**Effective Modern C++**

C++11과 C++14를 효과적으로 사용한다.

용어와 관례

# 좌측값(l-value)와 우측값(r-value)

좌측값(왼값), 우측값(오른값)이라고 불린다. 일단 정의부터 살펴보자.

## 정의

Lvalue: Lvalue 는 객체를 참조하는 표현식이다. **(참조 가능한)메모리 위치를 가지고 있다.** [The C Programming Language - Kernighan and Ritchie].

int temp;

int& lvalue = temp;

Rvalue: C++ 표준: "모든 표현식은 Lvalue 거나 Rvalue이다. 고로, Rvalue 는 Lvalue 가 아닌 모든 것이다.” 정확하게 말하자면, 임시 메모리 영역을 가지는 표현식이다.

int&& rvalue = 5;

## l-value vs. r-value

어떠한 경우에 어떤 값을 갖게 되는지 코드에서 살펴본다.

#### 숫자, 문자 = r-value

int b = 5;

char c = 'a';

5 = 3; ***//error***

#### 나열자(enumeration) = r-value

**enum** skill{ fire, ice };

int k = fire;

fire = 5; *//error: 식이 수정할 수 있는 l-value이어야 합니다.*

#### 사칙 연산 결과 = r-value

int c = b + 5; *//b+5 = r-value*

b + 5 = c; *//error: 2번과 동일*

#### &연산자 = l-value

int &d = 5; *//error: d = l-value는 맞지만 이의 초기값 역시 l-value이어야 한다.*

&8 = 1; *//error: &는 l-value앞에서만 사용할 수 있다.*

int &e = b; *//fine.*

int n, \*p;

p = &n; *// fine. p는 항상 어떤 객체를 가리키므로 l-value*

*//&는 l-value앞에만 붙지만, 반환 결과(&n)는 항상 r-value이다.*

&n = p; *// error: &n의 결과 = r-value*

#### &와 대조적으로, \*는 그 결과로 l-value를 만들어 준다.

int a[100];

int \*p = a;

\*p = 3; *// Fine.*

*// 그 결과가 l-value이긴 하지만, 피 연산자는 r-value가 될 수도 있다.*

\*(p + 1) = 4; *// Fine. (p+1) 는 r-value*

#### pre-increment 연산자 = l-value

int nCount = 0; *// nCount는 영속 객체 = l-value*

++nCount; *//이 표현식은 l-value 이다. 왜냐하면 결과가 nCount 객체이기 때문이다.*

*//이것이 l-value인 것을 증명하기 위해, 다음 연산을 할수 있다*

++nCount = 5; *// Fine. nCount 는 5 이다.*

#### 함수에서 &값 반환 = l-value

*//함수 호출을 l-value 로 만들기 위해 참조를 반환*

*//일반적인 경우에는 r-value를 반환한다.*

int& GetBig(int& a, int& b) {

**return** (a > b ? **a** : b);

}

void test1() {

int i = 10, j = 50;

GetBig(i, j) \*= 5;

*// 여기서, j = 250. GetBig()은 j의 참조를 리턴한다.*

*// 그리고 그것에 5가 곱해진 것으로 저장된다.*

}

#### r-value에 묶인 참조 = l-value

참조는 그냥 이름이다. 그래서 r-value에 묶인 참조 그 자체는 l-value 이다. 아래 예제에서는 l-value인 (int&)a를 r-value로 묶어서 반환한 것이다.

*//함수 호출을 r-value로 만들기 위해 int를 리턴*

int GetBig(int& a, int& b) {

**return** (a > b ? **a** : b);

}

void test2() {

int i = 10, j = 50;

*//const\_l-value에 r-value를 바인딩이 가능합니다.*

*//'big'를 GetBig()의 리턴 값(r-value)에 대해 l-value로 바인딩한다.*

**const** int& big = GetBig(i, j);

int& big2 = GetBig(i, j); *// Error: big2 가 const가 아니므로*

*// temporary 는 non-const reference 에 바인드 불가.*

}

#### r-value는 임시적이며 메모리영역을 가리킬 필요가 없다.

그러나 어떤 경우에는 메모리를 가리킬 수 있다. 하지만 이런 임시 값에 대해서 작업하는 것은 권장되지 않는다

char\* fun() { **return** "Hellow"; }

int main() {

char\* q = fun();

q[0] = 'h'; *// 예외 발생, fun()이 임시 메모리(r-value)에 접근*

}

#### post-increment 연산자 = r-value

*//전위 증가와 다르다는 점을 기억한다.*

int nCount = 0; *// nCount = l-value*

nCount++; *// 이 표현식은 r-value이다. 객체의 값을 복사하고,*

*//변경한 후 임시 복사를 반환하기 때문이다.*

*// 이것이 r-value라는것을 증명하기 위해, 다음 연산을 할 수 있다.*

nCount++ = 5; *//Error*

#### 함수에서 포인터를 반환 = l-value

*//반환 값 = r-value*

int NextVal\_1(int\* p) { **return** \*(p + 1); }

*//포인터 반환 값 = l-value*

int\* NextVal\_2(int\* p) { **return** (p + 1); }

int test2()

{

int a[] = { 1, 2, 3, 4, 5 };

NextVal\_1(a) = 9; *//Error. 왼쪽은 l-value이어야 합니다.*

\*NextVal\_2(a) = 9; *//Fine.*

}

#### 예외) l-value와 r-value는 모두 변경 가능 혹은 변경 불가능일 수 있다.

**const** string Fun() { **return** "Hellow World"; } *// const r-value*

string strName("Hello"); *// modifiable l-value*

**const** string strConstName("Hello"); *// const l-value*

string JunkFunction() { **return** "Hellow World"; */\*catch this properly\*/* } *//modifiable r-value*

## l-value, r-value 판별법

우측값이니 좌측값의 여부는 어떻게 판별하는 것일까?

### C++에서의 표현식 속성

C++에서 하나의 표현식은 **독립적인** 두 가지 속성으로 규정되는데, **형식과 값 범주(value category)**이다. **형식**이란, 변수 앞에 명시된 타입 그 자체이며, **값 범주**는 실제로 이 변수의 우측값인지 좌측값인지의 여부를 말한다. 이러한 규정의 중요한 특징은 형식과 값 범주는 독립적이다. 즉, 무관하다는 것이다.

조금 풀어서 말하자면, 형식에는 int, int&, int&&등 여러 가지가 올 수 있다. 하지만 이것의 값 범주, 실제 값의 의미는 좌측값 혹은 우측값이라는 것이다. 그리고 이 형식과 좌측값인지 우측값인지의 여부는 같을 수도 있지만 다를 수도 있다는 것이다.

분명 좌측값을 &, 우측값을 &&로 형식을 표현하지 않았던가? 형식으로 좌측값, 우측값을 결정할 수 없다면 어떻게 이를 판별하는가?

### 유용한 판별법

그 표현식의 주소를 취할 수 있는지 보는 것이다. 그 식이 넘어가면 무효한 표현식이라면 우측값이며, 그렇지 않고 그 식이 넘어가도 참조할 수 있다면 좌측값이다. 자세한 판별을 형식과 연관지어 알아보자.

void shot(Archer&& target)

{

Archer&& temp = std::move(target);

temp.die();

}

1. 위에서 매개변수로 있는 Archer&& target 을 보자. 이는 &&이므로 우측값인가? 이렇게 판별할 수 없다는 것이다. 형식은 Archer&& 이지만 이는 우측값이 아니다. target 은 엄연히 좌측값이다. 왜냐하면 이 함수 내에서는 어디서든 target 을 참조할 수 있지 않은가? 여기서 매개변수는 타입이 &이든 &&이든 좌측값이라는 말이 성립된다.  
2. Archer&& temp 는 무엇인가? 형식은 Archer&&이다. 하지만 temp는 이 표현식이 끝난 후에도 사용할 수 있다. temp는 좌측값인 셈이다.  
3. target 이 좌측값이라는 또 하나의 증거는 std::move 이다. 만약 이것이 우측값이라면 move를 사용할 필요가 없다.   
4. std::move 가 없다면 이 식은 컴파일되지 않는다. 좌측값을 Archer&&에 넣으려고 하니 말이다.

# 참조 한정자(Ref-qualifier)

객체가 좌측값이냐 우측값이냐에 따라 호출될 함수를 구분하기 위해 등장한다. const-qualifier를 사용한 것처럼, 함수 끝부분에서 선언한다.

void bar() **&**

const한정자는 객체가 상수냐 아니냐에 따라 호출 여부가 갈린다. 참조 한정자도 마찬가지다.

## 정의

객체가 좌측값이냐 우측값이냐에 따라 함수를 오버로드할 수 있게 해주는 한정자. 즉, &이 붙으면 해당 함수는 좌측값인 객체에서만 사용할 수 있으며, &&은 우측값인 객체에서만 사용할 수 있다.

## 사용법

다음과 같이 사용할 수 있다.

struct foo

{

public:

std::string mStr = "Resources";

public:

const std::string& f() const { return mStr; }

std::string&& f() { return std::move(mStr); }

};

int main()

{

// std::string&& f() 호출

// 우측값 반환 이후, obj1의 리소스(mStr)가 이동되어 사라져버림

foo obj1;

auto obj2 = obj1.f();

std::cout << obj1.mStr << "\t" << obj2 << std::endl;

// const std::string& f() const 호출

// 상수 좌측값 반환

const foo obj3;

auto obj4 = obj3.f();

std::cout << obj3.mStr << "\t" << obj4 << std::endl;

// std::string&& f() 호출

// 임시객체이므로 어차피 파괴될 거, 리소스가 이동되어도 문제가 없다

auto obj5 = foo{}.f();

}

# r-value-reference의 이해

출처: <http://cpplover.blogspot.jp/2009/11/rvalue-reference_23.html>

### lvalue, rvalue

C에서의 lvalue, rvalue의 차이는 대입 연산자의 오른쪽에 있는지 왼쪽에 있는지의 차이뿐이었다. 즉 left hand value, right hand value.

하지만 C++에서의 lvalue, rvalue는 그와는 완전히 다른 개념이다.

* lvalue는 명시적으로 실체가 있는 명명된 객체.
* rvalue는 일시적으로 생성된 무명의 객체.

### (l-value)reference

c++11이전의 reference는 c++11의 lvalue reference를 뜻한다.

**struct** S {};

void func (S &) {}

void func\_const (S **const** &) {}

*// =* *void func\_const (const S&) {}*

int main()

{

S obj;

func(obj); *// 1. OK*

func(S()); *// 2. Error*

*//vc++에서는 2번을 1번처럼 처리합니다. 여하튼 이건 r-value객체입니다.*

func\_const(S()); *// 3. OK*

}

1. lvalue 이므로 문제가 없다.
2. rvalue를 전달하기 때문에 에러
3. const reference는 rvalue를 참조할 수 있기 때문에 문제가 없다.

단, VC++의 독자 확장을 쓰면 (/Za) 2도 에러가 나지 않기 때문에 주의할 것(주석).

여기서 눈 여겨 볼 것이 3번 이다. 원래 rvalue(S())인 것을 lvalue reference에서 참조하는 것이다. 이를 가능하게 해주는 키워드는 const이다. 임시 생성 객체를 넘긴 상태에서 그 값을 참조할 수는 없지만, 상수 값으로 이용할 수는 있다는 말이다. 하지만 복사하는 것이 아니다. 그럼 어디서 가져올까? 여기서 ***rvalue reference***가 나오게 된다.

### r-value reference

**struct** S {};

int main()

{

S obj;

*// lvalue reference*

S & lr1 = obj; *// 1. OK*

S & lr2 = S(); *// 2. Error*

*// rvalue reference*

S && rr1 = obj; *// 3. Error*

S && rr2 = S(); *// 4. OK*

}

rvalue reference는 & 를 2개 사용한다.

1. obj가 lvalue이기 때문에 문제없음.
2. S()가 rvalue이기 때문에 에러.
3. obj는 lvalue이기 때문에 에러.
4. S()는 rvalue이기 때문에 문제없음.

### rvalue reference의 존재 이유

rvalue reference라는 것은 단순히 이것뿐이다. 이름 그대로 rvalue에 대한 참조이다. 이것만으로는 왜 rvalue가 존재하는지 알기 어렵다. const lvalue reference로도 rvalue를 참조할 수 있으니까 없어도 되는 것 아닌가? 라고 생각할 수 있다. rvalue 객체는 이름이 없기 때문에, 참조할 수 없게 되는 시점에서 자동으로 소멸된다. 자동으로 소멸하는 것을 참조해서 사용한다고 해도 아무 쓸모도 없는 것이다. const lvalue reference를 쓰지 않고도 rvalue를 참조할 수 있는 것이 왜 그렇게 유용한 것일까?

### Move semantics

#### copy semantics(복사 의미론)

어떤 객체를 다른 객체로 복사하는 것. 객체 복사는 복사 생성자와 복사 연산자가 담당한다는 것을 이미 알고 있다. 제대로 복사하기 위해서는 깊은 복사를 해야 하고 이는 사용자가 직접 정의해야 한다.

문제는 깊은 복사 시, 몇 개가 들어있건 복사할 대상에 메모리를 동일하게 할당하여 하나씩 복사한 다음, 기존 객체를 해제하는 과정이 너무 비효율적이라는 것이다.

**class** **DummyBuffer**

{

**public**:

DummyBuffer() {

ptr = **new** char[1000];

}

*// 복사생성자*

DummyBuffer(DummyBuffer **const** & r) {

ptr = **new** char[1000];

memcpy(r.ptr, ptr, 1000);

}

~DummyBuffer() {

**delete**[] ptr;

}

**private**:

char \* ptr;

};

이 클래스는 방금 위에서 언급한 복사 의미론이다. 생성자와 소멸자의 실행이 느리다. 만약 복사 생성자의 포인터만 바꿔 치기 한다면 성능 향상은 있겠지만, 원본 객체를 사용할 수 없게 되기 때문에 그렇게 해서는 안 된다.

#### move semantics(이동 의미론)

하지만 깊은 복사를 원하는 것이 아닌 대입을 위해서 이동만을 필요로 한다면 포인터만 바꿔도 될 것이다. 이를 효율적으로 하는 방법을 제시한다.

**struct** S {};

S func()

{

**return** S();

}

int main()

{

S obj(func()); *// 1.*

S tmp;

S obj2(tmp); *// 2.*

*// 이후 tmp는 사용하지 않는다.*

}

1번의 경우는 func()의 반환 값이 rvalue이므로 안전하게 포인터를 바꿔 치기 할 수 있다. 그리고 2번의 경우는 tmp는 더 이상 사용하지 않기 때문에 포인터를 바꿔 치기 해도 상관없다.

문제는 어떻게 이것을 구현하는가이다.

여기서 rvalue reference의 진가가 발휘된다. rvalue는 그 객체의 포인터를 가로채도 문제가 없다는 데 있다.

### Move constructor

위 예제에서 1번의 복사 생성자를 실행할 때(반환 값이 r-value인 경우) 포인터를 가로채기 위해서 rvalue reference의 복사 생성자를 추가해 보겠다.

**class** **DummyBuffer**

{

**public**:

DummyBuffer() {

ptr = **new** char[1000];

}

*// 복사생성자*

DummyBuffer(DummyBuffer **const** & r) {

ptr = **new** char[1000];

memcpy(r.ptr, ptr, 1000);

}

*// move 복사생성자*

DummyBuffer(DummyBuffer && r) {

ptr = r.ptr;

r.ptr = **nullptr**;

}

~DummyBuffer() {

**delete**[] ptr;

}

**private**:

char \* ptr;

};

move 복사 생성자는 원본의 포인터를 바꿔 치기 한다.

#### 이것이 합당한 이유

이 과정에서 r-value-reference의 동작 원리를 이해할 수 있다.  
기본 복사 생성자에서 포인터 바꿔 치기가 안 되는 이유는 원본 훼손 때문이었는데, 여기서는 왜 가능한 것일까? 그래도 원본의 것이니 훼손되지 않을까? 이 r-value는 일단 저장이라는 개념이 없다고 생각해야 한다. 잠깐 담아두는 것이지 존재하지 않는다는 것이다. 그럼 r-value의 것을 바꿔 치든 없애든 상관없음이 보여진다. 하지만 어떻게 가능한가? 복사를 한 적도 없고 원본도 아닌 것이 어떻게 원본의 정보를 가지고 있을까? 이해하기 이상한 이 과정이 r-value라면 가능하다. 복사하지 않았지만 고스란히 value(기존 r-value, 혹은 l-value에서 const로 바꾸면 된다는 것을 위에서 언급함)를 갖게 되는 이건 진짜 기이한 현상이다.

#### 이동 연산에서 r-value-reference의 의미

***(l-value) 🡪 r-value 🡪 && 🡪 이 안에는 모든 정보가 들어있음  
이 과정은 복사하지 않고 값을 가져오는 방법이 된다.***

여기서 원본의 포인터에 nullptr을 넣어주지 않으면 필요 없는 객체에서 소멸자가 호출되면서 원본을 지워버리므로 주의할 것.

### lvalue의 move

2번의 경우(반환 값이 l-value인 경우)는 어떻게 해결하면 될까? move 생성자를 구현했지만 2번의 경우는 lvalue이므로 move 생성자가 불리지 않는다.

tmp 변수는 이후로는 사용하지 않지만, 컴파일러는 프로그래머의 의도를 알 수 없으므로 당연히 일반 복사 생성자를 실행하게 될 것이다.

tmp 변수를 lvalue가 아닌 rvalue로 넘겨줄 수 있다면 move 복사 생성자를 불리게 할 수 있다.

S tmp;

S b(**static\_cast**<S &&>(tmp));

강제적으로 이렇게 static\_cast시켜주면 tmp를 rvalue로 넘기는 게 가능하다.

이것을 좀 더 간단하게 표현하는 방법이 std::move()이다.

S tmp;

S b(std::move(tmp));

move함수는 다음과 같이 설계된다.

**namespace** std {

**template** <**class** **T**>

**inline**

**typename** std::remove\_reference<T>::type&&

move(T&& t) {

**return** **static\_cast**< std::remove\_reference<T>::type&& >(t) ;

}

}

std::move()는 이것 뿐이다. 단순히 lvalue를 rvalue로 static\_cast 시키는 것이다.

### 영리한 컴파일러의 경우

영리한 컴파일러일 경우에는 일반 복사 생성자도 move 복사 생성자도 불리지 않을 경우가 있다. 어떤 경우에는 실제로 객체를 복사하지 않고 원본 객체를 그대로 사용하는 경우도 있다.

(N3000 § 12.8 Copying class objects p19)

만약 위 코드를 컴파일 했을 때 생성자가 불리지 않는 경우는 이렇게 코딩 해 보는 것도 좋다.

#include *<utility>*

**class** **DummyBuffer**

{

**public**:

DummyBuffer() {

ptr = **new** char[1000];

}

DummyBuffer & DummyBuffer(DummyBuffer && r) {

**if** (**this** == &r) {

**return** \***this**;

}

**delete**[] ptr;

ptr = r.ptr;

r.ptr = **nullptr**;

**return** \***this**;

}

~DummyBuffer() {

**delete**[] ptr;

}

**private**:

char \* ptr;

};

int main()

{

DummyBuffer tmp;

DummyBuffer x;

x = std::move(tmp);

}

### 오버로드

rvalue reference, lvalue reference는 당연히 함수 overload가 가능하다.

#include *<iostream>*

**struct** S {};

void f(S & s) {

std::cout << "lvalue reference" << std::endl;

}

void f(S && s) {

std::cout << "rvalue reference" << std::endl;

}

int main() {

S obj;

f(obj); *// lvalue reference*

f(S()); *// rvalue reference*

}

하지만 템플릿으로 하나의 함수로 만들 수 있다.

**struct** S {};

**template** <**typename** T>

void f(T && t) {

}

int main() {

S obj;

f(obj); *// lvalue reference*

f(S()); *// rvalue reference*

}

이 코드는 컴파일 성공한다. 여기에는 특별한 규칙이 있어서 템플릿 함수의 인수를 rvalue로 했을 경우에 lvalue를 전달하면 자동으로 lvalue로 인식하는 규칙이다.

(§ 14.9.2.1 Deducing template arguments from a function call p3)

조금은 이상한 규칙이지만, 이렇게 되지 않으면 프로그래머는 lvalue, rvalue reference 코드를 따로 작성해야 하는 수고가 따른다. 하지만 이 규칙 때문에 rvalue reference 코드만 작성하면 문제없이 동작하게 된다.

하지만 여기에 한가지 문제점이 있다.

### Perfect forwarding

**struct** S {};

**template** <**typename** T>

void f(T && t) {

S obj(t);

}

f() 안에서 S를 복사하고 싶을 경우이다. 위의 내용을 읽었다면 여기서도 std::move()를 사용하고 싶을 것이다. 하지만 t가 lvalue인지 rvalue인지 알 수 없기 때문에 일반적으로는 사용할 수 없다. 어떻게 하면 효율적으로 move()를 사용할 수 있을까?

**struct** S {};

**template** <**typename** T>

void f(T && t) {

S obj(std::move(t));

*// 이후 t는 사용 불가*

}

int main() {

S obj;

f(obj); *// lvalue reference*

}

#### cast를 사용

**template** <**typename** T>

void f(T && t) {

T x(**static\_cast**<T &&>(t));

}

직관적인 방법이다. 하지만 cast를 사용하는 것은 귀찮기도 하고, 버그를 낳는 코드가 되기 쉽다. 이를 위해서 존재하는 것이 ***std::forward()***다.

#### std::forward()

**template** <**typename** T>

void f(T && t) {

T x(std::forward<T>(t));

}

std::forward()도 std::move()와 마찬가지로 본질은 cast이다. 단순화시키면 아래와 같다.

**namespace** std {

**template** <**class** **T**, **class** **U**,

**class** = **typename** enable\_if<

(is\_lvalue\_reference<T>::value ?

is\_lvalue\_reference<U>::**value** :

true) &&

is\_convertible<**typename** remove\_reference<U>::type\*,

**typename** remove\_reference<T>::type\*>::value

>::type>

**inline**

T&&

forward(U&& u)

{

**return** **static\_cast**<T&&>(u);

}

}

[미결] 무시무시하게 보이는 메타프로그래밍 코드이지만 본질적으로는 cast이다. T가 lvalue reference라면 U도 lvalue reference 라야 한다. 참조를 제거한 상태에서 U에서 T로 변환이 되어야 한다.   
이 두 가지 조건을 만족하지 못한 경우, std::forward()는 overload resolution의 대상에서 제외된다. 즉 컴파일 에러가 된다. 이것으로 전형적인 오타를 방지할 수 있게 된다.   
std::forward()는 템플리트 함수의 인수를 다른 함수의 인수에 그대로 전달하기 위한 것이다. 이것을 perfect forwarding이라고 한다.

### 마치면서

rvalue reference는 정말 단순한 것이다. 이름 그대로 rvalue의 참조에 지나지 않는다. std::move(), std::forward()도 단순한 cast일 뿐이다. std::move()는 lvalue를 move시키고 싶을 때 사용하고, std::forward()는 템플리트 함수의 인수를 그대로 다른 함수에 전달할 때 사용한다.

## 보편 참조(forward reference or universal reference)

보편적인 참조라는 뜻이다. ***즉, l-value도 될 수 있고, r-value가 될 수도 있는 참조형식***이라는 것이다. 한가지 예를 들어보겠다.

**template**<**typename** T>

void f(T&& arg)

{

}

위의 코드에서 ***T&&는 universal reference***이다. 즉, arg는 r-value reference가 될 수도 있고, l-value reference가 될 수도 있다. 위 함수를 아래와 같이 사용해보자.

int x = 27;

**const** int cx = x;

**const** int& rx = x;

f(x); *//x = l-value -> T&& = int&, l-value reference*

f(cx); *//cx = l-value -> T&& = const int&, l-value reference*

f(rx); *//rx = l-value -> T&& = const int&, l-value reference*

f(27); *//27 = r-value -> T&& = int&&, r-value reference*

이로써 template code 한 줄로 l-value reference도 r-value reference도 모두 수용 가능하게 되었다.

### 그렇다면 언제 universal reference가 되는 것일까?

**universal reference는 타입 추론이 발생하게 되는 경우에만 적용**이 된다.

**template**<**typename** T>

void f1(T&& arg) {}

void f2(int&& arg) {}

f1함수는 template함수로써 T에 대한 타입 추론이 발생하게 된다. 즉, 이럴 경우 T&&가 l-value reference인지 r-value reference인지 compile 타임에서 파악하게 된다.

f2함수는 이미 타입이 int형으로 정해져 있다. 이럴 경우 타입추론을 하지 않게 되니 int&&는 r-value reference가 되는 것이다(&&는 r-value라는 의미). f2함수는 l-value를 인자로 사용할 수 없다.

### 멤버 함수에서 타입 추론: T&&라면 모두 universal reference일까?

**template**<**typename** T>

**class** **Foo**

{

...

void Bar(T&& arg) { }

}

위와 같은 클래스의 Bar 멤버 함수에서 **T&&는 universal referenece가 아닌 r-value reference가 된다.** 왜일까? 객체 생성시 T는 정해지기 때문에 멤버 함수 안의 타입도 정해져 버리기 때문이다.

**template**<**typename** T>

**class** **Foo**

{

...

void Bar(T& arg) { }

void Bar(T&& arg) { }

}

**멤버 함수 overloading을 통해서 해결할 수 있다.** 그러나 overloading을 사용하는 방법은 엄청난 노동을 필요로 하는 경우가 발생한다. Bar멤버함수에 파라미터가 두 개 이상이라면? 우리는 파라미터의 ^2에 해당하는 멤버함수가 필요할 것이다. 이를 해결하기 위해서 다음과 같이 작성한다.

**template**<**typename** T>

**class** **Foo**

{

**public**:

**template**<**typename** T>

void Bar(T&& arg) { }

}

Bar함수 전에 template을 명시해주면 된다. 그러면 T&&는 universal reference가 된다. 그러나 여기서 알아야 할 것은 class에서 명시한 T와 멤버함수에서 명시한 T는 타입이 다를 수 있다는 것이다. 같은 T라도 타입이 똑같다고 컴파일러는 보장해주지 않는다는 것이다. 전적으로 사용자의 책임이 되는 것이다. **두 개 T에 대해서 같은 타입을 보장해주고 싶다면 타입검사가 필요할 것이다.**

**template**<**typename** T>

**class** **Foo**{

**public**:

**using** value\_type = T;

**template**<**typename** T>

void Bar(T&& arg){

**if** (**typeid**(value\_type) == **typeid**(T)) {

std::cout << "equal" << std::endl;

}

}

}

### 사용 예시

variadic template과 universal reference, std::forward를 사용하면 극강의 코드를 만들 수도 있다. 어떤 객체를 만드는 factory 함수를 만든다고 하자.

**template**<**typename** T, **typename**... Args>

T\* factory(Args&&... args){

**return** **new** T(std::forward<Args>(args)...);

}

나는 메모리풀을 만들면서 위의 코드(와 비슷하게 작성했다)로 인해 어떤 고민도 없이 객체 생성의 극강을 맛볼 수 있었다. 만약 variadic template과 universal reference가 없었다면 파라미터 복사와 파라미터에 대한 개수 제한이 발생했을 것이다.

여기서 std::forward는 넘겨받은 파라미터를 다음 함수로 넘겨줄 때 어떠한 복사도 없이 바로 넘겨주는 역할을 하고 있다. 그리고 forward는 넘겨받은 참조가 rvalue이면 rvalue로 취급해주고, lvalue이면 lvalue로 취급하여 준다. 이것을 Perfect forwarding이라고 한다.

[항목1] 템플릿 형식 연역

템플릿 형식 연역 규칙을 숙지하자.

# 형식 연역(type deduction)이란?

함수나 클래스를 인스턴스화하려면, 컴파일러는 해당 형식을 반드시 알아야만 한다. 하지만 사용자가 직접 지정해야만 하는 것은 아니다. 사용자가 타입을 지정하지 않았을 때, 컴파일러가 그 타입을 추론하는 과정이 *형식 연역*이다.

template<class T> void f([std::initializer\_list](http://en.cppreference.com/w/cpp/utility/initializer_list)<T>);

f({1, 2, 3});

위에서는 함수f를 호출하지만 타입을 명시하지 않는다. 하지만 이는 정상적으로 작동한다. 왜냐하면 컴파일러가 **T를 int로 연역하기 때문**이다.

# 템플릿 형식 연역 규칙

템플릿에서의 형식 연역은 C++98부터 존재한 가장 기본적인 형식 연역이다. 위의 코드처럼 T를 어떻게 int로 연역한 것일까? 규칙이 생성된 동기와 그 세부사항을 살펴본다.

## 규칙이 왜 필요하지?

템플릿은 가장 기본적인 인자 추론의 대상이다. T를 명시하지 않아도 T를 알아내라고 하고 싶다.

**template**<**typename** T>

void func(T param){

cout << param << endl;

}

넘긴 인자가 int이든 string이든 그 인자에 맞게 T를 결정할 것이다. 이것이 형식 연역이다. 하지만 컴파일러는 있는 그대로 연역할 수 없다.

**template**<**typename** T>

void func(ParamType*/\*나는 누굴까?\*/* param){

cout << param << endl;

}

int main(){

func(*/\*무엇이 들어갈까?\*/*);

}

컴파일러는 T 가 무엇인지 컴파일 시간에 알아내야 한다. func에 const int&타입의 변수를 인자로 넘긴다고 하자. 그러나 T와 paramtype는 동일하지 않다. **그러므로 T를 찾기 위한 근본적인 규칙**을 이해해야 한다.

**아래에서 언급할 모든 경우는 아래 2가지 규칙으로 작동**한다.

1. func의 인자(x)가 참조 형식(&)이면 참조를 없앤다.  
2. 패턴 부합 방식으로 대응시켜서 T의 형식을 결정한다.

### func의 인자(x)가 참조 형식(&)이면 참조를 없앤다.

T의 타입의 연역은 참조를 무시한다. int&, cons tint&든 일단 &는 제거한다. 그것이 T의 타입이다.

T의 연역이 참조를 제거한다고 해서 param의 타입도 참조가 제거된다고 생각하지는 않으리라 믿는다. param의 타입은 T&이 아닌가? T에 참조를 추가로 붙이면 연역 끝이다.

### 패턴 부합 방식으로 대응시켜서 T의 형식을 결정한다.

간단하다. const나 volatile같은 인자(x)의 타입을 그대로 유지시킨다는 의미다.

## [경우1] ParamType이 참조(&)나 포인터(\*)인 경우

T&, const T&, T\*일 경우이다.

**template**<**typename** T>

void func(T& param){

cout << param << endl;

}

int main(){

*//모두 어딘가에 저장되어 있는 l-value가 됩니다.*

int x = 7;

func(x); *//T:int, Param: int&*

**const** int cx = 8;

func(cx); *//T:int, Param: const int&*

**const** int &rx = cx; *//T:int, Param: const int&*

func(rx);

}

아래는 param앞에 const가 추가되었다. 하지만 위와 규칙이 같다. 그냥 연역된 T에 const만 더 붙이면 된다. **참조 매개변수 타입(T&)은 인자를 참조한다는 것을 보여주고 싶은 것이다.**

template<typename T>

void func(**const** T& param){

cout << param << endl;

}

int main(){

*//첫 번째 결과만 const가 추가되고 나머지는 그대로*

int x = 7;

func(x); *//T:int, Param: const int&*

**const** int cx = 8;

func(cx); *//T:int, Param: const int&*

**const** int &rx = cx; *//T:int, Param: const int&*

func(rx);

}

주소를 넘기면 어떻게 되는 것일까? 말할 것도 없다. 위의 규칙처럼 T는 &(\*)를 제거하고, param은 그에 \*를 붙이니 T\*이다.

**template**<**typename** T>

void func(T\* param){

cout << param << endl;

}

int main(){

int x = 7;

func(x); *//error: 주소가 필요합니다.*

func(&x); *//T:int, Param: int\**

**const** int \*px = &x;

func(px); *//T:int, Param: const int\**

}

## [경우2] ParamType이 보편 참조(&&)

위는 r-value를 참조할 때 사용하는 기호이며 그와도 연관이 깊다.

**template**<**typename** T>

void func(T&& param){

cout << param << endl;

}

int main(){

*//l-value든 r-value든 모두 허용합니다.*

int x = 7; *//x: l-value*

func(x); *//T:int, Param: int&*

func(&x); *//T:int, Param: int\**

**const** int \*px = &x; *//px: l-value*

func(px); *//T:int, Param: const int\**

**const** int& cx = x; *//cx: l-value*

func(cx); *//T:int, Param: const int&*

**const** int& rx = cx; *//rx: l-value*

func(rx); *//T:int, Param: const int&*

func(27); *//27: r-value, T:int, Param: int&&*

}

이것이 무엇인가? 굉장히 많은 것들이 있다. 정리하자면 l-value, r-value모두 추론(허용)할 수 있다. 이것이 ***보편참조***다. – [항목24] 참조

### 인자(x)가 왼값일 경우

T는 왼값으로 연역된다. 즉, T는 &형태이다. 그럼 param은 T& &&이 되는데 이에 대한 규칙은 이후에 다룬다.

### 인자(x)가 오른값일 경우

규칙1과 동일하다.

## [경우3] ParamType이 일반 객체

군더더기 없는 T자체. 값을 복사하여 전달(pass-by-value)한다고 말한다. 중요한 것은 경우1과는 다르게 객체 자체가 아닌 복사본을 보낸다.

**template**<**typename** T>

void func(T param){

cout << param << endl;

}

int main(){

int x = 7;

func(x); *//T:int, Param: int*

**const** int& cx = x;

func(cx); *//T:int, Param: int*

**const** int& rx = cx;

func(rx); *//T:int, Param: int*

}

const가 있던 없던 참조건 아니건 모두 int로 추론한다(volatile도 마찬가지로 무시). **왜냐하면 복사본이기 때문이다. 값 매개변수 타입(T)은 본체랑은 관련이 없는 것을 보여주고 싶은 것이다.**

### 포인터에 상수가 붙으면?

**template**<**typename** T>

void func(T param){

param[0] = 'J'; *//error: const인 변수에 할당할 수 없습니다.*

cout << param << endl;

}

int main(){

**const** char\* **const** ptr = "I like programming.";

func(ptr); *//T:char\* Param: const char[]*

}

**포인터 주소는 값으로 전달된다. 이제 포인터의 주소 값은 const가 아니다. 하지만 그 내부의 값은 여전히 const이므로** 위의 코드에서 컴파일 에러를 볼 수 있다. 무조건 const가 사라지는 것은 아니다. 핵심을 알자.

## 연역의 근본적인 핵심

인자를 매개변수의 타입에 적절하게 맞춰야 한다. 직관적으로는 쉽지만 그럴 수 없는 컴파일러는 이를 위한 규칙이 필요한 셈이다.

## 배열 인수(T[…])

배열도 인자로 넘길 수 있을까? 혹은 인자 추론될 수 있을까?

### 이게 뭐지?

C언어에서 문자배열을 문자형 포인터와 마치 같은 것처럼 사용한다. 이가 가능한 이유는 *분명 배열과 포인터의 형식은 다르지만, 배열이 배열의 첫 원소를 가리키는 포인터로* ***붕괴****되기 때문이다.*

즉, T[] -> T\*으로 된다는 것이다. 아래와 같이 말이다.

const char name[] = "ytk";

const char\* niceProgrammer = name; //T[] -> T\*

### 배열은 어떻게 연역되나?

배열은 어떻게 추론될까? 위의 원리로 작동한다. 인자로 T[]이 넘어오면 T\*로 취급한다. 붕괴되기 때문이다. 실제 배열 타입이 아니다. 결국, 포인터를 값으로 넘기게 되며 그냥 포인터 자체를 연역하게 된다.

**template**<**typename** T>

void func(T param){

cout << param << endl;

}

int main(){

**const** char name[] = "I like programming.";

**const** char\* ptr = name;

func(name); *//T:* *const char[] Param: const char\**

func(ptr); *//T:* *const char\* Param: const char\**

}

### 배열 참조 연역

T&로 매개변수를 받으면, 배열 타입의 참조로 연역할 수 있다. (T(&) param[size]가능) **이것은 배열을 하나의 새로운 타입처럼 취급하는 것이다.**

**template**<**typename** T>

void func(T& param){

cout << param << endl;

}

int main(){

**const** char name[25] = "I like programming.";

**const** char\* ptr = name;

*//T:const char[], Param: const char\**

func(ptr); *//T&:const char[25], Param: const char (&)[25]*

}

이로 인해 배열의 크기까지 알 수 있다니 놀라지 않을 수 없다. – [항목14, 15] 참고

**template**<**typename** T, size\_t N>

*//constexpr: 컴파일 전에 반환 값을 알 수 있다.*

**constexpr** size\_t arraySize(T (&)[N]) **noexcept**{

**return** N;

}

int main(){

int keyVals[] = { 1, 3, 5, 7, 9 };

array<int, arraySize(keyVals)> mappedVals;

}

## 함수 인수

배열과 마찬가지다. 예제를 보자.

void makeAtom(int n, double d){

cout << "create " << n + 100000 + d << endl;

};

**template**<**typename** T>

void f1(T param){

param(1, 2); // void(\*)(int, double)

cout << "T" << endl;

};

**template**<**typename** T>

void f2(T& param){

param(3, 4); // void(\*)(int, double)

cout << "T&" << endl;

};

int main(){

*//함수를 인자로 보낸다.*

f1(makeAtom);

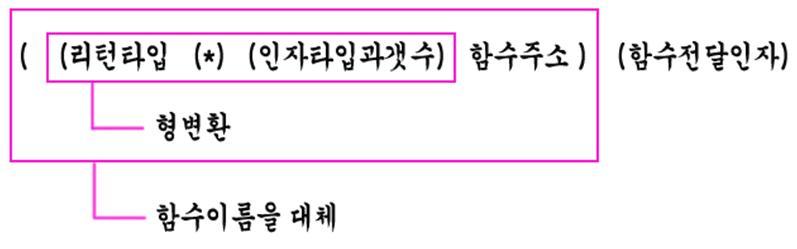
f2(makeAtom);

}

함수 형식은 함수 포인터로 붕괴한다. 함수 포인터는 조금 특별한 포인터다. 결국, 함수를 인자로 넘기면 void(\*)(int, double)형태가 된다. 핵심은 함수를 전달하면 일단 함수 이름은 포인터로 붕괴한다. 그리고 T는 함수 포인터로 연역된다는 것이다. 결과적으로 매개변수는 포인터 형태f1(T param)거나 참조 f2(T& param)형태이다.   
결과적으로 두 경우가 달라지는 경우는 드물다. 함수 이름 자체는 포인터이므로 포인터는 복사되나 참조하나 크게 다르지 않다.

### 함수 포인터

**함수의 이름**은 메모리상에 존재하는 **함수의 위치를 가리키는 포인터**입니다. 즉 함수의 이름은 주소 값을 나타냅니다. 결국, 함수 이름은 특별한 변수인 셈입니다. 그럼 이 변수를 표현하는 타입은 무엇일까? 함수의 타입을 결정짓는 요소는 **리턴형과 전달 인자**입니다.



아래는 함수 add의 포인터를 fptr1에 복사하여 fptr1과 add는 동일하게 된다.

**#include** <stdio.h>

**void** add(**int** a, **int** b);

**void** sprint(**char** \*str);

**int** main(**void**)

{

**char** \*string = "Function Pointer";

**int** a = 10;

**int** b = 20;

**void** (\*fptr1)(**int**, **int**)=add;    *//함수포인터 fptr1선언과 동시에 add함수로 초기화*

**void** (\*fptr2)(**char**\*)=sprint;    *//함수포인터 fptr2선언과 동시에 sprint함수로 초기화*

  fptr1(a,b);

  fptr2(string);

**return** 0;

}

아래는 함수 포인터의 주소를 직접 따와서 함수의 이름 대신 그 주소를 사용하여 호출하는 코드이다.

**void** test(**int**);

**int** main(**void**)

{

  test(10);

  printf("test함수의 주소값: %x\n",test);

  ((**void**(\*)(**int**))0x040102F)(20);

**return** 0;

}

**void** test(**int** a)

{

  printf("%d\n",a);

**return** ;

}

***템플릿 형식 연역 도중에 참조 형식의 인수들은 비참조로 취급된다. 즉, 참조성이 무시된다.***

***보편 참조 매개변수에 대한 형식 연역 과정에서 왼값 인수들은 특별하게 취급된다.***

***값 전달 방식의 매개변수에 대한 형식 연역 과정에서 const 또는 volatile(또는 그 둘 다인)는 무시된다.***

***템플릿 형식 연역 과정에서 배열이나 함수 이름에 해당하는 인수는 포인터로 붕괴한다. 단, 그런 인수가 참조를 초기화하는 데 쓰이는 경우에는 포인터로 붕괴하지 않는다.***

[항목2] auto의 형식 연역 규칙

# auto란?

자동으로 형을 추론하게 하는 키워드다. 임의의 클래스, 혹은 함수 안에서 사용하는 템플릿 타입과는 다르게 마치 하나의 기본타입처럼 간단하게 사용한다.

## auto의 특징

auto를 사용할 때는 변수 선언과 동시에 반드시 초기화가 이루어질 필요가 있으며, 초기화 시 주어지는 값을 통해 타입을 추론하여 가장 적합한 자료형이 지정됩니다.

auto는 코드의 양을 줄여주고, 가독성을 높여줍니다. iterA 변수의 선언식과 iterB의 선언식을 비교해보면, 코드가 상당히 줄어들게 됨을 확인할 수 있습니다.

int main(int argc, char\*\* argv)

{

vector vec;

vector::iterator\* iterA = (vector::iterator\*) &vec.begin(); //기존 코드

**auto** iterB = (vector::iterator\*) &vec.begin(); //auto로 인한 가독성 향상

}

## auto의 사용 방법

auto는 크게 변수 앞, 함수 리턴 값, 람다 리턴 값 정도로 사용된다.

#include *<iostream>*

#include *<cmath>*

#include *<typeinfo>*

**template**<**class** **T**, **class** **U**>

**auto** add(T t, U u) -> **decltype**(t + u) *// the return type is the type of operator+(T, U)*

{

**return** t + u;

}

**auto** get\_fun(int arg) -> double (\*)(double)

*// same as: double (\*get\_fun(int))(double)*

{

**switch** (arg)

{

**case** 1: **return** std::fabs;

**case** 2: **return** std::sin;

**default**: **return** std::cos;

}

}

int main()

{

**auto** a = 1 + 2;

std::cout << "type of a: " << **typeid**(a).name() << '\n';

**auto** b = add(1, 1.2);

std::cout << "type of b: " << **typeid**(b).name() << '\n';

**auto** c = {1, 2};

std::cout << "type of c: " << **typeid**(c).name() << '\n';

**auto** my\_lambda = [](int x) { **return** x + 3; };

std::cout << "my\_lambda: " << my\_lambda(5) << '\n';

**auto** my\_fun = get\_fun(2);

std::cout << "type of my\_fun: " << **typeid**(my\_fun).name() << '\n';

std::cout << "my\_fun: " << my\_fun(3) << '\n';

*// auto int x; error as of C++11: "auto" is no longer a storage-class specifier*

}

*/\*output:*

*Possible output:*

*type of a: int*

*type of b: double*

*type of c: std::initializer\_list<int>*

*my\_lambda: 8*

*type of my\_fun: double (\*)(double)*

*my\_fun: 0.14112\*/*

자세한 내용은 아래에서 살펴볼 수 있다.

# 변수 앞의 auto의 형식 연역 규칙

변수 앞에 있는 auto는 어떤 규칙으로, 어떻게 형을 추론할까?

## template 형식 연역 규칙과 비슷하다.

반가운 것은, [항목1]에서의 템플릿 형식 연역 규칙과 아주 비슷하게 작용한다는 것이다. **template**: T 🡪 paramType  
**auto**: 우측 타입 🡪 autoType

int main(){

**auto** x = 27; // = f(27) -> f(T param), auto: int

**const** **auto** cx = x; // = f(x) -> f(const T param), auto: const int

**const** **auto**& rx = cx; // = f(cx) -> f(T& param), auto: const int&

**auto**&& uref1 = x; // = f(x) -> f(T&& param), auto: int&

**auto**&& uref2 = cx; // = f(cx) -> f(T&& param), auto: const int&

**auto**&& uref3 = 27; // = f(27) -> f(T&& param), auto: int&&

**const** char name[] = “ytk”;

**auto** arr1 = name; // auto: const char\*

**auto**& arr2 = name; // auto: const char (&)[3]

}

## auto는 추가적으로 std::initializer\_list<T>를 연역한다.

auto는 템플릿이 할 수 없는 균일 초기화에 대해 형 추론을 시행한다.

### std::initializer\_list<T>

C++11에서부터는 다음과 같은 코드가 가능하다.  
int x3 = {27, 28, 29};   
배열을 경량화시켜 int에 담을 수 있는 것이다. 곧, int가 std::initializer\_list<int>형으로 되는 것이다.

#### initializer\_list의 특징

initializer\_list는 두 개의 포인터 (T\* begin, T\* end)를 이용해 구현하는 방법이 있고, 하나의 포인터와 사이즈 변수를 이용해 구현하는 방법이 있는데, **VS2013의 경우 두 개의 포인터(\*begin, \*end)를 이용하여 구현**되어 있다.

i**nitializer\_list 오브젝트의 복사**는 *deep-copy*가 아닌, **포인터의 *shallow-copy***로 이루어지므로 원본이 파괴되면 내가 바라던 진행에서 멀어질 것이다.

#### initializer\_list는 어디에 쓸까? : Uniform initialization

*Uniform initialization*: initializer\_list으로 모든 오브젝트 타입에 대해 동일한 방식으로 진행하는 리스트형 초기화(list initialization), 즉, 모든 타입을 brace-init-list { ... }로 초기화한다는 의미다.

int \_tmain(int argc, \_TCHAR\* argv[])

{

*//구조체/클래스*

*// bs1 x = 5, y = 3.2*

BasicStruct bs1{ 5, 3.2 };

*// 빈 {}은 데이터를 초기화시킨다 (x = 0, y = 0.0)*

BasicStruct bs2{};

*// (int,float)를 인자로 가지는 생성자 함수 호출*

AltStruct as{ 2, 4.3 };

*//정수 타입*

*// a = 0*

int a{};

*// b = 1.f*

float b{ 1.f };

*// uc1 = 10*

unsigned char uc{ 10 };

*// 하지만 narrow conversion(축소 변환)이 발생하면 아니된다.*

*// 즉, -1이 int로 취급되고, 이를 더 작은 타입인 unsigned char로 캐스팅하는 꼴이 되는 것이다*

*// unsigned char uc2{ -1 };*

*// 기존 -1의 의미를 정 주고 싶으면, 그냥 0xFF를 쓰자*

*//STL 컨테이너*

vector<int> v1{ 1, 2, 3, 4, 5 };

*// 빈 벡터*

vector<float> v2{};

map<int, int> m1

{

{ 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 }, { 5, 5 }

};

**return** 0;

}

출처: <http://sweeper.egloos.com/3058356>

#### initializer\_list를 사용해볼까?

#include *<iostream>*

#include *<vector>*

#include *<initializer\_list>*

**template** <**class** **T**>

**struct** S {

std::vector<T> v;

S(std::initializer\_list<T> l) : v(l) {

std::cout << "constructed with a " << l.size() << "-element list**\n**";

}

void append(std::initializer\_list<T> l) {

v.insert(v.end(), l.begin(), l.end());

}

std::pair<**const** T\*, std::size\_t> c\_arr() **const** {

**return** {&v[0], v.size()};   
// copy list-initialization in return statement

// this is NOT a use of std::initializer\_list

}

};

**template** <**typename** T>

void templated\_fn(T) {}

int main()

{

S<int> s = {1, 2, 3, 4, 5}; // copy list-initialization

s.append({6, 7, 8}); // list-initialization in function call

std::cout << "The vector size is now " << s.c\_arr().second << " ints:**\n**";

**for** (**auto** **n** : s.v)

std::cout << n << ' ';

std::cout << '\n';

std::cout << "Range-for over brace-init-list: **\n**";

**for** (int **x** : {-1, -2, -3}) // the rule for auto makes this ranged-for work

std::cout << x << ' ';

std::cout << '\n';

**auto** al = {10, 11, 12}; // special rule for auto

std::cout << "The list bound to auto has size() = " << al.size() << '\n';

// templated\_fn({1, 2, 3}); //error: 템플릿에서는 연역 안 됩니다.

templated\_fn<std::initializer\_list<int>>({1, 2, 3}); // OK

templated\_fn<std::vector<int>>({1, 2, 3}); // also OK

}

### auto는 std::initializer\_list<T>를 연역할 수 있다.

다음과 같은 면에서 template과 차이가 있을 수 있다.

auto x3 = {1, 3, 5} // auto: Std::initializer\_list<int>

*//템플릿에서 위를 연역하고자 한다면 아래처럼 해야 한다.*

template<**typename** T>

void f(Std::initializer\_list<T> initList);

f({11, 22, 9999});

template는 f안에 들어있는 {}를 std::initializer\_list<int>라고 연역할 수 없다. 반면 auto는 가능하다. 위의 코드에서는 인자 타입을 바꿔 이를 이해하도록 바꾼 템플릿 함수이다.

### Question. 템플릿은 안 되며 auto만 가능한데 특별한 이유가 있을까?

std::initializer\_list<T>자체는 템플릿이다. 즉, 타입 자체가 추론이 필요한 경우다. 그런데 현재 목적인 템플릿 인자 추론의 경우에서 타입 내부의 타입까지 중첩하여 추론해야 한다. 이런 고급스러운 행위는 컴파일러에게 어렵다.

반면, auto는 가능하다. auto는 대입을 한 다음에 추론이 생기지 않는가?

# 함수 반환에 auto 형식 연역(in C++14)

함수 반환 값에 auto를 사용하는 것 역시 가능하다. 사실 제목과는 다르게 이 과정은 앞에서 본 형식 연역과는 관련이 없고 ***후행 반환 형식***을 사용한다는 의미다. 그래서 C++11에서는 auto홀로 쓰일 수 없고 반환 형식을 표현할 decltype과 함께 써야 한다([항목3]참조). C++14에서는 auto만 사용해도 사용 가능하다.

### 후행 반환 형식(trailing return type)

어떤 함수의 반환 형식을 예측할 수 없는 경우도 있을까? 위에서 타입 추론에 대해 논하고 있다. 어떠한 함수에 템플릿이 씌어져 있어 타입을 추론해야 하는 상황이다. 그런데 반환 값은 이 추론 한 값에 영향을 받는다. 그럼 어떻게 할 것인가?   
**이럴 경우 후행 반환 형식을 반환 값도 추론하게 한다.** 결국, 타입 추론의 연장선이라고 볼 수 있다.

int& foo(int& i)

{

return i;

};

float foo(float& f)

{

return 0.0f;

}

template <class T>

auto transparent\_forwarder(T& t)

//-> decltype(foo(t)) //C++11

{

return foo(t);

}

int main()

{

int a = 9999;

std::cout << transparent\_forwarder(a);

return 0;

}

**결국, 후행 반환 형식이란 직접 반환 형식을 지정하지 않고 주어진 매개변수로 결정한다는 것을 의미한다.** 이는 위에서처럼 auto로 명시할 수 있다. 이 auto는 형식 연역과 상관없는 후행 반환 형식을 사용할 것이라는 사용자의 의지다.

### std::initializer\_list<T>의 후행 반환 형식

함수 반환에서의 auto는 위의 과정을 통해 템플릿 형식 형식 연역 규칙이 적용된다. 그러므로 std::initializer\_list<T>을 연역할 수 없다.

**auto** createArcher()

{

**return** {1, 2, 3} //error: 연역할 수 없음

}

중괄호의 추론과 반환 값의 추론은 동시에 일어나므로 템플릿 형식 연역 규칙과 동일할 수 밖에 없다.

# 람다 매개변수의 auto 형식 연역(in C++14)

람다식은 변수 앞에 auto를 붙여서 시작한다. 그리고 위의 원칙대로 반환 값을 뒤에서 추론한다.

## 람다식에서의 매개변수 auto 연역 규칙

auto로 반환한 식에서 후행 반환 형식을 더 이상 명시하지 않음으로 매개변수에 auto를 사용할 수 있게 되었다. 이는 **템플릿 형식 연역의 규칙**을 갖는다.

vector<int> v;

**auto** resetV = [&v](**const** **auto**& newValue) { v = newValue; };

resetV({1, 5, 9}); //error: newValue를 std::initializer\_list<T>로 연역할 수 없음

당연하지만, 여기서 auto는 템플릿 인자와 다른 것이 없다. 람다 함수용 인자 추론 장치라고 볼 수 있다. 그러므로 템플릿 형식 연역 규칙을 갖는 것이 당연하다.

***auto형식연역은 대체로 템플릿 형식 연역과 같지만, auto 형식 연역은 중괄호 초기치가 Std::initializer\_list<T>를 나타낸다고 가정하는 반면, 템플릿 형식 연역은 그렇지 못하다.***

***함수의 반환 형식이나 람다 매개변수에 쓰인 auto에 대해서는 auto 형식 연역이 아니라 템플릿 형식 연역이 적용된다.***

[항목3] decltype의 작동 방식

# decltype이 뭐죠?

*decltype*은 **decl**ared **type**(선언된 형식)의 줄임 말로써, 주어진 표현 식의 타입을 알려주는 키워드이다. 아래와 같다.

// 이런 경우엔 그냥 auto y = x;가 훨씬 쓰기도 보기도 좋다

decltype(x) y = x;

x의 타입을 추론하여 이를 y의 타입으로 사용한다. 하지만 auto보다 나은 것이 없어 보인다. 계속 사용해야 할까?

# decltype을 왜 쓸까요?

변수 선언문에는 물론 auto가 편리하다. 하지만 이 외 몇 가지 용도가 있다.

## decltype의 불변성

템플릿 타입 연역과 구별되는 아주 특별한 점은, **타입 변환을 하지 않고 그대로 보여준다는 것이다.**

**template** <**typename** T>

void func(T a); *// 어떤 값이 들어오든 a가 참조가 아닙니다.*

**decltype**(a); *// a가 참조라면 &a, 아니면 a를 보여줍니다.*

## auto반환 형식 표현

[항목2]에서도 잠깐 언급되었지만, ***trailing return type***(후행 반환 형식)으로 decltype을 사용함으로써, auto 반환 값을 명시할 수 있다.

1. // auto 반환함수와 후행 반환 형식으로 int 사용
2. auto add\_function(int a, int b) -> int
3. {
4. return a + b;
5. }
7. // auto 반환과 후행 반환 형식으로 decltype()을 사용
8. template <typename T, typename U>
9. auto add\_template(T&& x, U&& y) -> **decltype**(std::forward<T>(x)+std::forward<U>(y))
10. {
11. return std::forward<T>(x) + std::forward<U>(y);
12. }
13. // BUILDER의 makeObject() 반환 형식으로부터 자유로워짐
14. template <typename TBuilder>
15. auto MakeAndProcessObj(const TBuilder& builder) -> **decltype**(builder.makeObject())
16. {
17. auto val = builder.makeObject();
18. // process...
19. return val;
20. }

## auto반환 in C++14

C++11은 람다함수가 한 문장으로 이루어져 있다면 그 반환 형식의 연역을 허용하며, C++14는 허용 범위를 확장해서 모든 람다와 함수의 반환 형식 연역을 허용한다.

그러므로 반환 형식을 표기 안 해도 된다.

1. template <typename T, typename U>
2. auto add\_template(T&& x, U&& y)
3. // -> decltype(std::forward<T>(x) + std::forward<U>(y))
4. {
5. return std::forward<T>(x) + std::forward<U>(y);
6. }
8. auto add\_function(int a, int b) // -> decltype(a+b)
9. {
10. return a + b;
11. }
13. template <typename TBuilder>
14. auto MakeAndProcessObject(const TBuilder& builder)
15. // -> decltype(builder.makeObject())
16. {
17. auto val = builder.makeObject();
18. // process...
19. return val;
20. }
22. struct Builder
23. {
24. static std::string makeObject() { return std::string("hello"); }
25. };
27. int main()
28. {
29. add\_template(1, 2);
30. add\_function(1, 2);
32. // a is std::string type
33. Builder builder;
34. auto a = MakeAndProcessObject(builder);
35. }

# decltype를 이용한 auto타입 반환 함수 만들기

위의 규칙을 적용시켜 직접 사용해보자!

