[용어 수정]

폐쇄 객체

->

클로저 객체

기본 클래스

->

기반 클래스

[p.37: 아래에서 4행]

사용자가 16비트 분량 2개로 구성된 구조체를 선언할 경우 해당 게임에서의 구조체는

->

사용자가 16비트 분량 2개로 구성된 구조체를 선언할 경우 해당 해당 구조체는

[p.67: 2.2.1 Hello, World! 절에서 5행]

중괄호 { }는 포인터에서 그룹을 표현한다.

->

중괄호 { }는 C++에서 그룹을 표현한다.

[p.78: 아래에서 4행]

Vector(int)는 Vetor 타입의 객체가 생성되는 방식을 정의한다.

->

Vector(int)는 Vector 타입의 객체가 생성되는 방식을 정의한다.

[p.81: 아래에서 4행]

sum+=sqr t(v[i]);

->

sum+=sqrt(v[i]);

[p.95: 아래에서 12행]

Vector의 최기화 리스트 생성자는

->

Vector의 초기화 리스트 생성자는

[p.97: 7행]

public:은

->

:public은

[p.99: 3.2.4 클래스 계층 구조 단락 절에서 2행]
(이를테면 public:)
->
(이를테면 :public)

[p.106: 아래에서 4행]
'우변 값rvalue' 이란 단어는 '좌변 값 lvalue'과 짝을 이루기 위해 의도됐으며, 대략적으로 '대입문의 왼쪽에 등장하는 뭔가'라는 의미다.
->
'우변 값rvalue' 이란 단어는 대략적으로 '대입문의 왼쪽에 등장하는 뭔가'라는 의미를 갖는 '좌변 값 lvalue'과 짝을 이루기 위한 것이다.

[p.110: 아래에서 13행]
return x.size() * &x[0] : nullptr;
->
return x.size() ? &x[0] : nullptr;

[p.114: 아래에서 3행]
이것은 call f(2.2, "hello")를 호출하고
->
이것은 f(2.2, "hello")를 호출하고

[p.118: 아래에서 8행]
컨테이너의 프레임워크(vector나 map 등)와 알고리즘(find(), sort(), merge() 등)
->
컨테이너(vector나 map 등)와 알고리즘(find(), sort(), merge() 등)의 프레임워크

[p.145: 21행]
공간이나 시간 오버헤드를 없는
->
공간이나 시간 오버헤드가 없는

[p.153: 15행]
get()이 예외를 던지게 될 것이다(시스템으로부터 아니면 우리가 값을 get()하려고 시도했던 테스크로부터 전송돼).
->
get()이 예외(시스템, 또는 함수 get()을 호출하려는 태스크로부터 전송된)를 던지게 될 것이다.

[p.170: 12~13행]

따라서 데이터 경합을 방지하기 위해 동기화 메커니즘이 제공되지 않는 한 변수의 값은 두 개의 스레드로부터 할당된다. (41.2절).

->

두 개의 스레드에서 한 변수에 값을 할당하는 경우도 데이터 경합을 방지하기 위한 동기화 메커니즘을 사용하지 않으면 변수의 값을 예측할 수 없다. (41.2절).

[p.174: 아래에서 2행]

bool z = a.b;

->

bool z = a-b;

[p.175: 2행]

void g(int. p)

->

void g(int* p)

[p.177: 9행]

일반적인 '0'+i는 int이므로, static_cast<char>는 빼버렸다. 결과는 0, 1 등이 아니라 48, 49 등의 값이 될 것이다.

->

일반적으로 '0'+i는 int이므로, static_cast<char>를 빼버렸다면, 결과는 0, 1 등이 아니라 48, 49 등의 값이 될 것이다.

[p.177: 15~17행]

일반적인 '0'+i는 int이므로, static_cast<char>는 빼버렸다. 결과는 0,1 등이 아니라 48, 49 등의 값이 될 것이다.

->

일반적으로 '0'+i는 int이므로, 만약 여기서 static_cast<char>를 뺐다면 결과는 0, 1 등이 아니라 48, 49 등이 될 것이다.

[p.177: 아래에서 12행~]

char가 부호 있는 기기에서는 답이 255다.

->

char가 부호 없는 기기에서는 답이 255다.

[p.177: 아래에서 3~1행]

char. pc = &uc;

signed char. psc = pc;

unsigned char. puc = pc;
->
char＊ pc = &uc;
signed char＊ psc = pc;
unsigned char＊ puc = pc;

[p.178: 13행]
uc의 값이 지나치게 크면
->
c의 값이 지나치게 크면

[p.178: 17행]
signed char sc = .140;
->
signed char sc = -140;

[p.183: 6.2.6 접두사와 접미사 표 4행]
10진수
->
16진수

[p.185: 6행]
void. pv;
->
void* pv;

[p.188: 2행]
적성할 수 있다.
->
작성할 수 있다.

[p.188: 19행]
uninitialized_copy(vx.begin(),vx.begin()+max,reinterpret_cast<X.>(buffer));
->
uninitialized_copy(vx.begin(),vx.begin()+max,reinterpret_cast<X＊>(buffer));

[p.188: 6.3 선언 단락에서 9~16행]
const char. name = "Njal";
const char. season[] = { "spring", "summer", "fall", "winter" };

```
vector<string> people { name, "Skarphedin", "Gunnar" };
struct Date { int d, m, y; };
int day(Date. p) { return p.>d; }
double sqrt(double);
template<typename T> T abs(T a) { return a<0 ? .a : a; }
constexpr int fac(int n) { return (n<2)?1:n.fac(n.1); }
constexpr double zz { ii.fac(7) };
->
const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };
vector<string> people { name, "Skarphedin", "Gunnar" };
struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<typename T> T abs(T a) { return a<0 ? -a : a; }
constexpr int fac(int n) { return (n<2)?1:n*fac(n-1); }
constexpr double zz { ii*fac(7) };
```

[p.189: 15행, 19행]

```
const char. name = "Njal";
->
const char* name = "Njal";
```

```
int day(Date. p) { return p.>d; }
->
int day(Date* p) { return p->d; }
```

[p.189: 21행]

위의 선언 중 단 3개만 정의에 해당한다.

->

위의 선언 중 단 3개만 정의에 해당하지 않는다.

