Discovering Modern C++ 2/E

필요한 것만 골라 배우는 모던 C++

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

Discovering Modern C++ 2nd Edition
by Peter Gottschling

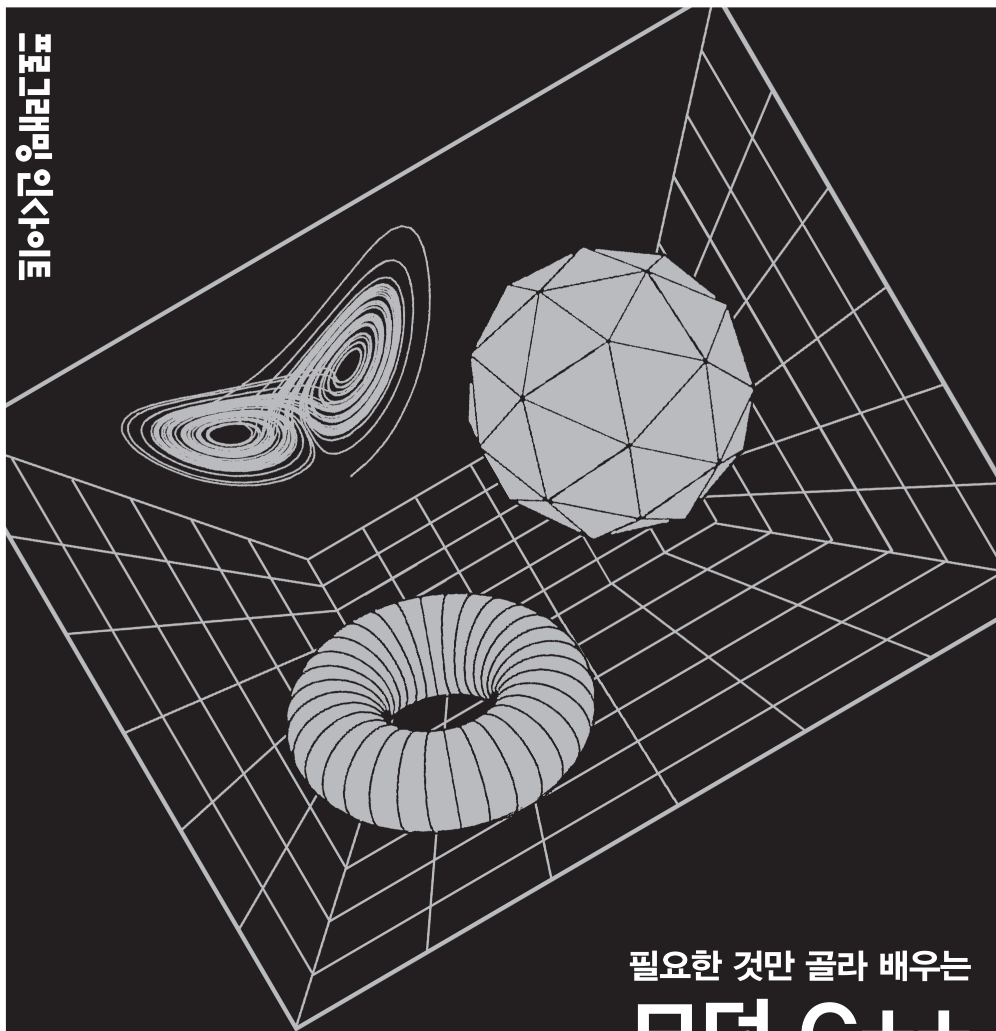
Authorized translation from the English language edition, entitled DISCOVERING MODERN C++ 2nd EDITION by PETER GOTTSCHLING, published by Pearson Education, Inc.
Copyright © 2021 Pearson Education Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
Korean language edition published by INSIGHT PRESS, Copyright © 2022
Korean language translation rights arranged with PEARSON EDUCATION, INC. through Agency-One, Seoul, Korea
이 책의 한국어판 저작권은 에이전시 원을 통해 저작권자와의 독점 계약으로 인사이트 출판사에 있습니다.
저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

필요한 것만 골라 배우는 모던 C++

전자책 1쇄 발행 2022년 10월 19일 **지은이** 페터 고칠링 **옮긴이** 류광 **편집** 정수진 **교정** 오현숙 **본문 디자인** 최우정 **펴낸이** 한기성 **펴낸곳** (주)도서출판인사이트 **등록번호** 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울시 마포구 연남로5길 19-5 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** https://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr
ISBN 978-89-6626-372-1

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/



필요한 것만 골라 배우는

모던 C++

페터 고칠링 지음 | 류광 옮김

인사이트

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

차례

옮긴이의 글 viii

서문 x

C++을 배우는 이유 xi
이 책을 읽는 이유 xi
미녀와 야수 xii
과학과 공학을 위한 프로그래밍 언어 xiii
조판 관례 xv
감사의 글 xvii

1장 C++ 기초 1

1.1 생애 첫 C++ 프로그램 1
1.2 변수 6
1.3 연산자 19
1.4 표현식과 문장 34
1.5 함수 45
1.6 오류 처리 54
1.7 입출력 63
1.8 배열, 포인터, 참조 78
1.9 소프트웨어 프로젝트의 구조화 96
1.10 연습문제 104

2장 클래스 107

2.1 기술적 세부사항보다는 보편적 의미를 체현하는 프로그래밍 접근 방식 107
2.2 멤버 110
2.3 값의 설정: 생성과 배정 118
2.4 소멸자 157
2.5 메서드 생성 요약 165

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

2.6 멤버 변수 접근 167
2.7 연산자 중복적재의 설계 174
2.8 연습문제 186

3장 일반적 프로그래밍 189

3.1 함수 템플릿 190
3.2 이름공간과 함수 조회 203
3.3 클래스 템플릿 215
3.4 형식 연역과 형식 정의 226
3.5 템플릿 특수화 237
3.6 비형식 템플릿 매개변수 254
3.7 함수자 258
3.8 람다 269
3.9 변수 템플릿 277
3.10 콘셉트를 이용한 프로그래밍 279
3.11 가변 인수 템플릿 290
3.12 연습문제 301

4장 표준 라이브러리 307

4.1 표준 템플릿 라이브러리(STL) 308
4.2 수치 347
4.3 메타프로그래밍 365
4.4 유틸리티 370
4.5 시간 386
4.6 동시성 390
4.7 표준 라이브러리 이외의 과학 라이브러리 409
4.8 연습문제 413

5장 메타프로그래밍 417

5.1 컴파일러가 계산하게 하라 417
5.2 형식 정보의 제공과 활용 430
5.3 표현식 템플릿 458
5.4 메타조율: 나만의 컴파일러 최적화 작성 471
5.5 의미론적 콘셉트를 이용한 최적화 505
5.6 튜링 완전성 514

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

5.7 연습문제 517

6장 객체 지향적 프로그래밍 521

6.1 기본 원리 522
6.2 중복성 제거 541
6.3 다중 상속 543
6.4 하위형식화(subtyping)를 통한 동적 선택 550
6.5 형식의 변환 554
6.6 고급 기법 565
6.7 연습문제 576

7장 과학 프로젝트 579

7.1 상미분방정식 해법의 구현 579
7.2 프로젝트 만들기 593
7.3 모듈 610
7.4 맺음말 616

부록A 지저분한 세부사항 619

A.1 좋은 과학 소프트웨어의 요건 619
A.2 기초 관련 세부사항 627
A.3 사례 연구: 역행렬 구하기 640
A.4 클래스 관련 세부사항 654
A.5 메서드 생성 659
A.6 템플릿 세부사항 674
A.7 표준 라이브러리의 세부사항 681
A.8 구식 스타일로 구현한 동적 선택 683
A.9 메타프로그래밍 세부사항 683
A.10 C 코드 링크 694

부록B 프로그래밍 도구 699

B.1 `g++` 699
B.2 디버깅 701
B.3 메모리 분석 706
B.4 `gnuplot` 708

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

B.5 유닉스, 리눅스, 맥OS 709

부록C C++ 언어 정의 713

C.1 값 범주 713
C.2 연산자 요약 714
C.3 변환 규칙 718

참고문헌 722

찾아보기 728

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

옮긴이의 글

2022년 1월에 출간된 《C++20: 풍부한 예제로 익히는 핵심 기능》(원서 *C++20: Get the Details*)을 번역하면서 C++20뿐만 아니라 '현대적(modern)' C++로 분류되는 C++11, C++14, C++17까지 아우르는 책이 있으면 좋겠다는 생각이 들었는데, 딱 그런 책을 번역하게 되어서 무척 기쁩니다. 이 책은 단지 C++ 언어의 개별 문법과 기능을 단편적으로 이야기하는 것이 아니라 현대적 C++ 프로그래밍 기법을 가르친다는 점에서 더욱 가치가 있습니다. 또한, 이제는 어느 정도 대중화되고 자료도 많은 C++11/C++14와 최신 표준으로 주목받는 C++20 사이에 "끼어서" 덜 주목받는 C++17에 관한 내용이 비교적 풍부하다는 점도 마음에 듭니다.

이 번역서의 저본은 2022년에 출간된 *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers*(제2판)입니다. 원서의 부제가 암시하듯이 이 책은 C++을 과학 연구나 공학 프로젝트에 사용한다는 가정을 두고 현대적 C++ 프로그래밍을 논의합니다. 그러다 보니 개별적인 문법이나 기능을 설명하는 짧은 예제를 제외한 본격적인 예제들은 대부분 과학(특히 물리학)이나 공학 문제와 관련이 있습니다. 그래서 해당 분야의 배경지식이 없는 독자라면 동기 부여가 덜 되거나 흥미가 떨어질 가능성이 있습니다.