## C++11

void authenticateUser(){

cout << "authenticateUser" << endl;

}

**template**<**typename** Container, **typename** Index>

**auto** authAndAccess(Container& c, Index i) -> **decltype**(c[i])

{

authenticateUser();

**return** c[i];

}

**class** **Archer**{

**public**:

Archer(string n, int l = 1) : name(n), level(l), invectory(vector<int>(10, -1)) {}

void addItem(int item){

**for** (int i = 0; i < 10; i++){

**if** (invectory[i] == -1) {

cout << name << " gets item." << endl;

invectory[i] = item; **return**;

}

}

}

int& getItem(int i){

**return** invectory[i];

}

**private**:

string name;

int level;

vector<int> invectory;

};

**template**<**typename** Arc, **typename** Item>

*//반환 값을 a와 i로 추론하고 어떤 객체(l-value)를 참조하여 진행합니다.*

**auto** showItem(Arc& a, Item i) -> **decltype**(a.getItem(i)){

**return** a.getItem(i);

}

int main(){

Archer a("ytk", 100);

**for** (int i = 10; i < 15; i++) a.addItem(i);

*//잘 됩니다.*

showItem(a, 3) = 100;

cout << "this is " << a.getItem(3) << endl;

vector<int> vec(10, 7);

*//역시 잘 됩니다.*

*// -> decltype(a.getItem(i))이 없다면 안 될 겁니다.*

authAndAccess(vec, 5) = 10;

**return** 0;

}

## C++14

위의 decltype문을 생략할 수 있지만, 이 과정에서 반환 타입의 연역 과정은 템플릿 연역 규칙을 따른다. 그러므로 참조 값을 반환한다면, 이를 없애서 위의 식이 안 될 것이다.

**template**<**typename** Container, **typename** Index>

*//여기서 연역된 반환 값은 int&가 아닌 int입니다. 규칙에 따라 &를 없애버리거든요.*

**auto** authAndAccess(Container& c, Index i)

{

authenticateUser();

**return** c[i];

}

int main(){

Archer a("ytk", 100);

**for** (int i = 10; i < 15; i++) a.addItem(i);

showItem(a, 3) = 100;

cout << "this is " << a.getItem(3) << endl;

vector<int> vec(10, 7);

*//안 됩니다.*

authAndAccess(vec, 5) = 10;

**return** 0;

}

이를 해결하기 위해서

**template**<**typename** Container, **typename** Index>

*//decltype은 명확한 타입을 반환하므로 컴파일에 성공할 수 있다.*

**decltype(auto)** authAndAccess(Container& c, Index i)

{

authenticateUser();

**return** c[i];

}

auto는 후행 반환 형식을 명시함으로 반환 값의 타입을 유추하게 한다. 그리고 decltype(x)는 x가 decltype방식으로 연역되어야 함을 뜻한다. 그러므로 반환 타입은 decltype의 방식으로 연역될 수 있는 것이다.

# decltype사용 시, 주의할 점

decltype이 제대로 작동하지 않을 수도 있다.

## 매개변수가 r-value임에도 성공해야 합니다.

위에서의 container은 참조 값이므로 l-value가 아닌가? r-value도 받으려면 어떻게 해야 하는가? 위에서도 언급되었다. 간단하게 &&(보편 참조)를 이용한다. 이는 r-value, l-value둘 다 허용한다.

**template**<**typename** Container, **typename** Index>

**auto** authAndAccess(Container&& c, Index i)

*//C++11버전입니다. C++14는 아래를 없애고 auto를 decltype(auto)로 바꿉니다.*

-> **decltype**(forward<Container>(c)[i])

{

authenticateUser();

*//return c[i];*

**return** forward<Container>(c)[i];

}

c의 원래 부호가 왼값인지 오른값인지 달라질 수 있기에 decltype을 사용하면 편리하게 값을 반환할 수 있다.

### Question. forward에 Container은 왜 있어야 하죠?

이는 보편 참조에서 인자로 받은 c가 실제로 왼값인지 오른값인지 구별하기 위해 사용된다. – [항목25] 참조

## 복잡한 표현식의 경우

위에서 내용을 정리하면 decltype은 안의 내용 그대로의 타입을 보여준다. decltype을 이름에 사용할 수도 있지만 위에서처럼 표현식에도 사용될 수 있다. 그런데 이렇게 이름보다 복잡한 왼값 표현식에서는 무조건 형식이 T&인 왼값 참조를 반환한다.

decltype(auto) fight()

{

int power = 1000;

return (power);

}

fight() = 9999;

이런 식이 가능하다. 미정의 급행 열차로 갈아타는 수가 있다.

왜 이래야 할까? 대부분의 왼값 표현식에는 태생적으로 왼값 참조를 포함하고 있다. 편하게 연역하려는 decltype의 꼼수인가!?

### 오른 값이면 어떻게 되는가?

int u = 5;

decltype((u)) v; // int&

decltype((3)) l = 9999; // int

그냥 정상적으로 작동된다(T로 연역). 문제될 여지가 없다.

***decltype은 항상 변수나 표현식의 형식을 아무 수정 없이 보고한다.***

***decltype은 형식이 T이고 이름이 아닌 왼값 표현식에 대해서는 항상 T&형식을 보고한다.***

***C++14는 decltype(auto)를 지원한다. decltype(auto)는 auto처럼 초기치로부터 형식을 연역하지만, 그 형식 연역 과정에서 decltype의 규칙들을 적용한다.***

[항목4] 연역된 형식 파악하기

# 연역된 형식을 알아내는 방법

코딩 중 ‘이것은 어떤 형식으로 연역되었을까?’라고 궁금해할 수 있을 것이다. 3가지 방법을 제시한다.

## IDE편집기

시점: 코드 작성, 수정 중

커서를 올리면 무슨 타입인지 알려준다.

**template**<**typename** T>

**auto** func(T\* a) -> **decltype**(\*a)

{

**return** (\*(a + 1));

}

int main(){

**const** int ytk = 9999;

**auto** x = ytk; *//커서 대니까 int x랍니다.*

**auto** y = func(&x); *//int\*\* y이네요.*

cout << y << endl;

}

생각보다 편리하다.

## 컴파일러 진단 메시지

시점: 컴파일 중

**template**<**typename** T>

*//TD: Type Displayer*

**class** **TD**;

int main(){

**const** int ytk = 9999;

TD<**decltype**(x)> xType;

TD<**decltype**(ytk)> yType;

}

1>d:\programming\ec++\ec++\item04.cpp(22): error C2079: 'xType'은(는) 정의되지 않은 class 'TD<int>'을(를) 사용합니다.

1>d:\programming\ec++\ec++\item04.cpp(23): error C2079: 'yType'은(는) 정의되지 않은 class 'TD<const int>'을(를) 사용합니다.

과 같이 타입을 알려준다.

## 실행시점 출력

시점: 실행 중

**template**<**typename** T>

void f(**const** T& param){

cout << "T = " << **typeid**(T).name() << endl;

cout << "param = " << **typeid**(param).name() << endl;

}

**class** **Archer**{};

vector<Archer> createVec(){

**return** vector<Archer>(10);

}

int main(){

**const** **auto** vw = createVec();

**if** (!vw.empty()){

f(&vw[0]);

}

}

//T = class Archer const \*

//param = class Archer const \* -> 원래는 const Archer const\*(&)이어야 한다.

***컴파일러가 연역하는 형식을 IDE편집기나 컴파일러 오류 메시지, Boost TypeIndex 라이브러리를 이용해서 파악할 수 있는 경우가 많다.***

***일부 도구의 결과는 유용하지도 않고 정확하지도 않을 수 있으므로, C++의 형식 연역 규칙들을 제대로 이해하는 것은 여전히 필요하다.***

[항목5] auto를 선호하라.

명시적 형식 선언보다는 auto가 낫다.

auto에 대한 자세한 내용은 [항목2] 참고

# [auto를 쓰자1] 초기화 문제 해결

int x;

“어이쿠 이런! 깜빡 하고 초기화를 안 해버렸네. 컴파일러가 알려주겠지?“ 아쉽게도 경우에 따라서 원하지 않은 값으로 초기화 해버리고 미정의 급행열차에 올라타게 된다.

하지만, auto를 사용한다면 이럴 일은 절대 없다. 왜냐하면 반드시 초기화를 해야 하는 것이 auto의 규칙이기 때문이다.

# [auto를 쓰자2] 간단한 코드

주석으로 된 부분을 auto로 간단하게 표현한 모습이다.

**template**<**typename** It>

void find(It a, It b){

**for** (; a != b; ++a){

*//typename iterator\_traits<It>::value\_type curValue = \*a;*

**auto** curValue = \*a;

cout << curValue << " ";

}

cout << endl;

}

int main(){

*//vector<int> v = { 1, 2, 3, 4, 5 };*

**auto** v = { 1, 2, 3, 4, 5 };

*//vector<int>::iterator iter = lower\_bound(v.begin(), v.end(), 3);*

**auto** iter = lower\_bound(v.begin(), v.end(), 3);

find(iter, v.end());

**return** 0;

}

# [auto를 쓰자3] 클로저를 담을 때의 유리함

어째서 auto를 사용하면 효율적인 걸까? 아래 코드에서 살펴본다.

int main(){

unique\_ptr<Archer> pA1, pA2;

**for** (int i = 0; i < 5; i++) pA1->add(i);

**for** (int i = 5; i < 10; i++) pA2->add(i);

*//원래 반환 형식은 이렇습니다.*

function<bool  
 (**const** unique\_ptr<Archer>& a, **const** unique\_ptr<Archer>& b)>

calcSub = []  
 (**const** unique\_ptr<Archer>& a, **const** unique\_ptr<Archer>& b)

{

**return** a->get() - b->get();

};

*//auto로 매우 간단해졌군요.*

**auto** calcSub = []

(**const** unique\_ptr<Archer>& a, **const** unique\_ptr<Archer>& b)

{

**return** a->get() - b->get();

};

*//매개변수까지 auto로 더 간단해졌군요. 물론 C++11에서는 안 되지만요.*

**auto** calcSub = [](**const** **auto**& a, **const** **auto**& b){

**return** a->get() - b->get();

};

**return** 0;

}

람다식은 function<반환값(매개변수)>으로 표현할 수 있다([용어: 람다] 참고). 위의 3가지 코드는 같은 의미지만 단순히 코드만 다른 것은 아니다.

## function보다 좋은 메모리 효율

[미결] auto로 선언된 클로저(람다식으로 만든 함수 객체)는 그 클로저에 요구되는 만큼의 메모리만 사용한다. 반면 function으로 선언된 변수의 형식은 function템플릿의 한 인스턴스 형태이며, 그 크기는 임의의 주어진 서명에 대해 고정되어 있다. 그런데 이 고정된 크기가 모자를 수도 있다는 것이다. 그러면 힙 메모리를 할당하여 어려운 과정을 겪게 된다.

결과적으로 function객체는 auto객체보다 메모리를 더 많이 소모한다.

## 시간 효율

function은 inline화를 제한하고 간접 함수 호출을 산출하는 구현 세부사항 때문에 auto보다 항상 느리다.

# [auto를 쓰자4] 형식 단축(type shortcut)

형식 단축과 관련된 문제를 피할 수 있다.

## 32bit, 64bit

vector<int>::size\_type은 64비트다. 하지만 대부분 개발자는 이것을 모르므로 이것을 담는 무언가의 변수를 32비트로 선언할 수 있다. 그러면 Windows환경에 따라 문제가 발생할 수 있겠지?

## 형식 불일치로 인한 비효율성

int counter[10];

int main(){

unordered\_map<string, int> map;

string s; cin >> s;

*//위에는 <string, int>지만 아래는 const<string, int>이다.*

*//for (const pair<string, int>& p : map)*

**for** (**const** std::pair<string, int>& **p** : map)

{

**if** (p.first == s) counter[0]++;

}

**return** 0;

}

선언된 형식은 const가 없지만, 아래에서 탐색하는 구간의 형식은 const가 있다. 여기서 형이 다르므로 형 변환이 생기는데, 위에서는 각 원소의 형을 변환해야 하므로 루프에서 각 원소마다 임시객체를 생성하고 그 임시객체를 참조p에 묶는 변환을 수행한다.

아래와 같이 auto로 선언하면 위의 문제가 싹 다 해결된다.

**for** (**auto**& **p** : map)

***auto변수는 반드시 초기화해야 하며, 이식성 또는 효율성 문제를 유발할 수 있는 형식 불일치가 발생하는 경우가 거의 없으며, 대체로 변수의 형식을 명시적으로 지정할 때보다 타자량도 적다.***

***auto로 형식을 지정한 변수는 항목2와 항목6에서 설명한 문제점들을 겪을 수 있다.***

[항목6] auto사용 시 주의할 점

auto가 원치 않은 형식으로 연역될 때에는 명시적 형식의 초기치를 사용하라.

# 무조건 auto만 사용하면 될까?

[항목5]에서 auto의 장점들은 장엄하다. 그러므로 이는 매우 바람직하다. 하지만 걸림돌이 있다. 그것이 이번 항목에서 다룰 내용이다.

# auto연역과 대리자 클래스(proxy class)

auto는 대리자 클래스를 이용한 대상에 대하여 올바르게 연역할 수 없다.

## 대리자 클래스(proxy class)란?

다른 객체의 동작을 흉내 내거나 그 동작을 보강하는 클래스이다. 무슨 말인지 도통 알기 어렵다.

### 기본 예시

아래 코드를 보자.

struct aproxy {

aproxy(int& r) : mPtr(&r) {}

void operator = (int n) {

if (n > 1) {

throw "not binary digit";

}

\*mPtr = n;

}

int \* mPtr;

};

struct array {

int mArray[10];

aproxy operator[](int i) {

return aproxy(mArray[i]);

}

};

int main() {

try {

array a;

a[0] = 1; // ok

a[0] = 42; // throws exception

}

catch (const char \* e) {

cout << e << endl;

}

}

배열 a에 값을 대입할 때, 유효한지 검사할 필요가 있다. 하지만 a에서 int&를 반환한다면 이 과정을 연산자 안에서 직접 할 수 없다. 대리자 클래스는 이러한 부가적인 기능을 더해준다.

### 사용 예시1: std::shared\_ptr<T>

std::shared\_ptr에는 참조 카운터를 관리하는 객체가 들어있다. **std::shared\_ptr은 원본 포인터를 들고 있으며, 이를 복사하거나 소멸시키면 참조 카운터 객체가 작동**한다. 결국, 이는 어느 객체를 가리키는 포인터의 기능에 참조 가능을 추가하는 대리자 클래스인 셈이다.

private:

\_Ty \*\_Ptr;

\_Ref\_count\_base \*\_Rep;

template<class \_Ty0>

friend class \_Ptr\_base;

void \_Reset(\_Ty \*\_Other\_ptr, \_Ref\_count\_base \*\_Other\_rep)

{ // release resource and take \_Other\_ptr through \_Other\_rep

if (\_Other\_rep)

\_Other\_rep->\_Incref();

\_Reset0(\_Other\_ptr, \_Other\_rep);

}

이는 위에서 노란 부분이 복사(소멸)이 발생할 때마다 객체 카운터를 증가(감소)시키는 역할을 한다.

### 사용 예시2: std::vector<bool>

std::vector<bool>은 특별한 std::vector<T>이다. std::vector<bool>의 각 원소는 하나의 bool타입을 가지지 않는다. 각 원소는 비트마스크 형태로 관리되며 어떠한 원소를 반환하려면 std::vector<bool>은 **bool이 아닌 std::vector<bool>::reference타입(해당 비트를 담는 워드를 가리키는 포인터와 오프셋)을 반환**한다.

reference operator[](size\_type \_Off)

{ // subscript mutable sequence

iterator \_It = begin();

\_It.\_Advance(\_Off);

return (\*\_It);

}

template<class \_Alloc>

class \_Vb\_reference

: public \_Vb\_iter\_base<\_Alloc>

{

// ...

\_Mytype& operator=(bool \_Val) \_NOEXCEPT

{ // assign \_Val to bit

if (\_Val)

\*(\_Vbase \*)\_Getptr() |= \_Mask();

else

\*(\_Vbase \*)\_Getptr() &= (~\_Mask()); // STET

return (\*this);

}

operator bool() const \_NOEXCEPT

{ // test if bit is set

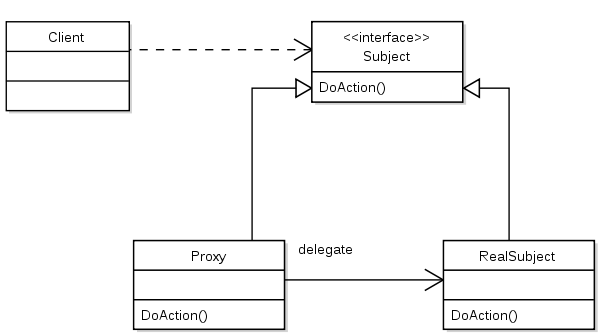
return ((\*\_Getptr() & \_Mask()) != 0);

}

std::vector<bool>의 operator[]의 반환값은 T&이 아니라 reference이다. 이 referene는 바로 아래에서 보여주고 있다. 이는 **bool을 대입 인자로 받음**과 **operator bool을 오버로딩**하여 포인터와 오프셋으로 값을 판단하는 모습을 보인다.

### 프록시 패턴(proxy pattern)

대리자 클래스는 프록시 패턴과 연관이 깊다. *프록시 패턴*이란, 대리자 클래스가 기능을 수행하고 나머지를 실질적인 클래스에게 넘겨주는 방식의 패턴이다.



프록시가 DoAction을 수행한다. 이걸 Real에게 준다. Real은 프록시가 마저 못한 DoAction을 수행하고 클라이언트에게 결과를 보여준다.

### 2가지 대리자 클래스

그리고 위의 스마트포인터처럼 직접적으로 사용할 수 있는, 눈에 *‘보이는’ 대리자 클래스*가 있으며, 한편 어떤 클래스 내부에 숨겨져 있는 *‘보이지 않는’ 대리자 클래스*가 있다.

대리자 클래스는 사용이 광범위하다. 사용자가 직접 호출하고 직접 처리하는 과정을 그리 아름답지 않다. 이를 피하려면 객체가 스스로 처리해야 하며 이러한 객체들이 대리자가 되는 것이다. 결국, 객체 지향의 한 재료인 셈이다.

## 대리자 클래스는 auto연역을 방해한다.

대리자 클래스 중에서도 ‘보이지 않는’대리자 클래스가 문제이다. 그 중 하나의 예시인 대리자 클래스를 가지고 있는 vector<bool>을 제시한다.

### std::vector<bool>가 일으키는 문제

위에서 언급한 것처럼, 이가 반환하는 reference라는 객체는 bool로 변환할 수 있다.

std::vector<bool> vec\_b(1000000, false);

auto element = vec\_b[100]; // std::\_Vb\_reference<alloc> element

if (flag | element) std::cout << "fuck\n";

위에서는 element가 bool연산을 하기 위해 bool로 암시적 변환이 이루어져야 한다. 여기서 operator bool()을 실행하여 비트 값을 반환한다.

이는 정확하게 작동한다. bool처럼 쓰는데 아무런 문제가 없다. 하지만 다른 치명적인 문제가 있다. 아래를 보자.

#include <iostream>

#include <vector>

const int MAX = 1000;

class Widget {};

std::vector<bool> features(const Widget& w) {

return std::vector<bool>(MAX);

}

std::vector<int> features\_test(const Widget& w) {

return std::vector<int>(MAX);;

}

int main() {

Widget w;

std::vector<bool> vec\_b(1000);

bool b1 = false;

auto b2 = vec\_b[100];

std::cout << b1 << "\n";

std::cout << b2 << "\n";

// 아무런 문제가 없다.

// 바로 사라지는 vector에서 비교해보자.

bool b3 = features(w)[5];

// std::vector<bool>::reference priority

auto b4 = features(w)[5];

std::cout << b3 << "\n";

std::cout << b4 << "\n"; // runtime error: priority을 참조할 수 없다.

return 0;

}

왜 b4만 작동하지 않을까? 원인은 바로 reference의 작동 원리에 있다. 위의 코드의 동작을 자세히 보자. 위에서는 vector본체가 살아있지만 아래에서는 그렇지 못하다. 위의 변수bi의 대입 과정에서 모든 값이 복사되어야 정상이지만 b4는 그렇지 못하다. b4의 정체는 바로 포인터이다.

std::vector<bool>의 인스턴스 temp를 보자. 이는 함수 features에서 반환한 임시 객체이다. b4는 어떨까? 우리가 원하는 bool값을 전해줘야 하는 reference객체이다. **이는 본래의 저장소인 temp의 포인터를 참조하여 값을 반환**한다. 근데 temp는 임시객체이므로 사라져버리면 b4의 참조 값은 무효하게 된다.

### 표현식 템플릿의 횡포

표현식 템플릿(expression template)에서의 auto형식 연역 또한 문제가 될 수 있다. 표현식 템플릿은 ec++[항목48]에서 정리한 바 있다.

간단하게 정리하면 이렇다.

Power p1, p2, p3;

Power totalP = p1 \* p2 \* p3;

일반적인 연산자 오버로딩으로 위의 계산을 진행하면 임시 객체를 3번을 만들어야 한다. 하나의 임시 객체의 할당량이 많다면 이는 적은 연산은 아니다.

이를 해결하기 위해서, 컴파일 타임에 어떤 계산을 할지 계산식 자체를 합쳐버리는 것이다. 이것을 표현식 템플릿이라는 템플릿 기법으로 가능하게 한다.

// 실제 타입을 반환하도록 도와준다.

template<typename SubType>

struct Exp {

// returns const reference of the actual type of this expression

inline const SubType& self(void) const {

return \*static\_cast<const SubType\*>(this);

}

};

// TL의 타입 식과 TR의 타입 식을 더한다.

template<typename TLhs, typename TRhs>

struct BinaryAddExp : public Exp<BinaryAddExp<TLhs, TRhs> > {

const TLhs &lhs;

const TRhs &rhs;

BinaryAddExp(const TLhs& lhs, const TRhs& rhs)

: lhs(lhs), rhs(rhs) {}

// 각각의 식에서 구한 Eval을 더하여 값을 반환한다.

inline float Eval(int i) const {

return lhs.Eval(i) + rhs.Eval(i);

}

};

// 실질적인 원소 역할을 하는 클래스

struct Vec : public Exp<Vec> {

int len;

float\* dptr;

Vec(void) {}

Vec(float \*dptr, int len)

:len(len), dptr(dptr) {}

// here is where evaluation happens

template<typename EType>

inline Vec& operator= (const Exp<EType>& src\_) {

const EType &src = src\_.self();

for (int i = 0; i < len; ++i) {

dptr[i] = src.Eval(i);

}

return \*this;

}

// 최소 단위의 값을 반환한다.

inline float Eval(int i) const {

return dptr[i];

}

};

// operator+를 이용하여 연산을 합친다.

template<typename TLhs, typename TRhs>

inline BinaryAddExp<TLhs, TRhs>

operator+(const Exp<TLhs> &lhs, const Exp<TRhs> &rhs) {

return BinaryAddExp<TLhs, TRhs>(lhs.self(), rhs.self());

}

const int n = 3;

int main(void) {

float sa[n] = { 1, 2, 3 };

float sb[n] = { 2, 3, 4 };

float sc[n] = { 3, 4, 5 };

Vec A(sa, n), B(sb, n), C(sc, n);

auto K = B + C + C;

// k는 타입이 Vec이 아닌 Vec<...>이다. 잘못 연역된 것이다.

for (int i = 0; i < n; ++i) {

printf("%d:%f == %f + %f + %f\n", i,

A.dptr[i], B.dptr[i],

C.dptr[i], C.dptr[i]);

}

return 0;

}

K는 실제로 작동하는 방식대로 연역된 것이다. 하지만 이는 결국 Vec로 작용할 수 있다. 왜냐하면 여러 개의 타입이 합쳐져서 만들어진 식은 결국 Vec라는 값을 반환하게 되어있기 때문이다. 이것이 표현식 템플릿의 핵심 원리이며, auto의 연역은 경우에 따라서 제대로 작동하지 않을 수 있다.

## auto연역이 방해되는 문제 정리

결론은 어느 클래스에 숨겨진 ‘보이지 않는’ 대리자 클래스는 auto와 잘 맞지 않는다. 그러므로

**auto** someVar = *'보이지 않는' 대리자 클래스 형식의 표현식;*

위와 같은 형식을 피해야 한다. 혹여나 당장은 문제가 없더라도 잠재된 폭발적인 미정의 에너지를 품은 것이므로 주의해야 한다.

전체적인 내용을 정리하면, 암묵적으로 형식을 바꾸는 경우에 문제가 발생함을 이해할 수 있다. vector은 bool에 대해 특별히 작동하며, 템플릿 표현식은 오버로딩으로 어떤 압축된 타입을 암시적으로 바라고 있다.

# 올바른 auto의 연역을 위한 해결 방법

안 된다고 안 쓰는 것이 해결책이 아니다.

## 라이브러리 문서나 헤더파일 참고

명시되어 있는 주석이나 글을 참고하라는 말이다. 반대로 내가 프록시 객체를 이용하여 구현했다면 반드시 명시해주는 것이 올바른 자세라고 할 수 있겠다.

## 형식 명시 초기치 관용구

진정한 해법은 auto가 올바른 형식을 연역할 수 있도록 강제하는 것이다. *이것을* ***형식 명시 초기치 관용구***라고 한다. 다음과 같이 사용한다.

**auto** priority = **static\_cast**<bool>(features(w)[5]);

**내가 원하는 형으로 형 변환을 강제**한다. 이는 features가 할 수 있는 bool로의 변환을 가능하게 해준다. 여전히 임시객체는 살아있으므로 코드가 정상적으로 작동한다.

***“보이지 않는” 대리자 형식 때문에 auto가 초기화 표현식의 형식을 “잘못” 연역할 수 있다.***

***형식 명시 초기치 관용구는 auto가 원하는 형식을 연역하도록 강제한다.***

[항목7] 중괄호 초기치 사용

객체 생성 시, 괄호와 중괄호를 구분하라.

# 여러 가지 객체 초기화 방법

int a(0);

int b = 0;

int c{ 0 };

int d = { 0 };

괄호(중괄호)를 사용하는 방법과 대입 연산자를 사용하는 방법이 있다.

### 대입과 초기화는 다르다.

C++초보들은 =에서 무조건 배정(assignment)이 일어난다고 착각한다. 초기화에는 복사 생성자가 호출되며, 배정에는 복사 배정 연산자가 호출된다.

**class** **Archer**{

**public**:

Archer(int n) : level(n){ std::cout << "init" << '\n'; }

Archer **operator**=(**const** Archer& a){

level = a.level;

std::cout << "assignment" << '\n';

**return** \***this**;

}

**private**:

int level;

};

int main(){

Archer a(1); *//init*

Archer b = 1; *//init*

Archer c{ 5 }; *//init*

Archer d = { 5 }; *//init*

b = d; *//assignment*

}

# 중괄호 초기화를 사용해볼까?

언제나 원리를 이해하고 적절히 사용해야 한다.

## [동기] “어느 때에 어떤 생성자를 사용해야 할까?”

중괄호 초기치는 어느 때나 사용할 수 있다.

### 균일 초기화(uniform initialization)

위의 혼동을 완화하기 위해서 등장한다. 어느 경우에나 공통적으로 사용할 수 있는 초기화라는 의미다. 사실 이것이 {}를 이용한 초기화다. 다르게 *중괄호 초기화(braced initialization)*라고 부를 수 있다.

**private**:

*//클래스 안에서의 멤버 초기화*

int level{ 1 };

int level\_1 = 1;

*//int level\_2(1); //error*

};

int main(){

*//복사할 수 없는 객체*

std::atomic<int> ai1{ 0 };

std::atomic<int> ai2(0);

std::atomic<int> ai3 = 0; *//안 되야 하는데 됨*

std::cout << ai1 << ai2 << ai3 << std::endl;

## [장점1] 비정적 자료 멤버의 기본 초기화 값 지정

중괄호를 이용하면 다음과 같이 기본 값을 클래스 안에서 지정할 수 있다.

**private**:

*//클래스 안에서의 멤버 초기화*

int level{ 1 };

int level\_1 = 1;

*//int level\_2(1); //error*

};

편리하다.

## [장점2] 암묵적 좁히기 변환(narrowing conversion)을 방지한다.

‘어떤 값을 초기화하려는 객체의 형식으로 온전하게 표현할 수 있음이 보장되지 않는다면, 컴파일러에게 알려야 한다.’

정리하면, {}초기화는 온전히 객체에 담을 수 있음을 보장한다.

int main(){

long double a{ 1e10 }, b{1e11}, c{1e12};

*//error: long double에서 int로 축소 변환이 필요합니다.*

*//위의 값 3개를 더하면 int안에 담길까?*

int sum{ a + b + c };

std::cout << a << std::endl;

}

## [장점3] 가장 성가신 구문 해석에 자유롭다.

성가신 구문 해석을 피할 수 있다. 성가신 구문 해석이 무엇일까?

### 가장 성가신 구분 해석(most vexing parse)

“선언으로 해석 할 수 있는 것은 항상 선언으로 해석해야 한다.”라는 사명으로 가지고 인자가 없는 객체를 생성하면 이것을 함수 호출로 이해해버리는 현상을 말한다.

int main(){

Archer a1(10);

Archer a2(); *//이걸 생성자가 아닌 함수로 해석하여 호출이 안 됩니다.*

Archer a3{}; *//인수 없는 생성자 호출*

}

중괄호를 사용하면 위와 같은 문제를 고려할 필요가 없다.

## [단점1] std::initializer\_list가 인자로 있는 경우

예상치 못한 행동이 벌어질 수 있다. std::initializer\_list, 그리고 생성자 중복적재 해소 사이의 관계에서 비롯된다.  
*예상치 못한 행동*이란, 생성자가 여러 개가 있을 때(생성자 중복적재) std::initializer\_list를 받는 버전을 강력하게 선호한다는 것이다.

**class** **Archer**{

**public**:

Archer(int n) : level(n){ std::cout << "init" << '\n'; }

Archer(int n, bool b) {}

Archer(double d1, double d2) {}

Archer(std::initializer\_list<double> il)

{ std::cout << "il" << '\n'; };

Archer **operator**=(**const** Archer& a){

level = a.level;

std::cout << "assignment" << '\n';

**return** \***this**;

}

**private**:

int level{ 1 };

};

int main(){

*//아래 모두 Archer(std::initializer\_list<double> il)버전 사용*

Archer a1{ 10 };

Archer a2{ 10, true };

Archer a3{ 0.5, 0.9 };

Archer a4{ 10, 20, 30, 40, 50 };

Archer a5(0.5, 0.9); *//괄호를 사용하면 il버전을 사용하지 않는다.*

}

### {}가 std::initializer\_list버전을 받지 않는 경우

**class** **Archer**{

**public**:

Archer(int n) : level(n){ std::cout << "init" << '\n'; }

Archer(int n, bool b) {}

Archer(double d1, double d2) {}

Archer(std::initializer\_list<**int**> il){ std::cout << "il" << '\n'; };

**private**:

int level{ 1 };

};

int main(){

Archer a3{ 0.5, 0.9 }; *//error! initializer\_list<int>로 받을 수 없다.*

}

**결론은, 암시적 형 변환이 가능할 때만 std::initializer\_list를 받고 그렇지 않으면 포기한다.**

하지만 위와 같은 경우는 컴파일이 안 된다. double->int의 암시적 형 변환은 가능하지만 중괄호는 *암묵적 좁히기 변환*을 방지하기 때문이다.

다음은 아예 암시적 형 변환이 불가능하며 이는 각자 매개변수에 맞는 생성자를 불러온다.

**class** **Archer**{

**public**:

Archer(int n) : level(n){ std::cout << "init" << '\n'; }

Archer(int n, bool b) {}

Archer(double d1, double d2) {}

Archer(std::initializer\_list<std::**string**> il){};

};

int main(){

*//매개변수에 맞는 생성자 호출*

Archer a1{ 10 };

Archer a2{ 10, true };

Archer a3{ 0.5, 0.9 };

}

### 빈 {}는 무엇인가?

비어있는 {} == 인수 없음 🡪 기본 생성자 호출

int main(){

*//기본 생성자 호출*

Archer a1;

*//Archer a2(); //호출 안 됩니다. 함수 취급합니다.*

Archer a3{};

Archer a4({});

*//il호출*

Archer a5{ {} };

*//중괄호와 괄호가 달라서 아래와 같은 일도 생깁니다.*

std::vector<int> v1(10, 20); *//10개의 원소에 20으로 초기화*

std::vector<int> v2{ 10, 20 }; *//2개의 원소에 10, 20으로 각각 초기화*

}

그러므로 빈 {}로 std::initializer\_list를 호출하고 싶다면 ({})를 붙인다.

## [단점2] 템플릿에서의 괄호와 중괄호의 구분

이는 특히 짜증스럽다. 왜냐하면 둘 중 어느 것을 선택해야 할지 판단이 불가능하다. 다음과 같은 상황이다.

template<typename T, typename... Ts>

void bash(Ts&&... params)

{

// 둘 중 어느 것을 써야 할까?

T party1(std::forward<Ts>(params)...);

T party2{ std::forward<Ts>(params)... };

for (auto e : party1)

{

std::cout << "throw1: " << e << "\n";

}

for (auto e : party2)

{

std::cout << "throw2: " << e << "\n";

}

}

int main(){

bash<std::vector<int>>(10, 20);

}  
함수 안에서의 객체 생성에는 ()와 {}중 어느 것을 사용해야 할까? 템플릿 함수 작성자가 20을 10개 만들라는 것인지, 10과 20을 만들라는 것인지는 알 수 없다. 2가지 경우가 있기 때문에 사용자의 의도를 그대로 반영하여 템플릿을 만들기란 어렵다.

이런 경우에는 인터페이스의 일부에 문서화함으로써 문제를 해결한다. 이는 std::make\_unique에서도 볼 수 있는 문제이다. – [항목21] 참조

## 결론

중괄호 초기화를 권장한다. 하지만, 다음을 조심한다.  
1. 초기화 매개변수에 std::initializer\_list가 하나라도 있으면 주의해야 한다. 반대로, 작성 시 중괄호를 인자로 받는 오버로딩은 최대한 피하는 것이 좋다.  
2. 클래스 사용자로써 객체를 생성할 때, 괄호와 중괄호를 각각의 차이를 인지하여 세심하게 선택해야 한다. 벡터의 생성이 그 예시가 될 것이다.

***중괄호 초기화는 가장 광범위하게 적용할 수 있는 초기화 구문이며, 좁히기 변환을 방지하며, C++의 가장 성가신 구문 해석에서 자유롭다.***

***생성자 중복적재 해소 과정에서 중괄호 초기화는 가능한 한 std::initializer\_list 매개변수가 있는 생성자와 부합한다.***

***괄호와 중괄호의 선택의 의미 있는 차이를 만드는 예는 vector이다.***

***템플릿 안에서 객체를 생성할 때 괄호를 사용할 것인지 중괄호를 사용할 것인지 선택하기가 어려울 수 있다.***

[항목8] nullptr의 사용

0과 NULL보다는 nullptr을 선호하라.

# T\* ptr은 아무데도 가리키기 싫다.

해당 포인터가 비어있다라고 표현하기 위해서 어떻게 해야 할까?

## 0, NULL의 사용

기존에 사용하던 방법이다. 지금도 사용되고 있다. 하지만 이는 실제로 0인 int형일 뿐이다. C++은 포인터만 사용할 수 있는 위치에 0이 있으면 그것을 마지못해 null포인터로 해석하기 때문에 이런 표현이 가능하다.

#include *<iostream>*

int main(){

char\* p = NULL;

printf("NULL = %d**\n**", p); *// NULL = 0*

**if** (p){

printf("%d**\n**", p);

}

**if** (p == NULL){

printf("NULL = %d**\n**", p);

}

*//제대로 파일이 열렸는지 확인*

**if** (fopen("test", "r") == NULL){

printf("NO FILE.**\n**");

}

**return** 0;

}

실제로 값은 0을 나타냄을 보여주는 코드이다. 만약 비교를 하게 된다면, NULL은 포인터 형식으로 바뀌어야 할 것이고, 이것을 C++은 null포인터로 해석하게 된다. 그러므로 해당 포인터가 비어있는지 확인하는 코드도 작성할 수 있다.

## nullptr의 사용

C++11이후부터는 위와 같이 구닥다리 방식은 사용하지 않는다. ***nullptr***은 비어 있음을 뜻하는 모든 형식의 포인터다. 그러므로 모든 원시 포인터에 대해 암시적 변환이 가능하다.

### nullptr

typedef decltype(nullptr) nullptr\_t;

#include *<cstddef>*

#include *<iostream>*

void f(int\* pi)

{

std::cout << "Pointer to integer overload**\n**";

}

void f(double\* pd)

{

std::cout << "Pointer to double overload**\n**";

}

void f(std::nullptr\_t nullp)

{

std::cout << "null pointer overload**\n**";

}

int main()

{

int\* pi; double\* pd;

f(pi);

f(pd);

f(**nullptr**); *// would be ambiguous without void f(nullptr\_t)*

*// f(NULL); // 실제는 0이므로 int에도 double에도 작동한다.*

}

## NULL vs. nullptr

사실 많은 경우에 두 가지 모두 사용할 수 있다. 하지만 분명 실제로 int형인 NULL의 한계는 존재한다. 어떠한 경우에 그러한가?

### 의외의 오버로딩 문제

실제로 int형식이기 때문에 NULL은 오버로딩에서 문제가 발생한다.

void f(int pi)

{

std::cout << "integer overload**\n**";

}

void f(bool pb)

{

std::cout << "bool overload**\n**";

}

void f(void\* func)

{

std::cout << "func pointer overload**\n**";

}

*/\*void f(std::nullptr\_t nullp)*

*{*

*std::cout << "null pointer overload\n";*

*}\*/*

int main()

{

f(0);

f(NULL); *//이 2가지는 절대로 void\*를 호출하지 않는다. int를 호출한다.*

f(**nullptr**); *// void\*를 호출한다.*

}

**실제로 int형이므로 포인터 형식보다 int형의 매개변수를 우선적으로 받게 된다.** long타입이라면 void\*타입으로 전환될 여지가 있지만 불안정한 것은 마찬가지다.

### 코드의 명확성

**nullptr은 코드의 명확성을 높여준다. auto가 관여하면 더더욱** 그런데, 이게 정수에 대한 비교인지, 포인터에 대한 비교인지를 명확히 한다.

int func(int n){ **return** n; }

int\* func(int\* p){ **return** p; }

int main()

{

int a = 5;

int \*p = &a;

**auto** ret = func(p);

**if** (ret == NULL); *//둘 다 작동한다. 모호하다.*

**if** (ret == **nullptr**); *//인자 a에 대해서는 작동하지 않는다.*

}

### nullptr의 템플릿 관여

다음은 0, NULL, nullptr을 인자로 넘겨 포인터를 매개변수로 받는 함수를 실행한다. 기대했던 대로 모두 정상적으로 실행된다.

**class** **Widget**{};

*//포인터를 인자로 받는다. 0과 NULL도 받을 수 있다.*

int f1(std::shared\_ptr<Widget> spw){

std::cout << "int f1**\n**";

**return** spw != NULL;

}

double f2(std::unique\_ptr<Widget> upw){

std::cout << "double f2**\n**";

**return** upw != NULL;

}

bool f3(Widget\* pw){

std::cout << "bool f3**\n**";

**return** pw != NULL;

}

int main(){

std::mutex f1m, f2m, f3m;

**using** MuxGuard = std::lock\_guard<std::mutex>; *//별칭 선언*

MuxGuard g1(f1m);

**auto** result1 = f1(0);

MuxGuard g2(f2m);

**auto** result2 = f2(NULL);

MuxGuard g3(f3m);

**auto** result3 = f3(**nullptr**);

}

0과NULL은std::shared\_ptr<Widget>로 변환되기 때문에 가능하다. 이는 다음의 생성자를 호출하게 된다.

constexpr shared\_ptr(nullptr\_t) \_NOEXCEPT

{ // construct empty shared\_ptr

}

한마디로 0(NULL) -> nullptr\_t로의 암시적 변환이 이루어지는 것이다.

위의 코드만 보면 문제가 없는 것이다. 하지만 3가지 함수는 패턴이 같다. 그러므로 템플릿화하여 코드를 줄일 수 있다.

**template**<**typename** FuncType, **typename** MuxType, **typename** PtrType>

**auto** lockAndCall(FuncType func, MuxType& mutex, PtrType ptr)

-> **decltype**(func(ptr))

{

**using** MuxGuard = std::lock\_guard<MuxType>;

MuxGuard g(mutex);

**return** func(ptr);

}

int main(){

std::mutex f1m, f2m, f3m;

**auto** result1 = lockAndCall(f1, f1m, 0); *//error*

**auto** result2 = lockAndCall(f1, f1m, NULL); *//error*

**auto** result3 = lockAndCall(f1, f1m, **nullptr**);

}

더 이상 0과 NULL을 포인터인 척 넘길 수 없게 되었다. 왜냐하면 템플릿에서 추론하기를 0과 NULL은 int형식이 되기 때문이다. std::shared\_ptr<Widget>로 바로 넘기면 0과 NULL은 nullptr\_t로 암시적 변환이 일어나겠지만, 연역하는 과정이 끼어든다면 상황이 달라진다. **결국, 0과 NULL은 암시적 변환이 작용하지 않는 상황이라면 작동하지 않는 불안정한 방법**인 셈이다.

***0과 NULL보다 nullptr을 선호하라.***

***정수 형식과 포인터 형식의 오버로딩을 피해라.***

[항목9] 별칭 선언 사용

typedef보다는 별칭 선언을 선호하자.

typedef의 관한 자세한 내용은 ec++문서를 참고한다.

# C++11의 별칭 선언

**class** **BladeMaster**{

**public**:

**class** **DragonSlayer**{};

};

*//두 번 다시 쓰기 싫다.*

**using** myMap = std::multimap<std::string, BladeMaster::DragonSlayer>;

typedef랑 똑같다.

# typedef와 구별되는 별칭 선언의 장점

뭔가 다르니까 새로 나왔겠다.

### 함수 포인터

**typedef** void(\*FP)(int, **const** std::string&);

**using** FP = void\*(int, **const** std::string&);

using으로는 이렇게 표현한다. 좀 더 직관적이라는 것을 알 수 있다.

### 템플릿으로의 선언

별칭 선언은 템플릿화할 수 있다. 반면 typedef는 템플릿화할 수 없다. typedef는 하나의 구조체를 만들어서 씌어야 템플릿을 적용시킬 수 있다. 게다가 이를 사용하려면 typename을 붙여서 의존적 형식을 만들어 사용해야 하므로 코드가 길어지고 지저분하다.

아래의 코드에서 차이를 살펴보자.

*//C++11*

**template**<**typename** T>

**using** myMapUsing = std::map<std::string, T>;

**template**<**typename** T>

**class** **Widget**{

**private**:

myMapUsing list;

};

*//구닥다리*

**template**<**typename** T>

**struct** myMapTypedef{

**typedef** std::map<T, T> type;

};

**template**<**typename** T>

**class** **Widget**{

**private**:

**typename** myMapTypedef<T>::type list;

};

#### typedef에서는 typename과 type이 꼭 필요하다.

typename은 이것이 타입으로 쓰인다라는 선언이다. type이라는 것이 어떠한 변수일 수도 있을 수 있기 때문이다. 그러므로 type은 전적으로 T가 무엇이냐에 따라 달라지므로 의존된다. template에서의 typedef사용의 자세한 내용은 ec++[항목42] 참고

# 형식 특질(type trait)과 별칭 선언

템플릿을 사용한다면, 그 타입이 템플릿 인스턴스마다 다르므로, 해당 타입을 확인해야 할 경우가 생긴다. 그리고 이것을 제어해야 하는 경우가 생기는데, 이러한 제어를 제공하는 **인터페이스**를 *형질**특질*이라고 한다. 이는 컴파일 타임에 시행된다.

## 형질 특질의 예시

형질 특질로 많은 일들을 할 수 있다. 사전: <http://en.cppreference.com/w/cpp/types>

### 해당 타입에 관한 검사

아래는 해당 타입이 널인지 아닌지를 검사한다. 이 외에도 거의 모든 것이 다 있다.

int main()

{

auto ptr = nullptr;

std::cout << std::is\_null\_pointer<decltype(ptr)>::value << '\n';

std::cout << std::is\_null\_pointer\_v<decltype(ptr)> << '\n';

return 0;

}

### 조건 검사

어떤 2타입에 대해서 조건이 일치하는지를 검사한다.

std::cout << std::is\_same<int, int32\_t>::value << '\n'; // true

### 타입 수정

typedef std::remove\_cv<const volatile int\*>::type type4;

std::cout << "test4 " << (std::is\_same<const volatile int\*, type4>::value

? "passed" : "failed") << '\n'; // out: passed

### 기타 변형

// 이 함수 템플릿은 T와 Archer와 같을 때에만 작동한다.

template<class T>

typename std::enable\_if<std::is\_same<T, Archer>::value>::type

target()

{

std::cout << "good archer.\n";

}

target<Archer>();

//target<Widget>(); // error

## C++11의 형식 특질

const를 없애는 수행 등의 타입의 군더더기를 제어하는 작업이다. 이것이 아래처럼 using을 사용하게 새로이 포함시켰다. 기존 방법과 비교해본다.

다양한 형 변환 형식: <http://en.cppreference.com/w/cpp/types>

std::remove\_const<T>::type *//typedef버전*

std::remove\_const\_t<T> *//using버전*

***typedef는 템플릿화를 지원하지 않지만, 별칭 선언은 지원한다.***

***별칭 템플릿에서는 typename접두사, ::type접미어를 붙일 필요가 없다.***

***C++14는 C++11의 모든 형식 특질 변환에 대한 별칭 템플릿을 제공한다.***

[항목10] enum class 사용

범위 없는 enum보다는 범위 있는 enum을 사용하자.

### C++11 - the new ISO C++ standard

<http://www.stroustrup.com/C++11FAQ.html#enum>

# enum

enum의 기본에 대해서 알아보자.

## C++98’s enum

기존의 enum은 범위 없는(unscoped) enum이 된다.

**enum** *name* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**

**enum** *name* **:** *type* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **} (C++11)**

**enum** *name* **:** *type* **; (C++11)**

### 열거형(enumeration)이란?

이름 있는 정수형 상수들을 열거하여 새롭게 만들어진 사용자 정의 데이터형(user-defined data type)입니다.

## C++11’s enum

범위 있는 enum이라고 부르며 enum class로 선언한다.

enum AttackType

{

normal = 0,

piercing = 1,

magic = 2,

};

enum class ArmorType : std::uint32\_t

{

iron,

magic,

electronic,

};

다음과 같은 경우 목록 초기화를 사용하여 캐스트없이 정수에서 초기화 할 수 있습니다.

enum byte : unsigned char {}; // byte is a new integer type

byte b{ 42 }; // OK as of C++17 (direct-list-initialization)

byte c = { 42 }; // error

byte d = byte{ 42 }; // OK as of C++17; same value as b

byte e{ -1 }; // error

struct A { byte b; };

A a1 = { { 42 } }; // error

A a2 = { byte{ 42 } }; // OK as of C++17

void f(byte);

f({ 42 }); // error

enum class Handle : std::uint32\_t { Invalid = 0 };

Handle h{ 42 }; // OK as of C++17

# enum class와 enum의 차이

둘 간에는 약간의 차이가 있는데 무엇인지 알아보자.

## 이름 공간의 오염 방지

**enum안에서 정의한 이름을 밖에서 사용할 수 있는지 없는지**에 관한 내용이다. 기존 enum은 범위 없는 enum이므로 말 그대로 범위가 없다. 그래서 enum에서 정의된 이름은 범위 밖에서 선언될 수 없다. 하지만 enum class는 범위 있는 enum으로써, 이 안에서 정의된 이름은 해당 범위 안에서만 쓰이기에 같은 이름도 변수로 쓰일 수 있다.

enum AttackType

{

normal = 0,

piercing = 1,

magic = 2,

};

int main()

{

std::cout << sizeof(AttackType::normal) << '\n'; // 4

std::cout << std::is\_enum\_v<AttackType> << '\n'; // 1

auto normal = 10; // 작동 시, 에러...

return 0;

}

표준과는 약간 다른 작동을 보인다. 기본으로 선언된 enum의 형식과 범위에 관한 내용이 그것이다. 결론은 컴파일러마다 다르다.

## 암시적 변환 방지

**자동 형 변환이 일어나지 않는다.** 이는 때로는 단점이 될 수 있기는 하지만 안전성의 면에서는 우월하다.

int main(){

// 형식을 직접 지정한다.

// 직접 지정하지 않으면 int형이다.

enum class Color : long long { RED, BLUE, BLACK };

auto RED = false; // 변수로써 가능, 위의 RED라는 이름은 열거자 범위 안에서만 적용

Color c = Color::RED; // 이렇게 써야 한다.

// if (c < 14.5)// int->double이 안 된다는 말이다.

if (static\_cast<double>(c) < 14.5) // 쓸려면 이렇게 써야 한다.

{

std::cout << "no color\n";

}

// 하지만 열거자의 타입이 일정치 않을 수도 있으므로 일반화시킨다.

if (static\_cast<std::underlying\_type\_t<Color>>(c) < 14.5)

{

std::cout << "no color\n";

}

auto d = Color::RED;

}

만약 enum class에서 변환을 원한다면 명시적으로 캐스팅해주면 가능하다. 그리고 타입을 얻어내어 일반화시키는 방법도 볼 수 있다.

### std::tuple<Ts…>

pair의 확장판이라고 볼 수 있다. 이동 의미론에 따라, 많은 인자를 완벽 전달로 넘겨서 생성자 중복 호출 없이 시간을 절약하게 해준다.

std::vector<

std::tuple<std::vector<int>, std::string, int, int>

> datas;

int main(){

auto d = std::make\_tuple(std::vector<int>(5, 0), "ytk", 9999, 9999);

datas.push\_back(d);

if (datas.size() > 0)

{

//get: 해당 tuple의 <1>번째 원소를 가져온다.

std::cout << std::get<1>(datas[0]) << '\n';

}

return 0;

}

### 컴파일 타임에 enum class캐스팅

그런데 컴파일 도중에 해당 enum의 값을 캐스팅하여 알고 싶다면 어떻게 해야 하는가? std::tuple를 이용하여 예시를 살펴보자. 열거자의 1번째에 속하는 색의 원소를 꺼내고 싶다면, 열거자를 size\_t로 형변환을 해야 할 것이다.

std::cout << std::get<Color::BLACK>(datas[0]) << '\n'; //컴파일 도중에 완성되어야 하므로 안 된다.

위의 코드는 Color::BLACK은 정수 타입이 아니다. 캐스팅되어야 하는데 컴파일 타임에 완성되어야 한다. 어떻게 하면 캐스팅을 컴파일 타임에 완성시킬 수 있을까? 그러므로 다음과 같이 한다.

std::cout << std::get<static\_cast<size\_t>(Color::RED)>(datas[0]) << '\n';

성공적으로 컴파일된다. 사실 static\_cast를 사용한 것 말고 별 다른 것이 없다.

이를 함수로 만들어보자. 타입을 반환해야 하는데 이런 함수를 어떻게 만들 것인가? Constexpr을 이용하는 것이다. – [항목14] 참조

template<typename E>

constexpr typename std::underlying\_type\_t<E> toUType(E enumerator) noexcept

{

return static\_cast<std::underlying\_type\_t<E>>(enumerator);

}

std::cout << std::get<toUType(Color::RED)>(datas[0]) << '\n';

### 전방 선언

기존 enum은 메모리와 공간을 효율적으로 이용하기 위해, 컴파일러가 최적의 타입을 알아서 정해준다. 이러한 작동이 생기려면 당연히 선언과 동시에 정의가 되어야 한다. 그러므로 전방 선언을 허용하지 않았다.

하지만 이러면 컴파일 의존 관계가 늘어난다는 단점이 있다. 왜냐하면 열거자 안의 내용을 추가할 시, 시스템 전체를 다시 컴파일 해야 하기 때문이다.

enum class에서는 전방 선언이 가능하므로 이를 방지할 수 있다. 미리 타입을 사용자가 직접 정할 수 있기 때문에 가능한 것이다.

enum class Color : unsigned int;

void func(Color); // 다시 컴파일 하지 않아도 된다.

int main(){

// 형식을 직접 지정한다.

// 직접 지정하지 않으면 int형이다.

enum class Color : long long { RED, BLUE, BLACK, DRAGON };

}

위에서 DRAGON이 추가된다. Color이 바뀌었으므로 이의 선언만 func함수에서도 모두 컴파일해야 한다. 하지만 전방 선언을 해두면 다시 할 필요가 없는 것이다.

## 범위 없는 enum은 존재할 필요가 없는가?

범위 없는 enum이 유용할 때가 있는데, 역시 암시적 변환을 요구할 때이다. 양날의 검이라고 볼 수 있다. 사용자의 코드를 편하게 작성할 수 있게 해주는 반면, 오류가 날 위험에 도사린다.

그런데 사실상 VC++13을 기준으로 enum도 전방 선언과 타입 설정 가능하다. 위에서 제시한 표준과는 다른 작동이다. 물론 변수 중복 선언과 암시적 변환에서는 둘이 다른 작동을 보인다. enum과 enum class의 구분을 최소화하기 위한 방침으로 보인다.

***C++98스타일의 enum을 이제는 범위 없는 enum이라고 부른다.***

***범위 있는 enum의 열거자들은 그 안에서만 보인다. 이 열거자들은 오직 캐스팅을 통해서만 다른 형식으로 변환된다.***

[항목11] 삭제된 함수 사용

정의되지 않은 비공개 함수보다 삭제된 함수를 선호하라.

# 기본 버전에서 함수 삭제하기

ec++의 [항목6]에서 함수를 금지하는 방법이 나온다. 비공개 함수에 정의하지 않는 방법이다.

class WisdomSword

{

public:

WisdomSword() {}

void info() { std::cout << "this is unique sword.\n"; }

private:

WisdomSword(WisdomSword&);

WisdomSword& operator=(WisdomSword&);

};

정의되지 않은 함수를 다른 클래스로부터 private상속받는 방법으로 일반화시킬 수도 있다.모든 입츌력 클래스의 기반이 되는 std :: basic\_ios 클래스가 이렇게 구현되어 있다.

# C++11에서 함수 삭제하기

사용법: func(param) **= delete**

**class** **Archer**{

**public**:

Archer() {}

Archer(**const** Archer& a) = **delete**;

Archer& **operator**=(**const** Archer& a) = **delete**;

};

아래에는 몇 가지 특징에 대해 언급한다.

## 더 이른 오류 보고

멤버 함수나 friend함수에서 삭제된 함수를 호출하려고 하면 컴파일 에러를 보인다. 반면, 정의되지 않은 함수를 private상속하면 링크 시점에서야 오류를 보인다. 삭제된 함수는 더 빨리 에러를 파악할 수 있는 것이다.

## private보다는 public으로 선언하기

삭제할 함수를 public으로 선언하면 더 나은 오류메시지를 보여준다.

## 어떤 함수도 삭제할 수 있다.

기존 방법은 멤버 함수만 방지할 수 있지만, 삭제자는 비멤버 함수에도 가능하다.

bool isStrong(int level) { **return** level > 100; }

bool isStrong(bool b) = **delete**;

bool isStrong(double d) = **delete**;

int main(){

isStrong(105);

isString(3.5); *// error*

}

어떠한 인자에 int형만 넣고 싶고 암시적으로 변할 수 있는 bool이나 double을 배제하고 싶다면 위의 방법이 아주 적절하다. 오버로딩하여 원하지 않는 버전을 삭제하는 것이다.

