

쉽게 설명한
C++ 핵심 가이드라인

C++ Core Guidelines Explained

by Rainer Grimm

Authorized translation from the English language edition, entitled C++ Core Guidelines Explained by Rainer Grimm, published by Pearson Education, Inc. Copyright © 2022 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 © 2023

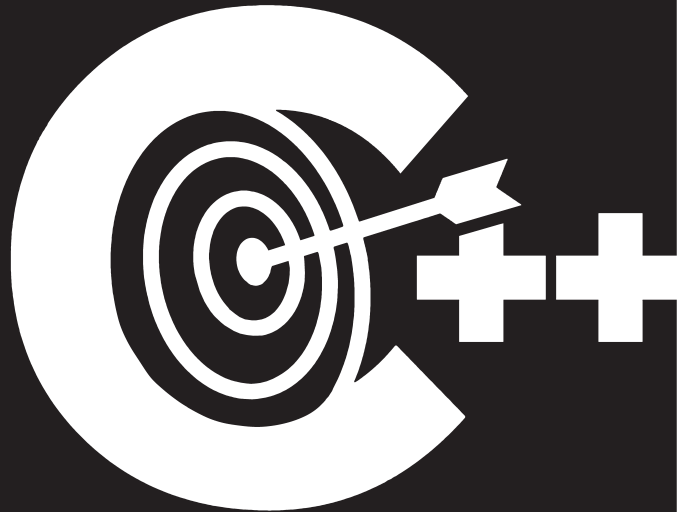
Korean language translation rights arranged with PEARSON EDUCATION, INC. through Agency-One, Seoul, Korea

이 책의 한국어판 저작권은 에이전시 원을 통해 저작권자와의 독점 계약으로 인사이트 출판사에 있습니다. 저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

쉽게 설명한 C++ 핵심 가이드라인

전자책 1쇄 발행 2023년 5월 18일 지은이 라이너 그림 옮긴이 류광 펴낸이 한기성 펴낸곳 (주)도서출판인사이트 편집 정수진
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 978-89-6626-411-7

인사이트
프로그래밍 이야기



쉽게 설명한
C++ 핵심 가이드라인

라이너 그림 지음 | 류광 옮김

인<+>이<+>트

차례

엄선한 핵심 가이드라인 규칙들	ix
옮긴이의 글	xviii
추천사	xxi
서문	xxiii
관례	xxiii
C++ 핵심 가이드라인에 주목한 이유	xxv
판타 레이	xxviii
이 책을 읽는 방법	xxix
감사의 글	xxx

1부 지침들 1

1장 소개	3
1.1 대상 독자	4
1.2 목적	4
1.3 목적이 아닌 것	4
1.4 규칙의 집행	4
1.5 구조	5
1.6 주요 섹션	5
2장 철학	7
3장 인터페이스	17
3.1 비const 전역 변수의 저주	18
3.2 해결책으로서의 의존성 주입	21
3.3 좋은 인터페이스 만들기	23

	3.4 관련 규칙들	28
4장	함수	29
	4.1 함수의 정의	30
	4.2 매개변수 전달: 입력과 출력	35
	4.3 매개변수 전달: 소유권 의미론	41
	4.4 값 반환 의미론	45
	4.5 기타 함수	50
	4.6 관련 규칙	56
5장	클래스와 클래스 위계구조	59
	5.1 클래스 일반 규칙	60
	5.2 구체 형식	64
	5.3 생성자, 배정, 소멸자	65
	5.4 클래스 위계구조	106
	5.5 중복적재와 중복적재된 연산자	127
	5.6 공용체	136
	5.7 관련 규칙들	139
6장	열거형	141
	6.1 일반 규칙들	142
	6.2 관련 규칙들	147
7장	자원 관리	149
	7.1 일반 규칙들	150
	7.2 할당과 해제	155
	7.3 스마트 포인터	161
	7.4 관련 규칙들	175
8장	표현식과 문장	177
	8.1 일반 규칙들	178
	8.2 선언	180
	8.3 표현식	199

8.4 문장	213
8.5 산술	218
8.6 관련 규칙들	225
9장 성능	227
9.1 잘못된 최적화	228
9.2 잘못된 가정	228
9.3 최적화의 활성화	233
9.4 관련 규칙들	244
10장 동시성	245
10.1 일반 지침	246
10.2 동시성	260
10.3 병렬성	283
10.4 메시지 전달	286
10.5 무잠금 프로그래밍	291
10.6 관련 규칙들	294
11장 오류 처리	297
11.1 오류 처리의 설계	299
11.2 구현	301
11.3 예외를 던질 수 없으면	307
11.4 관련 규칙들	311
12장 상수와 불변성	313
12.1 const 적용	314
12.2 constexpr 적용	318
13장 템플릿과 일반적 프로그래밍	321
13.1 템플릿의 용도	323
13.2 템플릿 인터페이스	326
13.3 템플릿 정의	341
13.4 템플릿과 위계구조	352

13.5	가변 인수 템플릿	353
13.6	메타프로그래밍	358
13.7	기타 규칙들	384
13.8	관련 규칙들	395
14장	C 스타일 프로그래밍	397
14.1	전체 소스 코드가 있는 경우	398
14.2	전체 소스 코드가 없는 경우	400
15장	소스 파일	405
15.1	인터페이스 파일과 구현 파일	406
15.2	이름공간	413
16장	표준 라이브러리	419
16.1	컨테이너	420
16.2	문자열	427
16.3	입력과 출력	433
16.4	관련 규칙들	441
2부	지원 섹션들	443
<hr/>		
17장	구조적 개념들	445
18장	비규칙과 미신	449
19장	프로파일	459
19.1	형식 안전성	460
19.2	경계 안전성	461
19.3	수명 안전성	462
20장	GSL: 가이드라인 지원 라이브러리	463
20.1	뷰	464

20.2 소유권 포인터	465
20.3 단언	465
20.4 유틸리티	466
3부 부록	469
<hr/>	
부록 A C++ 핵심 가이드라인의 집행	471
A.1 Visual Studio	472
A.2 clang-tidy	475
부록 B 콘셉츠	477
부록 C 계약	481
찾아보기	484

엄선한 C++ 핵심 가이드라인 규칙들

P.1 생각을 코드로 직접 표현하라.	8
P.2 ISO 표준 C++로 코드를 작성하라.	9
P.3 의도를 표현하라.	9
P.4 이상적으로, 프로그램은 정적 형식 안전성을 갖추어야 한다.	10
P.5 실행 시점 점검보다는 컴파일 시점 점검을 선호하라.	11
P.6 컴파일 시점에서 점검할 수 없는 것은 실행 시점에서 점검할 수 있어야 한다.	11
P.7 실행 시점 오류는 일찍 잡아라.	12
P.8 자원이 새지 않게 하라.	12
P.9 시간이나 공간을 낭비하지 말라.	12
P.10 변경 가능 데이터보다 변경 불가 데이터를 선호하라.	13
P.11 지지분한 프로그램 요소들을 코드 전체에 흩어놓지 말고 한 곳에 캡슐화하라.	14
P.12 지원 도구들을 적절히 활용하라.	14
P.13 지원 라이브러리를 적절히 활용하라.	14
I.2 비const 전역 변수를 피하라.	18
I.3 단일체(싱글턴)를 피하라.	19
I.13 배열을 단일 포인터로 전달하지 말라.	25
I.27 안정적인 라이브러리 ABI를 원한다면 PImpl 관용구를 고려하라.	27
F.4 컴파일 시점에서 평가될 수 있는 함수는 constexpr로 선언하라.	31
F.6 함수가 예외를 던지지 않는다면 noexcept로 선언하라.	33
F.8 순수 함수를 선호하라.	34
F.15 단순하고 통상적인 방식의 정보 전달을 선호하라.	35
F.16 '입력' 매개변수는 복사 비용이 낮은 형식이면 값으로, 그 밖의 형식이면 참조로 전달하라.	37
F.19 '전달' 매개변수는 TP&&로 받고 std::forward로만 전달하라.	37
F.17 '입출력' 매개변수는 비const 참조로 전달하라.	39
F.20 함수가 어떤 값을 '출력'할 때는 출력 매개변수보다는 반환값을 선호하라.	39
F.21 여러 개의 '출력' 값을 돌려줄 때는 구조체나 튜플을 돌려주는 방식을 선호하라.	40