그렇지만 예제를 조금만 유심히 들여다보면 저자가 예제를 통해서 이야기하고자 하는 보편적인 가치들을 발견할 수 있을 것입니다. 예를 들어 다수의 예제(특히 행렬과 관련한)에는 "저수준의 성능과 고수준의 생산성을 모두 노린다"라는, C++ 언어 자체가 꾸준히 추구해온 가치가 깔려 있습니다. 그 밖에도 공통 인터페이스의 중요성이나 추상화의 위력 등등, 꼭 과학과 공학에만 적용되는 것은 아닌 여러 개념과 가치를 발견할 수 있을 것입니다. 그런 만큼, 과학자나 공학자가 아닌 독자라도 생소한 전문용어의 장벽을 적당히 넘기고(위키백과 등을 이용해서 대략 어떤 범주인지만 파악해도 충분하다고 봅니다) 보편적인 가치들에 주목한다면 이 책을 최대한 활용할 수 있을 것입니다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

이 책을 번역하면서 현대적 C++뿐만 아니라 프로그래밍 자체에 관해서도 많은 것을 배웠고, 생각할 거리도 많이 생겼습니다. 독자 여러분도 그랬으면 좋겠습니다. 그런 긍정적인 독서 경험에 방해가 되는 오탈자와 오역이 없길 바랄 뿐입니다. 제 홈페이지(https://occamsrazr.net/)에 이 책을 위한 공간을 마련해 두었으니, 오탈자나 오역을 발견하면 꼭 알려 주세요. 이 책을 위한 페이지는 홈페이지 오른쪽 상단 '번역서 정보' 링크를 통해서 찾을 수 있습니다.

이번에도 훌륭하고 꼭 필요한 C++ 전문서를 선정해서 제게 번역을 맡겨 주신 도서출판 인사이트 한기성 사장님과 제가 번역과 교정에만 집중할 수 있도록 모든 것을 잘 안배해 주신 정수진 편집자님, 수식 많은 까다로운 책을 문제없이 조판해 주신 조판 디자이너 최우정 님께 감사드립니다. 그리고 이 책의 출간에 기여한 모든 관련자분께도 감사의 뜻을 전합니다. 마지막으로, 교정 전문가로서 명백한 오탈자와 오역은 물론이고 고치면 문장이 훨씬 나아지는 미묘한 표현상의 문제점을 다수 지적해 준 아내 오현숙에게 감사와 사랑의 마음을 보냅니다.

재미있게 읽으시길!

―옮긴이 류광

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

서문

이 세상은 C++(그리고 C++의 C 부분집합) 위에 세워져 있다.

—허브 서터Herb Sutter

구글, 아마존, 페이스북의 기반구조는 C++ 프로그래밍 언어로 설계, 구현된 구성요소들과 서비스들에 기초한다. 운영체제, 네트워크 기기, 저장 시스템의 기술 스택 중 상당 부분이 C++로 구현되어 있다. 통신 시스템들을 보면, 거의 모든 유선전화와 이동전화의 연결을 C++ 소프트웨어가 제어한다. 그리고 제조업과 운송업 역시, 예를 들어 자동 통행료 수거 시스템이나 승용차/트럭/버스의 자율주행 시스템 같은 핵심 구성요소들이 C++에 의존한다.

과학과 공학에서 오늘날 쓰이는 대부분의 고품질 소프트웨어 패키지는 C++로 구현된다. C++ 언어의 위력은 프로젝트의 크기가 일정 수준을 넘어설 때, 그리고 사소하지 않은 자료 구조(data structure)와 알고리즘이 요구될 때 빛을 발한다. 현재 계산 과학 분야에서 다수의(대부분은 아니라고 해도) 시뮬레이션 소프트웨어(FLUENT, Abaqus, deal.II, FEniCS, OpenFOAM, G+Smo 등)가 C++로 구현되는 것은 놀랄 일이 아니다. 내장형 시스템(embedded system)들도, 내장형 프로세서와 컴파일러의 발전 덕분에 C++로 구현되는 사례가 늘고 있다. 그리고 사물 인터넷(IoT)이나 내장형 에지 지능(embedded edge intelligence) 같은 새로운 응용 영역은 모두 TensorFlow나 Caffe2, CNTK 같은 C++ 플랫폼들이 주도한다.

우리가 일상에서 사용하는 핵심 서비스들은 C++에 기반한다. 휴대전화, 자동차, 통신, 산업 기반, 그리고 미디어와 연예 오락 서비스의 핵심 요소들에는 모두 C++ 구성요소가 들어 있다. 현대 사회에서 C++ 서비스와 응용 프로그램은 어디에나 있다. 이유는 간단하다. C++ 언어는 다양한 요구와 제안을 수용하면서 발전했으며, 프로그래밍 생산성과 실행 효율성 면에서 여러 방식으로 혁신을 이루었다. 그 두 특성 덕분에 C++은 대규모로 실행되어야 하는 응용 프로그램을 위한 언어로 선택받았다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

C++을[†] 배우는 이유

하드웨어에 대단히 가까운 저수준 프로그래밍에서 추상적인 고수준 프로그래밍에 이르기까지 프로그래밍의 스펙트럼 전체를 포괄하는 언어는 사실상 C++뿐이다. 직접적인 메모리 관리 같은 저수준 프로그래밍을 통해서 프로그래머는 프로그램 실행 도중에 실제로 벌어지는 일들을 파악할 수 있으며, 그런 지식과 경험은 다른 언어로 된 프로그램의 작동 방식을 이해하는 데 도움이 된다. C++을 이용하면 전문가가 초인적인 노력을 들여서 작성한 기계어 코드에 근접한 성능을 내는 극도로 효율적인 프로그램을 작성할 수 있다. 그렇지만 그런 하드코어 성능 조율은 잠시 뒤로 미루고, 일단은 명확하고 표현력 있는(expressive) 소프트웨어를 작성하는 데 초점을 두는 것이 바람직하다.

명확하고 표현력 있는 소프트웨어를 작성하는 데 필요한 것이 C++의 고수준 기능들이다. C++ 언어는 광범위한 프로그래밍 패러다임들을 직접 지원한다. 이 책에서 다루는 몇 가지를 거론하자면 객체 지향 프로그래밍(제6장), 일반적 프로그래밍(제3장), 메타프로그래밍(제5장), 동시적 프로그래밍(§4.6), 절차적 프로그래밍(§1.5)이 있다.

C++ 공동체는 C++을 위해 여러 프로그래밍 기법을 고안했다. RAII(§2.4.2.1)나 표현식 템플릿(§5.3)이 그러한 예이다. C++ 공동체에는 언어 자체를 바꾸지 않고도 새로운 기법을 고안해서 구현한 사례가 많은데, 이는 C++의 뛰어난 표현력 덕분이다. 어쩌면 이 책을 읽는 여러분도 언젠가 새로운 프로그래밍 기법을 고안할지 누가 알겠는가?

이 책을 읽는 이유

이 책의 내용은 진짜 사람들의 시험을 거친 것이다. 필자는 독일의 한 대학교에서 3년간 매년 두 학기씩 "C++ for Scientists"라는 제목의 강좌를 진행했다. 학생들은 대부분 수학과였지만, 물리학이나 공학 계열 학생들도 있었다. 대부분의 학생은 C++을 전혀 모르는 상태였지만, 학기말에는 표현식 템플릿(§5.3) 같은 고급 기법도 구현할 수 있었다. 그 학생들과는 달리 여러분은 이 책을 여러분 자신의 속도에 맞게 공부할 수 있다. 본문의 주된 경로를 따라 빠르게 진도를 나아

† [옮긴이] 오랜 관례에 따라 이 번역서는 독자가 C++을 '씨뿔뿔'이라고 발음한다고 가정한다. 그래서 C++'를'이 아니라 C++'을'이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

가도 되고, 때때로 부록 A에 있는 추가적인 예제들과 배경지식을 읽으면서 천천히 나아가도 된다.

미녀와 야수

C++ 프로그램을 작성하는 방식은 여러 가지이다. 이 책은 단순하고 직접적인 스타일에서 좀 더 정교한 스타일들로 여러분을 매끄럽게 이끈다. 정교한 스타일에서는 C++의 고급 기법들이 쓰인다. 그런 기법들이 처음에는 좀 난해하고 막막하겠지만, 점차 익숙해질 것이다. 사실 고수준 프로그래밍은 저수준 프로그래밍보다 적용 범위가 더 넓을 뿐만 아니라, 가독성이 더 좋고 성능 역시 저수준 프로그래밍과 비슷하거나 더 나은 코드가 나올 때가 많다.

C++ 프로그래밍 스타일에 대한 맛보기로 간단한 예제를 하나 살펴보자. 단계 크기(step size)가 고정된 경사하강법(gradient descent) 알고리즘의 원리는 아주 간단하다. 함수 $f(x)$가 주어졌을 때, 현재 위치에서 가장 가파르게 아래로 내려가는 기울기 $g(x)$를 계산하고, 고정된 단계 크기만큼 그 기울기를 따라 내려간다. 이런 과정을 반복하면 언젠가는 극소점(local minimum)에 도달하게 된다. 다음은 이 알고리즘을 의사코드로 표현한 것이다.

알고리즘 1: 경사하강법

입력: 시작 값 x, 단계 크기 s, 종료 기준 ε, 함수 f, 기울기 g

출력: 극소점 x

1 **do**
2 $\quad x = x - s \cdot g(x)$
3 **while** $|\Delta f(x)| \geq \varepsilon$

다음은 이 간단한 알고리즘을 두 가지 C++ 프로그래밍 스타일로 구현한 것이다. 기술적인 세부사항을 이해하려 들지 말고 그냥 전체적으로 훑어보기 바란다.

```
void gradient_descent(double* x,
    double* y, double s, double eps,
    double(*f)(double, double),
    double(*gx)(double, double),
    double(*gy)(double, double))
{
    double val= f(*x, *y), delta;
    do {
```

```
template <typename Value, typename P1,
          typename P2, typename F,
          typename G>
Value gradient_descent(Value x, P1 s,
    P2 eps, F f, G g)
{
    auto val= f(x), delta= val;
    do {
```


미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

```
        *x-= s * gx(*x, *y);
        *y-= s * gy(*x, *y);
        double new_val= f(*x, *y);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
}
```

```
        x-= s * g(x);
        auto new_val= f(x);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
    return x;
}
```

처음에는 두 코드가 상당히 비슷해 보일 것이다. 둘 중 이 책이 선호하는 스타일이 어떤 것인지는 잠시 후에 이야기하겠다. 첫(왼쪽) 버전은 사실 순수한 C 코드이며, 실제로 C 컴파일러로 컴파일할 수 있다. 이 버전의 장점은 구체적인 용도에 대단히 최적화되어 있다는 점이다. 이 버전은 `double` 값들(**강조된** 부분들에 주목)을 다루는 하나의 2차원 함수를 최대한 효율적으로 구현하려 노력한 결과이다. 그러나 이 책은 용도가 더 광범위한 둘째 버전을 선호한다. 둘째 버전은 임의의 형식(type)의 값을 다루는 임의의 차원의 두 함수를 지원한다(**강조된** 부분들이 이러한 일반성을 위한 장치이다). 놀랍게도, 이런 범용적인 구현이 첫 버전보다 덜 효율적인 것도 아니다. 오히려 둘째 버전은 `F`와 `G`가 인라인화(§1.5.3)될 여지가 있으며, 그러면 함수 호출의 추가부담(overhead)이 사라지므로 성능이 향상된다. 그러나 첫 버전은 (추한) 함수 포인터를 명시적으로 사용하기 때문에, 컴파일러는 그런 최적화를 적용하지 못한다.