참고로 위에서는 float도 배제된다. float버전은 오버로딩 되어있지 않고 int형으로 암시적으로 변환될 수 있지만, int보다는 double로의 변환을 우선시한다는 점을 볼 수 있다.

**이러한 원리가 작동할 수 있는 원리는 삭제되기는 했지만 컴파일러가 선택할 후보에는 남아있기 때문이다.** 들어갈 수 있는 입구는 남아있고 그것에 지뢰를 놓는 것이 삭제자 키워드의 역할인 셈이다.

## 원하지 않는 템플릿 인스턴스화 방지

템플릿에도 사용할 수 있다. 템플릿 특수화로 특정 템플릿에 대해 다시 정의할 수 있는데, 이는 특수한 타입에 인스턴스화를 막을 수 있다.

**class** **Archer**{

**public**:

Archer() {}

Archer(**const** Archer& a) = **delete**;

Archer& **operator**=(**const** Archer& a) = **delete**;

**template**<**typename** T>

void shot(T \*damage){

std::cout << \*damage << '\n';

}

};

*//void\*에 대해서만 호출을 금합니다.*

**template**<>

void Archer::shot<void>(void\*) = **delete**;

예를 들어, shot이란 함수는 T를 타입으로 받는데 void나 char에 대해 금지하고 싶다면 delete로 지우면 된다.

반면, 기존 버전의 삭제 방식으로는 위와 같이 할 수 없다. 왜냐하면 템플릿 특수화는 클래스 범위가 아니라 이름 공간 범위에서 작성되어야 하기 때문이다.

class Archer{

public:

Archer() {}

Archer(const Archer& a) = delete;

Archer& operator=(const Archer& a) = delete;

template<typename T>

void shot(T \*damage){

std::cout << \*damage << '\n';

}

private:

/\*template<>

void shot<void>(\*void); // 클래스 내에서는 특수화가 안 된다.\*/

};

template<>

void Archer::shot<void>(void\*)

{

// 기본 버전에서는 이렇게 분리해야 하는데

// 삭제할 방법이 없다.

}

***정의되지 않은 비공개 함수보다 삭제된 함수를 선호하자.***

***비멤버 함수와 템플릿 인스턴스를 비롯한 그 어떤 함수도 삭제할 수 있다.***

[항목12] override 사용

재정의 함수들을 override로 선언하라.

# overriding vs. overloading

재정의와 함수 적재. 말 그대로 전자는 기본 클래스의 함수를 재정의하며, 후자는 같은 함수를 여러 개 쌓아놓는다. 자세한 비교는 ec++문서 참조

# override키워드

이전에도 우리는 재정의를 아주 잘 써왔다. 이게 왜 필요한가?

## 재정의가 일어날 조건

재정의가 일어나려면 다음과 같은 필수 조건을 만족해야 한다.  
1. 가상 함수이어야 한다.  
2. 함수 이름이 같아야 한다.  
3. 매개 변수가 같아야 한다.  
4. const성이 같아야 한다.  
5. 반환 형식과 예외 명세가 반드시 호환되어야 한다.  
6. 참조 한정자들이 동일해야 한다. - new C++11

위를 하나라도 어기면 재정의가 일어나지 않고 문제가 발생한다.

## 문제: 재정의 실수

재정의가 실수하면 컴파일러가 친절하게 알려줄까? 그렇지 않다. 다음 함수를 보자.

**class** **Archer**{

**public**:

**virtual** void attack() **const**;

**virtual** void run(int speed);

**virtual** void casting() &;

void upgrade() **const**;

};

**class** **DarkKnight** : **public** Archer{

**virtual** void attack();

**virtual** void run(long long speed);

**virtual** void casting() &&;

void upgrade() **const**;

};

모두 위의 재정의 조건에 부합하지 않지만 컴파일 오류는 없다.

[미결] casting() &(&&)은 오류로 나온다. 문법이 적법하지 않은 것 같다.

## 해결: override 선언

override를 선언해줌으로써 무엇이 잘못되었는지 친절히 알려줄 것이다.

**class** **DarkKnight** : **public** Archer{

**virtual** void attack() **override**;

**virtual** void run(long long speed) **override**;

**virtual** void casting() && **override**;

void upgrade() **const** **override**;

};

모두 컴파일이 안 된다.

# 멤버 함수 참조 한정자

casting() &(&&) { … } 이것을 말한다.

## 이게 무슨 의미인가?

멤버 함수를 왼값(&)에만 혹은 오른값(&&)에만 사용할 수 있게 한정할 수 있다.

**class** **Archer**{

**public**:

**virtual** void attack() **const**;

**virtual** void run(int speed);

**virtual** void casting() & { cout << "lvalue**\n**"; }

**virtual** void casting() && { cout << "rvalue**\n**"; }

void upgrade() **const**;

**private**:

int value;

};

Archer setArcher(){

**return** Archer();

}

int main(){

Archer a; *// lvalue*

Archer setArcher();*// rvalue*

*//아래 2개는 다른 함수를 불러옵니다.*

a.casting(); *// lvalue*

setArcher().casting(); *// rvalue*

}

멤버 함수 뒤에 &(&&)를 붙이면 호출 객체가 왼값(오른값)인 경우, &(&&)를 호출한다.

## 어디에 쓸까?

어떠한 함수가 객체가 왼값이냐 오른값이냐에 따라 다른 경우다. 역시 무언가를 복사할 때다. 오른값은 포인터만 옮길 수 있으므로 왼값의 동작과 차별되어야 한다.

**class** **Skill**{

**public**:

**private**:

int power;

};

**class** **Archer**{

**public**:

**virtual** void attack() **const**;

**virtual** void run(int speed);

**virtual** Skill& casting() & { **return** s; }

**virtual** Skill&& casting() && { **return** std::move(s); }

void upgrade() **const**;

**private**:

int value;

Skill s;

};

***재정의는 override로 선언하라.***

***멤버 함수 참조 한정자를 사용하면 호출되는 객체 \*this의 왼값 버전과 오른값 버전을 구분할 수 있다.***

[항목13] const\_iterator 선호하기

iterator보다 const\_iterator를 선언하자.

# iterator vs. const\_iterator

반복자라 불리는 포인터이다. 전자는 내용을 수정할 수 있으며, 후자는 그럴 수 없다. 글자 그대로 후자는 전자의 상수 버전이다.

int main()

{

std::vector<int> item\_category = { 1, 2, 3, 4, 5, 6, 7 };

std::vector<int>::iterator **iter** = item\_category.begin();

std::vector<int>::const\_iterator **cIter**= item\_category.end() **- 1**;

while (iter != item\_category.end() **- 1** && cIter != item\_category.begin())

{

std::cout << "now state: " << \*iter << " : " << \*cIter << '\n';

\***iter** = -1;

//\***cIter** = -1; // 상수이므로 변경할 수 없다.

if (iter++ == cIter--)

{

std::cout << "nice to meet you.\n";

break;

}

}

return 0;

}

물론, 포인터를 사용하면서 그 내용물을 수정하는 것은 유지보수 면에서 그리 바람직하지 않다. 그러므로 const\_iterator를 사용하는 것이 좋은데, 이에 대해 논의해보자.

# C++11에서 달라진 const\_iterator

무엇이 달라졌길래 강조하는 것일까?

## 무엇이 문제일까? : 기존의 const\_iterator

예를 들어, iterator를 이용하여 어떠한 값을 찾는다고 해보자. 그럼 다음과 같이 작성할 수 있다.

typedef std::vector<int>::iterator IterT;

int main()

{

std::vector<int> vec{1,2,3,4,5,6,7,8,9,10,11,12};

IterT it = std::find(vec.begin(), vec.end(), 7);

vec.insert(it, 9999); // 7이 있었던 자리에 9999를 넣는다.

for (auto v : vec) std::cout << v << " ";

return 0;

}

아무런 문제가 없다. 하지만 설계 면에서 잘 생각해보자. 탐색만 하는 위의 코드에서 iterator의 내용이 변경될 수 있어야 맞는 걸일까? 우리는 상수 버전을 사용해야 한다.

typedef std::vector<int>::const\_iterator CIterT;

int main()

{

std::vector<int> vec{1,2,3,4,5,6,7,8,9,10,11,12};

CIterT cit = std::find(static\_cast<CIterT>(vec.begin()),

static\_cast<CIterT>(vec.end()), 9999);

vec.insert(static\_cast<IterT>(cit), 10); //상수에서 비상수로 전환이 안 됨

std::cout << \*cit << '\n';

return 0;

}

상수 버전의 iterator를 사용하긴 했는데 상수 버전의 iterator를 얻으려면 인자 또한 상수 버전으로 캐스팅이 필요하다. 벌써부터 머리가 띵해지지만 하나가 더 있다. 도중에 삽입이 필요하다고 할 때, C++98에서는 이를 상수 인자로 받을 수 없다. 이에 상수화를 제거해야 하는데 이런 캐스팅은 허용하지 않는다. 그냥 const\_iterator은 iterator로 변경될 수 없다.

**결론은, 기존의 C++에서는 상수버전을 사용하기가 너무 어렵다는 것이다.**

## 어떻게 해야 할까? : C++11에서의 const\_iterator

C++11에서는 const\_iterator을 사용하기도 쉽고, 얻기도 쉽다. 무엇이 이를 가능하게 하는가?

### cbegin, cend

cbegin과 cend로 상수 버전의 iterator을 돌려준다. 모든 컨테이너 함수들은 기본 반환 자체를 const버전으로 하며, 인자 또한 상수 버전을 받도록 되어 있다.

int main()

{

std::vector<int> vec{1,2,3,4,5,6,7,8,9,10,11,12};

auto cit = std::find(vec.cbegin(), vec.cend(), 10); // CIterT cit

vec.insert(cit, 9999); // 상수 버전인데도 너무 쉬움

std::cout << \*cit << '\n';

return 0;

}

### 모든 함수에서 값을 상수로 받는다.

굳이 비상수 버전의 iterator를 넣으려고 노력하지 않아도 된다. 상수도 환영이다.

vec.insert(it, 10000); *// it은 const\_iterator입니다.*

# iterator에 대한 일반적인 함수 제작

상수 버전이든 비상수 버전이든 모두 사용할 수 있는 함수를 만드는 것도 가능할까? 어떻게 만들면 좋을까?

## 비멤버 함수가 필요하다.

멤버 함수는 어떤 객체의 존재를 필수로 한다. 하지만 일반적인 경우라면 내장 배열이나 그런 존재가 없는 인터페이스일 경우도 있다. 멤버 함수도 존재하고 비멤버 함수도 존재한다. vec.begin()과 begin(vec)이 한 예시이다. 일반성을 극대화하기 위해 존재하는 것이다.

이를 적용하여 코드를 만들면 다음과 같다.

template<typename C, typename V>

void findAndInsertAndPrint(C& container, const V& targetVal, const V& insertVal)

{

auto iter = std::find(cbegin(container), cend(container), targetVal);

container.insert(iter, targetVal);

for (iter = cbegin(container); iter != cend(container; iter++)

{

std::cout << \*iter << " ";

}

}

중요한 것은 무엇인가? 비멤버 함수로 구현되었다는 점이다.

**비멤버 함수가 뭐가 좋길래 활용해야 하는가?**

1. 비멤버 함수는 클래스의 무게를 줄여 준다.  
  ① 이는, 좀더 유지보수가 편해 지게 해 주는 초석이 된다.  
  ② 가독성이 올라 간다. 왜냐구? 봐야 할게 적으니까~  
  
2. General Programming을 할 수 있다.  
  ① 이는, 한번 만들어 둔것을 손쉽게 재사용 할 수 있게 해주는 초석이 된다.  
  ② 코드의 재사용으로 역시 알아야 할 것이 줄어 들게 된다.

만약 위의 코드가 멤버 함수로 구현되어 있다고 생각해보자.

template<typename C>

class MyVector

{

public:

template<typename V>

void findAndInsertAndPrint(const V& targetVal, const V& insertVal)

{

auto iter = std::find(vec.cbegin(), vec.cend(), targetVal);

vec.insert(iter, targetVal);

for (iter = vec.cbegin(); iter != vec.cend(); iter++)

{

std::cout << \*iter << " ";

}

}

private:

std::vector<C> vec;

};

일단 멤버 함수는 비멤버 함수보다 캡슐화 정도에서 약하다. 당연한 것이 멤버 함수는 클래스 안에 있으므로 본인의 자료에 마음대로 접근할 수 있기 때문이다.

물론 크게 다른 점은 없다. 하지만 멤버 함수는 수정이 어렵다는 것을 볼 수 있다. 어떤 함수의 기능이 바뀌었다고 하자. 비멤버 함수는 그 함수 하나만 바꾸면 되지만 멤버 함수는 그 기능에 해당하는 모든 클래스의 멤버 함수를 수정해야 한다.

게다가 유연하지 못하다. 다른 함수를 묶어서 하나의 기능으로 만들고자 하는 경우, 멤버 함수로 만들면 그것이 상당히 제한됨을 알 수 있다.

물론 어떤 객체의 행위를 분명하게 해주지만 사용면에서는 그렇지 않을 수 있으므로 항상 고려해야 할 대상이다.

## 상수 버전이 비상수 버전을 호출한다.

**이는 상수 버전이 없는 컨테이너에 대해서도 작동한다는 장점이 있다.** 이것이 비멤버 함수가 유연한 이유 중 하나다. 어떤 타입이건 적용하기 쉽다.

**template**<**typename** C>

**auto** cbegin(**const** C &container) -> **decltype**(std::begin(container))

{

**return** std::begin(container); *//반환 값: const\_iterator*

}

중요한 것은 유연한 비멤버 함수지만 이는 ec++[항목3]에서도 등장한다. 멤버 함수에서도 상수 버전이 비상수 버전을 호출하여 코드량을 줄인 내용이다. 이런 변환은 효율적인 프로그래밍에 필수적이다. 비멤버 함수는 이가 더 유연하다는 말이다.

### rbegin, rend

참고로 알아두자. 뒤집어서 접근할 수 있는 iterator을 제공한다.

int main()

{

std::vector<int> myvector(5); // 5 default-constructed ints

int i = 0;

std::vector<int>::reverse\_iterator rit = myvector.rbegin();

for (; rit != myvector.rend(); ++rit)

\*rit = ++i; // 뒤에서부터 원소를 채운다.

std::cout << "myvector contains:";

for (std::vector<int>::iterator it = myvector.begin();

it != myvector.end(); ++it)

std::cout << ' ' << \*it; // 5 4 3 2 1

std::cout << '\n';

return 0;

}

***iterator보다 const\_iterator를 선호하라.***

***최대한 일반적인 코드에서는 begin, end, rbegin등의 비멤버 버전들을 해당 멤버 함수들보다 선호하라.***

[항목14] noexcept 사용하기

예외를 방출하지 않을 함수는 noexcept로 선언하라.

# noexcept란?

noexcept는 키워드와 연산자로 사용될 수 있다.

## 키워드로써의 noexcept

해당 함수가 예외를 던질 수 있는지 아닌지를 지정한다.

void attack() noexcept

{

std::cout << "예외가 발생할 리 없어.\n";

}

괄호 안의 표현식이 없거나(default) 참(true)이면, 해당 함수는 어떠한 예외도 방출하지 않는다는 보장이다. 반대로 거짓(false)이면, 그렇지 않다는 말이다. 이는 어떠한 조건에서의 noexcept를 처리하고 싶을 때 사용할 수 있다.

template<typename T>

// 이 함수는 T가 Archer일 경우에만 예외 불가를 보장합니다.

void attack(T&& attacker)

noexcept(std::is\_same<typename std::decay<T>::type, Archer>::value)

{

// Archer가 아니라면 예외를 던질 수 있다.

if (!std::is\_same<typename std::decay<T>::type, Archer>::value)

throw std::exception();

std::cout << "power: " << attacker.info() << '\n';

std::cout << "예외가 발생할 리 없어.\n";

}

int main()

{

Archer a;

try

{

attack(a);

}

catch (std::exception& e)

{

std::cerr << e.what() << '\n';

}

return 0;

}

### 여기서 예외가 발생하면 어떻게 될까?

abort()를 호출하여 프로그램이 죽여버린다. 디버깅도 특별한 기호가 있어야 하는 것으로 보인다. 한마디로 골치덩어리다.

## 연산자로써의 noexcept

괄호 안의 함수의 noexcept 여부를 반환한다.

int main()

{

Archer a;

try

{

if (noexcept(attack(a))) attack(a);

if (noexcept(attack(Monster())))

attack(Monster()); // 실행되지 않음

}

catch (std::exception& e)

{

std::cerr << e.what() << '\n';

}

return 0;

}

이는 참, 거짓 여부만 반환하고 함수를 실행하지는 않는다.

# 예외의 기본

예외란, 기대되지 않은 행위다. 곧, 예외가 발생한다는 것은 정상적이지 않은 행동을 보임을 의미한다.

## 예외 던지기(try~catch)

예외가 발생하면 어떻게 될까? 적어도 더 이상 그 함수 내에서는 진행할 수 없다고 판단한다. 예외를 방출한다는 것은 이렇게 발생한 예외를 어떠한 객체 형태로 던지는 것이다. 이것을 함수 내에서 받아서 처리할 수 있으며(try~catch), 아예 함수 밖으로 던질 수도 있다(throw).

만약에 예외를 던졌는데 받을 장치가 없다면 어떻게 될까? 이 때, 스택 풀기가 작동한다.

### 스택 풀기

예외가 발생되는 범위 안에서 예외처리가 되지 못할 때, 함수 호출 스택을 풀면서 함수가 호출되었던 부분으로 돌아가 예외처리를 시도하는 것

int calc(int a, int b)

{

CalcMaster cm;

PencilPrince pp;

int ret = a / b;

//예외가 발생했다!

//그럼 cm과 pp의 소멸자를 볼 수 있다.

varification(a, b, ret);

return ret;

}

int main()

{

try

{

calc();

}

catch (std::exception& e)

{

// ...

}

// ...

}

위의 함수에서는 에서 예외가 발생한다. 그런데 함수 자체에는 예외를 처리할 수단이 없으므로 함수를 풀게 된다. 그러면서 그 안에 들어있는 모든 객체도 풀어야(소멸자를 호출) 한다. 결국, calc를 호출한 main함수로 거슬러 올라간다.   
만약에 스택을 풀던 도중에(소멸자 호출 중에) 또 예외가 발생하면 std::terminate를 호출하여 프로그램을 종료시킨다.

## 프로그램 죽이기

예외가 발생하면 프로그램을 끝내버릴 수도 있다. 발생할 여지는 있지만 절대로 생겨서는 안 되는 예외인 경우에 적합할 것 같다.

## 예외 불가 보장

이것이 가장 바람직하지만 쉽지만은 않다. 이에 대안으로는 변경이 되기 전의 상태로 되돌리거나(강력한 보장), 일관성만을 유지하는 방법이 있다(기본적인 보장).

# noexcept를 사용해야 하는 이유

일단 예전에는 어떻게 예외를 명시했는지를 살펴본다.

C++98에서 사용한 예외 명세란, 해당 함수에서 어떤 예외를 내보낼 수 있는지를 명시하는 표현식이다. 다음과 같이 표현한다.

class MyException : public std::exception {};

void magic() throw(MyException)

{

// 이 함수는 MyException예외를 발생시킬 수 있다.

}

그런데 사용하다 보니 득보다 실이 크다. 별 의미가 없기 때문이다. 결론은 어떤 예외가 발생할지는 중요하지 않으며, 중요한 것은 예외의 존재 여부라는 판단을 내리게 된다. 그래서 throw의 예외 명세를 비권장하고 noexcept키워드가 등장한 것이다.

### Question. 도대체 이걸 왜 써야 하는 지 모르겠다.

Answer. 개인적으로, 나는 noexcept에 관심이 있다. 왜냐하면 어떤 종류의 최적화를 안전하게 적용하기 위해서 컴파일러에게 자유가 증가했기 때문이다.  
가장 큰 최적화 이득은 noexcept를 검사하고 과부하가 걸릴 가능성 때문에 컴파일러가 아닌 사용자 최적화에 의한 것 같습니다. 대부분의 컴파일러는 no-penalty-if-you-don-do-throw 예외 처리 방법을 따르므로 코드의 기계어 수준에서 많은 부분이 변경 될 것이라고 생각합니다.   
특히 생성자, 할당, 소멸자가 noexcept일 때, noexcept를 사용하면 std 컨테이너와 같은 템플릿 코드에서 noexcept검사가 '공통'이므로 최상의 개선이 이루어질 수 있습니다. 예를 들어, std :: vector는 noexcept라고 표시되어 있지 않으면 클래스 이동을 사용하지 않습니다 (그렇지 않으면 컴파일러가 달리 추론 할 수 있음).

## [이유1] 인터페이스 설계상의 문제

확실하게 예외가 발생하지 않음을 명시하면 안정성이나 효율성에 영향을 준다.

## [이유2] 컴파일러의 최적화

일단 **C++98**과 **C++11**버전이 다른 동작을 한다는 것을 이해하자. 스택 풀기는 위에서 살핀 것처럼, 해당 함수를 호출한 지점까지 메모리가 풀린다. 그 예외를 처리해줄 수단을 찾을 때까지 반복된다. 이것은 아무런 명세가 없을 시, 작동하는 방식이다.

**C++98**의 throw()은 예외 명세에 위반되면, 호출 스택이 f를 호출한 지점에 도달할 때까지 풀린다. 그리고 프로그램 실행이 종료된다.

그런데 **C++11**의 noexcept는 실행시점 행동이 약간 다르다. 스택이 풀릴 수도 있고 아닐 수도 있다. (이게 무슨 소리인가?) noexcept에서는 결코 catch이후의 표현들이 실행되지 않는다. 만약 예외가 발생한다 해도, 프로그램은 죽기 때문에 결코 예외가 방출될 일은 없다.

이게 뭐 어쨌다는 것일까? noexcept의 경우, 컴파일러의 ***최적화기(optimizer)***는 실행시점 스택을 풀기 가능 상태로 유지할 필요가 없다. 반면, throw()는 그러한 최적화 유연성이 없다. 아무것도 명시되지 않는다면 더더욱 유연성이 없다.

결국, noexcept의 명세만으로 는 컴파일러 최적화에 도움을 주는 것이다.

### 컴파일러

컴파일러는 보통 **프론트 엔드(front end), 최적화기(optimizer), 백 엔드(back end)** 3가지 요소로 구성된다.

**프론트 엔드(front end)**에서는 어휘 분석과 스캐닝 단계 다음에 코드를 중간 표현으로 변환한다. 프로그램 자체의 의미를 변경하지는 않는다.

**최적화기(optimizer)**에서는 코드의 효율성을 높이기 위해 다양한 종류의 기술을 이용한다. 2가지 기술로 볼 수 있으며, 의미는 그대로지만 구조를 바꾸어 효율적으로 바꾸는 **코드 구조 최적화**와 필요 없는 할당 등을 없애는 **잉여 제거 최적화**로 구분할 수 있다.

**백 엔드(back end)**에서는 중간 표현에서 특정 대상 코드를 생성해낸다.

## [이유3] 이동 연산 시, 안정성

std::vector를 살펴보자. 동적 배열은 크기가 늘어날 시, 2배의 크기를 새로 할당하여 기존의 자료를 새로운 자료에 집어넣는다. 이전에는 이를 복사하였다. 복사하면 도중에 실패한다 해도, 원래대로 되돌릴 수 있는 **강력한 예외 보장**을 가지고 있다.

하지만 이동 연산에서 예외가 발생하면 복구할 방법이 없다. 왜냐하면 위와 마찬가지 과정에서 배열 안의 각 객체를 새로운 공간에 복사가 아닌 이동을 시키는데, 포인터를 옮겼는데 예외가 발생하면 그 데이터들은 사라지기 때문이다. **그러므로 이동 연산은 반드시 예외에 대해 안전해야 한다.** stl의 여러 함수들은 예외에 대해 안전할 경우에만 이동할 수 있도록 되어있다.

struct Archer

{

Archer(int p) {}

Archer(Archer&&) //noexcept 이게 있어야 이동한다.

{}

};

int main()

{

std::vector<Archer> group;

group.emplace\_back(1);

// 이때, 벡터의 확장을 위해 reserve가 발생.

// 이동 생성자가 noexcept를 보장하지 않으면, copy construction이 발생한다

// noexcept가 아니라면, 이동이 아닌 복사가 발생하여 성능 이득을 얻지 못한다.

group.emplace\_back(2);

return 0;

}

그럼 이동 연산에 대하여 noexcept에 따라서 이동할 것인지 복사할 것인지를 판단해주는 형질 특질 함수가 있다면 좋지 않을까?

### std::move\_if\_noexcept

template <typename T>

typename std::conditional<

!std::is\_nothrow\_move\_constructible<T>::value

&& std::is\_copy\_constructible<T>::value,

const T&,

T&&

>::type move\_if\_noexcept(T& x);

이동할 수 있다면 이동하며(=noexcept로 선언되어 있다면), 그럴 수 없다면 복사한다.

Archer a1(1);

Archer a2 = std::move\_if\_noexcept(a1);

//Archer이 이동할 수 있다면 이동하고, 그럴 수 없다면 복사한다.

위와 같이 사용하면 예외에 대해서 안전하다.

## [이유4] 조건부 noexcept

std::pair에서의 swap의 예외 명세이다.

void swap(pair& pr) noexcept(

noexcept(swap(first, pr.first)) &&

noexcept(swap(second, pr.second)))

{

swap(first, pr.first);

swap(second, pr.second);

}

noexcept괄호 안에 내용이 있는 경우를 말하는 것으로, 앞에서 의미를 언급했다. 이는 유연성을 가진다. 어떠한 함수를 수정한다고 하자. 그랬더니 예외가 생길 수도 있다. 그럼 noexcept를 없애야 하는데 이로 인해서 그를 참조한 다른 함수가 영향을 받을 수 있다.

**그러므로 조건부 noexcept는 이러한 영향에서 유연성을 가진다.** 어떠한 함수의 noexcept여부에 따라서 해당 함수의 noexcept여부를 결정할 수 있는 것이다. 이러면 앞서 호출한 함수가 수정되어도 뒤에 함수에 영향이 최소화된다.

# noexcept사용 시, 주의할 점

### 예외에 중립적인 함수는 noexcept일 수 없다.

예외에 중립적이란, 스스로는 예외를 던지지 않지만, 예외를 던질 수 있는 함수를 호출할 수는 있는 것을 의미한다. 그리고 대부분의 함수들은 여러 함수들을 내포하므로 이러한 중립성을 띄고 있다.

### 가능하면 noexcept를 선언하라.

그러한 구현이 자연스러운 함수들에 한해서 이에 해당한다.

### noexcept가 자연스럽지 않은 함수를 억지로 뜯어 고치지 말아라.

구현도 복잡해질뿐더러, 호출 지점의 코드도 난해해질 수 있다.

### 암묵적 noexcept

소멸자에는 암묵적으로 noexcept가 적용되므로 굳이 noexcept를 명시할 필요가 없다. 메모리 해제 함수들도 같은 영향을 가진다. 예외에 치명적인 함수들이다.

### noexcept는 보통 넓은 계약에서만 사용된다.

넓은 계약이란, 어떠한 명세에 대해 제약이 없는 함수이다. 그렇지 않다면 모두 좁은 계약이다. 예를 들어, 어떤 함수가 문자열을 인자로 받는데 그 길이가 32이하일 경우에만 noexcept를 만족한다면, 이는 좁은 계약인 셈이다. 참고로 이러한 경우에는 조건부 noexcept를 사용할 수 없다. 이는 컴파일 타임에나 적용 가능하다.

그럼 어떻게 처리해야 바람직할까? 넓은 계약에 대해서만 noexcept를 사용하는 것이 일반적이다. 조건을 검사하는 방식은 런타임에 이루어지므로 noexcept에서는 예외를 던질 수 없기에 검사하는 방식은 어려울 수 있다.

### 꼭 호출하는 모든 함수가 noexcept가 아니더라도 noexcept일 수 있다.

일반적으로는 어떤 함수 내부에서 호출되는 함수 또한 예외가 없어야 본 함수에서도 예외가 없지만 이는 서류상의 문제일 수도 있다는 것이다. C언어로 작성된 함수라든지, 다른 이유일 수도 있다. 그러므로 C++은 이런 코드를 허용하는 것이다.

void thunderStorm() // noexcept가 아니다.

void clear() noexcept

{

thunderStorm(); // 예외 없이 모두 박살

// thunderStorm은 noexcept가 아니지만 적법하다.

}

***noexcept는 함수의 인터페이스의 일부다. 이는 호출자가 noexcept 여부에 의존할 수 있음을 뜻한다.***

***noexcept 함수는 비noexcept 함수보다 최적화의 여지가 크다.***

***noexcept는 이동 연산들과 swap, 메모리 해제 함수들, 그리고 소멸자들에 특히나 유용하다.***

***대부분의 함수는 noexcept가 아니라 예외에 중립적이다.***

[항목15] constexpr 사용하기

가능하면 항상 constexpr을 사용하라.

# const

변경되지 않을 값인 상수를 의미한다. 읽기만 가능하게 되며 인터페이스로써의 의미 외에도 컴파일러 최적화로 성능향상을 기대할 수 있다.

### const의 사용

<http://codemuri.tistory.com/entry/%EA%B3%A8%EC%B9%98%EC%95%84%ED%94%88-C-const-%EC%9A%A9%EB%B2%95-%EC%A0%95%EB%A6%AC>

int temp1 = 100;

int temp2 = 200;

//////////////////////////////////////////////////////////////////////////

// 1. 변수

//////////////////////////////////////////////////////////////////////////

const int i = 100;

// i = 200;

// &i는l-value가 아니므로 대입자체가 무효.

//////////////////////////////////////////////////////////////////////////

// 2. 포인터형

//////////////////////////////////////////////////////////////////////////

// \* 를기준으로왼쪽이냐오른쪽이냐에따라서다음과같다.

// \* 의왼쪽

const int \* piConst = &temp1;

// int const \* piConst = &temp1; <- \*의왼쪽으로위문장과동일.

// \*piConst 가상수이다.

// \*piConst = 200; // error

// 하지만ipConst 는상수가아니므로주소는변경이가능하다.

piConst = &temp2;

// \* 의오른쪽

int \* const piConst2 = &temp1;

// piConst2 가상수이다.

// piConst2 = &temp2; // error

// 하지만\*piConst2 는상수가아니므로값은변경이가능하다.

\*piConst2 = 200;

// 복합적

int const \* const piConst3 = &temp1;

//const int \* const const 와동일

// 완전상수다. ^^

// piConst3 = &temp2; // error

// \*piConst3 = 200; // error

//////////////////////////////////////////////////////////////////////////

// 3. 참조형

//////////////////////////////////////////////////////////////////////////

int const & refVal = temp1;

// refVal = 200; error;

// 역시&refVal 는l-value 가아니므로대입자체가무효

int & const refVal2 = temp1;

// --> 이렇게사용하는것은VC 에서테스트해본결과아무효과가없어보인다. why?

//////////////////////////////////////////////////////////////////////////

// 4. 함수

//////////////////////////////////////////////////////////////////////////

// class 의멤버함수에만const 선언가능.

class A {

private:

int m\_i;

mutable int m\_j; // const 로선언된멤버함수에서도변경이가능함. C++ 표준임.

public:

A() {

m\_i = 3;

m\_j = 5;

}

int GetValue() const {

// m\_i = 2; // error. const 로선언된멤버함수는멤버변수변경불가.

m\_j = 4;

return m\_i;

}

} a;

int k = a.GetValue();

//////////////////////////////////////////////////////////////////////////

// 5. 클래스

//////////////////////////////////////////////////////////////////////////

const class B {

private:

int m\_i;

public:

int GetValue()

{

return m\_i;

}

};

B b;

const B & constB = b;

// constB.GetValue(); // error. GetValue 가const 가아니면사용할수없다.

const A & constA = a;

constA.GetValue(); // GetValue() 가const 이므로사용가능하다.

//////////////////////////////////////////////////////////////////////////

// 6. const만차이가나는멤버함수들이오버로딩될수있다는사실

//////////////////////////////////////////////////////////////////////////

class C {

int Add(int i, int j) {

return (i + j);

}

int Add(int i, int j) const {

return (i + j);

}

/\* 아래는성립하지않음.

int Add(const int i, const int j) {

return (i+j);

}

\*/

};

const에 대한 자세한 내용은 ec++[항목3] 참고

### const 성능향상

상수객체들을 const로서 선언하는 다른 이익들에 덧붙여 최적화하는 컴파일러 역시 이 선언의 장점을 얻어서 그러한 객체를 보통의 메모리 대신에 머신 레지스터로 저장한다. 동일한 최적화가 const로 선언된 함수 파라미터들에 적용될 수 있음에 주목하라.

# constexpr

const의 강화된 버전처럼 작용하며, 일반화된 상수 표현식(Generalized constant expression)이라고 할 수 있다. 이게 무슨 말인지 아래에서 살펴본다.

## 정의

constexpr지정자는 컴파일 타임에 함수 또는 변수의 값을 평가할 수 있음을 선언합니다. 즉, constexpr로 선언된 변수나 객체는 컴파일 타임에 값을 알 수 있는 것이다.

템플릿 메타 프로그래밍을 간단하게 구현할 수 있겠군.

## 조건

아무데나 constexpr을 사용할 수는 없다. 다음과 같은 조건 하에 constexpr을 선언할 수 있다. 또 유념해야 할 점은, 변수와 함수에서 쓰일 때 상당히 다른 의미로 작용한다는 점이다.

### 변수에서의 사용

변수 혹은 객체에서 constexpr이 선언되면 단순히 const성을 지닐 뿐만 아니라, 컴파일 타임에 알려진다는 보장이 있다. 배열의 인자, 열거자의 값 등으로 쓰일 수 있는 것이다.

이렇게 쓰이기 위한 조건은 다음과 같다.

*1. 반드시 리터럴 타입(코드에 직접 알려진 고정된 값)이어야 한다.  
2. 반드시 초기화되어야 한다.  
3. 모든 암시적 변환, 생성자 호출 등을 포함하여 초기화의 전체 표현은 상수 표현이어야 합니다.*

class Archer

{

public:

constexpr Archer(int p) : power(p) {}

private:

int power;

};

int main()

{

constexpr int power = 9999; // 반드시 초기화

//constexpr Archer\* a = new Archer(); // 상수식 표현에 this\*사용 못함

constexpr Archer a = Archer(power); // 생성자에 constexpr이 없으면 안 됨

// 위의 멤버 변수 int power에는 constexpr을 사용할 수 없다.

return 0;

}

### 함수에서의 사용

함수에서의 constexpr선언의 의미는 약간 다르다. 일단 inline으로써의 선언이 적용된다. 그리고 중요한 점은 결과 값이다. 결과가 const나 컴파일 타임에 알려지지 않을 수 있다는 점인데, 이는 유연하게 작동하도록 도와준다.

*1. 가상함수일 수 없다.  
2. 모든 파라미터와 반환 값이 리터럴 타입이어야 한다.  
3. 함수의 호출이 핵심 상수 표현식의 평가 된 서브 표현식이 될 수 있는 하나 이상의 인수 값 집합이 존재해야 한다. (constexpr 함수 템플릿의 경우 최소 하나의 특수화가 이 요구 사항을 충족해야 생성자에 대해 상수 초기화 기가 사용됩니다).*

class Archer

{

public:

constexpr Archer(int p) : power(p) {}

constexpr int myPower() noexcept { return power; }

private:

int power;

};

constexpr int amazingPower(const Archer& target) noexcept

{

// C++14가 아니라서 여러 문장을 사용할 수 없다.

return target.myPower();

}

int main()

{

constexpr int power = 9999;

constexpr Archer a = Archer(power);

int arr[amazingPower(a)];

return 0;

}

인자와 호출 함수 모두 constexpr로 선언되어 있으므로 컴파일 타임에 값을 사용할 수 있다.

## 동기

개념을 혼동시킬 수 있는 이 constexpr을 사용해야 하는 이유는 무엇일까? 그것도 반드시 사용해야 하는 이유는 무엇일까?

### 넓은 의미에서 사용할 수 있다.

어떠한 변수, 객체를 constexpr로 선언했다고 하자. 이가 컴파일 타임에 알려질 수 있는 조건이라면 그렇게 될 것이다. 그렇지 않다면 어떻게 될까? 오류가 아닌 일반적으로 작동한다. 그러므로 이래도 가능하며 저래도 가능한 유연성을 지닌다. 물론 선언에는 적용되지 않는다. 이러한 점을 위에서 말한 일반화된 상수 표현식이라고 할 수 있다.

### 성능 향상

런타임 작업을 컴파일 타임으로 옮기므로 당연히 실행 속도가 상승한다. 이것은 확실하게 const와는 차별되는 부분이다.

### 인터페이스의 일부

constexpr을 지정한다는 것은 “이 함수를 C++이 상수 표현식을 요구하는 문맥에서 사용할 수 있다.”는 것이다. 배열의 크기 지정 등이 그러하다.

# constexpr사용하기

위에서 조건을 언급하면서 간단한 사용법을 배웠다. 조금 더 세분화하여 들어가보자.

## 변수에서의 사용

상수만 대입할 수 있습니다. constexpr타입의 다른 변수는 안 됩니다.

int a = 10;

**const** int a = 11; // OK

**const** int b = a; // const 변수로부터의 대입 OK

**constexpr** int c = 12; // OK

**constexpr** int d; // 초기화가 안 되어 C2737 컴파일 에러

**constexpr** int e = c; // 다른 변수로부터 대입 C2127 에러

## 정수 상수 표현식에 사용

컴파일 타임에 일어나므로 가능하다.

**constexpr** int size = 1000;

std::array<int, size> A;

## 함수에서의 사용

함수 반환 값을 constexpr로 선언할 수 있다.

### 함수에서 인자가 모두 constexpr일 때, 그로 인한 연산의 결과는 constexpr이다.

constexpr끼리 계산된 모든 값은 컴파일 타임에 계산된다.

//n과 e가 constexpr이라면, 결과도 constexpr이다.

//모두 컴파일 타임에 계산된다.

**constexpr** int pow(int n, int e) **noexcept**

{

**if** (e == 0) **return** 1;

**if** (e == 1) **return** n;

**if** (e & 1) **return** n\*pow(n, e - 1);

**auto** ret = pow(n, e / 2);

**return** ret\*ret;

}

int main(){

**constexpr** **auto** numExp = 100;

std::array<int, pow(3, numExp)> ret;

}

### 컴파일 타임에 계산될 수 없다면, constexpr을 반환하지 않는다.

이는 일반적인 특징을 갖게 한다. n, e가 constexpr가 아니라면 실행시점에서 계산된다. 즉, 계산 과정이 constexpr이 아니라도 이를 허용하며, 반환 타입은 constexpr이 아닌 것이다.

int main(){

//…

std::cin >> k >> m;

**auto** n = k;

**auto** e = m;

//n, e가 constexpr이 아니지만 동작한다.

//인자로 constexpr이든 아니든 그 버전에 맞는 값을 반환

**auto** ret = pow(n, e);

//…

}

### constexpr의 인자나 반환 형식은 반드시 리터럴 형식이어야 합니다.

반환 값은 반드시 존재해야 하며(void는 안 된다.), 리터럴 형식이어야 한다.

//반환 타입, 매개변수 타입 모두 리터럴 타입이어야 합니다.

**constexpr** int pow(int base, int exp) **noexcept**{

**auto** ret = 1;

**for** (int i = 0; i < exp; i++) ret \*= base;

**return** ret;

}

## 생성자에서의 사용

생성자를 constexpr로 선언되면 constexpr 형식의 객체를 만들 수 있다. 이가 가능 하려면, 받는 인자들이 컴파일 타임에 계산 가능해야 한다.

**class** **Archer**{

**public**:

**constexpr** Archer(int l, long long p) **noexcept**

: level(l), power(p) {}

**constexpr** int getLevel() **const** { **return** level; }

**constexpr** long long attackPower() **const** **noexcept**

{ **return** level \* power; }

**constexpr** long long defencePower() **const** **noexcept**

{ **return** (level + power) \* level; }

void increasePower(long long np) **noexcept**{ power += np; }

**private**:

int level;

long long power;

};

**constexpr** Archer getStrongArcher(**const** Archer& a1, **const** Archer& a2) **noexcept**{

*//{}로 객체를 초기화 및 생성*

**return**{ std::max(a1.getLevel(), a2.getLevel()),

(a1.attackPower() + a2.attackPower())

+ (a1.defencePower() + a2.defencePower()) };

}

int main(){

*//constexpr생성자로 constexpr객체를 만듬*

**constexpr** Archer a(10, 100000), b(7, 100);

*//constexpr함수들로 constexpr객체를 만들어낸다.*

*//모두 컴파일 타임에 발생할 수 있다.*

**constexpr** **auto** ultraArcher = getStrongArcher(a, b);

}

위의 모든 사항들은 C++11에서는 제약이 따른다. C++14를 기준으로 설명되었다.

# 템플릿 메타 프로그래밍 vs constexpr

참고: <http://egloos.zum.com/sweeper/v/3147813>

이 둘은 컴파일 타임에 시행되므로 같은 일을 할 수 있다.

template<int n>

struct Factorial

{

enum { value = n \* Factorial<n - 1>::value };

};

template<>

struct Factorial<1>

{

enum { value = 1 };

};

constexpr int factorial(int n)

{

return n > 1 ? n\*factorial(n - 1) : 1;

}

int main()

{

std::cout << Factorial<30>::value << '\n';

std::cout << factorial(30) << '\n';

int arr[Factorial<10>::value];

int brr[factorial(10)];

return 0;

}

***constexpr객체는 const이며, 컴파일 도중에 알려지는 값들로 초기화된다.***

***constexpr함수는 그 값이 컴파일 도중에 알려지는 인수들로 호출하는 경우에는 컴파일 시점 결과를 산출한다.***

***constexpr객체나 함수는 비constexpr객체나 함수보다 광범위한 문맥에서 사용할 수 있다.***

***Constexpr은 객체나 함수의 인터페이스의 일부다.***

[항목16] 스레드에 안전한 함수 작성

const멤버 함수를 스레드에 안전하게 작성하라.

# 스레드에 안전한 함수

여러 개의 스레드가 실행되는 환경이라고 하자. 그럼 하나의 자료에 여러 스레드가 동시에 접근할 수 있다. 이를 자료 경쟁이라고 하며, 이로 인해 결과에 영향을 주는 상태를 경쟁 조건이라고 한다.

위와 같은 경쟁조건이 있는 영역은 스레드로부터 안전하지 않다고 말한다. 왜냐하면 언제든지 자료가 오염될 수 있기 때문이다. 반대로 스레드로부터 안전 하려면, 어느 하나의 자료를 여러 스레드가 동시에 접근할 수 없게 만들어야 한다. 이렇게 한 구역에 하나의 스레드만 접근하도록 하는 과정을 동기화한다고 한다. 이를 가능하게 해주는 것이 뮤텍스, 세마포어 등이 있다.

# 읽기 전용 함수 만들기

읽기 전용 함수란, 쓰지 않으므로 변경될 일이 없는 함수로 해석된다. 그러면 떠오르는 키워드가 const이다. 다음과 같이 만들 수 있다.

struct Archer

{

void intro() const { std::cout << "I'm archer.\n"; }

};

class BattleField

{

public:

void entry(const Archer& attendent) // 변경되므로 읽기 전용 아님

{

field.push\_back(attendent);

}

void showField() const // 읽기 전용이므로 const

{

// 내부의 모든 내용도 const이어야 한다.

for (const Archer& a : field) a.intro();

}

private:

std::vector<Archer> field{ 0 };

};

위의 코드에서는 entry는 읽기 전용이 아니지만, showField는 읽기 전용이 된다.

정리하자면, const로 선언된 함수는 읽기 전용 함수로 해석해도 좋다.

## 멀티 스레드에서의 읽기 전용 함수

여러 스레드가 동시에 접근하는 읽기 전용 함수가 있다고 하자. 적어도 개념상으로는 동기화가 이루어져야 한다. 읽기만 하는데 여러 스레드가 접근한다고 해서 내용이 달라진다면 말이 안 된다.

int main()

{

BattleField bf;

std::thread t1([&] { bf.showField(); });

bf.entry(Archer());

std::thread t2([&] { bf.showField(); });

t1.join();

t2.join();

return 0;

}

적어도 위의 코드에서는 문제가 발생할 여지가 있다. showField의 반복문을 진행하는 도중에 벡터 원소가 변경이라도 되면 어떻게 되는 것인가? bf가 갱신되는 시점에 따라서 t2가 보여주는 결과는 다를 것이다. 함수 내부에서는 완벽하게 변경되지 않는 읽기 전용 함수라 할지라도, 멤버변수나 전역변수를 사용하고 있어 외부 함수에 의해 변경될 여지가 있다면 이는 안전하지 않은 것이다.

그럼 이를 어떻게 해결하면 좋을까? 우리의 목표는 완전한 읽기 전용 함수를 만드는 것이다.

### std::mutex를 이용한 동기화

어떤 구간 자체를 동기화시킨다. 이 구간에는 하나의 스레드만이 접근할 수 있다. 다른 스레드가 동시에 접근을 시도하면, 뮤텍스를 잠근 위치에서 해당 스레드를 기다리게 한다.

class BattleField

{

public:

void entry(const Archer& attendent) // 변경되므로 읽기 전용 아님

{

std::lock\_guard<std::mutex> guard(m);

if (flag)

{

flag = false;

field.push\_back(attendent);

flag = true;

}

}

void showField() const // 읽기 전용이므로 const

{

std::lock\_guard<std::mutex> guard(m);

// 내부의 모든 내용도 const이어야 한다.

if (flag)

{

flag = false;

for (const Archer& a : field) a.intro();

flag = true;

}

}

private:

std::vector<Archer> field{ 0 };

mutable std::mutex m;

mutable bool flag{ true };

};

구간 자체를 동기화시킨다는 의미에서 어느 정도 안전한 동기화를 제공한다. 하지만 이는 사실 부담스러운 작업이다. 병목현상으로 인한 성능 저하도 염두 해야 한다.

### std::atomic를 이용한 동기화

atomic은 어떤 변수를 동기화시킨다. 그러므로 2개의 동기화된 변수로 작업을 수행한다.

이는 사실 안전하지 못하다. 변수2개는 결국 떨어져 있으므로 개개인이 동기화될지라도 서로의 관계는 동기화되지 못한다. flag변수를 바꾸고 동시에 멤버변수를 변경하는 것은 자연스러운 일이므로 전혀 동기화되지 못하는 것이다.

### 결론

**const를 선언된 함수는 읽기 전용 함수라는 의미이다.** 단일 스레드에 대해서는 내부 변경에 대한 상수화만으로 해결되지만 멀티 스레드에 대해서는 동기화에 대한 고려도 해야 한다. 이는 변수를 각각 두어서는 안 되며, 구간을 동기화시키는 mutex로 진행해야 한다.

***동시적 문맥에서 쓰이지 않을 것이 확실한 경우가 아니라면, const 멤버 함수는 스레드에 안전하게 작성하라.***

***std::atomic변수는 뮤텍스에 비해 성능상의 이점이 있지만, 하나의 변수 또는 메모리 장소를 다룰 때에만 적합하다.***

[항목17] 특수 멤버 함수

특수 멤버 함수들의 자동 작성 조건을 숙지하라.

# 특수 멤버 함수?

특수 멤버 함수: C++이 스스로 만드는 함수

**class** **Archer**{

**public**:

Archer();

~Archer();

Archer(**const** Archer&);

Archer& **operator**=(**const** Archer&);

*//C++11에서 새로 추가된 부분*

Archer(Archer&&);

Archer& **operator**=(**const** Archer&&);

};

물론 무작정 스스로 만들지는 않는다. 다음에서 규칙을 살펴본다.

# 특수 멤버 함수 생성 규칙

기본적으로 아무것도 선언하지 않았다면 복사 생성, 대입 연산자, 이동 생성, 대입 연산자, 소멸자를 생성한다.

## 이동 연산

이동 생성자, 이동 배정 연산자를 이용하여 std::move로 객체를 이동시키는 연산을 말한다. 이는 요청에 가깝다. 이동이 활성화되지 않은 형식에 대해서는 작동하지 않는다. 결국 오버로딩의 과정을 거치는 셈이다. 그런 경우에는 복사 연산이 수행된다.

## 생성자

생성자 중 어느 버전이라도 직접 생성했다면 기본 생성자는 생성되지 않는다. 즉, 사용자 선언 생성자가 없어야만 자동 생성된다.

## 복사 생성자, 복사 대입 연산자

서로 독립적이다. **하나만 선언해도 다른 하나는 컴파일러가 작성한다.**

**class** **Archer**{

**public**:

Archer() : level(1) {

}

Archer(**const** Archer& other){

level = other.level;

}

*/\*Archer& operator=(const Archer& other){*

*level = other.level;*

*return \*this;*

*}\*/*

**private**:

int level;

};

Archer(**const** Archer& other) 혹은 Archer& operator=(const Archer& other) 둘 중 하나만 선언되도 잘 작동한다.

### 3의 법칙(Rules of Three)

복사 생성자, 복사 배정 연산자, 소멸자 중 하나라도 직접 선언했다면 나머지 둘도 선언해야 한다.

왜 그럴까? 일단 언제 직접 선언해야 할지를 생각해보자. 기본 복사 연산은 객체의 모든 자료를 복사한다. 어떤 자료는 복사가 안 될 수도 있으며 다른 방식의 자원 관리를 필요로 할 수도 있다. 그런데 3개 중 하나라도 변경해야 한다면 특별 관리가 필요한 자원이 있다는 것이고 그러면 자연스레 나머지 2함수에도 영향이 갈 수밖에 없다. 자료를 도중에 버릴 수 있는 것이 아니기 때문이다.

class Archer

{

public:

Archer(const Archer& other)

{

counter++;

\*impl = \*other.impl;

// mutex는 복사될 수 없다.

}

Archer& operator=(const Archer& other)

{

counter++;

\*impl = \*other.impl;

// mutex는 복사될 수 없다.

return \*this;

}

private:

struct Data

{

};

Data\* impl;

int counter{ 0 };

std::mutex m;

};

## 이동 생성자, 이동 대입 연산자

서로 독립적이지 않다.

*“기본 이동 생성자가 적합하지 않으면, 이동 대입 연산자도 적합하지 않을 것이다.”*

그러므로 **둘 중 하나만 선언했다면, 나머지 하나는 작성되지 않는다. 없는 함수다.**

**class** **Archer**{

**public**:

Archer() : level(1) {

}

*//~Archer();*

Archer(**const** Archer& other){

level = other.level;

}

Archer& **operator**=(**const** Archer& other){

level = other.level;

**return** \***this**;

}

*//C++11에서 새로 추가된 부분*

Archer(Archer&& other){

level = other.level;

}

*/\*Archer& operator=(const Archer&& other){*

*level = other.level;*

*return \*this;*

*}\*/*

**private**:

int level;

};

int main(){

Archer archer1, archer2;

Archer&& archer3 = Archer();

*//컴파일은 가능하지만, 이동 대입이 아닌 복사 대입 연산자를 수행합니다.*

archer1 = Archer();

}

## 복사 연산과 이동 연산

복사 연산이 하나라도 작성되었다면, 이동 연산은 삭제된다.

반대도 마찬가지로 이동 연산이 하나라도 작성되었다면, 복사 연산은 삭제된다.

## 소멸자

위에서 언급한대로, 소멸자가 명시적으로 선언되면 다른 특수 멤버 함수의 자동 작성은 비권장이다.

이미 가상으로 선언된 클래스를 파생하는 클래스이면 가상 소멸자가 자동 생성된다.  
그 외에는 직접 virtual을 선언해야 한다.

# 기본 특수 멤버 함수 작성

어떤 특수 멤버 함수를 사용자가 직접 작성하여 나머지 함수들이 자동 생성되지 않는다고 하자. 하지만 자동 생성된 함수를 사용할 수 있는 쉬운 방법은 없을까? 이에 기본 행동을 사용하겠다는 의지를 = **default**로 보여준다.

**class** **Archer**{

**public**:

Archer() = **default**;

~Archer() = **default**;

Archer(**const** Archer&) = **default**;

Archer& **operator**=(**const** Archer&) = **default**;

*//C++11에서 새로 추가된 부분*

Archer(Archer&&) = **default**;

Archer& **operator**=(Archer&&) = **default**;

**private**:

int level;

};

## 자동 생성에 의지하여 발생하는 오류에 주의하자.

소멸자를 추가하여 이동 연산이 삭제되는 현상이 발생할 수 있다.

## 멤버 함수 템플릿이 존재하면 자동 작성 비활성화가 되지 않는다.

템플릿으로 선언하면 무슨 상황이든 자동 생성된다.

**class** **Archer**{

**public**:

Archer() {};

~Archer() = **default**;

*//템플릿이라면 선언 안해도 링크가 됩니다.*

**template**<**typename** T>

Archer(**const** T&);

**template**<**typename** T>

Archer& **operator**=(**const** T&);

**private**:

int level;

};

### [참고] 멤버 함수 포인터

클래스 내의 함수를 콜백하고 싶거나 포인터로 사용하고 싶다면 어떻게 해야 할까?

#### 함수 포인터 사용하기

class GameManager

{

public:

int(GameManager::\*GF)(int, int);

int calcVelocity(int l, int p)

{

return l \* p;

}

};

class Archer

{

public:

int(GameManager::\*GF)(int, int);

void callBack(int(GameManager::\*func)(int, int))

{

//std::cout << func(level, power) << '\n';

}

private:

int level{ 1 };

int power{ 10 };

};

int main()

{

GameManager gm;

//pf gmFunc = &GameManager::calcVelocity; // 이건 안 된다.

int(GameManager::\*gmFunc)(int, int) = &GameManager::calcVelocity;

gm.GF = &GameManager::calcVelocity; // 객체 내부에 멤버 함수 포인터를 담고 있다.

// 포인터로 가져온 함수를 호출하는 법

std::cout << (gm.\*gmFunc)(9999, 9999) << '\n';

Archer a;

a.GF = &GameManager::calcVelocity; // 다른 멤버 함수에 쉽게 접근할 수 있다.

return 0;

}

멤버 함수 포인터의 형식은 int(GameManager::\*gmFunc)(int, int)와 같이 표현한다.

그 멤버 함수 포인터를 사용하려면 반드시 객체가 있어야 하는데, (gm.\*gmFunc)(9999, 9999)와 같이 일반 멤버 함수를 호출하듯 사용하면 된다.

객체 안에 사용하고자 하는 다른 객체의 멤버 함수 포인터를 담고 있으면 다른 객체에서 콜백 함수나 이를 불러오기 용이하다. a.GF = &GameManager::calcVelocity과 같이 담을 수 있다.

#### 콟백 함수를 호출하기 위해 인자로 넘기기

일반적인 방법으로는 전달되지 않는다. 다음과 같은 오류가 생긴다.

// 바인딩된 함수는 호출하는 데에만 사용할 수 있습니다. 고로 아래는 안 됩니다.

a.callBack(gm.GF);

일단 std::function<>으로 호출하는 방법에 대해 살펴보자.

class Archer

{

public:

int(GameManager::\*GF)(int, int);

void callBack(std::function<int(int, int)> func)

{

std::cout << func(level, power) << '\n';

}

private:

int level{ 1 };

int power{ 10 };

};

int globalMaster(int a, int b)

{

return a + b;

}

int main()

{

// 비멤버 함수 포인터를 담는 법

std::function<int(int, int)> gmfo = &globalMaster;

using namespace std::placeholders;

// 멤버 함수는 아래처럼 묶어서 포인터에 담아야 한다.

std::function<int(int, int)> gmfo2 = std::bind(&GameManager::calcVelocity, gm, \_1, \_2);

a.callBack(gmfo);

a.callBack(gmfo2);

return 0;

}

std::function에 본 객체와 함께 인자 자리에 std::bind함수로 묶어서 만들어야 한다.

