기 위해서는 백슬래시를 두번 써서 다음과 같이 만들어야 합니다:

```
fopen("c:\\newdir\\file.dat", "r")
```

MS-DOS에서는 다른 방법을 쓸 수 있습니다. 백슬래시 대신에 슬래시를 써도 디렉토리를 구분하는 문자로 인식되기 때문에 다음과 같이 하면 됩니다:

```
fopen("c:/newdir/file.dat", "r")
```

(그러나, 전처리기 directive인 `#include`에서 쓰는 파일 이름은 문자열이 아니라는 것을 명심해야 합니다. 따라서 거기에는 백슬래시를 그냥 한번만 써도 됩니다.)

[5] 원문은 pathname이지만 편의상 파일 이름으로 번역했음

Note 헤더 파일을 포함할 때, 절대 경로를 쓰는 것은 좋지 않은 습관입니다. 왜냐하면, 나중에 헤더 파일의 저장 위치가 바뀔 경우, 소스 파일 전체를 훑어가며 고칠 필요가 생길 지 모르기 때문입니다.

가능한 상대 경로를 쓰기 바라며, 상대 경로가 복잡해질 경우라면, 대부분의 컴파일러는 헤더 파일이 위치한 경로를 지정해 줄 수 있는 옵션을 제공하므로, 그 기능을 쓰시기 바랍니다 (대개 컴파일러는 `-I` 옵션을 이 목적으로 씁니다.)

Q 19.18 "Too many open files"라는 에러를 봤습니다. 어떻게 하면 동시에 열 수 있는 파일의 갯수를 늘릴 수 있을까요?

두 가지 이유에서 동시에 열 수 있는 파일 갯수에 제한이 있습니다: 하나는 운영 체제에서 쓸 수 있는 저수준 "파일 descriptor"나 "파일 핸들"의 갯수에 제한이 있거나, 표준 입출력(stdio) 라이브러리에서 쓸 수 있는 `FILE` 구조체의 갯수에 제한이 있기 때문입니다. 이 두 수치가 모두 충분해야 파일을 열 수 있습니다. MS-DOS 시스템에서는 `CONFIG.SYS` 파일을 수정해서 운영체제가 다룰 수 있는 파일의 갯수를 고칠 수 있습니다. 어떤 컴파일러는 어떤 명령을 (또는 몇몇의 소스 파일을) 써서 stdio `FILE` 구조체의 갯수를 늘릴 수 있습니다.

Q 19.20 C 프로그램에서 디렉토리를 읽을 수 있습니까?

Answer POSIX 표준 함수이고, 대부분 Unix 시스템에서 제공하는 `opendir()`과 `readdir()`을 쓸 수 있는 지 조사해 보기 바랍니다. 또 MS-DOS나 VMS, 기타 시스템에서도 이들 함수를 제공하는 경우가 많습니다. (MS-DOS에는 거의 비슷한 일을 해주는 `FINDFIRST`와 `FINDNEXT` 루틴을 제공하기도 합니다.) `readdir()`은 단순히 파일 이름만 알려주기 때문에, 파일에 대한 자세한 정보가 필요하다면 `stat()` 함수를 써야 합니다. 파일이름과 어떤 와일드카드 패턴을 비교하려면 질문 13.7을 참고하기 바랍니다.

Note 디렉토리를 처리하는 방법은, 엄밀히 말해, 표준 C 라이브러리에서는 제공하지 않습니다. 12 절의 머릿말을 참고하시기 바랍니다.

References [K&R2] § 8.6 pp. 179–184
[PCS] § 13 pp. 230–1
[POSIX] § 5.1
[Dale] § 8

Q 19.22 얼마만큼 메모리가 비어있는지 알 수 있을까요?

Answer 운영체제가 이런 정보를 알려 주는 루틴을 제공할 수도 있지만, 이 방법도 전적으로 시스템 의존적입니다.

Q 19.23 64K보다 큰 배열이나 구조체를 만들 수 있을까요?

Answer 제대로 된 컴파일러라면 사용 가능한 메모리만큼 아무렇게나 써도 동작을 해야 합니다만, 그렇지 못한 컴파일러라면 프로그램에서 쓰는 메모리 양을 줄이던지 아니면 시스템 의존적인 다른 방법을 써야 합니다.

64K는 (현재에도) 상당히 큰 블럭입니다. 여러분의 컴퓨터에 얼마나 많은 메모리가 비어있느냐와는 상관없이, 연속된 64K의 블럭을 할당하는 것은 쉽지 않습니다. (C 표준은 한 오브젝트가 32K보다 클 경우, 아무것도 보장하지 않습니다. [C9X]에서는 64K가 그 제한입니다.) 이 경우, 연속적인 공간이 아니어도 상관없는 그러한 방식으로 쓰는 것이 좋습니다. 예를 들어 동적으로 할당하는 다차원 배열[6]의 경우, 포인터를 가리키는 포인터[7]를 쓸 수 있고, 또 linked list나 포인터의 배열[8]을 쓸 수도 있습니다.

만약 PC 호환 (8086 기반의) 시스템이라면, 그리고 64K 또는 640K 메모리 제한에 걸린다면 "huge" 메모리 모델을 쓰거나 확장 메모리(expanded memory)나 연장 메모리(extended memory)를 쓰는 것도 생각해보기 바랍니다. 또 `malloc` 계열의 변종인 `halloc()`이나 `farmalloc()`을 쓰는 방법도 있고, 32-bit "flat" 컴파일러(예를 들면 djgpp, 질문 18.3 참고)를 쓰는 방법도 있습니다. 또는 DOS extender를 쓰거나 다른 운영체제를 쓰는 것도 생각할 수 있습니다.

References [C89] § 5.2.4.1
[C9X] § 5.2.4.1

Q 19.24 "DGROUP data allocation exceeds 64K"라는 에러 메시지가 나왔는데 무슨 말인가요? 'large' 메모리 모델을 썼으니 64K 이상의 데이터를 쓸 수 있다고 생각하는데요.

Answer 'large' 메모리 모델을 쓰더라도, MS-DOS 컴파일러는 어떤 데이터(문자열이나 전역, 또는 정적 변수들)들을 여전히 default 데이터 세그먼트에 두기 때문에, 이 세그먼트는 여전히 overflow가 일어날 수 있습니다. 전역 변수를 줄

[6] dynamically-allocated multidimensional arrays
[7] pointers to pointers
[8] array of pointers

이거나, 만약 이미 줄일만큼 줄였다면 (문제가 많은 문자열 때문이라면), 컴파일러에 어떤 옵션을 주어서 default 데이터 세그먼트를 쓰지 않도록 하면 됩니다. 어떤 컴파일러들은 "작은" 데이터들만 default 데이터 세그먼트에 위치시키지만, 이것도 여러분이 선택할 수 있게 해 줍니다 (예를 들어 Microsoft 컴파일러의 경우 /Gt 옵션을 써서 "작은" 데이터가 얼마만큼 작은 것을 뜻하는지 지정할 수 있습니다).

Q 19.25 어떤 주소에 위치한 메모리에 (memory-mapped 장치나 비디오 메모리) 접근하려면 어떻게 하죠?

Answer 적당한 타입의 포인터를 만들어 그 주소를 대입합니다. 이때 캐스트 연산을 써서 컴파일러에게 이식성이 없는 방식을 쓴다는 것을 알려야 합니다:

```
unsigned int *magicloc = (unsigned int *)0x12345678;
```

그러면 '*magicloc'이 여러분이 원하는 위치를 나타내게 됩니다. (MS-DOS에서는 `MK_FP()`와 같은, 세그먼트와 offset을 나타내는 편리한 방법을 찾을 수 있을 것입니다.)

Note 물론 위의 방법이 100% 동작한다고 보장할 수는 없습니다. 현존하는 대개의 multi-tasking OS는 각각의 프로그램이 다른 프로그램의 메모리에 접근하는 것을 제한하고 있으므로, 응용 프로그램에서 이 방법을 써서 접근하는 것은 거의 불가능합니다. 단, 응용 프로그램이 아닌 OS나 device driver 프로그램은 예외입니다.

References [K&R1] § A14.4 p. 210
[K&R2] § A6.6 p. 199
[C89] § 6.3.4
[ANSI Rationale] § 3.3.4
[H&S] § 6.2.7 pp. 171–2

Q 19.27 C 프로그램 안에서 다른 프로그램(운영 체제의 명령이나 독자적인 프로그램들)을 부르려면(실행하려면) 어떻게 하죠?

Answer 라이브러리 함수인 `system()`를 쓰면 됩니다. `system()`의 리턴 값은 불러서 실행한 프로그램의 끝냄 상태(exit status)라는 것을 기억하시기 바랍니다 (그러나 이는 표준에 정의되어 있지 않기 때문에 꼭 그렇지 않을 수도 있습니다).

이는 대개 그 프로그램의 출력과는 별 상관이 없습니다. 또한 `system()`이 받는 인자는 하나의 문자열이라는 것을 기억하기 바랍니다. 따라서 여러 개의 (복잡한) 문자열을 한 문자열로 바꿀 필요가 있다면 `sprintf()`를 쓰면 됩니다. 덧붙여 질문 19.30도 참고하시기 바랍니다.

References [K&R1] § 7.9 p. 157
[K&R2] § 7.8.4 p. 167, § B6 p. 253
[C89] § 7.10.4.5
[H&S] § 19.2 p. 407
[PCS] § 11 p. 179

Note [H&S]에 나온 말을 그대로 인용해보면 다음과 같습니다:

> The function **system** passes its string argument to the operating system's *command processor* for execution in some implementation-defined way. In UNIX systems, the command processor is the shell. The value returned by **system** is implementation-defined but is usually the completion status of the command.

해석하면 다음과 같습니다:

`system()`이 전달받는 인자는 운영 체제의 명령 처리기에 전달되어 시스템에 의존적인 방식으로 실행됩니다. UNIX 시스템에서는 명령 처리기는 shell입니다. `system()`의 리턴 값은 시스템에 의존적이나, 대개는 실행한 명령의 끝냄 코드가 됩니다.

