Hanbit eBook

Realtime 30

BACK TO THE BASIC

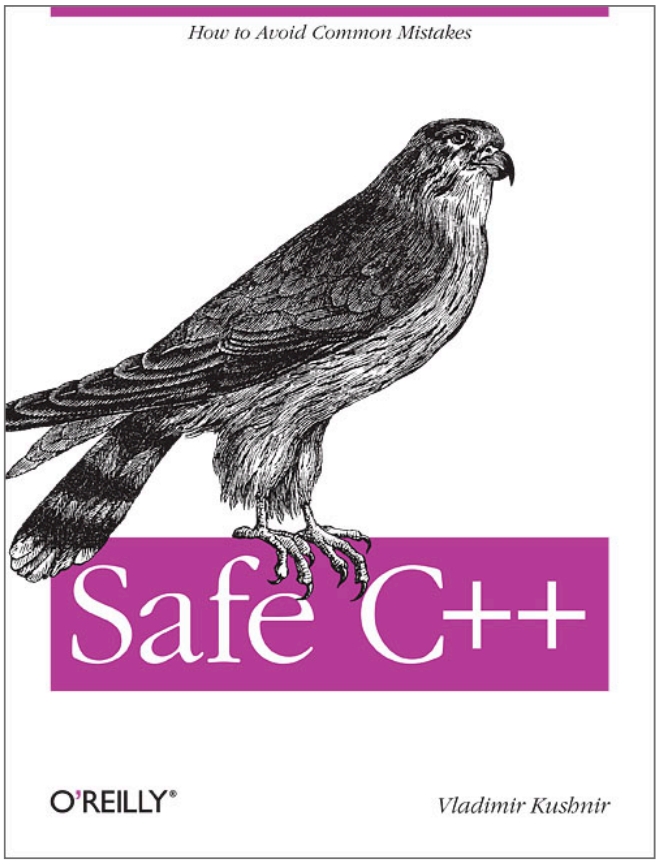
C++ 버그 헌팅

버그를 예방하는 11가지 코딩 습관

Safe C++

블라디미르 쿠스퀴니르 지음 / 정원천 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.



이 도서는 O'REILLY의
Safe C++의
번역서입니다.

BACK TO THE BASIC

C++ 버그 헌팅

버그를 예방하는 11가지 코딩 습관

BACK TO THE BASIC C++ 버그 헌팅 버그를 예방하는 11가지 코딩 습관

초판발행 2013년 5월 31일

지은이 블라디미르 쿠스퀴니르 / **옮긴이** 정원천 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-635-7 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 이중민 / **편집** 김연숙
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 박진희
마케팅 박상용, 박주훈, 정민하

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanb.co.kr / **이메일** ask@hanb.co.kr

Published by HANBIT Media, Inc. Printed in Korea
Copyright ⓒ 2013 HANBIT Media, Inc.
Authorized Korean translation of the English edition of ***Safe C++, ISBN 9781449320935***
ⓒ 2012 Vladimir Kushnir. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
이 책의 저작권은 오라일리 사와 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다. **지금 하지 않으면 할 수 없는 일이 있습니다.**
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

지은이_ **블라디미르 쿠스퀴니르**

블라디미르 쿠스퀴니르는 USSR 과학 아카데미의 고체물리연구소에서 물리학 박사 학위를 취득했다. 그 이후에는 실험 물리학자로 일했으며, 노스웨스턴대학과 아르곤 국립연구소에 근무하는 동안은 포트란, C, C++ 등을 업무에 사용해왔다. 또한 월스트리트로 근무지를 옮긴 후에는 '재무 분석'에 관한 업무를 주로 담당했다. 프로그래밍을 이용한 계산에 관해 이야기하는 것과 이를 빠르게 최적화하는 일에 많은 관심을 갖고 있다(가끔은 자릿수 계산에 흥미를 두기도 한다). 현재 아내 다리아와 코네티컷에 거주하며 C++ 프로그래밍을 하지 않는 한가한 시간에는 재즈 음악과 수중 사진 촬영을 즐긴다.

옮긴이_ **정원천**

현재 ㈜켐에쎈에서 소프트웨어 개발을 책임지고 있다. 소프트웨어 개발 전반에 관심이 많은 낭만 개발자로 2007년부터 C# 윈도우 애플리케이션 개발에 몰두하고 있으며, 어떻게 하면 좋은 구조를 가진 품질 좋은 소프트웨어를 개발할 수 있을까에 대해서 고민한다. 최근 분산 아키텍처에서의 빅데이터 관리에도 열정을 쏟고 있다.

저자 서문

영리한 독자라면 이 책의 원서 제목인 『Safe C++』가 이상하다고 생각할지 모른다. C++가 뭔가 안전하지 않다는 뜻인가? 그렇다!!

C++ 프로그래머는 할당된 배열의 범위를 벗어나는 접근, 초기화하지 않은 메모리 읽기, 메모리 할당 해제 잊어버리기 등 온갖 종류의 실수를 범하기 쉽다. 다시 말해, C++ 프로그래밍을 하는 동안 실수할 여지가 엄청나게 많다는 뜻이다. 이런 실수들은 프로그램이 갑자기 멈추거나 이상한 결과를 나타내거나 '예기치 않은 작동unpredictable behavior'이 발생할 때까지는 알 수 없다. 이런 이유 때문에 C++가 태생적으로 안전하지 않다는 것이다.