F.42 위치를 나타내려면(그리고 그럴 때만) T*를 반환하라. 45

F.44 복사가 바람직하지 않으며 '아무 객체도 돌려주지 않음'이 필요하지 않다면 T&를 반환하라. 46

F.45 T&&를 반환하지 말라. 48

F.48 std::move(지역객체)를 반환하지 말라. 48

F.46 main()의 반환 형식은 int이다. 49

F.50 람다는 보통의 함수로는 할 수 없는 일(지역 변수 갈무리, 지역 함수 정의)에 사용하라. 50

F.52 람다 안에서 지역적으로 쓰이는(알고리즘으로 전달하는 것도 포함해서) 데이터는 가능한 한 참조로 갈무리하라 52

F.53 지역적으로 사용할 것이 아닌(반환하거나, 힙에 저장하거나, 다른 스레드로 전달하는 등) 데이터는 참조로 갈무리하지 않는 것이 좋다. 52

F.51 선택할 수 있다면 중복적재보다는 기본 인수를 선호하라. 53

F.55 va_arg 인수는 사용하지 말라. 54

C.1 관련된 데이터를 구조체(structure; struct 또는 class)로 조직화하라. 60

C.2 불변식이 있는 클래스에는 class를 적용하라. 데이터 멤버들이 독립적으로 변할 수 있으면 struct를 적용하라. 61

C.3 인터페이스와 구현의 구분을 클래스를 이용해서 표현하라. 62

C.4 클래스의 표현에 직접 접근해야 하는 함수만 멤버 함수로 만들라. 62

C.5 클래스의 보조 함수들은 그 클래스와 같은 이름공간에 배치하라. 63

C.7 하나의 문장에서 class나 enum을 정의하고 즉시 해당 형식의 변수를 선언하지 말라. 63

C.8 public이 아닌 멤버가 있다면 struct 대신 class를 사용하라. 64

C.9 멤버들을 최소한으로 노출하라. 64

C.10 클래스 위계구조보다 구체 형식들을 선호하라. 65

C.11 구체 형식은 정규 형식으로 작성하라. 65

C.20 가능하면 기본 연산은 아예 정의하지 말라. 66

C.21 기본 연산을 하나라도 정의하거나 =delete로 삭제했다면, 다른 모든 기본 연산도 정의하거나 삭제하라. 67

C.22 기본 연산들의 일관성을 유지하라. 70

C.41 생성자는 완전하게 초기화된 객체를 생성해야 한다. 73

C.42 생성자에서 유효한 객체를 생성할 수 없으면 예외를 던져라. 74

C.43 복사 가능(값 형식) 클래스에는 반드시 기본 생성자가 있어야 한다. 75

C.45 데이터 멤버들을 초기화하기만 하는 기본 생성자는 정의하지 말고, 대신 멤버 초기화 구문을 사용하라. 76

C.46 기본적으로, 단일 인수 생성자는 <code>explicit</code> 로 선언하라.	78
C.47 멤버 변수들을 멤버 선언 순으로 정의하고 초기화하라.	80
C.48 생성자에서 멤버를 상수로 초기화할 때 멤버 초기화 구문보다는 클래스 안 초기화 구문을 선호하라.	81
C.49 생성자 안의 배정보다는 멤버 초기화 구문을 선호하라.	82
C.51 한 클래스의 모든 생성자에 공통인 작업들은 위임 생성자로 표현하라.	83
C.52 더 이상의 명시적 초기화가 필요하지 않은 파생 클래스에 기반 클래스의 생성자를 도입할 때는 상속 생성자를 사용하라.	84
C.67 다형적 클래스는 복사를 금지해야 한다.	88
C.30 객체가 파괴될 때 어떤 작업을 명시적으로 수행해야 하는 클래스에는 소멸자를 정의하라.	91
C.31 클래스가 획득한 모든 자원은 반드시 클래스의 소멸자에서 해제해야 한다.	92
C.32 클래스에 원시 포인터(T*)나 참조(T&) 멤버가 있으면, 그 멤버가 자원을 소유하는지 따져 봐야 한다.	92
C.33 클래스에 다른 객체를 소유하는 포인터 멤버가 있으면 소멸자를 정의하라.	93
C.35 기반 클래스의 소멸자는 <code>public</code> 과 <code>virtual</code> 의 조합이거나 <code>protected</code> 와 <code>virtual</code> 의 조합이어야 한다.	94
C.80 만일 기본 의미론이 적용됨을 명시적으로 밝혀야 한다면 <code>=default</code> 를 사용하라.	97
C.81 기본 행동 방식을 비활성화하려면 <code>=delete</code> 를 사용하라 (다른 여러 가능한 방법 대신).	98
C.82 생성자와 소멸자에서는 가상 함수를 호출하지 말라.	99
C.86 <code>==</code> 를 피연산자 형식들에 대칭으로 작성하고 <code>noexcept</code> 로 선언하라.	103
C.87 기반 클래스들에 대한 <code>==</code> 를 조심하라.	105
C.120 클래스 위계구조는 위계적인 구조가 자연스러운 개념들을 표현하는 용도로(만) 사용하라.	107
C.121 인터페이스로 사용할 기반 클래스는 추상 클래스로 만들어라.	109
C.122 인터페이스와 구현을 완전하게 분리해야 할 때는 추상 클래스를 인터페이스로 사용하라.	110
C.126 보통의 경우 추상 클래스에는 생성자가 필요 없다.	111
C.128 가상 함수에는 <code>virtual</code> , <code>override</code> , <code>final</code> 중 딱 하나만 지정해야 한다.	111
C.130 다형적 클래스에 깊은 복사를 적용할 때는 복사 생성/배정보다 가상 <code>clone</code> 함수를 선호하라.	112
C.132 이유 없이 함수를 <code>virtual</code> 로 만들지는 말라.	114
C.131 자명한 조회 함수와 설정 함수를 피하라.	115
C.133 <code>protected</code> 데이터를 피하라.	116