[p.190: 5행, 12행]

```
int day(Date. p) { return p.>d; }
->
int day(Date* p) { return p->d; }
```

```
const char. name {"Bjarne"};
->
```

const char* name {"Bjarne"};

[p.190: 6.3.1 선언의 구조 절 4 행]

선언은 크게 네 부분으로 나뉜다.

->

선언은 크게 다섯 부분으로 나뉜다.

[p.192: 6.3.2 여러 개의 이름을 선언하는 방법 절에서 6행(대문자->소문자)]

Int* p, y;

->

int* p, y;

[p.194: 아래에서 6행]

어떤 이름이 함수(12장), 람다(11.4절), 클래스(16장) 또는 enum 클래스(11.4절) 바깥에서 정의된 경우 그 이름은 멤버 이름(또는 클래스 멤버 이름)이라고 불린다.

->

어떤 이름이 어느 클래스의 안이면서 함수(12장), 람다(11.4절), 클래스(16장) 또는 enum 클래스(11.4절) 바깥에서 정의된 경우 그 이름은 멤버 이름(또는 클래스 멤버 이름)이라고 불린다.

[p.195: 아래에서 14행]

Int* p = &x;

->

int* p = &x;

[p.197: 10행]

C++에서 처음 도입된

->

C++11에서 처음 도입된

[p.197: 16행]

{}를 사용한 초기화, 리스트 초기화는 축소를 허용하지 않는다.

->

{}를 사용한 초기화, 즉 리스트 초기화는 축소를 허용하지 않는다.

[p.199: 6행]

constexper int max = 1024.1024

->

constexper int max = 1024*1024

[p.199: 아래에서 14행(띄어쓰기)]

int* p {newint};

->

int* p {new int};

[p.199: 아래에서 11행]

// 문자열의 기본 생성자이므로 v=={}

->

// 벡터의 기본 생성자이므로 v=={}

[p.199: 아래에서 10행]

// 문자열의 기본 생성자이므로 ps는 ""

->

// 문자열의 기본 생성자이므로 *ps는 ""

[p.200: 아래에서 6행]

() 기호를 사용한 초기화에서는

->

{} 기호를 사용한 초기화에서는

[p.206: 아래에서 7행]

typedef void(.PtoF)(int);

->

typedef void(*PtoF)(int);

[p.211: 아래에서 11행]

void*의 주된 용도는 객체의 타입을 예측할 수 없는 함수를 가리키는 포인터를 전달하거나 함수에서 타입 미지정 객체를 반환하는 것이다.

->

void*의 주된 용도는 타입을 알 수 없는 객체를 함수에 전달할 때나, 함수에서 타입 미지정 객체를 반환할 때 포인터로 사용하는 것이다.

[p.212: 아래에서 6행]

첨자 연산자 []나 포인터(연산자 *나 연산자 []를 이용해서, 7.4절)를 통해 배열에 접근할 수 있다.

->

첨자 연산자 []나 포인터를 통해(연산자 *나 연산자 []를 이용해서, 7.4절) 배열에 접근할 수 있다.

[p.213: 아래에서 11행]

배열은 간단히 자신의 첫 번째 원소를 가리키는 포인터로 암시적으로 변환된다.

->
배열의 이름은 간단히 자신의 첫 번째 원소를 가리키는 포인터로 암시적으로 변환된다.

[p.218: 마지막 행]
us 및 Rs의 순서와 대소문자 여부는 중요하다.
->
u 및 R의 순서와 대소문자 여부는 중요하다.

[p.220: 아래에서 15행]
++는 포인터 값을 증가시키는 역할을 하므로 p++는 배열의 다음 원소를 가리키게 된다.
->
++는 포인터 값을 증가시키는 역할을 하므로 p는 배열의 다음 원소를 가리키게 된다.

[p.220: 아래에서 10행]
모든 범위 a를 갖는 기본 제공 배열 a와 정수 j에 대한 예는 다음과 같다.
->
a의 범위 내에서 기본 제공 배열 a와 정수 j에 대한 예는 다음과 같다.

[p.221: 18행]
sizeof(int)라는 점을 보여준다.
->
sizeof(int)는 4라는 점을 보여준다.

[p.222: 7행(들여쓰기 오류)]
```
for (int x : v)
    use(x); // 오류: 포인터에는 통하지 않는다.
    const int N = 7;
```
->
```
for (int x : v)
    use(x); // 오류: 포인터에는 통하지 않는다.
const int N = 7;
```

[p.223: 14행]
int ouch = good[1,4]; int nice = good[1][4];
->
int ouch = good[1,4];

[p.224: 4행]
td::vector

->
std::vector

[p.228: 아래에서 12행]
이렇게 하면 자원 핸들 내에 존재하지 않는 모든 포인터는 소유권자가 아니기 때문에 delete 처리돼야 한다고 가정할 수 있다.
->
이렇게 하면 자원 핸들 내에 존재하지 않는 모든 포인터는 소유권자가 아니라고 가정할 수 있고, 이것들은 delete 처리하면 안 된다.

[p.228: 아래에서 4행]
obj와 p 대신 *p를 사용하고, obj.m 대신 p->m을 사용한다.
->
obj 대신 *p를 사용하고 obj.m 대신 p->m을 사용한다.

[p.229: 아래에서 3행]
보존한 필요가 없는
->
보존할 필요가 없는

[p.230: 8행]
int int x = r;
->
int x = r;

[p.233: 아래에서 6행]
string& r1 {var}; // 우변 값 참조자, r1(좌변 값)을 var에 묶는다.
string& r2 {f()}; // 우변 값 참조자, 오류: f()는 우변 값
string& r3 {""Princeton""}; // 우변 값 참조자, 오류: 임시 객체에 묶을 수 없다.
string&& rr1 {f()}; // 좌변 값 참조자, 문제없음: rr1을 rvalue(임시 객체)에 묶는다.
string&& rr2 {var}; // 좌변 값 참조자, 오류: var은 좌변 값
->
string& r1 {var}; // 좌변 값 참조자, r1을 var(좌변 값)에 묶는다.
string& r2 {f()}; // 좌변 값 참조자, 오류: f()는 우변 값
string& r3 {""Princeton""}; // 좌변 값 참조자, 오류: 임시 객체에 묶을 수 없다.
string&& rr1 {f()}; // 우변 값 참조자, 문제없음: rr1을 우변 값(임시 객체)에 묶는다.
string&& rr2 {var}; // 우변 값 참조자, 오류: var은 좌변 값

[p.233: 16행]

const가 아닌 우변 값 참조자는 참조자의 사용자가 값을 입력할 수 있는 객체를 참조한다.
->
const가 아닌 좌변 값 참조자는 참조자의 사용자가 값을 입력할 수 있는 객체를 참조한다.