포인터로는 어떻게 넘길까?

***컴파일러가 스스로 작성할 수 있는 멤버 함수들, 즉 기본 생성자와 소멸자, 복사 연산들, 이동 연산들을 가리켜 특수 멤버 함수라고 부른다.***

***이동 연산들은 이동 연산들이나 복사 연산들, 소멸자가 명시적으로 선언되어 있지 않은 클래스에 대해서만 자동으로 작성된다.***

***복사 생성자는 복사 생성자가 명시적으로 선언되어 있지 않은 클래스에 대해서만 자동으로 작성되며, 만일 이동 연산이 하나라도 선언되어 있으면 삭제된다. 복사 배정 연산자는 복사 배정 연산자가 명시적으로 선언되어 있지 않은 클래스에 대해서만 자동으로 작성되며, 만일 이동 연산이 하나라도 선언되어 있으면 삭제된다. 소멸자가 명시적으로 선언된 클래스에서 복사 연산들이 자동으로 작성되는 기능은 비권장이다.***

***멤버 함수 템플릿 때문에 특수 멤버 함수의 자동 작성이 금지되는 경우는 없다.***

[항목18] std::unique\_ptr사용

소유권 독점 자원의 관리에는 std::unique\_ptr를 사용하라.

# std::unique\_ptr<T>의 특징

C++14에서 제공하는 스마트 포인터 4개중 하나인 std::unique\_ptr을 소개한다.

## 기본적인 특징

한 대상에 대한 유일한 포인터를 갖고 싶다면 주저 없이 std::unique\_ptr를 택하라.

### 생포인터와 크기가 같다

가볍고 효율적이다.

### 독점적 소유권(exclusive ownership)

std::unique\_ptr<T>를 이동하면 소유권이 원본에서 사본으로 넘어가며 원본은 nullptr이 된다. (=이동) 그러므로 복사는 금지되어있다.

## std::unique\_ptr<T>의 응용

아래 코드는 std::unique\_ptr<T>를 이용하여 펙토리 함수를 만든 코드이다. 이를 통하여 더 많은 특징들을 살펴보자.

#include *<iostream>*

#include *<memory>*

#include *<string>*

**using** Money = unsigned long long;

**class** **Unit**{

**public**:

Unit(Money value = 0) : name(std::to\_string(value)) {}

**virtual** ~Unit() {}

std::string getInfo(){ **return** name; }

**private**:

std::string name;

};

**class** **Archer** : **public** Unit {

**public**:

Archer(Money value = 0) : Unit(value) {}

**virtual** ~Archer() {}

};

**class** **BladeMaster** : **public** Archer{

**public**:

BladeMaster(Money value = 0) : Archer(value) {}

};

**class** **DarkKnight** : **public** Archer{

**public**:

DarkKnight(Money value = 0) : Archer(value) {}

};

**class** **Logger**{

**public**:

**static** std::string log;

**static** void makeLog(Unit\* unit){

log += unit->getInfo() + " delete.**\n**";

}

};

std::string Logger::log;

**auto** delUnit1 = [](Unit\* unit){

Logger::makeLog(unit);

**delete** unit;

};

**template**<**typename** T>

**class** **Barrack**{

**public**:

std::unique\_ptr<Unit, **decltype**(delUnit1)> makeUnit(T&& money){

std::unique\_ptr<Unit, **decltype**(delUnit1)>

pUnit(**nullptr**, delUnit1);

**if** (money > 1000){

pUnit.reset(**new** Archer((std::forward<T>(money))));

money -= 1000;

std::cout << "make Archer.**\n**";

}

**if** (money > 10000){

pUnit.reset(**new** BladeMaster((std::forward<T>(money))));

money -= 10000;

std::cout << "make BladeMaster.**\n**";

}

**if** (money > 30000){

pUnit.reset(**new** DarkKnight((std::forward<T>(money))));

money -= 30000;

std::cout << "make DarkKnight.**\n**";

}

**return** pUnit;

}

};

int main(){

Barrack<Money> barrack;

Money ytk\_money = 1000000000000;

barrack.makeUnit(std::forward<Money>(ytk\_money));

std::cout << Logger::log << std::endl;

**return** 0;

}

### 삭제자 지정

스마트포인터는 **2번째 형식 인수로 삭제자의 형식을 지정**할 수 있다. decltype으로 삭제자의 형식이 무엇인지 연역할 수 있다.

std::unique\_ptr<Unit, **decltype**(delUnit1)> pUnit

이 삭제자는 람다식으로 정하면 효율적인데, 함수 포인터나 함수 객체와는 다르게 공간을 차지하지 않기 때문이다.

### 생포인터 배정

생포인터로 배정할 수 없다. 생포인터에서 스마트 포인터로 암시적 변환이 일어날 수 없기 때문이다. 그러므로 **reset()을 호출**하여 배정하여야 한다.

### 완벽한 인수 전달

팩토리 함수 makeUnit()에서 **전달된 인수를 완벽하게 전달하기 위해 std::forward<T>**를 사용할 수 있다. 이는 호출자가 함수에 적용한 모든 정보를 생성자에게 손실 없이 넘겨준다. 여러 개의 인수가 있어도, 우측값이든 좌측값이든 모든 넘길 수 있다.

# std::unique\_ptr<T>는 언제 쓰면 좋을까?

### 계통 구조의 팩토리 함수

위에서 예제 속에 들어있다. 하나의 기본 클래스를 상속하는 여러 개의 파생 클래스들이 있고, 이런 파생 클래스로 객체를 만드는 함수가 팩토리 함수이며, 기본 클래스의 포인터를 반환하여 하나의 함수로 여러 개의 객체를 처리하게 해준다. 어떠한 객체의 파생 포인터를 반환하는 팩토리 함수는 자원 관리 책임을 그대로 사용자에게 전가한다고 볼 수 있다. 하지만 std::unique\_ptr은 이를 자동으로 삭제해준다.

### Pimpl관용구

어떠한 구현 파일을 하나의 포인터에 담아두는 기법. std::unique\_ptr으로 구현하기에 아주 적합하다. – [항목22] 참고

### 배열과 일반 포인터의 명확한 구분

하지만 쓸 일은 없다. 배열을 쓰려면 vector, array등 STL컨테이너가 항상 더 나은 선택이다.

### 목적이 명확하다.

또 다른 스마트 포인터인 std::shared\_ptr은 독점적으로 소유하는지 공유하는지 선언만으로는 알 수 없다. std::unique\_ptr는 그것이 명확하며, 이후에 변환될 수 있는 유연함도 가진다.

#### std::shared\_ptr로의 변환

std::unique\_ptr는 std::shared\_ptr로 손쉽게 변환할 수 있다.

**std::shared\_ptr<Unit>** sp = barrack.makeUnit(std::forward<Money>(ytk\_money));

펙토리 함수에 반환 형식으로 아주 적합한 중요한 이유 중 하나다. 함수에서 반환하는 객체가 공용할 지, 유일하게 사용할 지는 알 수 없다. 하지만 std::unique\_ptr를 반환하게 된다면 std::shared\_ptr로 쉽게 변환할 수 있으므로 융통성이 생기는 것이다.

***std::unique\_ptr는 독점 소유권 의미론을 가진 자원의 관리를 위한, 작고 빠른 이동 전용 똑똑한 포인터이다.***

***기본적으로 자원 파괴는 delete를 통해 일어나나, 커스텀 삭제자를 지정할 수도 있다. 상태 있는 삭제자나 함수 포인터를 사용하면, std::unique\_ptr객체의 크기가 커진다.***

***std::unique\_ptr를 std::shared\_ptr로 손쉽게 변환할 수 있다.***

[항목19] std::shared\_ptr사용

소유권 공유 자원의 관리에는 std::shared\_ptr를 사용하라.

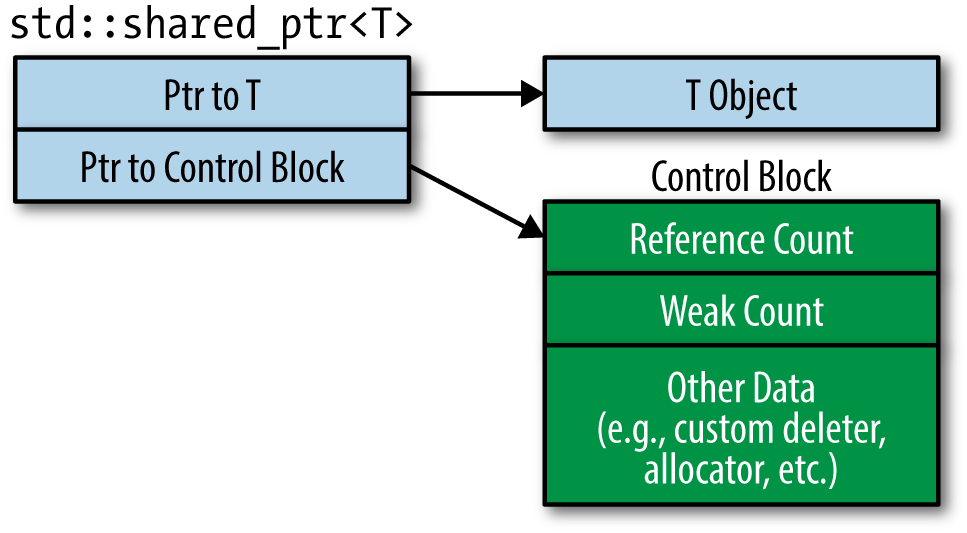
# std::shared\_ptr이란?

이에 대한 개략적인 내용은 ec++[항목13]에서 다루었다. 여기서는 조금 더 세분화하여 정리한다.

자동 작동 + 모든 자원에 적용 + 삭제 시점을 예측: std::shared\_ptr

가비지 컬렉션(자동 메모리 관리) + 수동 메모리 관리의 장점을 담았다.

# std::shared\_ptr의 구조



std::shared\_ptr에는 2가지 포인터가 있다. 그러므로 생포인터의 크기는 2배가 된다. 하나는 원본 포인터이고 나머지 하나는 제어블록을 가리키는 포인터다. 이 제어블록이라는 것은 std::shared\_ptr내부에 정의된 구조체로써 참조횟수와 약한 참조횟수, 삭제자 등을 담고 있다. 이에 대한 자세한 특징은 아래에서 살펴본다.

# std::shared\_ptr의 특징

### 크기가 생포인터의 2배이다.

위에서 언급한대로다.

### 참조 횟수를 담을 메모리를 반드시 동적으로 할당해야 한다.

객체가 언제 생성될 지 모르는데 정적으로 만들어놓을 수는 없는 노릇이다. 더 중요한 점은 서로 공유하는 자원인데 각 객체마다 공간을 마련하여 담아놓으면 공유도 어려울뿐더러 메모리도 심각한 낭비다.

### 참조 횟수의 증가와 감소가 반드시 원자적 연산이어야 한다.

#### 원자적 연산

understanding the linux kernel 책에 의하면,

원자적 연산은 '원자적인'방법 **즉, 중간에 어떠한 방해도 받지 않고   
어셈블리어 명령어 하나로 실행할 수 있다.**

- 메모리에 접근하지 않거나 한번 접근하는 어셈블리어 명령은  
원자적이다.

- inc나 dec같이 메모리에서 데이터를 읽고, 이를 갱신하고,  
갱신한 값을 메모리에 다시 스는 읽기/수정/쓰기 어셈블리어   
명령은 일기와 쓰기 사이에 다른 프로세서가 메모리  
버스를 점유하지 않는 한 원자적이다. 유니프로세서 시스템에서는  
같은 프로세서에서 메모리 접근이 일어나므로 메모리버스를 훔치는  
일은 발생하지 않는다.

- 연산 코드 앞에 lock(0xf0) 접두어가 붙은 읽기/수정/쓰기  
어셈블리 명령은 멀티프로세서 시스템에서도 원자적이다.  
제어 유닛이 이 접두어를 만나면 명령어 수행이 끝날때까지  
메모리 버스를 잠근다. 따라서 잠긴 명령을 실행하는 동안에는  
메모리에 다른 프로세서가 접근할 수 없다.

여러 스레드가 참조할 수 있기 때문이다. 만약 참조 카운터가 0이 되어 삭제하기 전에 다른 스레드가 점유하여 참조 카운터를 올려버리면 참조 카운터가 0이 아닌데 삭제되어버리는 이상한 현상이 보일 수 있다.

### std::shared\_ptr생성자가 참조를 증가시키지 않는 경우

이동 연산을 수행할 경우다. 당연한 것이 원본은 비게 되므로, 객체 수가 늘어난 것이 아니기 때문이다.

### 삭제자 사용

[항목18]에서 살펴본 std::unique\_ptr과 비교해본다.

**auto** **Deleter1** = [](Archer\* a){

std::cout << "normal delete a.**\n**";

**delete** a;

};

int main(){

*//타입에 삭제자 지정*

std::unique\_ptr<Archer, **decltype**(**Deleter1**)> upa(**new** Archer);

*//변수에 삭제자 지정*

std::shared\_ptr<Archer> spa(**new** Archer, **Deleter1**);

**return** 0;

}

비슷해 보이는데 무슨 차이가 있을까?

**auto** Deleter1 = [](Archer\* a){

std::cout << "normal delete a.**\n**";

**delete** a;

};

**auto** Deleter2 = [](Archer\* a){

std::cout << "special delete a.**\n**";

**delete** a;

};

int func(std::shared\_ptr<Archer> spa){

std::cout << "어떤 삭제자든 문제없이 받아들임.**\n**";

**return** 1;

}

int func(std::unique\_ptr<Archer> upa){

std::cout << "어떤 삭제자든 문제없이 받아들임.**\n**";

**return** 1;

}

int main(){

*//타입에 삭제자 지정*

std::unique\_ptr<Archer, **decltype**(Deleter1)> upa1(**new** Archer);

std::unique\_ptr<Archer, **decltype**(Deleter2)> upa2(**new** Archer);

*//변수에 삭제자 지정*

std::shared\_ptr<Archer> spa1(**new** Archer, Deleter1);

std::shared\_ptr<Archer> spa2(**new** Archer, Deleter2);

1. std::vector<std::shared\_ptr<Archer> > sp\_v{ spa1, spa2 };

3. **if** (func(spa1) + func(spa2) == 2)

std::cout << "미션 썩세스.**\n**";

2. spa1 = spa2;

*//std::vector<std::unique\_ptr<Archer> > up\_v = { upa1, upa2 }; // 안 됨*

*//func(upa1); // 형식이 다르므로 될 리가 없음*

*//upa1 = upa2; // 당연히 안 됨*

**return** 0;

}

1. 삭제자가 다른 2개를 같은 컨테이너에 담을 수 있다.  
2. 하나를 다른 하나에 배정할 수 있다.  
3. 삭제자가 다른 2개를 같은 매개변수로 넘길 수 있다.

# std::shared\_ptr::control\_block의 특징

참조를 관리하는 추가적인 자료구조이다. **최초로 객체를 생성하면 제어블록이 생기게 되는데 이미 존재한다면 그것을 공유하고 새로 만들지 말아야 할 것이다.** 이를 위해 몇 가지 규칙이 필요하겠다.

### 그 객체를 가리키는 shared\_ptr이 이미 존재하는지는 알 수 없다.

std::shared\_ptr을 선언할 때, 대상이 std::shared\_ptr이 아니라면 어떠한 공유되는 자료구조가 없으므로 어떠한 정보도 알 수 없다. **즉, 원본 대상에서 역추적할 수는 없다는 말이다.** 어떻게 알겠는가? 모른다.

### std::make\_shared는 항상 제어블록을 생성한다.

이 함수는 공유 포인터가 가리킬 객체를 새로 생성한다. **최초이므로 제어블록을 생성**해야 한다. – [항목21]

### 생 포인터로 생성하면 제어블록이 생긴다.

생 포인터에는 제어블록이 있을 수 없다. 앞서 말한 대로 생 포인터를 스마트 포인터로 감싸는 과정에서 이를 가리키는 또 다른 shared\_ptr이 있는지를 확인할 수는 없다. 이건 새로운 카운팅 객체가 되는 것이다. **그러므로 생 포인터에 여러 개의 참조 1개인 혹은 그 이상인 std::shared\_ptr이 있을 수 있다.** 하지만 이러면 하나의 std::shared\_ptr이 사라질 때 생 포인터의 메모리를 해제하므로 문제가 된다. 해결책은 아래에서 살핀다.

### 고유 소유권 포인터로부터 생성하면 제어블록이 생긴다.

당연하다. 이건 그냥 생 포인터와 개념이 같다.

### std::shared\_ptr이나 std::weak\_ptr로부터 생성하면 제어블록이 안 생긴다.

이것이 공유 개념이다. **같은 원본 포인터를 가리키며 대상 std::shared\_ptr이 가리키는 제어블록이 있을 것이므로 그 제어블록의 참조 카운터을 증가시키고 이 정보를 공유해야 한다.** 새로 생긴 std::shared\_ptr은? 그 제어블록을 가리키는 것이다! 그러므로 새로운 제어블록이 생기면 안 된다.

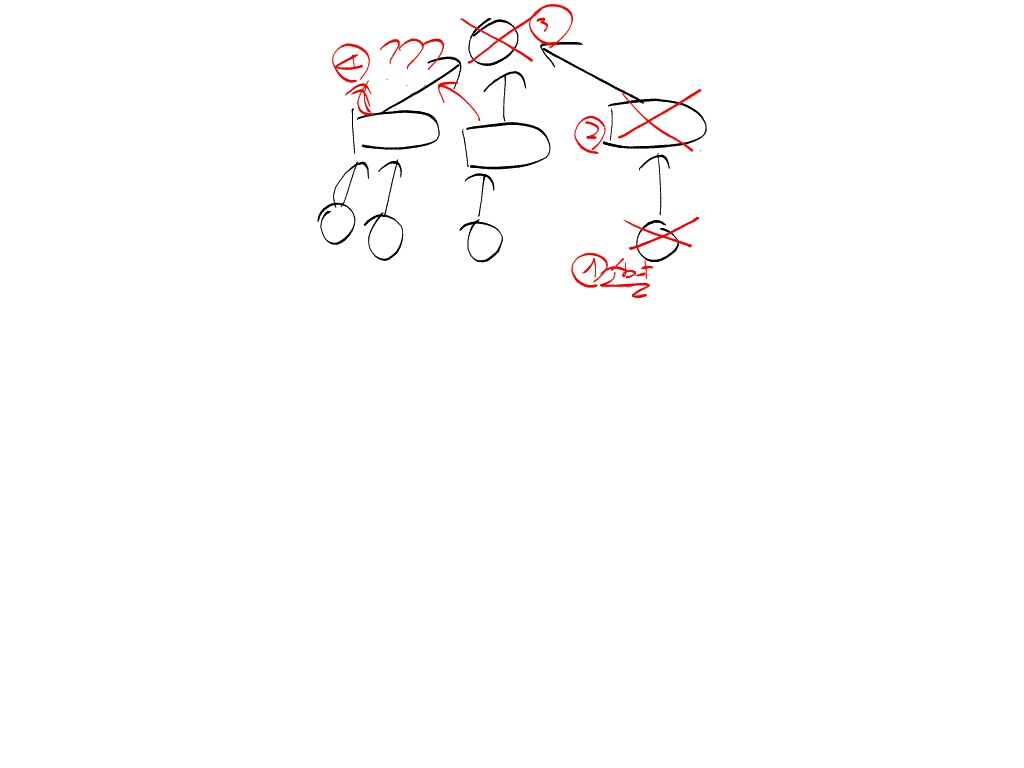
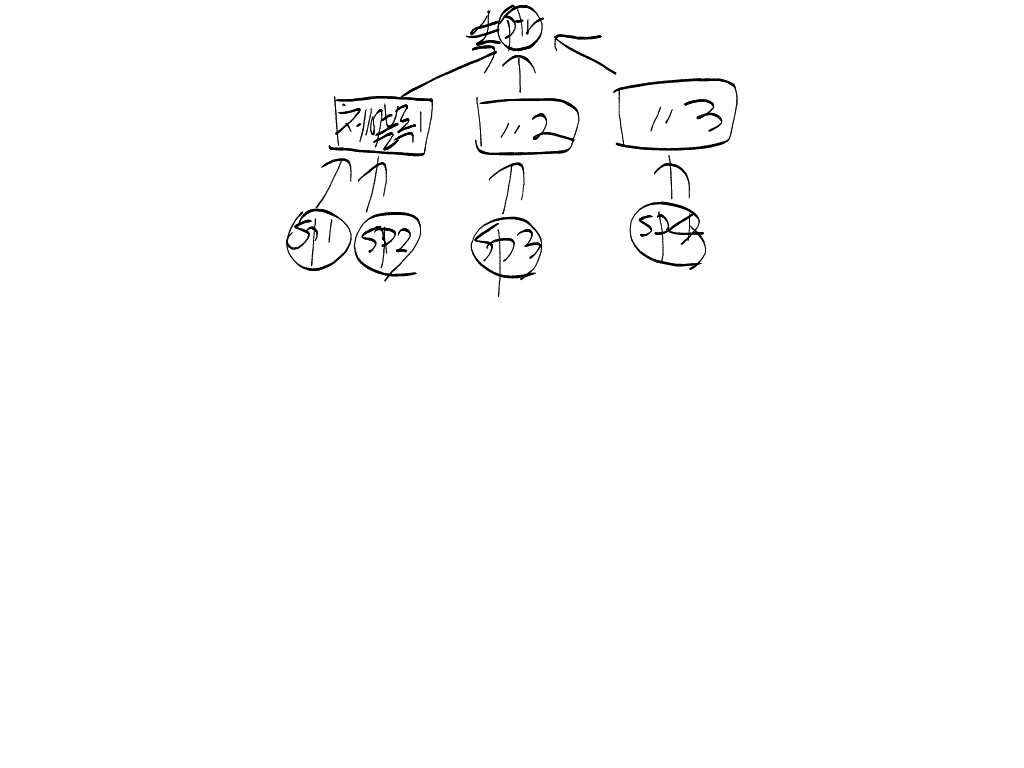
# std::shared\_ptr은 어떻게 생성하는가?

일반적인 객체 생성처럼 생성하면 되지만 위의 control\_block의 특징 때문에 생성시 고려해야 할 점이 있다.

## 생 포인터로 여러 개의 std::shared\_ptr생성

생 포인터로 std::shared\_ptr을 만들면 제어블록이 생긴다고 했는데, 하나의 원본 포인터에 제어블록이 여러 개면 무슨 일이 생기는가?

아래 좌측 그림은 본래의 포인터 현황이며 우측은 sp4가 소멸될 시, 생기는 현상이다.



sp1, sp2, sp3는 억울하게 nullptr을 바라보게 되었다.

### 어떻게 해야 하는가?

std::shared\_ptr<Archer> spa1(**new** Archer, Deleter1);

표현식 내부에 new로 선언해버리면 임시로 생성되었다가 사라지므로 다른 곳에서 쓸 수 없게 된다.

## this로 여러 개의 std::shared\_ptr생성

위의 현상과 동일하다. C++은 이를 해결하기 위한 템플릿을 제공한다.

**class** **Archer** : **public** std::enable\_shared\_from\_this<Archer>{

**public**:

void attack(){

*//sp\_v.emplace\_back(this);*

sp\_v.emplace\_back(shared\_from\_this());

}

};

**public** std::enable\_shared\_from\_this<Archer>를 상속받고 제어블록을 생성할 최초의 this를 얻기 위해서 shared\_from\_this()함수를 이용한다.

여기서 생길 수 있는 문제는 std::shared\_ptr이 유효한 객체를 가리키기도 전에 shared\_from\_this()를 호출하는 일이다. 있지도 않은 shared\_ptr을 찾는 일이다. **즉, 모든 객체의 생성은 shared\_ptr의 생성과 동시에 되어야 한다.** 그러므로 일반 생성자를 private에 넣어두고, 펙토리 함수로 shared\_ptr기반의 객체를 반환하도록 설계한다. 아래와 같다.

**class** **Archer**;

std::vector<std::shared\_ptr<Archer> > sp\_v;

**class** **Archer** : **public** std::enable\_shared\_from\_this<Archer>{

**public**:

**template**<**typename** T>

**static** std::shared\_ptr<Archer> create(){

**return** std::shared\_ptr<Archer>(**new** Archer);

}

void attack(){

*//sp\_v.emplace\_back(this);*

sp\_v.emplace\_back(shared\_from\_this());

}

**private**:

Archer() {}

};

int main(){

std::shared\_ptr<Archer> spa1 = Archer::create<int>();

**return** 0;

}

### 묘하게 되풀이되는 템플릿 패턴(Curiously Recurring Template Pattern, CRTP)

위에서 자기 자신을 템플릿으로 갖는 정적 객체를 상속받는다는 것이 이상하지 않은가?

참고: <http://alleysark.tistory.com/234>

c++에서 상속을 통해 부모의 속성을 물려받게 된다.

그런데 static 필드의 경우 상속을 하더라도 static하게 남아있는데**, derived class마다 static 필드를 새롭게 가지고 싶을 때**가 있다.

이런 경우 사용할 수 있는 c++ 패턴이 ***CRTP(Curiously Recurring Template Pattern)***이다.

페러다임이 참 독특하다. 부모가 상속하게 될 자식클래스의 타입을 템플릿타입으로 받아 컴파일단계에서 새로운 클래스로 처리되도록 한다.

서로 다른 두 자식클래스 DerivedA, DerivedB에서 부모클래스 Base를 상속할 때 템플릿 타입으로 자기 자신을 한다. **컴파일러는 부모의 타입을 Base<DerivedA>와 Base<DerivedB> 두 개의 서로 다른 타입으로 처리하기 때문에 Base클래스의 static 필드 또한 서로 다른 메모리를 가지게 되는 것이다!**

메소드에 대해선 조금 다른 식으로 처리한다. 부모클래스는 static 메소드의 인터페이스를 정의하고 내부에서 자식클래스의 static 구현 메소드를 호출한다. 모양이 썩 예쁘진 않지만 기능은 확실히 하니 필요한 순간이 있으리라.

**template**<**class** **DerivedType**>

**class** **Base**{

**public**:

**static** int value;

...

**static** void InterfaceFoo(Parameter parameter){

DerivedType::ImplementedFoo(parameter);

}

};

**template** <**class** **DerivedType**> int Base<DerivedType>::value = INITIAL\_VALUE;

**class** **DerivedA** : **public** Base<DerivedA>

{

...

**static** void implementedFoo(Parameter parameter){

*// foo for DerivedA's instances*

}

};

**class** **DerivedB** : **public** Base<DerivedB>

{

...

**static** void implementedFoo(Parameter parameter){

*// foo for DerivedB's instances*

}

};

**template**<**class** **DerivedType**>

**class** **Counter**{

**private**:

**static** int count;

**public**:

Counter(){

count++;

}

~Counter(){

count--;

}

};

**template** <**class** **DerivedType**> int Counter<DerivedType>::count=0;

이 패턴은 오브젝트 카운터의 구현에 사용될 수 있다. 혹은 polymorphic copy construction 페러다임의 구현에도 활용할 수 있다.

***std::shared\_ptr는 임의의 공유 자원의 수명을 편리하게 관리할 수 있는 수단을 제공한다.***

***대체로 std::shared\_ptr객체는 std::unique\_ptr객체의 두 배이며, 제어 블록에 관련된 추가 부담을 유발하며, 원자적 참조 횟수 조작을 요구한다.***

***자원은 기본적으로 delete를 통해 파괴되나, 커스텀 삭제자도 지원된다. 삭제자의 형식은 std::shared\_ptr의 형식에 아무런 영향도 미치지 않는다.***

***생 포인터 형식의 변수로부터 std::shared\_ptr를 생성하는 일은 피해야 한다.***

[항목20] std::weak\_ptr사용

std::shared\_ptr처럼 작동하되 대상을 잃을 수도 있는 포인터가 필요하면 std::weak\_ptr을 사용하라.

# std::weak\_ptr이란?

개략적인 내용은 ec++[항목13] 참고

## 배경

어떠한 대상을 가리키고 싶다. 물론 원시 포인터도 가능하다. 하지만 이 대상은 여러 포인터를 공유하고 있다. 그리고 어딘가에 이 객체의 생명을 좌우할 std::shared\_ptr이 존재한다고 가정하자. 여기서 이 객체를 가리키고 싶다면

class Archer{};

std::shared\_ptr<Archer> sp1(std::make\_shared<Archer>());

std::shared\_ptr<Archer> sp2 = sp1;

이렇게 할 수 있다. 하지만 이것은 우리가 원하는 것이 아니다! 이 객체가 사라지려면 sp1, sp2가 사라져야 하는데 우리는 sp2는 살아있어도 객체를 제거해야 할 수 있어야 한다.

이럴 때 쓰는 스마트 포인터가 std::weak\_ptr이다.

## 정의

std::shared\_ptr과 공유된다. 그리고 shared\_ptr이 가리키는 객체와 동일한 객체를 가리키며, 참조 횟수에는 영향을 주지 않는다. 정확하게는 생명에 영향을 주지 않는다.

# std::weak\_ptr사용하기

다음과 같이 사용한다.

int main() {

std::shared\_ptr<Archer> sp1(std::make\_shared<Archer>());

std::weak\_ptr<Archer> wp1(sp1);

return 0;

}

[이미지]

아래는 허용되지 않는다. std::shared\_ptr은 보강하는 것이지 그 자체로 생명주기를 갖지 않기 때문이다.

std::weak\_ptr<Archer> wp1(new Archer);

# std::weak\_ptr의 원리

어떻게 이를 가능하게 하는 것일까?

## 만료 여부 점검

일단 포인터가 살아 있는지, 없는지 확인해야 할 것이다.

if (!wp1.expired()) std::cout << "exist.\n";

## 객체를 참조하자!

기본적인 생각으로는 만료가 되지 않았다면 weak\_ptr이 가리키는 객체를 참조하면 될 것이다. 하지만 weak\_ptr은 이런 용법은 없다. (operator\*, get()등) 이유는 스레드 안전성과 연결된다. 만료 점검과 참조를 분리하면 없는 객체를 있다고 할 수도 있다.

## std::shared\_ptr반환

바로 이것이다. 스레드 안전성과 관련해서도 객체 자체를 반환하여 안전성을 보장한다. 이를 반환하는 함수를 사용하여 weak\_ptr을 참조한다고 생각해도 되겠다.

첫 번째 방법: 확인 후, 반환

if (!wp1.expired()) {

//만일 다중 스레드로 인하여 wp1이 만료되었다면,

//sp2는 nullptr이다.

auto sp2 = wp1.lock();

}

두 번째 방법: 생성자 사용

std::weak\_ptr<Archer> wp2;

//생성자를 이용하여 wp1의 값을 가져올 수 있다.

//만약 wp1이 만료라면, std::bad\_weak\_ptr이 발생한다.

//프로그램이 죽어버린다.

std::shared\_ptr<Archer> sp3(wp2);

이렇게 하면 성공적으로 std::weak\_ptr을 사용할 수 있다.

# std::weak\_ptr을 언제 쓸까?

유용해 보이지만 ‘이걸 뭐 하러 만들었을까?’라는 생각이 들 것이다.

## cache

어떤 연산을 수행 후, 결과값을 많이 이용해야 하는 프로그램이 있다. 이 때, 했던 계산을 또 해야 하는 경우가 많을 것이다. 이럴 때 우리는 DP를 이용한다. 그러므로 결과값을 캐시에 저장하여 연산을 반복하지 않는다.

여기서 이 객체를 무엇으로 가리킬까? ‘std::shared\_ptr, std::unique\_ptr이요!’ 아주 멍청한 생각이다. 아마 프로그램이 끝날 때까지 객체가 소멸하지 않을 것이다. 원본을 가리키는 포인터는 하나의 포인터는 끝까지 살아있을 것이기 때문이다. 필요 없는 객체는 캐시에서 제거해야 하며 이때, 이를 가리키는 포인터를 쉽게 정리할 방법이 있을까?

이때 이 포인터를 std::weak\_ptr로 설정하면 유용하다. 아래 그림으로 확인한다.

[이미지]

std::shared\_ptr<Archer> fastLoadArcher(ArcherID id) {

//ArcherID을 key값으로 std::weak\_ptr<Archer>을 반환한다.

//std::weak\_ptr은 std::shared\_ptr을 반환한다고 앞에서 언급

static std::unordered\_map<ArcherID, std::weak\_ptr<Archer> > cache;

auto ret = cache[id].lock();

//캐시에 없으면 새로 만들어 최초로 담는다.

if (!ret) {

ret = loadArcher(id);

cache[id] = ret;

}

return ret;

}

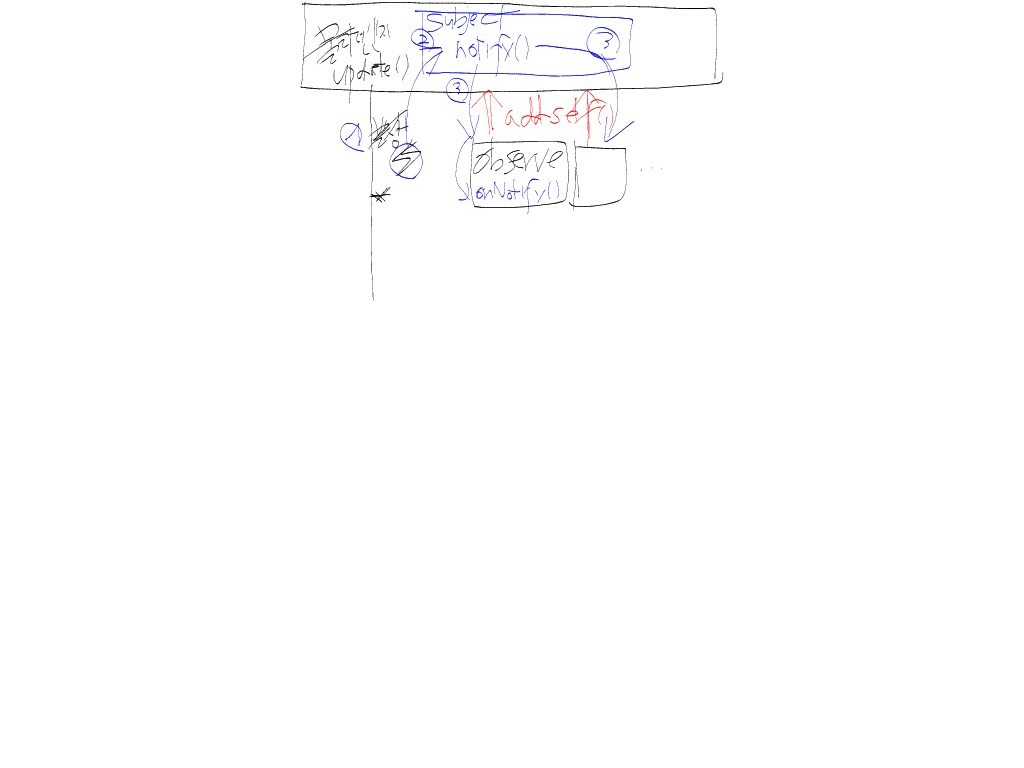
std::shared\_ptr<Archer> loadArcher(ArcherID id) {

return std::make\_shared<Archer>();

}

## 관찰자 패턴

이벤트(event)와 비슷하다. 간략하게 설명한다.



저 observer가 subject를 가리키고 있는 것이 관찰하는 것이다. 즉, 구독하면서 반응이 생기면 바로 무언가를 실행한다. 이는 어떤 포인터로 구현해야 할까? 물리엔진 subject가 사라져야 하는데 계속 포인터로 가리키고 있어 살아있는 괴이한 현상을 목격하기 싫다면 단연 std::weak\_ptr이 적합함을 이해할 수 있다.

## 순환 참조 문제

[이미지]

다음과 같은 상황이 있다. A와 B를 서로 std::shared\_ptr로 연결하면 어떠한 것도 저절로 파괴되지 않으며 영원히 참조 카운터는 1이 된다.

이것을 std::weak\_ptr로 바꾸면 해결된다. B는 A를 가리키지만, 참조 카운터는 증가시키지 않으므로 A는 사라질 수 있다. 정상적인 작동이다.

***std::shared\_ptr처럼 작동하며 대상을 잃을 수도 있는 포인터가 필요하면 std::weak\_ptr를 사용하라.***

***std::weak\_ptr의 잠재적인 용도로는 캐싱, 관찰자 목록, 그리고 std::shared\_ptr순환 고리 방지가 있다.***

[항목21] std::make\_shared의 사용

new를 직접 사용하는 것보다 std::make\_unique와 std::make\_shared를 선호하라.

# std::make\_unique, shared이 뭐지?

std::shared\_ptr이나 std::unique\_ptr을 만드는데 사용한다.

std::shared\_ptr<Archer> sp1(new Archer);

기본적으로 new로 생성자 호출 및 동적 할당하여 스마트포인터로 감싼다.

**auto sp2(std::make\_shared<Archer>());**

이렇게 만들 수도 있다. 그리고 이 장에서는 이 방법을 권장하고 있는 것이다.

## std::make\_unique구현

// note: this implementation does not disable this overload for array types

template<typename T, typename... Args>

[std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr)<T> make\_unique(Args&&... args)

{

return [std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr)<T>(new T([std::forward](http://en.cppreference.com/w/cpp/utility/forward)<Args>(args)...));

}

인자를 완벽 전달하여 그것을 이동하여 새로운 포인터를 반환하는 과정이다. 결국, 인자는 하나의 객체(rvalue or lvalue)가 들어간다.

# 이걸 왜 쓰지?

힘들게 이런 함수는 왜 만들었을까?

## 간단한 표현식

위에서만 비교해도 알 수 있다. 위는 Archer가 2번이나 나온다. **핵심 원리인 “코드 중복을 피하라.”**를 기억한다.

## 예외 안전성

이는 ec++[항목]에서 하나의 주제로 다룬다. 간단하게 언급한다.

void attackTarget(std::shared\_ptr<Archer> sp, CalcPower cp) {

}

int main() {

auto sp2(std::make\_shared<Archer>());

attackTarget(std::shared\_ptr<Archer>(new Archer), CalcPower());

}

attackTarget을 보자. 안에 매개변수에 1가지가 아닌 여러 가지의 행위가 진행될 것이다. 문제는 이 순서가 정해지지 않는다는 것이다.

**핵심 원리는 하나의 행위는 그와 관련된 것들을 감싸고 있어야 한다는 점이다.** 안 된다면 한번 저장을 하고(변수에 할당) 다음 과정을 진행해야 한다.

## 효율성

make\_shared에 한하여, 메모리 블록을 만들어낸다. 본 객체를 가리킬 메모리를 할당하고 한 번 더 제어 블록에 대한 메모리를 할당해야 한다. 하지만 make\_shared는 이를 하나로 묶어서 한번만 동적 할당을 하여 처리할 수 있다.

# 쓰면 안 되는 경우

그러므로 make\_shared, unique는 사용할 수 있다면 쓰는 것이 권장된다. 하지만 항상 쓸 수는 없다.

## 삭제자 지정

make\_shared, unique로는 삭제자를 지정할 수 없다.

## 중괄호(std::initializer\_list)

중괄호를 인자로 생성할 수 없다.

auto sp3(std::make\_shared<std::initializer\_list<int> >({1, 2, 3})); *// error*

auto sp4(std::make\_shared<std::vector<int> >(10, 20));

make\_shared에서 ()안에 vector<int>의 생성자에 들어가는 인자임을 볼 수 있다. 이는 구문적 특징이다. 중괄호를 인자로 생성하려면 미리 변수에 할당한 후, 생성해야 한다.

auto arg = { 1, 2, 3, 4, 5 };

auto sp5(std::make\_shared<std::initializer\_list<int> >(arg));

## 사용자 지정 operator new, delete

std::shared\_ptr에 의한 할당, 해제해야 할 메모리의 크기가 객체의 크기뿐 만이 아니라, 제어 블록의 크기가 더해진다. 그러므로 make\_shared는 이 더한 값을 요구하므로 이러한 경우에는 적합하지 않다.

## 메모리 해제

일반 생성은 제어블록과 본체의 메모리가 구분되어 있다. 그래서 본체가 제거된 이후에, 나머지 std::weak\_ptr이 가리키는 객체들을 해제한다. 하지만 make\_shared에서는 하나의 메모리로 되어 있으므로 모두 해제한 다음에 본체가 해제된다. 이런 점에서 시간 지연으로 인한 객체 유무 차이가 생길 수 있다.

### 시간 지연이 뭐가 문제일까?

객체의 메모리가 제법 큰 경우에만 해당된다. 그러면 본체가 해제되는 시간과 std::weak\_ptr이 해제되는 시간의 격차가 생기고 이 동안에 메모리를 해당 메모리를 사용하지 못하므로 열악한 환경에서는 안 좋을 수 있다.

# std::make\_unique, shared 결론

최대한 make\_shared, unique를 사용해야 한다. 하지만 사용할 수 없는 경우에는, new로 생성하되, 예외를 주의해야 한다.

## new생성과 예외 처리

다음과 같은 고려로 예외에서 안전해야 한다.  
1. 예외가 가능한 표현식은 따로 나누어서 저장하고 진행해야 한다.  
2. 이렇게 저장된 변수는 lvalue가 된다. 이를 개선하기 위해서 오른값으로 변경할 수 있어야 한다.   
3. 아래와 같이 만들면 된다.

auto a1 = std::shared\_ptr<Archer>(new Archer);

//attackTarget(a1, CalcPower());

attackTarget(std::move(a1), CalcPower());

***new의 직접 사용에 비해, make함수를 사용하면 소스 코드 중복의 여지가 없어지고, 예외 안전성이 향상되고, std::make\_shared와 std::allocate\_shared의 경우 더 작고 빠른 코드가 산출된다.***

***make함수의 사용이 불가능 또는 부적합한 경우로는 커스텀 삭제자를 지정해야 하는 경우와 중괄호 초기치를 전달해야 하는 경우가 있다.***

***std::shared\_ptr에 대해서는 make함수가 부적합한 경우가 더 있는데, 두 가지 예를 들자면 (1) 커스텀 메모리 관리 기능을 가진 클래스를 다루는 경우와 (2) 메모리가 넉넉하지 않은 시스템에서 큰 객체를 자주 다뤄야 하고 std::weak\_ptr들이 해당 std::shared\_ptr들보다 더 오래 살아남는 경우이다.***

[항목22] Pimpl관용구 구현

Pimpl관용구를 사용할 때에는 특수 멤버 함수들을 구현 파일에서 정의하라.

# Pimpl관용구

클래스의 자료 멤버들을 구조체에 담아두고, 본체는 이 구조체의 포인터만 갖고 있는 형태이다.

## Pimpl관용구의 특징

**Benefits:  
1.** Changing private member variables of a class does not require recompiling classes that depend on it, thus **make times are faster**, and **the FragileBinaryInterfaceProblem is reduced**. **2.** The header file does not need to #include classes that are used 'by value' in private member variables, thus **compile times are faster**. **3.** This is sorta like the way SmallTalk automatically handles classes... **more pure encapsulation**.

**Drawbacks:  
1.** More work for the implementor. **2.** Doesn't work for 'protected' members where access by subclasses is required. **3.** Somewhat harder to read code, since some information is no longer in the header file. **4.** Run-time performance is slightly compromised due to the pointer indirection, especially if function calls are virtual (branch prediction for indirect branches is generally poor).

**Implementation:**

//in header Widget.h

class Widget{

public:

Widget();

...

private:

struct Impl;

Impl\* pImpl;

};

## Pimpl관용구의 핵심 원리

*불완전 형식*이라는 것이 존재한다. 이는 선언만 있고 정의가 없는 형식을 의미한다. 모든 구현 파일을 하나로 만들 수 없으므로 어떤 자료가 필요할 때에는 이를 보여줄 수 있어야 하는데, 모든 구현을 보여주려면 그만큼 코드도 길어지며 컴파일 시간도 길어질 것이다. **하지만 선언만 할 수 있다면 이를 간단하게 만들어준다.** 이를 클래스에 적용하여 자료에도 적용하는 것이 Pimpl관용구라 하겠다.

# 특수 멤버 함수

컴파일러가 자동으로 생성해주는 생성자, 복사, 이동자 - [항목17]참고

**이가 불완전 형식과 만날 때에는 특별히 주의해야 한다.**

# std::unique\_ptr을 이용한 Pimpl관용구 구현

한 클래스 안의 자료 덩어리를 Impl이라고 정의한다. 이는 클래스 안에 몇 개나 있어야 할까? 당연히 1개이다. 그에 가장 적합한 포인터인 std::unique\_ptr을 사용한다.

## 구현: impl포인터를 std::unique\_ptr로 대체

class Archer

{

public:

Archer();

//소멸자를 정의하지 않으면, 불완전 형식으로 컴파일이 안 된다.

private:

struct impl;

std::unique\_ptr<impl> pImpl;

};

## 문제: 불완전 형식으로 인한 오류

위의 소스에서 impl은 분명히 불완전 형식이다. 구현은 cpp파일에 따로 구현되어 있기 때문이다. 문제는 기본 삭제자 함수가 delete하기 전에 불완전 형식 여부를 검사한다는 점이다. 아래는 std::unique\_ptr의 기본 삭제자 함수다.

struct default\_deleter

// …

void operator()(\_Ty \*\_Ptr) const \_NOEXCEPT

{ // delete a pointer

static\_assert(0 < sizeof (\_Ty),

"can't delete an incomplete type");

delete \_Ptr;

}

};

그럼 왜 불완전 형식이 문제가 되는 것일까?

### 불완전 형식을 바라보는 소멸자

기본적으로 컴파일러는 해당 파일에서 찾고자 하는 자료의 정의가 없으면 불완전 형식으로 간주한다. 위의 경우가 그러한 경우다. 소멸자는 호출하는데 소멸자가 없다.

*Q. 특수 멤버 함수는 컴파일러가 자동으로 만들지 않는가?*

문제는 자동으로 만들어진 소멸자는 암묵적으로 inline으로 선언되어 여전히 헤더파일에 남게 된다는 것이다. 이렇게 되면 어떻게 될까? **소멸자가 실행되는 시점에는 헤더파일에 있게 된다.** 그럼 구현은? 여전히 볼 수 없다. 이것이 본질적인 문제이다.

### 다른 특수 멤버 함수는 어떨까?

마찬가지다. 헤더파일에서 = default 를 사용했다면, 이 역시 오류를 내뿜을 것이다.

class Archer

{

public:

Archer();

~Archer();

Archer(const Archer&) = default;

Archer& operator=(const Archer&) = default;

Archer(Archer&&) = default;

Archer& operator=(Archer&&) = default;

private:

struct impl;

std::unique\_ptr<impl> pImpl;

};

같은 원리인데, 대상 Archer을 이동한다고 했을 때, 함수를 호출하는 시점은 헤더파일에 있게 된다. 근데 여기에서는 impl구현이 없으므로 컴파일러는 화를 낼 것이다.

## 해결: 특수 멤버 함수 호출 시점에 구현을 바라보게 하기

간단하다. **소멸자가 구현파일에 정의되어 있으면 자연히 구현 파일을 보게 되며, 그에 대한 정의를 볼 수 있게 된다.** 아래와 같다.

struct Archer::impl

{

std::string name;

DefaultAllow myAllow;

};

Archer::Archer()

: pImpl(std::make\_unique<impl>())

{

std::cout << "create Archer successfully.\n";

}

//.cpp파일에 정의해야 한다.

Archer::~Archer() = default;

//깊은 복사

Archer::Archer(const Archer& rhs)

: pImpl(nullptr)

{

std::cout << "try copy Archer.\n";

if (rhs.pImpl)

pImpl = std::make\_unique<impl>(\*rhs.pImpl);

}

Archer& Archer::operator=(const Archer& rhs)

{

std::cout << "try copy Archer.\n";

if (!rhs.pImpl) pImpl.reset();

else if (!pImpl)

pImpl = std::make\_unique<impl>(\*rhs.pImpl);

else \*pImpl = \*rhs.pImpl;

return \*this;

}

Archer::Archer(Archer&& rhs)

: pImpl(std::move(rhs.pImpl))

{

std::cout << "try move Archer.\n";

}

Archer& Archer::operator=(Archer&& rhs)

{

std::cout << "try move Archer.\n";

pImpl = std::move(rhs).pImpl;

return \*this;

}

위에 떡 하니 impl정의가 있지 않은가?

### 구현에 대한 세부사항

*이동 생성자, 이동 대입 연산자*

크게 문제되지 않는다. std::unique\_ptr가 자체적으로 대상을 하나만 가리키게 되므로 rhs.impl만 이동해주면(std::move()) 될 것이다.

*복사 생성자, 복사 대입 연산자*

포인터만 복사한다면 대상은 1개이므로 얕은 복사가 된다. 깊은 복사를 하려면, 실제로 데이터가 하나 더 만들어야 한다. 같은 원본 포인터로 새로운 std::unique\_ptr을 만들어서 impl객체를 복사해야 할 것이다. 위의 구현은 rhs.impl가 유효할 때만 적용됨을 보여준다.   
impl에 아무것도 없다면, std::make\_unique<impl>(\*rhs.pImpl)로 포인터를 새로 만든다. 이는 \*로 인해 포인터만 만드는 것이 아니라 새로운 객체에 포인터로 씌움을 말한다.  
또한 \*pImpl = \*rhs.pImpl역시 객체 자체를 나타내며, 포인터는 이미 있으므로 impl객체만 새로 복사함을 표현한다.

## 핵심: 불완전 구현과 컴파일러

**핵심은 어떤 함수를 호출할 때, 그 함수는 어디에서 어느 시점(컴파일인지 런타임인지)에 진행되는 지를 파악할 필요가 있으며, 필요한 구현이 빠졌는지 확인해볼 필요가 있다.**

# std::shared\_ptr을 이용한 Pimpl관용구 구현

위의 std::unique\_ptr과는 다르게 특수 멤버 함수의 구현과 상관없이 컴파일된다.

class Archer

{

public:

Archer();

~Archer();

Archer(const Archer&) = default;

Archer& operator=(const Archer&) = default;

Archer(Archer&&) = default;

Archer& operator=(Archer&&) = default;

private:

struct impl;

std::shared\_ptr<impl> pImpl;

};

## std::unique\_ptr과의 차이점

삭제자의 구현과 연관이 있다.

std::unique\_ptr은 삭제자가 포인터 형식의 일부분이므로 삭제자의 타입을 빠르게 접근할 수 있다. 곧, 더 작은 자료구조와 더 빠른 실행시점 코드가 된다. 그리고 그 더 빠른 시점에 호출하는 특수 멤버 함수들은 완전한 형태여야 한다. **여기서 호출되는 시점은 컴파일 시점이다. 컴파일 시점에 완전한 형태가 아니라면 문제가 된다.** **컴파일 시점에 완전한 형태를 파악할 수 있게 하거나 실행 단위까지 가라고 지시해야 한다.**

반면, std::shared\_ptr은 삭제자가 형식의 일부가 아니다. 제어블록 클래스 안에 들어있으며, 자연히 자료구조는 커진다. 중요한 점은 소멸자가 헤더파일에 있을 지라도, 소멸자가 호출되는 시점은 제어블록 내부까지 들어간 상태이다**. 컴파일 시점에서 불가능한 일이며 런타임 시점에 소멸자를 호출해야 한다.** **런타임에는 실행 파일들이 이미 링크되었으므로 정상적으로 작동한다.** 더 긴 시간이 필요하지만 컴파일 시점에 완전한 형태일 필요는 없는 것이다.

역시 이러한 차이점의 근본은 컴파일 타임과 런타임의 차이일 것이다. 위에서 언급한 핵심에서처럼, 실행 시점에 필요한 구현을 확인할 수 있는지 확인해야 한다.

***Pimpl관용구는 클래스 구현과 클래스 클라이언트 사이의 컴파일 의존성을 줄임으로써 빌드 시간을 감소한다.***

***std::unique\_ptr형식의 pImpl포인터를 사용할 때에는 특수 멤버 함수들을 클래스 헤더에 선언하고 구현 파일에서 구현해야 한다. 컴파일러가 기본으로 작성하는 함수 구현들이 사용하기에 적합한 경우에도 그렇게 해야 한다.***

***위의 조언은 std::shared\_ptr에는 적용되지 않는다.***

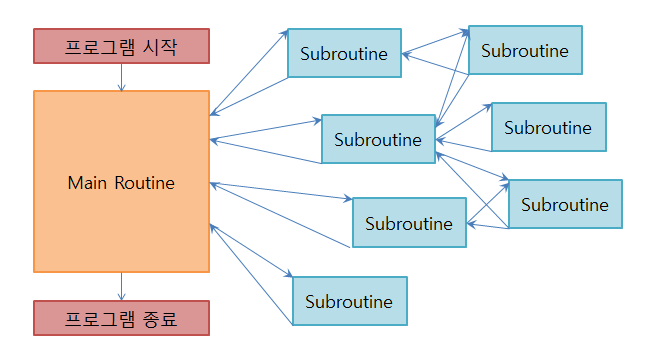
[항목35] 스레드 기반 vs. 과제 기반

스레드 기반 프로그래밍보다 과제 기반 프로그래밍을 선호하라.

스레드에 대해 이해하려면 프로그램의 흐름에 대해 이해할 필요가 있다.

# 흐름(routine)에 대하여

routine은 어떤 코드의 흐름을 말한다. 모든 프로그램은 시작과 동시에 하나의 **Main Thread(주 스레드)**를 가진다. 그리고 이 Main Thread에서 **Main Routine(main함수)**이 불려진다. 아래는 아주 기본적인 프로그램의 시작과 동시에 메인 루틴과 서브 루틴이 수행되는 모습이다.

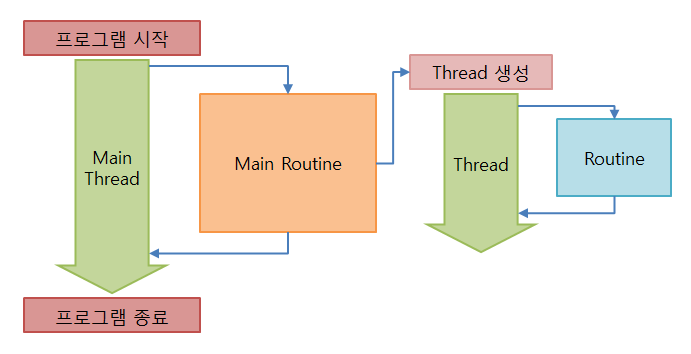


그림에서 보이듯이, 메인 루틴 중간에 서브 루틴이 끼어든다. 이건 새로운 흐름이 생기는 것이 아니라, 기존의 주된 흐름에 어떠한 흐름이 끼어드는 것이다.

## thread에 대하여: 스레드와 흐름의 관계

**Thread**란 프로그램 내에서 실행되는 흐름의 단위를 말합니다. Thread는 흐름의 단위이기 때문에 새로운 Thread가 만들어졌다는 것은 새로운 시간 흐름이 만들어졌다고도 볼 수 있습니다.   
이렇게 프로그램은 여러 개의 Thread를 동시에 실행할 수도 있고, 이것으로 인해 일종의 흐름이 동시에 진행될 수 있습니다. 이러한 실행 방식을 **Multithread(멀티스레드)**라고 합니다.

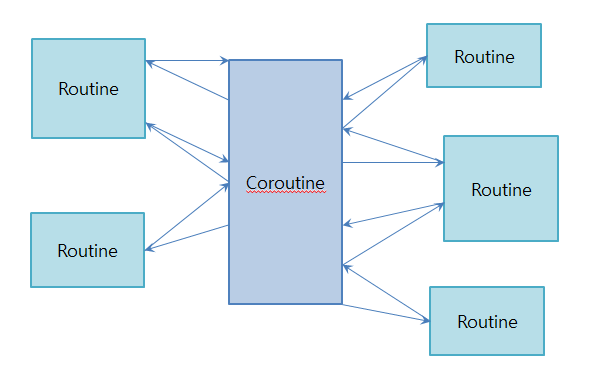
스레드에서는 흐름이 어떤 모습일지 그림으로 확인해보자.