C,134 모든 비const 데이터 멤버의 접근 수준을 동일하게 유지하라.-----116

C,129 클래스 위계구조를 설계할 때 구현 상속과 인터페이스 상속을 구분하라.-----117

C,135 서로 구별되는 여러 인터페이스는 다중 상속을 이용해서 표현하라.-----120

C,138 파생 클래스와 그 기반 클래스들을 위한 중복적재 집합을 작성할 때
using 선언을 사용하라.-----121

C,140 가상 함수와 재정의 함수의 기본 인수들이 서로 달라서는 안 된다.-----122

C,146 클래스 위계구조 안에서의 이동이 불가피하면 dynamic_cast를 사용하라.-----124

C,147 요구된 클래스를 찾지 못했을 때 오류가 발생해야 한다면
dynamic_cast를 참조 형식에 적용하라.-----125

C,148 요구된 클래스를 찾지 못한 것을 유효한 선택지로 간주할 수 있다면 dynamic_
cast를 포인터 형식에 적용하라.-----125

C,152 파생 클래스 객체들의 배열을 가리키는 포인터를 기반 클래스 객체를
가리키는 포인터에 지정하면 절대로 안 된다.-----126

C,167 연산자는 통상적인 의미의 연산을 수행하는 데 사용하라.-----128

C,161 대칭적인 연산자에는 비멤버 함수를 사용하라.-----128

C,164 암묵적인 변환 연산자를 피하라.-----132

C,162 대체로 동등한 연산들을 중복적재하라.-----134

C,163 대체로 동등한 연산들만 중복적재하라.-----134

C,168 중복적재된 연산자는 해당 피연산자의 이름공간 안에서 정의하라.-----135

C,180 union은 메모리를 절약하는 데 사용하라.-----136

C,181 ‘혈벗은’ union을 피하라.-----137

C,182 태그된 공용체는 익명 union을 이용해서 구현하라.-----138

Enum,1 매크로보다 열거형을 선호하라.-----143

Enum,2 열거형은 연관된, 이름 붙은 상수들의 집합을 표현하는 데 사용하라.-----143

Enum,3 ‘평범한’ enum보다 enum class를 선호하라.-----143

Enum,5 열거자에 ALL_CAPS 형태의 이름을 사용하지 말라.-----144

Enum,6 이름 없는 열거형을 피하라.-----145

Enum,7 열거형의 바탕 형식은 꼭 필요할 때만 명시하라.-----145

Enum,8 열거자의 값은 꼭 필요할 때만 명시하라.-----146

R,1 자원 핸들과 RAII를 이용해서 자원을 자동으로 관리하라.-----151

R,3 원시 포인터(T*)는 비소유(non-owing)이다.-----153

R,4 원시 참조(T&)는 비소유이다.-----153

R,5 범위 있는 객체를 선호하라; 불필요하게 힙을 할당하지 말라.-----153

R,10 malloc()과 free()를 피하라.-----156

R.11 new와 delete의 명시적인 호출을 피하라.	157
R.12 명시적인 자원 할당의 결과를 즉시 관리자 객체에 전달하라.	157
R.13 하나의 표현식 문장에서 명시적인 자원 할당은 많아야 한 번만 수행하라.	159
R.20 소유권은 unique_ptr나 shared_ptr를 이용해서 표현하라.	161
R.21 소유권을 공유할 필요가 없다면 shared_ptr보다 unique_ptr를 선호하라.	162
R.22 shared_ptr 객체는 make_shared()로 생성하라.	164
R.23 unique_ptr 객체는 make_unique()로 생성하라.	164
R.24 shared_ptr들의 순환 참조 관계를 깨려면 std::weak_ptr를 사용하라.	165
R.30 수명 의미를 명시적으로 나타내고자 할 때만 스마트 포인터를 매개변수로 사용하라.	168
R.37 재명명된 스마트 포인터로 얻은 포인터나 참조를 전달하지 말라.	174
ES.1 표준 라이브러리를 다른 라이브러리나 '손으로 짠 코드'보다 선호하라.	178
ES.2 언어의 기능을 직접 사용하기보다는 적절한 추상을 선호하라.	179
ES.5 범위를 작게 유지하라.	181
ES.6 for 문의 초기화 구문과 조건문 안에서 이름을 선언해서 범위를 최소화하라.	181
ES.7 혼하고 지역 범위에 있는 이름은 짧게, 혼하지 않고 지역 범위가 아닌 이름은 길게 지어라.	182
ES.8 비슷하게 보이는 이름들을 피하라.	182
ES.9 ALL_CAPS 이름을 피하라.	183
ES.10 하나의 선언문에서는 하나의 이름만 선언하라.	183
ES.11 auto를 이용해서 형식 이름의 불필요한 중복을 피하라.	184
ES.12 내포된 범위들에서 이름을 재사용하지 말라.	185
ES.20 객체를 항상 초기화하라.	188
ES.21 필요하기도 전에 변수(또는 상수)를 도입하지는 말라.	189
ES.22 초기화할 값을 갖추기 전에 변수를 선언하지 말라.	189
ES.23 {} 초기화 구문을 선호하라.	190
ES.26 하나의 변수를 서로 무관한 두 가지 용도로 사용하지 말라.	195
ES.28 복잡한 초기화에는, 특히 const 변수의 복잡한 초기화에는 람다를 사용하라.	195
ES.40 복잡한 표현식은 피하라.	199
ES.41 연산자 우선순위에 자신이 없으면 괄호를 사용하라.	200
ES.42 포인터는 간단하고 직관적으로만 사용하라.	200
ES.45 '마법의 상수'를 피하고 기호 상수를 사용하라.	203
ES.55 색인 범위 점검이 필요한 상황을 피하라.	204
ES.47 0이나 NULL 대신 nullptr를 사용하라.	205

ES.61 배열은 delete[]로 삭제하고 배열이 아닌 객체는 delete로 삭제하라.	207
ES.65 유효하지 않은 포인터를 역참조하지 말라.	207
ES.43 평가 순서가 정의되지 않는 표현식을 피하라.	208
ES.44 함수 인수들의 평가 순서에 의존하지 말라.	209
ES.48 형변환을 피하라.	210
ES.49 형변환이 꼭 필요하다면 명명된 형변환을 사용하라.	211
ES.50 const를 강제로 제거하지 말라.	212
ES.78 switch 문에서 암묵적인 실행 지속에 의존하지 말라.	215
ES.79 공통의 사례들은(그리고 그런 사례들만) default로 처리하라.	217
ES.100 부호 있는 산술과 부호 없는 산술을 섞지 말라.	219
ES.101 비트 조작에는 부호 없는 형식을 사용하라.	219
ES.102 산술에는 부호 있는 형식을 사용하라.	220
ES.106 unsigned를 이용해서 음수를 피하려 들지 말라.	220
ES.103 위넘침을 허용하지 말라.	223
ES.104 아래넘침을 허용하지 말라.	223
ES.105 0으로 나누기를 허용하지 말라.	224
Per.7 최적화가 가능하도록 설계하라.	233
Per.10 정적 형식 시스템에 의존하라.	237
Per.11 계산을 실행 시점에서 컴파일 시점으로 이동하라.	238
Per.19 메모리에 예측 가능한 방식으로 접근하라.	240
CP.1 여러분의 코드가 다중 스레드 프로그램의 일부로 실행될 것이라고 가정하라.	246
CP.2 데이터 경쟁을 피하라.	248
CP.3 쓰기 가능 데이터의 명시적인 공유를 최소화하라.	249
CP.4 스레드가 아니라 작업의 관점에서 사고하라.	251
CP.8 volatile을 동기화에 사용하지는 말라.	252
CP.9 가능하면 항상 적절한 도구를 이용해서 동시적 코드의 유효성을 검증하라.	252
CP.20 항상 RAII를 적용하고, lock()/unlock()은 절대로 직접 사용하지 말라.	261
CP.21 여러 개의 뮤텝스를 획득할 때는 std::lock()이나 std::scoped_lock을 사용하라.	262
CP.22 자물쇠를 잠근 상태에서 절대로 미지의 코드(이러테면 콜백 함수)를 호출하지 말라.	264
CP.23 주 스레드에 합류하는(joining) thread를 일종의 범위 있는 컨테이너로 간주하라.	266
CP.24 thread를 전역 컨테이너로 간주하라.	266