이 책에서는 C++ 프로그래머가 공통적으로 범하는 실수를 다룬 후에, 어떻게 하면 그런 실수를 피할 수 있는지 알려준다. C++ 커뮤니티는 몇 년 동안 다양하고 좋은 프로그래밍 사례를 상당수 비축해두었다. 이 책에서 필자는 이런 커뮤니티에서 다룬 다양한 방법들을 수집해서 필요한 부분을 더하거나 덜어냈다. 필자는 이런 규칙들이 모여서 하나의 버그 해결 전략이 되었으면 하는 바람이다.

'Hello, World'보다 복잡한 프로그램이라면 항상 에러(익숙하게 '버그'라고 말하는)를 가질 수밖에 없다. 프로그래밍할 때의 큰 과제는 어떻게 하면 프로그램을 중단하거나 느려지지 않게 하면서 버그 숫자를 줄일 수 있을까다.

우선 누가 버그를 잡을 것인가부터 생각해보자. 여기 소프트웨어 프로그램의 사이클에 참여하는 4명의 참가자가 있다.

1. 프로그래머
2. 컴파일러(유닉스/리눅스는 g++, 윈도우는 마이크로소프트 Visual Studio, Mac OS X는 Xcode)
3. 애플리케이션 실행 코드
4. 프로그램 사용자

물론 우리는 사용자가 버그를 만나지 않거나 버그의 존재를 눈치채지 못하기를 바란다. 이제 1번부터 3번 참가자가 남았다.

그럼 1번 참가자인 프로그래머에 관해 이야기해보자. 사용자처럼 프로그래머도 사람이므로 피곤할 수도 있고, 졸릴 수도 있고, 배고플 수도 있고, 동료의 요청이나 가족의 전화 혹은 자동차를 수리하느라 집중하지 못할 수도 있다. 즉, 프로그래머도 사람이므로 실수로 버그를 만들 수 있다는 뜻이다.

그림 P-1 4명의 참가자(버그가 있는 버전)

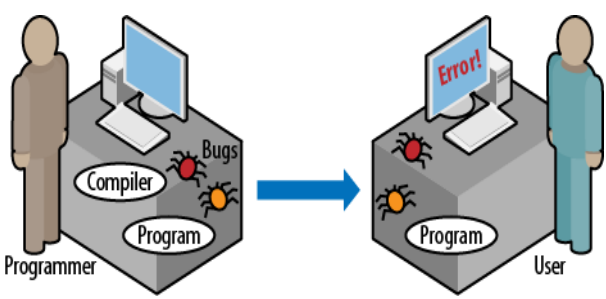
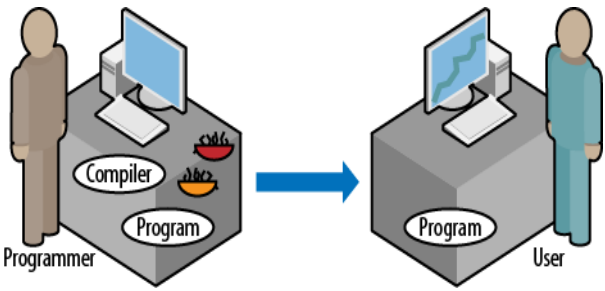


그림 P-2 4명의 참가자(버그가 없거나/적은 버전)



이번에는 2번과 3번 참가자에 관해 이야기해보자. 2번과 3번(컴파일러와 실행 코드)에는 몇 가지 장점이 있다. 피곤해하지도 않고, 졸려 하지도 않고, 우울해하거나 지치지도 않고, 회의에 참석하지도 않고, 휴가나 점심시간 등의 휴식 시간을 가지지도 않는다. 오직 명령을 제대로 실행할 뿐이고 일도 매우 잘한다.

따라서 참가자(한쪽은 프로그래머, 다른 한쪽은 컴파일러와 프로그램)의 특징을 고려한다면 버그를 줄이기 위해 둘 중 하나를 선택할 수 있다.

선택 1. 프로그래머가 실수하지 않게 만든다. 버그 하나당 보너스에서 10달러를 삭감한다고 하거나 생산성을 개선하라는 등으로 압박할 수 있다. 예를 들어 "메모리를 할당했으면 반드시 해제하세요!!"라고 말하는 것이다.

선택 2. 최고의 주의력과 집중력을 가진 프로그래머라 할지라도 버그를 만들 수 있다는 가정 아래 전체 프로그래밍 과정과 테스팅을 구조화한다. 프로그래머에게 차라리 "A를 하고 나면 항상 B를 하세요"라고 말하라. 이렇게 하면 버그 대부분은 그림 P-2처럼 사용자가 애플리케이션을 사용하기 전에 컴파일과 코드 실행 단계에서 잡힌다.

C++ 코드를 작성할 때는 항상 다음 세 가지 목적을 기억해야 한다.

1. 프로그램은 작성한 목적대로 동작해야 한다. 예를 들면 매월 은행 계좌의 입출금 상태 계산하기, 음악 재생하기, 비디오 편집하기 등이 있다.
2. 프로그램은 사람이 읽기 편해야 한다. 즉, 소스 코드는 컴파일이 전부가 아니라 사람이 읽을 수 있도록 작성해야 한다.
3. 프로그램은 자가 진단을 해야 한다. 즉, 자기가 가진 버그를 찾을 수 있어야 한다.