독립적인 시간 흐름을 가지고 Routine을 실행하는 것이 바로 Thread인 것입니다. 그래서 보통의 Routine들은 시작부터 끝까지 하나의 Thread에서 실행됩니다.

## 또 다른 흐름: coroutine

위에서 subroutine을 다시 보자. 기본적으로 루틴이란, 진입하기 위해서 처음부터 진입해야 한다. 메인 루틴이든 또 다른 서브 루틴이든 다른 루틴에 진입하기 위해서는 그 루틴의 처음 지점으로 진입해야 한다. 적어도 우리가 일반적으로 알고 있는 함수, subroutine은 그렇다.

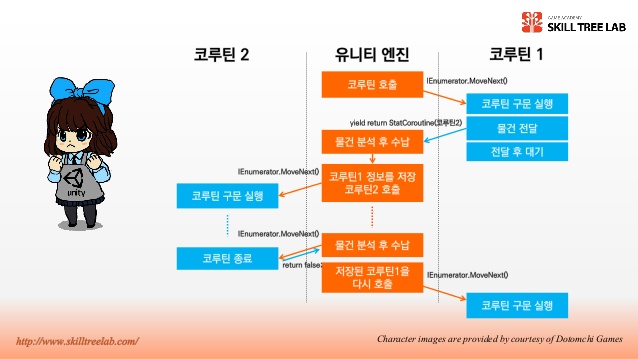


반면, 진입하는 지점을 여러 개 가질 수 있는 Subroutine을 **Coroutine(코루틴)** 이라고 한다. Coroutine은 호출한 Routine을 대등한 관계로 호출할 수 있기 때문에 다른 Routine의 종속관계가 아니라고 표현하기도 한다.

결국, Coroutine은 함수 내에서 호출한 쪽을 다시 호출할 수 있고 다시 다른 Routine에서 함수의 중간 지점을 호출할 수 있는 것이라고 할 수 있다.

## coroutine과 thread의 관계

아무런 관계는 없다. 둘이 전혀 다른 것이다. 하지만 코루틴으로 스레드의 효과를 낼 수는 있다. 어떻게 그럴 수 있을까?



다음 그림을 보자. 엔진은 메인 루틴이라고 볼 수 있겠다. 그리고 코루틴1과 코루틴2를 실행하기 원한다. 그리고 동시에 수행하고 있다. 물론 일정한 순서에 따라서 진행하지만 진입지점을 임의로 가질 수 있다는 점이 실행을 왔다갔다하게 해주며 동시에 일어나는 듯 보인다. 게다가 순서가 명확하므로 동기화의 문제도 없다.

출처: <http://www.gamedevforever.com/291>

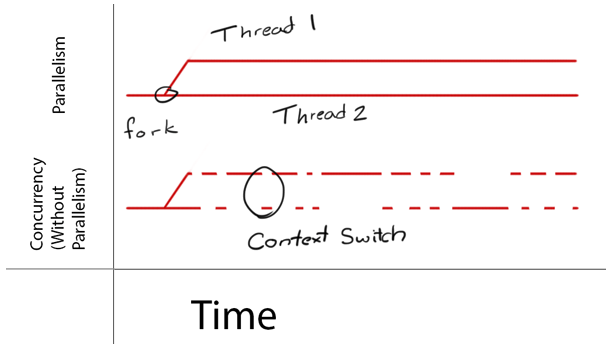
# multi-thread programming

스레드의 기본 개념은 위에서 확인한다.

하나의 프로세스에서 병렬적으로 여러 개 작업을 처리하기 위해서는 각 작업을 스레드화하여 멀티스레딩이 가능하게 해야 한다.

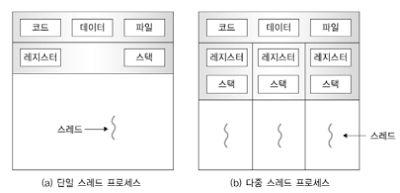
### 동시성(Concurrency) VS 병렬성(Parallelism)

동시성은 두 개 이상의 과업이 시작, 실행, 완료 등이 같은 시간대에서 행하여지는 것을 뜻한다. 그런데 주의 할 것은 두 과업이 반드시 같은 순간에 실행되는 것을 의미하지 않는다. 예를 들어, 싱글 코어 CPU에서 시간을 쪼개어 동시에 작업 중인 과업을 왔다 갔다 하면서 연산한다. 반면 병렬성은 말 그대로 동시에 과업을 수행하는 것을 뜻한다. 멀티 코어 CPU에서 가능하다.



### 스레드 시간 분담

프로세스가 CPU에 대한 사용권을 운영체제로부터 확보하면 일정 시간만큼 CPU에게 처리사항을 요청할 수 있는데 만약 이때 해당 프로세스가 멀티 스레드화되어 있다면 각 스레드 별로 주어진 시간을 다시 쪼개어 요구사항을 처리하게 한다.



이러한 방법으로 하나의 CPU를 통해 사용권을 시간에 따라 적절히 스위칭 함으로써 병렬처리가 가능하다. 위 그림처럼 각 스레드는 자신만의 CPU 레지스터 세트를 가지는데, 이를 스레드 컨텍스트라고 부른다. 이러한 컨텍스트는 쓰레드가 마지막으로 수행되었을 당시의 CPU 레지스터 값을 가지고 있다. 이러한 저장과 전환을 문맥전환(context switching)이라 한다. 아래에서 더 언급한다.

## thread-based programming

스레드 객체를 하나 생성해서 그 객체가 어떠한 함수를 실행한다. 아래처럼 사용한다.

std::thread t([]{ std::cout << "something do.\n"; });

t.join();

std::thread객체를 생성하여 그 객체가 함수를 실행하게 한다.

## task-based programming

std::async에게 어떠한 함수(task)를 넘겨준다. 그러면 std::async가 알아서 처리한다.

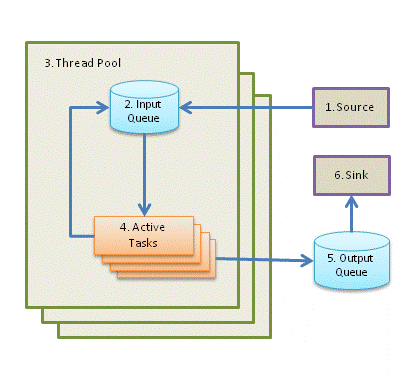
auto ret = std::async([]{ std::cout << "something do.\n"; });

ret.get();

이는 함수를 std::async에 넘겨준다. 사용자는 어떻게 실행되는지 모르지만 std::async에게 맡겨놓는 것이다.

### task-based programming의 의미

**과제 기반 접근법**은 추상적인 측면에서 수행할 작업을 동적으로 생성하는 소프트웨어 엔지니어링의 특정 전략을 말합니다. 이러한 태스크는 태스크를 수행 할 수 있는 스레드에 태스크를 할당하는 **태스크 매니저**에 의해 선택됩니다. 이것은 소프트웨어 아키텍처와 관련이 있습니다. 여기서 이점은 각각의 태스크를 하나씩 실행하는 큰 기능이나 프로그램을 작성하는 것이 아니라, **전체 프로그램의 실행이 작업의 연속**(작업 A가 완료 됨 -> 작업 B와 작업 C 모두 완료된 경우 -> 작업 D를 트리거하는 경우 등)이라는 것입니다. 이렇게 하면 어느 작업이 다른 작업보다 시간이 오래 걸릴 것인지, 작업이 느슨하게 결합 될 때 명확하지 않을 때 유연성이 제공됩니다. 이것은 일반적으로 **스레드 풀**(작업 할당 대기중인 스레드)과 데이터 및 작업 "계약"통신을 위한 일부 메시지 전달 인터페이스 (MPI)로 구현됩니다.



### std::async의 사용

The template function ***async*** runs the function *f* asynchronously (potentially in a separate thread which may be part of a thread pool) and returns a *std::future* that will eventually hold the result of that function call. – cppreference.com

template <typename T>

int parallel\_sum(T beg, T end)

{

auto len = end - beg;

if (len < 1000) return std::accumulate(beg, end, 0);

T mid = beg + len / 2;

//비동기 계산1

auto handle = std::async(std::launch::async, parallel\_sum<T>, mid, end);

//본체 계산2

int sum = parallel\_sum(beg, mid);

//계산1 + 계산2

return sum + handle.get();

}

int main()

{

std::vector<int> v(10000, 1);

std::cout << "The sum is " << parallel\_sum(v.begin(), v.end()) << '\n';

}

특별한 것은 없다. 비동기로 작동하며 반환할 수 있다는 것이다. 자세한 동작은 뒤에서 다룬다.

# 과제 기반 프로그래밍이 우월하다.

가장 근본적인 생각은 높은 수준의 추상을 체현한다는 것이다. 테스크 매니저에 의해 관리되는 스레드들이기 때문에 세부적인 부분에 신경을 덜 쓸 수 있으며, 사용하기에 좀 더 직관적이라는 의미로 볼 수 있겠다.

**과제 기반이 해결해주는 세부적인 부분이 무엇이 있는지** 아래에서 제시될 것이다.

## 반환 값

과제 기반은 어떠한 함수를 비동기적으로 실행할 때, 이의 반환 값에 쉽게 접근할 수 있다. 미래 객체(future)에 get()함수로 반환 값을 담아둘 수 있다.

## 예외 처리

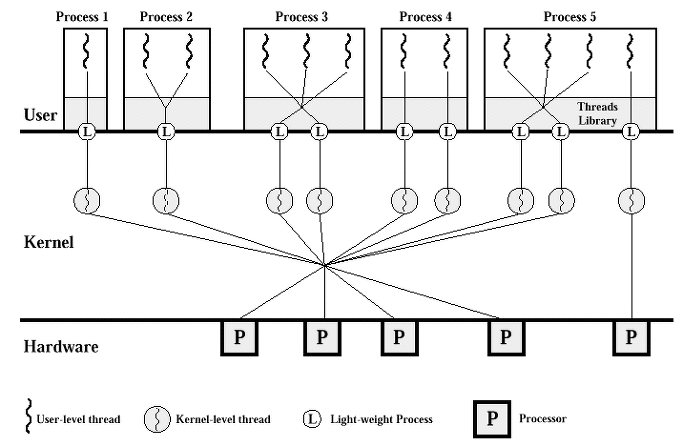
과제 기반은 예외가 발생하면, 그 예외에 역시 접근할 수 있다. 반면, 스레드 기반은 예외가 발생하면 프로그램이 죽어버린다.

## 소프트웨어 스레드가 너무 적거나 많은 상황

### 스레드의 의미 구분

스레드는 3가지 의미의 용어로 사용될 수 있다. 그것은 하드웨어, 소프트웨어, 스레드에 대한 핸들이 그것이다. 간단하게 살펴보자.

#### 소프트웨어 스레드와 하드웨어 스레드



*하드웨어 스레드:* 실제 계산을 수행하는 스레드

*소프트웨어 스레드:* 실행되는 모든 프로세서와 일정을 관리하는데 사용되는 스레드

소프트웨어 스레드가 하드웨어 스레드보다 많다면 이득이 될 수 있다. 하나의 프로세서를 여러 개의 작은 단위(스레드)로 나누어 처리 가능한 스레드부터 처리하면 지체 없이 진행되기 때문이다.

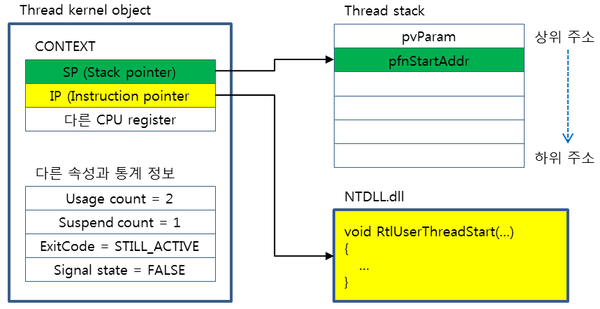
참고: <http://cissnei.tistory.com/category/?page=3>

<https://alitarhini.wordpress.com/tag/multi-core-processor/>

#### 바탕 소프트웨어 스레드에 대한 핸들(std::thread)

위에서도 언급하듯이 스레드는 실행 단위지만 우리가 객체로 생성하여 사용하는 std::thread와는 의미가 다르다. 스레드는 우리가 만드는 것이 아니다. 핸들을 만드는 것이다. 그 안에 할 일이 있으면 운영체제가 스레드를 배정해준다. 그러므로 std::thread객체 안에는 null값일 수도 있는 것이다. 기본 생성자, join으로의 합류, detach을 통한 탈착, 이동 연산이 그렇다.

일반적으로 스레드가 생성된 모습은 아래와 같이 볼 수 있다. (CreateThread의 경우)

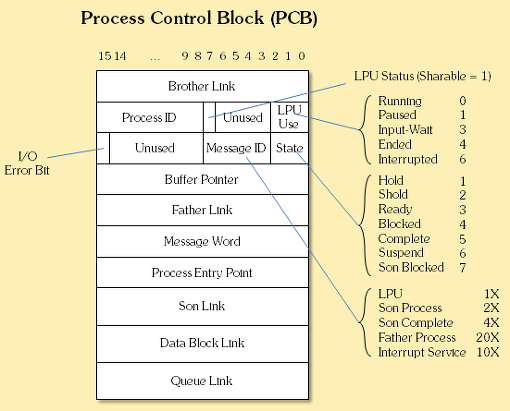


1. 시스템은 **스레드 커널 오브젝트를 생성**한다. 이 오브젝트는 **초기 usage count 2**를 가진다. (생성한 프로세스 1, 쓰레드 자신 1)  
2. 시스템은 **스레드 스택**으로 활용할 메모리 공간을 할당한다.  
3. 각 스레드는 자신만의 CPU 레지스터 세트를 가지는데, 이를 **스레드 컨텍스트**라고 부른다. 이러한 컨텍스트는 스레드가 마지막으로 수행되었을 당시의 CPU 레지스터 값을 가지고 있다. 쓰레드의 CPU 레지스터 세트는 **스레드 커널 오브젝트 내에 저장**된다.

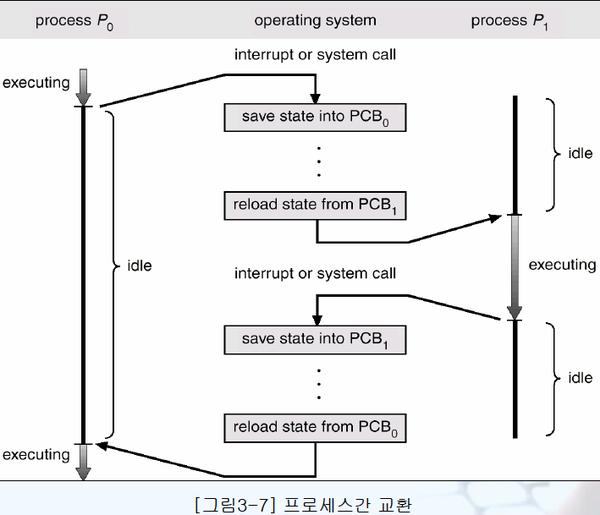
#### 문맥 전환(context switching)

*문맥 전환*이란 이전의 프로세스의 상태(문맥)를 보관하고 새로운 프로세스의 상태를 적재하는 작업을 말한다.

한 프로세스의 상태는 *프로세서 제어 블록(PCB)*에 저장되어 있다. 아래 그림처럼 나타난다.



상당히 내용이 많다. 이를 옮겨 담으려면 생각보다 시간이 오래 걸릴지도 모른다. **문맥 전환이란, 이러한 자료가 옮겨지는데 생기는 오버헤드**인 것이다.



실행중인 프로세스 (Pj)의 문맥전환  
Pj 제어 : 사용자 모드 -> 운영체제 모드 -> Pj의 현제 하드웨어 상태 PCB에 저장  
Pk 레지스터 내용 <- PCBk : CPU에 재저장하고 Pk를 실행

**문맥전환의 단점**1. 오버헤드로 인한 관리 부담이 증가한다.  
2. CPU cache에 적중률을 떨어뜨린다.

### [문제1] 지나치게 많은 스레드: 과다구독(oversubscription)

<https://andromedarabbit.net/wp/>

소프트웨어 스레드는 시스템이 제공할 수 있는 양이 제한되어 있다. 만약 시스템 수용량보다 더 요구된다면, std::system\_error예외가 발생한다.

int doWorkHard(int k) noexcept;

std::thread t(doWorkHard, 9999);

즉, 위와 같은 식도 noexcept지만 예외가 발생할 수 있는 위험에 도사린다.

하드웨어 스레드보다 소프트웨어 스레드가 많을 때, 운영체제는 보통 **라운드 로빈 스케줄링(우선 순위가 같은 작업끼리 작은 시간만 수행하고 넘기는 방법)**을 이용한다. 스케줄러는 소프트웨어 스레드마다 타임 슬라이스(time slice)를 부여하여 하드웨어 스레드 중 하나에서 실행될 수 있게 한다.   
하지만 이렇게 하드웨어 스레드를 공정하게 분배하려면 오버헤드가 걸린다. 소프트웨어 스레드가 너무 많으면 오버헤드 때문에 성능이 심각하게 저하될 수 있다.

#### 오버헤드의 종류

*스레드 레지스터 상태*

가장 명백한 오베헤드는 스레드의 레지스터 상태를 저장하고 복구하는 과정이다. 이것이 문맥교환으로 인한 오버헤드이다. 하지만 일반적으로, 스레드 스케줄러는 타임 슬라이스를 상당히 크게 잡기 때문에 레지스터를 저장하고 복구하는 오버헤드는 미미하다.

*스레드 캐시 상태*

이보다 미묘한 문제는 스레드의 캐시 상태를 저장하고 복구하는 오버헤드이다. 현대 프로세서는 캐시 메모리에 상당히 의존한다.   
캐시가 꽉 찼을 땐, 프로세서가 새 데이터가 들어갈 자리를 마련하기 위해 캐시에서 데이터를 제거해야 한다. 보통 제거할 데이터를 고를 때 최근에 사용되지 않은 데이터(Least Recently Used, LRU)를 고른다. 이렇게 하면 대개 이전 타임 슬라이스의 데이터가 선택된다. 그렇기 때문에 스레드는 서로 상대방의 데이터를 제거하곤 한다. **정리하자면 스레드가 너무 많으면 서로 캐시를 차지하려고 다투기 때문에 성능이 저하된다.**

*가상 메모리 스레싱*

비슷하긴 하지만 그 수준은 다른 오버헤드가 바로 가상 메모리 스레싱이다. 대부분의 시스템은 가상 메모리를 사용하는데, 프로세서는 실제 사용 가능한 주소보다 큰 주소 공간을 가진다. 가상 메모리는 디스크에 위치하며, 자주 사용되는 부분은 실제 메모리에 있다. 이 과정은 캐시와 비슷하다.  
소프트웨어 스레드 각각은 스택과 데이터 구조를 유지하려고 가상 메모리를 사용한다. 캐시와 마찬가지로 타임 슬라이스는 실제 메모리를 두고 소프트웨어 스레드끼리 경쟁하게 만들며 이는 성능을 떨어뜨린다.

*Convoying*

여태 설명한 캐시와 가상 메모리 이슈는 너무 많은 소프트웨어 스레드가 제한된 자원을 공유하기 때문에 발생한다. 이와는 다르고, 이보다 심각할 때가 많은 문제도 있는데 convoying이라고 한다. Convoying은 잠금 하나를 두고 여러 소프트웨어 스레드가 산더미처럼 쌓이는 현상이다.  
스레드가 잠금을 쥔 상태에서 해당 스레드의 타임 슬라이스가 만료되었다고 해보자. 그 잠금을 기다리던 스레드 모두는 그 스레드가 일어나 잠금을 풀기만을 기다려야 한다. 먼저 잠금을 획득한 순서대로 처리하는 구현이라면 문제는 심각해진다. 대기 중인 스레드가 잠시 중단되면 그 뒤를 이어 기다리는 스레드는 어느 것도 잠금을 획득하지 못하게 된다.

#### 지나치게 많은 스레드가 유발하는 문제를 정리하면 다음과 같다.

첫째, 정해진 분량의 일을 여러 스레드에 분배하면 스레드 하나하나는 할 일이 별로 없게 된다.   
둘째, 소프트웨어 스레드가 너무 많으면 한정된 하드웨어 자원을 공유하는데 드는 비용이 커진다.

#### 어떻게 해결하는가?

**보통 가장 좋은 해결책은 ’실행 가능한(Runnable)’ 스레드의 개수를 하드웨어 스레드의 개수에, 가능하다면 외부 캐시(Outer-level cache)의 개수에 맞추어 제한하는 것이다. 자세한 내용은 위 링크 참고**

**하지만 단순하지 않다. 적절한 스레드의 개수는 동적으로 변할 수 있기 때문이다.**

### [문제2] 소프트웨어 스레드가 부족한 상황

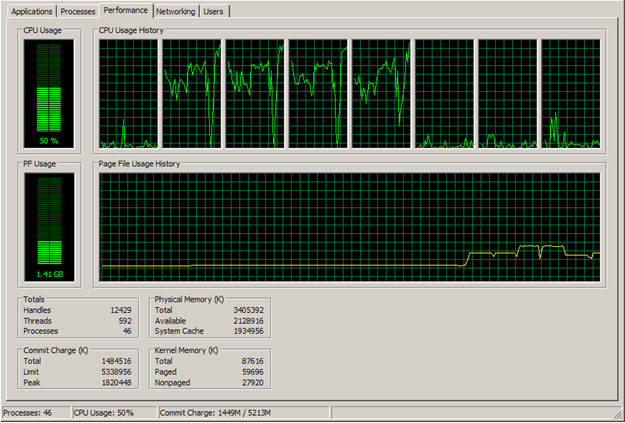
그냥 현재 스레드에서 어떠한 함수를 실행하면 된다. 하지만 이는 현재 스레드에 부하가 과중하게 걸릴 수 있다. 이 스레드가 GUI와 같은 반응성에 민감하다면 문제가 된다.   
또 다른 방법은 일부 소프트웨어 스레드가 완료되길 기다렸다가 실행하는 것인데, 기존 스레드가 특수한 동작(특정한 결과를 산출하거나, 조건 변수를 통지하는 등)을 기다리고 있을 수도 있으므로 해결하기가 쉽지 않아 보인다.

### [해결책] std::async

이러한 문제들에 상당량을 std::async에게 떠넘길 수 있다. 근본적 원리는 std::async를 관리하는 관리자가 이러한 문제들을 어느 정도 제어한다는 점이다. 이는 스레드를 생성하지 않고 대신 함수 결과가 필요한 스레드에서 실행하라고 요청할 수 있다. 이렇게 과다구독이나 부족한 상황에서 어느 정도 유연하게 대처할 수 있는 것이다.

## 부하 불균형

어느 한쪽의 스레드만 작업을 많이 하는 현상이다. 이는 단순히 스레드 관리를 std::async에 맡긴다고 해서 해결되지 않는다. 하지만 실행 스케줄러가 우리보다는 많이 알고 있을 것이다. 그러므로 std::async에 부하 불균형 관리를 맡기는 것이 괜찮다.

****

## 특정 스레드의 반응성

위에서 직접적으로 언급한 것처럼, GUI스레드처럼 특별히 반응성이 좋아야 하는 스레드는 특별한 처리를 원한다. **이에 대해 시동 방침(launch policy)을 std::async에 넘겨주는 것이 바람직하다.**

# 스레드 기반 프로그래밍을 사용하는 경우

한마디로 과제 기반 프로그래밍이 편하다는 말이다. 반면 스레드 기반 프로그래밍은 위와 같은 문제를 직접 다루어야 한다는 의미가 된다. 하지만 하부적인 내용을 직접 건드려야 한다면 스레드 기반 프로그래밍을 사용해야 할 것이다.

## 바탕 스레드 적용 라이브러리의 API에 접근하는 경우

저수준 플랫폼 고유 API를 이용해서 구현해야 하는 경우다. pthread나 windows라이브러리에는 훨씬 더 많은 기능을 사용하게 해주기에 이것을 사용한다는 의미다. std::thread에서 흔히 native\_handle라는 멤버 함수를 이용하여 접근할 수 있다. 반면 std::async에는 이런 기능이 없다.

### std::thread::native\_handle

std::thread t;

auto handler = t.native\_handle(); //type: void\*

Returns the implementation defined underlying thread handle.

#include <thread>

#include <mutex>

#include <iostream>

#include <chrono>

#include <cstring>

#include <pthread.h>

[std::mutex](http://en.cppreference.com/w/cpp/thread/mutex) iomutex;

void f(int num)

{

[std::this\_thread::sleep\_for](http://en.cppreference.com/w/cpp/thread/sleep_for)([std::chrono::seconds](http://en.cppreference.com/w/cpp/chrono/duration)(1));

sched\_param sch;

int policy;

pthread\_getschedparam(pthread\_self(), &policy, &sch);

[std::lock\_guard](http://en.cppreference.com/w/cpp/thread/lock_guard)<[std::mutex](http://en.cppreference.com/w/cpp/thread/mutex)> lk(iomutex);

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "Thread " << num << " is executing at priority "

<< sch.sched\_priority << '**\n**';

}

int main()

{

[std::thread](http://en.cppreference.com/w/cpp/thread/thread) t1(f, 1), t2(f, 2);

sched\_param sch;

int policy;

pthread\_getschedparam(t1.native\_handle(), &policy, &sch);

sch.sched\_priority = 20;

if (pthread\_setschedparam(t1.native\_handle(), SCHED\_FIFO, &sch)) {

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "Failed to setschedparam: " << [std::strerror](http://en.cppreference.com/w/cpp/string/byte/strerror)(errno) << '**\n**';

}

t1.join(); t2.join();

}

## 응용 프로그램의 스레드 사용량을 최적화해야 할 수 있어야 하는 경우

하드웨어 특성들이 미리 정해진 컴퓨터에서 유일하게 의미있는 프로세스로 실행될 서버 소프트웨어를 개발한다면 이러한 요구가 생길 수 있다.

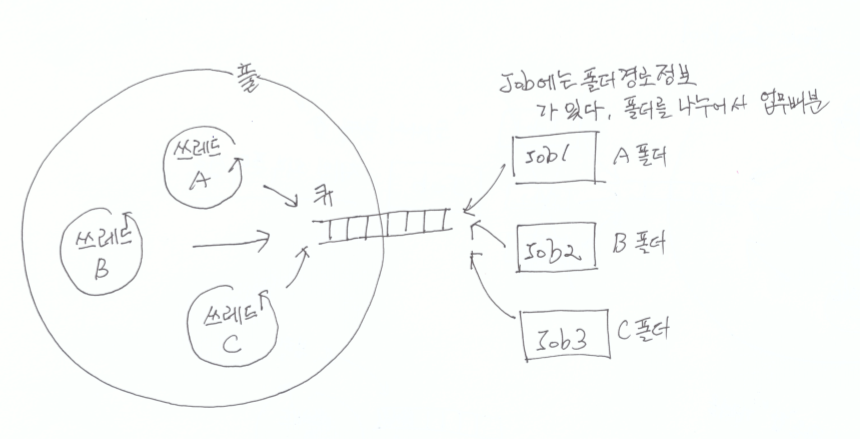
## C++동시성 API가 제공하는 것 이상의 기술을 구현해야 하는 경우

스레드 풀을 구현하지 않는 특정한 플랫폼을 위해 그에 맞는 스레드 풀을 직접 구현해야 할 수 있다.

### 스레드 풀 원리

출처: <http://okky.kr/article/345720>

생성 / 수거에 드는 비용을 줄이고 재사용성을 높이기 위해 풀장에다가 여러 개의 스레드를 먼저 만들어두고 사용하기도 해요. 큐 같은 자료구조를 가진 클래스에 쓰레드를 미리 몇 개 만들어서 놓아두면 그게 스레드 풀이에요.



1. 폴더 3개를 각각 잡으로 만들어서 스레드 풀에 줍니다.   
2. 스레드 풀은 각 잡을 내부의 스레드에 배분 해줍니다.  
3. 스레드들은 배분 받은 일을 합니다.

작업을 분배하는 방법에는 여러 가지가 있다. 해당 작업에 따라 그 구현은 달라질 것이다.

### 스레드 풀 구현

개락적인 과정만 명세한다. 내부 소스 참고

class ThreadPool {

public:

ThreadPool(size\_t);

template<class F, class... Args>

auto enqueue(F&& f, Args&&... args)

->std::future<typename std::result\_of<F(Args...)>::type>;

~ThreadPool();

private:

// need to keep track of threads so we can join them

std::vector< std::thread > workers;

// the task queue

std::queue< std::function<void()> > tasks;

// synchronization

std::mutex queue\_mutex;

std::condition\_variable condition;

bool stop;

};

// the constructor just launches some amount of workers

inline ThreadPool::ThreadPool(size\_t threads)

: stop(false)

{

//초기 스레드를 생성한다.

//생성된 스레드에 작업을 할당한다.

std::function<void()> task;

//동기화시켜 작업들을 큐에 넣는다.

std::unique\_lock<std::mutex> lock(this->queue\_mutex);

//락이 풀리기를 기다린다. 작업이 들어올때까지 기다리는 것이다.

this->condition.wait(……)

//스레드 풀이 종료되면 종료한다.

//큐에서 꺼내온 작업을 할당한다.

//작업을 수행한다.

}

//해당 작업을 스레드 풀에 할당하여 그 작업의 결과값을 반환한다.

template<class F, class... Args>

auto ThreadPool::enqueue(F&& f, Args&&... args)

{

using return\_type = typename std::result\_of<F(Args...)>::type;

//주어진 함수에 인자를 집어넣어 bind한 task를 만든다.

auto task = std::make\_shared<packaged\_task<return\_type()>>();

//만든 task의 반환 값을 기다린다.

std::future<return\_type> res = task->get\_future();

{

std::unique\_lock<std::mutex> lock(queue\_mutex);

//만든 task는 큐에 집어넣는다.

tasks.emplace([task]() { (\*task)(); });

}

//기다리는 스레드에게 작업이 부여되었음을 알린다.

condition.notify\_one();

//task의 반환값을 반환한다.

return res;

}

// the destructor joins all threads

inline ThreadPool::~ThreadPool()

{

//모든 스레드에게 종료를 알린다.

}

int main()

{

ThreadPool pool(4);

std::vector< std::future<int> > results;

for (int i = 0; i < 8; ++i) {

results.emplace\_back(할당할 작업);

}

for (auto && result : results)

std::cout << result.get() << ' ';

std::cout << std::endl;

return 0;

}

***std::thread API에서는 비동기적으로 실행된 함수의 반환 값을 직접 얻을 수 없으며, 만일 그런 함수가 예외를 던지면 프로그램이 종료된다.***

***스레드 기반 프로그래밍에서는 스레드 고갈, 과다구독, 부하균형화, 새 플랫폼으로의 적응을 독자가 직접 처리해야 한다.***

***std::async와 기본 시동 방침을 이용한 과제 기반 프로그래밍은 그런 대부분의 문제를 알아서 처리해준다.***

[항목36] 시동 방침: std::launch

비동기성이 필수일 때에는 std::launch::async를 지정하라.

# launch policy(시동 방침)

이는 std::async(f)를 호출할 때, 어떠한 방식으로 f를 호출할 지 지정한다.

## std::launch 정의

**enum** **class** **launch**{ async, deferred, any = async | deferred };

|  |  |  |
| --- | --- | --- |
| **launch label** | **int value** | **description** |
| [launch::async](http://www.cplusplus.com/launch) | *unspecified* | **Asynchronous:** The function is called asynchronously by a new thread and synchronizes its return with the point of access to the *shared state*. |
| [launch::deferred](http://www.cplusplus.com/launch) | *unspecified* | **Deferred:** The function is called at the point of access to the *shared state*. |

### std::launch::async

시동 방침을 지정하면 f는 반드시 **비동기적**으로, 다시 말해서 **다른 스레드로 떨어져서 실행**된다.

### std::launch::deferred

시동 방침을 지정하면 f는 std::async가 **get이나 wait이 호출될 때까지 f실행이 지연된다.** 이는 get, wait을 호출한 스레드에서 실행된다(**동기적**으로 실행). 그러므로 get, wait이 호출되지 않으면 f는 실행되지 않는다.

그래서 다음과 같은 현상을 확인할 수 있다.

auto fut1 = std::async(std::launch::async, []

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(3000));

return 3;

});

auto fut2 = std::async(std::launch::deferred, []

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(3000));

return 7;

});

std::cout << "start get fut1\n";

//fut1은 스레드의 시작과 동시에 작업을 수행합니다.

std::cout << fut1.get() << '\n';

std::this\_thread::sleep\_for(std::chrono::milliseconds(5000));

std::cout << "start get fut2\n";

//fut2는 이제서야 값을 계산하기 시작합니다.

std::cout << fut2.get() << '\n';

### 기본 시동 방침(default)

기본 시동 방침은 async | deferred가 된다. 이는 동기적으로도 실행될 수 있으며, 비동기적으로도 실행된다는 것을 의미한다. 이는 유연성을 갖게 하는데, [항목35]에서 언급된 스레드 생성과 파괴, 과다구독 회피, 부하 균형화의 책임 등이 이러한 유연성 덕분이다.

std::async는 테스크 매니저에 의해 작업이 할당되며 전체 작업을 연속적으로 표현할 수 있다. 그러므로 임의의 스레드가 어떤 동작을 가져야 하는지 판단할 수 있는데, 그 판단은 기본 시동 방침은 async와 deferred의 경우 모두에 대해 적용한다.

# 기본 시동 방침의 사용

위에서처럼, 기본 시동 방침은 동기적과 비동기적 사이의 유연성을 가진다.

## 기본 시동 방침의 영향

위의 유연성은 아래와 같은 영향을 준다. 아래 문장이 스레드 t에서 실행된다고 가정하자.

auto fut = std::async(f);

### f가 지연될 수 있으므로, f가 t와 동시에 실행될지 예측하는 것은 불가능하다.

deferred의 시동 방침이라면 t가 실행되더라도 f는 기다릴 것이다.

### f가 fut에 대해 get이나 wait를 호출하는 스레드와는 다른 스레드에서 실행될지 예측하는 것은 불가능하다.

t가 get이나 wait을 호출하는 스레드라고 할 때, f가 t와는 다른 스레드에서 실행될지는 예측할 수 없다. 비동기라면 다른 스레드에서, 지연이라면 같은 스레드에서 실행될 테니 말이다.

### f가 반드시 실행될 것인지 예측하는 것이 불가능하다.

프로그램의 모든 가능한 경로에서 fut에 대한 get이나 wait호출이 일어난다는 보장이 없을 수도 있기 때문이다.

## 기본 시동 방침의 유연성이 주는 문제

위의 영향으로 인하여 이 모든 경우에 적합하지 않을 수도 있다.

### 스레드 지역 저장소(Thread Local Storage, TLS)

***저장소 클래스 지정자(Storage class specifiers)***

저장소 클래스 지정자는 **이름의 범위와 함께** 이름의 두 가지 독립 속성인 **저장 기간**과 **연결**을 제어합니다. static, extern, thread\_local이 있다.

thread\_local unsigned int rage = 1;

std::mutex cout\_mutex;

void increase\_rage(const std::string& thread\_name)

{

++rage; // modifying outside a lock is okay; this is a thread-local variable

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for " << thread\_name << ": " << rage << '\n';

}

int main()

{

std::thread a(increase\_rage, "a"), b(increase\_rage, "b");

{

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for main: " << rage << '\n';

}

a.join();

b.join();

}

**static혹은 global 메모리를 쓰레드에 지역적으로 쓸 수 있게 하는 프로그래밍 메소드**다. 즉, 정적 혹은 전역에 선언한 변수가 스레드의 지역 변수처럼 사용된다는 의미다.

이러한 TLS를 읽거나 쓰는 코드는 유연성과 맞지 않다. 왜냐하면 그 코드가 어떤 스레드의 지역 변수에 접근할 지 예측할 수 없기 때문이다. 즉, f의 TLS가 독립적인 스레드의 것일 수도 있으며, fut에 대해 get이나 wait을 호출하는 스레드의 것일 수도 있다. 이름이 같은 변수지만 사용되는 스레드가 다르면 완전히 다른 변수이므로 다른 연산이 되어버린다.

### 만료 기간이 있는 wait기반 루프

기본 시동 방침은 비동기적 혹은 지연되는데, 비동기적이라면 문제가 없는 경우다. 하지만 지연된다면, std::async는 wait에 대해서 항상 std::future\_status::deferred를 반환한다. 왜냐하면 결과값이 필요할 때까지 지연하기 때문이다.

auto f = []()

{

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout << "async func.\n";

};

int main()

{

auto fut = std::async(f);

while (fut.wait\_for(std::chrono::milliseconds(100))

!= std::future\_status::ready)

{

std::cout << "loop\n";

}

return 0;

}

결국, 시동 방침이 불분명하여 생길 수 있는 문제이다.

이를 아래와 같이 해결할 수 있다. fut의 상태를 고려해야 한다.

int main()

{

auto fut = std::async(f);

if (fut.wait\_for(std::chrono::seconds(0)) == std::future\_status::deferred)

{

}

else

{

while (fut.wait\_for(std::chrono::milliseconds(100))

!= std::future\_status::ready)

{

std::cout << "loop\n";

}

}

return 0;

}

## 기본 시동 방침이 적합한 경우

1. 과제가 get이나 wait를 호출하는 스레드와 반드시 동시적으로 실행되어야 하는 것이 아닌 경우  
2. 여러 스레드 중 어떤 스레드의 thread\_local변수들을 읽고 쓰는지가 중요하지 않은 경우  
3. std::async가 돌려준 미래 객체에 대해 get이나 wait가 반드시 호출된다는 보장이 있거나, 과제가 전혀 실행되지 않아도 괜찮은 경우  
4. 과제가 지연된 상태일 수도 있다는 점이 wait\_for이나 wait\_until을 사용하는 코드에 반영되어 있는 경우

## 기본 시동 방침이 적합하지 않다면: 비동기적 실행 강제

위의 기본 시동 방침이 적합하지 않은 경우, 강제로 비동기적 실행을 할 필요가 있다.

auto fut2 = std::async(std::launch::async, f);

### std::launch::async시동 방침을 명시적으로 지정하지 않는 법

std::async를 이렇게 강제하되, async를 호출할 때마다 시동 방침을 명시하지 않아도 된다면 편리할 것이다. 다음과 같이 할 수 있다.

template<typename F, typename... Ts>

//inline std::future<typename std::result\_of<F(Ts...)>::type> //C++11

auto reallyAsync(F&& f, Ts&&... params) //C++14

{

return std::async(

std::launch::async,

std::forward<F>(f),

std::forward<Ts>(params)...);

}

함수와 인자들을 완벽 전달하여 캡슐화한 것이다. 어떻게 했는지 기억할 필요가 있다.

#### std::result\_of

template< class >

class result\_of; // not defined

template< class F, class... ArgTypes >

class result\_of<F(ArgTypes...)>;

컴파일 타임에 반환 형식을 유추합니다. F에 ArgTypes를 대입한 결과 값의 타입을 유추합니다.

struct C { double Func(char, int&); };

std::result\_of\_t<decltype(&C::Func)(C, char, int&)> g = 3.14;

static\_assert(std::is\_same<decltype(g), double>::value, "");

2번째 줄에서 타입 연역 과정은 이렇습니다.  
1. decltype(&C::Func)은 멤버 함수 타입입니다.  
2. (C, char, int&)은 해당 함수를 갖고 있는 객체의 타입과 좌측 멤버 함수에 들어갈 인자 타입입니다.   
3. 이 함수의 연산 반환 값의 타입을 연역합니다.

***std::async의 기본 시동 방침은 과제의 비동기적 실행과 동기적 실행을 모두 허용한다.***

***그러나 이러한 유연성 때문에 thread\_local접근의 불확실성이 발생하고, 과제가 절대로 실행되지 않을 수도 있고, 시간 만료 기반 wait호출에 대한 프로그램 논리에도 영향을 미친다.***

***과제를 반드시 비동기적으로 실행해야 한다면 std::launch::async를 지정하라.***

[항목37] ThreadRAII만들기

std::thread들은 모든 경로에서 합류 불가능하게 만들어라.

# 합류가능, 불가능이란?

모든 std::thread는 합류 가능(joinable)상태이거나 합류 불가능(unjoinable)상태를 가진다.

### 합류 가능 상태의 std::thread

일단 합류한다는 것은 무엇인가? 코드를 보면서 이해해보자.

int main()

{

std::thread t([]{ std::cout << "thread start.\n"; });

if (t.joinable())

{

std::cout << "thread join.\n";

t.join();

}

std::cout << "main routine end.\n";

return 0;

}

합류가 가능한지 불가능한지는 t.joinable()로 확인한다. 위 식에서 t는 정상적으로 합류될 수 있으며 바로 t.join()이 합류한다는 기능을 수행한다.

#### std::thread::join

이 함수는 스레드 실행이 완료되면 반환됩니다.

이 함수는 스레드에서 모든 작업이 완료 될 때, 이 함수가 반환하는 순간을 동기화합니다. 이 함수는 생성시 호출 된 함수가 반환 될 때까지 이 함수를 호출하는 스레드의 실행을 차단합니다 (아직 수행하지 않은 경우). 이 함수를 호출하면 스레드 객체는 결합 할 수 없게 되며 안전하게 파괴 될 수 있습니다. - <http://www.cplusplus.com/>

즉, 합류란 해당 루틴에서 그 스레드가 끝나기를 기다리는 것이다. 위 코드에서는 main routine에서 t스레드가 끝날 때까지 기다린다고 해석할 수 있다.

#### 어떤 thread가 joinable한가?

어떤 스레드가 합류할 수 있는가? 의미상으로 본다면 실행 중이거나 실행 상태로 전이할 수 있는 스레드를 말한다. 쉽게 말하자면, 살아있는 스레드다.

### 합류 불가능 상태의 std::thread

합류할 수 없는 스레드. 죽은 스레드다.  
1. 기본 생성된 std::thread  
2. 다른 std::thread객체로 이동된 후의 std::thread객체  
3. join에 의해 합류된 std::thread  
4. detach에 의해 탈착된 std::thread

4가지의 공통점은 이미 끝나거나 떨어진 스레드, 죽은 스레드라는 것이다. 예제에서 보자.

int main()

{

std::thread t1; //빈 스레드: 합류 불가능

std::thread t([]{ std::cout << "thread start.\n"; });

if (t.joinable())

{

std::cout << "thread join.\n";

t.join();

}

if (t.joinable()) std::cout << "no accecpt.\n"; //join이후: 합류 불가능

std::thread t2([]{ std::cout << "thread start.\n"; });

std::thread temp = std::move(t2);

if (t2.joinable()) //이동된 스레드(t2): 합류 불가능

{

std::cout << "impossible.\n";

t2.join();

}

if (temp.joinable()) //temp는 합류 가능

{

std::cout << "possible.\n";

temp.join();

}

std::thread([]{ while (true); }).detach(); //탈착 스레드: 합류 불가능

std::cout << "main routine end.\n";

return 0;

}

# 합류 가능 여부가 중요한 이유

합류 가능한 스레드의 소멸자가 호출되면 프로그램이 종료된다.

이는 간단하게 확인할 수 있다. main스레드에서 호출한 스레드가 끝나기 전에 main루틴이 끝나면 프로그램이 죽는다. 이는 아직 합류 가능한 스레드가 남아있는데 루틴이 죽으면서 그 스레드의 소멸자가 호출된 것이다.

### 어떤 문제가 생길까?

다음과 같은 문제가 생긴다. 많은 예시 중 하나일 뿐이다.

const auto bigbig = 1000000;

bool conditionAreSatisfied(){ return true; }

bool doWork(std::function<bool(int)> filter, int maxVal = bigbig)

{

std::vector<int> goodVals;

std::thread t([&filter, maxVal, &goodVals]

{

for (auto i = 0; i <= maxVal; i++)

if (filter(i)) goodVals.push\_back(i);

});

SetThreadPriority(t.native\_handle(), THREAD\_PRIORITY\_HIGHEST);

if (conditionAreSatisfied())

{

t.join();

std::cout << "do work.\n";

return true;

}

return false;

}

위의 코드에서 어떤 문제가 있을까? 간단하다. conditionAreSatisfied() 가 false를 반환하면, t가 끝나기 전에 소멸자가 호출되어 프로그램은 죽어버릴 것이다.

#### 스레드 우선순위 정하기

그 외에 native\_handler로 하부적인 API를 사용하려면 window.h를 참고한다.

<https://msdn.microsoft.com/en-us/library/windows/desktop/ms686277(v=vs.85).aspx>

### 왜 이렇게 작동할까?

다른 방법은 더 나쁘기 때문이다.

**암묵적 join:** 성능적으로 문제가 생긴다. 위의 코드로 예를 들면, 어떠한 조건이 만족하지 못하여 conditionAreSatisfied()가 false를 반환했는데 계속해서 계산을 기다리고 있다면 난감하다.

**암묵적 detach:** 좋지 않은 기이한 현상이 생길 수 있다. 그 기이한 현상을 간단히 설명하자면, 암묵적 detach일 경우, 스레드t가 참조하는 goolVals와 같은 지역 변수들은 해당 루틴이 끝났어도 계속 진행할 것이다. 루틴이 끝났는데 지역 변수를? 무슨 일이 생길지 모른다.

# 합류 불가능 스레드 만들기: RAII

한 범위의 바깥으로 나가는 모든 경로에서 어떤 동작이 반드시 수행되어야 할 때에는 그 객체의 소멸자 안에 그 동작을 넣는 것이다. 이것을 ***RAII(Resource Acquisition Is Initialization) 클래스***라고 부른다.

스마트포인터가 객체 메모리 반환을 소멸자에서 수행하는 것처럼, 사용자가 마무리해야 할 여지를 남기고 싶지 않다면 RAII 객체를 만드는 것이 핵심이다.

## 구현

class ThreadRAII

{

public:

enum class DtorAction { join, detach };

ThreadRAII(std::thread&& \_t, DtorAction \_a)

: action(\_a), t(std::move(\_t)) {}

~ThreadRAII()

{

if (t.joinable())

{

if (action == DtorAction::join) t.join();

else t.detach();

}

}

std::thread& get() { return t; }

private:

DtorAction action;

std::thread t;

};

int main() {

ThreadRAII tr(std::thread(

[]{ for (int i = 0; i < 100; i++) std::cout << "!"; }),

ThreadRAII::DtorAction::join);

//스레드에 대해 뒷처리를 안 해도 되는 아름다운 코드이다.

return 0;

}

추가로 응용하여 시작 시간을 조절하거나 스스로 잠그는 등 여러 동작을 알아서 할 수 있다.

class threadContainer {

std::thread\* mT;

std::mutex m;

void lockMe() {

// wait for mT to be assigned:

{

std::unique\_lock<std::mutex> lock(m);

}

std::cout << "lockMe():" << mT << "\n";

auto h = mT->native\_handle();//causes a crash

std::cout << "Done lockMe!\n";

}

public:

void run() {

// release lock only after mT assigned:

{

std::unique\_lock<std::mutex> lock(m);

mT = new std::thread([&](){ this->lockMe(); });

}

std::cout << "run():" << mT << "\n"; //00326420

mT->join();

}

};

## 세부사항

ThreadRAII구현에는 다음을 주의해야 한다.

### 생성자는 오른값만 받는다.

std::thread는 복사할 수 없기 때문이다.

### std::thread는 가장 마지막에 초기화

ThreadRAII(std::thread&& \_t, DtorAction \_a)

: action(\_a), t(std::move(\_t)) {}

### get함수 제공

바탕 std::thread에 접근할 수 있는 get함수를 제공한다.

### 소멸자는 합류 가능을 확인해야 한다.

ThreadRAII소멸자는 std::thread의 멤버함수를 호출하기 전에 합류 가능인지 확인해야 한다. 그렇지 않으면 미정의 행동을 유발한다.

### 가로챌 수 있는 스레드(interruptible thread)

위의 RAII가 모든 합류에 관한 해결책은 아니라는 것이다. std::thread소멸 시, join을 실행하게 하면 통신 상호작용으로 인하여 프로그램이 멈출 수도 있다. – [항목39] **근본적인 방법은 어떤 스레드에게 더 이상 일할 필요가 없으니 일찍 반환하라고 알려주는 것이다.** C++ Concurrency in Action을 참고해서 구현하기 바란다.

***모든 경로에서 std::thread를 합류 불가능으로 만들어라.***

***소멸 시 join방식은 디버깅하기 어려운 성능 이상으로 이어질 수 있다.***

***소멸 시 detach방식은 디버깅하기 어려운 미정의 행동으로 이어질 수 있다.***

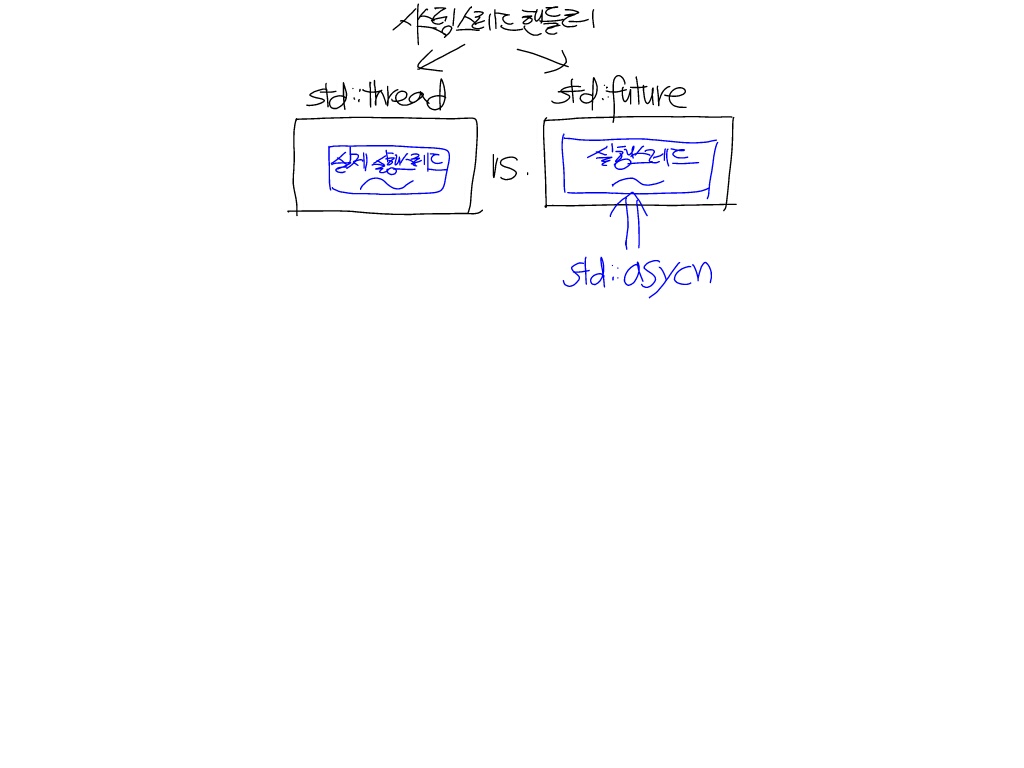
***자료 멤버 목록에서 std::thread객체를 마지막에 선언하라.***

[항목38] 시스템 핸들의 행동 방식

스레드 핸들 소멸자들의 다양한 행동 방식을 주의하라.

# 시스템 스레드 핸들(handle)

어떤 스레드를 담고 이에 접근할 수 있는 객체이다. C++에서는 2가지의 시스템 스레드 핸들러가 있다고 볼 수 있는데, 하나는 std::thread로 만든 객체이며, 또 하나는 std::async로 만든 스레드를 핸들하는 std::future라고 볼 수 있다.



std::thread t(attack, 1);

std::future<int> f = std::async(attack, 1);

핸들(std::thread, std::future)은 새로운 스레드에 접근하고 있다.

## std::thread

실행 스레드를 담을 수 있는 객체 중 하나로써, 만드는 순간 작업이 있다면 실행하게 되며, 실행하는 도중에 소멸자가 호출된다면 프로그램은 종료된다. 이런 경우를 대비하여 온전히 실행될 수 있도록 사용자가 직접 처리해야 한다. (join, datach)

## std::future

std::thread와는 다르게 작동한다. 특히 소멸자에서 다르게 행동한다는 점이 흥미롭다.

### 정의

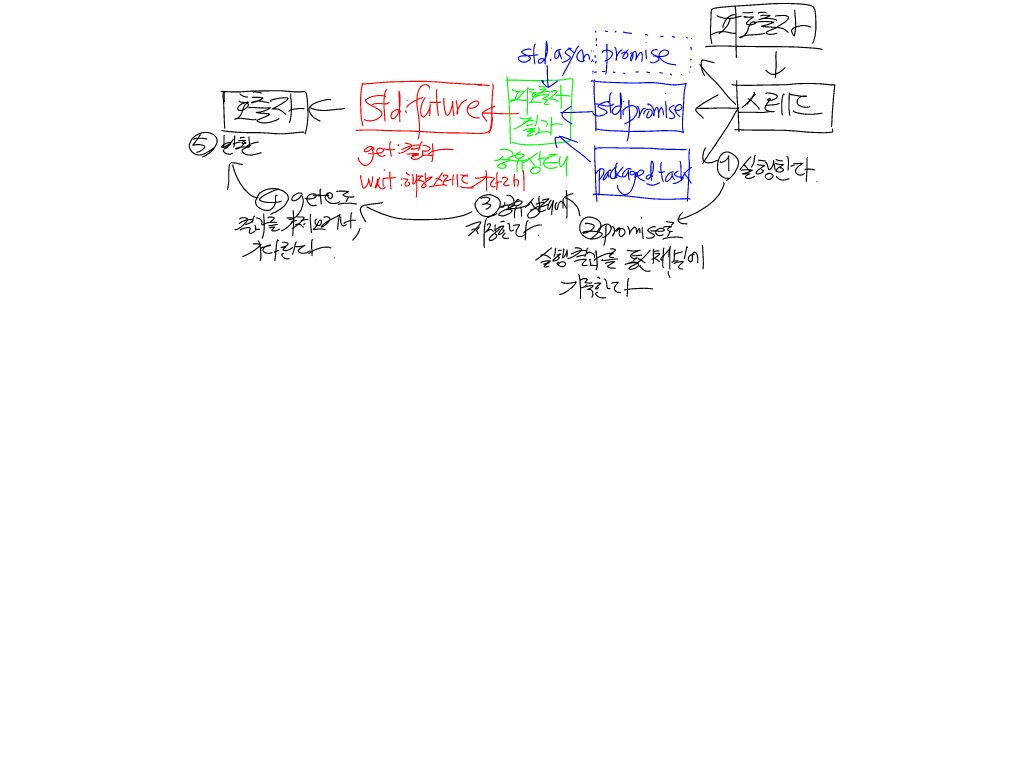
template< class T > class future;

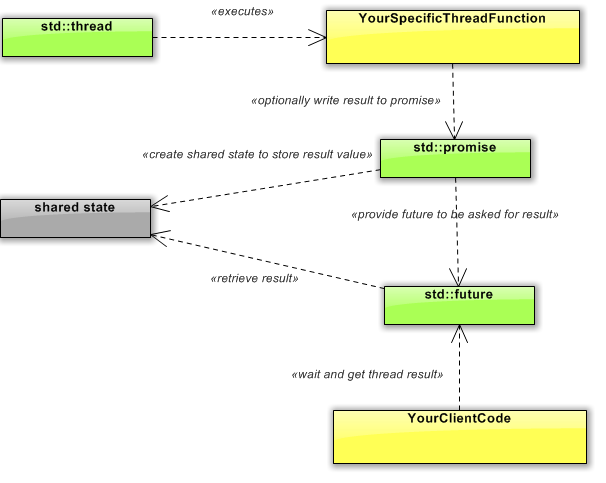
template< class T > class future<T&>;

template<> class future<void>;

**std :: future**는 비동기 작업의 결과에 액세스하는 메커니즘을 제공합니다. – cppreference.com

추상적 개념: 피호출자가 결과를 호출자에게 전송하는 통신 채널의 한쪽 끝





### 사용법

아래 코드는 std::future을 3가지 방법으로 얻는 과정을 보여준다. 공통점은 어떠한 실행 스레드의 결과를 받아오는 핸들이라는 점. 이는 스레드를 멈추거나, 반환 값을 얻어올 수 있다.

int main()

{

// future from a packaged\_task

std::packaged\_task<int()> task([](){ return 7; }); // wrap the function

std::future<int> f1 = task.get\_future(); // get a future

std::thread(std::move(task)).detach(); // launch on a thread

// future from an async()

std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });

// future from a promise

std::promise<int> p;

std::future<int> f3 = p.get\_future();

std::thread([&p]{ p.set\_value\_at\_thread\_exit(9); }).detach();

std::cout << "Waiting..." << std::flush;

f1.wait();

f2.wait();

f3.wait();

std::cout << "Done!\nResults are: "

<< f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';

}

#### std::packaged\_task

함수를 감싸는 기능이다. std::function과 비슷해 보이지만 이는 비동기적으로 호출할 수 있도록 감싼다. std::future와 통신을 통하여 이를 가능하게 한다.

void task\_lambda()

{

std::packaged\_task<int(int,int)> task([](int a, int b) {

return [std::pow](http://en.cppreference.com/w/cpp/numeric/math/pow)(a, b);

});

[std::future](http://en.cppreference.com/w/cpp/thread/future)<int> result = task.get\_future();

task(2, 9);

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "task\_lambda:**\t**" << result.get() << '**\n**';

}

## std::thread vs. std::future

굳이 이런 객체가 존재해야 할 이유는 무엇인가? std::thread의 join과 detach로 스레드 실행을 제어할 수 있는데 std::future만이 할 수 있는 일은 무엇인가?

