[내용정리] Effective C++

# 목적: C++ 언어를 효과적으로 구사하는 방법을 배운다.

C++의 조각조각 나뉜 여러 부분들을 이해하고 제대로 합쳐 효과적인 프로그램을 만들 줄 안다.

보다 나은 설계, 문제를 회피하는 법

항목의 이면에 숨겨진 근본 원리를 이해한다.

C++이 어떻게 동작하는지, 왜 이렇게 돌아가는지, 어떻게 사용할지

# 과정: 이후 진행예정

[용어] *C++* 프로그래밍 용어

선언: ‘어떤 대상(객체, 함수, 클래스, 템플릿)’의 이름과 타입을 컴파일러에게 알려 주는 것

**C auto, static, extern 키워드 의미**

시그니처: 그 함수의 매개변수 리스트와 반환 타입

정의: 구체적인 세부사항(객체에 대한 메모리 마련, 함수 본문, 템플릿 멤버를 넣어 준 결과)을 컴파일러에게 제공

초기화: 어떤 객체에 최초의 값을 부여하는 것

복사 생성자: 어떤 객체를 다른 객체와 똑같이 초기화 할 때 호출  
*cf)* 복사 생성자 = 값에 의한 전달 = 인자로 들어간 객체의 값이 함수에서 그대로 복사되어 수행된다.

복사 대입 연산자: 값을 복사할 때 호출  
*cf)* 복사 생성자 vs. 복사 대입 연산자

STL(Standard Template Library)

함수 객체: 연산자 operate()를 오버로딩하여 함수(func(매개변수))처럼 동작하는 C++ 객체

미정의 동작: 정의되어 있지 않은, 실행 시간에 어떤 현상이 터질지 예측할 수 없는 동작

인터페이스: 함수의 시그니처, 클래스의 접근가능요소, 템플릿 타입 매개변수 등 일반적인 설계 아이디어

사용자: 해당 코드를 사용하는 모든 사람

[항목1] C++의 법을 따른다.

# C++이 제공하는 강력한 기능들

C with classes에서 발전하여 어떤 언어보다도 다양한 기능을 가진 언어이며, 이에 강력하다. 하지만 그만큼 쓰기도 어렵다. 효율적으로 사용하기 위해서 C++이 제공하는 기능들을 분명하게 하는데 목적이 있다.

1. **Procedural programming: C**
2. **Object-oreinted: class**
3. **Functional: pure-funtion**
4. **Generic programmng: template**
5. **STL: Standard Template Library**

각 기능들은 무엇을 의미하는가?

#### 1. 절차지향 언어: C

**정의**: 하나의 실행 단위(=함수)를 순서에 따라 설계하는 프로그래밍, 이를 장려하는 언어는 절차지향 언어가 된다.

**정리**: 절차지향 프로그래밍의 주된 특징은 순서와 단위를 가진다는 것이다.   
순서에 따른다함은 goto문이나 jump문으로 흐름을 뒤바꾸는 것이 아닌, 하나의 흐름을 따라서 지속적으로 진행함을 말한다.   
또 단위(모듈성)을 가진다. 하나의 함수는 실행 단위가 되며, 서브 루틴이라 부른다. 재활용할 수도 있으며, 해당 실행의 입력과 출력을 명확히 명세할 수 있다. 그리고 유효범위를 가짐으로써, 변수 범위가 제한되어 혼동을 막아준다.

#### 2. 객체지향 언어: class

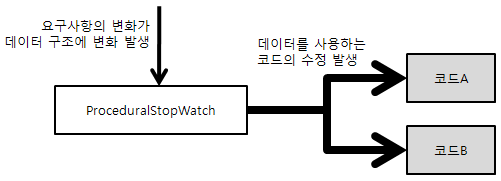
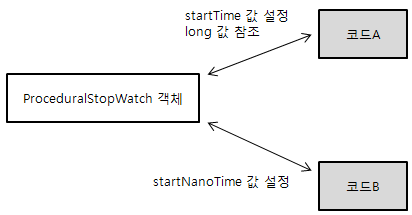
**정의**: 자료(data)와 행위(method)를 갖는 객체(class)를 만들어 이들을 조합하여 설계하는 프로그래밍.

**정리**: **객체지향의 중요한 특징으로 캡슐화, 상속, 다형성**이 있다. 이 중에서도 위의 정의를 받쳐주는 요소는 캡슐화이다. 상속과 다형성은 이것을 풍성하게 해준다. 아래를 보자.

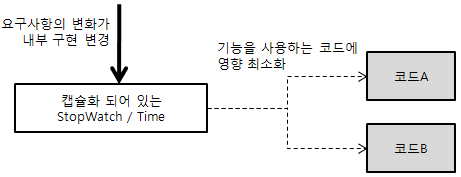
###### 객체지향 vs. 절차지향(비객체지향)

일단 객체지향과 절차지향은 반대되는 개념이 아니다. 정확한 비교는 절차지향이라기 보다는 비객체지향이 더 옳다. 자세한 것은 [항목21]에서 다룬다.

비객체지향은 데이터를 중심으로 변경하면서 진행하지만 객체지향은 멤버 함수를 중심으로 진행한다. 다음은 비객체지향 과정이다.



다음은 객체지향 과정이다.



**즉, 절차지향에서는 데이터를 바꾼다.** 이 데이터를 바꾼다면 이 데이터를 참조하는 모든 함수가 바뀌어야 할 수 있다. 그러므로 유지보수 시, 상당한 시간이 소모될 수 있다. **객체지향은 데이터를 그대로 두고** 새로운 제안이 들어오면 어떤 함수만 변경하거나 추가만하면 된다. 왜냐하면 모든 멤버 함수는 기능을 분명히 구분하고 있기 때문이다.