참을성 있는 독자를 위해, 구식 스타일과 새 스타일을 비교하는 좀 더 긴 예제를 부록 A에 수록해 두었다(§A.1). 그 예제는 이 장난감 수준의 예제보다 현대적인 프로그래밍 스타일의 장점을 좀 더 극명하게 보여준다. 그렇지만 많은 내용을 너무 급하게 제시하면 오히려 독자의 학습에 방해가 될 것이므로 부록으로 뺐다.

과학과 공학을 위한 프로그래밍 언어

모든 종류의 수치 해석 소프트웨어를 효율성 손실 없이 C++로 작성할 수 있다면 좋을 것이다. 그렇지만 C++의 형식 시스템을 훼손하지 않고도 그런 목표를 달성할 수 있는 뭔가가 발견되지 않는다면, 포트란이나 어셈블러, 또는 아키텍처에 특화된 확장 기능에 의존하는 것이 더 나을 수 있다.

—비야네 스트롭스트룹Bjarne Stroustrup

과학 및 공학 소프트웨어는 다양한 언어로 작성된다. 주어진 프로젝트에 가장 적합한 언어가 무엇인지는 프로젝트의 목표와 가용 자원에 달려 있다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

- MATLAB이나 Mathematica, R 같은 수학/통계학 패키지들은 패키지에 구현된 기존 알고리즘들을 활용할 수 있는 상황에서는 훌륭하다. 그러나 조밀한(fine-grained) 연산들(이를테면 스칼라 연산)이 관여하는 독자적인 알고리즘을 구현해야 하는 경우에는 성능이 크게 떨어진다. 풀어야 할 문제가 작거나 사용자의 인내심이 무한하다면 별문제가 아니겠지만, 그렇지 않다면 다른 언어를 고려해야 한다.
- 파이썬[Python]은 빠른 소프트웨어 개발(rapid software development, RAD)에 적합하며, "scipy"나 "numpy" 같은 과학 라이브러리들이 갖추어져 있다. 그런 라이브러리들은 흔히 C나 C++로 구현되어 있기 때문에, 그런 라이브러리를 사용하는 파이썬 응용 프로그램은 상당히 효율적이다. 그러나 조밀한 연산들이 관여하는 알고리즘을 직접 구현하는 경우에는 역시 성능이 떨어진다. 파이썬은 소규모 또는 중간 규모 과제를 효율적으로 구현하는 데 탁월하다. 그러나 프로젝트가 충분히 커지면, 좀 더 엄격한 규칙이 적용되는(이를테면 인수 형식들이 부합하지 않으면 설정을 거부하는 등) 언어가 필요해진다.
- 수학 패키지들처럼 포트란도 잘 조율된 기존 연산들(밀집행렬 연산 등)을 사용할 수 있을 때는 훌륭하다. 포트란은 노교수가 낸 과제(숙제)를 푸는 데 적합하다(노교수들은 포트란으로 풀기 쉬운 과제들만 내니까). 그러나 필자의 경험으로 포트란은 새로운 자료 구조를 도입하기가 상당히 번거롭다. 그리고 포트란으로 대규모 시뮬레이션 프로그램을 작성하기란 대단히 어려운 일이다. 요즘 자발적으로 그런 프로그램을 포트란으로 작성하는 사람은 소수이며, 점점 줄어들고 있다.
- C는 좋은 성능을 제공하며, 아주 많은 소프트웨어가 C로 작성되었다. C의 핵심 언어는 비교적 작고 배우기 쉽다. 그러나 크고 버그 없는 소프트웨어를 단순하고 위험한 언어 기능들로 작성하기란 쉬운 일이 아니다. 특히 포인터(§1.8.2)와 매크로(§1.9.2.1)가 그런 기능들이다. 가장 최근의 C 표준은 C17인데, 2017에 발표되었기 때문에 그런 이름이 붙었다. 늦든 빠르든 대부분의(전부는 아니지만) C 기능은 C++에도 도입된다.
- 자바나 C#, PHP 같은 언어는 응용 프로그램이 웹이나 GUI 위주이고 수치 계산이 그리 많지 않은 경우라면 좋은 선택일 수 있다.
- C++은 고성능 대규모 고품질 소프트웨어의 개발에 특히나 적합하다. 그렇다고 개발 과정이 반드시 느리고 고통스러운 것은 아니다. 적절한 추상을 도입

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

한다면 C++ 프로그램도 상당히 빠르게 작성할 수 있다. 필자는 향후 C++ 표준들에 더 많은 과학 라이브러리가 포함될 가능성이 크다고 생각한다.

아는 언어가 많을수록 선택의 폭이 넓어지는 것은 당연하다. 게다가, 그 언어들을 더 깊게 알수록 좀 더 근거 있는 선택이 가능하다. 그리고 대규모 프로젝트들은 여러 가지 언어로 작성된 구성요소들로 이루어질 때가 많다. 그렇지만 대부분의 경우 성능이 중요한 핵심부는 C나 C++로 구현된다. 다 떠나서, C++ 학습은 흥미로운 여정이며, C++을 깊게 이해하는 것은 여러분이 훌륭한 프로그래머로 성장하는 데 도움이 된다.

조판 관례

새로운 용어는 **돋움체**로 표시한다. C++ 소스 코드는 `int i = 1;`처럼 고정폭 글꼴로 표시하고, 중요한 세부사항은 **`auto`** `i = 1`처럼 굵게 강조한다. 클래스, 함수, 변수, 상수의 이름에는 영문 소문자와 밑줄 문자를 사용한다. 행렬은 예외인데, 일반적으로 행렬을 나타내는 변수는 대문자 하나로 표기한다. 템플릿 매개변수와 콘셉트는 대문자로 시작하며, 소문자들 다음에 또 다른 대문자가 올 수도 있다. 즉, 소위 낙타등 표기법(CamelCase)를 따른다. 프로그램의 출력이나 셸 명령은

```
g++ hello.cpp
```

처럼 소스 코드와 구별되는 형태로 표시한다

C++11이나 C++14, C++17, C++20의 기능이 필요한 항목(장, 절, 문단 등)에는 페이지 여백에 작은 상자로 해당 표준 이름을 명시했다. 단, C++11의 기능을 가볍게만(C++03의 표현식으로 쉽사리 대체할 수 있는 수준으로) 사용한 경우에는 C++11을 명시하지 않았다.

아주 짧은 예제 코드 조각들을 제외할 때, 이 책의 모든 예제 프로그램은 적어도 하나의 컴파일러로 시험한 것이다. 대부분의 경우는 세 가지 컴파일러(`g++`, `clang++`, Visual Studio)로 시험해 보았다. 이해를 돕기 위해 모든 예제를 최대한 짧게 작성했음을 밝혀 둔다. 그런 만큼, 실제 응용 프로그램에 쓰이는 모든 기능과 기법이 예제에 반영되지는 않았다. 또한, 아직 설명하지 않은 기능은 최소한으로만 사용했다. 이 책을 끝까지 다 읽은 후에 예제들을 처음부터 다시 훑어보

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

면서, 새로 배운 기능들과 기법들을 활용해서 기존 예제를 어떻게 개선할 수 있을지 생각해 보는 것도 좋을 것이다.

C++20의 기능을 사용하는 예제들은 여러분의 시스템에서 제대로 컴파일 · 실행되지 않을 수 있음을 주의하기 바란다. 이 책을 쓰는 현재 모든 컴파일러가 C++20의 모든 기능을 지원하지는 않는다. 그리고 지원한다고 되어 있는 기능이라도 100% 정확하게 지원하지는 않을 수 있다. `<format>` 라이브러리처럼 아직 컴파일러들이 표준 구현을 제공하지 않는 몇몇 새 표준 라이브러리에 대해서는 표준의 원형(prototype)이 된 라이브러리를 사용하기도 했다.

⇒ `directory/source_code.cpp`

문단이나 절에서 하나의 프로그램을 주되게 예제로 사용하는 경우, 문단이나 절 도입부에 위처럼 오른쪽 화살표와 함께 프로그램 소스 코드의 경로를 표시해 두었다. 모든 예제 프로그램은 깃허브의 공개 저장소 *https://github.com/petergottschling/dmc3*에 있다. 다음은 이 깃허브 저장소를 여러분의 시스템에 복제하는 명령이다.

```
git clone https://github.com/petergottschling/dmc3.git
```

Windows에서는 TortoiseGit(*https://tortoisegit.org*) 같은 GUI 기반 깃 클라이언트가 더 편할 것이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

감사의 글

연대순으로 감사의 글을 전한다. 이 책의 단초가 된 80쪽 분량의 교재를 만든 카를 메르베르겐Karl Meerbergen과 그의 동료들에게 감사한다. 칼과 나는 2008년에 뢰번 카톨릭 대학교에서 그 교재로 단기 집중 강좌를 진행했다. 시간이 흐르면서 대부분의 문장을 고쳐쓰긴 했지만, 원래의 문서는 이 책의 전체 저술 과정을 이끈 초기 원동력으로 작용했다. §7.1 "ODE 해법의 구현"에 기여한 마리오 물란스키Mario Mulansky에게 큰 빚을 졌다.