CP.25 <code>std::thread</code> 보다 <code>std::jthread</code> 를 선호하라.	268
CP.26 스레드를 분리하지(<code>detach()</code>) 말라.	269
CP.42 조건 없이 대기하지(<code>wait</code>) 말라.	270
CP.31 스레드에서 스레드로 작은 데이터를 넘겨줄 때는 참조나 포인터 대신 값으로 전달하라.	274
CP.32 서로 무관한 <code>thread</code> 들이 소유권을 공유할 때는 <code>shared_ptr</code> 를 사용하라.	274
CP.40 문맥 전환을 최소화하라.	278
CP.41 스레드의 생성과 파괴를 최소화하라.	278
CP.43 임계 영역에서 소비하는 시간을 최소화하라.	280
CP.44 <code>lock_guards</code> 와 <code>unique_locks</code> 에 이름을 붙이는 것을 잊지 말라.	281
CP.100 절대적으로 필요하지 않은 한 무잠금 프로그래밍은 사용하지 말라.	291
CP.101 여러분의 하드웨어/컴파일러 조합을 신뢰하지 말라.	291
CP.102 문헌들을 세심하게 공부하라.	294
E.3 예외는 오류 처리에만 사용하라.	302
E.14 목적에 맞게 설계한 사용자 정의 형식을 예외로 사용하라(내장 형식들 말고).	302
E.15 위계구조에 속한 예외들은 참조로 잡아라.	304
E.13 객체를 직접 소유한 상황에서는 절대로 예외를 던지지 말라.	305
E.30 예외 명세는 사용하지 말라.	306
E.31 <code>catch</code> 절들을 적절한 순서로 배치하라.	307
Con.1 기본적으로 객체를 변경 불가(<code>immutable</code>)로 만들어라.	314
Con.2 기본적으로 멤버 함수에 <code>const</code> 를 적용하라.	315
Con.3 기본적으로 포인터와 참조는 <code>const</code> 로 전달하라.	317
Con.4 생성 이후 값이 변하지 않는 객체를 정의할 때 <code>const</code> 를 적용하라.	318
Con.5 컴파일 시점에서 계산할 수 있는 값에는 <code>constexpr</code> 를 적용하라.	318
T.1 템플릿을 코드의 추상 수준을 높이는 데 사용하라.	323
T.2 다수의 인수 형식들에 적용할 알고리즘을 표현할 때 템플릿을 사용하라.	324
T.3 컨테이너와 구간을 표현할 때 템플릿을 사용하라.	325
T.40 연산을 알고리즘에 전달할 때 함수 객체를 사용하라.	326
T.42 표기를 간단하게 만들고 구현 세부사항을 숨기기 위해 템플릿 별칭을 사용하라.	331
T.43 별칭을 정의할 때 <code>typedef</code> 보다 <code>using</code> 을 선호하라.	332
T.44 클래스 인수 형식을 연역할 때 함수 템플릿을 사용하라(가능한 경우에).	332
T.46 템플릿 인수의 요구조건을 적어도 <code>Regular</code> 또는 <code>SemiRegular</code> 로 지정하라.	334
T.47 가시성이 높지만 제약이 없는 혼한 이름의 템플릿을 피하라.	336

T.48 컴파일러가 콘셉츠를 지원하지 않는다면 <code>enable_if</code> 로 흉내 내라.	340
T.60 템플릿의 문맥 의존성을 최소화하라.	342
T.61 멤버들을 과도하게 매개변수화하지 말라.	342
T.62 비의존적 클래스 템플릿 멤버들을 비템플릿 기반 클래스에 배치하라.	344
T.80 클래스 위계구조를 어수룩하게 템플릿화하지 말라.	352
T.83 멤버 함수 템플릿을 <code>virtual</code> 로 선언하지 말라.	353
T.140 재사용할 수 있는 연산에는 이름을 붙여라.	384
T.141 한 곳에서만 사용할 간단한 함수 객체가 필요할 때는 익명 람다를 사용하라.	387
T.143 무심코 비일반적인 코드를 작성하지 말라.	387
T.144 함수 템플릿은 특수화하지 말라.	390
CPL.1 C보다 C++을 선호하라.	397
CPL.2 C를 꼭 사용해야 한다면 C와 C++의 공통 부분집합을 사용하고, C 코드를 C++로써 컴파일하라.	398
CPL.3 C 인터페이스를 꼭 사용해야 한다면, 그런 인터페이스를 사용하는 호출 코드를 C++로 작성하라.	400
SF.1 프로젝트가 다른 어떤 관례를 따르지 않는 한, 코드 파일의 확장자로는 .cpp를 사용하고 인터페이스 파일의 확장자로는 .h를 사용하라.	406
SF.2 .h 파일에 객체 정의나 비인라인 함수 정의를 담으면 안 된다.	407
SF.5 .cpp 파일은 자신의 인터페이스를 정의하는 .h 파일(들)을 포함해야 한다.	408
SF.8 모든 .h 파일에 <code>#include</code> 가드를 사용하라.	409
SF.9 소스 파일들 사이의 순환 의존관계를 피하라.	410
SF.10 간접적으로 <code>#include</code> 된 이름들의 의존관계를 피하라.	412
SF.11 헤더 파일은 자기 완결적이어야 한다.	413
SF.6 <code>using namespace</code> 지시문을 전이(transition)를 위해서(만), 기반 라이브러리 (<code>std</code> 같은)를 위해서(만), 또는 지역 범위 안에서(만) 사용하라.	413
SF.7 헤더 파일의 전역 범위에서 <code>using namespace</code> 를 사용하지 말라.	416
SF.20 namespace들을 논리적인 구조를 표현하는 데 사용하라.	416
SF.21 헤더에서 이름 없는(익명) 이름공간을 사용하지 말라.	417
SF.22 내부에서만 사용할, 외부로 내보내지 않을 개체들은 모두 이름 없는(익명) 이름공간 안에 포함시켜라.	417
SL.con.1 C 배열보다 STL <code>array</code> 나 <code>vector</code> 를 선호하라.	420
SL.con.2 다른 컨테이너를 사용할 특별한 이유가 없는 한 기본적으로 STL의 <code>vector</code> 를 선호하라.	424
SL.con.3 경계 오류를 피하라.	425