Q 19.30 다른 프로그램이나 명령을 실행시켜서 그 출력을 받아올 수 없을까요?

Answer Unix와 어떤 시스템들은 `popen()` 함수를 제공합니다. 이 함수는 stdio 스트림을 지정한 명령을 처리하는 프로세스와 파이프로 연결해 줍니다. 따라서 그 명령의 출력은 (또는 입력도 가능) 여러분의 프로그램에서 읽을 수 있습니다. (덧붙여, `pclose()` 함수를 불러줘야 합니다.)

`popen()` 함수를 쓸 수 없다면 `system()` 함수를 써서 출력이 파일로 저장되게 한 다음, 그 파일을 읽는 방법을 쓸 수도 있습니다.

Unix를 쓰고 있고, `popen()` 함수로 충분치 않다면, `pipe()`, `dup()`, `fork()`, `exec()` 함수를 써서 할 수 있습니다.

(한 가지 기억해 두어야 하는 것은 `freopen()` 함수는 원하는 대로 동작하지 않을 수도 있다는 것입니다.)

References [PCS] § 11 p. 169

Q 19.31 프로그램 안에서, 프로그램의 절대 경로를 얻어낼 수 있을까요?

Answer `argv[0]`이 절대 경로를 나타내거나, 일부분을 나타낼 수도 있지만, 아무것도 나타내지 않을 수도 있습니다. 이 경우 명령 처리기[9]의 검색 경로 알고리즘을 흉내내서 실행 파일의 절대 경로를 얻어낼 수 있습니다. 그러나 이것이 완벽한 해결책이라고 말할 수는 없습니다.

References [K&R1] § 5.11 p. 111
[K&R2] § 5.10 p. 115
[C89] § 5.1.2.2.1
[H&S] § 20.1 p. 416

Q 19.32 실행 파일이 있는 위치에, 프로그램의 설정 파일을 저장하고 싶은데, 어떻게 하면 그 경로를 찾을 수 있을까요?

Answer 어렵습니다; 질문 19.31의 내용을 먼저 참고하기 바랍니다. 여러분이 그 방법을 알았다고 하더라도, 그 설정 파일의 위치를 바꿀 수 있는 방법(예를 들어 환경 변수를 쓰는 것)을 제공하는 편이 좋습니다. (특히 여러분의 프로그램이 멀티 유져 시스템에서 각각 다른 사람에 의해, 각각 다른 설정 파일을 써서 동작할 필요가 있다면 이 기능은 더욱 중요합니다.)

Q 19.33 프로세스가 그 프로세스를 호출한 parent 프로세스의 environment variable(환경 변수)를 바꿀 수 있나요?

Answer 가능할 수도, 그렇지 않을 수도 있습니다. 대부분의 운영 체제가 환경 변수를 다루는 방식이 Unix와 비슷하기는 하지만 완전히 같지는 않습니다. 환경 변수가 진행 중인 프로그램에 의해 바뀔 수 있는 것이 편리할 수도 있지만, 어쨌든 시스템에 의존적인 문제입니다.

Unix에서 process는 자신의 environment만을 (대부분 `setenv()`, `putenv()` 함수를 써서) 고칠 수 있고, 변경된 환경 변수들은 그대로 자식(child) 프로세스에게 전달되지만, 부모(parent) 프로세스에는 전달되지 않습니다. MS-DOS에서는 마스터 환경 변수[10]를 고칠 수 있지만, 이 방법은 매우 복잡합니다. 자세한 것은 MS-DOS FAQ 목록을 참고하기 바랍니다.)

Q 19.36 오브젝트 파일을 읽어서 그 내용을 실행할려면 어떻게 하죠?

[9] command langauge interpreter
[10] master copy of the environment

Answer 동적 링커(dynamic linker)나 동적 로더(dynamic loader)가 필요합니다. 메모리를 동적으로 할당하고(`malloc`), 오브젝트 파일을 읽어 오는 것은 가능하지만, 오브젝트 파일 형식의 복잡함을 안다면 여러분은 매우 놀랄지도 모릅니다. BSD Unix에서는 `system()` 함수와 `ld -A` 명령을 써서 링크를 할 수 있습니다. 대부분의 SunOS와 System V의 버전들은 `-ldl` 라이브러리를 써서 오브젝트 파일이 동적으로 로딩되게 할 수 있습니다. VMS에서는 `LIB$FIND_IMAGE_SYMBOL`을 써서 이 기능을 만들 수 있고, GNU에서는 이 목적으로 "dld" 패키지를 제공합니다. 덧붙여 질문 15.13도 참고하시기 바랍니다.

Q 19.37 주어진 시간만큼 기다리거나, 사용자의 응답 시간을 잴 때, 초 단위보다 더 세밀한 단위를 쓸려면 어떻게 하죠?

Answer 안타깝게도 그런 일을 처리하는 이식성이 뛰어난 방법은 없습니다. V7 Unix와 그 후계 시스템들은 `ftime()`이라는 매우 쓸모있는 함수를 제공하며, 이 함수는 millisecond 단위로 동작합니다. 또 다른 함수로는 `clock()`, `delay()`, `gettimeofday()`, `msleep()`, `nap()`, `napms()`, `nanosleep()`, `sleep()`, `setitimer()`, `times()`, `usleep()`이 있습니다. (Unix에서 제공되는 `wait()` 함수는 이런 기능이 아닙니다.) `select()`와 `poll()`을 쓸 수 있다면 간단한 기다리기 함수를 만들 수 있습니다. MS-DOS 시스템에서는 시스템 타이머를 프로그래밍하고 타이머 인터럽트를 쓰는 방법을 쓰면 됩니다.

물론, `clock()` 함수 만이 ANSI 표준입니다. `clock()` 함수를 두 번 호출하고 그 리턴 값을 비교해서 걸린 시간을 비교할 수 있으며, 그 시간의 단위는 `CLOCKS_PER_SEG`가 1보다 클 경우, 초 단위보다 더 세밀할 수 있습니다. 그러나 `clock()`은 현재 프로세스에서만 시간차를 계산할 수 있기 때문에, 멀티태스킹 시스템에서는 진짜 시간과는 약간 다를 수 있습니다.

만약 여러분이 `delay` 함수를 만들기를 원하고, 시간을 알려주는 함수를 쓸 수 있다면, CPU-intensive busy-waiting을 써서 만들 수 있을 것입니다. 그러나 이는 싱글 태스크 머신에서 혼자 쓸 경우에만 쓸 수 있습니다. 멀티태스킹 운영 체제에서는 여러분의 프로세스가 어떤 기간동안 잠들어 있을 수 있다(예를 들어 `sleep()`, `select()`, `pause()`를 `alarm()`과 `setitimer()`와 함께 써서)는 것을 명심해야 합니다.

시간을 지연시키는 간단한 방법은 다음과 같이 아무것도 하지 않는 루프를 사용하는 것입니다:

```
long int i;
```

```
for(i = 0; i < 1000000; i++)
  ;
```

그러나, 이러한 코드를 쓰는 것은 될 수 있으면 피하기 바랍니다. 일단 몇 번 반복할 것인가를 잘 선택해야 하며, 더 빠른 프로세서에서는 이 값이 더 늘어날 것입니다. 엎친데 덮친 격으로, 좋은 컴파일러는 위와 같이 아무 것도 하지 않는 코드는 최적화 과정에서 빼 버릴 수 있습니다.

References [H&S] § 18.1 pp. 398–9
[PCS] § 12 pp. 197–8, pp. 215–6
[POSIX] § 4.5.2

Q 19.38 Control-C와 같은 키보드 인터럽트를 무시하거나 잡아낼 수 있습니까?

Answer 간단한 방법은 다음과 같이 `signal()` 함수를 쓰는 것입니다:

```
#include <signal.h>
signal(SIGINT, SIG_IGN);
```

위는 인터럽트 시그널을 무시하는 것이고, 아래처럼 인터럽트 시그널이 발생했을 때 특정 함수인 `func()`가 불려지게 만들 수도 있습니다:

```
extern void func(int);
signal(SIGINT, func);
```

Unix와 같은 멀티 태스킹 시스템에서는 다음과 같은 방법을 쓰는 것이 더 좋습니다:

```
extern void func(int);
if (signal(SIGINT, SIG_IGN) != SIG_IGN)
  signal(SIGINT, func);
```

위에서 조건문과 추가적으로 부른 `signal()`은 foreground에서 실행된 프로그램에서 발생한 키보드 인터럽트가 background에서 동작하는 프로그램에게 잘못 전달되는 것을 막아 줍니다. (이런 형태의 코드는 `signal()`을 부르는 방식에 어긋나지 않습니다.)

어떤 시스템에서는 키보드 인터럽트가 터미널 입력 시스템에 따라 영향을 받기도 합니다; 질문 19.1을 참고하기 바랍니다. 어떤 시스템에서는 키보드 인터럽트를 검사하는 것은 프로그램이 입력을 읽고 있을 때에만 적용되며, 그럴 경우, 인터럽트 처리는 어떤 입력 루틴이 쓰이느냐에 따라 달라집니다. MS-DOS 시스템에서는 `setcbrk()`이나 `ctrlbrk()` 함수를 쓰기도 합니다.

References [C89] § 7.7, 7.7.1
[H&S] § 19.6 pp. 411–3
[PCS] § 12 pp. 210–2
[POSIX] § 3.3.1, 3.3.4

Q 19.39 floating-point(실수) exception(예외)를 멋지게 처리하는 방법이 있을까요?

Answer 대부분의 시스템에서는 `matherr()` 함수를 정의해서 어떤 실수 에러가 (예를 들어 `<math.h>`안의 함수에 의해) 났을 때, 불려지게 할 수 있습니다. 또는 `signal()` 함수를 써서 (질문 19.38 참고) SIGFPE를 잡아낼 수도 있습니다. 덧붙여 질문 14.9도 참고하시기 바랍니다.

References [ANSI Rationale] § 4.5.1

Q 19.40 소켓(socket)을 쓰는 방법은? 또 네트워크 client/server를 만드는 방법은?