결국 위의 객체지향에서 데이터와 모든 기능을 멤버 함수로 따로 구현한 이 방법을 캡슐화라고 부른다. 곧, ***임의의 값 접근에 멤버 함수를 통하게 한다(=임의의 값을 캡슐화한다.).***

***비객체지향은 데이터와 행위를 구분짓지 않는다. 이를 설계와 연관짓지 않는다. 반면 객체지향에서는 데이터와 행위를 구분짓고 데이터는 건드리지 않는다. 이는 설계나 유지보수시, 다른 부분에 영향을 최소화한다는 보장이므로 빠르고 편한 프로그래밍을 돕는데서 나온 개념이 객체지향 프로그래밍이다.***

###### 객체 지향의 의미

말 자체에는 객체라는 단위로 만들어 설계하는 목적을 의미한다. 여기서 중요한 의미는 추상적 인터페이스 제공이다. 명령어의 복잡한 관계가 아닌 사람이 읽고 다루기 편하도록 인터페이스를 제공하는 것이다. 어떠한 기능을 제공하기 위해서 사람이 명령하듯 하나의 메소드만 호출하면 되지 여러 자료들을 제어하면서 복잡한 명령어를 직접 작성하지 않아도 된다. 이러한 객체를 만들기 위해서 객체 지향 세계를 회전시키는 3가지 요소인 상속, 다형성, 캡슐화가 있는 것이다.

#### 3. 함수형 언어: 순수 함수

###### ‘함수형 프로그래밍’이란 무엇인가?

***순수 함수***: 부작용(side-effect)이 없는 함수, 즉, 함수의 실행이 외부에 영향을 끼치지 않는 함수를 뜻한다. 따라서 순수한 함수는 스레드 안전하고, 병렬적인 계산이 가능하다. 말이 좀 어려운데 정리하여 입력이 같다면 어떠한 상황에서든 출력이 같다는 것이다.

*//이것이 순수 함수다. 숨겨진 입출력이 없다.*

double getDiff\_x(double a, double b){

**return** b - a;

}

double getDiff\_y(double a, double b){

**return** b - a;

}

bool isAvail(double a){

**if** (a <= 1e-10) **return** false;

**return** true;

}

double calcSlope(double a, double b){

*//숨겨진 입력: a, b가 일정해도 getDiff에 따라 결과가 다르다.*

double diff\_x = getDiff\_x(a, b);

double diff\_y = getDiff\_y(a, b);

*//숨겨진 출력: a, b가 일정해도 isAvail()에 따라 결과가 다르다.*

*//이러한 입출력을 부작용(side-effect)라고 부른다.*

**if** (!isAvail(diff\_y)) **return** -1;

**return** diff\_x / diff\_y;

}

**함수형 프로그래밍은 이러한 순수 함수를 작성하는 것이다.**

장점과 단점: *부작용(=숨겨진 입력과 출력)*을 완전히 피할 수는 없다. 대부분의 프로그램은 반환 값을 얻기 위해서가 아니라 어떤 동작을 하기 위해 실행하기 때문이다.   
하지만 프로그램 내부에서는 엄격하게 통제하고자 한다. 우리는 가능한 모든 곳에서 부작용(과 부원인)을 제거하고, 또 제거할 수 없는 경우에는 철저하게 통제할 것이다.

###### ‘함수형 프로그래밍 언어’는 무엇인가?

모든 언어는 순수 함수를 지원한다. 그럼 모든 프로그래밍 언어가 ‘함수형’인가? 아니다. 만약 그렇다면 굳이 용어를 둘 필요도 없을 것이다.

**함수형 프로그래밍 언어는 부작용없는 프로그래밍을 지원하고 장려하는 언어이다.**

더 구체적으로 말하자면, 함수형 언어는 여러분이 가능한한 부작용을 제거하고 그렇지 않은 곳에는 철저히 제어 할 수 있도록 적극적으로 도와주는 언어이다.

#### 4. 일반화 프로그래밍: template

**정의:** 일반화 프로그래밍이란 효과적인 알고리즘, 데이터 구조와 다른 소프트웨어 개념에 대한 추상적인 표현을 찾는 것과 그들의 시스템적인 조직을 다루는 컴파일 과학의 하위 학문이다. -Krzysztof Czarnecki’s Generative Programming인용

**정리:** 어떠한 객체나 행위에 대하여 일반적인 파라미터를 가진 프로그래밍이다. 간단하게 템플릿을 이용한 프로그래밍이다. 이를 가능하게 하기 위한 몇 가지 재료(STL)가 필요하며, 이는 일반화되어 있음을 보장한다. 컴파일 타임에 시행되어 실용적이다. – [항목41]이후 참조

###### 메타 프로그래밍: compile-time

템플릿을 사용하는 프로그래밍 기법으로, 컴파일러에게 프로그램 코드를 생성하도록 하는 방식이다. 즉, 컴파일 시점에 연산을 미리 처리하여 실행 효율을 증진시킬 수 있다.

**template** <int N>

**struct** Factorial {

**enum** { value = N \* Factorial<N - 1>::value };

};

**template** <>

**struct** Factorial<0> {

**enum** { value = 1 };

};

#### 5. STL: Standard Template Library

컨테이너(container)라는 일반화 시켜서 사용할 수 있는 자료구조와 알고리즘, 함수자 등을 포함하는 라이브러리의 모음이다.