[p.233: 17행]
const 우변 값 참조자는 참조자의 사용자 관점에서 변할 수 없는 상수를 참조한다.
->
const 좌변 값 참조자는 참조자의 사용자 관점에서 변할 수 없는 상수를 참조한다.

[p.233: 아래에서 6~4행]
// 우변 값 참조자
->
// 좌변 값 참조자

[p.233: 아래에서 3~2행]
// 좌변 값 참조자
->
// 우변 값 참조자

[p.235: 아래에서 5행]
여기의 예제는 최선 버젼의 swap()에 의해 처리된다.
->
여기의 예제는 마지막 버전의 swap()에 의해 처리된다.

[p.236: 아래에서 1행]
어떤 객체를 참조할지 바꿔야 한다면 포인터를 사용한다.
->
가리키는 객체가 변경될 수 있다면, 포인터를 사용한다.

[p.237: 1행]
-를 사용할 수 있다.
->
--를 사용할 수 있다.

[p.239: 아래에서 3행]
저수준의 코드가
->
저수준의 코드를

[p.241: 3행]
재래식 데이터Plain Old Data),
->
재래식 데이터Plain Old Data,

[p.243: 14행]
address prev
->
Address prev

[p.251 17행]
C++ 배열 바이너리 인터페이스
->
C++ 애플리케이션 바이너리 인터페이스

[p.251: 아래에서 11행, 10행]
클래스가 virtual 함수인 경우
->
클래스가 virtual 함수를 갖고 있는 경우

클래스가 virtual 기본 클래스인 경우
->
클래스가 virtual 기본 클래스를 갖고 있는 경우

[p.252: 16행]
기본이 아닌 생성자의 덧셈이나 뺄셈은 구조나 성능에 영향을 미치지 않는다는 점에 주목하기 바란다.
->
기본 생성자 외의 생성자를 추가하거나 제거하는 것은 구조나 성능에 영향을 미치지 않는다는 점에 주목하기 바란다.

[p.254: 아래에서 10행(띄어쓰기 오류)]
Fudg e a;
->
Fudge a;

[p.254: 아래에서 5행]
정수에 홀수 주소가 들어갈 수 없는 경우도 있다
->

정수가 홀수 주소를 가질 수 없는 경우도 있다.

[p.260: 9행(줄바꿈 오류)]

static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2 static_cast<int>(Warning::red)==3
->
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3

[p.260: 아래에서 10행(띄어쓰기 오류)]

acknowledg e=1
->
acknowledge=1

[p.261: 13행(띄어쓰기 오류)]

if((x&Printer_flags::acknowledg e)!=none){
->
if((x&Printer_flags::acknowledge)!=none){

[p.263: 10행]

단일 유효 범위 내에 있는 두 개의 순수 열거형에서 red를 정의한 관계로 찾기 어려운 오류가 일어나지 않았다는 점에서 우리는 운이 좋았다.
->
운 좋게도 단일 유효 범위 내에 있는 두 개의 순수 열거형에서 red를 정의하는 바람에, 찾기 어려운 오류를 발견할 수 있었다.

[p.263: 22행(대문자->소문자)]

컴파일러는 X==red 란 문장을 ~
->
컴파일러는 x==red란 문장을 ~

[p.264: 아래에서 9행]

작은 멤버보다 좀 더 큰 멤버를 가진 구조체 데이터 멤버로 구조로 만든다.
->
구조체의 큰 멤버들을 작은 멤버들보다 앞에 배치하도록 한다.

[p.264: 아래에서 7행]

하드웨어에 의해 강제되는 데이터 구조를 표시하려면 비트필드를 활용한다.

->

하드웨어의 구조를 나타내는 데이터 구조를 표시하려면 비트필드를 활용한다.

[p.266: 아래에서 3행]

대입문 프로시저 호출문 같은 건 존재하지 않는다

->

대입문이나 프로시저 호출문 같은 건 존재하지 않는다

[p.267: 아래에서 7행]

위 코드는 대입이 끝난 후에 (빈 문자열로의)기본 초기화를 요청한다.

->

위 코드는 (빈 문자열로의) 기본 초기화를 하고 나서 대입을 한다.

[p.268: 10행]

error()는 값을 반환하지 않을 것으로 추측된다. 값을 반환한다면

->

error()는 제어를 돌려주지 않을 것으로 가정한다. 제어를 돌려준다면

[p.269: 22행]

특히 이름은 if문의 또 다른 분기문에 대해서는 사용될 수 없다

->

if문의 한 분기문에서 선언한 이름은 다른 분기문에서 바로 사용할 수 없다

[p.269: 아래에서 9행]

+x;

->

++x;

[p.270: 17행]

f();

->

들여쓰기

[p.274: 5행]

원소(여기서는 x)를 지칭하는 변수의 첫 번째는 for문이다.

->

원소를 지칭하는 변수(여기서는 x)의 유효 범위는 for문이다.

[p.274: 7행]

begin()과 v.end() 또는

->

v.begin()과 v.end() 또는

[p.278: 9행(들여쓰기 오류)]

```
for (int i = 0; i!=v.size(); ++i) {
   if (prime(v[i]))
return v[i];
```

->

```
for (int i = 0; i!=v.siz e(); ++i) {
      if (prime(v[i]))
         return v[i];
```

[p.284: 3행]

Pi는 사전에 정의돼 있음

->

pi는 사전에 정의돼 있음

[p.284: 3행과 4행 사이]

3행 (Pi는 사전에 정의돼 있음) 계산기 프로그램은 다음의 결과를 출력한다.

4행 여기서 2.5는 첫 번째 행의 계산 결과이고 , 19.635는 두 번째 행의 계산 결과다.