실제 프로그래밍할 때는 이 세 가지를 순서대로 고려해야 한다. 첫 번째는 당연하고, 두 번째는 일부 사람에게 해당하며, 세 번째가 이 책의 주제다. 버그를 직접 잡으려 하지 말고 컴파일러와 실행 코드가 잡도록 하는 것이다. 컴파일러와 실행 코드에게 궂은일을 시키고, 그 시간에 휴식을 취하거나 알고리즘과 설계-즉, 재미있는 일-에 집중할 수 있을 것이다.

집필을 마치며

블라디미르 쿠스니르

감사의 말

우선 이 책의 진가를 알아보고 이 책을 세상에 선보일 수 있도록 격려해준 오라일리의 마이크 핸드릭손Mike Hendrickson에게 감사의 말을 전한다.

영어가 모국어가 아닌 필자의 첫 번째 책을 편집하느라 많은 고생을 한 편집자 앤디 오람Andy Oram에게도 감사한다. 이 책을 읽기 편하게 편집해줬으며 필자가 편하게 작업할 수 있게 도와줘서 집필 작업을 즐길 수 있게 해준 것에 감사한다. 또한 본문의 디자인 스타일과 가독성을 개선해준 에밀리 퀼Emily Quill에게도 고마움을 전하고 싶다.

이번 기회에 필자에게 프로그래밍을 가르쳐주고 첫 번째 프로그램의 수많은 아이디어를 제공해준 발레리 프레드코브Valery Fradkov 박사에게도 고마움을 전한다.

마이크로소프트 Visual Studio 최신 버전에서 달라진 점을 이해할 수 있게 도와준 아들 미샤Misha에게도 감사한다. 그리고 이 책을 집필하는 동안 도움을 아끼지 않았던 아내 다리아Daria에게 무한한 감사를 보낸다.

역자 서문

이 책은 C++ 코드를 어떻게 하면 보다 견고하게 작성할 수 있을 것인가에 관한 저자의 고민이 담겨 있는 책입니다. 프로그래밍을 해본 사람이라면 누구나 마주쳤을 배열의 인덱스에 관한 잘못된 접근부터 시작해서 조금은 까다로울 수 있는 포인터 처리 부분까지, 실제로 수없이 만나게 되는 버그를 어떻게 하면 줄일 수 있을지에 관해 다루고 있습니다. 또한 그러한 방법들을 라이브러리 형태로 만들어서 예제로 제공함으로써 이 책의 내용을 적용해보고 싶은 독자들이 실제로 적용하기 쉽도록 구성되어 있습니다.

프로그래머라면 한 번쯤은 자신이 가진 노하우를 공유해 같은 주제를 고민하는 사람들과 함께하고 싶다는 생각을 해본 적이 있을 것입니다. 이 책의 저자는 그러한 생각을 생각이 아니라 행동으로 실천했고 결국 책에 소개된 라이브러리까지 만들게 된 것이라 느낍니다. 이 책은 그러한 측면에서 단순히 책의 내용뿐만 아니라 실제 프로그래밍에서 부딪치는 문제에 대한 해법을 어떻게 정리해 나가야 할 것인가에 대한 아이디어 또한 간접적으로 제시해주고 있습니다.

이렇게 좋은 책을 번역할 기회를 제공해 준 한빛미디어에 감사하며, 이 책이 나오기까지 많은 정성을 들여 원고를 손봐준 한빛미디어의 이중민 대리에게도 감사의 마음을 전합니다. 그리고 번역하는 작업 동안 집안일에 소홀했음에도 넓은 마음으로 이해해준 사랑하는 아내에게도 감사의 말을 전하고 싶습니다.

번역을 마치며

정원천

대상 독자

초급 **초중급** 중급 중고급 고급

이 책은 C++ 프로그래밍 경험이 있는 독자를 대상으로 한다. C++ 초급자에게는 적합하지 않다. 또한 독자가 C++의 생성자, 복사 생성자, 연산자 할당, 파괴자, 연산자 오버로딩, 가상 함수, 예외 처리 등의 문법에 익숙해야 한다. 즉, 초보자와 중급자 중간 수준의 C++ 프로그래머에게 적당하다.

이 책의 구성

1부에서는 다음 세 가지 질문을 중심으로 풀어나간다.

1장, C++ 버그가 어디에서 발생하는지를 살펴본다(힌트: 전부 다 같은 종류다).

2장, 왜 컴파일할 때 버그를 잡는 게 좋은지 살펴본다. 2장 뒷부분은 이를 어떻게 하는지 설명한다.

3장, 실행 시 발견되는 버그는 어떻게 다뤄야 하는지를 배운다. 에러를 어떻게 잡는지 시연해보고, 가능한 한 모든 새너티 체크(즉, 에러를 진단하는 코드)를 작성한다. 실제로 이미 이 작업은 마친 상태다. 예제 파일에는 새너티 체크sanity check를 하는 소스 코드들이 포함되어 있다. 이 소스 코드들은 프로그래머가 많은 작업을 하지 않고도 어디서, 무슨 일이 왜 일어났는지에 관한 정보를 제공해준다.

2부에서는 각각 다른 종류의 에러를 다룬다. 이런 에러(버그)를 만들 수 있는 규칙들을 공식화하고, 그럴 수 없을 때는 최소한 에러를 잡기 편하게 만든다.

3부에서는 2부에서 소개한 'Safe C++' 라이브러리의 모든 규칙과 소스 코드를 적용해 어떻게 버그를 가장 효율적으로 잡을지에 관한 전략을 세워본다. 또한 어떻게 프로그램을 '디버그할 수 있게' 만들지도 이야기한다. 프로그램을 작성하는 목적 중 하나는 디버그하기 쉽게 만들어야 한다는 것이다. 이 책은 컴파일러, 실행 코드에 에러 처리를 추가하는 방법을 보여준다. 이렇게 디버거 친화적으로 작성한 소스 코드와 디버거가 함께 작동할 때 위력을 발휘한다.