STL은 **제네릭 프로그래밍, 효율성을 잃지 않은 추상화, 폰 노이만 구조 그리고 밸류 시멘틱스(value semantics)**을 기초로 구현된다. 이것을 바탕으로 **알고리즘, 컨테이너, 함수자** 그리고 **반복자**라고 불리는 4가지의 구성 요소로 구성된다. 이런 구성 요소들은 위의 4가지 기능이 STL과 연관되어 있음을 보여준다. 이러한 점이 편리한 구현을 제공하며 효율적인 동작을 돕는다.

[항목47]에서 자세히 논한다.

# C++은 여러 하위 언어들의 연합체

= 다중패러다임 프로그래밍 언어

**C++은 한 가지 규칙을 고집하며 작동하는 언어가 아니라, 각각의 규칙을 가진 하위 언어들의 연합체**이다. 위의 기능과 연관되어 C++은 다음과 같이 하위언어로 구분된다.

* C: 블록, 문장, 선행 처리자, 기본제공 데이터타입, 배열 포인터
* 객체 지향 개념의 C++: 클래스를 쓰는 C
* 템플릿 C++: 일반화 프로그래밍 🡪 template meta programming(TMP)
* STL: 컨테이너, 반복자, 알고리즘, 함수 객체

중요한 것은, 각 언어들에서의 규칙과 효율적인 사용 방법이 다르다. 그러므로 어떤 부분을 사용할 지, 그 언어 부분을 사용할 때 어떻게 효율적으로 구현할 지 충분히 이해해야 한다.

결국 C++의 어떤 부분을 사용하느냐에 따라 효과적인 프로그래밍 규칙이 달라진다.

[항목2] #define보다는 const, enum, inline을 떠올리자

가급적 선행 처리자보다 컴파일러를 더 가까이 하자.

# 선행 처리자(preprocessor)란?

#include, #define처럼 앞에 #이 *선행 처리자*이다. 이것은 *선행 처리기*가 한다.

#### 컴파일 순서: source code 🡪 preprocessor 🡪 compiler 🡪 object

#### 선행 처리자로 어떠한 것들이 쓰이고 있나?

실제로 많이 쓰이는 선행 처리자에 대해서 알아보자.

#### #ifdef, #ifndef: 조건문

|  |
| --- |
| #ifndef AAA  #define AAA  #include *<stdio.h>*  #include *<vector>*  ...  #endif |

*#ifdef AAA = if define AAA*: 만약 AAA가 선언되어 있다면 *#endif*까지를 시행한다. *#ifndef*는 그의 반대  
위와 같은 식으로 헤더파일 중복을 방지할 수 있다.

혹은 #pragma를 이용하여 다음과 같이 쉽게 나타낼 수 있다.

#pragma once

*// contents of the header*

같은 용도로 사용되지만 차이가 있다.   
#ifndef..#define방식의 경우에는 파일 내의 #ifndef \_USERCLASS\_H\_ 부분을 평가해야 하므로 파일이 포함될 때마다 파일을 읽어야 하지만, #pragma once는 포함횟수의 관계없이 컴파일러에 의해 처리가 되므로 단 한번만 처리한다. 결과적으로 빌드 시간을 절약할 수 있는 장점이 있다. 일부 구형 컴파일러에서는 지원이 안 되는 경우도 있다. 그러므로 둘 다 사용하면 효율, 이식성 둘 다 얻을 수 있다.

#### ***#pragma***

이는 선행 처리자 확장자로써 컴파일러에게 구체적인 행동 방식을 명시한다. 뒤에 여러 가지 매개변수를 붙여 다양한 행동을 명시할 수 있다.

###### *블록 설정하기*

*#pragma region*을 사용하면 +, -로 확장하거나 축소할 수 있는 코드 블록을 지정할 수 있다.

*// pragma\_directives\_region.cpp*

#pragma region Region\_1

void Test() {}

void Test2() {}

void Test3() {}

#pragma endregion Region\_1

###### 경고 해제하기

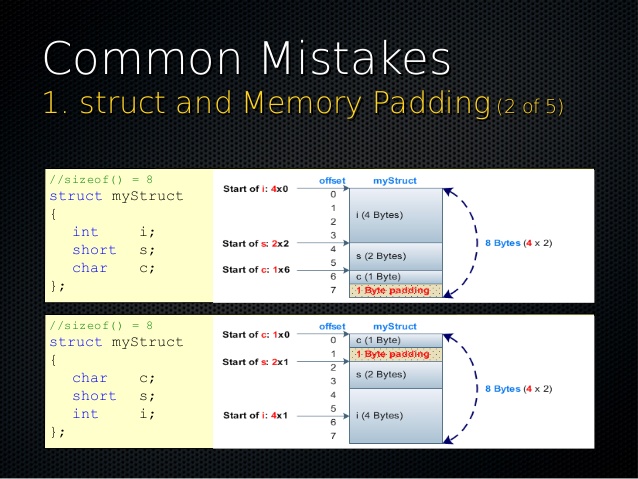
컴파일러가 지나칠 수 있는 경고를 무시하도로 지시한다.

#pragma warning (disable : 4018 )

다음과 같이 해 놓으면, signed, unsigned의 짝의 여부를 경고로 보여준다.

###### 구조체와 공용체 사이즈 변경

구조체는 다음과 같이 저장된다.



멤버 변수를 메모리에서 CPU 레지스터로 한번에 읽을 수 있도록 CPU 레지스터의 읽기 블록에 맞춰 정렬하는 컴파일러의 최적화 작업을 **바이트 패딩**이라 한다.

CPU 레지스터는32비트 운영체제일때 4바이트64비트 운영체제일때 8바이트 단위로 메모리를 읽으므로 구조체를 4바이트 단위로 끊어주면 최적화를 이룰 수 있다.