->

3행 (Pi는 사전에 정의돼 있음) 계산기 프로그램은 다음의 결과를 출력한다.

```
2.5
19.635
```

6행 여기서 2.5는 첫 번째 행의 계산 결과이고 , 19.635는 두 번째 행의 계산 결과다.

[p.285: 11행]

문자열은 사용자가 커맨드라인에서 cin에 입력하는 것에서부터 바로 들어오든지, 여타 입력 스트림에서 들어오는 것을 보게 될 것이다

->

문자열은 사용자가 cin에 입력하는 것, 프로그램의 커맨드 라인, 혹은 여타 입력 스트림 어디에서도 들어올 수 있다

[p.285: 아래에서 16행]

토큰을 정수 값으로 표시하는 방식은 간편하면서 효율적이며

->

토큰을 해당 문자의 정수 값으로 표시하는 방식은 간편하면서 효율적이며

[p.287: 아래에서 22행]
유효 범위는 해당 조건식에 의헤
->
유효 범위는 해당 조건식에 의해

[p.289: 아래에서 4행]
Token_stream은 자신의 입력 스트림을 가리키는 포인터(ip), 불리언(owns), 입력 스트림과 현재의 토큰의 소유권 표시(ct)의 3개 값을 보관한다
->
Token_stream은 입력 스트림을 가리키는 포인터(ip), 이 입력 스트림의 소유권을 표시하는 불리언(owns), 현재 토큰(ct), 이렇게 3개 값을 보관한다

[p.293: 6행]
₩n1을 제외한 공백은 건너뛴다
->
'₩n'을 제외한 공백은 건너뛴다

[p.295: 2행]
oid calculate()
->
void calculate()

[p.296: 아래에서 9, 7행]
Stringstream
->
istringstream

[p.297: 아래에서 6행]
시용하라고
->
사용하라고

[p.299: 9행]
함수 스타일 타입 변환 type { expr-list }
->
함수 스타일 타입 변환 type (expr-list)

[p.300: 1행]
*p는 *(p++)를 뜻하며 (*p)++ 가 아니다
->
*p++는 *(p++)를 뜻하며 (*p)++ 가 아니다

[p.301: 중간 표 제목]
토크 요약 -> 토큰 요약

[p.301: 아래에서 8행]
&&나 - 같은 기호를
->
&&나 ~같은 기호를

[p.303: 아래에서 14행]
명시적으로 문자열로 표현할 때만 작성해야 한다
->
명시적으로 표현할 때만 작성해야 한다

[p.306: 아래에서 10행]
지역 변수의 참조자를 반환하는 것은 오류이고(12.14절)
->
지역 변수의 참조자를 반환하는 것은 오류이고(12.1.4절)

[p.306: 아래에서 9행]
const가 아닌 참조자에 ~
->
const가 아닌 좌변 값 참조자에 ~

[p.313: 8행]
// 주의: 배정밀도 부동소수점에서 char로의 변환
->
// 에러: 배정밀도 부동소수점에서 char로의 변환

[p.314: 18행]
*는 const T*로 암시적으로 변환될 수 있다
->
T*는 const T*로 암시적으로 변환될 수 있다

[p.316: 10행]

그렇지 않은 경우 어느 한쪽 피연산자가 unsigned long이면 다른 쪽도 unsigned long으로 변환된다
->
그렇지 않은 경우 어느 한쪽 피연산자가 unsigned long long이면 다른 쪽도 unsigned long long으로 변환된다

[p.319: 3행]
"내가 희망하는 하는 것만 말하면 되는 프로그래밍 언어를 원한다"
->
"내가 희망하는 것을 말하기만 하면 되는 프로그래밍 언어를 원한다"

[p.319: 아래에서 1행]
여기서 p는 nullptr이 아니라면 역참조되지 않는다.
->
여기서 p가 nullptr이면 역참조되지 않는다.

[p.321: 6, 7행]
static_asser t(sizeof(int)==4, ""unexpected int size"");
static_asser t(sizeof(short)==2, ""unexpected short size"");
->
static_assert(sizeof(int)==4, ""unexpected int size"");
static_assert(sizeof(short)==2, ""unexpected short size"");

[p.323: 3행(들여쓰기, 중괄호 누락)]
```
while (*q != 0) {
*p = *q;
p++; // 다음 문자를 가리킨다.
q++; // 다음 문자를 가리킨다.
*p = 0; // 끝을 나타내는 0
```
->
```
while (*q != 0) {
    *p = *q;
    p++; // 다음 문자를 가리킨다.
    q++; // 다음 문자를 가리킨다.
}
*p = 0; // 끝을 나타내는 0
```

[p.324: 22행]
반환 노드

->
노드 반환

[p.327: 아래에서 15행]
따라서 어떤 원소를 복사하는 것이라 단순히 ss를 reverse() 밖으로 이동시킨다
->
여기서는 ss의 어떤 원소도 복사하지 않고 간단히 ss를 reverse() 밖으로 이동시킨다

[p.327: 아래에서 1행]
즉, new는 생성자나 비슷한 연산 속에 포함되고 delete는 소멸자에 포함돼 둘이 함께 일관성 있는 메모리 관리 전략을 제공하는 것이다
->
즉, new는 생성자에 포함되고 비슷한 연산인 delete는 소멸자에 포함돼 둘이 함께 일관성 있는 메모리 관리 전략을 제공하는 것이다

[p.329: 아래에서 7행]
if (p)가 0이 아니면
->
if (p)가 참이면

[p.329: 아래에서 5행]
if (p)가 0이 아니면 new() 연산자로 할당된 공간을 할당 해제한다.
->
if (p)가 참이면 new[]() 연산자로 할당된 공간을 해제한다.

[p.336: 3행]
오류: 대입 연산의 왼쪽 피연산자가 아님
->
오류: 대입 연산의 왼쪽 피연산자는 안됨

[p.336: 3행]
오류: 대입이 아닌 연산자의 피연산자가 아님
->
오류: 대입이 아닌 연산자의 피연산자는 안됨

[p.340: 아래에서 11행]
STL 컨테이너를 그런 순회 탐색을
->
STL 컨테이너는 그런 순회 탐색을

[p.341: 아래에서 5행]
이름을 가진 변수들는 앞에 &가 붙고 참조에 의해 캡처된다.
->
앞에 &가 붙은 이름을 가진 변수들은 참조로 캡처된다.