SL, str, 1 문자 순차열을 소유하려면 <code>std::string</code> 을 사용하라.	427
SL, str, 2 문자 순차열을 참조하려면 <code>std::string_view</code> 를 사용하라.	429
SL, str, 4 문자 하나를 참조하려면 <code>char*</code> 를 사용하라.	431
SL, str, 5 반드시 문자를 표현하는 것은 아닌 바이트 값을 참조할 때는 <code>std::byte</code> 를 사용하라.	431
SL, str, 12 표준 라이브러리 문자열로 취급할 문자열 리터럴에는 접미사 <code>s</code> 를 사용하라.	432
SL, io, 1 문자 수준 입력은 꼭 필요할 때만 사용하라.	433
SL, io, 2 읽기 작업 시에는 입력이 잘못된 형태일 수 있음을 항상 고려하라.	434
SL, io, 3 입출력 작업에는 <code>iostream</code> 라이브러리를 선호하라.	435
SL, io, 10 <code>printf</code> 류 함수들을 사용하지 않는다면 <code>ios_base::sync_with_stdio(false)</code> 를 호출하라.	437
SL, io, 50 <code>endl</code> 을 피하라.	438
A, 1 안정된 코드와 덜 안정된 코드를 분리하라.	445
A, 2 잠재적으로 재사용 가능한 부품들을 라이브러리로 표현하라.	446
A, 4 라이브러리들 사이에 순환 의존관계가 없어야 한다.	448
NR, 1 모든 선언을 함수의 최상단에 두려고 고집하지 말라.	449
NR, 2 함수에 반환문을 하나만 두려고 고집하지 말라.	450
NR, 3 예외를 꺼리지 말라.	452
NR, 4 클래스 선언을 개별 소스 파일에 두려고 고집하지 말라.	453
NR, 5 2단계 초기화를 사용하지 말라.	453
NR, 6 아무리 작업들을 제일 끝에 몰아넣고 <code>goto exit</code> 로 넘어가는 방식으로 함수를 작성하지 말라.	456
NR, 7 모든 데이터 멤버를 <code>protected</code> 로 만들지는 말라.	458

웁긴이의 글

한때 시대를 주도하던 C++은 언제부터인가 변화에 적응하지 못한 공룡 취급을 받기도 했습니다. 그러나 다행히도 시기를 크게 놓치지 않고 “C++ 현대화” 작업이 시작되어서 C++11과 C++14, C++17을 거쳐 C++20까지 나왔고, C++23도 정식 발표를 눈앞에 두고 있습니다. 이러한 C++ 현대화의 흐름 속에서 저는 도서 출판 인사이트의 후의 덕분에 2015년의 《이펙티브 모던 C++》에서 2022의 《필요한 것만 골라 배우는 모던 C++》까지 현대적 C++(모던 C++)에 관한 여러 좋은 책을 번역하는 행운을 누렸습니다. 그리고 마치 그 책들에 담긴 지혜와 통찰을 한 권의 책으로 정리할 때가 왔다는 듯이 “C++ 핵심 가이드라인(C++ Core Guidelines)”을 해설한 *C++ Core Guidelines Explained: Best Practices for Modern C++*(Addison-Wesley Professional, 2021)가 세상에 등장했고, 그 책을 부족하나마 제가 번역해서 본서 《쉽게 설명한 C++ 핵심 가이드라인》을 독자 여러분께 선보이게 되었습니다.

C++은 방대하고 복잡한 언어로 알려져 있고 실제로 방대하고 복잡하지만, 그러한 방대함과 복잡함은 본질적으로 현실의 반영일 뿐 그 자체를 단점으로 보는 것은 바람직하지 않겠습니다. 예를 들어 어떤 여객용 제트 비행기의 설계도가 세발자전거의 설계도만큼이나 단순하다면 오히려 걱정스러운 것입니다. 문제는 그런 복잡한 언어를 어떻게 나의 요구에 맞게 다스리고 활용할 것인가인데, 여기에는 지식뿐만 아니라 지혜와 통찰이 크게 작용합니다. 오랜 경험 없이도 그런 지혜와 통찰을 더 많은 C++ 프로그래머가 갖출 수 있게 하려는 노력의 하나가 바로 이 책이 다루는 C++ 핵심 가이드라인입니다.

C++ 생태계의 두 주요 인사인 비야네 스트롭스트롭(C++의 창시자)과 허버스터(C++ 표준 위원회 의장)가 주도하는 오픈소스 프로젝트(<https://github.com/isocpp/CppCoreGuidelines>)인 C++ 핵심 가이드라인은 경험, 지혜, 통찰이 가득한 보물상자와도 같지만, 라이너 그림은 C++ 핵심 가이드라인이 “좀 더 많은 사람이 접하기에 이상적인 형태는 아니”라고 생각했습니다. 그래서 저자는 C++ 핵심

가이드라인의 규칙들을 해설한 글을 써서 자신의 블로그를 비롯한 여러 매체에 발표했고, 급기야는 한 권의 책으로 내게 되었습니다.

이상의 소개에서 짐작하겠지만, 그리고 저자 서문을 읽으면 더욱 확실해지겠지만, 이 책은 C++ 언어를 처음 배우려는 사람이 아니라 현대적 C++을 좀 더 효과적이고 안전하게 활용하고자 하는 사람을 위한 책입니다. 따라서 C++11에서 시작하는 현대적 C++에 관한 기본적인 지식이 필요합니다. C++에 익숙하다고 생각하는 프로그래머라도, 예를 들어 `auto mult = [&val](auto&& x) -> decltype(val) { return x * val; };` 같은 문장(현대적 C++의 특징적인 요소들을 보여주기 위해 작위적으로 만든 것입니다)을 보고 “이게 뭐야? 내가 아는 C++이 아닌데?”라는 생각이 든다면 C++을 새로이 공부할 때가 된 것입니다. 미리 현대적 C++을 어느 정도 체계적으로 공부한 후 이 책으로 돌아올 수도 있고 이 책을 읽다가 모르는 부분이 나오면 그때그때 공부할 수도 있겠는데, 어떤 경우이든 2015년부터 제가 번역한 도서출판 인사이트의 현대적 C++ 관련 번역서들이 도움이 될 것입니다. 제 번역서들은 홈페이지 류광의 번역서 이야기(<https://occamsrazr.net>)의 “번역서 정보” 섹션에서 확인할 수 있습니다.

저자 서문의 ‘판타 레이’ 절에 나오듯이 C++ 핵심 가이드라인은 계속해서 변합니다. 번역하면서 원서에 나온 규칙과 C++ 핵심 가이드라인에 실제로 나온 규칙이 조금 다른 경우를 몇 개 발견했는데, 독자가 저자 서문의 판타 레이 부분을 읽었으리라고 가정하고 별다른 역주 없이(이런 사실 자체를 언급하는 역주를 빼고는) 번역문에 새로운 내용을 반영했음을 알려 드립니다. 그밖에 원서의 문장을 그대로 옮기는 대신 “저자가 쓰려고 했을” 또는 “썼어야 했을” 문장을 짐작해서 옮긴 경우도 있는데, 어쩌면 제가 오해한 것일 수도 있겠습니다. 내용상의 오류와 오역, 오타를 발견하면 다른 독자들을 위해서라도 제게 꼭 알려주시기 바랍니다. 앞서 언급한 제 홈페이지의 번역서 정보 섹션에 이 책을 위한 페이지로 가는 링크가 있습니다.