**struct** A {

char c;

int i;

};

이는 5바이트지만 실제로 sizeof(A) = 8이 된다는 것이다. 그런데 실제 크기와 다르면 네트워크로 실어 나르는 과정 등에서 디코딩하는데 문제가 생길 수 있다.

이 크기를 조절하는 것이 ***#pragma pack***이다.

*// 현재의 pack 사이즈를 저장하지 않고 사이즈를 2로 설정*

#pragma pack(2)

*// 현재의 pack 사이즈를 저장하고 pack 사이즈를 1로 설정*

#pragma pack(push, 1)

. . . . .

*// 저장한 pack 사이즈로 변경*

#pragma pack(pop)

끊어서 읽는 단위의 사이즈를 설정한다. new처리자처럼 기존 것을 저장하고 불러올 수도 있다.

#### ***#define: 매크로 상수 설정***

#define ASPECT\_RATIO 1.653  
우리에게 기호식 이름으로 보이는 ASPECT\_RADIO은 컴파일러에게는 전혀 보이지 않는다. 한마디로 ASPECT\_RADIO라는 기호는 컴파일러가 쓰는 기호테이블에서 밀어버리고 1.653으로 바꾸어서 숫자로 밖에 인식되지 않으므로 이 1.653이 어디서 왔는지 알기 어렵다. 그래서 아래와 같이 선언한다.  
const double AspectRatio = 1.653;  
1. 위와 같은 이유로 기호 테이블에 들어간다. Read-only Memory에 올라간다고 한다. 이는 빠른 디버깅에 도움을 준다. 이 숫자가 어디서 왔는지 알려주기 때문이다.  
2. 컴파일 후 최종 크기가 더 작아질 수 있다. 매크로를 사용하면 모든 ASPECT\_RATIO에 대한 1.653의 사본을 만들어내지만, const는 사본을 딱 한 개만 만든다.  
3. 데이터 형을 명시하기 때문에 역시 디버깅에 도움을 준다.  
[but] #define은 메모리 사용면에서 효율적이다. 선언한 값을 치환만 하기 때문에 메모리 할당이 없다.

###### *기호테이블(symbol table)*

Memory Map과 Symbol 이야기

출처: <http://recipes.egloos.com/5009181>

Linker를 잘~ 다루기 위하여 메모리에 관련한 내용을 이것 저것 다시 한번 정리해 볼 셈입니다. 뭐 여러 가지 복잡한 얘기들이 난무했었다는 걸 상기한다면 이걸 다시 체계화 한다는 것은 상당히 뭔가 복합적인 의미가 있을 것 같습니다.

아방가르드하게 접근을 해본다면, 나름대로의 Category가 있습니다.

그것은 Symbol이라는 것과 그것이 아닌 것들 입니다. 이 Symbol이라는 것은 제가 앞에서 한번 언급한 적이 있습니다. Symbol이란, Linker가 알아볼 수 있는 기본 단위인데, Link를 한 후에는 자신만의 주소를 갖게 되는 특별한 단위를 말합니다. Symbol의 이름은 그 Symbol이 갖는 메모리 영역의 시작 주소를 가리키는 Linker만의 pointer이기요. Debugging시에 이런 Symbol의 이름이 사용되기도 하지요. 이는 전역변수의 이름이나, 함수이름이 그 예이며, 아주 중요한 의미이니 꼭 새겨 두세요. Linker를 위해서 ELF object file내에는 symbol table이라는 것을 두는데, source code에 의하여 참조되는 symbol들의 이름과 위치 정보가 들어 있으며, 다른 file에서 정의된 Symbol을 가져다 쓰는 경우에는 그 해당 file에 Symbol이 없기 때문에, 그 object에서는 symbol table은 완전하지 못합니다.  다른 file에서 선언된 이런 완전치 못한 symbol들은 한 개의 object file내에서 모두 처리되지는 못하는데, 이런 불완전한 symbol들은 linker에 의해서 처리하여 다른 파일에 있는 symbol을 연결하여 사용할 수 있도록 만듭니다. 그러면 Symbol은 Memory에 실제로 적재되는 내용인가 하면? 그건 아니에요. Linker만이 이 Symbol을 참조하며, 실제로 Linker는 이른 Symbol들을 주소로 모두 변환해서 binary로 만들어 준답니다.

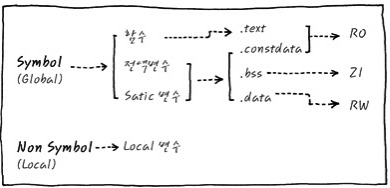
라고 했습니다. 결국 symbol이란 자기 자신만의 주소를 갖는 단위를 symbol이라고 부르며, 저는 Global이라고 명명합니다. - 뭐, 저는 나름 그렇게 구분합니다만, 아니라고 굳이 주장 하시는 분들도 있습니다. 아, 그렇습니까 정도로 하고요- 또한 symbol이 아닌 것들은 자기 자신만의 주소를 갖지 못하는 것들을 말하며, 또한 Local이라고 명명합니다. 이것 참, 곤란한 구분이지만, 저는 편리하게 사용하고 있습니다.

Global에는 함수, 전역변수, static 변수가 Global로 분류될 수 있을 것 같고요, Local에는 나머지 Local 변수입니다.

다시 말해, 함수, 전역변수, static 변수는 자기만의 주소를 가지며, Local 변수는 자기만의 주소를 갖지 못합니다. Symbol들, 즉 Global들이 자기만의 고유 주소를 갖는 다는 것은 다른 파일의 함수들에서도 직접 access하여 만질 수 있는 이유이기도 합니다.