[p.342: 19행]
이와 달리 지정하지 않는다면
->
이렇게 지정함으로써

[p.346: 아래에서 4행]
reinterprint_cast
->
reinterpret_cast

[p.347: 아래에서 18~10행]
던질 것이다
->
예외를 던질 것이다

[p.350: 13행]
일부 static_cast는 이식 가능하지만, reinterpret_case는
->
일부 static_cast는 이식 가능하지만, reinterpret_cast는

[p.351: 12행]
11.5.4 함수 스타일 클래스
->
11.5.4 함수 스타일 캐스트

[p.354: 5행]
선언문 역시 함수 정의라고 볼 수 있으므로 컴파일러는 그런 이름을 무시할 수도 있다.
->
선언이 정의와 별개로 있는 경우에 컴파일러는 인자이름을 무시한다.

[p.355: 9행]
함수 호출이 함수 본체에 인라인으로 넣어서
->

함수 호출이 함수 본체를 인라인으로 넣어서

[p.355: 아래에서 3행]
void swap(int*, int*); // 정의
->
void swap(int*, int*); // 선언

[p.355: 아래에서 3행]
void swap(int* p, int* q) // 선언
->
void swap(int* p, int* q) // 정의

[p.357: 11행]
전위형 반환 타입 앞에는 ->가 붙는다
->
후위형 반환 타입 앞에는 ->가 붙는다

[p.358: 14행]
static이 아닌 지역 변수를 가리키는 포인터는 결코 반환되지 않을 것이다
->
static이 아닌 지역 변수를 가리키는 포인터는 결코 반환해서는 안 된다.

[p.359: 아래에서 2행]
인라인 함수가 한 가지 이상의 해석 단위로 정의돼 있다면(보통 헤더에서 정의된 경우가 그런 예다) 다른 해석 단위에 있는 그것의 정의는 동일해야 한다.
->
인라인 함수가 여러 소스 파일에 정의돼 있다면(헤더에서 인라인 함수를 정의하면 이런 경우가 흔히 발생한다) 각 소스 파일의 인라인 함수의 정의는 동일해야 한다.

[p.361: 7행]
ODR('하나의 정의 규칙')
->
ODR(One Definition Rule, '하나의 정의 규칙')

[p.362: 9행]
constexpr 함수 내에서 받아들여지지 않는 조건 표현식의 분기는 평가되지 않는다. 이는 받아들여지지 않는 분기의 경우 런타임 평가를 필요로 한다는 뜻이다.
->
constexpr 함수 내에서 결과를 알 수 없는 조건 표현식은 컴파일 타임에 결과값을 구하지 않는다.

이런 경우는 런타임에 결과값을 구하게 된다.

[p.362: 19행]
던진다
->
예외를 던진다

[p.362: 20행]
low와 high는 설계 시점이 아니라 컴파일 타임에 알 수 있는 환경 구성 매개변수이며, f(x,y,z)는 일부 구현별 정의를 따르는 값을 계산한다는 점을 추측할 수 있을 것이다.
->
low와 high는 함수를 작성하는 시점에는 모르지만 컴파일 타임에는 알 수 있는 환경 구성 매개변수이며, f(x,y,z)는 구현에 따라 결과값이 달라진다는 점을 추측할 수 있을 것이다.

[p.363: 2행]
지역 변수는 실행 스레드가 자신의 정의에 다다를 때
->
static 지역 변수는 실행 스레드가 자신의 정의에 다다를 때

[p.363: 아래에서 2행]
그런 것을 사용할 정도로 여러분이 무모하다면 레이블의 유효 범위(9.6절)는 전체 함수여야 하며, 그것이 중첩된 유효 범위 중 어디에 포함돼 있는지는 고려하지 말아야 한다.
->
무모하게도 레이블을 굳이 쓰겠다면, 레이블이 함수 내의 중첩된 유효 범위 중 어디에 있더라도 해당 레이블의 유효 범위는 함수 전체가 된다.

[p.364: 1행]
12.2.
->
12.2

[p.364: 2행]
전위형 ()를 이용해서
->
후위형 ()를 이용해서

[p.364: 19행]
연산으로 변경되지 않는다
->

연산에 의해 변경되지 않는다

[p.365: 아래에서 8, 7행]

fsqr t

->

fsqrt

[p.366: 5행]

update()는 즉시 삭제됐던 임시 객체들을

->

update()는 직전에 삭제됐던 임시 객체들을

[p.366: 7행]

정확히 하고 싶다면 참조에 의한 전달은 좌변 값 참조에 의한 전달이 돼야 한다

->

정확히 말하자면 참조에 의한 전달은 '좌변 값 참조에 의한 전달'이다.

[p.366: 20행]

함수는 우변 값 인자를 변경할 것이고, 그로 인해 우변 값 인자는 소멸과 재대입에만 쓰일 수 있다고 가정해야 한다.

->

함수가 우변 값 인자를 변경하는 것을 가정해야 하기 때문에, 우변 값 인자는 소멸과 재대입에만 사용하는 것이 좋다.

[p.369: 아래에서 4행]

타입을 대한

->

타입에 대한

[p.377: 아래에서 9, 8행]

sqr t

->

sqrt

[p.385: 아래에서 2행]

따라서 매크로를 사용할 때는 디버거, 상호 참조 도구, 프로파일러 같은 하위 수준의 서비스만 기대해야 한다.

->

따라서 매크로를 사용할 때는 디버거, 상호 참조 도구, 프로파일러 같은 도구에서 제공하는 서비

스의 수준이 떨어지게 된다.

[p.386: 아래에서 12행]
FA C
->
FAC

[p.397: 14행]
예외 처리 계획은 한 가지 측면은 처리되지 못하는 오류(잡히지 못한 예외)에 대응해 궁극적인 응답이 프로그램의 종료라는 점이 일부 프로그래머들에게는 신기하게 여겨질 수 있다는 것이다.
->
예외 처리 계획 중에 일부 프로그래머들에게 신기하게 여겨질 수 있는 한 가지는, 처리되지 못하는 오류(잡히지 못한 예외)에 대응하는 궁극적인 응답으로 프로그램을 종료하는 것이다.