제1판의 원고를 사소한 세부사항까지 철저히 검수하고 표준 준수와 가독성 측면에서 여러 개선안을 제안한 얀 크리스티안 판 빙켈Jan Christiaan van Winkel과 파비오 프라카시Fabio Fracassi에 크나큰 감사의 마음을 전한다. 제2판의 감수자 척 앨리슨Chuck Allison, 션 페어런트Sean Parent, 마르크 그레구아Marc Gregoire에게도 그만큼 감사한다. 특히 마르크는 이 책의 모든 세부사항을 일일이 점검했다. 또한, 얀 크리스티안 판 빙켈은 제1판의 제작 과정에도 크게 기여했음을 특별히 언급한다.

이 책의 기획에 전략적인 팁을 제공하고, 원서 출판사 애디슨-웨슬리에 나를 소개하고, 그의 잘 준비된 저작물을 재사용하도록 허락해주고, (혹시 잊은 독자를 위해 말하자면) C++을 만들어 낸 비야네 스트롭스트룹Bjarne Stroustrup에게 특별한 감사의 뜻을 전한다. 이상의 모든 사람은 내 아이디어를 최신 언어 기능들로 최대한 갱신하도록 나를 밀어부쳤다.

또한, 여러 제안을 한 카르슈텐 아네르트Karsten Ahnert와 서문의 군더더기를 제거하는 데 도와준 마르쿠스 아벨Markus Abel과 테오도레 옴칙트Theodore Omtzigt에게 감사한다.

§4.2.2.6을 위해 난수의 흥미로운 응용을 고민하던 나에게 얀 루틀Jan Rudl은 자신의 강의[60]에 사용한 주가 변동 예제를 제안했다.

고맙게도 드레스덴 공과 대학은 3년간 내게 수학과의 C++ 강좌를 맡겼다. 생산적인 피드백을 제공한 모든 학생에게 감사한다. 또한, (내가 따로 진행하는)

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

C++ 교육 과정의 수강생들에게도 감사의 뜻을 전한다.

원서 출판사의 편집자 그레그 도엔치Greg Doench에도 큰 빚을 졌다. 도엔치는 반은 진지하고 반은 실없는 문체를 받아주었을 뿐만 아니라 전략적 결정 사항들에 대해 우리 둘 다 만족하는 결론이 나올 때까지 길고 긴 논의를 견뎠고 전문적인 지원을 아끼지 않았다. 그의 지원이 없었다면 이 책은 출판되지 못했을 것이다.

마지막으로, 나와 함께 보낼 시간을 수없이 희생한 내 아이들 야니스Yanis, 아니사Anissa, 빈첸트Vincent, 다니엘Daniel에게 마음 가득한 고마움을 전한다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1장

D i s c o v e r i n g   M o d e r n   C + +

C++ 기초

아이들아,
컴퓨터 가르쳐 주면서 너무 놀리지 말아.
너희들 숟가락질 나한테 배웠으면서.

—수 피츠모리스Sue Fitzmaurice

이 책의 첫 번째 장(chapter)인 제1장에서는 C++의 가장 기초적인 기능을 훑어본다. 이 기능들을 이후의 장들에서 다른 각도에서 살펴볼 것이다. 이번 장에서든 이 책 전체에서는, 각 기능의 모든 가능한 세부사항을 일일이 거론하지는 않는다. 어차피 그것은 한 권의 책으로는 불가능한 일이다. 특정 기능에 대해 대한 좀 더 세부적인 의문 사항이 생긴다면, *https://en.cppreference.com*에 있는 온라인 레퍼런스를 추천한다.

1.1 생애 첫 C++ 프로그램

C++ 언어로 들어가는 첫 관문으로 다음과 같은 예제 프로그램을 제시하겠다.

```
#include <iostream>

int main ()
{
    std::cout ≪ "The answer to the Ultimate Question of Life,\n"
              ≪ "the Universe, and Everything is:"
              ≪ std::endl ≪ 6 * 7 ≪ std::endl;
    return 0;
}
```

이 프로그램을 실행하면 다음이 출력된다.

```
The answer to the Ultimate Question of Life,
the Universe, and Everything is:
42
```


미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

더글러스 애덤스Douglas Adams에 따르면[2] 그렇다고 한다.† 이 짧은 예제는 C++의 다음과 같은 여러 기능을 보여준다.

- 입력 기능과 출력 기능은 핵심 언어(core language)‡의 일부가 아니라 라이브러리가 제공한다. 즉, 입출력 기능은 반드시 명시적으로 프로그램에 포함시켜야(include) 한다. 그렇지 않으면 프로그램은 데이터를 읽거나 쓸 수 없다.
- 표준 입출력 라이브러리에는 스트림stream 모형이 있다. <iostream>이라는 표준 헤더의 이름이 이를 반영한 것이다. 스트림 모형의 기능을 사용하기 위해, 예제의 첫 행에서 #include <iostream>으로 <iostream>을 프로그램에 포함시킨다.
- 모든 C++ 프로그램은 main이라는 함수의 호출로 실행이 시작된다. main은 return 문으로 하나의 정수를 돌려주는데, 관례상 정수 0은 프로그램이 정상적으로 종료되었음을 나타낸다.
- 중괄호({})는 여러 코드 문장(statement; §1.4)을 하나의 코드 블록으로 묶는 역할을 한다. 그런 코드 블록을 복합문(compound statement)이라고 부르기도 한다.
- std::cout과 std::endl은 <iostream>에 정의되어 있다. 전자는 화면에 텍스트를 출력하는 데 사용할 수 있는 출력 스트림 객체이다. std::endl는 한 줄을 끝내고 줄을 바꾸는 효과를 낸다. std::endl 대신 특수 문자 \n을 이용해서 줄을 바꾸는 것도 가능하다.
- 연산자 ≪는 임의의 객체를 std::cout 같은 출력 객체에 전달해서 출력 연산을 수행하게 만드는 데 쓰인다. 실제 프로그램 소스 코드에서 이 연산자는 미만 기호 두 개(<<)임을 주의하기 바란다. 이 책에서는 미려한 표현을 위해 한 글자짜리 프랑스어 기메guillemet †† 문자를 사용한다.

† [옮긴이] 출력은 "생명과 우주 만물에 대한 궁극적인 질문의 답은 42"라는 문구인데, 더글러스 애덤스의 소설 은하수를 여행하는 히치하이커를 위한 안내서에 나오는 이야기이다. 참고로 프로그래밍 서적을 비롯해 기술 서적에서 뜬금없이 42라는 수가 나온다면 바로 이 궁극의 답을 뜻할 가능성이 크다.

‡ [옮긴이] '핵심 언어'는 간단히 말해서 #include나 import를 이용해서 외부에서 가져오지 않아도 사용할 수 있는 C++의 기능들을 통칭하는 용어이다.

†† [옮긴이] 프랑스어 기메에 관해서는 위키백과 "기메" 페이지(*https://ko.wikipedia.org/wiki/기메*)를 참고하기 바란다. 참고로 '합자(ligature)' 기능을 지원하는 코딩용 글꼴과 코드 편집기에서는 ≪를 실제로 ≪와 비슷한 모습으로 표시해 준다. 그런 글꼴과 편집기로는 이를테면 D2Coding Ligature(*https://github.com/naver/d2codingfont*)와 VS Code(*https://code.visualstudio.com/*)가 있다. 참고로 이 번역서에 실제로 쓰인 기호들은 프랑스어 기메 문자들이 아니라 그보다 조금 큰 겹화살괄호 ≪, ≫이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

- std::는 그다음에 나오는 이름(형식이나 함수, 객체 등등)이 표준 이름공간(namespace)에 속한다는 뜻이다. 이름공간은 이름(식별자)들을 관리하는 수단인데, 특히 이름들이 충돌하지 않게 하는 데 도움이 된다.
- 이 책의 여러 예제는 std 이름공간에 있는 형식(type)†들을 사용하는데, 간결함을 위해 접두사 std::을 생략할 때가 많다. 그런 예제는 헤더 파일들을 도입한 후에 다음과 같은 이름공간 선언이 있다고 가정한다.

```
using namespace std;
```

  이름공간은 §3.2.1에서 자세히 이야기할 것이다.
- 문자열 상수(좀 더 정확히는 문자열 리터럴literal)는 큰따옴표로 감싼다.
- 표현식(expression; §1.4) 6 * 7은 하나의 구체적인 정수로 평가(evaluation)된 후 std::cout에 전달된다. C++에서 모든 표현식에는 형식이 있다. 프로그램을 작성할 때는 형식을 명시적으로 지정할 때도 있고 컴파일러가 알아서 연역(deduction)하게 둘 때도 있다. 6과 7은 int 형식의 리터럴literal 상수이므로, 그 둘의 곱도 int이다.

다음으로 넘어가기 전에, 이 작은 예제 프로그램을 여러분의 컴퓨터에서 컴파일하고 실행할 것을 강력히 권한다. 컴파일하고 실행한 후에는 코드를 조금씩 수정하면서 가지고 놀아 보기 바란다. 예를 들어 더 많은 연산을 추가하거나 출력을 더 추가해 보면 좋을 것이다. 그리고 오류 메시지가 나오면 유심히 들여다보기 바란다. 프로그래밍 언어를 배우는 유일한 길은 실제로 사용해 보는 것이다. 다음은 주요 C++ 컴파일러와 IDE의 간단한 사용법이다. 컴파일러나 IDE의 사용법을 이미 아는 독자는 이번 절의 나머지 부분을 건너뛰어도 좋다.

리눅스: 모든 리눅스 배포판은 적어도 GNU C++ 컴파일러(부록 §B.1에서 짧게나마 소개한다)를 제공하며, 배포판 설치시 GNU C++ 컴파일러가 함께 설치되는 경우도 많다. 여기서도 GNU C++이 이미 깔려있다고 가정한다. 앞의 예제 프로그램을 hello42.cpp라는 파일에 저장했다고 할 때, 셸에서 다음을 실행하면 프로그램이 컴파일된다.