오, 여기에서 한가지 더 나아가자면, ADS등에서는 RW, ZI, RO 세가지 형태의 구분도 있을 수 있는데, 이런 구분은 어떻게 나뉠까 한번 생각해 보시지요. RW는 read-write 로서, 초기값이 있는 전역변수를 의미하고, ZI는 Zero-initialized로서, 초기값이 0인 전역변수를 의미하고, 마지막으로 RO는 Read only로서 수정이 불가능한 const 전역변수와 text인 code를 의미합니다. 한가지만 더 comment하자면 전역변수 중에서도 RO로 분류되는 변수가 있는데, 그 녀석들은 const로 선언된 data들이에요. 후후. const는 Read Only 변수니까요.

그림으로 그려보면, 조금 더 간단할까요?



뭐 이 정도라고 할까요? 간단합니다. .text나 .bss 이런 식의 분류가 더 자세합니다만, RO, ZI, RW는 그 성질이 어떠하냐만 잘 보시면 됩니다. Read only는 읽기만 하니까, const global변수나,code를 의미하고, Read Write는 Global 변수 중 초기값이 있는 경우, 그리고 초기값이 0인 경우에는 모두 ZI로 구분하시면 됩니다. 엄청 간단간단입니다.

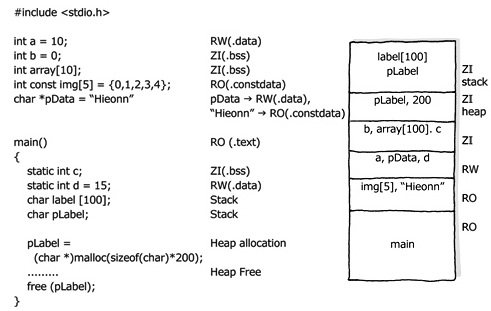
한가지 더, RO냐 RW, ZI만 구분하실 수 있으시면, Flash MCP를 따질 때 어디에 위치할 것이냐에 대해서도 따질 수가 있는 거지요. - Flash란 우리가 흔히 말하는 ROM을 말하는데, 전원이 꺼져도 그 내용을 계속 유지하는 메모리를 말합니다 - 예를 들어 볼까요, RO는 code하고, const data니까 우리는 항상 그 값을 유지하고 있어야 하니까, ROM, 즉 Flash에 그 내용이 들어 있어야 합니다. 또한 RW는 초기 값을 가지고 있는 전역변수니까, 언제라도 modify가능해야 하면서도 초기값을 가지고 있어야 합니다. 그러므로, ROM에 그 위치가 있어야 하며, 또한 그 값이 manipulation이 가능한 RAM에도 위치해야 합니다. 마지막으로 ZI는 그 값이 모두 0이므로 굳이 ROM에 저장될 필요도 없고, RAM에 위치하면 됩니다.

이런 것들을 자유자재로 구사하여 원하는 영역에 symbol 들을 특성에 따라 위치 시킬 수 있는 방법이 있는데, 이것이 linker script인 scatter loading file이라고 부르는 것이죠. 이 내용은 linker script 내용에서 talk talk하시고, 변수들이 메모리에 어떻게 자리 잡는지 정도만 가시지 말입니다.

이런 절대 주소를 갖는 Symbol 외에 절대 주소를 갖지 않는 그때 그때 사용하고 버리는 변수들이 있습니다. 그런 변수에는 Local 변수와 Dynamic Memory Allocation 변수들이 그렇습니다. Local 변수는 stack에 자리 잡으며, Dynamic Memory Allocation 변수들은 Heap이라는 곳에 자리 잡습니다. 이런 변수들은 Temporary로 사용하는 것들로서 굳이 절대 주소를 가질 필요가 없습니다. 이런 녀석들까지 모두 메모리에 잡으려면 엄청난 메모리 양이 필요하겠지요.

지금 symbol 얘기 한 걸로만 이용해서 아래 code에 있는 것들이 메모리에 어떻게 장착 되는지를 확인해 보시도록 하시죠.

RW, ZI, RO로 구분해 보세요.



main()함수 위에 선언되어 있는 전역변수들은 RW, ZI, RO등의 영역에 자리 잡으며, main()함수자체는 RO 영역에 자리 잡습니다. main()함수 내부의 Local 변수들을 잘 보시면, static이라고 선언되어 있는 변수들은 Local이긴 하지만, 앞서 변수의 scope에서 살펴 보았듯이, Global로 취급되므로 RW나 ZI에 속하게 되지요. 나머지 static이 아닌 Local 변수들은 label[100], pLabel은 satck에 pLabel은 heap에 자리 잡습니다. 뭐, 별거 아닌 것 같은 이야기 이지만, 이것만 알면 변수의 life span 따위의 원리는 모든 것이 설명되니, 잘 기억해 주세요.

이런 의미에서라면 Symbol이라 불리 우는 것들은 자신만의 주소를 가질 수 있기 때문에 모든 compile과 link 과정에서 기본 단위가 됨을 잊지 마세요.

마지막으로 한가지 더, Stack과 Heap 을 위와 같이 따로 표시했지만, 전역변수 array로 구현한 경우에는 staqck과 heap은 모두 ZI 영역에 포함 되는 것입니다. 결국 ZI = initial 되지 않은 전역변수 + initial 값이 0인 전역변수 + stack + heap 이라고 봐야 합니다. 으하하.

http://pds15.egloos.com/pds/200906/18/90/c0098890_4a3a213ec0f14.jpg : 자, 여기저기 보면 흘러나오는 용어들 중에 .data, .bss, .constdata, .text등을 다시 한번 구분해 보겠습니다. 얘네들은 RW, ZI, RO로 차례차례 Mapping될 수 있고요. Unix쪽에서 사용하는 용어에요. RW = .data, ZI = .bss, RO = .constdata + .text 로 말이죠. 이런 것들은 모두 Symbol (Global)들만이 그 구분을 가질 수 있죠.