[p.398: 13행]
주어진 작업을 수행하지 못하는 프로그램의 어떤 부분에 특별히 예외적일 만한 것이 없다는 사실을 감안하면 '예외'란 단어가 부적절하다고 여기질 수도 있다.
->
프로그램의 어느 부분이 주어진 작업을 수행하지 못할 수도 있는 것이 특별히 예외적인 상황이 아니라는 사실을 감안하면 '예외'란 단어가 부적절하다고 여겨질 수도 있다.

[p.398: 아래에서 6행]
예외를 '요청된 작업을 처리할 수 없는 시스템의 어떤 부분'이라는 의미로 생각해보자.
->
예외를 '시스템의 어떤 부분에서 요청된 작업을 처리할 수 없는 것'이라는 의미로 생각해보자.

[p.404: 16행]
따라서 유효한 상태valid state 생성자는 완료됐고 소멸자에는 아직 진입하지 않았다는 뜻으로 해석될 수 있다.
->
따라서 유효한 상태valid state란, 생성자는 완료됐고 소멸자에는 아직 진입하지 않았다는 뜻으로 해석될 수 있다.

[p.408: 아래에서 15행]
클래스 X의 생성자는 파일 획득을 절대로 완료하지 않아야 하지만, 뮤텍스는 획득할 수도 있다(또는 뮤텍스는 안 되는데 파일은 된다든지, 아니면 둘 다 불가능하든지).
->
클래스 X의 생성자는 파일만 획득하고 뮤텍스는 획득하지 않은 채로 완료해서는 안된다(또는 뮤텍스만 획득하고 파일은 획득하지 않거나, 아니면 둘 다 획득하지 않은 채로).

[p.409: 13행]
사람들은 예외 이후의 마무리를 담당하는 임의의 코드를 작성하는 데 필요한 언어 구조를 거듭해서 '최종적으로' 개발해 왔다.
->
사람들은 예외 이후의 마무리를 담당하는 임의의 코드를 작성할 수 있도록 'Finally' 언어 구조를 거듭해서 개발해 왔다.

[p.412: 1행]
런타임 단정의 경우에는 던지기, 종료, 무시 중에서
->
런타임 단정의 경우에는 예외 던지기, 종료, 무시 중에서

[p.427: 아래에서 3행]
체크 같은 본적인 예외 안정성을
->
체크 같은 기본적인 예외 안정성을

[p.433: 19행]
예외로 인한 스팩 풀기 도중에
->
예외로 인한 스택 풀기 도중에

[p.435: 14행]
할당된 vector의 용량이 할당된 vector를 보관하기에
->
할당된 vector의 용량이 대입될 vector를 보관하기에

[p.437: 22행]
uninitializ ed_move
->
uninitialized_move

[p.446: 12행]
인자 독립적 탐색(14.2.3절)이 있다.
-> 인자 독립적 탐색(14.2.4절)이 있다.

[p.449: 아래에서 11행]
14.2.4 인자 독립적 탐색

->
14.2.4 인자 의존적 탐색

[p.449: 아래에서 10행]
사용자 정의 타입 X의 인자를 받아들이는 함수는 종종 동일한 네임스페이스 안에서 X로 정의된다.
->
사용자 정의 타입 X의 인자를 받아들이는 함수는 종종 X와 동일한 네임스페이스 안에서 정의된다.

[p.449: 아래에서 9행]
따라서 어떤 함수가 이용 상황 속에서 발견되지 않는다면 네임스페이스에서 그것의 인자를 찾아볼 수 있다.
->
따라서 어떤 함수가 이용 상황 속에서 발견되지 않는다면 함수의 인자가 정의된 네임스페이스를 찾아볼 수 있다.

[p.450: 5행]
인자 독립적 탐색
->
인자 의존적 탐색

[p.451: 6행]
인자 독립적 탐색에 대한 규칙이
->
인자 의존적 탐색에 대한 규칙이

[p.451: 12행]
인자 독립적 탐색은
->
인자 의존적 탐색은

[p.454: 1행]
14.5.5
->
14.4.5

[p.455: 5행]
언어 규칙상 main()이 전역 함수여야 하기 때문에 드라이버는 네임스페이스 안으로 완전히 넣을

수 있다.
->
언어 규칙상 main()이 전역 함수여야 하기 때문에 드라이버는 하나의 네임스페이스 안으로 완전히 넣을 수 없다.

[p.459: 4행]
개별적인 구현 네임스페이스를 사용하기로 결정했다면 설계가 사용자에게 다르게 보이지는 않을 것이다.
->
개별적인 구현 네임스페이스를 사용하기로 결정했더라도 설계가 사용자에게 다르게 보이지는 않을 것이다.

[p.460: 아래에서 16행]
f(1)의
->
f1()의

전역 이름은 전역 유효 범위에서 접근 가능하게 바뀐 이름보다 우선권을 갖지 않는다.
->
전역 이름은 전역 유효 범위에서 접근 가능하게 바뀐 네임스페이스의 이름보다 우선 순위가 높지 않다.

[p.468: 14행]
nam espace
->
namespace

[p.479: 아래에서 9행]
함수나 데이터 정의가 ve 때문에 포함된 파일의 뒤에는 .cpp가 붙는다.
->
함수나 데이터 정의가 포함된 파일의 뒤에는 .cpp가 붙는다.

[p.481: 3행]
ODR의 취지는 공통 소스 파일과 다른 해석 단위 안에 클래스 정의를 포함시킬 수 있게 하자는 것이다.
->
ODR의 취지는 하나의 공통 소스 파일에서 파생된 다른 해석 단위들이 하나의 클래스 정의를 갖도록 하자는 것이다.