† [옮긴이] '타입'이라는 용어도 흔히 쓰이지만, 이 번역서에서는 '자료형'이나 '형변환' 같은 기존 용어와의 잘 연결되며 '추상', '가상', '구체', '추론', '연역' 같은 관련 개념어들과 결이 맞는다는 장점을 지닌 '형식'을 사용한다. 그냥 '형' 대신 '형식'을 사용한 것은, 한 글자짜리 '형'은 동음이의어가 많다는 점과 Visual Studio의 C++ 컴파일러(한글로 된 오류/경고 메시지를 제공하는 거의 유일한 컴파일러이다)에서도 '형식'이 쓰인다는 점을 고려한 것이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

```
g++ hello42.cpp
```

지난 세기의 전통에 따라, g++는 기본적으로 a.out이라는 이름의 이진 실행 파일을 생성한다. 그런데 모든 프로그램이 같은 이름의 실행 파일로 컴파일되면 곤란하다. 다행히, 다음과 같이 -o 플래그를 이용해서 실행 파일 이름을 지정할 수 있다.

```
g++ hello42.cpp -o hello42
```

또한 빌드 도구 make를 이용할 수도 있다. make는 이진 실행 파일 생성을 위한 기본 규칙들을 내장하고 있기 때문에, 그냥 다음을 실행하기만 하면 실행 파일이 만들어진다.

```
make hello42
```

make는 현재 디렉터리에 주어진 이름과 비슷한 소스 코드 파일이 있는지 찾는다. 지금 예에서 make는 hello42.cpp를 발견한다. .cpp가 C++ 소스 코드의 표준 파일 확장자임을 알기 때문에 make는 시스템의 기본 C++ 컴파일러를 실행해서 이 소스 코드를 컴파일한다. 결과적으로 주어진 이름과 동일한 이름의 이진 실행 파일이 생성된다. 이제 다음을 실행하면 그 실행 파일이 실행된다.

```
./hello42
```

이 이진 파일은 다른 어떤 소프트웨어에 의존하지 않는 독립적인 실행 파일이므로, 이 파일을 현재 리눅스 시스템과 호환되는 다른 리눅스 시스템에 복사해서 실행할 수 있다.[1]

Windows: MinGW를 사용하는 경우에는 리눅스에서와 동일한 방식으로 프로그램을 컴파일하고 실행할 수 있다. Visual Studio(이하 VS)를 사용하는 경우에는 먼저 프로젝트를 만들어야 한다. 가장 간단한 방법은 예를 들면 *http://www.cplusplus.com/doc/tutorial/introduction/visualstudio*에 나온 대로 콘솔 앱을 위한 프로젝트 템플릿을 이용하는 것이다. VS의 버전에 따라서는 프로그램 실행이 끝

1 표준 라이브러리는 동적으로 링크될 때가 많다(§7.2.1.4 참고). 그런 경우, 다른 시스템에서 현재 시스템의 것과 호환되는 버전의 동적 링크 라이브러리가 존재해야 실행할 수 있다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

나는 즉시 콘솔 창이 닫혀서 프로그램의 결과를 확인할 겨를이 없을 수 있다.[2] 그런 경우 한 가지 해결책은 프로그램에 `<windows.h>`를 포함시키고 `main`의 끝에 `Sleep(1000);`을 추가해서 1초 동안 기다리게 하는 것인데, 이 방법은 이식성이 없다. C++11 이상에서는 이식성 있는(portable) 방식으로 시간을 지연할 수 있다. 프로그램 시작 부분에 표준 헤더 `<chrono>`와 `<thread>`를 포함시키고, `main`의 끝에 다음 문장을 추가하면 된다.

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

Microsoft는 '커뮤니티 에디션'이라는 이름으로 VS의 무료 버전을 제공한다. 이 무료 버전도 유료 '프로' 에디션들처럼 C++ 표준 언어를 잘 지원한다. 프로 에디션들은 더 많은 개발자 라이브러리들과 도구들을 제공한다는 점이 다를 뿐이다. 이 책은 그런 라이브러리들을 사용하지 않으므로, 커뮤니티 에디션으로도 이 책의 예제들을 실행할 수 있다.

IDE: 이 책의 예제들처럼 짧은 프로그램은 그냥 보통의 텍스트 편집기로도 충분히 만들고 고칠 수 있다. 그렇지만 좀 더 큰 프로젝트라면 **IDE**(integrated development environment; 통합 개발 환경)를 사용하는 것이 바람직하다. IDE들은 이를테면 함수가 정의된 곳이나 쓰이는 곳을 찾거나, 라이브러리 도움말을 편집 창에 직접 표시하거나, 프로젝트 전체에서 이름들을 찾고 바꾸는 등의 편의 기능을 제공한다. 그런 IDE로 KDevelop이 있다. KDevelop은 KDE 공동체가 C++로 작성한 자유(무료) IDE로, 아마도 리눅스에서 가장 효율적인 IDE일 것이다. KDevelop은 `git`, `subversion`, `CMake`와 잘 통합된다. 또 다른 유명한 IDE는 자바로 개발된 이클립스(Eclipse)가 있는데, 속도가 느리다는 평이 많다. 그렇지만 최근에는 C++ 지원을 개선하는 데 많은 노력이 있었으며, 이클립스를 아주 생산적으로 활용하는 개발자들이 많다. Visual Studio는 대단히 견고한 IDE로, Windows에서 개발 생산성이 아주 좋다. 또한, 최신 버전들은 `CMake` 프로젝트 통합도 지원한다.

가장 생산적인 개발 환경을 찾으려면 시간과 실험이 필요하다. 또한, 무엇이 최고의 IDE인지는 개인 또는 팀의 취향에 좌우되는 면이 있다. 그런 만큼 개발 환경은 시간이 지남에 따라 점차 바뀌게 된다.

---

2 VS 2019부터는 콘솔 창이 자동으로 일시 정지된다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1.2 변수

다른 여러 스크립팅 언어와는 달리 C++은 강 형식 언어(strongly typed language)이다. 이는 모든 변수(variable)에 형식이 있으며, 한 번 정해진 형식은 바뀌지 않음을 뜻한다. 변수는 형식 이름 다음에 변수 이름이 오는 형태의 문장으로 선언한다. 변수 이름 다음에 변수의 초기화 구문을 둘 수도 있으며,† 쉼표를 이용해서 변수 이름(그리고 초기화)을 여러 개 나열할 수도 있다.

```
int    i1= 2;              // 탭 정렬은 단지 가독성을 위한 것일 뿐임
int    i2, i3= 5;          // 참고: i2는 초기화되지 않음
float  pi= 3.14159;
double x= -1.5e6;          // -1500000
double y= -1.5e-6;         // -0.0000015
char   c1= 'a', c2= 35;
bool   cmp= i1 < pi,       // -> true
       happy= true;
```

슬래시 두 개(//)는 한 줄 주석(single-line comment)을 나타낸다. 즉, 슬래시 두 개부터 그 줄의 끝까지는 컴파일에서 제외된다. 원칙적으로 한 줄 주석에 관해 알아야 할 것은 이것이 전부이다. 그렇지만 설명이 너무 짧으면 뭔가 중요한 사실을 저자가 빼먹은 것이 아닌가 의심하는 독자가 있을까 해서, §1.9.1에서 한 줄 주석을 좀 더 이야기하겠다.

1.2.1 내장 형식

C++의 가장 기본적인 형식들인 **내장 형식**(intrinsic type)들이 표 1-1에 정리되어 있다. 이들은 핵심 언어의 일부이므로 언제 어디서든 사용할 수 있다.

표의 처음 다섯 행은 정수 형식들을 짧은 것에서 긴 것 순서로,‡ 좀 더 정확히는 길이가 더 짧아지지 않는 순서로 나열한 것이다. 예를 들어 `int`는 적어도 `short`만큼은 길다. 실제로 `int`가 더 긴 환경††이 많지만, 반드시 그런 것은 아니다. 각 형식의 구체적인 길이는 구현(컴파일러)에 따라 다르다. 예를 들어 `int`는

† [옮긴이] 초기화 없는 선언과 구분하기 위해 초기화가 있는 선언을 '정의(definition)'라고 부르기도 한다. 그런 구분법에서 아래의 예는 모두 변수 정의이다. 이 책에서는 꼭 필요한 경우가 아니면 굳이 구분하지 않는다.

‡ [옮긴이] 수치 형식에서 '길이'는 값을 구성하는 비트들의 개수(비트수)를 말한다. 문맥에 따라서는 형식의 크기를 '너비'의 관점에서 서술하기도 하는데, 예를 들어 더 큰 형식의 값을 더 작은 형식의 변수에 넣으면 값이 "좁아진다(narrowed)".

†† [옮긴이] 이 문맥에서 '환경'은 CPU 아키텍처, 컴파일러, 운영체제 등의 조합이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

환경에 따라 16비트일 수도 있고 32비트나 64비트일 수도 있다. 이 다섯 형식 각각에 signed나 unsigned가 붙을 수 있다. char를 제외한 정수 형식들은 기본적으로 부호 있는 형식이므로, 그런 형식들에 signed를 붙이는 것은 아무런 효과도 없다.

정수 형식에 unsigned를 붙이면 음수들이 사라지고 양수들의 개수가 두 배가 된다(그리고 0을 양수도, 음수도 아니라고 간주하면 양수 개수는 두 배 더하기 하나가 되는 셈이다). signed와 unsigned를 '명사'로서의 정수 형식들 앞에 붙일 수 있는 형용사라고 생각하기 바란다. 또한, 정수 형식 없이 signed와 unsigned만 사용해서 변수를 선언할 수도 있는데, 그런 경우에는 기본적으로 int가 된다. short, long, long long도 마찬가지 방식으로 작용하는 형용사들이다.

표 1-1 내장 형식

이름	의미론†
char	문자와 아주 짧은 정수
short	꽤 짧은 정수
int	보통의 정수
long	긴 정수
long long	아주 긴 정수
unsigned	위의 정수들의 부호 없는 버전들
signed	위의 정수들의 부호 있는 버전들
float	단정도(single-precision) 부동소수점 수
double	배정도(double-precision) 부동소수점 수
long double	긴 부동소수점 수
bool	부울 값(참 또는 거짓)

char는 두 가지 용도로 쓸 수 있다. 하나는 문자이고 다른 하나는 아주 짧은 정수이다. 아주 색다른 아키텍처를 제외하면 char는 거의 항상 8비트이다. 따라서 char로 선언된 변수로는 -128에서 127까지의 값들(signed) 또는 0에서 255까지의 값들(unsigned)을 표현할 수 있으며, 정수에 대해 성립하는 모든 수치 연산

† [옮긴이] 다소 현학적인 용어인 '의미론(semantics)'은 간단히 말하면 실행 시점에서의 특징이나 행동 방식을 뜻하며, 지금 문맥에서 더 간단히 말하면 '용도'라고 이해해도 될 것이다. 의미론은 컴파일 시점에서의 특징이나 행동 방식을 말하는 '구문론(syntax)'과 대조된다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

을 그런 값들에 대해 수행할 수 있다. signed나 unsigned가 붙지 않은 char의 부호 여부는 언어의 구현(컴파일러)이 결정한다. 작은 값들을 아주 많이 담은 컨테이너들이 프로그램에 필요한 경우라면 char나 unsigned char를 사용하는 것이 유용하다.