감사 인사로 옮긴이의 글을 마무리하고자 합니다. 먼저, 현대적 C++ 책들을 꾸준히 내주시는 도서출판 인사이트 한기성 사장님께 감사합니다. C++의 현대화에는 C++ 언어 자체뿐만 아니라 C++ 사용자의 현대화도 포함된다는 점에서, 좋은 책의 출간은 아주 중요한 문제입니다. 그리고 이번에도 번역과 교정 과정을 매끄럽게 진행해 주신 정수진 편집자님과 다소 산만한 원서의 조판과는 달리 차분하면서도 지루하지 않은 조판으로 번역서의 품격을 높여 주신 조판 디자

이녀 최우정님께 감사드립니다. 그 밖에도 이 책의 출간에 많은 분이 기여하셨지만, 일일이 거론하지 못해 죄송할 따름입니다. 마지막으로, 여러 가지 악조건 속에서도 세밀한 교정·교열로 미숙한 원고를 출판에 적합한 원고로 탈바꿈해 준 아내 오현숙에게 감사와 사랑의 마음을 전합니다.

재미있게 읽으시길!

웁긴이 류광

추천사

C++은 수많은 기능을 가진, 대단히 다채롭고 표현력이 큰 언어이다. C++이 그런 언어가 된 것은 당연하다면 당연하다. 범용 프로그래밍 언어가 성공하려면 한 사람의 개발자가 필요로 하는 것보다 더 많은 기능을 제공해야 하며, 언어가 계속 살아남고 진화하다 보면 어떠한 하나의 개념을 표현하는 다양한 관용구들이 축적되기 때문이다. 하지만 그러면 개발자가 선택할 것이 너무 많아질 수 있다. 개발자는 다양한 프로그래밍 스타일과 전문 지식을 적절하게 선택하는 능력을 갖추어야 한다. 또한, 낡고 효과적이지 않은 기법과 프로그래밍 스타일에 묶여서 더 나아가지 못하는 사태를 피할 수 있어야 한다.

C++ 핵심 가이드라인(C++ Core Guidelines)¹은 그런 문제점들을 해결하기 위해 현대적 C++의 널리 알려진 모범 관행(best practice)들을 한곳에 모으려는 오픈소스 프로젝트로, 지금도 계속 진행 중이다. C++ 핵심 가이드라인은 수십 년간의 경험과 기존 코딩 규칙들에 기반한다. C++ 핵심 가이드라인의 지침들과 규칙들은 C++ 자체와 개념적 틀을 공유하며, 형식 안전성(type safety)과 자원 안전성, 그리고 피할 수 있는 복잡성과 비효율성의 제거에 초점을 둔다. C++ 핵심 가이드라인은 알려진 문제 영역들을 해결하기 위한 지침들과 규칙들을 제공한다. 그중 일부는 정적 코드 분석 도구를 이용해서 집행할 수 있도록 만들어졌다.

C++ 핵심 가이드라인은 특정한 하나의 주제를 손쉽게 참조하고 공유하기 쉽도록 참고자료의 형태로 구성되어 있다. 다른 말로 하면, C++ 핵심 가이드라인은 현대적 C++을 잘 활용하기 위해 처음부터 끝까지 차례대로 읽으면서 여러 주제를 익힐 수 있는 튜토리얼이 아니다. 그런 만큼, C++ 핵심 가이드라인의 규칙들을 좀 더 많은 사람이 수월하게 익히게 한다는 어렵고도 꼭 필요한 작업을 저자라이너 그림이 자신의 교육 기술과 업계 경험을 적용해서 완수했다는 것은 우리 C++ 핵심 가이드라인 편집자들에게 매우 기쁜 소식이다. 독자가 이 책으로 C++ 핵심 가이드라인을 익히면서 많은 영감을 얻기를, 그리고 C++ 핵심 가이드라인을

1 <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>

실제 업무에 적용함으로써 훨씬 더 효과적이고 즐겁게 일할 수 있게 되길 희망한다.

비야네 스트롭스트롭^{Bjame Stroustrup}

허브 서터^{Herb Sutter}

서문

이 서문의 목적은 단 하나, 친애하는 독자가 이 책을 최대한 활용하는 데 필요한 배경지식을 전달하는 것이다. 이 배경지식에는 저자인 나에 관한 세부사항과 내 저술 스타일, 이 책을 쓰게 된 동기, 그리고 이런 책을 쓰면서 어려웠던 점이 포함된다. 원한다면 이 서문을 건너뛰어도 좋지만, 서문 다음에 나오는 감사의 글은 꼭 읽어주기 바란다.

관례

다음은 이 책 전반에 쓰이는 그리 많지 않은 관례들이다.

규칙 대 지침

C++ 핵심 가이드라인은 지침(guideline)을 ‘규칙(rule)’이라고 칭할 때가 많다. 그래서 이 책에서 규칙이라는 용어를 사용하기로 한다. 이 책 전반에서 지침과 규칙은 사실상 같은 말이다.

특별한 글꼴

- 중요한 용어나 문구는 **굵은 글자**나 고딕체로 표시한다.
- 소스 코드, 명령, 키워드, 형식, 변수, 함수, 클래스 이름은 `name`처럼 고정폭 글꼴로 표시한다.

관련 규칙

C++ 핵심 가이드라인에는 서로 연관된 규칙들이 있다. 일부 장(챕터)들은 그 장에서 설명한 규칙들과 관련된 규칙들을 장의 끝부분의 '관련 규칙' 절에서 소개한다.

요약 글상자

대부분의 장은 그 장의 주요 내용을 불릿 목록으로 요약한 글상자로 끝난다.

요약

주요 사항

- 이번 장의 주요 내용

소스 코드

나는 `using` 선언과 `using namespace` 지시문을 좋아하지 않는다. 소스 코드에 쓰인 이름들의 출처(origin)가 무엇인지(이른다면 이 함수가 어떤 라이브러리에 속한 것인지) 알기 어렵기 때문이다. 그러나 지면의 제약 때문에 종종 `using` 지시문과 `using namespace` 선언을 사용해야 했다. `using namespace std;`나 `using std::cout;` 등 익숙한, 따라서 출처를 충분히 짐작할 수 있는 이름들에 대해서만 지시문과 선언을 사용하려고 노력했다. 일부 헤더는 `#include` 문을 생략하기도 했다. 부울 값은 `true` 또는 `false`로 출력된다고 가정했으며, 이에 필요한 입출력 스트림 조작자 `std::boolalpha`는 굳이 예제 코드에 포함시키지 않았다.

특별한 언급이 없는 한 예제 코드에서 마침표 세 개(...)는 코드 생략을 뜻한다(가변 인수가 아니라).

예제 코드가 하나의 완결적인 프로그램일 때는 첫 행에 소스 파일 이름을 주석으로 표시했다. 독자가 적어도 C++14를 지원하는 컴파일러를 사용한다고 가정하고 예제 코드를 작성했다. C++17이나 C++20이 필요한 경우에는 소스 파일 이름 다음에 해당 C++ 표준을 표시해 두었다.

소스 코드를 설명하기 편하도록 코드에 // ❶ 같은 주석을 붙이기도 했다. 본문에서 설명할 코드가 있는 행(지면이 여의치 않으면 그 행 바로 위의 행)에 그런 숫자를 표시해 두었다. 이 책(원서)의 깃허브 저장소(<https://github.com/>)

RainerGrimm/CppCoreGuidelines)에 있는 소스 코드에는 그런 표시가 없다. 또한, 지면의 제약으로 이 책에서는 소스 코드의 포매팅(줄 바꿈, 들여쓰기)을 저장소에 있는 것과 다르게 조절했다는 점도 밝혀 둔다.