뭐, 쉽게 말해서. .data는 초기값이 있는 전역변수, .bss는 초기값이 없는, 또는 Zero인 전역변수, .constdata는 const로 선언된 전역변수, text는 함수 등의 code가 자리잡을 수 있습니다.

# #define을 const로 바꾸자.

여기서 중요한 것은 **이 매크로 상수 대신 const를 사용하는 것이 좋다**는 것이다. 위와 같은 이유로 장점이 더 많기 때문에 바꾸어준다고 생각하자.

그리고 주의해야 할 점은  
1. 상수 포인터를 정의하는 경우: [항목3]에서 자세히 다룬다.  
2. *클래스 상수* 정의: 클래스 내부에서 사용할 상수를 정의한다.

*//.h*

#include *<iostream>*

**using** **namespace** std;

**class** **GamePlayer**{

**private**:

*//const int NumTurns; // 이것은 사본이 여러 개일 수 있습니다.*

**static** **const** int NumTurns = 10;

*// 오직 1개의 사본: static변수로 NumTurns을 선언합니다.*

**public**:

GamePlayer(int n);

void print(int n);

};

*//.cpp*

#include *"ITEM02.h"*

**const** int GamePlayer::NumTurns;

*// NumTurns을 정의합니다.*

*// 드물게 정의를 따로 해줘야 하는 컴파일러가 있는데 지금은 아닙니다. 안 해도 됩니다.  
// 선언과 정의 둘 다 값을 줄 수 없습니다. 2번 초기화할 수 없습니다.*

GamePlayer::GamePlayer(int n){

}

void GamePlayer::print(int n){

cout << NumTurns << endl;

}

# 나열자(enum)에 대해 알아보자.

열거형 변수라고 말하며, 상수를 정의하는데 많이 사용된다. 다음과 같이 쓰인다.

|  |
| --- |
| **enum** {  mon=100,  tue, *// enum의 값 지정 방식은 이전 원소값의 +1이므로 101이 된다.*  wed=200,  thr, *// 201*  fri=300,  sat, *// 301*  sun  } day;  *//enum을 효율적으로 사용하는 방법*  **typedef** **enum** CDays { **enum** { mon, tue, wed, thr, fri, sat, sun };  *//CDays: 현재의 요일을 반환한다.*  구조체와 형태는 비슷하지만, 동작이나 용도는 완전히 다르다. |

#### 역시 #define 대신 enum을 사용할 수 있다.

나열자(enum)는 *#define*과 구동방식이 비슷하다. 하지만 나열자는 전처리가 아닌 컴파일 도중에 실행된다. 그리고 #define과 마찬가지로 영역 안에 선언한 문자를 쓴다면 메모리에서 불러오는 형태가 아닌 치환해버리는 형태이다. 메모리도 차지하지 않으면서 기호 테이블에도 올라가다니!  
메모리는 기본적으로 int형과 같다. 밑에서도 보이지만, C++11에서 등장한 *enum class*를 사용하면 이를 달리 명시할 수 있다.

#### enum hack(나열자 둔갑술)

|  |
| --- |
| *//내가 원하는 숫자 n으로 다음을 실행한다.*  int a = n;  int scorse[a]; *// 컴파일 에러*  **const** int a = n;  int scorse[a]; *// 컴파일은 되지만 이후 const\_cast로 변경할 수 있다.*  *//enum hack(나열자 둔갑술)이야.*  **enum** { a = n };  int scorse[a]; *// enum안의 값은 변경할 수 없으며 동작 타임은 컴파일이므로 배열에 넣을 수 있다. 게다가 쓸데없는 메모리 할당도 없다.*  나열자 둔갑술을 사용한 사례이다. 앞에서 언급한 특징을 적용한 모습이 보인다. |

#### enum class - 안전하고 쓰기 쉬운 enum

다음과 같은 특징을 갖는다.  
1. 기존의 *enum*은 전방 선언할 수 없었다. 그 이유는 enumerator에 어떤 값이 들어있을지 알 수 없으면 그 크기를 정할 수 없기 때문이다. 하지만 *enum class*는 underlying type을 명시하지 않으면 int 타입과 같은 크기의 변수로 선언되고, int 값 안에 들어가지 못할 값을 지정하면 컴파일 에러를 발생시킨다.  
2. 기존 *enum*의 또 다른 문제는 enumerator의 이름의 범위가 한정되지 않는다는 것이다.  
3. *enum class*는 정수형 변수로 암시적 변환이 되지 않는다. *enum class*를 정수형 변수처럼 사용하려고 하면 컴파일 에러를 발생시킨다. 만약 정수형 변수로 사용하고 싶으면 *static\_cast*를 이용해 명시적으로 캐스팅해서 사용해야 한다.