[p.481: 아래에서 9행]
S가 멤버 이름이 다른
->
S2가 멤버 이름이 다른

[p.481: 아래에서 2행]
여기서는 S의 두 정의가
->
여기서는 S3의 두 정의가

[p.487: 아래에서 8행]
// dc.h 에서는 군더더기
->
// 군더더기, dc.h 에 있음

[p.489: 4행]
// dc.h 에서는 군더더기
->
// 군더더기, dc.h 에 있음

[p.489: 아래에서 4행]
각각의 논리적 모듈이 자신에 제공하는 기능을 제공하는 자체적인 헤더를 가진다
->
각각의 논리적 모듈이 자신이 제공하는 기능을 정의하는 자체적인 헤더를 가진다

[p.494: 4행]
여러 개의 함수를 동시에 살펴보기가 불편하다면
->
여러 개의 파일을 동시에 살펴보기가 불편하다면

[p.499: 아래에서 8행]
그것의 함수를 구현하는 소스 파일 안에 헤더를 #include 한다.
->
헤더 안에 선언한 함수를 구현하는 소스 파일 안에 그 헤더를 #include 한다.

[p.499: 아래에서 5행]
헤더 안에서 인라인이 아닌 함수는 피한다.
->
헤더 안에서 인라인이 아닌 함수의 정의는 피한다.

[p.506: 아래에서 11행]

호출되면 m==mm

->

m=mm

[p.511: 아래에서 9행]

1BC(year==1)

->

1BC(year==-1)

[p.524: 2행]

소속된 클래스의

->

소속된 클래스는

[p.539: 17행]

17.2.2 소멸자와 생성자

->

17.2.2 소멸자와 자원

[p.539: 아래에서 15행]

'보완 관계complement'(11.1.2절)란 의미로

->

'보수 관계complement'(11.1.2절)란 의미로

[p.558: 6행]

여기서 name은 n의 사본으로 초기화된다. 반면 address는 우선 빈 문자열로 초기화되고, 이어서 a의 사본이 대입된다.

->

삭제

[p.570: 아래에서 16행]

복사생성자는 우변 값을 받아드리는 반면,

->

복사생성자는 좌변 값을 받아드리는 반면,

[p.593: 맨 마지막 행]

rea()

->
real()

[p.595: 아래에서 8행]
있다며
->
있다면

[p.604: 아래에서 2행]
값을 소멸시키는('축소손실')
->
값을 파괴시키는("축소손실") 변환은

[p.611: 아래에서 12, 13행]
후위형 ++와 –
->
후위형 ++와 --

[p.669: 아래에서 6행]
[8] 가상함수를 가진 클래스는 가상 소멸자를 갖지 말아야 한다.
->
[8] 가상함수를 가진 클래스는 가상 소멸자를 가져야 한다.

[p.701: 아래에서 11행]
RTⅡ
->
RTTI

[p.722: 아래에서 17행]
RⅡT
->
RTTI

[p.731: 7행('S' 단문자와 :(이니셜라이져)가 빠져 있음)
tring<C>::String() // String<C>의 생성자
sz{0}, ptr{ch} // 짧은 문자열: ch를 가리킨다.
->
String<C>::String() // String<C>의 생성자
:sz{0}, ptr{ch} // 짧은 문자열: ch를 가리킨다.

[p.753: 아래에서 7행]
"대체 오류는 실패가 아니다"
-> "대체 실패는 오류가 아니다"

[p.929: 5행]
이 때문에 C 표준 라이브러리 qsort()는 비교 함수를 < 같이 고정된 뭔가가 아닌 인자로 받아들인다.
->
이 때문에 C 표준 라이브러리의 qsort()는 정렬 기준으로 < 같이 고정된 연산자를 사용하지 않고 비교 함수를 인자로 받아들인다.

[p.929: 6행]
반면 각각의 비교를 위한 함수 호출에 의해 발생되는 오버헤드 때문에 qsort()는 추가적인 라이브러리 구축을 위한 구성 요소가 될 수밖에 없다.
->
반면 각각의 비교를 위한 함수 호출에 의해 발생되는 오버헤드 때문에 qsort()는 추가적인 라이브러리 구축을 위한 구성 요소로서의 가치가 떨어진다.

[p.929: 11행]
이 때문에 사용자는 대안을 찾을 수 있다.
->
이 때문에 사용자는 대안을 찾을 수 밖에 없다.

[p.930: 6행]
p는 반복자를 가리키는 포인터
->
p는 반복자 혹은 포인터

[p.930: 아래에서 13행]
이 절의 나머지 부분은 함수에 의해 그룹으로 묶여진 헤더의 리스트에 간단한 설명을 싣고
->
이 절의 나머지 부분은 기능에 따라 그룹으로 묶여진 헤더의 리스트에 간단한 설명을 싣고

[p.930: 아래에서 1행]
이중 연결 오류
->
이중 연결 리스트

[p.933: 맨 위의 표]
현지화 좌측 locale, clocale, codecvt 색상+폰트 변경

[p.933: 2~4행]
테이블 포맷이 다른 테이블들과 다릅니다.

[p.933: 아래에서 11행]
<csetjmp>
->
<csetjmp>

[p.942: 7행]
asser t
->
assert

[p.943: 17행]
cat은 an error_category이고
->
cat은 error_category이고

[p.943: 19행]
ec = {n, g eneric_category}
->
ec = {n, generic_category}

[p.943: 아래에서 4행]
ec나 e2 중 한쪽
->
ec나 ec2 중 한쪽

[p.957: 아래에서 2행]
좀 더 긴 string의 경우에는 원수들이
->
좀 더 긴 string의 경우에는 원소들이

[p.959: 9행]
cmp(x,y)는 !cmp(x,y)의 뜻이다.
->

cmp(x,y)는 !cmp(y,x)의 뜻이다.

[p.963: 16행]
데이터 구조에 대해 const는 '저렴한' 편이다. O(n)은 '비싼' 편이고, O(log(n))는 '상당히 저렴한' 편이다.
->
데이터 구조에 대해 const는 '저렴한' 편이다. O(n)은 '비싼' 편이고, O(log(n))는 '웬만큼 저렴한' 편이다.

[p.966: 15행]
원하 않는 경우에는
->
원하지 않는 경우에는

[p.968: 아래에서 13, 11행]
c의 마지막 i번째 원소에 대한 참조자
->
c의 i번째 원소에 대한 참조자

[p.969: 14행]
추가로 list와 queue는
->
추가로 list와 deque는

[p.975: 5행]
원소를 핸들과 별도로 저장하는 방식의 데이터 구조(기본 제공 배열이나 array등)와 비교할 때
->
원소를 핸들과 별도로 저장하지 않는 방식의 데이터 구조(기본 제공 배열이나 array등)와 비교할 때

[p.977: 아래에서 18행]
p는 lst나 lst.end()의 원소를 가리킨다.
->
p는 lst의 원소나 lst.end()를 가리킨다.