이제 실제 버그를 해결할 준비가 끝났다. 2부에서는 일반적인 형태의 C++ 버그를 다루고, 각 전략을 공식화하거나 잡기 어려운 에러를 실행할 때 쉽게 잡을 수 있도록 하는 규칙들을 만들고 각 규칙의 장단점과 제한사항을 살펴본다고 했다. 실제로

각 장의 마지막에는 이런 규칙을 짧은 공식 형태로 마무리해서 독자들이 복잡한 내용을 건너뛰고 마지막 부분을 살펴보면 알 수 있도록 했다. 특히 17장은 각 장의 모든 규칙을 요약해서 다시 설명했다.

여기서 독자는 " 'A를 할 때 B를 잊지 말라' 대신 'A를 할 때 규칙 C를 따르라'는 말이 왜 더 좋은 걸까? 버그를 제거하는 다른 방법이 있지 않을까?"와 같은 의문을 가질 수도 있을 것이다. 좋은 질문이다. 우선 메모리 해제 같은 일부 문제는 프로그래밍 언어 자체에서 해결할 수 있다. 그리고 실제로 이는 자바나 C#에서는 이미 해결한 문제다. 하지만 이 책은 어떤 이유에선가 이미 레거시 코드가 있거나 매우 엄격한 성능 요구 사항이 있는 프로그램을 C++로 만들어야 하는 상황이라고 가정한다.

이런 규칙을 따르는 일이 왜 전통적인 '잊지 말라'는 말보다 좋은가는, 많은 경우 다음과 같은 장점이 있기 때문이다.

- 과거 규칙: "여기에 메모리를 할당했으면 해제해야 하는 위치 스무 군데를 잊지 말고 확인하시오. 이 함수에 return 구문을 추가하면 정리 코드를 추가해야 하는 일을 잊지 마시오"
- 새로운 형식: "메모리를 할당했다면 스마트 포인터를 할당한 후에 긴장을 풀고 포인터 문제는 잊어버리시오"

필자는 독자 역시 두 번째 방법이 더 간단하고 신뢰할 만하다고 생각할 것으로 본다. 물론 프로그래머가 메모리를 할당한 다음에 스마트 포인터를 사용할 거라고 100% 장담할 수는 없지만 과거 규칙보다는 이 방법이 더 기억하기 쉽다.

이 책에서는 멀티스레딩을 다루지 않는다. 정확히 말하면 메모리 누수 부분에서 멀티스레딩을 간단하게 언급하지만 그게 전부다. 멀티스레딩은 엄청나게 복잡할 뿐만 아니라 프로그래머가 확인하기도 어렵고, 버그 재현도 잘 안 되며, 실수한 부분을 찾는 것도 어려워서 훨씬 더 두꺼운 책에서 다뤄야 할 주제다. 또한 이 책에서 제시하는 방법이 유일한 방법이라는 건 아니다. 수많은 프로그래머가 열정적으로 토론한 다른 대안이 바람직한 방법일 수도 있다. C++ 코드를 작성하는 데는 수없이 다양한 방법들이 있기 때문이다.

하지만 필자가 말하고 싶은 내용은 다음과 같다.

- 이 책에서 제시하는 규칙과 방향(자신의 규칙도 추가할 수 있다)을 따른다면 여러분은 더 빠른 시간 안에 코드를 작성하게 될 것이다.
- 테스팅을 시작하고 수 분에서 수 시간 안에 전부는 아니더라도 에러 대부분을 잡아내서 에러 때문에 받는 스트레스를 상당 부분 줄일 수 있을 것이다.
- 테스팅을 끝냈을 때는 프로그램에 어떤 종류의 버그도 없을 것으로 자신할 수 있다. 여러분은 이미 새너티 체크를 추가했고, 모든 소스 코드가 새너티 체크를 통과했기 때문이다.

마지막으로 "실행 코드의 효율성은 어떠한가?"라는 의문이 들 수 있다. 물론 이렇게 버그를 찾다 보면 효율성이 떨어질까 걱정될 수 있다. 하지만 3부 '버그 해결 즐기기: 테스팅부터 생산 단계를 위한 디버깅까지'에서 코드 작성의 효율성을 높이는 방안을 설명한다.

예제 파일

- https://github.com/vladimir-kushnir/SafeCPlusPlus

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

차례

3부 버그 해결 즐기기: 테스팅부터 생산 단계를 위한 디버깅까지

1부

C++ 버그 해결 전략

여기에서는 C++ 프로그램에서 발생하는 에러의 종류를 구분한다. 테스팅 이전에 컴파일 단계에서 에러를 잡는 것의 중요함을 인식시켜주고, 뒷부분에서 설명하는 버그를 잡거나 미리 방지하는 특별한 방법을 적용할 때 기억해야 할 기본 원리를 제시한다.

1 | C++ 버그는 어디에서 발생하는가?

C++는 특별하다. 실제로 모든 프로그래밍 언어는 아이디어, 문법 요소, 키워드 등을 기존 언어에서 빌려왔지만, C++는 C와는 완전히 다른 언어다. 사실 C++의 창시자인 비얀 스트롭스트럽Bjarne Stroustrup은 원래 자신이 만든 새로운 언어를 '클래스가 있는 C'라고 불렀다. 이는 과학 연구나 상거래 등 어떤 목적으로 사용하든 간에 기존 C 코드가 있다면 객체 지향 언어로 변경하기 위해 아무런 일을 하지 않아도 됨을 의미한다. 즉, 그저 새로운 C++ 컴파일러를 설치하면 이전 C 코드는 모두 정상적으로 작동한다는 뜻이다.