Answer 이러한 질문은 이 FAQ 목록의 범위를 벗어납니다. C 언어로 이런 네트워킹 기능을 처리하는 것은 여러 책에서 설명하고 있습니다. 추천할 만한 책으로는 Douglas Comer씨의 3권짜리 책인 *Internetworking with TCP/IP*와 W. R. Stevens씨의 *UNIX Networking Programming*이 있습니다. Net에도 "UNIX Socket FAQ"를 포함한 여러 정보가 널려 있습니다. 이 socket FAQ는 다음 URL에서 구할 수 있습니다:

`http://kipper.york.ac.uk/~vic/sock-faq/`

Q 19.40b BIOS를 호출하는 방법은요? TSR을 만드는 방법을 알고 싶습니다.

Answer 이는 어떤 (MS-DOS를 쓰는 PC) 시스템에 매우 의존적인 문제입니다. `comp.os.msdos.programmer`나 이 그룹의 FAQ 목록을 보면 관련된 정보가 있습니다. 또 Ralf Brown씨의 인터럽트 목록[11]도 참고하기 바랍니다.

Q 19.40c 어떤 프로그램을 컴파일하려 하는데, 컴파일러가 "`union REGS`"가 정의되어 있지 않다고 에러를 출력하고, 링커는 `int86()`이 정의되어 있지 않다고 에러를 냅니다.

[11] Ralf Brown's interrupt list

Answer 모두 MS-DOS 인터럽트 프로그래밍을 하는데에 필요한 것입니다. 다른 시스템에서는 존재하지 않습니다.

Q 19.41 그러나 저는 비표준 함수나 시스템에 의존적인 함수를 쓰면 안되는 처지입니다. 제 프로그램은 ANSI 호환이 되어야 하니까요. 무슨 방법이 없을까요?

Answer 매우 좋지 않은 상황이거나 아니면 처음부터 잘못 이해하고 있는 경우입니다. 왜냐하면 ANSI/[C89] C 표준은 — 운영체제가 아닌 언어에 대한 정의라서 — 이러한 기능을 전혀 정의하고 있지 않기 때문입니다. 이러한 분야에 대한 국제 표준은 POSIX[12]에서 다루고 있으며, (Unix 뿐만 아니라) 많은 운영 체제에서 POSIX 호환의 인터페이스를 제공하고 있습니다.

물론 프로그램의 많은 부분을 ANSI 호환으로 작성하는 것은 매우 바람직하며 또 가능합니다. 즉 시스템에 의존적인 부분들만 몇몇의 파일에 따로 모아두면, 나중에 그 부분만 새로 작성하면 프로그램이 돌아갈 수 있도록 만들면 좋습니다.

[12]IEEE 1003.1, [C89]/IEC 9945-1

Chapter 20

Miscellaneous

이 절에서는 제목이 뜻하듯, 다른 절에 해당하지 않는 여러가지 주제를 다룹니다. 처음 두 단락에서는 여러가지 프로그래밍 테크닉과 bit나 byte 단위로 제어하는 방법을 소개합니다. 그 다음으로 효율성(efficiency)과 C 언어의 `switch`에 대해 다룹니다. "기타 언어 기능" 단락은 상당히 역사적인(historical) 내용이 많습니다; 이 단락은 C 언어의 기능이 왜 그렇게 제공되는지, 또 많은 사람들이 필요한 어떠한 기능이 왜 C 언어에서 제공되지 않는지를 설명합니다. 여기에 관한 사항은 C 언어와 다른 언어와의 차이점을 소개하기도 합니다.

알고리즘에 관한 것은 책 한 권으로 쓰기에도 모자랍니다. 또한 이 글은 알고리즘 설명을 목적으로 한 것이 아니기에, "알고리즘" 단란에서는 모든 C 프로그래머가 다루어야 하는 것만 소개합니다.

이 절의 질문과 단락은 다음과 같이 나눌 수 있습니다.

Miscellaneous Techniques 20.1 – 20.6

Bits and Bytes 20.7 – 20.12

Efficiency 20.12 – 20.16

`switch` **Statements** 20.16 – 20.18

Miscellaneous Language Features 20.19 – 20.24

Other Languages 20.25 – 20.27

Algorithms 20.28 – 20.33

Trivia 20.34 – 20.40

20.1 Miscellaneous Techniques

Q 20.1 한 함수에서 여러 개의 값을 리턴할 수 있을까요?

Answer 세 가지 방법을 쓸 수 있습니다: 하나는 함수 내부에서 고칠 수 있도록 포인터를 인자로 받는 것이고, 하나는 여러 개의 멤버를 가지는 구조체를 리턴하도록 하는 것입니다. 마지막으로 전역 변수를 이용하는 방법이 있습니다. 덧붙여 질문 2.7, 4.8, 7.5a도 참고하시기 바랍니다.

Q 20.3 'command-line argument'에 접근하는 방법은?

Answer `main()`에 전달되는, `argv` 배열에 대한 포인터를 쓰면 가능합니다. 덧붙여 질문 8.2, 13.7, 19.20도 참고하시기 바랍니다.

References [K&R1] § 5.11 pp. 110–114
[K&R2] § 5.10 pp. 114–118
[C89] § 5.1.2.2.1
[H&S] § 20.1 p. 416
[PCS] § 5.6 pp. 81-2, § 11 p. 159, pp. 339–40 Appendix F
[Dale] § 4 pp. 75–85

Q 20.5 각각 다른 워드 크기나 (word size), 다른 바이트 순서를 (byte order) 쓰는, 또는 각각 다른 실수 포맷을 (floating point format) 쓰는 컴퓨터에서 데이터 파일을 주고 받으려 합니다. 이런 데이터 파일을 만들 수 있을까요?

Answer 가장 이식성이 뛰어난 방식은 (대개 ASCII 코드로 된) 텍스트 파일을 사용하는 것입니다: `fprintf()`나 `fscanf()`와 같은 함수를 쓰면 됩니다. (네트워크 프로토콜 상에서도 비슷한 문제가 발생할 수 있습니다.) 그러나, 텍스트 파일은 바이너리 파일에 비해 처리 속도가 떨어지고, 같은 데이터를 포함할 때에도 크기가 훨씬 클 수 있다는 사실을 알아두어야 합니다. 이러한 단점에도 텍스트 파일로 저장할 경우, 각각 다른 환경의 컴퓨터에서도 이 데이터를 쓸 수 있다는 장점이 있고, 또 이러한 텍스트 파일을 처리하는 많은 표준 툴들이 있다는 것을 생각할 때, 텍스트 파일의 중요성은 강조해도 지나치지 않습니다.

꼭 바이너리 파일을 써야만 한다면, 이식성을 높이기 위해, 또는 기존의 라이브러리를 쓰는 잇점을 누리기 위해, Sun의 XDR (RFC 1014)이나 OSI의 ASN.1 (CCITT X.409와 [C89] 8825 "Basic Encoding Rules"에서 참조됨),

CDF, netCDF, HDF를 쓰는 것이 좋을 것입니다. 덧붙여 질문 2.12도 참고하시기 바랍니다.

References [PCS] § 6 pp. 86, 88.

Q 20.6 문자열에 어떤 함수의 이름이 저장되어 있을 때, 이 문자열 만으로 함수를 호출할 수 있을까요?

Answer 가장 직접적인 방법은 각각의 함수 이름과 함수 포인터를 테이블에 저장하는 것입니다:

```
int func(), anotherfunc();

struct { char *name; int (*funcptr)(); } symtab[] = {
  "func",         func,
  "anotherfunc",  anotherfunc,
};
```

그 다음, 각각의 이름에 해당하는 테이블 목록을 뒤져서 해당하는 함수 포인터를 호출하면 됩니다. 덧붙여 질문 2.15, 18.14, 19.36도 참고하시기 바랍니다.

References [PCS] § 11 p. 168

20.2 Bits and Bytes

Q 20.8 비트(bit)로 이루어진 집합 또는 배열을 만들려면 어떻게 하죠?

Answer `char` 또는 `int` 타입의 배열을 만들고 각각의 비트를 제어하기 위한 인덱스를 사용하는 매크로를 쓰면 됩니다. `char` 타입을 써서 만든 예는 다음과 같습니다:

```
#include <limits.h>   /* for CHAR_BIT */

#define BITMASK(b)    (1 << ((b) % CHAR_BIT))
#define BITSLOT(b)    ((b) / CHAR_BIT)
#define BITSET(a, b)  ((a)[BITSLOT(b)] |= BITMASK(b))
#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
```

(만약 <limits.h>가 없다면 CHAR_BIT 대신 8을 쓰기 바랍니다.)

References [H&S] § 7.6.7 pp. 211–216

Q 20.9 컴퓨터의 바이트 순서가 (byte order) big-endian인지 little-endian인지 어떻게 알아내죠?

Answer 한가지 방법은 포인터를 쓰는 것입니다:

```
int x = 1;
if (*(char *)&x == 1)
  printf("little-endian\n");
else
  printf("big-endian\n");
```

union을 써서도 가능합니다.

덧붙여 질문 10.16도 참고하시기 바랍니다.

References [H&S] § 6.1.2 pp. 163–4

Q 20.10 정수를 2 진수나 16 진수로 바꿀수 있을까요

Answer 먼저 무엇을 질문하려 하는 지를 잘 생각하기 바랍니다. 10진, 16진, 8진, 2진 등 어떤 식의 표현이 편리할 지에 상관없이 정수는 내부적으로 2 진수로 저장됩니다. 이러한 진법이 관심 대상이 될 경우는, 실제 사용자와 인터페이스하는 부분일 (outside world) 때입니다.

소스 코드에서 10진수가 아닌 표현으로는, 8진수를 표기하기 위해 수치 앞에 0을 붙이거나, 16진수를 표현하기 위해 0x를 붙이는 것이 있습니다. 실제 I/O를 처리할 때에는 이러한 수치해석은 printf와 scanf 계통의 함수들의 포맷에서 (format specifier) 지정합니다 (%d, %o, %x 등). 또는 strtol(), strtoul() 함수의 세번째 인자로도 지정할 수 있습니다. 어떤 진수로 수치를 문자열로 변환하고자 한다면 여러분 스스로 이러한 일을 하는 함수를 만들어야 합니다 (기능상 strtol 함수와 반대되는 일을 해야 합니다.). 바이너리 (binary) I/O를 할 때에는 이러한 작업이 전혀 불필요합니다.