논리적 참, 거짓 값들은 bool로 표현하는 것이 최선이다. 부울 변수는 참에 해당하는 값 true와 거짓에 해당하는 값 false를 담을 수 있다.

표의 나머지 네 행은 길이가 더 짧아지지 않는 순서의 부동소수점 수(floating-point number) 형식들이다. double은 float보다 짧지 않으며, long double은 double보다 짧지 않다. float의 전형적인 크기(길이)는 32비트이고 double은 64비트 long double은 128비트이다.

1.2.2 문자와 문자열

앞에서 언급했듯이 char 형식은 문자(character)를 담는 데 사용할 수 있다.

```
char c= 'f';
```

8비트로 표현할 수 있는 그 어떤 문자라도 char 형식으로 표현할 수 있다. 또한, 문자와 수치를 한 표현식 안에서 함께 사용할 수도 있다. 예를 들어 통상적인 문자 부호화(character encoding) 방식에서 'a' + 7은 'h'가 된다. 그렇지만 코드를 읽는 사람이 혼동해서 쓸데없이 시간을 낭비할 위험이 있으므로, 이런 형태의 연산은 사용하지 말 것을 강력히 권한다.

C++은 char들의 배열로 문자열(string)을 표현하는 기능을 C에서 물려받았다.

```
char name[8]= "Herbert";
```

이 구식 C 문자열은 항상 char 형식의 값 0으로 끝난다. 어떠한 이유로 문자열 끝에 0이 없으면, 구식 C 문자열을 다루는 알고리즘들은 바이트 0이 나올 때까지 계속해서 메모리의 다음 장소들로 나아가게 된다.† C 문자열에 다른 문자열을 덧붙일 때도 큰 위험이 존재한다. 지금 예에서 name에는 여분의 공간이 없으므로, 또 다른 문자들을 추가하면 기존의 어떤 데이터를 덮어쓰게 된다. 이런 구

† [옮긴이] 이는 웹 서버 등에서 발견된/발견될 수많은 버퍼 넘침(buffer overflow) 또는 버퍼 덮어쓰기(buffer overwrite) 취약점의 근본 원인 중 하나이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

식 C 문자열로는 모든 종류의 문자열 연산을 제대로(메모리가 깨지지 않게 또는 긴 문자열이 잘리지 않게) 수행하기가 아주 까다롭다. 따라서 구식 C 문자열은 오직 문자열 리터럴에만 사용할 것을 강력히 권한다.

C++ 컴파일러는 작은따옴표와 큰따옴표를 구분한다. 'a'는 하나의 문자 "a"(형식은 char)이고 "a"는 끝에 0이 있는 하나의 문자 배열이다(형식은 const char[2]).

문자열을 다룰 때는 표준 라이브러리의 클래스 string을 사용하는 것이 훨씬 편하다(이 클래스를 사용하려면 <string> 헤더를 포함시켜야 한다).

```
#include <string>

int main()
{
    std::string name= "Herbert";
}
```

C++ 문자열은 동적 메모리를 사용하며, 메모리를 스스로 관리한다. 따라서 문자열에 텍스트를 추가할 때도 메모리 깨짐이나 문자열 잘림을 걱정할 필요가 없다.

```
name= name + ", our cool anti-hero"; // '허버트'는 잠시 후에 소개한다
```

현재 여러 컴파일러는 짧은 문자열(이를테면 16바이트 이하)을 동적 메모리에 저장하는 대신 string 객체 자체에 직접 포함시키는 방식의 최적화를 사용한다. 이 최적화는 값비싼 메모리 할당 및 해제 연산을 크게 줄일 수 있다.

C++14 컴파일러는 큰따옴표로 감싼 문자열 리터럴을 char 배열로 취급한다. 문자열 리터럴을 string 객체로 취급하게 만들려면 "Herbert"s처럼 s라는 접미사를 붙이면 된다.[3] 이 기능은 C++14에 와서야 생겼다. 그 전에는 string("Herbert") 같은 명시적 변환을 사용해야 했다. C++17에서는 좀 더 가벼운 상수 문자열 뷰가 도입되었는데, 이에 관해서는 §4.3.5에서 이야기하겠다.

3 다른 여러 예제에서처럼 이 예제 역시 프로그램에 using namespace std가 있다고 가정한다. 표준 이름공간 전체 대신 이런 접미사들만 프로그램에 도입하거나, 심지어는 특정 접미사만 선택적으로 도입하는 것도 가능하다. 그렇지만 C++을 배우는 도중에는 그냥 표준 이름공간 전체를 도입하길 권한다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1.2.3 변수 선언

조언

변수는 최대한 늦게 선언하라. 보통은 처음 사용하기 직전에 선언하는 것이 바람직하다. 단, 변수를 초기화할 수 있는 지점 이후이어야 한다.†

앞의 조언을 잘 따르면, 프로그램이 길어져도 코드를 읽기가 수월하다. 또한, 중첩된 범위들과 관련해서 컴파일러가 메모리를 좀 더 효율적으로 활용할 수 있다.

C++11 C++11에서는 변수의 형식을 컴파일러가 연역하게‡ 만들 수 있다.

```
auto i4= i3 + 7;
```

`i4`의 형식은 `i3 + 7`의 형식과 같은 `int`이다. 형식이 자동으로 결정되긴 해도, 일단 결정된 형식은 변하지 않으며, 이후 `i4`에 배정(assignment)된†† 모든 것은 `int`로 변환된다. 이 예에 쓰인 `auto`가 고급 프로그래밍에서 얼마나 유용한지를 차차 보게 될 것이다. 그러나 이번 절에 나오는 것 같은 단순한 변수 선언들에서는 그냥 형식을 명시적으로 선언하는 게 나을 때가 많다. `auto`는 §3.4에서 자세히 논의한다.

1.2.4 상수

구문론적으로 C++의 상수(constant)는 상수성(constancy)이라는 추가적인 특성을 가진 특별한 변수라고 할 수 있다.‡‡

```
const int    ci1= 2;
const int    ci3;              // 오류: 값이 없음
const float  pi= 3.14159;
const char   cc= singlequoteasinglequote;
const bool   cmp= ci1 < pi;
```

† [옮긴이] 예를 들어 `int a = b + 3;` 형태의 선언은 `b`가 이미 초기화된 이후이어야 한다.

‡ [옮긴이] 다른 언어에서는 형식의 '추론(inference)'이라는 용어가 많이 쓰이지만, 정적/강 형식 언어인 C++에서는 거의 예외 없이 연역(deduction)이라는 용어가 쓰인다. 추론과 연역의 차이에 관해서는 "C++의 형식 추론과 연역, 귀납, 귀추"(*https://occamsrazr.net/tt/311*)을 참고하기 바란다.

†† [옮긴이] assignment를 '할당'이라고 부르기도 하지만, 할당은 명시적인 메모리 관리 수단을 제공하는 언어들에서allocation의 번역어로 쓰인다. C++도 그런 언어이므로, 할당과의 구별을 위해 이 번역서에서는 assigment를 '배정'으로 옮긴다.

‡‡ [옮긴이] 이런 맥락에서 상수를 '상수 변수'라고 부르기도 한다. 다소 모순적인 이름이지만("변하지 않는 변하는 수"), 리터럴을 그냥 '상수'라고 부르는 경우에는 상수 변수라는 용어가 유용하다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

상수는 변할 수 없으므로, 선언할 때 반드시 값을 지정해야 한다. 이 예제의 둘째 행은 이 규칙을 지키지 않으며, 컴파일러는 그런 규칙 위반을 절대 허용하지 않는다.

값을 수정하지만 않는다면, 상수는 변수가 쓰이는 곳 어디에서나 사용할 수 있다. 변수와는 달리, 앞의 예제에 나온 것 같은 상수는 그 값이 컴파일 시점(compile time; 컴파일러가 소스 코드를 컴파일하는 도중)에서 이미 알려진다. 이 덕분에 다양한 종류의 최적화가 가능하다. 또한 상수는 형식에 대한 인수(argument)로도 사용할 수 있다(이에 관해서는 §5.1.4에서 다시 이야기한다).

1.2.5 리터럴

`2`나 `3.14` 같은 리터럴[literal](값 자체)에도 형식이 있다. 간단하게만 말하자면, 정수 리터럴은 그 크기에 따라 `int`나 `long`, `unsigned long`으로 간주된다. 소수점이나 지수가 있는 수(이를테면 `3e12` $\equiv 3 \cdot 10^{12}$)는 `double`로 간주된다.

그 밖의 수치 형식의 리터럴은 다음 표에 나온 접미사를 이용해서 표기한다.

리터럴	형식
`2`	`int`
`2u`	`unsigned`
`2l`	`long`
`2ul`	`unsigned long`
`2.0`	`double`
`2.0f`	`float`
`2.0l`	`long double`

사실 리터럴의 형식을 굳이 접미사를 이용해서 명시적으로 지정할 필요가 없을 때가 많다. 내장 수치 형식들 사이의 암묵적 변환(소위 **코어션**[coercion]) 덕분에, 대부분의 경우에는 프로그래머가 기대한 형식이 적용된다.

그래도 리터럴의 형식에 신경을 써야 한다. 주된 이유는 다음 네 가지이다.