|  |
| --- |
| **enum** **class** **StateType** : unsigned short {  ST\_NONE = 0,    ST\_IDLE,  ST\_MOVE,  ST\_ATTACK,  ST\_DEAD,    ST\_MAX,  };    **enum** **class** **MoveType** {  MT\_NONE = 0,    MT\_WALK,  MT\_DASH,  MT\_JUMP,  MT\_CROUCH,    *// 열거자의 scope가 enum class {};안으로 국한되므로,*  *// StateType의 열거 이름과 같은 것을 사용해도 무방하다.*  ST\_DEAD,    MT\_MAX,  };    int \_tmain(int argc, \_TCHAR\* argv[]) {  *// StateType의 열거이름은 StateType의 scope 안에서만 유효하다*  *// 따라서, 반드시 StateType에 대한 영역 지정을 해야 한다*  *// error C2065: 'ST\_IDLE': 선언되지 않은 식별자입니다.*  StateType a = ST\_IDLE;    *// enum class 열거자를 정수형에 암시적으로 대입할 수 없다*  *// error C2440: '초기화 중': 'StateType'에서 'unsigned char'(으)로 변환할 수 없습니다.*  unsigned char b = StateType::ST\_IDLE;    *// StateType은 unsigned short로 타입 지정되었으므로, 그 크기는 2 바이트이다*  int c = **sizeof**(StateType);    MoveType d= MoveType::MT\_WALK;    *// 서로 다른 타입의 열거자끼리는 비교가 불가능하다*  *// error C2676: 이항 '==': 'MoveType'이(가) 이 연산자를 정의하지 않거나*  *// 미리 정의된 연산자에 허용되는 형식으로의 변환을 정의하지 않습니다.*  **if** (d == StateType::ST\_IDLE)  {  }  *// 다른 타입의 enum으로는 대입이 불가능하다*  *// error C2440: '초기화 중': 'StateType'에서 'MoveType'(으)로 변환할 수 없습니다.*  MoveType e = StateType::ST\_IDLE;    *// 정수 값을 enum에 대입시킬 수 없다*  *// error C2440: '초기화 중': 'int'에서 'StateType'(으)로 변환할 수 없습니다.*  StateType f = 1;    **return** 0;  } |

결론은 *enum class*를 쓰는 것이 좋다.

# 매크로를 사용할 텐가?

#### 매크로 역시 #define대신 inline을 사용한다.

|  |
| --- |
| #define MAX(a, b) f((a) > (b) ? (a) : (b)) *// #define으로 매크로를 사용합니다.*  **template**<**typename** T> *// 인라인 함수로 매크로를 만듭니다.*  **inline** int Max(**const** T& a, **const** T& b){  **return** f(a > b ? **a** : b);  }  int main(){  int a = 11, b = 0;  cout << MAX(++a, b) << endl; *// a가 12가 아닌 13으로 적용됩니다.*  a = 11;  cout << Max(++a, b) << endl;  *// 결과*  *// MAX: 13000*  *// Max: 12000*  }  *#define*사용시, 나올 수 있는 오류이다. 그러므로 template + inline형태의 함수를 작성하는 것이 바람직하며, 두 가지 방법으로 작성할 수 있어야 한다. |

*단순한 상수를 쓸 때는 #define보다 const, enum을 사용합시다.*

*매크로 또한 #define보다 inline을 사용합시다.*

[항목3] const를 남발하자

# ‘const’가 무엇인가?

const키워드는 외부변경 불가능하게 한다.

#### 외부변경이 불가능해야 하는 이유는?

일종의 소통이다. 어떤 객체는 변하면 안 된다라는 것을 컴파일러 및 프로그래머끼리 알려준다고 볼 수 있다. 성능 향상, 디버깅에도 도움이 된다.

# ‘const’는 어떻게 쓰지?

예시를 들어 간단하게 설명한다.

|  |
| --- |
| **const** char **\*** **const** p = abc;  *// \*을 기준으로*  *// 왼쪽의 const(const char)는* ***포인터 값(데이터)****를 상수화,*  *// 오른쪽의 const(const p)는* ***포인터(주소)****를 상수화한다.* |

함수에 대한 사용은 밑에 [용도4]에서 자세히 설명한다.

# ‘const’는 어디에 쓰면 좋을까?

용도를 다음과 같이 정리한다.

#### [용도1] 전역 변수 및 네임스페이스 유효범위의 상수 선언 및 정의

클래스 바깥에서 선언할 수 있다.

#### [용도2] 클래스 내부에서 정적 멤버(static) 혹은 비정적 멤버 상수화하기

클래스 내부에서도 사용할 수 있다. 용도1과 범위의 차이일 뿐이다.

#### [용도3] 포인터나 그의 데이터를 상수화할 수 있다.

|  |
| --- |
| *//포인터 주소는 바꿀 수 없지만 그 안의 값은 바꿀 수 있다.*  **const** std::vector<int>::iterator iter = vec.begin();  \*iter = 10; *// 가능*  iter++; *// 불가능*  *//포인터 안의 값은 바꿀 수 없지만 주소는 바꿀 수 있다.*  std::vector<int>::const\_iterator cIter = vec.begin();  \*iter = 10; *// 불가능*  iter++; *// 가능*  std::iterator를 통하여 예시를 들었다. 되도록 둘 다 상수화하길 권장한다. |

#### [용도4] 함수 선언에서 const를 선언할 수 있다.

함수에서는 크게 4가지로 const를 선언할 수 있다.