바이너리 I/O에 관한 것은 질문 2.11을 참고하기 바랍니다. 덧붙여 질문 8.6, 13.1도 참고하시기 바랍니다.

References [C89] § 7.10.1.5, 7.10.1.6.

Q 20.11 상수를 지정할 때 2진수를 (예를 들면 0b101010처럼) 쓸 수 있나요? 또 2진수를 출력하기 위한 `printf()` 포맷이 있나요?

Answer 두 경우 모두 없습니다. 2진수를 출력하려면 `strtol()` 함수를 써서 직접 문자열을 만들어야 합니다. 덧붙여 질문 20.10도 참고하시기 바랍니다.

20.3 Efficiency

Q 20.12 주어진 수치에서 비트 1이 몇 개인지 세는(count) 효과적인 방법은 무엇인가요?

Answer 이러한 "비트에 관련된 귀찮은" (bit-fiddling) 문제들은 lookup 테이블을 써서 해결될 수 있습니다 (그러나 질문 20.13을 꼭 참고하기 바랍니다).

Q 20.13 어떻게 하면 효과적인 프로그램을 만들 수 있을까요?

Answer 좋은 알고리즘을 사용하고, 주의깊게 구현하고, 프로그램이 불필요한 작업을 하지 않도록 하면 됩니다.

예를 들어 전 세계에서 가장 최적화된 문자 복사 코드는 문자를 전혀 복사하지 않은 코드보다 느립니다. 원문을 그대로 옮기면 다음과 같습니다:

> For example, the most microoptimized character-copying loop in the world will be beat by code which avoids having to copy characters at all.

효율성을 생각할 때는, 멀리서 바라보아야 합니다. 무엇보다도, "얼마나 효과적인가"가 인기있는 이야깃거리가 될 수 있지만, 사람들이 항상 그렇게 생각하는 것은 아닙니다. 대부분의 코드는 시간에 민감할(time-critical) 필요가 없습니다. 대신 얼마나 코드가 읽기 쉬운가, 또 코드의 이식성이 높은지가 관심사일 때가 더 많습니다. (컴퓨터는 매우 빠른 기계라는 것을 잊으면 안됩니다. "비효과적인" 코드라도 효과적인 컴파일을 거치면 문제가 없는 경우가 많습니다.)

프로그램에서 어떤 부분이 가장 중요한 부분(hot spot)인지 미리 짐작하는 것은 매우 어렵습니다. 효율성이 중요한 문제가 된다면 프로파일링(profiling) 소프트웨어를 쓰면 좋습니다. 때때로 실제 시간은 주변 장치의 일, 예를 들어 I/O나 메모리 할당과 같은 곳에서 많이 걸릴 때가 많으며, 이런 문제들은 버퍼링이나 캐시로 해결합니다.

시간에 민감한 코드를 만들어야 하더라도 코드의 미세한 부분까지 일일히 신경을 쓰는 것은 좋지 않습니다. "효과적인 코딩 방법"으로 알려져 있는 것 중에 2의 지수꼴로 나타나는 수치를 곱하는 대신 쉬프트(shift) 연산을 쓰는 것이 있습니다. 그러나, 아주 간단하거나 기초적인 컴파일러라도 이를 자동으로 처리해 주는 경우가 대부분입니다. 최적화한다고 코드에 여러가지를 덕지덕지 끼워넣는다면 오히려 전체적인 속도가 떨어지는 경우가 대부분이며, 또한 이식성이 떨어지게 됩니다 (다시 말하면, 어떤 시스템에서는 빠르게 동작하더라도 다른 시스템에서는 오히려 느려지는 경우를 말합니다). 어떠한 경우라도 코드를 이리저리 바꿔보는 것은 잘해봐야 linear 성능 향상을 가져올 뿐입니다. 차라리 더 좋은 알고리즘을 선택하도록 하는 것이 낫습니다.

효율성 tradeoff[1] 및 효율성이 중요한 경우, 이를 향상시키는 방법에 대한 조언을 얻고 싶다면, Kernighan과 Plauger의 *The Elements of Programming Style*의 Chapter 7, 그리고 Jon Bentley의 *Writing Efficient Programs*를 참고하기 바랍니다.

Q 20.14 정말로 포인터는 배열보다 빠르게 동작하나요? 함수 호출은 어느 정도 시간을 잡아먹죠? `++i`가 `i = i + 1`보다 빠른게 확실한가요?

Answer 이런 질문들은 프로세서와 컴파일러에 따라 답이 달라집니다. 간단히 알고 싶다면, 테스트 프로그램을 만들어 시간을 재면 됩니다. 그러나 그 차이가 매우 적기 때문에 같은 프로그램을 수천번 돌려야 할지도 모릅니다. 만약 컴파일러가 어셈블리 언어 출력을 지원한다면, 두 가지 코드가 같은 방식으로 컴파일되는지 먼저 확인하기 바랍니다.

일반적으로 큰 배열을 훑어 나갈때에는 '`[]`' 연산보다 포인터 연산이 빠른 것으로 알려져 있지만, 어떤 프로세서에서는 반대입니다.

함수 호출은 인라인(in-line) 코드보다 느린 것이 확실하지만 코드의 모듈화(modularity)와 명쾌함(clarity)을 생각한다면 함수 호출을 쓰는 편이 낫습니다.

'i = i + 1'과 같은 코드를 좀 더 나은 방식으로 바꾸기 전에, 여러분은 간단한 계산기를 쓰는 것이 아니라 컴파일러와 씨름한다는 것을 기억하기 바랍니다. 현대 컴파일러는 대부분 `++i`, `i += 1`, `i = i + 1`과 같은 것은 모두 똑같은 코드를 만들어 냅니다. 즉 현대에 `++i`, `i += 1`, `i = i + 1` 중 어떤 것을 쓰느냐는 문제는 스타일에 관한 문제이지, 더 이상 효율성에 관한 문제가 아닙니다. (덧붙여 질문 3.12도 참고하시기 바랍니다.)

[1] 흥정이라고 번역할 수도 있지만 원문의 느낌을 그대로 전달하기 위해 번역하지 않습니다.

Q 20.15b 스피드 향상을 위해 어셈블러를 사용할 필요가 없다는 이유로 최적화 컴파일러가 (optimizing compiler) 좋다고 합니다. 그러나 제 최적화 컴파일러는 `i /= 2`조차도 shift로 대체시키지 못하는군요.

Answer `i`가 `signed`입니까, 아니면 `unsigned`입니까? 만약 `signed` 타입이었다면, 'shift' 연산과 같지 않습니다. (힌트: `i`가 음수이고 홀수일 경우를 생각해보기 바랍니다.) 그래서 컴파일러가 'shift' 연산을 쓰지 않았을 것입니다.

Q 20.15c 임시 저장 변수 없이 두 변수의 내용을 바꿀 수 있을까요?

Answer 오래된 어셈블리 프로그래머들의 트릭을 쓰면 다음과 같이 만들 수 있습니다:

```
a ^= b;
b ^= a;
a ^= b;
```

그러나 이와 같은 코드는 현대의 HLL[2]에는 맞지 않습니다. 임시 변수는 꼭 필요한 것이며, 다음과 같이 쓰는 것은,

```
int t = a;
a = b;
b = t;
```

코드를 읽는 사람이 이해하기 쉽도록 만들어 줄 뿐만 아니라, 컴파일러가 이를 이해해서 가장 효과적인 코드를 만들 수 있도록 해 줍니다 (가능하다면 swap 명령을 써서). 게다가 이렇게 사용하면 두 변수가 포인터, 실수와 같은 타입일 때도 사용할 수 있습니다. 당연히 XOR를 사용한 방법은 이와 같은 데이터 타입에는 쓸 수 없습니다. 덧붙여 질문 3.3b, 10.3도 참고하시기 바랍니다.

20.4 `switch` Statements

Q 20.17 문자열에 대해 `switch`를 쓸 수 있을까요?

Answer 직접적으로는 쓸 수 없습니다. 그러나, 문자열을 정수로 매핑시키는 함수를 써서 이 함수를 `switch`에 쓰면 가능합니다. 그렇지 않다면 `strcmp()`와 `if/else`를 반복해서 비교해야 합니다. 덧붙여 질문 10.12, 20.18, 20.29도 참고하시기 바랍니다.

[2]High Level Language

References [K&R1] § 3.4 p. 55
[K&R2] § 3.4 p. 58
[C89] § 6.6.4.2
[H&S] § 8.7 p. 248

Q 20.18 `case` label에 상수가 아닌 표현 (예를 들어 어떤 범위나 일반 수식)을 사용하는 방법이 있을까요?

Answer 없습니다. `switch` statement는 원래 컴파일러가 매우 간단하게 번역할 수 있도록 디자인되었기 때문에, `case` label은 단일한(single), 상수(constant)이면서, 정수 수식(integral expression)만 사용할 수 있습니다. 물론 하나의 statement에 여러 `case` label을 붙이는 것으로 간단한 범위를 흉내낼 수 있습니다. (아래 예제 참고).

일반 수식이나 상수가 아닌 수식을 사용하려면, `if/else`를 사용해야 합니다.

덧붙여 질문 20.17도 참고하시기 바랍니다.

Note
```
switch (c) {
case 1:
case 2:
case 3:
  /* case 1-3: some statements */
case 4:
case 5:
  /* case 4-5: some statements */
default:
  /* ...  */
}
```

References [K&R1] § 3.4 p. 55
[K&R2] § 3.4 p. 58
[C89] § 6.6.4.2
[ANSI Rationale] § 3.6.4.2
[H&S] § 8.7 p. 248

20.5 Miscellaneous Language Features

Q 20.19 `return` 문장에서 바깥쪽(outter) 괄호는 정말로 생략 가능한가요?

Answer 생략 가능합니다 (optional).

예전에 C 초창기에는 이 괄호가 필요했습니다. 그래서 많은 사람들이 C 언어를 쓸 때, 여전히 이 괄호를 사용하곤 합니다.