C++ 핵심 가이드라인에 있는 예제 코드를 가져온 경우에는 가독성을 위해 `using namespace std`를 추가하거나 포매팅을 변경하기도 했다.

C++ 핵심 가이드라인에 주목한 이유

내가 C++ 핵심 가이드라인을 설명하는 책을 쓰기로 한 데에는 C++과 파이썬, 그리고 소프트웨어 전반에 대한 교육자로 15년을 보낸 경험이 크게 작용했다. 지난 몇 년 동안 나는 심장 제세동기(defibrillator)에 내장되는 소프트웨어를 개발하는 팀을 이끌었다. 내 책임에는 제세동기에 관한 규제 조항들을 준수하는 것도 포함되었다. 제세동기는 환자와 사용자의 사망이나 치명상을 유발할 수도 있는 장치인 만큼, 그런 장치를 위한 소프트웨어를 작성하는 것은 대단히 어려운 일이다.

이 책은 C++ 공동체의 일원으로서 우리가 반드시 답해야 하는 질문 하나에 기초한 것이다. 그 질문이란, “우리에게 현대적 C++을 위한 지침(guideline)들이 필요한 이유는 무엇인가?”이다. 그리고 이 질문에 대한 답은 지난 경험에서 얻은, C++에 대한 나의 세 가지 의견에서 찾을 수 있다.

C++은 초보자에게 너무 복잡하다

C++은 원래부터 복잡한(complex) 언어이고, 초보자는 특히나 복잡하게 느낀다. 근본적인 이유는 C++로 풀고자 하는 문제들 자체가 원래부터 까다롭고(complicated) 복잡할 때가 많기 때문이다. C++을 누군가에게 가르칠 때는 모든 용례(use case)의 적어도 95%에서 유효한 규칙들을 학습자에게 제시해야 한다. 내가 생각하는 그런 규칙들은 다음과 같다.

- 형식(type)을 컴파일러가 연역하게 한다.
- 초기화에는 중괄호({}) 구문을 사용한다.
- 스레드보다 작업(task)을 우선시한다.
- 원시(raw) 포인터 대신 스마트 포인터를 사용한다.

나는 내 세미나에서 이런 규칙들을 가르친다. 그렇지만 C++ 공동체 전체 차원에서 좀 더 공식적이고 규범적인 규칙 또는 모범 관행(best practice)들을 만들고 합의할 필요가 있다. 그런 규칙들은 부정적인 방식이 아니라 긍정적인 방식으로 형식화해야 할 것이다. 즉, 코드를 이리저리하게 작성하지 말라는 규칙들이 아니라 이리저리하게 작성하라는 규칙들이어야 한다.

C++은 전문가에게도 쉽지 않다

3년마다 나오는 새 C++ 표준에 포함되는 다수의 새 기능들을 익히는 것은 그리 큰 문제가 아닐 수 있지만, 경력 있는 전문 C++ 프로그래머라도 현대적 C++이 지원하는 새로운 개념들과 착안들에 잘 적응하지 못하는 경우가 있다. 코루틴과 느긋한 평가를 이용한 이벤트 주도적 프로그래밍, 무한 데이터 스트림, 구간 라이브러리(Ranges library)를 이용한 함수 합성을 생각해 보라. 템플릿 매개변수에 의미론적 범주를 도입하는 컨셉츠를 생각해 보라. C 프로그래머가 객체 지향적 프로그래밍의 개념들을 익히기란 꽤 어려운 일일 수 있다. 구식 C++에 익숙한 프로그래머들도 마찬가지이다. 여러분이 새로운 패러다임으로 전환하려면 지금까지 자신이 프로그래밍 문제를 푸는 데 사용해 온 방식을 다시 생각해 보아야 하고, 필요하다면 뜯어고쳐야 한다. 현대적 C++에는 새로운 개념이 많이 도입되었다. 나는 초보자보다는 전문 프로그래머가 오히려 그런 개념들에 적응하기가 더 어려울 것이라고 가정한다. 전문 프로그래머들은 오랫동안 성공적으로 사용해온 문제 해결 기법들을 고집하기 쉬우며, 그러다 보니 소위 망치-못 함정(hammer-nail trap)[†]에 빠지기 쉽다.

C++은 안전성이 중요한 소프트웨어에 쓰인다

내가 가장 신경 쓰는 사항이 이것이다. 안전성이 중요한(safety-critical) 소프트웨어를 개발할 때는 특정한 코딩 지침을 준수해야 할 때가 많다. 안전성을 강조하는 코딩 지침으로 가장 두드러진 예는 MISRA C++이다. MISRA C++은 영국의 *Motor Industry Software Reliability Association*(자동차 산업 소프트웨어 신뢰성 협회)¹가 발행하는 코딩 지침으로, 현재 버전은 MISRA C++:2008이다. MISRA C++

[†] [옮긴이] “망치를 들고 있으면 모든 것이 못으로 보인다”라는 속담을 비유한 것으로, 문제에 맞게 도구나 기법을 선택하는 대신 익숙한 도구나 기법에 맞게 문제를 재단하는 오류를 뜻한다.

¹ <https://www.misra.org.uk/>

은 1998년 발표된 *MISRA C guidelines*²에 기초한다. 원래는 자동차 산업을 위해 만들어진 지침이지만 이제는 항공, 군사, 의료 분야의 안전성 중요 소프트웨어의 구현을 위한 사실상의 표준이 되었다. MISRA C처럼 MISRA C++도 C++의 안전한 부분집합을 위한 지침들을 제공한다. 그런데 개념적인 문제점이 하나 있다. MISRA C++은 현대적인 C++ 소프트웨어 개발을 위한 지침이라고 말하기에는 너무 낡았다. MISRA C++ 이후로 새 C++ 표준이 네 개나 나왔다! 극명한 예로, MISRA C++은 연산자 중복적재(operator overloading)를 허용하지 않는다. 내 세미나에서는 `auto constexpr dist = 4 * 5_m + 10_cm - 3_dm;`처럼 사용자 정의 리터럴을 이용해서 형식에 안전한 산술을 구현해야 한다고 가르친다. 산술 연산자들과 적절한 접미사들을 위한 리터럴 연산자들을 중복적재하지 않고는 이런 형식에 안전한 산술을 구현할 수 없다. 솔직히 나는 MISRA C++이 빠르게 진화해서 현재의 C++ 표준을 따라잡을 것이라고는 믿지 않는다. C++ 핵심 가이드라인처럼 공동체가 주도하는 지침만이 계속 발전하는 C++ 표준과 보조를 맞출 수 있다.



MISRA C++과 AUTOSAR C++14의 병합 계획

하지만 희망이 있다. MISRA C++이 AUTOSAR C++14를 포함할 것이라고 한다. *AUTOSAR C++14*³는 C++14에 기초한 지침으로, MISRA C++을 확장하기에 아주 적합하다. 하지만 업계의 단체가 주도하는 지침이 현대적 C++의 역동적인 변화를 계속해서 따라잡을 수 있을지는 심히 의심스럽다.