[p.978: 15~17행]
lst2의 p 뒤에 접합한다.
->
lst2를 p 뒤에 접합한다.

p2의 p 뒤에 접합한다.
->
p2를 p 뒤에 접합한다.

[b:e)의 p 뒤에 접합한다.
->
[b:e)를 p 뒤에 접합한다.

[p.989: 13행]
c의 버킷 카운를
->
c의 버킷 카운트를

[p.990: 8행]
즉, 동일한 해시 값에 대해 많은 키들이 매핑되게 한다는 것이다.
->
즉, 많은 키들을 동일한 해시 값에 매핑되게 한다는 것이다.

[p.995: 14행]
배타적 or를 이용하는 원소에 대한 표준 해시 함수를 조합해서 얻어진 해시 함수는 많은 경우 훌륭하다.
->
원소에 대한 표준 해시 함수들을 배타적 or로 조합해서 얻어진 해시 함수는 많은 경우 훌륭하다.

[p.997: 아래에서 8행]
STL의 나머지 부분은 31장과 32장에서 소개된다.
->
STL의 나머지 부분은 31장과 33장에서 소개한다.

[p.998: 아래에서 6행]
static_assert(Sorable
->
static_assert(Sortable

[p.999: 6행]
두 번째 오류는 표현 불가능하게 되며,
->
두 번째 오류는 컴파일을 통과할 수 없고

[p.1003: 맨 위의 표]
all_of, any_of, none_of 색상+폰트 변경

[p.1005: 아래에서 8행]
p=search(b.e,b2,e2,f)
->
p=search(b,e,b2,e2,f)

[p.1009: 아래에서 16행]
p=remove_if(b,e,f)　　[b:p)가 !(*q)를 만족하는 원소
->
p=remove_if(b,e,f)　　[b:p)가 !f(*q)를 만족하는 원소

[p.1011: 아래에서 20, 18, 10행]
사선 편집
->
사전 편집 순서

[p.1014: 아래에서 3행]
때로는 정렬된 시퀀스의 첫 번째 원소만 필요한 경우가 있다. 이런 경우에는 순서의 첫 번째 부분을 얻기 위해
->
때로는 정렬된 시퀀스의 앞 부분 몇 개의 원소만 필요한 경우가 있다. 이런 경우에는 순서의 앞 부분을 얻기 위해

[p.1014: 19, 21행]
난수 접근 반복자
->
임의 접근 반복자

[p.1015: 1행]
여기서 N은 출력 시퀀스의 원소 개수에서 가장 적은 쪽이자 입력 시퀀스의 원소 개수다.
->
여기서 N은 출력 시퀀스의 원소 개수와 입력 시퀀스의 원소 개수 중에 작은 쪽이다.

[p.1015: 14행]
난수 접근 반복자
->

임의 접근 반복자

[p.1015: 19행]
nth_element() 알고리즘은 N번째 원소보다 작은 것으로 비교되는 원소가 시퀀스에서 뒤에 배치되지 않는 N번째 원소의 적절한 위치를 구하는 데 필요한 만큼만 정렬한다.
->
nth_element() 알고리즘은 N번째 원소보다 작은 것으로 비교되는 원소가 시퀀스에서 N번째 원소 앞에 배치되도록 하는 데까지만 정렬한다.

[p.1015: 아래에서 6행]
nth_element()는 단지 n번째 원소보다 작기만 하면 되지, n 앞의 원소를 반드시 정렬하지 않아도 된다는 점에서 partial_sort()와 차이가 있다.
->
nth_element()는 n번째 원소보다 앞의 원소들이 n번째 원소보다 작기만 하면 되지, 반드시 정렬되지 않아도 된다는 점에서 partial_sort()와 차이가 있다.

[p.1017: 2, 3행]
binar y_search
->
binary_search

[p.1018: 2행]
난수 접근 반복자
->
임의 접근 반복자

[p.1021: 4~5행]
사선 편집식 비교
->
사전 편집 순서 비교

[p.1025: 아래에서 6행]
31장에서는 STL 반복자와 유틸리티, 그리고
->
33장에서는 STL 반복자와 유틸리티, 그리고

[p.1026: 15행]
일부는 난수 접근 반복자만이 <을 지원하기 때문이다.
->

일부는 임의 접근 반복자만이 <을 지원하기 때문이다.

[p.1027: 아래에서 1행]

==, !=, <, <=, >, >= 를 이용해서 양방향 반복자를

->

==, !=, <, <=, >, >= 를 이용해서 임의 접근 반복자를

[p.1028: 1행]

양방향 반복자는 vector에서 제공하는 반복자의 일종이다(31.4절).

->

임의 접근 반복자는 vector에서 제공하는 반복자의 일종이다(31.4절).

[p.1039: 아래에서 14행]

_1, _2, _3같은 임시 이름을 사용하기 위해 args에 있는 임시 이름을 arg2의 인자로 대체함으로써 args3가 얻어진 경우

->

args에서 사용되는 _1, _2, _3 같은 임시 이름에 대해, 이런 임시 이름을 args2의 인자로 대체하여 args3가 얻어진 경우

[p.1040: 9행]

특이해 보이는 -1이라는 바인더 인자는

->

특이해 보이는 _1이라는 바인더 인자는

[p.1042: 15행]

g는 f의 연산자 타입으로 실행될 수 있는

->

g는 f의 인자와 같은 타입의 인자로 호출할 수 있는

[p.1054: 아래에서 10행]

vector<boo>은 할당자를 갖고 있으며,

->

vector<bool>은 할당자를 갖고 있으며,

[p.1144: 아래에서 8행]

[5] 절제해야 한다. 정규 표현식은 아차하면 읽기 전용 언어가 될 수 있다(37.1.1절).

->

[5] 절제해야 한다. 정규 표현식은 아차하면 쓰기 전용 언어가 될 수 있다(37.1.1절).

[p.1233: 14, 15행]

conver ter

->

converter

[p.1347: 아래에서 14행]

strftim() 함수는

->

strftime() 함수는