(마찬가지의 이유로 `sizeof` 연산자에서도 괄호를 쓰곤 하지만 사실은 이 괄호도 생략 가능합니다.)

References [K&R1] § A18.3 p. 218
[C89] § 6.3.3, § 6.6.6
[H&S] § 8.9 p. 254

Q 20.20 왜 C 언어의 주석(comment)는 중첩해서 쓸 수 없을까요? 주석이 들어있는 코드의 일부분을 주석 처리하고 싶거든요. 또, 문자열 안에 주석을 쓸 수 있나요?

Answer C 언어의 주석은 중첩해서 쓸 수 없습니다. C 언어의 주석은 PL/I의 주석 구문을 빌어왔으며, PL/I에서는 중첩된 주석문을 허용하지 않습니다. 따라서 — 내부적으로 주석을 포함한 — 어떠한 코드 블럭을 주석 처리하고 싶다면, `#ifdef`나 `#if`를 쓰는 것이 좋습니다 (그러나 질문 11.19를 꼭 읽어보기 바랍니다.)

문자열 "`/*`"와 "`*/`"는 특별한 의미가 없습니다. 따라서 문자열 안에서는 주석을 쓸 수 없습니다. 왜냐하면 — 특히 C 소스 코드를 출력해주는 — 프로그램이 그러한 문자열을 출력하기를 원할 수도 있기 때문입니다.

또한 C++에서 제공하는 `//` 주석은 C 언어에서 쓸 수 없습니다. 따라서 C 프로그램에서는 이 주석을 쓰지 않는 습관을 길러야 합니다 (여러분의 컴파일러가 `//` 주석을 쓸 수 있는 확장 기능을 제공한다 하더라도 말입니다).

References [K&R1] § A2.1 p. 179
[K&R2] § A2.2 p. 192
[C89] § 6.1.9, Annex F
[ANSI Rationale] § 3.1.9
[H&S] § 2.2 pp. 18–9
[PCS] § 10 p. 130

Q 20.20b Is C a great language, or what? Where else could you write something like a+++++b ?

Answer Well, you can't meaningfully write it in C, either. The rule for lexical analysis is that at each point during a straightforward left-to-right scan, the longest possible token is determined, without regard to whether the resulting sequence of tokens makes sense. The fragment in the question is therefore interpreted as

```
a ++ ++ + b
```

and cannot be parsed as a valid expression.

References [K&R1] § A2 p. 179
[K&R2] § A2.1 p. 192
[C89] § 6.1
[H&S] § 2.3 pp. 19–20

Q 20.24 C 언어에서는 왜 중첩된 함수를 제공하지 않을까요?

Answer 중첩된 함수에서 지역 변수와 함수를 제어하는 것은 쉽지 않기 때문입니다. 따라서 C 언어에서는 간결화(simplification)를 추구하기 위해 이 기능을 제공하지 않습니다. (gcc는 확장 기능으로 중첩 함수를 제공합니다.) 대개 이러한 기능은 qsort에서 비교 함수를 만들기 위해 필요하다고 생각할 수 있습니다. 이런 함수들은 `static` 선언과 정적 변수를 써서 만들면 됩니다. (깨끗한 해결책으로는 필요한 정보를 가지고 있는 구조체에 대한 포인터를 전달하는 것이지만, `qsort()`에서는 제공하지 않습니다.)

Q 20.24b `assert()`은 어떤 함수인가요?

Answer `assert()` 함수는 `<assert.h>`에 정의되어 있는 매크로 함수이며 "assertion"을 테스트하기 위한 함수입니다. 이 때 'assertion'이란 프로그래머가 결정하는 어떤 가정(assumption)으로, 이 'assertion'이 어긋난다는 것은 심각한 프로그래밍 오류가 됩니다. 예를 들어 널이 아닌 포인터를 입력받는 함수는 다음과 같은 코드를 추가할 수 있습니다:

```
assert(p != NULL);
```

'assertion'이 실패하면 프로그램은 강제 종료됩니다. 따라서 `malloc()`이나 `fopen`과 같이 예상할 수 있는 에러를 검사하기 위해 쓰일 수는 없습니다.

References [K&R2] § B6 pp. 253–4
[C89] § 7.2
[H&S] § 19.1 p. 406

20.6 Other Languages

Q 20.25 C 언어에서 FORTRAN (C++, BASIC, Pascal Ada, LISP) 함수를 부를 수 있을까요? (반대로 다른 언어가 C 함수를 가져다 쓸 수 있을까요?)

Answer 답은 전적으로 사용하고 있는 컴퓨터 기종과 컴파일러가 사용하는 'calling sequence'에 따라 달려 있습니다. (전혀 불가능할 수도 있습니다.) 일단 컴파일러 문서를 주의깊게 읽어보시기 바랍니다. 대개는 "mixed-language programming guide"라는 이름의 문서가 함께 제공될 것이니, 이를 참고하기 바랍니다. 올바른 방법으로 인자를 전달하고 적절한 startup code를 사용해야 하지만, 그리 쉬운 것이 아님을 미리 밝혀 둡니다. 좀 더 자세한 것은 Gleen Geers씨의 FORT.gz 파일을 참고하기 바라며, ftp로 suphys.physics.su.oz.au의 `src` 디렉토리에서 얻을 수 있습니다.

대부분 인기있는 컴퓨터에서 C/FORTRAN interface를 간편하게 해주는 `cfortran.h`라는 헤더 파일이 있으니 참고하기 바랍니다. zebra.desy.de에서 anonymous ftp로, 또는 `http://www-zeus.desy.de/ burow/`에서 얻을 수 있습니다.

C++에서는 `"C"` modifier가 external function 선언이 C calling convention을 사용하는 것을 지정해 줍니다.

References [H&S] § 4.9.8 pp. 106–7

Q 20.26 Pascal이나 FORTRAN (또는 LISP, Ada, awk, 구 스타일 C 등)을 C 언어로 변환해주는 프로그램이 있을까요?

Answer 자유롭게 배포되는 프로그램들은 다음과 같습니다:

p2c	Dave Gillespie씨가 만든, Pascal을 C로 변경해주는 프로그램입니다. 이 프로그램은 1990년 3월에 `comp.sources.unix`에 (Volumn 21) 게시되었습니다. 그리고 `csvax.cs.caltech.edu`에 익명의 (anonymous) FTP를 써서 `pub/p2c-1.20.tar.Z`로 받을 수 있습니다.
ptoc	마찬가지로 Pascal을 C 언어로 변경해주는 프로그램입니다. 이 프로그램은 Pascal로 작성되었으며, `comp.sources.unix` Volumn 10에 게시되었고, Volumn 13?에서 패치되었습니다.
f2c	FORTRAN을 C로 변경해 주는 프로그램으로 Bell Labs, Bellcore, Carnegie Mellon의 사람들에 의해서 개발되었습니다. f2c에 대한 더 자세한 사항을 알고 싶으면, `netlib@research.att.com`이나 `research!netlib.netlib.att.com`으로 "send index from f2c"라는 메일을 보내시면 얻을 수 있습니다. (물론 익명 FTP로 `netlib.att.com`에서 `netlib/f2c`로 받을 수 있습니다.)

이 FAQ 리스트의 관리자는 다른 (상업용) 변환 프로그램과 다른 잘 알려지지 않은 언어에 대한 프로그램의 목록을 유지하고 있습니다.

덧붙여 질문 11.31, 18.16도 참고하시기 바랍니다.

Q 20.27 C++은 C의 'superset'입니까? C 코드를 컴파일하기 위해 C++ 컴파일러를 써도 상관없을까요?

Answer C++는 C 언어에서 유래한 언어입니다. 그리고 실제로 많은 부분을 C 언어에 기초를 두고 있습니다만, 어떤 부분에 대해서는 C 언어 관점으로 올바른 것이 C++에서는 올바르지 않은 경우가 있습니다. 거꾸로, ANSI C에는 몇 가지 기능들을 (프로토타입, `const`, 등) C++에서 가져왔습니다. 따라서 한 언어가 다른 언어의 완전한 'superset'은 아닙니다; 또 어떤 의미를 정의할 때, 서로 다르게 정의한 부분도 있습니다. 이런 차이점에도 불구하고, 많은 C 프로그램들이 C++에서도 올바르게 동작합니다. 그리고 대부분의 최근 컴파일러들은 C와 C++ 컴파일 모드를 다 지원합니다. 덧붙여 질문 8.9, 20.20도 참고하시기 바랍니다.

References [H&S] p. xviii, § 1.1.5 p. 6, § 2.8 pp. 36–7, § 4.9 pp. 104–107

20.7 Algorithms

Q 20.28 `strcmp`와 비슷한 루틴이 필요한데, 두 문자열이 정확하게 일치하지 않는 경우도 일치하는 것으로 처리하는 기능이 필요합니다.

Answer 대충 비슷한 문자열을 비교하기(approximate string matching) 위한 정보와 알고리즘은 Sun Wu와 Udi Manber's Paper의 "AGREP — A Fast Approximate Pattern-Matching Tool"에서 찾아볼 수 있습니다.

또, 여러 비슷한 단어를 같은 코드로 mapping시켜주는 "soundex" 알고리즘을 생각해 볼 수 있습니다. Soundex는 원래 (전화번호부 도우미 목적으로) 비슷하게 발음되는 이름을 발견하기 위한 목적으로 디자인된 것입니다. but it can be pressed into service for processing arbitrary words.

References [Knuth] § 6 pp. 391–2 Vol. 3
[Wu]

Q 20.29 해싱(hashing)이란 무언가요?

Answer 해싱(hashing)은 문자열을 (대개 범위가 작은) 정수로 대응시키는 (mapping) 작업을 의미합니다. "해시 함수"는 (hash function) 문자열을 (또는 다른 data structure) 어떤 범위를 가지는 수치로 (hash bucket) 매핑시키는 함수입니다. 이 수치는 대개 배열의 인덱스 값으로 사용하거나, 반복되는 비교를 처리하기 위해 사용합니다. (물론, 상당히 큰 문자열의 집합을 작은 범위의 수치로 매핑한다면, 일대일의 관계가 성립하지 않을 수도 있습니다. 따라서 해싱을 처리하는 알고리즘은 이런 "collision" 발생을 적절하게 처리해 주어야 합니다.) 이미 해시에 관한 알고리즘과 함수가 많이 개발되었습니다; 이런 주제에 대한 더 자세한 것은 이 리스트의 목적과 맞지 않기 때문에 생략합니다.