그런데 컴파일러 설치만으로 C++로의 전환이 완료되었다고 생각해서는 안 된다. 실제 C++로 작성한 코드는 C 코드와는 많이 다르다. 그래서 기존 C 코드를 C++에 맞춰 일부를 변경해야 하는 경우도 있다. 즉, 기존 C 코드를 컴파일하고 실행하면서 C++로 작성하는 새로운 코드는 조금씩 추가해야 한다. 그러면서 C와 C++ 코드를 여러분이 원하는 만큼 완전한 C++로 변경해 나가면 된다. 사실 C++의 이런 계층화한 디자인은 C++에 관한 독창적인 홍보 방법이기도 했다.

그런데 여기에는 시사점이 있다. C 문법 전체가 새로운 언어(C++)의 기본이 되면서 프로그래밍 언어의 구조에 문제가 생긴 것이다. 원래 C는 1969년부터 1973년까지 벨 연구소Bell Labs의 데니스 리치Dennis Ritchie가 유닉스 운영체제를 개발하려고 만들었다. 개발 목표는 고급 프로그래밍 언어(컴퓨터 명령을 어셈블러로 작성하지 않음)의 강력함과 효율성을 조합하는 것이었다. 즉, 컴파일한 코드는 가능한 한 빠른 속도로 실행되어야 한다는 뜻이다. 또한 C의 원칙 중 하나는 (사용자가) 사용하지 않는 기능에 어떤 비용도 지불하지 않아야 한다는 점이었다. 그래서 C는 컴파일한 코드의 효율성을 위해서라면 프로그래머가 명시적으로 요청하지 않은 어떤 것

도 수행하지 않는다. 그 결과, 속도는 빨라졌지만 편리하지는 않았다. 그리고 몇 가지 문제를 일으켰다.

우선 프로그래머가 배열을 생성한 후에도 배열 범위를 넘어서는 인덱스를 이용해서 원소에 접근할 수 있었다. C에서 자주 사용하는 포인터 연산은 계산할 수 있는 값을 메모리 주소로 사용함으로써, 원래 프로그램이 사용하려던 곳이 아닌 다른 메모리에 접근하는 문제가 발생했다(실제로 이 두 문제는 문법만 다를 뿐 같은 종류의 문제다).

따라서 프로그래머는 런타임 시 calloc()나 malloc() 함수를 이용해서 메모리를 할당할 수 있었고 free() 함수를 사용해서 해제해야 하는 책임이 뒤따랐다. 그러나 프로그래머가 메모리 해제를 잊거나 한 번 이상 해제함으로써 치명적인 결과를 불러올 수도 있었다.

이런 문제를 해결하는 세부 사항은 2부에서 다룬다. 기억해야 할 것은 C++가 C가 가진 효율성에 관한 철학을 계승했기 때문에 C의 이런 문제점까지 함께 가져왔다는 점이다. 그래서 버그가 어디서 발생하느냐는 질문을 던지면 일부는 'C에서 왔다'고 말할지도 모른다.

그러나 이걸로 끝이 아니다. C를 계승했기 때문에 발생하는 문제 이외에도, C++ 스스로가 가지는 문제점도 있다. 예를 들면 사람들 대부분이 나쁜 아이디어라고 말하는 friend 함수와 다중 상속이 있다. 그리고 C++는 calloc()나 malloc() 함수 말고도 메모리를 할당하는 자체 연산자인 new 연산자를 가지고 있다. new 연산자는 메모리 할당뿐만 아니라 객체도 생성하기 때문에 생성자를 호출한다. 그리고 C와 마찬가지로 delete 연산자를 사용해 이렇게 할당한 메모리를 해제해야 하는 책임은 프로그래머에게 있다.

C와 비슷한 다음 상황을 살펴보자. 메모리를 할당한 다음 해제했지만 C++에는 두 개의 서로 다른 new 연산자가 있다는 문제를 확인할 수 있다.

```
MyClass* p_object = new MyClass(); // 객체 하나를 생성
MyClass* p_array = new MyClass[number_of_elements]; // 배열을 생성
```

첫 번째 new 연산자는 MyClass 타입의 객체 하나를 생성하며 두 번째 new 연산자는 같은 타입의 객체 배열을 생성한다. 그래서 각각 다른 두 개의 delete 연산자를 사용해야 한다.

```
delete p_object;
delete [] p_array;
```

그리고 물론, '[]bracket 연산자를 가진 new 연산자'를 사용해서 객체를 생성했다면 '[] 연산자를 가진 delete 연산자'를 사용해서 해당 객체들을 해제해야 한다. 따라서 [] 연산자가 있거나 없는 new와 delete 연산자를 섞어서 사용하다 보면 새로운 실수를 할 수도 있다.

이 내용을 잘 이해할 수 없는 독자라면 다음에 설명하는 메모리 힙memory heap을 이해하는 데 어려움을 겪을 수도 있다. 따라서 C++의 버그는 대부분 C를 계승하기 때문에 발생하지만 프로그래머가 치명적인 실수를 할 여지가 있는 새로운 방법을 추가했다는 것만 기억하면 된다. 2부에서 이 내용을 다룰 예정이다.