### std::future은 결과값을 받아오는 유일한 수단이다.

std::thread만으로는 결과값을 받아올 수 없다. std::promise와 함께 공유상태를 통하여 std::future를 통해야 한다. 즉, std::future은 결과를 받아오는 스레드 핸들인 것이다.

### std::future의 소멸자의 행동을 그 객체와 연관된 공유 상태가 결정한다.

공유 상태에 따라서 행동이 바뀐다는 의미다. 이는 간단한 정상 행동과 단 하나의 예외를 말한다.

#### 정상 행동: 미래 객체의 소멸자가 미래 객체를 파괴한다.

어떠한 루프에서의 미래 객체를 선언하고 그 안에서는 이 미래 객체가 다루는 어떠한 연산이 작동하고 있다. 그런데 이 루프가 종료되었다. 그럼 미래객체는 어떻게 되는가? 그냥 파괴되는 것이다. 암묵적 detach와 비슷하다. 한편 std::thread같은 경우에는 프로그램이 종료된다.

#### 예외: 과제가 완료될 때까지 기다린다.

다음과 같은 조건이 모두 성립해야 한다.  
1. 미래 객체가 std::async호출에 의해 생성된 공유 상태를 참조한다.  
2. 과제의 시동 방침이 std::launch::async이다.  
3. 미래 객체가 공유 상태를 참조하는 마지막 미래 객체이다.

이러한 스레드에 대해서 암묵적 join을 수행한다. 즉, 스레드가 완료될 때까지 미래 객체의 소멸자는 차단된다.

### std::future의 소멸자 행동 예시

몇 가지 상황을 두고 소멸자가 어떻게 작동하는지 살펴보자.

#### 암묵적 detach가 수행되는 상황

일반적으로는 이렇게 수행된다. 범위와 상관없이 알아서 수행한다.

int main()

{

int counter1 = 0, counter2 = 0, counter3 = 0;

{

// future from a packaged\_task

std::packaged\_task<int()> task([&](){

for (int i = 0; i < MAX; i++) counter1++;

return 7; }); // wrap the function

std::future<int> f1 = task.get\_future(); // get a future

std::thread(std::move(task)).detach(); // launch on a thread

// future from an async()

std::future<int> f2 = std::async(std::launch::async, [&](){

for (int i = 0; i < MAX; i++) counter2++;

return 8; });

// future from a promise 이건 안 됨…?

}

std::cout << counter1 << '\n'; //208158

std::cout << counter2 << '\n'; //31538

std::cout << counter3 << '\n'; //0

return 0;

}

첫 번째, 두 번째 모두 future객체는 공유상태를 참조하지 않고 있으므로 합류하지 않고 detach가 수행된다.

#### 암묵적 join이 수행되는 상황

wait()이나 get()을 사용했다면 공유 상태를 참조하므로 암묵적 join이 수행된다. 그 외에도 해당 스레드가 공유 상태를 참조한다면 암묵적 join상태가 될 것이다.

const long long MAX = 1000000000;

int main()

{

std::atomic<long long> counter = 0;

{

// future from a packaged\_task…

// future from an async()…

// future from a promise

std::cout << "Waiting...\n" << std::flush;

std::cout << "Done!\nResults are: "

<< f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';

}

std::cout << counter << '\n';

return 0;

}

### 결론

여러 가지 규칙이 복잡하게 나열되었지만 핵심은 std::future은 확실한 결과를 반환한다는 보장이 있다. 마무리가 확실한 것이다. 반면 std::thread자체는 결과를 반환하는 기능이 없다. 결과를 반환하지 않는데 굳이 확실한 끝 처리가 있을 필요는 없다. 사용자가 정의할 뿐이다. std::future의 사용은 결과에 집중하며 이러한 점은 루틴 제어나 통신에도 이용될 수 있다.

***미래 객체의 소멸자는 그냥 미래 객체의 자료 멤버들을 파괴할 뿐이다.***

***std::async를 통해 시동된 비지연 과제에 대한 공유 상태를 참조하는 마지막 미래 객체의 소멸자는 그 과제가 완료될 때까지 차단된다.***

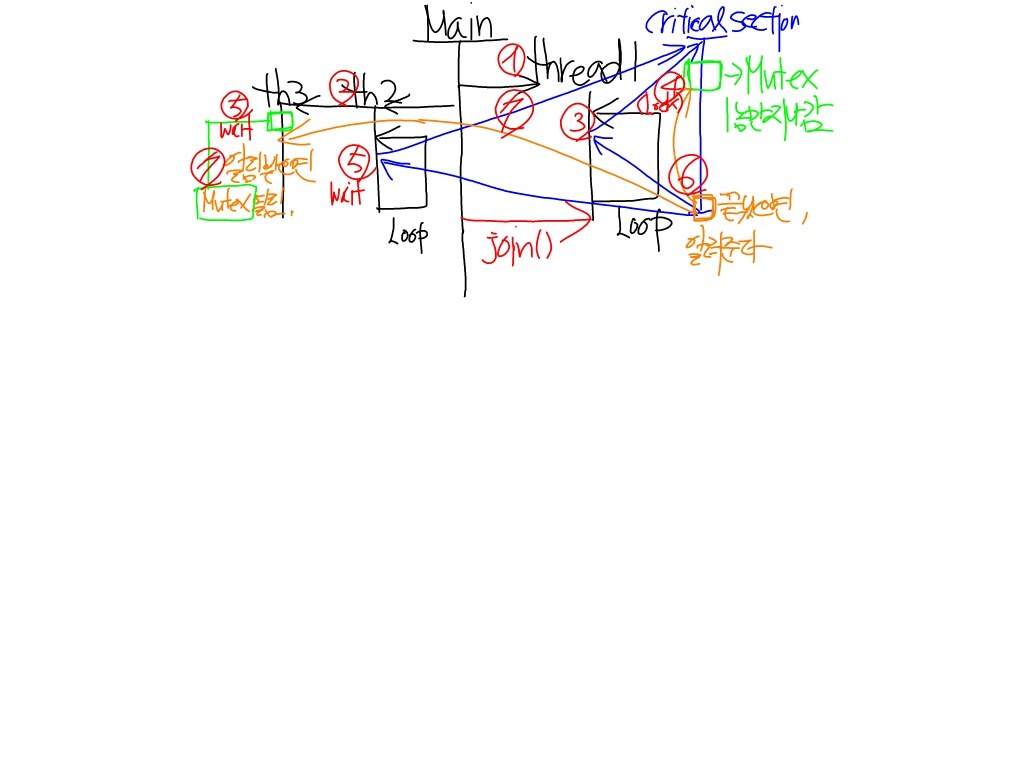
[항목39] 스레드간 사건 통신

단발성 사건 통신에는 void 미래 객체를 고려해라.

# 동기화 과정

동기화 참고: <http://bartysways.net/?tag=gcd>

동기화되는 하나의 예시를 그림으로 살펴본다.



스레드 관련 설계에서 다음과 같이 고려해 보았다.  
1. 임의의 스레드를 만들 것이다. 이것의 루틴은 단일인가, 반복인가?  
2. 어떠한 서브루틴을 공유하는지 연결해본다. 2개 이상의 스레드가 접근한다면 동기화가 필요한지 고려한 후, 필요하다면 잠가야 한다.  
3. 스레드의 합류, 탈착의 여부를 고려한다.   
4. 스레드간 어떤 통신(이벤트)가 필요한지 고려한다. 루틴이 아니라면 조건 변수나 promise로 막아놓아야 하며, 루틴이라면 flag로 가능하지만, 역시 막아놓는 것이 효율상 좋다.   
5. 필요한 통신의 전송 부분, 송신 부분을 구체적으로 표시하여 설계한다.

어떤 상황에 어떤 객체가 적절한지는 이제부터 살펴볼 것이다.

# std::mutex

뮤텍스는 동기화를 가능하게 한다. 어떻게 가능하게 하는가? 어떠한 범위에서 임의의 스레드가 뮤텍스를 잠그는 행동을 할 수 있다. 이렇게 잠긴 상태에서 또 다른 스레드가 접근한다면 뮤텍스는 이 스레드를 잠재우게 만든다. 그러므로 뮤텍스가 잠겨있는 부분 이후로는 하나의 스레드만 접근하게 된다. std::mutex는 혼자서 잠그고 열수도 있지만 아래의 뮤텍스 관리 클래스와 함께 쓰는 것이 일반적이다.

# std::lock\_guard<std::mutex>

이는 mutex를 감싸는 RAII클래스이다. 생성하면 뮤텍스가 저절로 잠기며, 소멸자에서 스스로 뮤텍스를 해제시킨다. 즉, 범위를 벗어나면 자동 해제이며, 강제로 해제할 수 없다.

class DarkWorld

{

public:

DarkWorld()

{

loh.monsterSpawnStart(1000);

}

void entry(Archer\* pA)

{

loh.entryLakeOfHell(pA);

}

private:

struct LakeOfHell

{

std::mutex m;

int monsterCount;

void monsterSpawnStart(int count)

{

std::async(std::launch::async, **[&, count]**

{

monsterCount = 0;

while (true)

{

monsterCount += count; std::this\_thread::sleep\_for(std::chrono::seconds(5));

}

});

}

void entryLakeOfHell(Archer\* pA)

{

std::lock\_guard<std::mutex> lock(this->m);

while (pA->isContinue() && monsterCount > 0)

{

pA->huntMonster(monsterCount);

std::this\_thread::sleep\_for(std::chrono::milliseconds(2));

}

}

};

LakeOfHell loh;

};

class Player

{

public:

template<typename TN>

void createArcher(TN&& \_name)

{

pA = new Archer(std::forward<TN>(\_name));

}

void startPlay(DarkWorld\* dw)

{

flag = true;

std::async(std::launch::async, [&]

{

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout << "player가 게임을 시작합니다.\n";

while (flag)

{

dw->entry(pA);

}

});

}

void stopPlay() { flag = false; }

private:

Archer\* pA;

bool flag;

};

# std::unique\_lock<std::mutex>

이는 std::lock\_guard에 더해진 기능을 갖는 객체로, 잠금과 해제를 유연하게 조절하는 기능을 가진 mutex관리 객체이다.

완전하게 std::lock\_guard와 같이 사용할 수 있다. 게다가 잠그고 여는 작업을 임의로 할 수 있으며, 이는 뮤텍스가 여러 군데에서 공유하는 점을 이용해서 다양한 루틴에서의 통신을 가능하게 한다. 예외 발생 시, 뮤텍스를 열어주는 처리까지 동반한다.

void entryLakeOfHell(Archer\* pA)

{

std::unique\_lock<std::mutex> lock(this->m);

while (pA->isContinue() && monsterCount > 0)

{

//…

}

}

# std::condition\_variable

어떤 특정한 사건이 발생했을 때, 뮤텍스에게 알려주는 장치이다. 동기화 관련 이벤트 장치라 생각하면 좋겠다.   
어떠한 조건이 발생하면 이를 조건 변수 cv(std::condition\_variable)는 반응해야 하는 **구독자들에게 알려준다. 이를 cv.notify\_one, cv.notify\_all**로 할 수 있다.   
한편, 구독자는 무엇인가? 어떠한 구간에서 조건 변수의 반응을 기다리고 있는 스레드들이다. 이는 **cv.wait(열리는 조건)**으로 가능하게 한다. **스레드들을 멈추게 하려면 잠금 장치가 필요하므로 std::unique\_lock이 필수**이다. 예제를 보자.

std::mutex m;

std::condition\_variable cv;

std::string data;

bool ready = false;

bool processed = false;

void worker\_thread()

{

// 1. Wait until main() sends data

std::unique\_lock<std::mutex> lk(m);

cv.wait(lk, []{return ready; });

// 3\_worker. after the wait, we own the lock.

std::cout << "2\n";

data += " after processing";

// 4. Send data back to main()

processed = true;

std::cout << "3\n";

// 5. Manual unlocking is done before notifying, to avoid waking up

// the waiting thread only to block again (see notify\_one for details)

//이걸 열지 않으면 여기서는 작동하지만 뮤텍스를 잠근 채로 깨우게 된다.

lk.unlock();

cv.notify\_one();

}

int main()

{

std::thread worker(worker\_thread);

data = "Example data";

// 2. send data to the worker thread

{

std::lock\_guard<std::mutex> lk(m);

ready = true;

std::cout << "1\n";

}

cv.notify\_one();

// 3\_main. wait for the worker

{

std::unique\_lock<std::mutex> lk(m);

cv.wait(lk, []{return processed; });

}

std::cout << "4, data = " << data << '\n';

worker.join();

}

위는 주석의 번호 순서대로 진행된다. 출력도 번호 순서대로 진행된다.

또 다른 예제로 위에서 std::lock\_guard로 구현한 내용을 std::condition\_variable로 구현한다. 이것은 반복하면서 flag를 확인하는 방식이 아니라, 몬스터를 다 잡았으면 cv에게 알려서 스포너를 깨우는 방식이다.

void monsterSpawnStart(int count)

{

std::async(std::launch::async, [&, count]

{

monsterCount = 0;

while (true)

{

std::unique\_lock<std::mutex> lock(this->m);

cv.wait(lock, [this]{ return monsterCount == 0; });

monsterCount += count;

//범위가 바로 끝나므로 안 잠가도 된다.

//lock.unlock();

cv.notify\_one();

}

});

}

void entryLakeOfHell(Archer\* pA)

{

std::unique\_lock<std::mutex> lock(this->m);

cv.wait(lock, [this]{ return monsterCount > 0; });

while (pA->isContinue() && monsterCount > 0)

{

pA->huntMonster(monsterCount);

std::this\_thread::sleep\_for(std::chrono::milliseconds(2));

}

//lock.unlock();

cv.notify\_all();

}

참고로 위의 wait에 조건식([this]{ return monsterCount > 0; })은 반드시 넣어주어야 한다. 작동하지 않는다.

## [악취1] 필요 없는 mutex

위 코드에서는 mutex가 필요 없다. 하지만 C++11의 요구로 std::condition\_variable을 사용하려면 std::unique\_lock으로 잠가주어야 한다.

## [악취2] wait이 실행하기 전에 통지가 된다면 영원히 멈춘다.

main스레드에 활발한 활동의 결과로 wait에 들어가기도 전에 notify가 호출된다면 이는 실패하게 될 것이다.

void monsterSpawnStart(int count)

{

std::async(std::launch::async, [&, count]

{

monsterCount = 0;

while (monsterCount == 0)

{

std::unique\_lock<std::mutex> lock(this->m);

monsterCount += count;

std::cout << "현재 몬스터의 수: " << monsterCount << '\n';

lock.unlock();

cv.notify\_all();

}

});

}

void entryLakeOfHell(Archer\* pA)

{

std::unique\_lock<std::mutex> lock(this->m);

cv.wait(lock);

while (pA->isContinue() && monsterCount > 0)

{

pA->huntMonster(monsterCount);

std::this\_thread::sleep\_for(std::chrono::milliseconds(2));

}

lock.unlock();

}

위와 같은 상황인데, 시작하면 monsterSpawnStart 함수가 호출되어 몬스터를 생산한다. 하지만 이는 벌써 실행한 이후이므로 entryLakeOfHell 함수를 깨울 수 없다. 이런 경우 통제하기가 난감하다.

## [악취3] cv.wait은 가짜 기상을 고려하지 않는다.

실제로 원하지 않는 반응에도 일어난다. 민감한 것이다. 이는 다음과 같이 해결한다. 공유 플래그를 전역변수로 두어 합치는 것이다.

std::condition\_variable cv;

std::mutex m;

bool flag = false;

int main()

{

std::async([]()

{

std::unique\_lock<std::mutex> ul(m);

cv.wait(ul, []{ return flag; });

std::cout << "react.\n";

});

std::cout << "start norify.\n";

flag = true;

cv.notify\_one();

return 0;

}

위 코드는 잘 작동한다. 하지만 조건 변수가 있음에도 한번 더 변수를 확인해야 하며, 여전히 뮤텍스의 악취는 피할 수 없다. 말끔한 통신 방법이라고 할 수 없다.

# std::promise<T>

또 하나의 통신 채널이다. 스레드 간의 통신을 만들어 주는데, 한 장소에서 다른 장소로 정보를 전송해야 하는 모든 상황에서 사용할 수 있다.  
전송 단자는 std::promise가 되며, 피전송 단자는 미래 객체가 된다.   
이를 위의 상황에 적용하면 통지를 알리는 객체(검출 과제)는 std::promise가 되며, set\_value로 값을 지정하면 통지를 할 수 있다. 한편, 통지를 받는 객체(반응 과제)는 미래객체인데, 이는 p.get\_future()로 얻을 수 있으며, p.get\_future().wait은 해당 미래객체가 통지를 받을 때 까지 기다린다는 의미가 된다.

std::promise<void> p;

int main()

{

std::async([]()

{

//std::unique\_lock<std::mutex> ul(m);

//cv.wait(ul, []{ return flag; });

p.get\_future().wait();

std::cout << "react.\n";

});

std::cout << "start norify.\n";

//flag = true;

//cv.notify\_one();

p.set\_value();

int x; std::cin >> x;

return 0;

}

이를 위에서의 예시에 적용해보자.

std::mutex m;

std::condition\_variable cv;

std::promise<void> p;

int monsterCount;

void monsterSpawnStart(int count)

{

std::async(std::launch::async, [&, count]

{

monsterCount = 0;

std::this\_thread::sleep\_for(std::chrono::seconds(3));

while (true)

{

if (monsterCount > 0) p.get\_future().wait();

monsterCount += count;

std::cout << "현재 몬스터의 수: " << monsterCount << '\n';

p.set\_value(); //통지할 수 있다.

}

});

}

void entryLakeOfHell(Archer\* pA)

{

p.get\_future().wait();

while (pA->isContinue() && monsterCount > 0)

{

pA->huntMonster(monsterCount);

std::this\_thread::sleep\_for(std::chrono::milliseconds(2));

}

p.set\_value();

}

훨씬 간단해지고 효율도 좋은 모습이 보인다. 그러나 이는 한번만 작동한다.

## std::promise 통신의 장점

이는 뮤텍스가 필요하지 않으며, 반응 과제가 wait으로 대기하기 전에 검출 과제가 자신의 std::promise를 설정해도 작동하며, 가짜 기상도 없다. 그리고 기다리는 동안 시스템 자원도 소모하지 않는다.

## [악취1] 공유 상태는 동적 할당된다.

std::promise와 미래객체 사이에는 공유 상태가 있으며, 그것은 동적으로 할당된다. 그러므로 할당 및 해제 비용을 지불해야 한다.

## [악취2] 단발성 메커니즘

std::promise는 1번밖에 사용할 수 없다. 반면 std::condition\_variable은 그렇지 않다.

## 스레드 유보상태 만들기

std::promise로 여러 개의 스레드들을 유보했다가 하나의 통지로 여러 개의 스레드들이 한번에 통지 받을 수는 없을까? 핵심은 std::future대신에 std::shared\_future를 반환하는 것이다.

std::promise<void> p;

void detect()

{

auto sf = p.get\_future().share();

std::vector<std::thread> vt;

for (int i = 0; i < 3; i++)

{

vt.emplace\_back([sf]

{

sf.wait();

std::cout << "react.\n";

});

}

std::cout << "start norify.\n";

p.set\_value();

for (auto& t : vt)

{

t.join();

}

}

int main()

{

detect();

int x; std::cin >> x;

return 0;

}

### std::shared\_future의 사용

클래스 템플릿 std::shared\_future는 다중 스레드가 동일한 공유 상태를 기다리는 것을 가능하게 한다. std::future는 단지 이동만 가능하며, std::shared\_future는 복사 가능하며 여러 공유 미래 객체는 동일한 공유 상태를 참조 할 수 있다.

int main()

{

std::promise<void> ready\_promise, t1\_ready\_promise, t2\_ready\_promise;

std::shared\_future<void> ready\_future(ready\_promise.get\_future());

auto result1 = std::async(std::launch::async,

[&, ready\_future]() -> std::chrono::duration<double, std::milli>

{

t1\_ready\_promise.set\_value();

ready\_future.wait(); // waits for the signal from main()

return std::chrono::high\_resolution\_clock::now() - start;

};

);

auto result2 = std::async(std::launch::async,

[&, ready\_future]() -> std::chrono::duration<double, std::milli>

{

t2\_ready\_promise.set\_value();

ready\_future.wait(); // waits for the signal from main()

return std::chrono::high\_resolution\_clock::now() - start;

};

);

// wait for the threads to become ready

t1\_ready\_promise.get\_future().wait();

t2\_ready\_promise.get\_future().wait();

// signal the threads to go

ready\_promise.set\_value();

std::cout << "Thread 1 received the signal "

<< result1.get().count() << " ms after start\n"

<< "Thread 2 received the signal "

<< result2.get().count() << " ms after start\n";

}

**ready가 여러 스레드에서 복사되고 있다. 그리고 복사된 모든 스레드에서 신호를 기다리고 있다.** 간단하게 생각해야 한다**. wait은 신호를 기다리는 구간이다. 그리고 set\_value는 신호를 주는 위치이다.** 즉, 이 과정은 3개의 스레드이며, 메인에서 스레드1, 스레드2의 신호를 메인에서 기다린다. 그리고 ready는 복사된 모든 wait구간에 신호를 주어 실행이 완료된다.

### ThreadRAII를 사용해도 될까?

도중에 예외가 발생하면 너무 일찍 join을 호출하기 때문에 영원히 멈추게 된다. 이를 해결하려면 The View From Aristeia의 ThreadRAII + Thread Suspension = Trouble? 참고

***간단한 사건 통신을 수행할 때, 조건 변수 기반 설계에는 여분의 뮤텍스가 필요하고, 검출 과제와 반응 과제의 진행 순서에 제약이 있으며, 사건이 실제로 발생했는지를 반응 과제가 다시 확인해야 한다.***

***플래그 기반 설계를 사용하면 그런 단점은 없지만, 대신 차단이 아니라 폴링이 일어난다는 단점이 있다.***

***조건 변수와 플래그를 조합할 수도 있으나, 그런 조합을 이용한 통신 메커니즘은 필요 이상으로 복잡하다.***

***std::promise와 미래객체를 사용하면 이러한 문제점들을 피할 수 있지만, 그런 접근 방식은 공유 상태에 힙 메모리를 사용하며, 단발성 통신만 가능하다.***

[항목40] std::atomic vs. volatile

동시성에는 std::atomic<T>을 사용하고, volatile은 특별한 메모리에 사용하라.

C#에서는 하나의 변수에 대해 동시성을 제어하기 위해 volatile을 사용할 수 있다. 하지만 C++에서는 아니다.

# std::atomic<T>

std::atomic<T>로 선언한 변수는 이에 대한 연산의 동시성을 보장한다.

## std::atomic이란?

template< class T >

struct atomic;

하나의 원자 원소에 대하여 객체를 만든다. 하나의 스레드가 다른 스레드가 스레드를 읽는 동안 그 객체에 쓰는 경우, 그 동작은 잘 정의되어 있습니다. (데이터 모델에 대한 자세한 내용은 *메모리 모델* 참조)

[조사] 이전에는 atomic 연산을 하기 위해서는 volatile 변수와 interlocked 계열 함수를 일일이 사용해 주어야 했지만, std::atomic 클래스로 인해 그러한 번거로움을 피할 수 있게 된다.  
#include *<atomic>*

**using** **namespace** std;

int \_tmain(int argc, \_TCHAR\* argv[])

{

*///////////////////////////////////////////////////////////////////////////////*

*/// 생성*

*///////////////////////////////////////////////////////////////////////////////*

*// 먼저 atomic 객체를 선언하고, 값을 대입 OK*

atomic<int> intAtomic;

intAtomic = 2;

*// atomic 객체간 복사생성/대입연산은 금지되어 있다.*

*// 복사생성 시도 -> 컴파일 에러*

atomic<int> intAtomic2 = intAtomic;

*// 대입연산 시도 -> 컴파일 에러*

atomic<int> intAtomic2;

intAtomic2 = intAtomic;

*// atomic<int>와 atomic<short>는 엄연히 다른 클래스*

*// 그렇기에 아래 문장은 다음과 같이 수행된다.*

*// 1) int tempInt = intAtomic.load();*

*// 2) atomic<short> shortAtomic = tempInt;*

*// 따라서 컴파일에 아무런 문제가 없다.*

atomic<short> shortAtomic = intAtomic;

*///////////////////////////////////////////////////////////////////////////////*

*/// +,- 연산/논리 연산*

*///////////////////////////////////////////////////////////////////////////////*

*// 값 더하기 : 멤버 함수 fetch\_add*

intAtomic.fetch\_add(1);

*// 값 더하기 : 멤버 함수 operator++(), operator++(int)*

intAtomic++;

*// 값 더하기 : 일반 함수 atomic\_fetch\_add*

atomic\_fetch\_add(&intAtomic, 1);

*// 값 빼기 : 멤버 함수 fetch\_sub*

intAtomic.fetch\_sub(1);

*// 값 빼기 : 멤버 함수 operator--(), operator--(int)*

intAtomic--;

*// 값 빼기 : 일반 함수 atomic\_fetch\_sub*

atomic\_fetch\_sub(&intAtomic, 1);

*// and : 멤버 함수 fetch\_and (0x10 & 0x10)*

intAtomic.fetch\_and(2);

*// and : 멤버 함수 operator &= (0x10 & 0x10)*

intAtomic &= 2;

*// and : 일반 함수 atomic\_fetch\_add (0x10 & 0x01)*

atomic\_fetch\_add(&intAtomic, 1);

*// or : 멤버 함수 fetch\_or (0x00 | 0x01)*

intAtomic.fetch\_or(1);

*// or : 멤버 함수 operator |= (0x01 | 0x10)*

intAtomic |= 2;

*// or : 일반 함수 atomic\_fetch\_or (0x11 | 0x01)*

atomic\_fetch\_or(&intAtomic, 1);

*// xor : 멤버 함수 fetch\_xor(0x11 ^ 0x01)*

intAtomic.fetch\_xor(1);

*// xor : 멤버 함수 operator ^= (0x10 ^ 0x01)*

intAtomic ^= 1;

*// xor : 일반 함수 atomic\_fetch\_xor (0x11 ^ 0x01)*

atomic\_fetch\_xor(&intAtomic, 1);

*///////////////////////////////////////////////////////////////////////////////*

*/// 읽기/저장/교환/비교 교환*

*///////////////////////////////////////////////////////////////////////////////*

*// 값 로드 : 멤버 함수 load*

int value = intAtomic.load();

*// 값 로드 : 멤버 함수 Operator T*

value = intAtomic;

*// 값 로드 : 일반 함수 atomic\_load*

value = atomic\_load(&intAtomic);

*// 값 저장 : 멤버 함수 store*

intAtomic.store(3);

*// 값 저장 : 멤버 함수 operator= (T value)*

intAtomic = 3;

*// 값 저장 : 일반 함수 atomic\_store*

atomic\_store(&intAtomic, 3);

*// 값 교환 : 멤버 함수 exchange*

int oldValue = intAtomic.exchange(5);

*// 값 교환 : 일반 함수 atomic\_exchange*

oldValue = atomic\_exchange(&intAtomic, 3);

int comparand = 5;

int newValue = 10;

*// 값 비교 교환 : 멤버 함수 (value = 3, 5와 같다면, 10으로 value를 바꾸어라)*

*// 수행 후 comparand는 원래 value인 3로 바뀐다.*

bool exchanged = intAtomic.compare\_exchange\_weak(comparand, newValue);

*// 값 비교 교환 : 일반 함수*

*// 앞서 comparand가 3로 바뀌었기에, 값이 10으로 바뀐다*

exchanged = atomic\_compare\_exchange\_weak(&intAtomic, &comparand, newValue);

*// VS2013에서는 compare\_exchange\_weak와 compare\_exchange\_strong 구현이 동일하다.*

*// compare\_exchange\_weak가 compare\_exchange\_strong을 호출한다.*

*///////////////////////////////////////////////////////////////////////////////*

*/// 유틸 함수*

*///////////////////////////////////////////////////////////////////////////////*

*// std::atomic은 해당 오브젝트 크기가 8바이트 이하이면 atomic을 보장한다.*

*// \_ATOMIC\_MAXBYTES\_LOCK\_FREE = 8*

bool is\_lock\_free = intAtomic.is\_lock\_free();

**return** 0;

}

### 메모리 모델

메모리 모델이란, C ++ 추상 기계의 목적을 위한 컴퓨터 기억 장치 기억의 의미론을 정의합니다.

C ++ 프로그램에서 사용할 수 있는 메모리는 하나 이상의 인접한 바이트 시퀀스입니다. 메모리의 각 바이트에는 고유한 주소가 있습니다. 각 바이트의 고유한 주소(메모리 위치)는 다음과 같이 되어있다.

struct S {

char a; // memory location #1

int b : 5; // memory location #2

int c : 11, // memory location #2 (continued)

: 0,

d : 8; // memory location #3

struct {

int ee : 8; // memory location #4

} e;

} obj; // The object 'obj' consists of 4 separate memory locations

또한, 원자적 객체에 대한 액세스는 스레드 간 동기화를 설정하고 *std::memory\_order*에 지정된 비 원자 메모리 액세스를 정렬 할 수 있습니다.

### std::memory\_order

[미결] std::memory\_order는 원자 연산(중간에 어떠한 방해도 받지 않고 어셈블리어 명령어 하나로 실행할 수 있다.)을 통해 일반, 비 - 원자 메모리 액세스를 정렬하는 방법을 지정합니다. 다중 코어 시스템에서 제약 조건이 없으면 여러 스레드가 동시에 여러 변수를 읽거나 쓸 때 한 스레드가 다른 스레드가 작성한 순서와 다른 순서로 값이 변경되는 것을 관찰 할 수 있습니다. 실제로 변경 순서는 여러 리더 스레드간에 다를 수 있습니다. 메모리 모델에 의해 허용되는 컴파일러 변환으로 인해 유니 프로세서 시스템에서도 유사한 효과가 발생할 수 있습니다.

## 증가, 감소 연산을 원자적 연산으로 인식한다.

++a: a값을 적재 -> a값을 증가 -> a에 대입

이러한 BMW(읽기, 수정, 쓰기)연산을 하나의 원자적 연산으로 수행한다는 의미다. 일반적으로 아무것도 보장되지 않는다면 동시에 a에 접근하고 쓰기 때문에 미정의 동작을 유발한다. 이렇게 한 메모리에 기록자(writer)들과 판독자(reader)들이 동시에 접근하려는 상황을 *자료 경쟁(data race)*라 한다.

### Race Condition vs. Data Race

<http://blog.regehr.org/archives/490>

*경쟁 조건*은 이벤트의 타이밍 또는 순서가 프로그램의 정확성에 영향을 미칠 때 발생하는 결함입니다. 일반적으로 말하면, **경쟁 조건을 생성하기 위해서는 일종의 외부 타이밍 또는 비 결정성 순서가 필요**합니다. 반면, *데이터 경쟁*은 어떠한 데이터에 동시에 접근하는 행위 자체를 말합니다.

실제로는 상당한 중복이 있습니다. 많은 경쟁 조건은 데이터 경쟁으로 인해 발생하며 많은 데이터 경쟁으로 인해 경쟁 조건이 발생합니다. 반면에 경쟁 조건이 없는 데이터 경쟁 및 데이터 경쟁 없이 경쟁 조건을 가질 수 있습니다. 두 은행 계좌간에 돈을 이동하는 간단한 함수로 시작합시다.

class Account

{

public:

int balance;

};

bool transfer1(int amount, Account& account\_from, Account& account\_to)

{

if (account\_from.balance < amount) return false;

account\_to.balance += amount;

account\_from.balance -= amount;

return true;

}

위 함수는 데이터 경합(여러 스레드가 동시에 계정 잔액을 업데이트하려고 시도 할 수 있음)과 경쟁 조건(병렬 컨텍스트에서 돈을 생성하거나 잃게 됨)을 허용합니다. 우리는 이것을 다음과 같이 고칠 수 있습니다.

class Account

{

public:

std::atomic<int> balance;

};

Account의 balance가 원자적으로 수행됩니다. 하나의 표현식은 원자적으로 실행되지만, 각 표현식끼리의 순서는 동기화되지 않고 뒤죽박죽 수행될 수 있습니다. 데이터 경쟁으로는 자유롭지만 경쟁 조건은 있습니다.

std::mutex m;

bool transfer3(int amount, Account& account\_from, Account& account\_to)

{

std::unique\_lock<std::mutex> lock(m);

if (account\_from.balance < amount) return false;

account\_to.balance += amount;

account\_from.balance -= amount;

return true;

}

위 함수는 함수 내에서 완벽하게 동기화합니다. 그러므로 데이터 경쟁과 경쟁 조건으로부터 자유로운 함수식입니다. 그럼 데이터 경쟁은 있지만 경쟁 조건은 없다는 것은 무엇일까요?

class Account

{

public:

bool activity;

int balance;

};

bool transfer4(int amount, Account& account\_from, Account& account\_to)

{

account\_from.activity = true;

account\_to.activity = true;

std::unique\_lock<std::mutex> lock(m);

if (account\_from.balance < amount) return false;

account\_to.balance += amount;

account\_from.balance -= amount;

return true;

}

계정의 계산은 동기화되어 문제가 없다. 하지만 activity는 동시에 접근해도 문제가 없다. 이는 굳이 동기화하지 않아도 되는 것이다. 즉, 데이터 경쟁은 허용되지만 경쟁 조건은 없다.

이 4가지 함수를 데이터 경쟁과 경쟁 조건을 기준으로 비교하면 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
|  | data race | no data race | |
| race condition | transfer1 | transfer2 |
| no race condition | transfer4 | transfer3 |

**결론:** 경쟁 조건이 생기지 않는 범위에서 데이터 경쟁은 허용해도 된다. 데이터가 경쟁해도 정확할 수 있으며 오히려 동기화로 인한 성능 문제를 피할 수 있으므로 설계 시 고려해야 한다.

### 경쟁 조건과 데이터 경쟁 예시

std::atomic<int> totalSell;

class Archer

{

public:

Archer(std::string \_name) : name(\_name), account(10000) {}

void buyItem(int amount)

{

account -= amount;

totalSell += amount;

std::cout << name + "는 포션을 구입했습니다.\n";

}

void intro() { std::cout << name + "입니다.\n"; }

private:

std::string name;

int account;

};

class Shop

{

public:

Shop() : onWorking(false), account(0) {}

void sellItem(Archer& buyer)

{

onWorking = true; //데이터 경쟁

std::unique\_lock<std::mutex> lock(m);

buyer.buyItem(potion); //데이터 경쟁 + 경쟁 조건 없음

account += potion;

lock.unlock();

onWorking = false;

}

int getSettlement() { return account; }

private:

static const int potion = 100;

std::mutex m;

bool onWorking;

int account;

};

int main()

{

Shop shop;

std::vector<Archer> shoppers;

shoppers.emplace\_back(Archer("ytk"));

shoppers.emplace\_back(Archer("dk"));

shoppers.emplace\_back(Archer("yth"));

for (auto& s : shoppers)

{

s.intro();

std::thread([&]()

{

while (true)

{

shop.sellItem(s);

std::this\_thread::sleep\_for(std::chrono::milliseconds(rand() % 1000 + 1000));

}

}).detach();

}

std::this\_thread::sleep\_for(std::chrono::seconds(10));

std::cout << totalSell << '\n';

std::cout << shop.getSettlement() << '\n';

return 0;

}

## 코드 순서 재배치에 대한 제약이 생긴다.

일반적으로 컴파일러가 서로 무관한 여러 배정문장 사이에서 순서를 바꾸는 것은 적법하다.

하지만 std::atomic<T>변수를 기록하는 문장 이전의 모든 문장들은 해당 문장 이후에 실행될 수 없다.

std::atomic<int> value(9999);

int a = 1;

int b = 3;

value -= a;

// a, b대입은 절대로 이 앞으로 실행될 수 없다.

// 만약 그렇게 된다면 value는 동기화가 보장되지 않을 것이다.

a = 9999;

value -= b;

//마찬가지

## std::atomic은 복사나 이동연산이 불가능하다.

std::atomic에는 복사 생성자, 대입 연산자가 삭제되었다. 왜냐하면 복사 자체가 불러오고 대입하는 2가지 연산이 있는데 하드웨어 수준에서 이 2가지 연산을 하나의 원자적 연산으로 제공할 수 없다. 대신 load, store함수를 사용하여 적재하고 불러올 수 있다.

## 컴파일러 최적화

컴파일러 최적화를 수행한다.

### 중복 대입을 제거한다.

//원래 코드

std::atomic<int> x;

auto y = x;

x = 10;

x = 20;

//최적화 코드

auto y = x;

x = 20;

컴파일러는 쓸데없이 같은 값을 2번 불러오지 않게, 혹은 필요 없는 대입을 하지 않게 스스로 최적화시킨다. 이를 위해서 *남아도는 적재와 죽은 저장*이 생긴다. 필요 없는 부분을 제거하기 때문이다.

### 레지스터에 저장하여 중복 읽기를 방지한다.

std::atomic은 어떻게 처리할까? 위의 최적화를 허용한다.

//원래 코드

std::atomic<int> x = 10;

std::atomic<int> y(x.load());

y.store(x.load());

//최적화

std::atomic<int> x = 10;

register int ry = x.load();

std::atomic<int> y(ry);

y.store(ry);

위는 반복해서 불러오는 변수를 레지스터 변수에 저장하여 최적화시킨다. 이는 x가 한번만 읽힐 때 가능한 최적화이며 동시성 제어에서 의미 있다.

# volatile

해당 코드가 특별한 메모리를 다룬다는 점을 컴파일러에게 알려주는 수단이다.

### 특별한 메모리가 무엇인가?

여러 프로세스가 공유하는 메모리 장소이다. 메모리 대응 IO에 쓰이는 메모리가 그러하다. 일반적인 메모리는 CPU만이 접근하는데 반해, 이러한 메모리들은 외부 장치가 수시로 접근한다. 언제든지 유효한 값을 읽을 수 있게 보장해야 하는 변수가 volatile인 것이다.

## 동시성 비 제어

동시성 제어와는 연관이 없다. 위의 std::atomic처럼 재배치에 대한 제약도 없으며, 특정 묶음 연산을 원자적 단위로 처리하지도 않는다. 사실 volatile이 이를 처리할 이유가 없다. **이러한 제어는 입력과 출력의 순서를 제어하는 것이 핵심이다.** 즉, 입력만 2번 받고 출력을 진행하면 결과가 다르다는 것이다. 입력1, 출력1, 입력1…… 이 순서를 제어하는 것이 동기화이다. 반면 volatile은 어떠한 시간에 들어오든 읽기만 유효하면 된다. 다른 의미라는 것이다.

## 컴파일러 최적화

위에서 언급한 어떤 최적화도 적용되지 않는다. 최적화란 연산을 생략한다는 것인데, 도중에 값이 임의로 자료를 변경하면 중간에 변경되는 중간 결과 값이 달라진다는 것이 이유다.

### 중복 대입을 제거할 수 없다.

중간에 대입된 과정이 특별한 메모리에서는 필요할 수 있다. 결과와 과정은 값이 다르기 때문이다.

### 레지스터에 저장하지 않는다.

같은 값을 쓴다는 보장 하에 레지스터에 저장하여 중복 호출하지 않는 것이 컴파일러 최적화이다. 역시 volatile에는 적절치 않다. 연산이 수행되는 과정에 새로운 값이 읽힐 수도 있기 때문이다.

***std::atomic은 뮤텍스 보호 없이 여러 스레드가 접근하는 자료를 위한 것으로, 동시적 소프트웨어의 작성을 위한 도구다.***

***volatile은 읽기와 기록을 최적화로 제거하지 말아야 하는 메모리를 위한 것으로, 특별한 메모리를 다룰 때 필요한 도구이다.***

[항목23] std::move, std::forward

std::move와 std::forward를 숙지하라.

# std::move

template< class T >

typename std::remove\_reference<T>::type&& move(T&& t);

template< class T >

constexpr typename std::remove\_reference<T>::type&& move(T&& t);

## 정의

std::move는 객체 t가 "이동"될 수 있음을 나타내기 위해 사용됩니다. 즉, t에서 다른 객체로 자원을 효율적으로 전송할 수 있습니다.

std::move는 실제로는 캐스팅에 불과하다. 그래서 이동될 수 있음을 나타내기 위해 사용된다는 말이 여기서 나온다. **std::move라는 것은 실질적인 이동을 수행하는 것이 아니라, 이 객체가 이동 가능하다는 것을 알려주는 역할을 한다.** 이 객체를 인자로 보내어 생성하면 이동할 수 있다는 것이다. **결국, std::move는 rvalue로 캐스팅한다.**

## 구현

아래는 std에 구현된 모습이다.

// TEMPLATE FUNCTION move

template<class \_Ty> inline

typename remove\_reference<\_Ty>::type&&

**move**(\_Ty&& \_Arg) \_NOEXCEPT

{ // forward \_Arg as movable

return (**static\_cast**<typename remove\_reference<\_Ty>::type&&>(\_Arg));

}

// TEMPLATE CLASS remove\_reference

template<class \_Ty>

struct remove\_reference

{ // remove reference

typedef \_Ty type;

};

C++14에서 아래와 같이 직접 구현할 수 있다.

template<typename \_Ty> inline

decltype(auto) myMove(\_Ty&& param)

{

using ReturnType = std::remove\_reference\_t<\_Ty>&&;

return static\_cast<ReturnType>(param);

}

std::remove\_reference\_t 는 remove\_reference<\_Ty>::type 와 같다.

## 동작과정

위의 구현을 아래와 같이 요약할 수 있겠다.

template<class \_Ty>  
r\_Ty&& **move**(\_Ty&& \_Arg)   
{ return **static\_cast**<r\_Ty&&>(\_Arg); }

과정은 이러하다.   
1. 이 함수는 오른값을 인자로 받는다. 한편, 템플릿 매개변수 \_Ty는 형식 연역이 일어난다. 이는 왼값 참조가 될 수 있으며, 오른값 참조가 될 수도 있다. – [항목1]  
2. \_Ty&&은 참조 축약으로 인해 왼값이면 왼값, 오른값이면 오른값이 된다. – [항목25]  
3. 반환값은 오른값으로 고정할 필요가 있다. 방법은 \_Ty의 참조를 없앤다. remove\_reference가 그 역할이다. 이를 \_Ty가 r\_Ty가 된다고 표현한다.  
4. 이를 오른값으로 캐스팅한다.  
5. 캐스팅된 오른값을 반환한다. 그럼 어떠한 인자라도 명백히 오른값을 반환하게 된다.

## 세부사항

정상적으로 오른값의 이동 후보라고 표현할 수 없는 경우도 있다.

### const를 붙이면 이동할 수 없다.

basic\_string(const \_Myt& \_Right)

: \_Mybase(\_Right.\_Getal().select\_on\_container\_copy\_construction())

{ // construct by copying \_Right

\_Tidy();

assign(\_Right, 0, npos);

}

basic\_string(\_Myt&& \_Right) \_NOEXCEPT

: \_Mybase(\_STD move(\_Right.\_Getal()))

{ // construct by moving \_Right

\_Tidy();

\_Assign\_rv(\_STD forward<\_Myt>(\_Right));

}

이동생성자에는 const를 받을 수 없다. 왜냐하면 이동하면 대상은 사라지기 때문에 변하지 않는다고 보장할 수 없기 때문이다. 그렇다고 오류가 되지는 않는다. const \_Myt&을 const오른값 에 묶는 것이 가능하기 때문이다. 그냥 오른값을 복사하게 되는 것이다.

### 객체가 이동 자격을 갖추게 된다는 보장은 없다.

이게 무슨 소리인가? std::move가 이동 가능을 의미한다는 정의와는 어긋난다. 왜냐하면 위에서처럼 오른값을 왼값 참조가 받아버려서 복사해버리면 이는 이동을 나타내지 않는다. **하지만 확실한 것은, 오른값을 반환한다는 점이다.**

class Archer

{

public:

explicit Archer(const std::string \_name);

~Archer();

void merge(const Archer&);

void merge(Archer&&);

private:

std::string myName;

};

Archer::Archer(const std::string \_name)

: myName(std::move(\_name))

{}

Archer a1("ytk"), a2("dk");

a1은 생성자에서 오른값을 받을 것이다. 그리고 std::move는 오른값을 반환할 것이다. 하지만 이 자체가 myName에서 오른값으로 이동생성자를 호출할 수 없다. 왜냐하면 \_name에는 여전히 const한정자가 붙어있다. 이는 \_name이 오른값이건 왼값이건 변경될 수 없다는 것을 의미하는데, 이동연산 자체가 본체의 이동을 요구하므로 이동생성자가 가능하지 않다. 결국 이 원칙은 위의 **const규칙에 기반**한 것이다. **결국 이는 const \_Myt& \_Right복사생성자를 불러온다. 그리고 std::move는 오른값 반환은 보장하지만, 이동을 보장하진 못했다.**

# std::forward<T>

완벽전달이라 한다. 다음과 같이 정의되어 있다.

template< class T >

T&& forward(typename std::remove\_reference<T>::type& t) noexcept;

template< class T >

constexpr T&& forward(typename std::remove\_reference<T>::type& t) noexcept;

template< class T >

T&& forward(typename std::remove\_reference<T>::type&& t) noexcept;

template< class T >

constexpr T&& forward(typename std::remove\_reference<T>::type&& t) noexcept;

## 정의

**rvalue는 rvalue로, lvalue는 lvalue로 전달**해주는 역할을 한다. 이를 가능하게 하는 것은 **조건부 캐스팅**이다.

## 구현

template<class \_Ty> inline

constexpr \_Ty&& forward(

typename remove\_reference<\_Ty>::type& \_Arg) \_NOEXCEPT

{ // forward an lvalue as either an lvalue or an rvalue

return (static\_cast<\_Ty&&>(\_Arg));

}

template<class \_Ty> inline

constexpr \_Ty&& forward(

typename remove\_reference<\_Ty>::type&& \_Arg) \_NOEXCEPT

{ // forward an rvalue as an rvalue

static\_assert(!is\_lvalue\_reference<\_Ty>::value, "bad forward call");

return (static\_cast<\_Ty&&>(\_Arg));

}

std::move와 차이는 해당 인자가 rvalue일 경우에만 오른값으로 캐스팅하여 반환한다는 점이다. 위에서 2가지 모두 static\_cast<\_Ty&&>를 반환하는데, 이는 lvalue를 반환할 수 있으며, rvalue도 반환할 수 있다. **어떻게 이를 가능하게 하는가?** 다음을 보면서 이해해보자.

//lvalue 반환

int b = 5;

int& c = b;

typename std::remove\_reference<decltype(c)>::type& e = c;

int& a = static\_cast<decltype(c)&&>(e); //헐...이게 lvalue라니? – [항목28]

//rvalue 반환

int&& k = 7;

//typename std::remove\_reference<decltype(k)>::type&& p = k; //k는 lvalue.

typename std::remove\_reference<decltype(k)>::type&& p = 7;

int&& q = static\_cast<decltype(k)&&>(p);

위의 과정을 통해 조건부 캐스팅을 진행한다.

그럼 위에서는 \_Ty가 오른값으로 초기화되었는지 왼값으로 초기화되었는지 어떻게 알 수 있을까? 그 답은 \_Ty에 초기화 정보가 부호화되어 있다는 점이다. 이는 [항목28]에서 다룬다.

## 동작과정

이제 위의 구현을 살펴보자.   
1. 무엇을 매개변수로 받는가? 형식 연역한 매개변수의 참조를 제거하여 새로 참조를 붙인 값이다. remove\_reference<\_Ty>::type&은 왼값을 받으며 remove\_reference<\_Ty>::type&&은 오른값을 받을 것이다. 그런데 함수로 전달되는 과정을 보면 모두 &버전을 호출하는 것을 볼 수 있다. 함수 매개변수는 모두 왼값이기 때문이다.  
2. &&로 캐스팅을 진행한다. \_Ty가 왼값이면 & && 🡪 &이며, 오른값이면 && && 🡪 &&이 될 것이다. – [항목25]

3. 결국, std::forward는 템플릿 매개변수를 함께 넘겨받아 이에 맞는 타입을 반환한다.

### 왜 매개변수에 remove\_reference<\_Ty>::type으로 구현되어 있을까?

std::forward구현은 왼값을 받아서 부호화된 타입을 돌려주어야 한다.

template<class \_Ty>  
r\_Ty&& **move**(\_Ty&& \_Arg)   
{ return **static\_cast**<r\_Ty&&>(\_Arg); }

**std::move는 템플릿 매개변수를 받지 않으며, std::forward는 이를 전해주어야 한다.** 이는 std::move는 arg가 왼값일수도, 오른값인지 신경쓰지 않고 받는다(일반적인 보편 참조). 그러므로 왼값, 오른값을 구분할 재량이 없다. 하지만 템플릿 매개변수를 넘김으로써 std::forward는 부호를 그대로 전달할 수 있다. 이렇게 템플릿 변수를 넘기려면 이러한 구현이 필수적이다.

## 세부사항

std::forward가 왼값과 오른값을 유동성있게 반환하는 캐스팅 연산이라면, std::move와 큰 차이가 없지 않은가? 오히려 유연함이 있으니 모든 경우에 std::move대신 std::forward를 사용할 수 있을 것 같다.

이는 명세의 관점에서 답을 구한다. 유연함은 오류의 여지와 명확성을 흐릴 수 있다는 단점을 가지고 있다. 이를 오용하면 안 된다.

**명확하게 오른값으로 변환해서 반환해야 한다면 std::move**를 사용해야 할 것이다. **std::forward는 오른값, 왼값 둘 다 반환**한다는 의미이므로 명세가 완전히 다름을 이해할 수 있다. 의미상으로 둘은 완전히 다른 함수인 것이다. 결국, 내가 하고자 하는 동작을 정확히 하고 이에 맞는 함수를 사용해야 할 것이다.

***std::move는 오른값으로의 무조건 캐스팅을 수행한다. std::move자체는 아무것도 하지 않는다.***

***std::forward는 주어진 인수가 오른값에 묶인 경우에만 그것을 오른값으로 캐스팅한다.***

***std::move와 std::forward 둘 다, 실행시점에서는 아무 일도 하지 않는다.***

[항목24] 오른값 참조, 보편 참조

보편 참조와 오른값 참조를 구분하라.

# 오른값 참조(r-value reference)

일반적으로 생각하는 것이다. **오른값을 복사 아닌 참조한다.**

void buyItemInStore(Archer& buyer, Item&& i)

{

buyer.getItem(std::move(i));

std::cout << "buyer buy item successfully.\n";

}

이 함수가 호출되기 전, 매개변수로 넘어온 i는 오른값이어야만 한다. 그리고 함수 내에서는 왼값을 나타낸다.

# 보편 참조(universal reference)

**왼값이든 오른값이든 모두 참조한다.** 심지어 const, non-const, volatile, non-volatile도 구분하지 않고 참조한다. 이런 참조를 보편 참조라 한다.

## 템플릿의 보편 참조 구현

**구현 방법**은 다음과 같다.   
1. 형식 연역이 발생하도록 함수(클래스 아닌)에 템플릿을 씌어야 한다.  
2. 매개변수 타입은 반드시 T&&이어야 한다.  
3. 이 i를 내부에서 전달할 때에는 std::forward<T>로 전해주어야 한다. 컴파일타임에 i가 무슨 형태인지 알 수 없기 때문이다.

template<typename T>

void buyItemInStore(Archer& buyer, T&& i)

{

buyer.getItem(std::forward<T>(i));

std::cout << "buyer buy item successfully.\n";

}

위에서의 i는 오른값 참조만 받을 수 있었지만, 보편 참조는 Item&, Item&&, const Item&등 모든 경우를 받아들일 수 있다. 그리고 그 형식 자체로 함수에서 사용한다.

### 템플릿인데 보편 참조가 일어나지 않는 경우

**중요한 점은 함수 호출 수준에서 형식 연역이 발생해야 한다는 것**이다.

template<typename T>

class Store

{

public:

void introItem(T&&);

};

얼핏 보면 보편 참조 형태를 갖춘 것 같지만 보편 참조는 일어나지 않는다. T타입은 클래스가 생성될 때 정해진다. 따라서 T의 형식 연역이 일어나지 않는다.

## auto의 보편 참조 구현

구현 방법은 auto&&로 선언하면 된다. 다음과 같이 사용한다.

Item i1;

Item& i2 = i1;

Item&& i3 = Item();

const Item& i4 = i1;

auto&& item1 = i2;

auto&& item2 = i3;

auto&& item3 = i4;

void sellItemInStore(Archer& seller, auto&& i)

{

seller.outItem(std::forward<decltype(i)>(i));

std::cout << "seller sell item successfully.\n";

}

## 보편 참조의 예시: insert vs. emplace

For example, with the traditional insert() function of a std::map, you have to create a temporary, which will then be copied into a std::pair<Key, Value>, which will then be copied into the map :

std::map<int, Complicated> m;

int anInt = 4;

double aDouble = 5.0;

std::string aString = "C++";

// cross your finger so that the optimizer is really good

m.insert(std::make\_pair(4, Complicated(anInt, aDouble, aString)));

// should be easier for the optimizer

m.emplace(4, anInt, aDouble, aString);

insert는 pair을 새로 만들어 이에 원하는 데이터를 복사하여 이의 참조를 보낸다. 하지만 우리는 데이터를 새로 만들 필요가 없다. 집어넣고만 싶다. 이는 오른값을 전달하게 한다.   
하지만 더 핵심적인 부분은 insert는 형식이 정해져 있다는 점이다. 반면, emplace는 형식이 정해져 있지 않다. 이점이 바로 **굳이** Complicated**과 pair을 만들지 않아도 곧바로 map에 집어넣을 수 있게 한다. 여러 개의 인자를 보내면 알아서 형식 연역을 하기 때문이다. – [항목25] 참고**

//push\_back 정의

void push\_back(const value\_type& \_Val) // 왼값 참조 전달

{ // insert element at end

if (\_Inside(\_STD addressof(\_Val)))

{ // push back an element

size\_type \_Idx = \_STD addressof(\_Val) - this->\_Myfirst;

if (this->\_Mylast == this->\_Myend)

\_Reserve(1);

\_Orphan\_range(this->\_Mylast, this->\_Mylast);

this->\_Getal().construct(this->\_Mylast,

this->\_Myfirst[\_Idx]);

++this->\_Mylast;

}

else

{ // push back a non-element

if (this->\_Mylast == this->\_Myend)

\_Reserve(1);

\_Orphan\_range(this->\_Mylast, this->\_Mylast);

this->\_Getal().construct(this->\_Mylast,

\_Val);

++this->\_Mylast;

}

}

void push\_back(value\_type&& val); // 위와 비슷함

// emplace\_back 정의

template<class... \_Valty>

void emplace\_back(\_Valty&&... \_Val)

{ // insert by moving into element at end

if (this->\_Mylast == this->\_Myend)

\_Reserve(1);

\_Orphan\_range(this->\_Mylast, this->\_Mylast);

this->\_Getal().construct(this->\_Mylast,

\_STD forward<\_Valty>(\_Val)...);

++this->\_Mylast;

}

## 보편 참조의 원리

std::forward에서처럼 이 원리를 설명하는 핵심은 참조 축약에 있다. – [항목28] 참고

***함수 템플릿 매개변수의 형식이 T&& 형태이고 T가 연역된다면, 또는 객체를 auto&&로 선언했다면, 그 매개변수나 객체는 보편 참조이다.***

***형식 선언의 형태가 정확히 형식&&가 아니면, 또는 형식 연역이 일어나지 않으면, 형식&&는 오른값 참조를 뜻한다.***

***오른값으로 초기화되는 보편 참조는 오른값 참조에 해당한다. 왼값으로 초기화되는 보편 참조는 왼값 참조에 해당한다.***

[항목25] 올바른 참조 구분

오른값 참조에는 std::move를, 보편 참조에는 std::forward를 사용하라.