가용성: 표준 라이브러리는 복소수를 위한 형식을 제공한다. 사용자†는 이 형식

† [옮긴이] 이 책에서 별 다른 수식이 없는 '사용자'는 프로그래머를 뜻한다. 일반적인 의미의 사용자는 '앱 사용자'나 '프로그램 사용자', '최종 사용자' 등으로 표기하겠다. 단, 과학 및 공학 소프트웨어 패키지에서 프로그래머로서의 사용자와 최종 사용자의 경계가 명확하지는 않다는 점도 기억하기 바란다. 최종 사용자가 미리 만들어진 기능들로 작업을 진행하다가 필요하면 직접 코드를 작성해서 기능을 확장하거나 특별한 작업을 수행할 수 있는 환경을 제공하는 소프트웨어 패키지들이 있다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

의 객체를 생성할 때 실수부와 허수부의 형식을 명시적으로 지정할 수 있다.

```
std::complex<float> z(1.3, 2.4), z2;
```

안타깝게도 복소수에 대한 연산들은 복소수 형식 자체와 바탕 실수 형식들 사이에서만 정의된다(그리고 이 연산들에서는 인수들이 암묵적으로 변환되지 않는다)[4]. 그래서 복소수 객체 z에 float를 곱할 수는 있어도 int나 double을 곱할 수는 없다.

```
z2= 2 * z;       // 오류: int * complex<float> 연산은 없음
z2= 2.0 * z;     // 오류: double * complex<float> 연산은 없음
z2= 2.0f * z;    // OK:  float * complex<float>
```

중의성: 함수가 서로 다른 인수 형식들에 대해 중복적재된(§1.5.4) 경우, 0 같은 인수는 다수의 수치 형식으로 해석할 수 있어서 중의성(ambiguity) 문제가 발생한다. 그러나 0u처럼 접미사로 한정된 인수는 특정한 하나의 형식에 대응되므로 중의성이 없다.

정확성: long double과 관련해서 정확성(accuracy) 문제가 발생할 수 있다. 한정되지 않은 실수 리터럴은 double이므로, long double 변수에 그런 리터럴을 배정하면 유효숫자들이 소실될 수 있다.

```
long double third1= 0.3333333333333333333;   // 유효숫자들이 소실될 수 있음
long double third2= 0.3333333333333333333l;  // 정확함
```

십진수가 아닌 수: 숫자 0으로 시작하는 정수 리터럴은 8진수로 해석된다.

```
int o1= 042;          // int o1= 34;
int o2= 084;          // 오류! 8진수에는 8이나 9가 없음
```

16진수 리터럴은 0x나 0X로 시작한다.

```
int h1= 0x42;         // int h1= 66;
int h2= 0xfa;         // int h2= 250;
```

C++14 C++14에서는 이진수 리터럴이 도입되었다. 접두사는 0b이나 0B이다.

```
int b1= 0b11111010;  // int b1= 250;
```

4 혼합 산술을 구현하는 것이 불가능하지는 않다. [19]에 그러한 예가 나온다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

긴 리터럴의 가독성을 개선하기 위해, C++14부터는 숫자들 사이에 어포스트로피를 삽입할 수 있게 되었다.

C++14

```
long                d=   6'546'687'616'861'129l;
unsigned long       ulx= 0x139'ae3b'2ab0'94f3;
int                 b=   0b101'1001'0011'1010'1101'1010'0001;
const long double pi=  3.141'592'653'589'793'238'462l;
```

C++17

그리고 C++17부터는 16진 부동소수점 리터럴도 사용할 수 있다.

```
float  f1= 0x10.1p0f; // 16.0625
double d2= 0x1ffp10;  // 523264
```

지수(exponent)를 지정하기 위해 문자 p가 도입되었다. 지수가 0이라도 생략하면 안 된다. 첫 예의 p0이 그러한 예이다. 접두사 f 때문에 f1은 값 $16^1 + 16^{-1} = 16.0625$를 담은 float 변수이다. 이러한 리터럴에는 세 가지 밑(base; 진법의 기수)이 관여한다. 우선, 유사 가수(pseudo-mantissa) 자체는 0x로 시작하므로 16진수이다. 그러나 p 다음의 지수는 십진수이다. 그리고 그 지수는 2의 거듭제곱에 쓰인다. 예를 들어 d2는 $511 \times 2^{10} = 523264$이다. 16진수 리터럴은 처음에는 난해해 보이지만, 이진 부동소수점 값을 반올림 오차 없이 선언하는 데 요긴하다.

접미사 없는 문자열 리터럴은 char의 배열이 된다.

```
char s1[]= "Old C style"; // 사용하지 않는 게 좋다
```

그렇지만 이런 배열은 다루기가 아주 불편하므로, '진짜' C++ 문자열, 즉 <string>에 있는 string 형식의 객체를 사용하는 것이 낫다. string 객체는 다음과 같이 문자열 리터럴로 직접 생성할 수 있다.

```
#include <string>
std::string s2= "In C++ better like this";
```

아주 긴 텍스트는 여러 개의 문자열 리터럴로 분할해서 지정할 수 있다.

```
std::string s3= "This is a very long and clumsy text "
                "that is too long for one line.";
```

C++14

s2와 s3 둘 다 형식이 string이지만, const char[] 형식의 리터럴로 초기화되었음을 주목하자. 지금은 이 점이 문제가 되지 않지만, 컴파일러가 형식을 연역하

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

는 경우에는 문제가 될 수 있다. C++14부터는 접미사 s를 이용해서 string 형식의 리터럴을 직접 지정할 수 있다.

```
f("I'm not a string");     // const char[] 형식의 리터럴
f("I'm really a string"s); // string 형식의 리터럴
```

이전 예제들처럼 이 예제도 이름공간 std가 현재 범위에 도입되었다고 가정한다. 표준 이름공간 전체를 도입하고 싶지 않다면 다음과 같은 using 문들을 이용해서 특정한 하위 이름공간들만 도입하면 된다.

```
using namespace std::literals;
using namespace std::string_literals;
using namespace std::literals::string_literals;
```

리터럴에 관한 좀 더 자세한 사항은 이를테면 [62, §6.2]를 보라. 사용자 정의 리터럴은 이 책의 §2.3.6에서 이야기한다.

C++11 1.2.6 좁아지지 않는 초기화

long 변수를 다음과 같이 아주 긴 정수로 초기화한다고 하자.

```
long l2= 1234567890123;
```

long이 64비트인 환경(대부분의 64비트 플랫폼이 그렇다)에서는 이 문장은 잘 컴파일되고 정확하게 작동한다. 그러나 long이 32비트인 환경(컴파일 시 -m32 플래그를 지정하면 이런 환경을 흉내 낼 수 있다†)에서는 우변의 리터럴이 너무 길다. 그래도 여전히 컴파일되긴 하지만, 리터럴과는 다른 값(선행(leading) 값들이 잘린)이 변수에 배정된다.

C++11은 값이 **좁아지는**(narrowed) 문제를 미연에 방지하는, 즉 데이터의 소실이 일어날 것 같으면 오류를 발생하는 초기화 방식을 도입했다. **중괄호 초기화**(braced initialization)라고도 부르는 **균일 초기화**(uniform initialization)가 그것이다. 균일 초기화는 §2.3.4에서 자세히 이야기할 것이므로 여기서는 간단하게만 소개한다. 핵심은, 중괄호로 감싼 값에는 좁아지는 변환(narrowing conversion; 또는 좁히기 변환)이 허용되지 않는다는 것이다.

```
long l= {1234567890123};
```

† [옮긴이] 특별한 언급이 없는 한, 컴파일러 오류 메시지나 플래그는 g++를 기준으로 한다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

이 변수 선언문에 대해 컴파일러는 현재 아키텍처에서 변수 l이 우변의 값을 온전하게 담을 수 있는지 점검한다. 중괄호를 사용할 때는 다음처럼 등호를 생략할 수 있다.

```
long l{1234567890123};
```

중괄호 초기화를 사용하면 컴파일러가 좁아지는 변환을 방지해 주기 때문에 초기화 과정에서 값의 정밀도가 소실되는 일을 피할 수 있다. int를 부동소수점 수로 초기화할 때도 데이터가 소실되는데, 중괄호 초기화는 그러한 소실도 방지해 준다.

```
int i1= 3.14;          // 좁아지지만 컴파일은 성공(손실을 감수함)
int i1n= {3.14};       // 좁아지는 오류: 소수부가 소실됨
```

이 예에서 보듯이 부동소수점 값을 중괄호로 감싸면 암묵적인 정수 변환에 의해 소수부가 사라지는 문제를 컴파일러가 검출해 준다. 비슷하게, 음수를 부호 없는 변수나 상수에 배정할 때도 보통의 초기화 구문은 통과되지만 중괄호 초기화 구문은 컴파일 오류를 발생한다.

```
unsigned u2= -3;       // 좁아지지만 컴파일은 성공(손실을 감수함)
unsigned u2n= {-3};    // 좁아지는 오류: 음수들이 사라짐
```

앞의 예제들에서는 리터럴을 초기화에 사용했다. 컴파일러는 주어진 리터럴을 해당 형식의 변수가 담을 수 있는지 점검한다. 다음도 그런 예이다.

```
float f1= {3.14};      // OK
```

사실 3.14는 그 어떤 이진 부동소수점 형식으로도 절대적으로 정확하게 표현할 수 없다. 그래도 컴파일러는 3.14에 가장 가까운 부동소수점 수를 f1에 설정한다. 그런데 float 변수를 double 변수로(리터럴이 아니라) 초기화할 때는 모든 가능한 double 값이 손실 없이 float로 변환되는지 고려해야 한다.

```
double d;
...
float f2= {d};         // 좁아지는 오류
```

두 형식 사이에서 좁아지는 오류가 양방향으로 발생할 수도 있음을 주의하자.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

```
unsigned u3= {3};
int      i2= {2};

unsigned u4= {i2};   // 좁아지는 오류: 음수들이 사라짐
int      i3= {u3};   // 좁아지는 오류: 너무 큰 값들이 있음
```

signed int와 unsigned int는 같은 크기의 형식들이지만, 한 형식의 값 중에는 다른 형식으로 표현할 수 없는 것들이 있다.

1.2.7 범위

범위(scope)는 (정적이 아닌) 변수와 상수의 수명(lifetime)과 가시성(visibility)을 결정하며, 프로그램의 구조를 확립하는 데 기여한다.