2 | 언제 버그를 잡아야 하는가?

2.1 왜 버그를 잡기에 가장 좋은 곳이 컴파일러인가?

버그는 컴파일할 때나 런타임 시에 잡을 수 있지만 가능하다면 컴파일할 때 잡는 게 좋다. 여기에는 여러 가지 이유가 있다.

첫째, 컴파일러가 버그를 감지하면 에러가 정확히 어떤 파일의 몇 번째 줄에서 발생했는지를 평범한 영문 메시지로 받을 수 있기 때문이다. 어떤 경우(STL이 포함된 경우 등)에는 컴파일러가 매우 불분명한 에러 메시지를 출력한다. 이때는 컴파일러가 어떤 부분 때문에 에러를 발생시켰는지 알아내는 데 어려움이 있다(그래도 컴파일러는 대부분 어떤 부분이 문제인지를 명확하게 알려준다).

둘째, (마지막 링크를 포함해) 컴파일을 완료했다는 건 프로그램의 모든 코드가 정상이라는 것을 의미하기 때문이다. 컴파일러가 에러나 경고를 표시하지 않는 경우라면 컴파일할 때 프로그램에서 발견할 수 있는 버그가 없다는 것을 100% 확신할 수 있다. 이는 충분히 많은 테스트 코드가 있더라도 런타임 테스트하고는 절대 같을 수 없다. 즉, 아무리 많은 테스트 코드가 있더라도 런타임 시에 실행 가능한 모든 경우의 수에 맞춰 모든 코드를 최소한 한 번 이상 실행한다고 보장하기는 어렵다는 뜻이다.

그리고 만약 모든 경우를 테스트했다고 확신하더라도 아직 부족한 부분이 있다. 똑같은 코드라도 입력값에 따라서는 정상 작동할 수도 있고 실패할 수도 있기 때문이다. 그래서 런타임 테스트 시에는 모든 경우의 수를 완벽히 테스트했다고 확신할 수가 없다.

마지막으로 시간적인 요소도 있다. 코드를 실행하기 전에 컴파일을 하기 때문에 컴파일할 때 에러를 잡으면 시간을 절약할 수 있다. 프로그램에서 어떤 런타임 에러는 나중에 발생한다. 그러므로 에러를 확인하려면 몇 분 또는 몇 시간이 소요될 수도 있다. 게다가 해당 에러는 재현조차 안 될 수도 있고, 실행할 때마다 에러가 발생할 수도 발생하지 않을 수도 있다. 모든 경우를 고려했을 때, 컴파일할 때 에러를 잡기가 가장 쉽다.

2.2 컴파일러는 어떻게 버그를 잡는가?

이제 독자 여러분은 컴파일할 때 에러를 잡는 게 가장 좋은 방법이라는 걸 알게 됐을 것이다. 그러면 어떻게 해야 컴파일할 때 에러를 잡을 수 있을까? 다음 예제를 살펴보자.

우선 Variant 클래스의 사례를 살펴보자. 예전에 엑셀에 사용하는 플러그인을 개발하는 회사가 있었다. MS 엑셀에서 열려 있는 파일을 엑셀의 셀에 불러와서 추가 기능을 제공하는 프로그램이었다. 엑셀의 셀에는 정수형(1), 실수형(3.1415926535), 날짜형(1/1/2000), 문자형(“여긴 잭이 지은 집입니다”) 등 여러 가지 종류의 데이터형을 입력할 수 있다.

이러한 점에 착안해 이 회사는 마치 카멜레온처럼 이런 다양한 종류의 데이터를 가질 수 있는 Variant라는 클래스를 개발했다. Variant 클래스는 다른 Variant 클래스 객체를 포함할 수도 있고, 벡터형의 Variant 클래스 객체들(std::vector<Variant>)을 포함할 수도 있었다. 그리고 엑셀하고만 상호작용하는 게 아니라 내부 코드하고도 상호작용해 작동했다. 다음 함수 표기법을 살펴보자.

```
Variant SomeFunction(const Variant& input);
```

위 코드를 살펴보고 SomeFunction() 함수가 어떤 종류의 데이터를 입력받고, 어떤 종류의 반환값을 가지는지 추측하기는 불가능하다. 예를 들어 함수가 날짜형의 데이터를 입력받는데 사용자가 날짜하고는 전혀 닮지 않은 문자열형의 데이터를 입력하면 이런 경우는 실행 중에만 에러를 발견할 수 있다. 앞에서 얘기했던 것처럼 컴파일할 때 에러를 발견하고 싶어도 이렇게 코드를 작성하면 컴파일러가 변수 자료형의 안정성을 확인할 수가 없다. 이 문제를 어떻게 해결할지는 앞으로 설명하겠지만 간단히 말하면 각 자료형마다 C++ 클래스를 분리해서 사용하면 된다.

앞 예제는 실제 예지만 약간 극단적인 경우이므로 일반적인 경우를 생각해보자. 주식 가격 같은 금융 데이터를 처리한다고 했을 때 이 가격이 어느 시점의 가격이었는지에 관한 시간 정보를 추가하려 한다고 할 때 시간을 어떻게 확인해야 할까? 간단한 해결책은 과거(1970년 1월 1일 같은 특정 기준 시간)로부터 몇 초나 지났는지를 세는 것이다.

이때 이 라이브러리가 32비트 정수값을 사용한다면 32비트 정수형의 최대값은 20억이므로 20억이 넘어가면 음수값이 되어버린다. 그래서 기준 시간에서 68년 뒤인 2038년이 되었을 때 시간을 세는 데 문제가 생긴다.