# 왼값과 오른값의 전달

왼값은 복사를 하며, 오른값은 이동을 해야 한다. 다음은 생성자를 통하여 인자가 어떻게 전달되는지를 보여준다.

class Archer

{

public:

Archer() {}

Archer(std::string \_name, int \_level)

: name(\_name)

, level(\_level) {}

Archer(const Archer& \_other)

: name(\_other.name)

, level(\_other.level) {}

Archer(Archer&& \_other)

: name(std::move(\_other.name))

, level(std::move(\_other.level)) {}

private:

std::string name;

int level;

};

Archer a1("ytk", 9999);

Archer a2(a1);

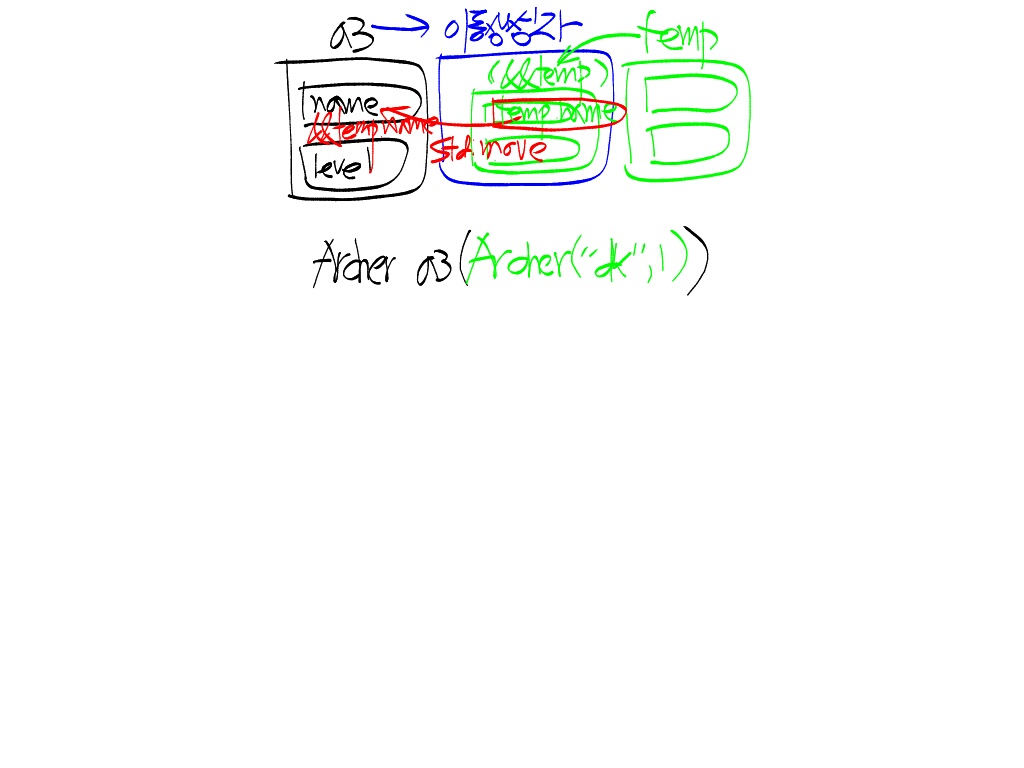
Archer a3(Archer("dk", 1));

name이라는 string타입의 변수를 보자. 복사 생성자에서는 name을 복사해야 한다. 그리고 이동 생성자에서는 name을 이동할 뿐이다. **이런 차이가 생기는 근본적인 이유는 오른값은 임시 값이며, 곧 파괴되므로 기존 값은 사라져도 좋다는 전제가 있기 때문**이다.

## 오른값의 전달

이동을 하기 위해서는 오른값으로 바꿔주어야 한다. 왼값으로 이동하고 싶다고? 마음은 알겠지만 컴파일러는 왼값을 주면 복사 생성자를 불러온다. **오른값을 바꾸어주는 이유는 이동 생성자, 이동 대입 연산자를 불러오기 위함이다. 이는 std::move가 가능**하게 한다. – [항목23] 참고

위의 이동 생성자에서의 이동 관계를 그림으로 살펴본다.



과정은 다음과 같다.  
1. 이동 생성자로 &&temp를 전해준다. **오른값을 전해야 이동 생성자를 불러온다.**  
2. **이동 생성자를 다시 이동**해야 한다. **역시 temp의 name을 오른값으로 바꿔주어야 한다. 이를 std::move로 한다.**3. 만약에 std::move를 사용하지 않으면 name은 왼값으로 취급되어 복사될 것이다.  
4. 반대로 왼값으로 전해진 것을 오른값으로 바꾸면 어떻게 될까? 왼값은 이동되며, 원본은 사라질 것이다. 이는 우리가 원하던 결과는 아닐 것이다. 물론 위에서는 왼값을 매개변수로 받을 수는 없긴 하다. 하지만 이런 상황은 아래 보편 참조에서 생긴다.

### std::move vs. std::move\_if\_noexcept

template< class T >

typename std::conditional<

!std::is\_nothrow\_move\_constructible<T>::value && std::is\_copy\_constructible<T>::value,

const T&, T&&>::type move\_if\_noexcept(T& x);

template< class T >

constexpr typename std::conditional<

!std::is\_nothrow\_move\_constructible<T>::value && std::is\_copy\_constructible<T>::value,

const T&, T&&

>::type move\_if\_noexcept(T& x);

**move\_if\_noexcept obtains an rvalue reference to its argument if its move constructor does not throw exceptions, otherwise obtains an lvalue reference to its argument.** It is typically used to combine move semantics with strong exception guarantee. – [항목14] 참고

## 보편 전달

위의 과정은 매개변수가 &&이어야만 한다. 때로는 왼값도 오른값도 받고 싶을 것이다. 이때는 보편 참조와 std::forward가 적절하다.

template<typename T>

void copyName(T&& \_otherName)

{

//name = std::forward<T>(\_other).name;

//위처럼 하면 \_other은 이동되지만, name은 이동되지 않습니다.

name = std::forward<T>(\_otherName);

}

위의 과정 중, 4번에 의해서 왼값은 왼값으로 넘겨줄 필요가 있다. std::move를 사용하면 오른값에는 잘 작동하지만 왼값에서는 미지의 작동을 할 텐데 어찌할 것인가? **왼값은 왼값으로 전달하고 오른값은 오른값으로 전달하는 std::forward를 사용**하면 이를 가능하게 한다.

## 왼값과 오른값 전달의 중복적재

보편참조는 혼란스럽다며 아래와 같이 할 수도 있다.

void setName(const std::string& \_otherName)

{

name = \_otherName;

}

void setName(std::string&& \_otherName)

{

name = std::move(\_otherName);

}

물론 아주 잘 작동한다. std::vector.push\_back도 이렇게 구현되어 있다. 그럼 이 중복적재와 보편참조의 차이를 살펴보자.

### 코드 양 증가 + 효율성 하락

a3.setName("ultra ytk");

코드 양이 증가하는 것은 눈으로 보면 알 것이다.

위와 같은 식에서 매개변수의 문자열은 무엇인가? 중복적재는 이를 string으로 만들어서 넘겨주어야 하며, 보편 참조는 그럴 필요가 없다. **중요한 원칙은 무엇일까? 보편 참조는 string으로 명시한 적이 없다.** string일 필요가 없으며 무엇이든 전달받은 타입T에서 string으로 이동시켜주면 된다. 반면, 중복 적재는 그렇지 않다는 것이다. 중복 적재는 string을 만드는 생성자 한번과 소멸자 한번이 추가된다. 이건 타락한 이벤트와 같다.

### 가변성(scalability)이 나쁘다.

간단하게 말해서, 매개변수가 여러 개일 때, 중복 적재는 각각의 경우를 모두 고려해야 하지만, 보편 참조는 이를 모두 전달할 수 있다. 딱 위의 예시의 insert vs. emplace의 차이다.

// cross your finger so that the optimizer is really good

m.insert(std::make\_pair(4, Complicated(anInt, aDouble, aString)));

// should be easier for the optimizer

m.emplace(4, anInt, aDouble, aString);

**emplace는 여러 개의 매개변수를 타입과 개수에 상관없이 전달하는 모습이 보인다.** 게다가 이는 객체를 생성할 필요도 없다. 인자를 곧바로 map에 집어넣기 때문이다.

## 오른값 참조와 보편 참조는 가장 마지막에 수행해야 한다.

이유는 전달 후에 내용이 바뀌기 때문이다. 이동했으니까 말이다.

template<typename T>

void setSignTest(T&& text)

{

auto start = std::chrono::system\_clock::now();

log.setText(text); //만약 text를 이동하면 이후 text는 사라진다.

auto end = std::chrono::system\_clock::now();

std::chrono::duration<double> diff = end - start;

log.setText(std::forward<T>(text));

}

# 함수가 결과를 값으로 반환할 경우

함수의 결과로 참조가 아닌 값으로 전달할 경우, 상황에 따라 다음 규칙을 따른다.

## 오른값 참조나 보편 참조로 묶인 객체 반환의 경우

즉, 함수에서 만든 객체가 아닌 오른값으로 넘어온 매개변수를 반환함을 의미한다. return문에서 std::move나 std::forward를 사용하는 것이 바람직하다.

friend Archer operator+(Archer&& lhs, const Archer& rhs)

{

lhs.level += rhs.level;

return std::move(lhs); //std::move가 없다면 복사하여 전달해야 한다.

}

**반환 역시 다른 위치로 전달하는 것**이므로 오른값으로 전달하려면 std::move가 필요하다. 중요한 점은, 여기서는 Archer가 이동을 지원하지 않아도 해가 없다. 복사 생성자를 호출해 복사하면 되기 때문이다.

**핵심은 반환을 값으로 하는 경우, std::move나 std::forward를 사용하여 이동할 수 있으면 이동하라는 것이다.** 만약 이동할 수 없어도 복사하면 되므로 해가 없어 아주 적절하다는 것이다.

## 지역객체를 값으로 반환할 경우

함수가 지역객체를 값으로 반환한다면 위의 경우는 적용되지 않는다. 반환값 최적화가 가능하기 때문이다.

### 반환값 최적화(return value optimization, RVO)

아래처럼 하는 것도 멍청하고 위험한 짓이다.

// 1.

const Rational\* operator \* (const Rational &lhs, const Rational &rhs);

// 2.

const Rational& operator \* (const Rational &lhs, const Rational &rhs);

함수 안에서 파괴되는 것을 어찌 참조한단 말인가?

객체를 값으로 반환하는 함수에 대해 원론적으로 임시 객체 생성을 피할 수는 없다.

다만, 최신의 컴파일러들이 지원하는 RVO 최적화를 이용하는 길이 있을 뿐이다.

최적화를 이용하는 그 방법은 바로 객체 대신에 객체 생성자 인자를 반환하는 것이다.

inline const Rational operator \* (const Rational &lhs, const Rational &rhs)

{

return Rational(lhs.numerator \* rhs.numerator,

lhs.denominator \* rhs.denominator);

}

얼핏 보기에 위 코드가 무슨 이득이 있나 싶다.

operator \* 안에서 객체가 1번 만들어지고 다시 이것을 반환하면서 임시 객체까지 만들어지니

오히려 손해일 것 같다는 느낌이 팍팍 드는데 말이다.

**하지만, 컴파일러는 반환 시 임시 객체를 없애고,**

**계산 결과값을 반환 값을 받는 객체에 대해 할당된 메모리에 직접 넣어 초기화해 준다.**

**결국 생성자 한번, 소멸자 한번의 호출 비용만 들어가는 것이다.**

이것이 바로 ***RVO(Return Value Optimization)***인 것이다.

이러한 **복사 제거(copy elision)**는  
1. 그 지역 객체의 형식이 함수의 반환 형식과 같아야 하고  
2. 그 지역 객체가 바로 함수의 반환 값이어야 한다.

*Q. 만약 반환 경로가 복잡하여 적절한 지역 객체를 파악하지 못하는 경우, 컴파일러가 최적화를 구현하지 못할 수 있으므로 이동을 하는 게 낫지 않을까?*

그렇지 않다. 컴파일러는 반환값 최적화를 구현할 수 없는 경우, **암묵적으로 객체를 오른값으로 취급하게 되어있다.**

**결론은, 이동 연산자는 이러한 조건에 만족하지 못하므로 최적화를 돕기는커녕, 최적화 여지를 제한하는 꼴이 된다.**

### 함수 내에서 생성한 객체를 반환하는 경우

위에서 살펴본 경우다. 반환값 최적화 기법이 적용된다.

### 값으로 전달된 매개변수를 반환하는 경우

함수의 반환값과 형식이 같은 값 전달 방식의 매개변수를 반환하는 경우다. 매개변수의 경우에는 최적화 수행 대상이 아니지만, 이는 암묵적으로 객체를 오른값으로 취급한다.

Archer getFriend(Archer a)

{

std::cout << a.getName() << "은 나의 친구\n";

return a;

//return std::move(a);와 동일하게 처리

}

***오른값 참조나 보편 참조가 마지막으로 쓰이는 지점에서, 오른값 참조에는 std::move를, 보편 참조에는 std::forward를 사용해라.***

***결과를 값 전달 방식으로 돌려주는 함수가 오른값 참조나 보편 참조를 돌려줄 때에도 각각 std::move, std::forward를 적용하라.***

***반환값 최적화의 대상이 될 수 있는 지역 객체에는 절대로 std::move나 std::forward를 적용하지 말아야 한다.***

[항목26] 보편 참조의 중복적재 문제

보편 참조에 대한 중복적재를 피하라.

# 보편 참조와 중복적재

어떠한 함수에 대해 보편 참조하는 함수와 중복적재할 함수를 만든다. 이러한 상황은 아래와 같다.

std::unordered\_map<int, std::string> dictionary;

class Archer

{

public:

template<typename T>

void setItem(T&& item) //보편 참조 중복적재

{

store.emplace(item);

}

void setItem(int index) //int 중복적재

{

auto&& item = dictionary[index];

if (!item.empty())

{

store.emplace(item);

}

}

private:

std::unordered\_set<std::string> store;

};

int main()

{

std::string i1("masterPlan");

Archer a1;

a1.setItem(i1); // 1

a1.setItem(std::string("dragonHeart")); // 2

a1.setItem("motherEye"); // 3

a1.setItem(1); // 4

return 0;

}

4가지 경우의 함수를 호출한다. store.emplace(item);에서 각각의 경우에 어떻게 작동하는지 살펴본다.

## [1] 왼값 전달

왼값을 전달하므로 복사 생성자를 피해갈 수 없다. 복사하지 않으면 기존 값이 날아가버릴 테니 말이다. 즉, 복사하여 임시 객체를 만들고 이 객체를 자료구조에 넣는다.

## [2] 오른값 전달

오른값이므로 이동 생성자로 임시 객체를 만들 수 있다. 그리고 이 임시 객체를 자료구조에 넣는다.

## [3] 문자열 리터럴

이는 문자열을 직접 a1객체에 전달한다. string객체를 생성할 필요도 없다. 문자열만 전달하면 되므로 std::string의 복사는 커녕 이동 생성자도 호출할 필요가 없다. 보편 참조에 인자가 문자열 리터럴로 추론되어 이를 호출한다.

## [4] 중복적재된 int전달

이는 문자열을 알 수 없을 때, int로 함수를 호출하기 위해 제작될 것이다. 그러므로 int형 매개변수를 갖는 setItem함수를 호출할 것이다.

# 중복 적재 시, 발생하는 문제

보편참조는 대부분의 인수를 빨아들인다. 이로 인해 여러 가지 문제가 발생한다. 구체적으로 아래와 같은 경우에 문제가 발생한다.

## 승격(short->int)을 해야 하는 경우

setItem(int index)

a1.setItem(short(1));

//error C2664: 인수 1을(를) 'short'에서 'const std::basic\_string<...> &'(으)로 변환할 수 없습니다.

놀랍게도 이 함수에는 short를 받을 수 없다. short는 보편 참조가 받을 것이다. **왜 이런 현상이 생길까? short->int의 승격보다 T가 short로 연역되는 규칙이 우선시**되어 보편 참조 중복적재가 호출되는 것이다. 보편참조는 욕심이 굉장히 많은 함수인 것이다.

## 완벽전달 생성자 만들기

string자체를 넘기지 말고 이를 담는 하나의 객체를 만드는 경우가 있을 수 있다. 완벽 전달을 하는 생성자와 int매개변수 생성자를 중복적재하면 왠지 short를 int로 승격시켜줄 것 같다는 멍청한 생각은 다음과 같다.

class Item

{

public:

template<typename T>

explicit Item(T&& item)

: name(std::forward<T>(item)) {}

explicit Item(int index)

: name(dictionary[index]) {}

std::string getName() const { return name; }

private:

std::string name;

};

a1.setItem(Item("chocolate").getName());

a1.setItem(Item(1).getName());

a1.setItem(Item(short(1)).getName()); //역시 error.

### 문제1. 이전과 똑 같은 문제인 연역이 승격보다 우선 처리된다.

아무런 해결의 진전을 보이지 않는다.

### 문제2. 복사 생성자의 호출

오히려 아까보다 문제가 늘었다. Item을 복사할 복사 생성자도 보편참조가 빨아먹는다. 보편참조는 아래와 같기 때문이다.

explicit Item(Item& item)

: name(Item& item)) {} //보편참조가 추론한 생성자

Item(const Item&); //컴파일러가 만든 복사생성자

그리고 아래를 수행한다.

Item i1(masterPlan);

Item i2(i1);

i1은 const가 없는 상태다. 컴파일러는 더 가까운 형식을 제공하는 보편참조 생성자를 호출할 것이다. 그래서 아래와 같이 생각할 수도 있다.

“그럼 const를 붙여서 호출하면 되지.”

const Item i2("candy");

Item i3(i2);

아래는 컴파일이 된다. ***‘C++의 중복적재 해소 규칙에는 어떤 함수 호출이 템플릿 인스턴스와 비템플릿 함수에 똑같이 부합한다면 비템플릿 함수를 우선시한다.’***라는 규칙에 의해 성공되는 것이다.

**이로써 복사 생성자와 보편 참조 생성자의 문제는 해결될 수 있다.**

### 문제3. 상속에서의 완벽 전달 생성자

시작부터 문제가 발생한다.

class EquipmentItem : public Item

{

public:

EquipmentItem(const EquipmentItem& item)

: Item(item) {}

EquipmentItem(EquipmentItem&& item)

: Item(std::move(item)) {}

};

위는 컴파일되지 않는다. Item 에 EquipmentItem 을 넘겨주어야 한다. 하지만 여기서 불러온 생성자는 완벽 전달 생성자가 호출되기 때문에 Item 을 인자로 받을 수 없다. 이유는 위에서 살펴본 승격과 연역의 관계와 비슷하다. Item 이 EquipmentItem 로 바뀌어야 하는데 연역이 우선 처리되기 때문이다. 결국 이 문제는 해결될 수 없다.

## 이 문제를 어떻게 해결할 것인가?

[항목27]에서 살펴볼 것이다.

***보편 참조에 대한 중복적재는 거의 항상 보편 참조 중복적재 버전이 예상보다 자주 호출되는 상황으로 이어진다.***

***완벽 전달 생성자들은 특히나 문제가 많다. 그런 생성자들은 대체로 비const왼값에 대한 복사 생성자보다 더 나은 부합이며, 기반 클래스 복사 및 이동 생성자들에 대한 파생 클래스의 호출들을 가로챌 수 있기 때문이다.***

[항목27] 보편 참조의 중복적재 해결

보편 참조에 대한 중복적재 다신 사용할 수 있는 기법들을 알아 두라.

# 중복적재 시, 발생하는 문제 해결

[항목26]에서는 중복적재로 발생하는 문제들을 살펴보았다. 이제 이를 해결할 시간이다.

## 중복적재를 포기한다.

아예 함수를 나누어서 선언한다. 하지만 이는 해결책이 아니다.

## const T& 매개변수를 사용한다.

보편 참조를 사용하지 말고 왼값, 오른값을 각각 받을 함수를 오버로딩한다. 이전 스타일의 방법이다. 약간의 효율성 문제는 있지만, 이상한 매개변수를 받는 보편 참조의 문제점을 해결할 수 있다.

## 값 전달 방식의 매개변수를 사용한다.

복사될 것이 확실할 때에는, 값으로의 전달을 고려하라. – [항목41] 참고

class Item

{

public:

/\*template<typename T>

explicit Item(T&& item)

: name(std::forward<T>(item)) {}\*/

explicit Item(int index)

: name(dictionary[index]) {}

explicit Item(std::string item)

: name(item) {}

std::string getName() const { return name; }

private:

std::string name;

};

## 꼬리표 배분을 사용한다.

보편 참조를 피하지 않고 중복적재된 함수의 호출을 둘 다 사용하려면 어떻게 해야 할까? 해결책은 꼬리표라는 하나의 인자를 더 넣는다. 이는 확실한 중복적재가 일어나게 한다.

### std::is\_integral: 전달한 형식이 정수인지 아닌지를 판단한다.

template< class T >

struct is\_integral;

Checks whether T is an integral type. Provides the member constant value which is equal to true, if T is the type bool, char, char16\_t, char32\_t, wchar\_t, short, int, long, long long, or any implementation-defined extended integer types, including any signed, unsigned, and cv-qualified variants. Otherwise, value is equal to false.

이를 이용하여 다음과 같이 한다.

class Archer

{

public:

template<typename T>

void setItem(T&& item)

{

setItem**Impl**(std::forward<T>(item), std::is\_integral<T>());

}

template<typename T>

void setItemImpl(T&& item, std::false\_type) //보편 참조 중복적재

{

store.emplace(item);

}

void setItemImpl(int index, std::true\_type) //int 중복적재

{

auto&& item = dictionary[index];

if (!item.empty())

{

store.emplace(item);

}

}

private:

std::unordered\_set<std::string> store;

};

typedef integral\_constant<bool, true> true\_type; //이것이 꼬리표이다.

typedef integral\_constant<bool, false> false\_type;

위 타입은 참, 거짓에 해당하는 어떤 형식을 인자로 받는다는 것이다. 참, 거짓은 런타임에 수행되므로 컴파일타임에 어떤 함수를 사용할 지 판단할 수 없다. **이는 런타임 결과를 가지고 함수를 선택할 수 있게 해준다.**

이것으로 중복적재된 함수를 참, 거짓으로 판단하고 어떠한 함수를 사용할 것인지를 판단할 수 있다.

### int&로 연역되는 경우, int와 구분되어야 한다.

T로 전달받은 lvalue는 int&로 연역된다. 그러므로 int를 전달했는데 참조를 전달하게 되는 현상이 생긴다.

template<typename T>

void setItem(T&& item)

{

setItemImpl(std::forward<T>(item),

std::is\_integral<typename std::remove\_reference\_t<T>>());

}

이는 간단히 해결할 수 있다. 모든 참조 한정자를 제거할 수 있는 형질 특질인 typename std::remove\_reference\_t<T>을 사용하는 것이다.

## 보편 참조를 받는 템플릿을 제한한다.

T가 되는 후보들을 적절히 제한할 수 있는 기법이 존재한다.

### std::enable\_if: 컴파일 타임에 조건을 검사한다.

enable\_if을 이용하면 특정 템플릿을 활성화/비활성화시킬 수 있다. enable\_if안의 조건이 만족할 때 만이 템플릿이 활성화된다.

template<bool \_Test,

class \_Ty = void>

struct enable\_if

{ // type is undefined for assumed !\_Test

};

template< bool B, class T = void >

struct enable\_if;

If B is true, **std::enable\_if** has a public member typedef type, equal to T; otherwise, there is no member typedef.

This metafunction is a convenient way to leverage [SFINAE](http://en.cppreference.com/w/cpp/language/sfinae) to conditionally remove functions from [overload resolution](http://en.cppreference.com/w/cpp/language/overload_resolution) based on type traits and to provide separate function overloads and specializations for different type traits. **std::enable\_if** can be used as an additional function argument (not applicable to operator overloads), as a return type (not applicable to constructors and destructors), or as a class template or function template parameter.

이 메타함수는 ***SFINAE***로 인하여 작동한다.

### **SFINAE(Substiution-Failuer-Is-Not-An-Error, 치환실패는 오류가 아님)**

참고: <https://github.com/jwvg0425/ModernCppStudy/wiki/SFINAE>

#### 정의

Substitution은 앞에서 말한 Template Argument Substitution(템플릿 타입을 추론하고 그 추론한 타입을 대입하여 템플릿 인스턴스를 만드는 과정)을 가리키는 것이다. 즉, Template Argument Substitution을 수행하는 과정에서 Substitution이 실패했다 하더라도 그건 컴파일 에러가 아니라는 것이다.

#### 동작과정

예제를 통해 이게 무슨 의미인지 살펴보자.

template<typename T>

void f(typename T::type a);

template<typename T>

void f(T a);

f<int>(0); //두번째 템플릿 함수 호출

class Widget

{

public:

using type = char;

};

char k;

f<Widget>(k); //첫번째 템플릿 함수 호출

void f(int::type a);  
void f(int a);

f(int::type a); 이게 문법적으로 말이 되는가? 이 substitution은 정상적인 substitution이 아니다. Substitution에 **실패(failure)**한 것이다. 여기서 SFINAE의 의미가 드러난다. SFINAE는 Substitution의 실패가 에러가 아님을 뜻하는 말이라고 했다. 즉, 이와 같은 경우에 컴파일러는 **이를 에러로 처리하지 않고 단순히 overload set(오버로딩에서 호출될 수 있는 후보 함수 목록)으로부터** **substitution에 실패한 함수를 제외해버린다.** 이로 인해 실제 호출될 수 있는 후보 함수 목록에는void f(int a); 하나만이 남는 것이다. 따라서 컴파일러는 두 번째 함수를 호출하게 된다.

### Type SFINAE

SFINAE는 크게 나누면 Type SFINAE와 Expression SFINAE로 나눌 수 있다. Substitution을 할 때 어느 부분에서 실패를 했는지에 따라 분류를 하는 것이다.

#### 스코프 지정 연산(::)에서 지정된 스코프 내에 존재하지 않는 타입을 사용한 경우

위 예제에서 다룬 케이스다.

template <typename T> int f(typename T::B\*);

template <typename T> int f(T);

int i = f<int>(0); // 두번째 오버로딩 사용

f<int>(0);에서 int::B\*는 int 스코프 내에 B 타입이 존재하지 않으므로 Substitution Failure를 일으키게 된다. 이 경우 첫 번째 함수 템플릿은 **무시(overload set에서 제외)**된다.

#### T의 멤버에 대한 포인터를 만들려고 시도했는데, T가 클래스 타입이 아닌 경우

template<typename T>

class is\_class {

typedef char yes[1];

typedef char no [2];

template<typename C> static yes& test(int C::\*); // C가 클래스인 경우 선택

template<typename C> static no& test(...); // 그 외의 경우 선택

public:

static bool const value = sizeof(test<T>(0)) == sizeof(yes);

};

class Widget {};

bool test1 = is\_class<int>::value; //false

bool test2 = is\_class<Widget>::value; //true

SFINAE의 유용한 활용 중 하나가 될 수 있는 예제다. is\_class::value 에서 value가 어떤 식으로 결정되는지 한 번 짚어보자. 우선 타입 T가 int이므로 test<T>(0)은 test<int>(0);이 된다. 여기서 substitution 과정을 거치면, 두 개의 멤버 함수 템플릿은

static yes& test(int int::\*);

static no& test(...);

이 된다. 여기서 첫 번째, int int::\* 에서 int는 클래스가 아니므로 멤버에 대한 포인터를 만들 수 없다. 여기서 substitution failure가 일어나고, 두 번째 함수가 선택되는 것이다(가변 인수 함수는 모든 인자를 받아들일 수 있으므로, 첫 번째에서 실패하면 무조건 두 번째가 선택된다). 그 결과 test<int>(0)의 리턴 값은 no&가 되는데, no 타입의 size와 yes 타입의 size는 서로 다르므로 value 값은 false가 되는 것이다. 반면 test<Widget>(0)의 경우 int Widget::\*은 Widget의 멤버에 대한 타당한 포인터이므로 substitution failure가 일어나지 않는다. 이 때 가변 인수 함수보다는 그렇지 않은 함수가 오버로딩에서 더 우선순위가 높으므로 value 값이 true가 나오게 되는 것이다.

### Expression SFINAE

Expression SFINAE는 expression의 형태가 이상할 때 발생한다. 이 역시 간단한 예제를 살펴보는 것으로 이해하고 넘어가자.

#### 함수 타입에 잘못된 표현식이 사용된 경우

struct X {};

struct Y { Y(X){} }; // X 는 Y로 변환 가능

template <class T>

auto f(T t1, T t2) -> decltype(t1 + t2); // 오버로딩 함수 1

X f(Y, Y); // 오버로딩 함수 2

X x1, x2;

X x3 = f(x1, x2); // 오버로딩 함수 1에서 SFINAE 발생(x1 + x2는 잘못된 형태의 expression)

// 따라서 오버로딩 함수 2가 호출됨

### std::enable\_if와 SFINAE

SFINAE 개념이 잘 활용되는 케이스로 std::enable\_if가 있다.

template<bool B, class T = void>

struct enable\_if

{};

template<class T>

struct enable\_if<true, T>

{

using type = T;

};

std::enable\_if가 실제로 어떤 형태로 활용되는지는 [item 27](https://github.com/jwvg0425/ModernCppStudy/wiki/item-27)에 잘 설명되어 있다. 간단히 요약하자면 위 Type SFINAE에서 **스코프 지정 연산(::)에서 지정된 스코프 내에 존재하지 않는 타입을 사용한 경우**를 활용하여 enable\_if<true>::type은 존재하지만 enable\_if<false>::type은 존재하지 않으므로 이를 바탕으로 특정한 경우에만 함수 템플릿을 사용할 수 있도록 만드는 것이다.

template<typename T,

typename = typename std::is\_enable<is\_class<T>::value>::type>

void f(T a);

예를 들어 위와 같이 f 함수를 선언할 경우, T 타입이 클래스 타입일 때만 Substitution이 성공하여 f 함수를 쓸 수 있게 된다(만약 T가 클래스가 아니라면 is\_class::value는 false가 되고, std::is\_enable::type은 존재하지 않으므로 Substitution이 실패하게 된다). 그 외의 경우는 SFINAE에 의해 위 함수 템플릿 자체가 무시되므로 템플릿의 사용에 일종의 제약을 걸 수 있게 되는 것이다.

### std::is\_same: 같은 타입인지 확인한다.

// TEMPLATE CLASS is\_same

template<class \_Ty1, class \_Ty2>

struct is\_same

: false\_type

{ // determine whether \_Ty1 and \_Ty2 are the same type

};

template< class T, class U >

struct is\_same;

If T and U name the same type with the same const-volatile qualifications, provides the member constant value equal to true. Otherwise value is false.

### std::decay: 모든 참조와 cv한정자(const, volatile)를 제거한다.

// TEMPLATE CLASS decay

template<class \_Ty>

struct decay

{ // determines decayed version of \_Ty

typedef typename remove\_reference<\_Ty>::type \_Ty1;

typedef typename \_If<is\_array<\_Ty1>::value,

typename remove\_extent<\_Ty1>::type \*,

typename \_If<is\_function<\_Ty1>::value,

typename add\_pointer<\_Ty1>::type,

typename remove\_cv<\_Ty1>::type>::type>::type type;

};

template< class T >

struct decay;

Applies lvalue-to-rvalue, array-to-pointer, and function-to-pointer implicit conversions to the type T, removes cv-qualifiers, and defines the resulting type as the member typedef type. Formally:

* If T names the type "array of U" or "reference to array of U", the member typedef type is U\*.
* Otherwise, if T is a function type F or a reference thereto, the member typedef type is [std::add\_pointer](http://en.cppreference.com/w/cpp/types/add_pointer)<F>::type.
* Otherwise, the member typedef type is [std::remove\_cv](http://en.cppreference.com/w/cpp/types/remove_cv)<[std::remove\_reference](http://en.cppreference.com/w/cpp/types/remove_reference)<T>::type>::type.

These conversions model the type conversion applied to all function arguments when passed by value.

이를 모두 적용하여 다음과 같이 표현할 수 있다.

class Item

{

public:

template<

typename T,

typename = typename std::enable\_if<

!std::is\_same<Item,

typename std::decay<T>::type //decay<T>: T의 한정자를 제거한다.

>::value //제거한 한정자가 Item과 타입이 같은지 값을 반환한다.

>::type // enable\_if<조건>: 조건의 값을 가지는 타입

>

explicit Item(T&& item)

: name(std::forward<T>(item)) {}

explicit Item(int index)

: name(dictionary[index]) {}

/\*explicit Item(std::string item)

: name(item) {}\*/

std::string getName() const { return name; }

private:

std::string name;

};

## 파생 클래스가 기본 클래스 완벽전달 생성자를 부르는 경우

문제는 파생클래스에서 기본클래스의 생성자를 불러온다는 점인데, 이로 인해서 파생 클래스 타입과 기본 클래스 타입을 비교하게 된다. 이를 어떻게 비교할까?

class Item

{

public:

template<

typename T,

typename = typename std::enable\_if<

!std::is\_base\_of<Item,

typename std::decay<T>::type //decay<T>: T의 한정자를 제거한다.

>::value //제거한 한정자가 Item과 타입이 같은지 값을 반환한다.

>::type // enable\_if<조건>: 조건의 값을 가지는 타입

>

explicit Item(T&& item)

: name(std::forward<T>(item)) {}

explicit Item(int index)

: name(dictionary[index]) {}

std::string getName() const { return name; }

private:

std::string name;

};

class EquipmentItem : public Item

{

public:

EquipmentItem(const EquipmentItem& item)

: Item(item) {}

EquipmentItem(EquipmentItem&& item)

: Item(std::move(item)) {}

};

## 복사, 이동 생성자와 정수 형식도 같이 구분하기

위에서 접근한 내용들을 합치는 것이다.

template<

typename T,

typename = typename std::enable\_if\_t<

!std::is\_base\_of<Item, typename std::decay\_t<T>>::value

&&

!std::is\_integral<std::remove\_reference\_t<T>>::value

> // enable\_if<조건>: 조건의 값을 가지는 타입

>

explicit Item(T&& item)

: name(std::forward<T>(item)) {}

정수형인지 아닌지 검사하는 코드만 추가해주면 된다.

# 보편 참조에서 오류를 검출하는 법

유효하지 않은 인수를 전달할 때, 보여주는 오류 메시지는 굉장히 보기 어렵다. 어떤 오류인지 구별하기 어려울 정도로 말이다. static\_assert로 컴파일 타임에 검사하면 된다.

### Static Assertion

기능: Performs **compile-time** assertion checking

A static assert declaration may appear at namespace and block scope (as a [block declaration](http://en.cppreference.com/w/cpp/language/declarations)) and inside a class body (as a [member declaration](http://en.cppreference.com/w/cpp/language/class))

If *bool\_constexpr* returns true, this declaration has no effect. Otherwise a compile-time error is issued, and the text of *message*, if any, is included in the diagnostic message.

template<

typename T,

typename = typename std::enable\_if\_t<

!std::is\_base\_of<Item, typename std::decay\_t<T>>::value

&&

!std::is\_integral<std::remove\_reference\_t<T>>::value

> // enable\_if<조건>: 조건의 값을 가지는 타입

>

explicit Item(T&& item)

: name(std::forward<T>(item))

{

//T으로 string을 생성할 수 있는지 검사한다.

static\_assert(

std::is\_constructible<std::string, T>::value,

"Parameter item can't be used to construct a std::string!"

);

}

### std::is\_constructible

template <class T, class... Args> struct is\_constructible;

Trait class that identifies whether *T* is a *constructible type* with the set of argument types specified by Arg.

***보편 참조와 중복적재의 조합에 대한 대안으로는 구별되는 함수 이름 사용, 매개변수를 const에 대한 왼값 참조 전달, 매개변수를 값으로 전달, 꼬리표 배분 사용 등이 있다.***

***std::enable\_if를 이용해서 템플릿의 인스턴스화를 제한함으로써 보편 참조와 중복적재를 함께 사용할 수 있다. std::enable\_if는 컴파일러가 보편 참조 중복적재를 사용하는 조건을 프로그래머가 직접 제어하는 용도로 쓰인다.***

***보편 참조 매개변수는 효율성 면에서 장점인 경우가 많지만, 대체로 사용성 면에서는 단점이 된다.***

[항목28] 참조 축약

참조 축약을 숙지하라.

# 연역된 형식의 부호화

인수가 템플릿에 전달되었을 때 템플릿 매개변수에 대해 연역된 형식에는 그 인수가 원래 왼값이었는지 아니면 오른값이었는지에 대한 정보를 기억한다. 이것을 부호화한다고 한다. 무슨 말인가 하면,

template<typename T>

void func(T&& args);

여기서 컴파일러는 T가 무엇인지 연역해낸다. 그리고 부호화된 이 T를 보고 인수 원래의 왼값, 오른값을 구분할 수 있다.

## 부호화의 매커니즘

args의 기억한 정보를 활용한다. 이 인수가 왼값이면 T는 왼값 참조로 연역된다. 오른값이면 T는 비참조로 연역된다.

template<typename T> // T: Archer&

void func(Archer& args)

template<typename T> // T: Archer

void func(Archer&& args)

근데 이게 왜 필요할까? 일단 참조 축약의 정체를 먼저 밝히고, 존재의 의미를 다룬다. 아래에서 살펴보자.

# 참조 축약(reference collapsing)이 뭘까?

일반적으로 참조에 대한 참조는 위법이다.

int& &&a = a1; // error

int&& &&b = 7; // error

## 참조 축약을 허용하는 문맥

놀랍게도 위의 코드를 허용하는 문맥이 있다. &-&&, &&-&&처럼 두 참조가 하나의 참조로 압축되는데, 이것이 참조 축약이다. 다음과 같다.

### 참조 축약의 규칙

만일 두 참조 중 하나라도 왼값 참조이면 결과는 왼값 참조이다. 그렇지 않으면 결과는 오른쪽 참조이다.

### [참조 축약이 일어나는 문맥1] 템플릿 인스턴스화

보편 참조를 말하는 것이다. 축약되는 과정은 아래에서 자세히 다룬다.

template<typename T>

void func(T&& args);

func(a1); //T: Archer& / T&&: Archer& && == Archer&

func(Archer::createArcher()); //T: Archer / T&&: Archer&&

### [참조 축약이 일어나는 문맥2] auto변수의 형식 연역

이는 위와 사실상 같은 개념이다.

Archer a1;

auto&& a2 = a1; // Archer& && a2 = a1;

auto&& a3 = Archer::createArcher(); // Archer&& a3 = rvalue;

### [참조 축약이 일어나는 문맥3] typedef, 별칭 선언(using)

평가 도중에 또 다른 참조가 끼어들면 참조 축약이 발생한다.

template<typename T>

class BladeMaster

{

public:

typedef T&& WOW;

//T: int& / WOW: int& && == int&

};

### [참조 축약이 일어나는 문맥4] decltype

decltype에 관여하는 형식을 분석하는 도중에 참조가 등장하면 참조 축약이 일어난다.

Archer a0;

Archer& a1 = a0;

Archer a2;

auto&& a3 = static\_cast<decltype(a1) && >(a2); //<Archer& &&> == <Archer&>

# 참조 축약이 뭐 어쨌다고?

이걸 뭐 어디에 쓸려는 것일까? 결론부터 말하면 보편 참조와 완벽 전달에서 사용한다. 이를 작동하게 하는 핵심 원리는 참조 축약이다. 어째서 그런지를 아래에서 살펴보자.

## 보편 참조에서의 참조 축약

보편 참조란, 왼값, 오른값 모두 참조 가능하다 배웠다. 이것은 특별한 규칙이 아니다. 참조 축약일 뿐이다. 아래는 보편 참조하는 함수이다.

template<typename T>

void func(T&& args)

{

std::cout << "func.\n";

}

### T에 대한 연역, 부호화

위의 함수는 T가 무엇인지 연역해야 할 것이다. 위에서 살펴본 형식의 부호화가 다음과 같이 적용된다.

int main()

{

Archer a1;

func(a1); //T: Archer& / T&&: Archer& && == Archer&

func(Archer::createArcher()); //T: Archer / T&&: Archer&&

return 0;

}

a1은 왼값이므로 1번째 호출은 왼값이 전해지면 T는 왼값 참조(T: Archer&)로 연역된다.

2번째 호출은 오른값을 전해주므로 이 때의 T는 비참조(T: Archer)로 연역된다.

단지 다르게 저장되는 것이 아닌 중요한 의미가 있다. **이것으로 본래의 인자는 왼값인지 오른값인지 구분할 수 있는 것이다. 바로 부호화이다!**

### 참조 축약 발생

그럼 각각의 경우에 대해(왼값, 오른값) T가 정해졌다. 이제 참조 축약이 일어난다.

1번째 호출에서는 T&& args이다. T가 Archer&이므로 결국 (Archer&)&& args가 된다. &, &&이 만나면 어떻게 될까? 참조 축약 규칙을 보라. 왼값으로 설정되어 결국 Archer&가 되는 것이다.

2번째 호출에서는 비참조이므로 그대로 Archer&&가 된다.

**결국에는 왼값이 전달되면 \_Ty(템플릿 타입)은 T(객체)&이며, 오른값이 전달되면 \_Ty은 T가 된다.** 이로써 함수에서의 형식은 모두 정해진 것이다.

어떠한가? 매개변수로 **분명 연역과 참조 축약에 의해 왼값을 받을 수도 있으며, 오른값을 받을 수도 있지만 모두 \_Ty&&로 받으면서 다르게 기억하지 않는가?**

## std::forward에서의 참조 축약

보편 참조에는 std::forward가 따라다니게 되어있다. 왜냐하면 왼값과 오른값을 전달함에 따라 형식이 달라지는 것이 보편 참조의 특징이다. 이를 그대로 전달해야 한다. 이도 마찬가지로 참조 축약의 원리로 작동한다. 다음을 보자.

template<typename T>

void func(T&& args)

{

std::cout << "func.\n";

someFunc(std::forward<T>(args));

}

template<class \_Ty> inline

constexpr \_Ty&& forward(

typename remove\_reference<\_Ty>::type& \_Arg) \_NOEXCEPT

{ // forward an lvalue as either an lvalue or an rvalue

return (static\_cast<\_Ty&&>(\_Arg));

}

forward가 어떤 형식을 반환할까? 과정은 다음과 같다.

### \_Ty가 무엇인지 어떻게 알지?

함수의 매개변수는 오버로딩도 없는 분명 한 개이다. 바로 부호화의 힘이 여기서 나온다. 우리는 본래 매개변수가 왼값이냐 오른값이냐에 따라 다르게 전해주어야 하는데, 이 정보를 누가 가지고 있느냐? **바로 부호화된, \_Ty가 가지고 있다.**

### args가 왼값 참조이며, \_Ty가 왼값 참조일 경우

remove\_reference<\_Ty>::type&: 벗기고 &를 더하므로 T&  
static\_cast<\_Ty&&>: &에 &&가 만나 참조 축약의 규칙에 의해 T&  
결국, T&를 반환하게 된다.

### args가 왼값 참조이며, \_Ty가 비참조일 경우

remove\_reference<\_Ty>::type&: 비참조에서 &를 더하므로 T&  
static\_cast<\_Ty&&>: 비참조에 &&가 만나 T&&  
결국, T&&를 반환하게 된다.

### args가 오른값 참조일 경우

사실 \_Ty가 어떻게 연역되든 Archer&이거나 Archer이므로 일반적으로 왼값으로 넘어가게 된다. 하지만 args가 오른값일 수도 있다. std::forward는 이에 대해서도 오른값을 반환한다.

template<class \_Ty> inline

constexpr \_Ty&& forward(

typename remove\_reference<\_Ty>::type&& \_Arg) \_NOEXCEPT

{ // forward an rvalue as an rvalue

static\_assert(!is\_lvalue\_reference<\_Ty>::value, "bad forward call");

return (static\_cast<\_Ty&&>(\_Arg));

}

***참조 축약은 템플릿 인스턴스화, auto 형식 연역, typedef와 별칭 선언의 지정 및 사용, decltype의 지정 및 사용이라는 네 가지 문맥에서 일어난다.***

***컴파일러가 참조 축약 문맥에서 참조에 대한 참조를 만들어 내면, 그 결과는 하나의 참조가 된다. 원래의 두 참조 중 하나라도 왼값 참조이면 결과는 왼값 참조이고, 그렇지 않으면 오른값 참조이다.***

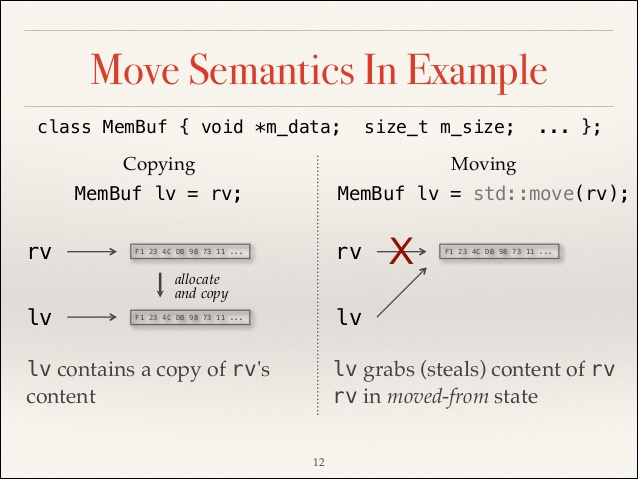
***형식 연역이 왼값과 오른값을 구분하는 문맥과 참조 축약이 일어나는 문맥에서 보편 참조는 오른값 참조이다.***

[항목29] 이동 의미론의 합당한 기대

이동 연산이 존재하지도 않고, 저렴하지도 않고, 적용되지도 않는다고 가정하라.

# 이동 의미론이 효율적인 이유

std::vector<Archer> va가 있다고 하자. 이를 vb에 옮기고 싶다. 복사 연산이라면 각 원소를 하나씩 옮겨야 하므로 O(n)이 될 것이다. 하지만 이동 연산이라면 컨테이너 내용이 저장된 힙 메모리의 주소를 가리키는 포인터만 맞바꾸면 된다. 이는 상수 시간에 이루어진다.



# 이동 의미론이 도움되지 않을 수도 있다.

위에서의 개념만으로는 상당한 이득이지만, 항상 이러한 과정이 이루어질 수 없다. 어떠한 경우에 그럴까? 다음과 같다.

## 이동 연산이 없다.

이동 연산이 없다면, 오른값이 넘어와도 왼값으로 변경해 복사 연산을 진행할 수 밖에 없다.

### 적법한 복사 연산과 이동 연산의 구현

일단 복사 연산과 이동 연산이 갖추어진 클래스를 살펴보자.

class Archer

{

public:

Archer() : pImpl(new Impl)

{ std::cout << "init\n"; }

~Archer() {}

Archer(const Archer& \_other)

: pImpl(new Impl(\*\_other.pImpl))

{

std::cout << "copy\n";

}

Archer(Archer&& \_other)

: pImpl(std::move(\_other.pImpl))

{

std::cout << "move\n";

}

private:

struct Impl

{

int data[100000000];

};

Impl\* pImpl;

};

int main()

{

Archer a1;

auto t1 = clock();

Archer a2 = a1;

std::cout << clock() - t1 << '\n'; //t:210

auto t2 = clock();

Archer a3 = std::move(a1);

std::cout << clock() - t2 << '\n'; //t:0 🡪 빠름

return 0;

}

### 이동 연산이 없으면?

**이동 연산 제거**하면? (다른 특수 함수가 사용자 정의되어 있다면 delete안해도 자동 생성 안 함)

Archer(Archer&& \_other) = delete;

오른값을 받지조차 못하도록 컴파일이 안 된다.

복사 연산은 정의했지만 **이동 연산을 정의하지 않으면?**

class Archer

{

public:

static Archer createArcher()

{

return Archer();

}

Archer() : pImpl(new Impl)

{ std::cout << "init\n"; }

~Archer() {}

Archer(const Archer& \_other)

: pImpl(new Impl(\*\_other.pImpl))

{

std::cout << "copy\n";

}

/\*Archer(Archer&& \_other)

: pImpl(std::move(\_other.pImpl))

{

std::cout << "move\n";

}\*/

Archer& operator=(const Archer& \_other)

{

pImpl = new Impl(\*\_other.pImpl);

std::cout << "copy operator\n";

return \*this;

}

/\*Archer& operator=(Archer&& \_other)

{

pImpl = std::move(\_other.pImpl);

std::cout << "move operator\n";

return \*this;

}\*/

private:

struct Impl;

Impl\* pImpl;

};

int main()

{

auto t1 = clock();

Archer a1;

Archer a2 = a1;

std::cout << clock() - t1 << '\n'; //210

auto t2 = clock();

Archer a3 = std::move(a1);

//Archer a4 = Archer::createArcher();

//이는 복사 생성자도 이동 생성자도 호출하지 않는다.

std::cout << clock() - t2 << '\n'; //144

return 0;

}

std::cout << clock() - t2 << '\n'; //144

//output  
init -> copy -> time: 5  
init -> copy -> time: 5

복사만 2번 수행한다. 오른값을 넘기지만 복사를 수행하는 것이다. 저 성능의 차이는 초기 Archer생성의 동적 할당 시간이다.

### 컴파일러가 자동으로 이동 연산을 만들어주지 않는가?

**Move Constructor, Move assignment operator** : static이 아닌 멤버들과 부모 클래스 부분을 이동시킨다. **클래스가 복사 연산, 이동 연산, 소멸자 그 어느 것도 선언하지 않았을 때만 자동 생성**된다. – [항목17]

이동 연산이 삭제되었을 수도 있다. – [항목11]

위와 같은 경우에는 이동 연산이 자동으로 작성되지 않는다. 기본적으로 이동 연산이 없다고 생각해야 계획의 오류가 없다.

## 이동이 더 빠르지 않다.

전체자료를 가리키는 포인터를 맞바꿈으로써 모든 자료를 옮기지 않고 시간을 절약하는데, 이러한 전체자료를 가리키는 포인터에 접근할 수 없는 경우가 있다.

### std::array

이 자료구조는 내장 배열을 컨테이너로 만든 것이다. 여기에는 전체를 가리키는 포인터가 없고 객체 자체에 직접 저장된다.

// TEMPLATE CLASS array

template<class \_Ty,

size\_t \_Size>

class array

{ // fixed size array of values

public:

// ……

\_Ty \_Elems[\_Size];

// ……

};

좀 어이없게 구현되어 있다. 당황스럽다.

여기서 이동 연산을 수행해도 \_Elems 의 전체 포인터가 없다. 그러므로 각 원소에 접근해야 하므로 효율적인 면이 없다.

### std::string의 작은 문자열 최적화(small string optimization, SSO)

a "short string optimization" (SSO) implementation. In this variant, the object contains the usual pointer to data, length, size of the dynamically allocated buffer, etc. But if the string is short enough, it will use that area to hold the string instead of dynamically allocating a buffer

std::string은 상수 시간 이동과 선형 시간 복사를 제공한다. 그러나 이렇게 내부 버퍼에 저장하면, 길이가 충분히 짧을 때 동적으로 할당하지 않으므로 효율이 좋아진다. 결국, std::array와 같은 개념이 된다.

## 이동을 사용할 수 없다.

해당 원소가 move시 noexcept를 지원하지 않으면, move semantics가 아닌 copy semantics로 element를 처리한다.

std::vector에 push\_back을 사용한다고 하자.

Archer a5;

std::vector<Archer> va;

va.push\_back(a5);

위는 복사를 수행한다, 하지만

Archer a5;

std::vector<Archer> va;

noexcept(va.push\_back(a5));

위는 이동을 수행하게 된다. [미결] 이동 생성자는 호출되지 않는다.

### noexcept operator (since C++11)

noexcept 연산자는 **예외를 throw하지 않도록 식이 선언 된 경우 true를 반환**하는 컴파일 타임 검사를 수행합니다.함수 템플릿의 noexcept 지정자 내에서 함수가 일부 유형에는 예외를 발생시키고 다른 유형에는 예외를 발생 시키도록 선언하는 데 사용할 수 있습니다.

if(noexcept(weAreFriend(a1, a2)))

std::cout << "weAreFriend\n";

if(noexcept(weAreTwince(a3, a4)))

std::cout << "weAreTwince\n";

## 원본 객체가 왼값이다.

오직 오른값만 이동 연산의 원본이 될 수 있는 경우도 있다. 이런 상황에서 만약 원본이 왼값이라면, 이동 연산은 일어나지 않을 것이다. – [미결] 무슨 말인지 모르겠다.

***이동 연산들이 존재하지 않고, 저렴하지 않고, 적용되지 않을 것이라고 가정하라.***

***형식들과 이동 의미론 지원 여부를 미리 알 수 있는 경우에는 그런 가정을 둘 필요가 없다.***

[항목30] 완벽 전달의 실패

완벽 전달이 실패하는 경우들을 잘 알아두라.

# 완벽 전달이란?

앞에서도 언급되었지만 정확한 의미는 다음과 같다.

### 전달(forwarding)

한 함수의 인수들을 다른 함수에 넘겨주는 것. 이는 값에 의한 전달과 비교된다. 값에 의한 전달은 복사하기 때문에 인수 속성을 그대로 넘겨줄 수 없다. 즉, 부호화가 안 된다.

### 완벽 전달(perfect forwarding)

**단순히 객체들을 전달하는 것만이 아니라, 그 객체들의 주요 특징, 즉 그 형식, 왼값 또는 오른값의 여부, const나 volatile여부까지도 그대로 전달하는 것을 말한다.**

완벽 전달을 사용하려면 보편 참조 매개변수가 필수적이다. 왜냐하면 매개 변수의 왼값 오른값에 대한 정보를 부호화하는 것을 필요로 하는데 이는 보편 참조가 유일하다.

# 완벽 전달의 사용 – [항목28] 참고

완벽 전달은 왜 써야 할까? 기본적 동기는 타입 연역으로 인해 매개변수 타입이 일정하지 않다라는 점. 매개변수 타입을 그대로 사용하려면 그대로 전달할 수 있는 방법이 필요하다.

template<typename T>

void fwd(T&& param) //보편 참조: 어떠한 타입의 매개변수도 받아들인다.

{

f(std::forward<T>(param));

}

위에서는 보편 참조를 사용하며 T는 연역될 것이다. 그리고 (연역된 타입 != T) param을 f에 전달한다.