References [K&R2] § 6.6
[Knuth] § 6.4 pp. 506–549 Vol. 3
[Robert] § 16 pp. 231–244

Q 20.31 주어진 날짜로 요일을 계산하려면 어떻게 하죠?

Answer `mktime()`과 `localtime()`을 쓰면 됩니다. (덧붙여 질문 13.13, 13.14도 참고하시기 바랍니다.. 그러나 `tm_hour`가 0이 될 때, DST adjustment를 주의해야 합니다.) 또는 Zeller's congruence를 (`sci.math`의 FAQ 리스트 참고) 쓰거나, Tomohiko Sakamoto씨의 다음 코드를 쓰면 됩니다:

```
int dayofweek(int y, int m, int d)    /* 0 = Sunday */
{
  static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
  y -= m < 3;
  return (y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
}
```

덧붙여 질문 13.14, 20.32도 참고하시기 바랍니다.

References [C89] § 7.12.2.3.

Q 20.32 2000년이 윤년인가요? `(year % 4 == 0)`으로 윤년을 계산하는게 올바른가요?

Answer 2000년은 윤년이 (leap year) 맞습니다. 그리고 윤년을 계산하는 공식은 다음과 같습니다:

```
year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
```

좋은 천문학 달력이나 (astronomical almanac), 다른 참고 도서를 참고하기 바랍니다. (To forestall an eternal debate: references which claim the existence of a 4000-year rule are wrong.) 덧붙여 질문 13.14, 13.14b도 참고하시기 바랍니다.

20.8 Trivia

Q 20.34 자신의 소스 코드를 출력하는 프로그램을 만들려면 어떻게 하죠?

Answer 이식성이 뛰어난, 자신을 재 출력하는 프로그램을 (self-reproducing program) 만드는 것은 매우 어려운 일입니다. 특히 따옴표와 문자 코드 때문에 어렵습니다.

전형적인 예는 (한 출로 되어 있지만, 일단 수행되면 자신을 고치게 됩니다) 다음과 같습니다:

```
char *s = "char *s=%c%s%c; main() {printf(s, 34, s, 34);}";
main() { printf(s, 34, s, 34); }
```

(이 프로그램은 많은 다른 장르의 프로그램처럼, `#include <stdio.h>`를 포함하고 있지 않고, 큰따옴표 문자인 "가 (ASCII에서) 34라고 가정하고 작성된 것입니다.)

Q 20.35 What is "Duff's Device"?

Answer It's a devastatingly deviously unrolled byte-copying loop, devised by Tom Duff while he was at Lucasfilm. In its "classic" form, it looks like:

```
register n = (count + 7) / 8; /* count > 0 assumed */
switch (count % 8) {
case 0: do { *to = *from++;
case 7:      *to = *from++;
case 6:      *to = *from++;
case 5:      *to = *from++;
case 4:      *to = *from++;
case 3:      *to = *from++;
case 2:      *to = *from++;
case 1:      *to = *from++;
  } while (--n > 0);
}
```

where count bytes are to be copied from the array pointed to by from to the memory location pointed to by to (which is a memory- mapped device output register, which is why to isn't incremented). It solves the problem of handling the leftover bytes (when count isn't a multiple of 8) by interleaving a switch statement with the loop which copies bytes 8 at a time. (Believe it or not, it **is** legal to have case labels buried within blocks nested in a switch statement like this. In his announcement of the technique to C's developers and the world, Duff noted that C's switch syntax, in particular its "fall through" behavior, had long been controversial, and that "This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.")

Q 20.36 다음 "International Obfuscated C Code Contest(IOCCC)"는 언제 열리나요? 저번 수상자의 작품을 볼 수 있을까요?

Answer 이 컨테스트는 항상 진행 중입니다. 다음 사이트를 참고하기 바랍니다:

```
http://www.ioccc.org/index.html
```

수상자는 대개 Usenix 포럼에 공표되며 그 후로 net에도 게시됩니다. 과거의 (1984년 이후) 수상 작품들은 `ftp.uu.net`의 `pub/ioccc` 디렉토리에 저장되어 있습니다(질문 18.16 참고)

Q 20.37 [K&R1]에 나오는 `entry` 키워드는 어디에 쓰는 거죠?

Answer 함수가 - FORTRAN에서처럼 - 하나 이상의 entry point를 가질 수 있도록 하기 위해 예약되어 있는 것입니다.

실제로 이 기능이 들어있는 컴파일러는 만들어진 적이 없다고 알려져 (이 문법이 어떤 식으로 쓰이는 지 기억하는 사람도 없으며) 있습니다. 이 키워드는 ANSI C에서는 제거되었습니다. (덧붙여 질문 1.12도 참고하시기 바랍니다.)

References [K&R2] p. 259 Appendix C.

Q 20.38 "C" 언어의 "C"는 어디에서 유래한 말인가요?

Answer C 언어는 Ken Thompson씨의 실험 언어인 B 언어에서 유래한 것입니다. 그리고 이 B 언어는 Martin Richards씨의 BCPL에서 (Basic Combined Programming Language) 유래한 것입니다. 이 BCPL은 CPL을 (Cambridge Programming Language) 간략화한 (simplification) 언어입니다. C 언어를 상속한 P 언어도 있습니다; 'D'가 아닌 'P'를 선택한 것은 BCPL의 세번째 글자가 'P'이기 때문입니다. 그러나, 현재 C 언어를 상속한 가장 잘 알려진 언어는 C++입니다.

Q 20.39 "`char`"를 어떻게 발음하죠?

Answer C 키워드(keyword)인 "`char`"는 세 가지의 발음을 쓸 수 있습니다: 영어 단어인 "char", "care", "car" (또는 "character"로); 어떤 것을 쓰느냐는 별로 중요하지 않습니다.

Q 20.39b "lvalue"와 "rvalue"는 무엇인가요?

Answer 간단히 말해서, "lvalue"는 대입(assignment) 연산에서 왼쪽에 올 수 있는 수식을 말합니다; 따라서 어떤 위치를 나타내는 오브젝트로 생각할 수 있습니다. (그러나 질문 6.7을 참고하기 바랍니다.) "rvalue"라 함은 어떠한 값을 가지는 모든 수식을 말합니다 (즉 대입 연산에서 오른쪽에 올 수 있는 수식을 말합니다).

Q 20.40 이 문서의 사본은 어디에서 얻을 수 있죠? 이전 버전도 얻을 수 있나요?

Answer 가장 최신의 사본은 `ftp.eskimo.com`의 `u/s/scs/C-FAQ/`에서 얻을 수 있습니다. 또는 뉴스 그룹에서 얻을 수도 있습니다; 최신 문서는 newsgroup `comp.lang.c`에 한달 주기로 게시됩니다. 동시에 이 문서를 (변경 사항 포함) 요약한 버전도 게시됩니다.

이 문서의 다양한 버전이 `comp.answers`와 `news.answers`에도 게시됩니다. `news.answer` 그룹에는 다른 FAQ 목록과 함께 이 목록이 제공되며, 이러한 목록들을 구할 수 있는 곳 중 두 곳을 들면 다음과 같습니다:

```
rtfm.mit.edu  pub/usenet/news.answers/C-faq/
ftp.uu.net    usenet/news.answers/C-faq
```

ftp를 쓸 수 없다면 `rtfm.mit.edu`의 메일 서버가 여러분에게 메일로 FAQ 목록을 보내줄 수 있으니 이 기능을 쓰면 됩니다: 메일 내용으로 "help"라는 한 단어를 `mail-server@rtfm.mit.edu`로 보내면 됩니다. 자세한 것은 news.answers의 meta-FAQ 목록을 참고하기 바랍니다.

하이퍼텍스트 (HTML) 버전은 WWW에서 제공됩니다: URL은 다음과 같습니다:

```
http://www.eskimo.com/~scs/C-faq/top.html
```

모든 유즈넷 FAQ 리스트는 아래 사이트에서 얻을 수 있습니다:

```
http://www.faqs.org/faqs/
```

이 FAQ 리스트의 확장판은 Addison Wesley사에서 "C Programming FAQs: Frequently Asked Questions"라는 이름의 책으로 출판되었습니다 (ISBN 0-201-84519-9). 정정 목록은 (errata list) 다음에서 구할 수 있습니다:

```
http://www.eskimo.com/~scs/C-faq/book/Errta.html
```

또는 `ftp.eskimo.com`의 `u/s/scs/ftp/C-faq/book/Errta`로 구할 수도 있습니다.

각각의 문서들은 매달 흥미로운 질문들에 대한 답변이 아니라, 전체 질문/답변 목록을 유지하고 있습니다. 따라서 오래된 버전은 필요치 않을 것입니다.

Note 이 문서의 한국어판은 아래 URL에서 얻을 수 있습니다:

`http://www.cinsk.org/cfaqs/`

위 사이트에서 제공하는 형식은 postscript (`.ps`), portable document format (`.pdf`), 하이퍼 텍스트 (`.html`) 입니다.

Chapter 21

Extensions

이 chapter에서는 역자가 추가한 질문들을 정리해 놓았습니다.

21.1 miscellaneous code

이 단락에서는 여러분들이 거의 모든 프로그램 개발에 쓸 수 있는 utility 함수들에 대한 설명입니다.

Q 21.1 라이브러리 함수들이 성공적으로 수행되었는지 일일이 리턴값을 조사해서 에러 메시지를 출력하는 것이 번거롭습니다. 간단히 할 수 있는 방법이 없을까요?

Answer 다음과 같이 error 함수[1]를 만들기 바랍니다.

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* 아래 변수는 error()가 몇 번 불렸는지 횟수를
 * 저장합니다. */
unsigned int error_message_count;

/* 아래 함수 포인터가 NULL이면 error() 함수는
```

[1] 이 error 함수는 GNU software 소스 코드에서 얻은 것을 간략화한 것입니다.