이게 그 유명한 'Y2K' 문제다. 이 문제를 해결하려면 이 변수를 사용하는 엄청난 수의 파일들을 찾아서 32비트 대신에 64비트를 사용하는 int64 타입으로 변경해야 한다. 그러면 시간이 40억 배나 늘어나므로 아무도 걱정할 필요가 없을 만큼 충분한 시간을 셀 수 있다.

그러나 다른 문제가 남아있다. 프로그래머들이 각각 int64 num_of_seconds, int64 num_of_millisec, int64 num_of_microsec 등을 사용하는 경우다. 이러면 컴파일러는 마이크로세컨드microsecond를 사용하는 시간 변수에 밀리세컨드millisecond 값이 저장되면 알 방법이 없다.

물론 1990년부터 3000년까지의 주식 가격을 분석한다면 이 기간을 만족하는 값만 저장되는지 실행 중에 확인하는 새너티 체크를 추가할 수 있다. 그러나 이런 새너티 체크를 많은 수의 함수에 추가하려면 너무 많은 수작업이 필요하다. 그리고 "20세기 당시의 주식 가격을 확인하려면 어떻게 해야 할까?"라는 문제도 남는다.

2.3 자료형을 다루는 적절한 방법

현재 이런 복잡한 문제는 간단하게 해결할 수 있는 상태다. Time 클래스를 생성하기만 하면 시간 측정을 언제 시작하고 측정해야 하는 단위는 무엇인지(초, 밀리세컨드 등) 등의 세세한 내부 구현 상태를 몰라도 된다.

이렇게 했을 때의 장점은 Time 타입의 시간 말고 다른 자료형을 전달하려고 했을 때 컴파일러가 쉽게 알아챌 수 있다는 것이다. 또한 현재 밀리세컨드 단위로 구현되어 있는 Time 클래스의 정확도를 마이크로세컨드 단위로 높이려고 할 때 다른 코드를 건드리지 않고 Time 클래스 하나만 수정하면 된다는 장점도 있다.

그러면 어떻게 해야 이런 자료형에 관한 에러를 실행 중이 아니라 컴파일할 때 잡아낼 수 있을까? 여기에서는 각각의 자료형에 대한 클래스로 분리하는 일부터 시작할 것이다. 즉, 정수형에는 int, 실수형에는 double, 텍스트에는 std::string, 날짜형에는 Date, 시간형에는 Time을 사용하는 것이다. 물론 다른 자료형의 데이터도 마찬가지다.

그러나 아직 이것만으로는 충분하지 않다. Apple과 Orange라는 두 개의 클래스가 있는데 Orange 클래스 객체를 입력받는 함수가 있다고 가정하자.

```
void DoSomethingWithOrange(const Orange& orange);
```

여기에 다음처럼 실수로 Apple 클래스 객체를 입력할 수도 있다.

```
Apple an_apple(some_inputs);
DoSomethingWithOrange(an_apple);
```

이 코드는 특정 상황에서는 컴파일이 가능하다. 즉, Apple 클래스 객체를 Orange 클래스 객체로 변환할 수 있으면 친절하게도 C++ 컴파일러가 알아서 변환 작업을 처리해준다. 구체적으로 살펴보면 다음 두 경우에 이것이 가능하다.

1. Orange 클래스가 Apple 클래스형의 인자 하나만 받는 생성자를 가지고 있을 경우
2. Apple 클래스가 Orange 클래스 객체로 변환하는 연산자를 가지고 있을 경우

첫 번째 경우는 Orange 클래스가 다음처럼 구성되어 있을 때다.

```
class Orange {
  public:
    Orange(const Apple& apple);
    // 추가 코드 작성
};
```

다음 경우에도 가능하다.

```
class Orange {
  public:
    Orange(const Apple& apple, const Banana* p_banana=0);
    // 추가 코드 작성
};
```

두 번째 예제의 생성자는 두 개의 입력값을 받긴 하지만 인자 하나만 이용해서 호출할 수 있다. 그래서 내부적으로 Apple 클래스 객체를 Orange 클래스 객체로 변환할 수 있다. 인자 하나만 이용해서 호출하는 문제를 해결하는 방법은 위와 같은 생성자에 explicit 키워드를 사용하는 것이다. 이러면 컴파일러가 인자를 (묵시적으로) 변환하는 것을 방지할 수 있기 때문에 프로그래머가 Orange 클래스를 사용해야 하는 곳에 Orange 클래스 객체를 사용하도록 강제할 수 있다.

```
class Orange {
  public:
    explicit Orange(const Apple& apple);
    // 추가 코드 입력
};
```

두 번째 경우에도 다음처럼 선언한다.

```
class Orange {
  public:
    explicit Orange(const Apple& apple, const Banana* p_banana=0);
    // 추가 코드 입력
};
```

이번에는 컴파일러가 Apple 클래스 객체를 Orange 클래스 객체로 변환하는 연산자를 아는 경우를 살펴보자.

```
class Apple {
  public:
    // 생성자와 그외 다른 코드 입력
    operator Orange () const;
};
```

이런 연산자가 존재한다는 것은 프로그래머가 컴파일러에 Apple 클래스 객체를 Orange 클래스 객체로 변환하는 방법을 명시적으로 알려주려 한다는 것이므로 자료형 변환에 문제가 생길 수가 없다. 그래도 생성자 앞에 explicit 키워드를 빼먹다 보면 실수할 수 있기 때문에 가능하면 인자를 하나만 가지는 모든 생성자 앞에 explicit 키워드를 선언하는 것이 좋다.