나의 도전 과제

나는 2019년 5월에 비야네 스트롭스트롭과 허브 서티에게 C++ 핵심 가이드라인에 관한 책을 쓰고 싶다는 메일을 보냈다. 그 메일에서 핵심 부분을 인용하자면 다음과 같다: “저는 C++ 핵심 가이드라인의 가치를 절대적으로 지지합니다. 왜냐하면 저는 우리 C++ 프로그래머들에게 현대적 C++의 올바른/안전한 사용법에 관한 지침들이 꼭 필요하다고 믿기 때문입니다. 제 C++ 강좌에서 C++ 핵심 가이드라인의 예제들과 착안들을 자주 사용하고 있습니다. C++ 핵심 가이드라인의 포맷은 대체로 MISRA C++이나 AUTOSAR C++14의 규칙들과 비슷한데, 아마 의도적

² https://en.wikipedia.org/wiki/MISRA_C

³ https://www.autosar.org/fileadmin/standards/adaptive/18-03/AUTOSAR_RS_CPP14Guidelines.pdf

으로 그렇게 하겠지만 좀 더 많은 사람이 접하기에 이상적인 형태는 아니라고 생각합니다. 만일 C++ 핵심 가이드라인의 일반적인 개념들을 설명하는 또 다른 문서가 있다면 더 많은 사람이 C++ 핵심 가이드라인의 지침들을 읽고 고민하게 되리라 생각합니다.”

이와 관련해서 몇 가지 더 언급하고자 한다. 지난 몇 년 동안 나는 C++ 핵심 가이드라인에 관한 백여 건의 글을 써서 내 독일어 및 영어 블로그에 올렸다. 또한 독일어 잡지 *Linux-Magazin*⁴에서 C++ 핵심 가이드라인에 관한 글을 연재했다. 이런 글들을 쓴 이유는 두 가지이다. 첫째로, 나는 C++ 핵심 가이드라인을 더 많은 사람이 알아야 한다고 생각했다. 둘째로, 나는 C++ 핵심 가이드라인의 규칙들을 좀 더 읽기 쉬운 형태로 설명하고 필요하다면 배경 정보도 함께 제공하고자 했다.

이런 글들을 쓰는 것이 쉽지는 않았다. C++ 핵심 가이드라인에는 지침이 500개가 넘는다. 대부분의 경우 C++ 핵심 가이드라인은 지침들을 그냥 ‘규칙’이라고 부른다. 이 규칙들은 대부분 정적 분석을 염두에 두고 작성된 것들이다. 전문 C++ 소프트웨어 개발자에게 생명줄과도 같은 규칙들이 많지만, 너무 전문적인 규칙도 많고, 불완전하거나 다른 규칙들과 겹치는 규칙들도 드물지 않다. 심지어는 다른 규칙과 모순되는 규칙들도 있다. 이 책을 쓰면서 내가 설정한 도전 과제는, C++ 핵심 가이드라인의 규칙 중에서 C++을 사용하는 전문적인 소프트웨어 개발자에게 꼭 필요한 주요 규칙들을 선별하고, 거기서 너무 난해한 내용을 제거하고 누락된 배경 정보를 채워서 읽기 좋은(심지어 재미있는) 이야기를 만들어 내는 것이었다.

판타 레이

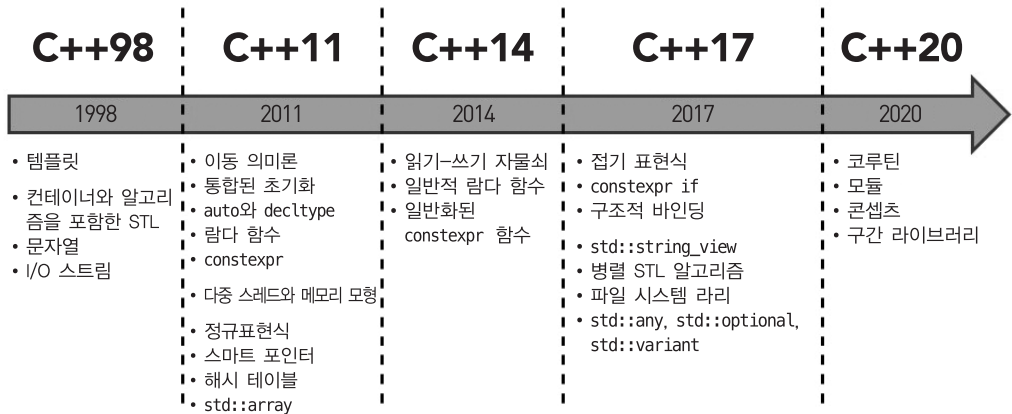
판타 레이^{Panta rhei}는 그리스 철학자 헤라클레이토스(Heraclitus of Ephesus)⁵의 유명한 말로, “모든 것은 흐른다”라는 뜻이다.* 판타 레이는 이 책을 쓰면서 내가 겪은 어려움을 잘 표현한다. C++ 핵심 가이드라인은 깃허브 저장소로 관리되는, 250여 명이 참여한 공개 프로젝트⁶이다. 이 책을 쓰면서 근거로 삼은 규칙들과 문장들이 계속해서 바뀌었고 앞으로도 바뀔 것이다.

4 <https://www.linux-magazin.de/>

5 <https://en.wikipedia.org/wiki/Heraclitus>

* [옮긴이] 같은 뜻의 사자성어로 만물유전(萬物流轉)이 있다.

6 <https://github.com/isocpp/CppCoreGuidelines>



게다가 C++ 표준 자체도 계속 진화한다. 실제로 C++ 핵심 가이드라인에는 향후 표준(C++23 등)에 채용될 기능들에 관한 내용도 포함되어 있다. 계약(contract) 관련 규칙들이 대표적인 예이다. 이런 점들을 고려해서 나는 몇 가지 사항을 결정했다.

1. 언급한 C++ 핵심 가이드라인 규칙의 URL을 각주로 제시한다.
2. C++17 표준에 초점을 두되, 필요하다면 C++20 표준과 관련한 규칙들도 제시한다(컨셉츠 등).
3. C++ 핵심 가이드라인은 새 C++ 표준이 발표되면 더욱 크게 변할 것이다. 그에 따라 이 책의 개정판을 낼 계획이다.

이 책을 읽는 방법

이 책의 구성은 C++ 핵심 가이드라인의 구조를 반영한다. 제1부의 장들은 C++ 핵심 가이드라인의 주요 섹션들과 대응되고, 제2부의 장들은 지원 섹션들에 대응된다. C++ 핵심 가이드라인의 섹션들과는 대응되지 않는 부록들도 있다. 부록들에서는 C++ 핵심 가이드라인의 규칙들을 실제로 집행하는 방법과 C++20의 컨셉츠, 그리고 이후 표준에 포함될 계약 기능을 소개한다.

마지막으로, 이번 절의 제목인 “이 책을 읽는 방법”을 이야기하겠다. C++ 핵심 가이드라인의 주요 섹션들에 대응되는 제2부의 장들은 모두 읽기 바란다. 가능하면 각 장을 처음부터 끝까지 읽어 주었으면 한다. 지원 섹션들을 다루는 제2부는 추가적인 정보를 제공하는데, 특히 GSL(Guidelines Support Library)을 소개한다. 제3부의 부록들은 주요 섹션들을 이해하는 데 도움이 되는 필수 배경지식에 대한 참고자료로 의도한 것이다. 이런 추가 정보는 이 책의 완성도에 꼭 필요하다.