```
 * stdout을 flush시키고 stderr로 프로그램 이름을
 * 출력하고, ": "를 출력합니다. */
void (*error_print_progname);

/* error() 함수를 쓰는 프로그램은 반드시
 * program_name이라는 변수를 정의하고 프로그램의
 * 이름(실행 파일 이름)을 지정해야 합니다. */
extern const char *program_name;

void
error(int status, int errnum, const char *msg, ...)
{
  va_list argptr;

  if (error_print_progname)
    (*error_print_progname)();
  else {
    fflush(stdout);
    fprintf(stderr, "%s: ", program_name);
  }

  va_start(argptr, msg);
  vfprintf(stderr, msg, argptr);
  va_end(argptr);

  ++error_message_count;

  if (errnum)
    fprintf(stderr, ": %s", strerror(errnum));
  putc('\n', stderr);
  fflush(stderr);
  if (status)
    exit(status);
}
```

일단 위와 같이 error() 함수를 만들었으면 여러분의 코드에서 다음과 같이 쓸 수 있습니다.

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

const char *program_name = "sample"

int
main(void)
{
  FILE *fp;
  char *filename = "sample.dat";

  fp = fopen(filename, "r");
  if (fp == NULL)
    error(EXIT_FAILURE, errno,
          "cannot open file %s.", filename);
  /* ... */
}
```

위의 코드를 설명하겠습니다. error()의 첫번째 인자는 0이 아닌 경우, 라이브러리 함수 exit()로 전달됩니다. 따라서 <stdlib.h>에 정의된 0이 아닌 값을 가지는 EXIT_FAILURE 상수를 써서 error()가 메시지를 출력하고 프로그램을 종료하도록 합니다

error()에 전달된 두번째 인자인, errno는 <errno.h>에 정의된 전역 변수로서 몇몇 라이브러리 함수들이 보고한 에러의 종류를 나타내는 변수입니다. 이 값은 error()가 strerror 함수를 써서 에러를 나타내는 문자열로 바꾼 다음 출력합니다. 시스템과 라이브러리 함수가 보고한 에러가 아니라면 error()의 두번째 인자로 0을 주면 됩니다.

error()의 세번째 인자부터는 printf()가 쓰는 형식 그대로 쓰시면 됩니다.

물론 위의 예제가 완벽하지는 않습니다. 여러분은 error() 함수를 쓰기 위해, 앞에서 제시한 error() 정의와, 이 함수가 쓰는 전역 변수들을 적당하게 추가해야 합니다.

에러 함수의 선언과 va_list, va_start(), va_end()에 관한 것은 15 절을 참고하기 바랍니다.

References [GLIBC] misc/error.*

Q 21.2 User input을 string으로 받은 다음, `atoi` 함수를 써서 수치로 바꾸어 작업했습니다. 그런데 가끔 이상한 문제가 발생합니다. 왜 그럴까요?

Answer `atoi`는 주어진 문자열을 10진수로 해석해서 `int`로 바꾸어 주지만, 에러는 처리해 주지 않습니다. 다시 말해서 입력 문자열을 10진수로 처리할 수 없을 때에는 대개 0을 리턴합니다.

따라서 다음과 같은 함수를 만들어 쓰는 것이 좋습니다:

```
#include <errno.h>

int
convint(long int *d, const char *s)
{
  char *p;
  long int i;

  if (*s == '\0')        /* error: nothing in the source string */
    return -1;
  i = strtol(s, &p, 0);
  if (errno == ERANGE)   /* error: underflow or overflow */
    return -1;

  if (*p != '\0')        /* error: there is unexpected character */
    return -1;

  *d = i;
  return 0;
}
```

위 함수는 두 개의 인자를 받습니다. 첫 인자는 변환한 `long int`를 저장할 포인터를 받고, 두 번째 인자는 변환할 문자열을 받습니다. 리턴 값은 성공적으로 변환한 경우에는 0을, 그렇지 않은 경우에는 -1을 리턴합니다 – 입력에 예상할 수 없는 문자값이 오거나, 입력이 아예 없거나(문자열이 `'\0'`인 경우), 변환한 값이 `long int`에 담을 수 없을 만큼 크거나 작은 경우, 에러로 처리합니다.

실제 사용한 예는 다음과 같습니다:

```
void
```

```
foo(void)
{
  char instr[80];
  int i;

  /* instr이 user input을 가지고 있다고 가정 */

  if (convint(&i, instr) < 0) {
    /* error: invalid user input */
  }
  /* ... */
}
```

주의할 것은 이 `convint` 함수가 `atoi`와 똑같이 동작하는 게 아니라는 것입니다 — 단순히 리턴값과 인자만의 문제가 아닙니다. `atoi`와 `convint`의 다른 점은 크게 다음과 같습니다:

(a) `atoi`는 에러를 검사하지 않습니다. 따라서 입력이 `"12xyz"`인 경우에 12를 리턴합니다. `convint`는 입력 문자열이 완전한 경우에만 처리하므로, 입력이 `"12xyz"`인 경우에는 `int` 변환을 수행하지 않습니다 — 즉 에러로 처리합니다.

(b) `atoi`의 리턴 타입이 `int`인 반면, `convint`의 경우 `long int`입니다.

(c) `atoi`는 10진수만 처리할 수 있는 반면, `convint`는 `strtol`의 세번째 인자로 0을 주었기 때문에, 8진수나 16진수도 처리할 수 있습니다; 16진수의 경우 문자열 앞에 `"0x"`를 붙여야 하며, 8진수의 경우에 문자열 앞에 `"0"`을 붙여야 합니다.

Q 21.6 주어진 파일(`FILE *`)에서 한 줄씩 읽으려고 합니다. 질문 12.23에 따르면 `gets`를 쓰는 것은 좋지 않다고 했는데, 그럼 어떻게 해야 하나요? `fgets`을 쓰면, 버퍼의 크기를 지정해야 하기 때문에, 파일의 한 줄이 지정한 버퍼보다 긴 경우, 잘려나가 버립니다.

Answer 이 문제를 해결하기 위한 표준 함수는 존재하지 않습니다. 아주 긴 줄도 읽기 위해서는, 버퍼의 크기를 미리 지정하지 않고, 필요할 경우 늘려나가는 방식을 써야 합니다. 표준 함수는 아니지만, [GLIBC]는 이러한 일을 처리하기 위해 `getline` 함수를 제공합니다. 좀 더 향상된 기능이 필요하다면 GNU readline 패키지를 써 보기 바랍니다:

http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html

getline 함수는, 미리 malloc 또는 realloc을 써서 할당한 버퍼와 그 크기를 입력받아서, 한 줄을 읽어 버퍼에 저장해 줍니다. 이 때 \n 문자도 포함합니다. 만약 버퍼의 크기가 부족하면 realloc을 불러서, 버퍼의 크기를 자동으로 키워 줍니다. EOF를 만나거나 에러가 발생하면 -1을 리턴하고, 성공한 경우에는 문자열의 길이를 리턴합니다. 아래는 간단히 만든 getline 함수입니다:

```
ssize_t getline(char **lineptr, size_t *n, FILE *fp)
{
  char *buf, *ptr;
  size_t newsize, numbytes;
  int cont;
  int ch;
  int pos;

  if (lineptr == NULL || n == NULL || fp == NULL)
    return -1;

  buf = *lineptr;

  if (buf == NULL || *n < MIN_LINE_SIZE) {
    buf = realloc(*lineptr, DEFAULT_LINE_SIZE);
    if (!buf)
      return -1;
    *lineptr = buf;
    *n = DEFAULT_LINE_SIZE;
  }

  numbytes = *n;
  ptr = buf;
  cont = 1;

  while (cont) {
    /* fill buffer - leaving room for nul-terminator */
    while (--numbytes > 0) {
      if ((ch = getc(fp)) == EOF) {
```

```
      cont = 0;
      break;
    }
    else {
      *ptr++ = ch;
      if (ch == '\n') {
        cont = 0;
        break;
      }
    }
  }

  if (cont) {
    /* Buffer is too small so reallocate a larger buffer. */
    pos = ptr - buf;
    newsize = (*n << 1);
    buf = realloc(buf, newsize);
    if (!buf) {
      cont = 0;
      break;
    }

    /* After reallocating, continue in new buffer */
    *lineptr = buf;
    *n = newsize;
    ptr = buf + pos;
    numbytes = newsize - pos;
  }
}

/* if no input data, return failure */
if (ptr == buf)
  return -1;

/* otherwise, nul-terminate and return number of bytes read */
*ptr = '\0';
return (ssize_t)(ptr - buf);
```

```
}
```

위 함수를 써서, 표준 입력(stdin)에서 한 줄씩 읽어서 출력하는 함수는 다음과 같이 만들 수 있습니다:

```
int main(void)
{
  char *line = NULL;
  ssize_t len;
  size_t size;

  while ((len = getline(&line, &size, stdin)) >= 0) {
    printf("%s", line);
  }

  return 0;
}
```

References [GLIBC] `stdio-common/getline.c`

21.2 Debugging

이 단락에서는 debugging시 참고할 수 있는 여러가지 hint를 모았습니다.

Q 21.3 좋은 debugger를 추천해 주세요.

Answer 어떤 C 컴파일러가 가장 좋은 것인가?와 마찬가지로 해답이 나오기 힘든 질문입니다 ;-)

성능 좋고, 공개용인 GDB(GNU symbolic debugger)를 추천합니다. GDB를 바로 command-line에서 사용하는 것은 가장 쉬운 방법이긴 하지만, `vi`를 쓰지 않고 `ed`를 써서 작업하는 격이 됩니다.

GNU Emacs에서는 GDB를 편하게 쓸 수 있는 mode를 제공하며, DDD는 GDB를 써서 X window system에서 편하게 쓸 수 있는 interface를 제공합니다.

Q 21.4 제가 만든 프로그램은 `main()`을 끝낸 다음에 "segmentation fault"를 출력하고 끝나 버립니다. 왜 그럴까요?