일반적으로 암시적 형변환은 좋지 않은 생각이므로 앞 예제처럼 Apple 클래스 내부에서 Apple 클래스 객체를 Orange 클래스 객체로 변경하는 방법을 제공하고 싶으면 다음처럼 하면 된다.

```
class Apple {
  public:
    // 생성자와 그외 다른 코드 입력
    Orange AsOrange() const;
};
```

이 경우에 Apple 클래스 객체를 Orange 클래스 객체로 변경하려면 다음과 같이 하면 된다.

```
Apple apple(some_inputs);
DoSomethingWithOrange(apple.AsOrange()); // 명시적 형변환
```

서로 다른 자료형을 혼합하는 방법이 하나 더 있다. enum 타입을 사용하는 것이다. 다음 예제와 같이 요일과 달을 표현하는 두 개의 enum 타입을 정의했다고 가정하자.

```
enum { SUN, MON, TUE, WED, THU, FRI, SAT };
enum { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

상수는 실제로는 정수형(즉, C의 int 타입)이다. 예를 들어 요일을 입력받는 함수가 있다고 하자.

```
void FunctionExpectingDayOfWeek(int day_of_week);
```

다음 코드는 경고 없이 컴파일될 것이다.

```
FunctionExpectingDayOfWeek(JAN);
```

그리고 실행 중에 문제가 발생할 것이다. JAN과 MON 배열 변수가 똑같은 상수 1을 가지기 때문이다.

이런 버그를 잡으려면 정수형을 생성하는 '전형적인' enum 타입을 사용하지 않고 다른 자료형의 enum 타입을 생성해야 한다.

```
typedef enum { SUN, MON, TUE, WED, THU, FRI, SAT } DayOfWeek;
typedef enum { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }
Month;
```

그러면 함수는 요일을 다음처럼 선언할 것이라고 예상할 수 있다.

```
void FunctionExpectingDayOfWeek(DayOfWeek day_of_week);
```

여기에 다음처럼 Month라고 선언한 자료형 변수 하나를 입력하면

```
FunctionExpectingDayOfWeek(JAN);
```

컴파일 에러가 발생한다.

```
error: cannot convert 'Month' to 'DayOfWeek' for
       argument '1' to 'void
       FunctionExpectingDayOfWeek(DayOfWeek)'
```

이는 원하는 결과다. 그러나 이런 접근법은 불리한 면이 있다. enum 타입의 변수가 정수형 상수를 생성하면 다음처럼 코드를 작성할 수 있다.

```
for(int month = JAN; month <= DEC; ++month)
  cout << "Month = " << month << endl;
```

그런데 다음처럼 Month라는 새로운 자료형을 사용해서 enum 타입의 변수를 사용하면 컴파일되지 않는다.

```
for(Month month = JAN; month <= DEC; ++month)
  cout << "Month = " << month << endl;
```

그래서 enum 타입의 변수값을 루프 안에서 사용하려면 정수형을 사용해야 한다.

물론 모든 규칙에는 예외가 있기 때문에 프로그래머가 특수한 목적에서 암시적 형변환을 하려고 Variant 클래스를 이러한 방식으로 작성했을 수도 있다. 그래도 가능하면 암시적 형변환은 피하는 편이 좋다. 그렇게 해야 컴파일러가 자료형을 확인해서 컴파일할 때 잠재적 에러를 초기에 잡을 수 있다.

이로써 자료형의 안정성을 위해 해볼 수 있는 모든 것을 했다. 불행하게도 bool과 char 타입은 빠졌지만 각 자료형이 가질 수 있는 값이 천문학적으로 많은 것에 비하면 이 두 가지가 차지하는 부분은 작은 편이다. 예를 들어 주식 가격으로 double

타입을 사용하면 값이 0과 10000 사이에 있을 것으로 예상할 수 있다. 그런데 워런 버핏이 소유한 버크셔 해서웨이의 주식 같은 경우는 이 범위가 적당하지 않다. 이 글을 작성하는 현재 한 주당 가격이 10만 달러가 넘기 때문이다.

double 타입이 10^{308}을 넘으면 음수가 되므로 주식 가격에 사용하기에는 적절하지 않지만, 버크셔 해서웨이 주식조차 double 타입으로 표현할 수 있는 전체 범위(음수를 포함한)를 생각한다면 여전히 작다. 즉, 대부분의 자료형을 사용할 때 자료형의 범위를 넘는 경우는 거의 없기 때문에 큰 문제는 아니긴 하지만 이런 문제는 실행 중에만 발견할 수 있는 에러가 된다.

사실 C에서 발생하는 문제의 대부분은 '범위를 벗어나는 인덱스'를 사용하거나 포인터 연산을 잘못해서 잘못된 메모리에 접근하는 경우다. 이런 문제는 실행 중에만 발견할 수 있다. 그래서 이 책의 남은 부분에서는 런타임 에러를 잡는 방법을 주로 다룬다.

컴파일할 때 에러를 진단하기 위한 이 장의 규칙

- 암시적 형변환 금지: 인자를 하나만 가지는 생성자에는 explicit 키워드를 사용하고 형변환 연산자에는 사용하지 않기
- 자료형에 맞는 클래스 사용하기
- int 타입의 상수에는 enum 타입을 사용하지 말고 해당 enum 타입을 위한 새로운 자료형 생성하기