1.2.7.1 전역 정의

프로그램에서 사용할 모든 변수는 그것이 쓰이기 전의 어떤 지점에서 해당 형식과 함께 선언되어야 한다. 변수는 전역 범위(global scope)에서 선언할 수도 있고 지역 범위(local scope)에서 선언할 수도 있다. 전역 변수(전역 범위의 변수)는 모든 함수의 바깥에서 선언된다. 일단 선언된 전역 변수는 코드의 어디에서나(함수 안에서도) 사용할 수 있다. 어디에서나 사용할 수 있으므로 아주 편할 것 같지만, 프로그램이 커지면 전역 변수들이 언제 어디서 변경되는지 추적하고 관리하기가 대단히 어렵고 힘들어진다. 그러다 보면, 코드를 조금만 변경했는데도 그 여파가 프로그램 전체에 퍼져서 오류가 양산되는 사태가 벌어질 수 있다.

조언

전역 변수는 사용하지 말라.

전역 변수를 사용하면 언젠가는 후회할 일이 생긴다. 전역 변수는 프로그램의 어디에서나 접근할 수 있으므로, 전역 변수가 언제 어디서 어떻게 변경되는지를 추적하기란 대단히 지겨운 일이다.

다음과 같은 전역 상수는 괜찮다.

```
const double pi= 3.14159265358979323846264338327950288419716939;
```

상수는 변경할 수 없으므로 부작용이 없다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1.2.7.2 지역 정의

지역 변수는 함수의 본문(body) 안에서 선언된다. 지역 변수의 가시성/가용성은 그 선언을 포함한 { } 블록에만 한정된다. 좀 더 정확히는, 한 변수의 범위는 선언문이 있는 지점에서 시작해서 그 선언문을 포함한 블록을 닫는 중괄호(})에서 끝난다.

한 예로, 다음은 π 값을 담은 상수를 main 함수 안에서 선언한 예이다.

```
int main ()
{
    const double pi= 3.14159265358979323846264338327950288419716939;
    std::cout ≪ "pi is " ≪ pi ≪ ".\n";
}
```

이 상수 변수 pi는 main 함수 안에서만 존재한다. 다음 예처럼 함수 안에 또 다른 블록을 정의할 수도 있다(일반화하자면, 한 블록 안에 다른 블록을 얼마든지 여러 번 정의할 수 있다).

```
int main ()
{
    {
        const double pi= 3.14159265358979323846264338327950288419716939;
    }
    std::cout ≪ "pi is " ≪ pi ≪ ".\n"; // 오류: pi가 현재 범위에 없음
}
```

이 예에서 pi의 정의는 함수 안의 블록 안으로만 한정되므로, 블록 바깥에 있는 함수의 문장에서 pi에 접근하는 것은 오류이다. 컴파일러는 다음과 같은 오류 메시지를 출력한다.

```
error: 'pi' was not declared in this scope
```

이런 오류를 **범위 벗어남**(out of scope) 오류라고 부른다.

1.2.7.3 이름 가리기(숨기기)

바깥 범위에 있는 변수와 같은 이름의 변수를 안쪽 범위에서 선언하면 어떻게 될까? 그런 경우 바깥쪽 범위의 변수는 더 이상 보이지 않는다. 이를 가리켜 안쪽 변수가 같은 이름의 바깥쪽 변수를 "가렸다(hide)" 또는 바깥쪽 변수가 안쪽

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

변수 때문에 "숨겨졌다(hidden)"라고 말한다(여러 컴파일러는 이런 이름 가리기에 대해 경고 메시지를 출력한다). 다음 예를 보자.

```
int main ()
{
    int a= 5;               // a#1 선언
    {
        a= 3;               // a#1 배정, a#2는 아직 선언되지 않음
        int a;              // a#2 선언
        a= 8;               // a#2 배정, a#1은 숨겨짐
        {
            a= 7;           // a#2 배정
        }
    }                       // a#2 범위의 끝
    a= 11;                  // a#1 배정(a#2는 범위에서 벗어남)

    return 0;
}
```

이러한 이름 가리기 때문에, 변수의 수명과 변수의 가시성을 구분할 필요가 있다. 예를 들어 a#1은 선언 지점에서 main 함수의 끝까지 살아있지만, a#2가 선언된 지점부터 a#2의 선언을 담은 블록의 끝까지는 보이지 않는다. 정리하자면, 변수의 가시 범위는 변수의 수명 범위에서 변수가 숨겨진 부분을 뺀 것이다. 그리고 당연하겠지만, 한 범위에서 같은 이름의 변수를 두 번 이상 선언하는 것은 오류이다.

범위의 장점은, 어떤 변수를 선언할 때 같은 이름의 변수가 범위 밖 어딘가에서 이미 정의되어 있는지 걱정할 필요가 없다는 것이다. 이미 정의되어 있어도 그냥 숨겨질 뿐 새 변수와 충돌하지 않는다.[5] 단, 숨겨진 바깥 범위 변수에는 현재 범위에서 접근할 수 없다. 이 문제를 그냥 변수 이름을 조금 다르게 지어서 해결할 수도 있지만, 궁극적인 해결책은 아니다. 범위 중첩과 접근성 문제는 이름 공간을 이용해서 해결하는 것이 나은데, 이에 관해서는 §3.2.1에서 이야기한다.

정적 변수, 즉 static으로 선언된 변수는 이 규칙의 예외이다. 정적 변수는 실행의 마지막 순간까지 살아있지만 해당 범위 안에서만 보인다. 지금 정적 변수를 소개하면 오히려 학습에 혼란만 생길 것이므로, 정적 변수에 관한 논의는 §A.2.1로 미루겠다.

5 매크로는 그렇지 않음을 기억하기 바란다. C에서 물려 받은 무모한 기능인 매크로는 C++ 언어의 모든 구조와 신뢰성을 무너뜨리므로 반드시 피해야 한다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1.3 연산자

C++에는 다양한 연산자(operator)가 내장되어 있다. C++ 내장 연산자들은 다음과 같이 여러 범주로 나뉜다.

- 계산:
  - 산술: ++, +, *, %, …
  - 부울:
    - 비교: <=, !=, …
    - 논리: &&, ||
  - 비트 단위: ~, ≪, ≫, &, ^, |
- 배정: =, +=, …
- 실행 흐름: () (함수 호출), ?:, ,
- 메모리 관리: new, delete
- 접근: ., ->, [ ], *, …
- 형식 처리: dynamic_cast, typeid, sizeof, alignof, …
- 오류 처리: throw

이번 절에서는 이 연산자들을 개괄적으로 소개한다. 연산자 중에는 C++의 다른 기능과 연관해서 설명하는 것이 더 나은 것들이 있다. 예를 들어 범위 해소 연산자 ::는 이름공간과 함께 설명하는 것이 제일 좋다. 대부분의 연산자는 사용자 정의 형식에 대해 중복적재할 수 있다. 즉, 여러분이 만든 형식의 객체가 표현식에 쓰일 때 그 객체에 적용되는 연산을 여러분이 직접 정의할 수 있다.

이번 절의 끝에는 연산자 우선순위를 깔끔하게 정리한 표가 나온다(표 1-8). 그 표를 복사해서 여러분의 모니터 옆에 붙여 놓으면 도움이 될 것이다. 실제로 그런 표를 모니터 옆에 붙여 놓는 사람이 많으며, C++의 연산자 우선순위를 완벽하게 외우고 다니는 사람은 아주 드물다. 또한, 연산자 우선순위가 잘 기억이 나지 않을 때는 주저 없이 괄호를 추가하길 권한다. 기억이 나는 경우에도, 코드를 읽는 다른 프로그래머에게 도움이 될 것 같다면 괄호를 사용하는 것이 좋다. 더 나아가서, 컴파일러 옵션에 따라서는 컴파일러가 괄호를 더 추가하라고 권하기도 한다(프로그래머가 연산자 우선순위에 익숙치 않은 것 같다고 판단하고는). §C.2에 C++의 모든 연산자를 간략한 설명과 함께 요약한 표가 나온다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/

1.3.1 산술 연산자

표 1-2는 C++이 제공하는 산술 연산자(arithmetic operator)들을 우선순위별로 묶은 것이다(우선순위가 높은 그룹부터 나열했다). 그럼 이들을 차례로 살펴보자.

표 1-2 산술 연산자

연산자	표현식
후위 증가	x++
후위 감소	x--
전위 증가	++x
전위 감소	--x
단항 플러스	+x
단항 마이너스	-x
곱하기	x * y
나누기	x / y
나머지	x % y
더하기	x + y
빼기	x - y

처음 넷은 변수를 증가(increment)하거나 감소(decrement)하는 연산자들이다. 이들은 변수에 1을 더하거나 뺀다. 이 연산자들은 변수에만 적용할 수 있으며, 리터럴이나 상수, 임시 객체(temporary; 연산의 결과로 만들어지는, 이름이 붙지 않은 객체)에는 적용할 수 없다.

```
int i= 3;
i++;              // i는 이제 4
const int j= 5;
j++;              // 오류: j는 상수
(3 + 5)++;        // 오류: (3 + 5)는 임시 객체
```

좀 더 정확히 말하면, 증가 연산과 감소 연산은 수정이 가능하고 주소 지정이 가능한(addressable) 뭔가만 적용된다. 주소 지정이 가능한 데이터 항목을 C++에서는 **왼값**(lvalue)이라고† 부른다(왼값의 좀 더 공식적인 정의가 부록 C의 정의

† [옮긴이] §C.1에 나오듯이 원래 lvalue은 등호(배정 연산자)의 좌변에 올 수 있는 값을 뜻하는 left value를 줄인 용어이다. 그래서 lvalue를 '좌측값'이나 '왼쪽 값'으로 옮기기도 한다. 그러나 역시 §C.1에 나오듯이 "좌변에 올 수 있는 값"은 더 이상 C++의 lvalue를 온전하게 정의하지 못한다. lvalue의 l이 정확히 left는 아니지만 left와 아예 무관한 것도 아니라는(적어도 역사적으로) 점에서, 이 번역서에서는 '왼쪽 값'에서 '쪽'을 생략한 '왼값'을 사용한다. rvalue/오른값도 마찬가지 조어법을 따른 것이다.

미리보기용 PDF 입니다. 구매: http://ebook.insightbook.co.kr/