## 완벽 전달을 써야 하는 이유

보편 참조로 param의 형식이 정확하게 추론되었다고 가정하자. 그럼 형식은 이미 정해진 셈인데 완벽 전달이 무슨 소용일까? 정해진 형식의 전달에 형식을 왜 제어해야 하는가?

void f(const std::vector<int>& data)

{

//작업 수행 중

std::cout << "complete.\n";

}

template<typename T>

void fwd(T&& param) //보편 참조: 어떠한 타입의 매개변수도 받아들인다.

{

//f(std::forward<T>(param));

f(param);

}

int main()

{

std::vector<int> v{ 1, 2, 3, 4, 5 }; //vector<int>&

fwd(v);

fwd(std::vector<int>(5, 3)); //vector<int>&&

return 0;

}

아무런 차이가 없다. 정상적으로 전달되며 결과도 같다.

void f(const std::vector<int>& data)

{

//작업 수행 중

std::cout << "lvalue complete.\n";

}

void f(std::vector<int>&& data)

{

//작업 수행 중

std::cout << "rvalue complete.\n";

}

하지만 이렇게 되면 결과가 다르다. 완벽 전달은 f에 rvalue로 전달하지만 일반 전달은 그럴 수 없다. 즉, param이 오른값일지라도 data는 왼값으로 전달되는 것이다.

### Q. 분명히 2번째 호출에서 param은 오른값이고 이를 전달했는데 왜 이런 결과가 생길까?

어떠한 인자의 왼값 오른값의 여부와 형식은 다른 것이다. 왼값 오른값의 여부는 주소를 취할 수 있는가의 여부이다. 매개변수의 타입은 어떤 타입을 인자로 받을 것인가이다.   
위는 어떤 상황인가? 인자로 오른값을 넘겼다. 그럼 이 매개변수는 함수 안에서도 오른값인가? 아니다. 함수 안에서는 왼값이다. 함수 내부에서는 함수 스택에 적재함으로써 주소를 취할 수 있다.

**이렇게 함수 안으로 들어오면 왼값이든 오른값이든 함수에서 그 변수는 왼값이다. 이게 완벽 전달이 필요한 이유다.** 원래 이 값이 무엇(왼값 or 오른값)이었는지 알 수가 없다. 이를 어떻게 알 수 있는가? 바로 템플릿 매개변수 T의 부호화이다. 이를 이용하여 부호를 확인하여 전달한다.

## 가변 인수 템플릿

임의의 형식과 개수의 인수들을 받는 템플릿을 말한다. 아래에서 사용법을 살펴보자.

<http://narss.tistory.com/entry/5%EC%9D%BC%EC%B0%A8-%EA%B0%80%EB%B3%80%EC%9D%B8%EC%9E%90-%ED%85%9C%ED%94%8C%EB%A6%BF-variadic-template>

template <typename ... Types> class Test {};

template <typename ... Types> void foo(Types ... args) {}

int main(int argc, char\* argv[])

{

foo(1, 3.4, "a");

Test<int, double> t;

return 0;

}

아래는 실용적인 예제이다.

template <typename T>

void xtuple\_size(const T& a)

{

cout << T::N << endl;

}

// tuple (primary template)

template <typename ...Types> class xtuple;

// empty tuple (specialization)

template <> class xtuple<> {};

// recursive tuple definition (partial specialization)

template <typename T, typename ... Types>

class xtuple<T, Types...> : private xtuple<Types ...>

{

public:

xtuple() {}

xtuple(const T& a, Types ... args) : first\_(a), xtuple<Types...>(args...)

{}

enum { N = 1 + sizeof...(Types) };

public:

T first\_;

};

int main(int argc, char\* argv[])

{

xtuple<int, char, double> t(1, 'a', 3.4);

cout << xtuple\_size(t) << endl; //< 3

return 0;

}

# 완벽 전달을 실패하는 경우

이제까지 완벽 전달을 왜 사용해야 하는지, 언제 사용할 지, 어떻게 사용하는지를 살펴보았다. 그럼 이제 완벽하게 전달할 수 없는 인수들은 어떤 것들인지 살펴보자.

## 중괄호 초기치

int main()

{

std::vector<int> v{ 1, 2, 3, 4, 5 }; //vector<int>&

fwd(v);

fwd({ 1, 3, 5 }); //중괄호 초기치

return 0;

}

이는 컴파일되지 않는다. 중괄호는 보편 참조 혹은 완벽 전달을 할 수 없다.

보편 참조에 전달은 인자의 형식을 추론하여 진행한다. 중괄호 전달은 추론이 불가능하다가 안 되는 이유다. 중괄호 초기치는 std::initializer\_list라는 타입인데, {}가 이 타입을 가리키지는 않는다. **암묵적 변환이 생길 뿐인데, 매개변수에서 암묵적 변환에 형식 연역을 동시에 진행할 수는 없다.** 반면, auto는 가능하다. – [항목1] 참고

### 해결책

그러므로 다음과 같이 해결한다.

auto temp = { 1, 2, 3 };

fwd(temp);

## 널 포인터를 뜻하는 0 or NULL

이는 포인터 형식이 아닌 명백히 정수형식이며, 단지 null포인터처럼 보이게 할 뿐이다.

### 해결책

간단하다. nullptr을 사용하면 된다.

## 선언만 된 정수 static const 및 constexpr 자료 멤버

클래스 안의 static멤버는 정의할 필요 없이 선언만 하면 된다. 컴파일러가 const전파를 적용해서 그 멤버 값을 위한 메모리를 따로 마련할 필요가 없어지기 때문이다.

class Archer

{

public:

static const int MAX\_POWER = 1000; //초기화 구문에서 따로 정의하지 않음

static Archer\* createArcher()

{

return new Archer;

}

};

아래와 같은 코드를 보자.

int main()

{

auto a1 = Archer::createArcher();

fwd(Archer::MAX\_POWER);

return 0;

}

참조는 포인터와 본질적으로 같다. 참조하려면 그 변수가 저장된 위치를 알고 있어야 한다. 하지만 const전파 적용으로 인해 static의 선언만으로는 이를 확인할 수 없다. 반드시 변수는 정의되어야 한다.

### 해결책

다음과 같이 하면 해결할 수 있다. 정의를 해주는 것이다.

const int Archer::MAX\_POWER;

int main()

{

auto a1 = Archer::createArcher();

fwd(Archer::MAX\_POWER);

return 0;

}

## 중복적재된 함수 이름과 템플릿 이름

선언으로 명확하게 대상을 알 수 없다면, 완벽 전달 역시 불가능하다. 누군지 알 수가 없기 때문이다.

### 중복적재된 함수를 완벽 전달하기

void f(int (\*pf)(int)) // void f(int pf(int))

{

std::cout << pf(3) << '\n';

}

int processVal(int param)

{

return param \* 3;

}

int processVal(int param, int exp)

{

return std::pow(param, exp);

}

template<typename T>

void fwd(T&& param) //보편 참조: 어떠한 타입의 매개변수도 받아들인다.

{

std::cout << param(3) << '\n';

}

int main()

{

fwd(processVal); //error: 중복적재된 함수 중 어느 것인지 확인할 수 없습니다.

return 0;

}

중복적재 시, 선언만으로는 어떤 함수를 사용할 지 알 수가 없다. 그러므로 완벽 전달은 불가능하다.

int main()

{

using processFuncType = int(\*)(int);

processFuncType processValPtr = processVal;

fwd(processValPtr);

fwd(static\_cast<processFuncType>(processVal));

return 0;

}

원하는 타입으로 캐스팅해주면 된다. 함수 타입은 다음과 같이 캐스팅하거나 명시해줄 수 있다.

### 템플릿 함수를 완벽 전달하기

같은 원리이다. 전달하고자 하는 중복적재나 템플릿 인스턴스를 직접 명시해주는 것이다.

template<typename T>

T workOnVal(T param)

{

std::cout << "work\n";

return param \* 3;

}

fwd(processValPtr);

fwd(static\_cast<processFuncType>(processVal));

//템플릿도 똑같이 함수 타입으로 캐스팅하여 해결한다.

fwd(static\_cast<processFuncType>(workOnVal));

## 비트필드

비트필드가 함수 인수로 쓰일 때, 완벽 전달에 실패한다. 왜냐하면 비트필드를 가리키는 포인터라는 것이 존재하질 않는다.

### 비트필드란?

클래스와 구조체는 정수 형식보다 작은 저장 공간을 차지하는 멤버를 포함할 수 있습니다. 이러한 멤버는 비트 필드로 지정됩니다.

// bit\_fields1.cpp

// compile with: /LD

struct Date {

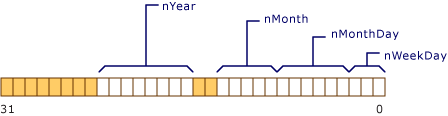
unsigned short nWeekDay : 3; // 0..7 (3 bits)

unsigned short nMonthDay : 6; // 0..31 (6 bits)

unsigned short nMonth : 5; // 0..12 (5 bits)

unsigned short nYear : 8; // 0..100 (8 bits)

};



nYear는 길이가 8비트이며 선언된 형식인 **부호 없는 short**의 단어 경계를 벗어납니다. 따라서 새 **부호 없는 short**의 시작 부분에서 시작됩니다. 모든 비트 필드가 기본 형식의 한 개체에 맞아야 할 필요는 없습니다. 선언에서 요청된 비트 수에 따라 새 저장소 단위가 할당됩니다.

**Microsoft 전용**

비트 필드로 선언되는 데이터의 순서는 위의 그림에 나온 것처럼 낮은 비트에서 높은 비트의 순서입니다.

### 해결책

위와 같은 비트필드를 매개변수로 전달하는 방법은 2가지다. 하나는 값으로 전달하기와 또 하나는 const에 대한 참조로 전달하는 것이다.

**중요한 것은 전달하고자 하는 변수의 공간을 만들어 주어야 한다는 것이다.** 값을 전달하는 경우는 복사본이 만들어진다. 하지만 참조의 경우에는 그렇지 않으므로 완벽 전달에 실패한다. 그러므로 참조로 넘어온 매개변수를 새로운 복사본으로 만들어서 그 복사본을 넘겨주면 된다.

struct IPv4Header

{

std::\_Uint4\_t version : 4, IHL : 4, DSCP : 6, ECN : 2, totalLength : 16;

};

int main()

{

IPv4Header h;

processVal(h.totalLength); //int로써의 역할을 할 수 있습니다.

fwd(h.totalLength); //error: 해당 인스턴스가 없습니다.

return 0;

}

## 실패에 대한 결론

대부분의 경우, 완벽 전달은 적절한 인자를 추론하고 전달할 수 있다. 하지만 크게 보면 다음과 같은 조건이 필요하다.

### 타입을 추론할 수 있어야 한다.

추론 근거에 의해서 타입을 명확하게 추론할 수 있는지 확인해야 한다. 매개변수에서 암시적 변환과 타입 추론이 동시에 생길 수 없다.

### 참조할 수 있는 주소가 있어야 한다.

실제로 참조할 수 있는 정의가 되어 있어야 한다. 보편 참조도 결국 참조이다.

***완벽 전달은 템플릿 형식 연역이 실패하거나 틀린 형식을 연역했을 때 실패한다.***

***인수가 중괄호 초기치이거나 0 또는 NULL로 표현된 널 포인터, 선언만 된 정수 static const 및 constexpr 자료 멤버, 템플릿 및 중복적재된 함수 이름, 비트필드이면 완벽 전달이 실패한다.***

[항목31] 기본 갈무리 모드

기본 갈무리 모드를 피해라.

# 람다 기본 지식

MSDN: C++11에서 람다 식(종종 람다라고 함)은 호출되었거나 인수로서 함수에 전달된 위치에서 바로 익명 함수 개체를 정의하는 편리한 방법입니다. 일반적으로 람다는 알고리즘이나 비동기 메서드에 전달되는 몇 줄의 코드를 캡슐화하는 데 사용됩니다.

## 1. lambda (람다)

**lambda**는 **람다 표현식**또는**람다 함수**, 그리고**이름 없는 함수(anonymous function)**라고 불리우며, **그 성질은 "함수 객체(functor)와 동일하다"** 할 수 있다.

많은 프로그래밍 언어들이 이름없는 함수 컨셉을 지원한다.  
이름없는 함수는 말 그대로 body는 가지나 이름이 없는 함수를 의미한다.  
lambda 함수는 암시적으로 함수 객체 클래스를 정의하고, 그 함수 객체를 생성한다.

참고로, lambda는 특별한 타입을 가지고 있다고 하나, **decltype, sizeof 는 사용할 수 없다**고 한다.

## 2. 함수 객체 vs lambda

코딩하다 보면, 함수 포인터와 함수 객체를 사용해야 할 때가 있다.  
함수 포인터와 함수 객체 모두 아래와 같은 장점과 단점이 존재한다.

**함수 포인터**는**작성이 가장 단순하나, 상태를 가지지 못한다**.

**함수 객체**는 **상태를 가질 수 있으나, 클래스나 구조체를 정의해야 하는 번거로움**이 있다.

lambda 함수는 함수 포인터와 함수 객체의 단점은 배제하고, 장점을 모두 가지고 있다.  
lambda 함수는 작성하기 용이하며(구조체 정의 작성 따윈 필요 없다), 상태 역시 가질 수 있다.

우선 간단한 예를 들어, 함수 객체에 비해 lamba가 얼마나 편리한 것인지 확인해 보도록 하자.

1. #include <algorithm>
2. #include <vector>
4. using namespace std;
6. // 함수 객체를 위한 구조체 정의
7. struct EvenCountFunctor : public unary\_function<int, void>
8. {
9. EvenCountFunctor(int& count) : evenCount(count) {}
11. void operator () (int number)
12. {
13. if (number % 2 == 0) { evenCount++; }
14. }
16. private:
17. int& evenCount;
18. };
20. int main()
21. {
22. vector<int> v;
23. for (int i = 0; i < 10; ++i)
24. {
25. v.push\_back(i);
26. }
28. int evenCount = 0;
29. // 함수 객체
30. for\_each(v.begin(), v.end(), EvenCountFunctor(evenCount));
32. evenCount = 0;
33. // lambda 함수
34. for\_each(v.begin(), v.end(), [&evenCount] (int n) {
35. if (n % 2 == 0) { evenCount++; }
36. });
37. }

위 예제에서 lambda의 편리함이 느껴지는가?  
구조체를 정의할 필요도 없이, functor의 장점을 가지면서도 단 두 줄로 코딩이 끝나 버렸다. [&event] 는 capture 라는 녀석인데, 이는 추후 다시 설명하도록 하겠다.

지금까지 lambda가 어떤 녀석이고, 어떠한 장점이 있는지 설명하였으니, 이제부터는 어떻게 사용해야 하는지에 대해 살펴보도록 하자.

## 3. lambda의 기본 사용법

아래 코드는 동일한 표현을 각각 auto와 function으로 선언한 모습이다. 자세한 내용은 아래에서 진행한다.

int main()

{

*//매개변수도 반환 형식이 없어요.*

**auto** func1\_a = [] { cout << "Hello " << endl; };

function<void()> func1 = [] { cout << "Hello" << endl; };

*//매개변수만 없어요.*

**auto** func2\_a = [] () -> int { **return** 5; };

function<int()> func2 = [] { **return** 5; };

*//둘다 있어요.*

**auto** func3\_a = [](int a, int b) -> int { **return** a\*b; };

function<int(int, int)> func3 = [](int a, int b) { **return** a\*b; };

**return** 0;

}

### 3-1. lambda의 문법 구성

lambda의 기본 문법은 다음과 같이 구성되어 있다.

1. **[]**      // lambda **capture** 영역
2. **()**      // **parameter list** 영역
3. **{}**      // **body** 영역
4. **()**      // 이는 위 lambda 함수를 **명시적으로 호출**할 때

위 문법을 이용하는 가장 간단한 예제를 살펴보자. 식 자체이다. 아무것도 하지 않는다.

1. int main()
2. {
3. // no capture, no parameter
4. [] { std::cout << “Welcome to lambda...” << std::endl; } ();
5. }

### 3-2. lambda의 parameter

lambda의 parameter 영역에 다음과 같이 인자들을 정의할 수 있다.

1. #include <iostream>
2. int main()
3. {
4. [] (int n) { std::cout << "Number : " << n << std::endl; } ();
5. }

### 3-3. lambda를 tr1::function에 대입

**lambda**는**function으로 대입이 가능**하다. 하나의 이름을 갖는 함수가 된다는 말이다.

**auto 키워드**를 쓰면 더더욱 편리하게 대입이 가능하다 (auto는 근래 C++ 변화 중 가장 고마운 녀석)

1. #include <iostream>
3. int \_tmain(int argc, \_TCHAR\* argv[])
4. {
5. auto lambaFunc = [] (int n)
6. {
7. std::cout << "Number : " << n << std::endl;
8. };
10. lambaFunc(100);
12. return 0;
13. }

그리고 auto를 사용하지 않고 다음과 같이 선언할 수 있다.

#include *<functional>*

#include *<iostream>*

int main()

{

**using** **namespace** std;

// Assign the lambda expression that adds two numbers to an auto variable.

**auto** f1 = [](int x, int y) { **return** x + y; };

cout << f1(2, 3) << endl;

// Assign the same lambda expression to a function object.

function<int(int, int)> f2 = [](int x, int y) { **return** x + y; };

cout << f2(3, 4) << endl;

}

다음 코드는 캡처된 변수에 람다 식을 바인딩한다.  지역 변수 i값과 참조로서 변수 j를 캡처하며 인자를 받지 않는 람다 식을 함수 객체로 만드는 과정이다.

int main()

{

**using** **namespace** std;

int i = 3;

int j = 5;

// The following lambda expression captures i by value and

// j by reference.

function<int (void)> f = [i, &j] { **return** i + j; };

// Change the values of i and j.

i = 22;

j = 44;

// Call f and print its result.

cout << f() << endl;

}

C++14에서부터 반환 값을 따로 정해주지 않아도 가능하다(람다3-7 참고).

### 3-4. lambda를 함수의 파라미터로 사용

lambda를 tr1::function으로 대입이 가능하다고 하였다. 즉, tr1::function 타입으로 템플릿 함수의 인자로써 lambda를 사용할 수 있다.

어떤 객체의 상황 별 콜백 함수를 불러줄 때 써 먹어도 나쁘진 않을 듯 하다.

단, 콜백 함수 개수가 너무 많지 않다는 가정 하에...

1. template<typename TFunc>
2. void TemplateFunc(TFunc func)
3. {
4. func();
5. };
7. int \_tmain(int argc, \_TCHAR\* argv[])
8. {
9. auto func = [] { std::cout << "Hello World" << std:: endl; };
10. TemplateFunc(func);
12. return 0;
13. }

### 3-5. lambda를 반환하는 함수

위에서 lambda를 함수의 인자로 받았으니, 함수가 lambda를 반환하는 것도 당연히 가능하다.

1. #include <iostream>
2. #include <functional>           // for tr1::function
4. using namespace std;
5. using namespace std::tr1;
7. // void 반환, 인자 없음의 함수타입을 반환한다
8. function<void ()> FunctionReturn()
9. {
10. return [] { cout << "대한민국~~~짝짝짝 짝짝~~~" << endl; };
11. }
13. int \_tmain(int argc, \_TCHAR\* argv[])
14. {
15. auto func = FunctionReturn();
16. func();
18. return 0;
19. }

### 3-6. lambda 함수를 STL container에 저장

3-4 와 3-5 가 가능하면, 이것 역시 당연히 가능하다.

1. #include <algorithm>
2. #include <functional>
3. #include <vector>
5. using namespace std;
6. using namespace std::tr1;
8. int main()
9. {
10. vector<function<const char\* ()> > v;
12. v.push\_back( [] { return "대한민국 "; } );
13. v.push\_back( [] { return "짝짝짝~~~짝짝~~~"; } );
15. printf("%s %s**\n**", v[0](), v[1]());
16. }

### 3-7. lambda의 반환값 타입 명시

lambda는 당연하게도 값을 반환할 수 있으며, **반환 값의 타입을 명시적으로 지정**할 수 있다. C++0x부터 도입된 **trailing return type (후행 반환 형식)**을 사용하는 것이다.

또한, 함수 포인터와는 다르게 암묵적으로 타입을 추론할 수도 있다. (함수 포인터는 한 번 반환 값의 타입을 지정하면, 무조건 그 타입으로만 반환이 가능하다)

1. int \_tmain(int argc, \_TCHAR\* argv[])
2. {
3. // 암시적 타입 반환
4. auto func1 = [] { return 3.14; };
6. // 후행 반환 형식을 이용한 명시적 반환값 타입 지정 ( [ **-> (return type)** ] )
7. // **파라미터가 없더라도 반드시 파라미터 블록을 지정해 주어야 한다**
8. auto func2 = [] () **-> float** { return 3.14f; };
10. // func2의 반환값은 double 타입으로 추론되었다. 아래 경고 발생
11. // warning C4244: '초기화 중' :
12. //                  'double'에서 'float'(으)로 변환하면서 데이터가 손실될 수 있습니다.
13. float f1 = func1();
15. // 명시적으로 float 반환했기에 OK
16. float f2 = func2();
18. return 0;
19. }

위 예제에서 **-> float**이 바로 명시적인 반환 타입을 지정하는 **trailing return type(후행 반환 형식)**이다. (즉, **() -> return type**)

**1. C++11까지의 lambda 반환 타입 지정**

TR1부터 적용된 lambda는 lambda 함수의 인자 타입이 반드시 명시적이어야 했다.

즉, 람다 함수의 인자들은 auto 키워드를 가질 수 없었기에, 인자 타입이 달라질 경우 람다 함수를 모두 오버로드 하듯이 만들어 주어야 했다.

1. auto iAdd = [] (int a, int b) -> int { return a + b; };
2. auto fAdd = [] (float a, float b) -> float { return a + b; };
4. int iResult = iAdd(1, 2);
5. float fResult = fAdd(1.f, 2.f);

**2. 이후 lambda반환 타입 지정**

C++14(VS2015) 부터는 이러한 제약이 없어져서, 람다 함수의 인자에 auto 키워드를 사용할 수 있게 되었다.

1. // C++11에서는 -> decltype(a + b) 후행 반환 형식을 써도, 컴파일이 안 되며...
2. // C++14에서는 -> decltype(a + b) 후행 반환 형식 없이도 아무 문제가 없다.
3. auto Add = [] (**auto** a, **auto** b) */\*****-> decltype(a + b)****\*/* { return a + b; };
5. iResult = Add(2, 4);
6. fResult = Add(2.f, 4.f);

auto 키워드에 대한 타입 추론을 해야 하기에, generic lambda 함수는 템플릿 인자 추론의 규칙을 따른다. (비슷하지만, 완전히 똑같지는 않다고 한다. 정확히 어떤 부분이 다른지는 아직 모르겠지만...)

따라서, 위의 예제는 다음과 같다.

1. struct unnamed\_add
2. {
3. template<typename T, typename U>
4. auto operator() (T x, U y) const
5. {
6. return x + y;
7. }
8. };
9. auto Add = unnamed\_add{};

이를 통해, 람다(익명) 함수를 마치 템플릿 함수처럼 사용할 수 있게 되었다.

1. int \_tmain(int argc, \_TCHAR\* argv[])
2. {
3. // 3.14에 **float suffix**를 붙여, 이 값이 float 타입임을 명시한다.
4. auto func1 = [] { return 3.14**f**; };
6. float f1 = func1();
8. return 0;
9. }

## 4. lambda의 capture

lambda **외부에 정의되어 있는 변수를 lambda 내부에서 사용하고 싶을 때 그 변수를 capture** 한다.

Capture는 참조나 복사 방식 모두 지원하며, **참조는 '&variable'** 로, **복사는 'variable'**로 기술한다. (뭐 딱히 평소에 사용하는 함수 파라미터에 적용되는 것이랑 다를 바가 없다)

Capture는 위 문법에서 살펴 보았듯이 **[] capture 영역에 기술**한다.

### 4-1. 변수를 참조로 캡쳐

이 예제는 위 lambda 소개에서 써먹었던 예제를 재활용하겠다. 아래 예제에서는 evenCount 변수를 lambda 함수가 **참조로 캡쳐**하였다.

1. #include <algorithm>
2. #include <vector>
4. using namespace std;
6. int main()
7. {
8. vector<int> v;
9. for (int i = 0; i < 10; ++i)
10. {
11. v.push\_back(i);
12. }
14. int evenCount = 0;
16. // lambda 함수
17. for\_each(v.begin(), v.end(), [**&evenCount**] (int n)
18. {
19. if (n % 2 == 0) { evenCount++; }
20. });
21. }

물론, 포인터를 넘기는 것도 동일한 효과를 가져온다.

1. int evenCount = 0;
2. **int\* pEvenCount = &evenCount;**
4. for\_each(v.begin(), v.end(), [**pEvenCount**] (int n)
5. {
6. if (n % 2 == 0) { **(\*pEvenCount)++;** }
7. });

### 4-2. 변수를 복사로 캡쳐

위 예제의 코드를 복사로 캡쳐하도록 해 보자.

1. int evenCount = 0;
3. // lambda 함수
4. for\_each(v.begin(), v.end(), [evenCount] (int n)
5. {
6. if (n % 2 == 0) { evenCount++; }
7. });

응? 그런데, 아래 에러 메시지를 뿜으며 컴파일이 되지 않는다.

**error C3491: 'evenCount': 변경 불가능한 람다에서 값 방식 캡처를 수정할 수 없습니다.**

꼭, 복사로 캡쳐한 변수를 lambda 내부에서 변경을 해야 한다면, ***mutable*** 키워드를 사용할 수 있다.

1. int evenCount = 0;
3. // lambda 함수
4. for\_each(v.begin(), v.end(), [evenCount] (int n) **mutable**
5. {
6. if (n % 2 == 0) { evenCount++; }
7. });

하지만, 당연히 복사 캡쳐이기 때문에, 외부의 evenCount는 당연히 값이 변하지 않는다.

### 4-3. 복수의 변수 캡쳐

복수의 변수 역시 캡쳐가 가능하다.

1. int a;
2. short b;
3. float c;
4. char d;
6. // a, d 참조로... b, c 복사로...
7. auto func = [&a, b, c, &d] { blah~blah~blah~};

### 4-4. 전역 변수 캡쳐시 주의사항

**lambda 바깥 함수의 범위보다 더 넓은 범위의 영역(전역 포함)**에 있는 **모든 변수들을**

* 참조로 캡쳐하고 싶을 땐 **[&]**
* 복사로 캡쳐하고 싶을 땐 **[=]** 로 사용한다.

**[&]와 [=]는 생각보다 자주 사용되니, 꼭 기억해 두길 바란다**.

그리고 이건 뭐 한 눈에 딱 봐도 에러인데, **같은 변수를 두 번 캡쳐하려 하면 아래와 같은 컴파일 에러**가 발생한다.

**error C3483: 'outer\_variable xxx'은(는) 이미 람다 캡처 목록의 일부입니다.**

아래와 같이 전역 변수를 사용하려 하면, 다음 에러가 발생한다.

**'g\_evenCount': 람다 캡처 변수는 바깥쪽 함수 범위에 속해야 합니다. 🡪 전역 변수를 캡쳐하고 싶어요.**

1. #include <algorithm>
2. #include <vector>
4. using namespace std;
6. int g\_evenCount = 0;
8. int main()
9. {
10. vector<int> v;
11. for (int i = 0; i < 10; ++i)
12. {
13. v.push\_back(i);
14. }
16. // lambda 함수
17. for\_each(v.begin(), v.end(), [&g\_evenCount] (int n)
18. {
19. if (n % 2 == 0) { g\_evenCount++; }
20. });
21. }

g\_evenCount는 lambda의 바깥쪽 함수(main 함수) 범위를 벗어난 전역에 위치하고 있다.

이런 경우엔 lambda 바깥 함수의 범위보다 더 넓은 범위의 영역(전역 포함)에 있는 모든 변수들을 참조할 수 있는 [&]를 써야 한다.

즉, 코드를 아래와 같이 변경해야 캡쳐가 가능해 진다.

1. // lambda 함수
2. for\_each(v.begin(), v.end(), [&] (int n)
3. {
4. if (n % 2 == 0) { g\_evenCount++; }
5. });

### 4-5. default 캡쳐

위에서 lambda 바깥의 모든 변수를 참조/복사로 캡쳐할 때 [&]/[=]를 사용한다고 하였다. 이와 같이 모든 변수에 대해 [&]/[=]를 통해 캡쳐하게 설정해 놓은 다음, 특정 변수에 대해서만 자신이 원하는 다른 방식으로 캡쳐하려 할 때, [&] 또는 [=] 를 **default 캡쳐**라 한다.

주의해야 할 사항이 하나 있는데, 이미 default 캡쳐가 참조 방식인데, 특정 변수를 또 다시 참조로 캡쳐하려 하면 아래와 같은 컴파일 에러가 발생한다. (복사의 경우도 마찬가지)

1. int \_tmain(int argc, \_TCHAR\* argv[])
2. {
3. int a, b, c;
5. // default 캡쳐모드는 참조, a와 b는 복사 캡쳐. OK
6. [&, a, b] {};
8. // default 캡쳐모드는 복사, c는 참조 캡쳐. OK
9. [=, &c] {};

12. // default 캡쳐모드가 참조인데, b를 또다시 참조로 캡쳐.
13. // error C3488: 기본 캡처 모드가 참조 방식인 경우 '&b'을(를) 사용할 수 없습니다.
14. [&, &b] {};
16. // default 캡쳐모드가 복사인데, a와 c를 또다시 복사로 캡쳐
17. // error C3489: 기본 캡처 모드가 값 방식인 경우 '&a'이(가) 필요합니다.
18. [=, a, c] {};
20. return 0;
21. }

출처:[[TR1] lambda #1](http://sweeper.egloos.com/3015035)

## 5. lambda in class

클래스의 멤버 함수 내에 lambda 함수를 정의하고, 이 lambda 함수에서 해당 클래스의 멤버를 호출하는 것이 가능하다.

클래스 멤버 함수 내 lambda 함수는 해당 클래스에서 friend 함수로 인식하므로, lambda 함수에서 클래스의 private 멤버에도 접근이 가능하다.

**클래스의 멤버를 호출할 때는 반드시 ‘this’ 를 캡쳐해야 한다.**

1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
5. using namespace std;
7. class PrimeNumber
8. {
9. public:
10. PrimeNumber()
11. {
12. m\_primeList.push\_back(1);
13. m\_primeList.push\_back(2);
14. m\_primeList.push\_back(5);
15. m\_primeList.push\_back(7);
17. // ...
18. }
20. void PrintPrimeNumbers() const
21. {
22. // this를 캡쳐한 것에 주목!!!
23. for\_each(m\_primeList.begin(), m\_primeList.end(), **[this]** (int primeNumber)
24. {
25. \_Print(primeNumber);
26. });
27. }
29. private:
30. typedef vector<int> PrimeNumberList;
31. PrimeNumberList m\_primeList;
33. void \_Print(int primeNumber) const
34. {
35. cout << "The prime number : " << primeNumber << endl;
36. }
37. };
39. int \_tmain(int argc, \_TCHAR\* argv[])
40. {
41. PrimeNumber pn;
42. pn.PrintPrimeNumbers();
44. return 0;
45. }

위 예제의 lambda 함수에서 보듯이 같이 [] 캡쳐 영역에 반드시 this 를 기술하여야 하며, PrimeNumber 클래스의 private 멤버 함수인 \_Print 함수를 호출하고 있다. 그리고, 당연히 lambda 함수를 통해 멤버 변수에도 접근이 가능하다.

## 6. Recursive lambda

lamba 함수 역시 함수의 특성을 모두 가지고 있으므로, 재귀(recursive)가 가능하다.

1. #include <functional>
3. using namespace std;
4. using namespace std::tr1;
6. int \_tmain(int argc, \_TCHAR\* argv[])
7. {
8. // function을 사용하였음에 주의하라
9. function<int (int)> **Factorial** = [&**Factorial**] (int num) -> int
10. {
11. return num <= 1 ? 1 : num \* **Factorial**(num - 1);
12. };
14. // 5 \* 4 \* 3 \* 2 \* 1 = 120
15. int fact5 = Factorial(5);
17. return 0;
18. }

흔하디 흔한 팩토리얼 코드이지만, 위 예제에서 중요한 포인트가 하나 숨어 있다.

보통 lambda 함수를 대입시킬 변수의 타입을 tr1::function이 아닌 auto로 잡는다. lambda 함수를 recursive 함수로 작성할 땐 반드시 auto가 아닌 tr1::function 타입으로 잡아야 한다. 그렇지 않으면, 아래 예제와 같은 컴파일 에러가 발생한다.

1. // 1. error C3536: 'Factorial': 초기화되기 전에 사용할 수 없습니다.
2. // 2. error C3533: 'auto &': 매개 변수에 'auto'가 포함된 형식을 포함할 수 없습니다.
3. // 3. error C3531: 'Factorial': 형식에 'auto'가 포함된 기호에는 이니셜라이저가 있어야 합니다.
4. // 4. error C2064: 항은 1개의 인수를 받아들이는 함수로 계산되지 않습니다.
6. **auto** Factorial = [&Factorial] (int num) -> int
7. {
8. return num <= 1 ? 1 : num \* Factorial(num - 1);
9. };

auto는 타입 추론을 하는 녀석이다.

위 함수에서 auto가 허용되지 않는 것은 아직 타입이 제대로 추론도 되기 전에 재귀 함수를 사용하려 하니 문제가 되는 것이다.

다시 얘기하지만, **lambda recursive 함수의 대입 타입으로 auto를 사용하면 안 된다**.

## 7. lambda와 closure

### 클로저

람다 표현식으로 만들어진 **함수 객체**를 **클로저**라고 부른다. 람다 표현식과 그로부터 생성된 객체를 구분하는 것은 큰 의미가 없으므로 모두 람다라고 칭한다.

간혹 lambda가 clousure라고 혼동하는 사람들이 꽤 있다. lambda는 closure가 아니다. **단지 lambda를 이용하여 closure의 특성을 구현해 낼 수 있는 것**이다.

아래 예제를 한 번 살펴 보자. 아래 예제의 두 번째 f 호출의 결과는 38인가? 70인가?

1. #include <iostream>
2. #include <functional>
4. using namespace std;
5. using namespace std::tr1;
7. int \_tmain(int argc, \_TCHAR\* argv[])
8. {
9. int a = 7, b = 3;
11. // 외부 변수 a와 b를 복사 캡쳐하고, int x를 파라미터로 받는 lambda 함수
12. // a \* x + b 의 결과를 출력한다.
13. auto closureFunc = [a, b] (int x)
14. {
15. cout << a \* x + b << endl;
16. };
18. closureFunc(5);  // 7 \* 5  + 3 = 38 를 출력
20. // 여기에서 a와 b를 각각 7 -> 10, 3 -> 20으로 바꾸었다
21. a = 10, b = 20;
23. **// 그렇다면 지금의 closureFunc(5)는**
24. **// 7 \* 5 + 3 = 38 일까? 아니면 10 \* 5 + 20 = 70일까?**
25. **closureFunc(5);**
27. return 0;
28. }

두 번째 closureFunc 함수 호출의 결과는 38이 출력된다.

**function<void (int)> closureFunc 는 이미 생성될 때 a와 b가 7과 3으로 바인딩** 되었다. 따라서, closureFunc 는 이미 7 \* 5 + 3 의 결과만을 출력하는 **하나의 객체**의 특성을 가지게 되었다.

다르게 표현하자면,**closureFunc 함수는 a 와 b 의 값에 "Closed over(Closure)" 되었다**고 할 수 있다. 이와 갈은 변수의 바인딩 방식을 static(lexical) binding 이라 부른다.

**Closure 특성이 유용한 이유**는 위 예제에서도 볼 수 있듯이 **late evaluation 이 가능하다**는 것이다. 즉, **closure 함수는 바뀐 주변 환경과 관계없이 자신이 정의될 때의 환경에 따라 행동**한다. 이는 이후 해당 함수들이 필요할 때 실행시킬 수 있는 이른바 multiple function 의 사용을 가능하게 한다.

## 8. 정리

1. lambda는 람다 표현식 또는 람다 함수, 그리고 이름 없는 함수라고 불리며, 함수 오브젝트 중 하나이다.
2. lambda는 특별한 타입을 가지고 있다고 하지만 decltype 이나 sizeof 는 사용 불가
3. lambda를 정의한 곳의 외부 변수를 lambda 내부에서 사용하고 싶을 때는 참조 또는 복사로 캡쳐한다.
4. 클래스에서도 lambda를 사용할 수 있으며, 클래스는 lambda를 friend 함수로 인식한다.
5. lamba는 closure가 아니다. 단지 lambda를 이용하여 closure의 특성을 구현할 수 있는 것이다.

출처:[[TR1] lambda #2](http://sweeper.egloos.com/3015050)

# 기본 갈무리 모드(default capture mode)

람다식에서 외부에서 변수값을 가져올 때, 캡처를 사용한다. 이를 사용한 람다 상태를 기본 갈무리 모드라 한다.

## 참조에 의한 갈무리 모드

아래와 같이 표현될 수 있다. &divisor을 외부에서 캡쳐하여 사용한다. 자세한 사용법은 람다 기본 지식 참고

[미결] 참조를 전달하면 참조 값이 이상해지는 문제가 있다. ITEM39.cpp

filters.emplace\_back([&](int value){ return value % divisor == 0; } );

## 값에 의한 갈무리 모드

아래와 같이 표현된다. 자세한 사용법은 람다 기본 지식 참고

filters.emplace\_back([=](int value){ return value % divisor == 0; });

# 기본 갈무리 모드의 위험성

이번 장의 핵심은 위 두 가지를 피하라는 것이다. 왜 그래야 하는가?

## 참조 갈무리 모드의 위험성

참조가 대상을 잃을 위험이 있다.

using FilterContainer = std::vector<std::function<int(int)> >;

FilterContainer filters;

int main()

{

for (int i = 2; i < 100; i++)

{

auto divisor = i;

filters.emplace\_back([&](int value){ return value % divisor == 0; });

}

for (auto& v : filters)

{

std::cout << v(100) << " ";

}

std::cout << '\n';

return 0;

}

위의 코드에서 divisor을 참조하여 사용한다. 하지만 해당되는 반복문의 범위를 벗어나면 divisor은 저 범위에서 설정한divisor이 아닌 새로운 divisor이 된다. 즉, 값을 잃어버린다.

### 명시적 갈무리를 사용하면 어떨까?

다음과 같이 되는 것이다.

filters.emplace\_back([&divisor](int value){ return value % divisor == 0; });

위의 문제를 해결하지는 못한다. 하지만 해당 매개변수가 수명에 의존한다는 점을 명확히 나타낸다는 장점이 있기는 하다.

## 값에 의한 갈무리의 위험성

참조 갈무리에서 보았던 범위로 인한 문제는 해결될 수 있지만, 문제는 남아있다.

### 포인터 값을 갈무리하는 경우

포인터 값을 갈무리하면 그 포인터는 클로저 안으로 복사되는데, 그 포인터는 언제든지 사라질 수 있으며 그러면 클로저가 복사한 객체는 유효하지 않게 된다.

### 멤버 변수 값을 갈무리하는 경우

클래스 멤버 함수 안에서 값에 의한 갈무리를 사용한다고 하자.

class Archer

{

public:

void addSkill() const

{

filters.emplace\_back(

[=](int value) { return value \* power; }

);

}

private:

int power;

};

멤버 변수들에는 암묵적으로 this가 숨겨져 있다. 그러므로 범위 안에서 새로 복사하여 캡처하면 된다. 아래와 같다.

void addSkill() const

{

auto powerCopy = power;

filters.emplace\_back(

[powerCopy](int value) { return value \* powerCopy; }

//[power = power](int value) { return value \* power; } //C++14

);

}

[powerCopy]대신 [=]로 캡쳐해도 가능하다.

### 값에 의한 갈무리도 자기 완결적이지 않다.

값을 복사하여 넣었으므로 외부의 변화에 영향을 받지 않는다고 생각하기 쉽지만 전혀 그렇지 않다. 왜냐하면 람다가 지역 변수, 매개변수뿐만 아니라 정적 저장소 수명 기간을 가진 객체에도 의존할 수 있기 때문이다.

int main()

{

for (int i = 2; i < 100; i++)

{

static auto divisor = i;

filters.emplace\_back([=](int value){ return value % divisor == 0; });

}

for (auto& v : filters)

{

std::cout << v(100) << " ";

}

std::cout << '\n';

return 0;

}

divisor이 정적 변수이다. 놀라운 것은 값을 캡쳐하였지만 이는 값을 공유하고 있다. 즉, 외부의 divisor의 변화에 각 람다식들은 영향을 받는다.

결국, static변수를 캡쳐하면 참조 갈무리 모드와 같은 상황이 되어버린다.

***기본 참조 갈무리는 참조가 대상을 잃을 위험이 있다.***

***기본 값 갈무리는 포인터(특히 this)가 대상을 잃을 수도 있으며, 람다가 자기 완결적이라는 오해를 부를 수 있다.***

[항목32] 일반화된 람다 갈무리

객체를 클로저 안으로 이동하려면 초기화 갈무리를 사용하라.

# 일반화된 람다 갈무리(generalized lambda capture)

C++14에서 추가된 새로운 갈무리 매커니즘인 초기화 갈무리(init capture)와 같다.

기본 갈무리 모드인 값 갈무리, 참조 갈무리도 마땅치 않은 경우가 있다. 게다가 이전 항목에서는 이들의 사용을 피하라고 하고 있다.

## 정의

하나의 표현식으로 나타낸다.   
1. 좌항: 람다로부터 생성되는 클로저 클래스에 속한 자료 멤버의 이름  
2. 우항: 그 자료 멤버를 초기화하는 표현식

코드로는 다음과 같이 표현할 수 있다.

int main()

{

auto pa = std::make\_unique<Archer>();

auto func = **[pa = std::move(pa)]**{ return pa->intro(); }

return 0;

}

## 사용법

단순히 변수의 참조와 복사가 아닌 표현식을 대입할 수 있다는 점에서 다양한 캡쳐가 가능하게 되었다.

### 이동 연산

위에서 살펴본 방식이다. 오른값을 전해주기 위한 방법이다.

auto pa = std::make\_unique<Archer>();

auto func = **[pa = std::move(pa)]**{ return pa->intro(); }

### 직접 초기화

초기화도 표현식이므로 캡쳐하여 값을 전해줄 필요 없이 즉시 초기화할 수 있다.

auto func = **[pa = std::make\_unique<Archer>()]**{ return pa->intro(); };

# 초기화 갈무리 따라하기

C++11에는 초기화 갈무리를 사용할 수 없다. 이를 따라한다는 것은 일반화된 갈무리의 핵심을 더 잘 이해하기 위함이다.

## 클래스 직접 생성

람다 표현식은 컴파일러가 하나의 클래스를 자동으로 작성하여 그 클래스의 객체를 생성하게 만드는 수단일 뿐이다. 그러므로 람다가 할 수 있는 모든 일들은 클래스를 직접 만들어서 따라 할 수 있다.

class IntroArcher

{

public:

using DataType = std::unique\_ptr<Archer>;

explicit IntroArcher(DataType&& ptr)

: pa(std::move(ptr)) {}

std::string operator()() const

{

return pa->intro();

}

private:

DataType pa;

};

int main()

{

auto pa = std::make\_unique<Archer>();

//auto func = [pa = std::move(pa)]{ return pa->intro(); };

//auto func = [pa = std::make\_unique<Archer>()]{ return pa->intro(); };

auto func = IntroArcher(std::make\_unique<Archer>());

return 0;

}

## std::bind사용하기

std::bind를 이용하여 임의의 매개변수를 묶어 클로저로 만든다.

### std::bind

어떤 함수를 원하는 매개변수와 묶어서 새로운 함수를 반환한다. 간단한 예제를 보자.

#### std::bind 사용하기

std::bind 를 사용하기 위해서는 <functional> 표준 라이브러리 해더를 include해주어야 한다. std::bind를 이용해서 위의 문제를 어떻게 깔끔하게 처리할 수 있는지 아래의 예제를 보자.

int add(int a, int b) {

return a + b;

}

int main() {

auto f = bind(add, 1, 2);

cout << f() << "\n";

}

bind 함수가 여기서 한 일은 첫번째 인자로 함수명을 받고 나머지 인자들을 첫번째 인자로 받은 함수의 인자에 **bind**해서 필요한 시점에 호출할 수 있는 객체를 만들어 준 것이다. 일단 여기서 어떤 객체가 생성되었는지는 고민할 필요는 없다. 천천히 알아가도록 하자.

#### 변수 혹은 참조 바인딩

위의 예제에서는 상수를 바인딩하는 예제를 썼는데 변수 또는 참조 변수도 바인딩 가능하다. 아래 예제를 보자.

int main() {

int x = 1, y = 2;

auto addxy = bind(add, x, y);

cout << addxy() << "\n";

}

변수들의 경우 bind가 호출되는 시점의 변수 값으로 bind가 일어 난다는 것에 주의를 할 필요가 있다. bind이후에 변경된 변수 값은 결과에 반영되지 않는다.

int main() {

int x = 1, y = 2;

auto addxy = bind(add, x, y);

cout << addxy() << "\n";

x = 3;

y = 4;

cout << addxy() << "\n";

}

위 예제의 결과는 3, 7이 아니라 3, 3이다.

그러나 이것을 해결할 방법도 있다. 아래의 예제와 같이 바인딩할 때 cref라는 헬퍼 함수의 도움을 받아 참조를 넘기면 된다.

int main() {

int x = 1, y = 2;

auto addxy = bind(add, cref(x), cref(y));

cout << addxy() << "\n";

x = 3;

y = 4;

cout << addxy() << "\n";

}

위의 결과는 아래와 같이 이전과는 다른 결과를 보인다.

3

7

### std::bind를 이용하여 클로저 만들기

std::bind로 왼값 참조 매개변수를 받아 클로저로 만드는 코드를 살펴보자.

std::vector<Archer> group;

auto func14 = [group = std::move(group)] // C++14

{

for (auto& a : group)

{

a.intro();

}

};

auto func11 = **std::bind**([](const std::vector<Archer>& group) // C++11

{

for (auto& a : group)

{

a.intro();

}

}, std::move(group));

std::bind로 구현할 시, 유념해야 할 사항들이 있다.  
[미결] 1. 매개변수는 왼값 참조이다. group의 복사본을 초기화하는 데 쓰인 표현식(std::move(group))은 오른값이지만, group의 복사본 자체는 왼값이다.  
[미결] 2. 람다로부터 만들어진 클로저 클래스의 operator()멤버함수는 const이다. 즉, 매개변수의 변경을 허용하지 않는다. 만약 허용하려면, mutable을 사용해야 한다.

auto func11 = std::bind([](std::vector<Archer>& group) mutable // C++11

{

for (auto& a : group)

{

a.intro();

}

}, std::move(group));  
3. 바인드 객체는 람다가 산출한 클로저의 복사본을 포함한 모든 인수를 저장한다. 그러므로 바인드의 수명은 클로저의 수명과 같다.

### std::bind의 근본적인 요점

객체를 C++11 클로저 안으로 이동 생성하는 것은 불가능하나, 객체를 C++11 바인드 객체 안으로 이동 생성하는 것은 가능하다.

객체를 바인드 객체 안으로 이동 생성하고, 이동 생성된 객체를 람다에 참조로 전달하는 것이다.

바인드 객체의 수명은 클로저의 수명과 같다.

***객체를 클로저 안으로 이동할 때에는 C++14의 초기화 갈무리를 사용하자.***

***C++11에서는 직접 작성한 클래스나 std::bind로 초기화 갈무리를 흉내낼 수 있다.***

[항목33] 람다 매개변수 auto&&

std::forward를 통해서 전달할 auto&& 매개변수에는 decltype을 사용하라.

# 일반적 람다(generic lambda)

매개변수에 auto를 사용한 람다를 말한다. 이는 템플릿 연역처럼 자동으로 추론될 수 있다.

auto f = [](auto x) { return x \* 3; }; //lambda

class GetPower

{

public:

template<typename T> // template

auto operator()(T x) const -> decltype(T) { return x \* 3; }

};

int main()

{

auto value1 = f(9999);

auto value2 = GetPower(9999);

return 0;

}

위의 코드에서 두 가지 경우는 완전히 동일하다.

## 람다에서의 보편 참조

템플릿과 같다고 했는데 람다에서도 보편 참조가 가능할까? 아래와 같이 한다.

auto f = [](auto&& x) { return x \* 3; }; //lambda

보편 참조의 원리와 완전히 동일하다.

## 람다에서의 완벽 전달

그럼 보편 참조를 완벽 전달하려면 어떻게 해야 할까?

auto f = [](auto&& x) { return calc(std::forward<???>(x)); }; //lambda

std::forward로 전달한다고 해보자. 위에서 ???안에는 무엇이 들어가야 할까? T가 없으므로 난감해 보인다.

하지만 x의 형식을 추론할 수 있는 간단한 방법이 있다. 이미 배운 decltype이다.

auto f = [](auto&& x) { return calc(std::forward<decltype(x)>(x)); }; //lambda

***std::forward를 통해서 전달할 auto&& 매개변수에는 decltype을 사용하라.***

[항목34] std::bind vs. lambda

std::bind보다 람다를 선호하라.

# std::bind

함수 포인터를 인자로 받아 매개변수를 묶어 새로운 함수를 반환한다. 기존의 함수의 매개변수를 묶어 사용할 수 있다.

# lambda

간단하게 함수 객체를 만들어낸다. 기존 함수를 호출하여 간단하게 매개변수를 고정할 수 있다. 물론 이 외에도 다양한 함수를 만들 수 있다.

# lambda를 std::bind보다 선호해야 한다.

사실 C++14에서는 언제나 lambda가 더 나은 선택이다. 그 이유를 살펴보자.

## 가독성이 더 좋다.

다음과 같은 함수가 있다.

using Time = std::chrono::steady\_clock::time\_point;

enum class Sound { Beep, Siren, Whistle };

using Duration = std::chrono::steady\_clock::duration;

void setAlarm(Time t, Sound s, Duration d)

{

std::thread th([&]()

{

std::cout << "sound.\n";

std::this\_thread::sleep\_for(d);

});

th.join();

}

### 람다

람다에서는 다음과 같이 매개변수를 묶을 수 있다.

auto setAlarmL = [](Sound s)

{

using namespace std::chrono;

using namespace std::literals;

setAlarm(steady\_clock::now() + 1h, s, 30s);

};

### std::bind

std::bind로는 다음과 같이 묶을 수 있다.

sing namespace std::chrono;

using namespace std::literals;

using namespace std::placeholders;

auto setAlarmB =

std::bind(setAlarm, steady\_clock::now() + 1h, \_1, 30s);

위에서만 봐도 std::bind에서의 인자가 하나 더 늘어남으로써 일단 코드가 길어졌다. 게다가 \_1의 인자전달은 그리 직관적이지 않다.

더 심각한 문제는 함수 호출 시점이 달라진다는 것이다. 람다식의 경우에는 기존 함수를 그대로 불러오는 것이라서 해당 표현식에 대한 평가는 기존 함수에서 시작된다. 하지만 std::bind는 새로운 함수를 만들어낸다. 시점도 새로이 만든 함수를 기준으로 진행되어 표현식에 대한 평가가 std::bind를 기준으로 시작된다. 이를 해결하려면 표현식에 대한 평가를 기존 함수의 시점으로 미뤄주어야 한다. 다음과 같이 한다.

using namespace std::chrono;

using namespace std::literals;

using namespace std::placeholders;

auto setAlarmB =

std::bind(setAlarm,

std::bind(std::plus<>(), steady\_clock::now() + 1h),

\_1, 30s);

무엇이 더 간단한지는 눈으로 보면 알 수 있다.

## 중복적재시, 문제가 없다.

인수 3개짜리와 4개짜리 함수가 중복적재되어 있다고 해보자.

void setAlarm(Time t, Sound s, Duration d);

void setAlarm(Time t, Sound s, Duration d, Volume v)

### 람다

람다에서는 중복적재가 아무런 영향을 주지 않는다. 왜냐하면 기존 함수를 직접 불러오기 때문에 컴파일러가 함수를 구분할 수 있다.

auto setAlarmL = [](Sound s)

{

using namespace std::chrono;

using namespace std::literals;

setAlarm(steady\_clock::now() + 1h, s, 30s);

};

### std::bind

std::bind에서는 실행되지 않는다. [미결] std::bind입장에서는 함수의 이름뿐만이 모르므로 어떤 함수를 선택해야 할지 알 수가 없다. 이에 대한 해결책으로 캐스팅을 진행하면 된다. 아래와 같이 진행한다.

using SetAlarm3ParamType = void(\*)(Time t, Sound s, Duration d);

auto setAlarmB =

std::bind(static\_cast<SetAlarm3ParamType>(setAlarm),

std::bind(std::plus<>(), steady\_clock::now() + 1h),

\_1, 30s);

### inline가능성으로 인한 속도 차이

람다식은 통상적으로 인라인화할 수 있는 보통 함수이다. 하지만 std::bind는 함수 포인터로 접근한다. 함수 포인터로써의 접근은 인라인화할 가능성을 낮춘다. 함수 포인터로 접근하면 목적파일에 합칠 수 없기 때문이다.

## 복잡한 코드의 경우

주어진 인수가 최대값과 최소값 사이에 있는지 여부를 반환하는 함수이다.

### 람다

const int lowVal = 1, highVal = 9999;

auto betweenL = [lowVal, highVal](const auto& val)

{

return lowVal <= val && val <= highVal;

};

일반 함수를 만들듯이 아주 직관적이고 간단하다.

### std::bind

auto betweenB =

std::bind(std::logical\_and<bool>(),

std::bind(std::less\_equal<int>(), lowVal, \_1),

std::bind(std::less\_equal<int>(), \_1, highVal));

당황스럽다. 이런 코드는 어디에서 발생한 것일까?

## 인자 전달 과정

함수로 인자를 전달할 때, 매개변수가 어떻게 전달되는지 확인한다.

### 람다

Archer a1;

//대상a에게 damage의 피해를 입힙니다.

auto targetShotL = [a](int damage) mutable

{

a1.hit(damage);

};

캡쳐한 외부 변수는 타입 그대로 전달된다. &이면 참조, 그렇지 않으면 값으로 전달된다.   
매개변수 역시 타입 그대로 전달된다.

### std::bind

auto targetShotB = std::bind(hit, a1, \_1);

뭐가 뭔지도 알아보기 어렵지만 더 중요한 것은 타입 전달 방식이다.   
a1은 값으로 전달된다. 그대로 전달된다. (참조로 전달하려면 std::ref를 사용한다.)  
모든 인수는 참조로 전달된다. 함수 호출 연산자가 완벽 전달을 사용한다.

더 읽기가 어렵게 되었다.

## 다형적 함수 객체

어떠한 형식이든 인수를 받는 함수 객체이다.

### std::bind

class MasterPlan

{

public:

template<typename T>

void operator()(const T& param) const

{

std::cout << "masterPlan\n";

}

};

MasterPlan mp;

auto getMasterPlanB = std::bind(mp, \_1);

getMasterPlanB(1);

getMasterPlanB("ytk");

getMasterPlanB(Archer());

어떠한 인수든 받아들일 수 있습니다.

### 람다

중요한 것은 람다는 더 쉽게 위의 기능을 가능하게 합니다.

auto getMasterPlanL = [mp](const auto& param)

{

mp(param);

};

getMasterPlanL(1);

getMasterPlanL("ytk");

getMasterPlanL(Archer());

이 정도면 무시무시할 정도로 간단하다.

***std::bind를 사용하는 것보다 람다가 더 읽기 쉽고 표현력이 좋다. 그리고 더 효율적일 수 있다.***

***C++14가 아닌 C++11에서는 이동 갈무리를 구현하거나 객체를 템플릿화된 함수 호출 연산자에 묶으려 할 때, std::bind가 유용할 수 있다.***