Answer 간단히 말해 C 언어로 작성한 프로그램은 `main()`과 함께 시작하여 `main()`이 끝나면 종료하게 됩니다. 컴파일러가 만들어 낸 object file을 executable 파일로 변환하는 과정에서, command-line 인자인 `argc`와 `argv`를 설정하고, `main()`이 종료한 다음 OS에게 control를 넘겨주는 assembler 루틴이 들어가는 데, 프로그래머의 실수로 assembler routine으로 돌아가는 리턴 address가 저장되어 있는 stack을 건드리게 되면 (in technical term, write over the stack where the return address is stored.) 그러한 상황이 발생하곤 합니다.

일단 `main` 함수 내에서 선언된 local 배열이 있나 조사하고, 있다면, 그 배열의 범위를 벗어난 작업이 있었는지 확인하시기 바랍니다.

Q 21.5 이 글은 어떤 툴로 만든 것인가요?

Answer 이 글은 LaTeX와 한글 LaTeX패키지인 HLaTeX을 써서 만든 것입니다.

21.3 Internationalization

Q a Internationalization과 localization이라는 말이 무슨 뜻인가요?

Answer Internationalization, 또는 줄여서 I18N[2]은 어떤 소프트웨어가 하나 이상의 문화,지역 또는 locale에 맞게 쓰일 수 있도록 준비하는 것을 의미합니다. 이와 반대로 Localization, 또는 줄여서 L10N은 소프트웨어가 한 문화, 지역 또는 locale에 맞게 쓰일 수 있도록 고치는 작업을 뜻합니다.

References [Ken99] p. 17

Q b Multiple-byte, wide character가 정확히 뭔가요?

Answer byte 단위로 처리하는 encoding에서는 endianness(인디언)을 고려할 필요가 전혀 없습니다. 이러한 encoding에서 처리하는 문자를 multiple-byte character라고 합니다.

[2] 18이란 숫자는, 'internationalization'이 열여덟 글자로 이루어졌기 때문에 붙었습니다

Encoding	Encoding Length	Locale
ASCII	one-byte	*not applicable*
ISO-2022	one- and two-byte	CJKV
EUC	one- through four-byte, depending on locale	CJKV
GBK	one- and two-byte	China
Big Five	one- and two-byte	Taiwan
Big Five Plus	one- and two-byte	Taiwan
Shift-JIS	one- and two-byte	Japan
Johab	one- and two-byte	Korea
UHC	one- and two-byte	Korea
UTF-8	one- through six-byte	*not applicable*

이와 반대로, byte 단위로 처리할 수 없는 encoding에서는 endian 문제가 매우 중요한데, 이렇게 특별히 endian 처리가 필요한 문자를 wide character 라고 합니다.

Encoding	Encoding Length
UCS-2	16-bit fixed
UCS-4	32-bit fixed
UTF-16	16-bit variable-length
Unicode Version 2.0	*Same as UTF-16*

References [Ken99] pp. 25-26

Chapter 22

ISO C Standard Consideration

Q 22.1 Terminology

Answer **access** <execution-time action> to read or to modify the value of an object.

Note 1 Where only one of these two actions is meant, "read" or "modify" is used.

Note 2 "Modify" includes the case where the new value being stored is the same as the previous value.

Note 3 Expressions that are not evaluated do not access object.

alignment requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

argument actual argument or actual parameter(deprecated). expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation.

behavior external appeareance or action

implementation-defined behavior unspecified behavior where each implementation documents how the choice is made.

Example An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

locale-specific behavior behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

Example An example of locale-specific behavior is whether the `islower` function returns true for characters other than the 26 lowercase Latin characters.

undefined behavior Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

Note Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

Example An example of undefined behavior is the behavior on integer overflow.

unspecified behavior Behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

Example An example of unspecified behavior is the order in which the arguments to a function are evaluated.

bit Unit of data storage in the execution environment large enough to hold an object that may have one of two values.

Note It need not be possible to express the address of each individual bit of an object.

byte addressable unit of data storage large enough to hold any member of the basic character set of the execution environment. NOTE 1 It is possible to express the address of each individual byte of an object uniquely. NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

character <abstract> member of a set of elements used for the organization, control, or representation of data

character single-byte character <C> <C> bit representation that fits in a byte

multibyte character sequence of one or more bytes representing a member of the extended chracter set of either the source or the execution environment.

NOTE The extended character set is a superset of the basic character set.

Bibliography

[ANSI] American National Standards Institute, *American National Standard for Information Systems — Programming Language — C*, ANSI X3.159–1989 (질문 11.2 참고).

[ANSI Rationale] American National Standards Institute, *Rationale for American National Standard for Information Systems — Programming Language — C* (질문 11.2 참고).

[C99 Rationale] International Organization for Standardization, *Rationale for International Standard — Programming Language — C*

[Bentley] Jon Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982, ISBN 0-13-970244-X.

[Burki] David Burki, "Date Conversions," *The C Users Journal*, February 1993, pp. 29–34.

[Darwin] Ian F. Darwin, *Checking C Programs with lint*, O'Reilly, 1988, ISBN 0-937175-30-7.

[Goldberg] David Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23 #1, March, 1991, pp. 5–48.

[H&S] Samuel pp. Harbison and Guy L. Steele, Jr., *C: A Reference Manual*, Fourth Edition, Prentice-Hall, 1995, ISBN 0-13-326224-3. [H&S]

[H&S2002] Samuel pp. Harbison and Guy L. Steele, Jr., *C: A Reference Manual*, Fifth Edition, Prentice-Hall, 2002, ISBN 0-13-089592-X. [H&S5]

[PCS] Mark R. Horton, *Portable C Software*, Prentice Hall, 1990, ISBN 0-13-868050-7.

[POSIX] Institute of Electrical and Electronics Engineers, *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface(API) [C Language]*, IEEE Std. 1003.1, [C89]/IEC 9945–1.

[SUS] Single UNIX Specification Version 3 – IEEE Std 1003.1, 2004 Edition, Single UNIX Specification Version 3 Formerly known as [POSIX]. http://www.unix.org/version3/

[C89] International Organization for Standardization, ISO 9899:1990 (질문 11.2 참고).

[C9X] International Organization for Standardization, WG14/N794 Working Draft (질문 11.1과 11.2b 참고).

[C99] Internatinoal Organization for Standardization, ISO 9899:1999, *Programming Languages — C*

[1] Brian W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, Second Edition, McGraw-Hill, 1978, ISBN 0-07-034207-5.

[K&R1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978, ISBN 0-13-110163-3.

[K&R2] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition, Prentice Hall, 1988, ISBN 0-13-110362-8, 0-13-110370-9. (질문 18.10 참고)

[Knuth] Donald E. Knuth, *The Art of Computer Programming*. Volume 1: *Fundamental Algorithms*, Second Edition, Addison-Wesley, 1973, ISBN 0-201-03809-9. Volume 2: *Seminumerical Algorithms*, Second Edition, Addison-Wesley, 1981, ISBN 0-201-03822-6. Volume 3: *Sorting and Searching*, Addison-Wesley, 1973, ISBN 0-201-03803-X. (새 증보판이 나올 예정임!)

[CT&P] Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989, ISBN 0-201-17928-8. [CT&P]

[Marsaglia] G. Marsaglia and T.A. Bray, "A Convenient Method for Generating Normal Variables," *SIAM Review*, Vol. 6 #3, July, 1964.

[Park] Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones are Hard to Find," *Communications of the ACM*, Vol. 31 #10, October, 1988, pp. 1192–1201 (also technical correspondence August, 1989, pp. 1020–1024, and July, 1993, pp. 108–110).

[Plauger] P.J. Plauger, *The Standard C Library*, Prentice Hall, 1992, ISBN 0-13-131509-9.

[Plum] Thomas Plum, *C Programming Guidelines*, Second Edition, Plum Hall, 1989, ISBN 0-911537-07-4.

[Saul] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian pp. Flannery, *Numerical Recipes in C*, Second Edition, Cambridge University Press, 1992, ISBN 0-521-43108-5.

[Dale] Dale Schumacher, Ed., *Software Solutions in C*, AP Professional, 1994, ISBN 0-12-632360-7.

[Robert] Robert Sedgewick, *Algorithms in C*, Addison-Wesley, 1990, ISBN 0-201-51425-7. (새 증보판이 준비되고 있습니다. 첫판 ISBN 0-201-31452-5.)

[Simonyi] Charles Simonyi and Martin Heller, "The Hungarian Revolution," *Byte*, August, 1991, pp.131-138.

[Straker] David Straker, *C Style: Standards and Guidelines*, Prentice Hall, ISBN 0-13-116898-3.

[Summit] Steve Summit, *C Programming FAQs: Frequently Asked Questions*, Addison-Wesley, 1995, ISBN 0-201-84519-9. [이 FAQ list를 확장 정리한 책. `http://www.eskimo.com/~scs/C-faq/book/Errata.html` 참고]

[Wu] Sun Wu and Udi Manber, "AGREP — A Fast Approximate Pattern-Matching Tool," USENIX Conference Proceedings, Winter, 1992, pp. 153–162.

개정된 Indian Hill 스타일 가이드 (질문 17.9 참고)에는 다른 참고 문헌들이 있습니다. 질문 18.10을 참고하기 바랍니다.

[GLIBC] GNU C Library version 2.1.3의 소스. 자세한 것은 `http://www.gnu.org/software/glibc/`를 참고하시기 바랍니다.

[Lewine] Donald Lewine, "POSIX PROGRAMMER's GUIDE — Writing Protable UNIX Programs," O'Reilly & Associates, Inc., ISBN 0-937175-73-0

[Raymond] Eric S. Raymond, *The New Hacker's Dictionary*. Online version 은 Eric S. Raymond씨의 홈페이지인, `http://www.tuxedo.org/~esr/`에서 구할 수 있습니다.

[Stevens98] W. Richard Stevens, *UNIX Network Programming, Volumn 1, Networking APIs: Sockets and XTI*, 2nd edition, Prentice Hall, ISBN 0-13-490012-X.
`http://www.kohala.com/start/unpv12e.html`

[Ken99] Ken Lunde, *CJKV Information Processing*, 1st edition, O'Reilly, ISBN 1-56592-224-7