[선언1] 반환 값에 const선언

|  |
| --- |
| **const** Factory **operator**\* (**const** Factory& unit1, **const** Factory& unit2);  Factory a, b, c;  a \* b = c; *// 무의식으로 키보드를 잘못 눌러 비교가 아닌 대입이 된다.*  기본 제공 타입은 위와 같은 연산을 허용하지 않는다.  *사용자 정의 타입은 기본 제공 타입과의 쓸데없는 호환을 거부한다(=[항목18]).* r-value와 함수 반환값의 const *r-value*란, 메모리를 가리킬 필요가 없는 값이다. 어떤 함수에서 반환되는 값이 기본 제공 타입일 경우, 이가 허용되지 않는다는 말은 반환 값이 *r-value*라는 말이다.  int power(int n, int e){  **if** (e == 0) **return** 1;  **if** (e == 1) **return** n;  **if** (e & 1) **return** n\*power(n, e - 1);  int ret = power(n, e / 2);  **return** ret\*ret;  }  int main(){  int n = 2, e = 10;  *//const와 상관없이 아래 같은 코드는 작동하지 않는다.*  *//기본적으로 기본 제공 타입에는 이러한 작동을 허용하지 않는다.*  power(n, e) = 2 << 10;  }  그럼 반환 값이 사용자 정의 객체라면 어떨까?  **class** **Rational**{  **public**:  Rational(int \_a, int \_b) : a(\_a), b(\_b){}  *//반환 값에 const가 없는 사용자 정의 함수*  **friend** Rational **operator**\*(**const** Rational& lhs, **const** Rational& rhs){  **return** Rational(lhs.a\*rhs.a, lhs.b\*rhs.b);  }  Rational **operator**\*=(**const** Rational& rhs){  a \*= rhs.a, b \*= rhs.b;  **return** \***this**;  }  void printR(){  cout << a << "/" << b;  }  **private**:  int a, b;  };  int main(){  Rational a(5, 3), b(3, 2), c(3, 3);  *//사용자 정의에서는 이 코드는 적법합니다.*  *//이 때, Rational operator\*앞에* ***const를 붙이면 r-value가 되어 대입을 방지***  a \* b = c;  a.printR();  cout << endl;  }  위와 같이 반환 값에 새로운 값을 대입하는 경우는 일반적으로 타당하지 않습니다(반환 값에 대입할거면 왜 반환을 하는가, 만약 배열처럼 임의의 인덱스에 새로 채우고 싶다면 참조를 반환해야 함). 그러므로 **반환 값 앞에 const는 필히 사용**해주는 것이 좋습니다. |

[선언2] 매개변수에 const선언

const타입의 지역 객체와 같다. 해당 범위의 함수 내에서 변하지 않음을 보장하고 싶다면 **역시나 꼭 사용해주는 것**이 좋다.

**[선언3]** **상수 객체에 const선언 (**🡪 [선언4] 멤버 함수에 const선언**)**

상수 멤버 함수(const 선언된)라는 것은 무슨 의미인가요?

멤버 함수 중 상수인 객체에 대하여 사용할 함수. ***즉, 일반 객체에 쓸 함수와 상수 객체에 쓸 함수가 구분되어야 한다.***

**class** **TextBlock**{

**public**:

TextBlock(**const** char\* ntext, size\_t nsize) : tb\_size(nsize) {

**for** (size\_t i = 0; i < nsize; i++) text[i] = ntext[i];

}

size\_t size() **const**{ **return** tb\_size; }

*//operator함수를 상수용과 비상수용 2가지로 오버로딩합니다.*

*//상수용 함수입니다. 반환 값은 r-value이며 내부는 절대로 변하지 않습니다.*

**const** char& **operator**[](std::size\_t position) **const**{

**return** text[position];

}

*//비상수용 함수입니다. 대입이 허용되며(참조값 반환: l-value) 내부 값은 변합니다. 이 객체가 상수가 아니라면 이 함수를 사용할 것입니다.*

char& **operator**[](std::size\_t position){

**return** text[position];

}

**private**:

char text[500];

size\_t tb\_size;

};

상수 멤버 함수는 왜 써야 하나요?

위에서도 언급했지만 상수 객체에 대해 쓸 함수가 필요할 때 이 함수가 필요하다. *객체 전달을 상수 객체를 사용하여 참조자로 진행하면 효율을 증대시킬 수 있다.* 그러므로 *상수 멤버 함수가 준비되면 상수 객체를 사용하여 효율을 증대*시킬 수 있다. 이 원리는 [항목20]에 자세히 설명되어 있으며 요약하면 값을 온전히 복사하는 것보다 주소만 넘기는 것이 효율적이기 때문이다. 아래 코드에서 *reference-to-const*로 진행한 코드를 확인한다.

일반 객체 vs. 상수 객체

|  |
| --- |
| *//인자 역시 reference-to-const로 진행하면 효율적입니다.*  void printTb(**const** TextBlock& ctb){  **for** (int i = 0; i < ctb.size(); i++)  cout << ctb[i];  cout << endl;  }  int main(){  char\* text = "abcde";  TextBlock tb(text, 5);  **const** TextBlock ctb(text, 5);  *//tb를 넘기는 것 보다는 ctb를 넘기는 것이 효율적입니다.*  printTb(tb);  printTb(ctb);  }   1. 인자 전달에 상수 객체 참조 전달 2. 상수화된 객체를 만들어 참조 전달 |

그럼 같은 함수가 일반용과 상수용으로 2개가 있어야 하지 않겠는가?

그래서 함수 중복을 방지한 오버로딩이 필요하다. const용과 아닌 것으로 2개의 함수가 필요하다. 아래에서 다시 설명한다.

[선언4] 함수 전체에 const선언(멤버 함수에 const선언)

위와 같은 이유로 상수 멤버 함수를 만들어야 한다. 상수 멤버 함수는 내부의 내용을 변경하지 않는다고 보장하며 데이터 멤버를 변경하려는 함수를 사용한다면 컴파일 오류가 생긴다. 아래 코드와 같다.

**class** **TextBlock**{

**public**:

*//...*

*//const가 있는 함수*

size\_t size() **const**{ **return** tb\_size; }

*//const가 없는 함수*

void change(){ tb\_size++; }

};

int main(){

char\* text = "abcde";

TextBlock tb(text, 5);

**const** TextBlock ctb(text, 5);

*//상수 객체(ctb)는 내부를 변하게 하는 함수는 사용할 수 없다.*

ctb.change();

}

***상수 객체를 위하여 데이터를 변경하지 않겠다는 함수를 선언할 필요가 있다.*** *그럼 어떻게 만드는가? 함수 전체에 const선언을 하여 만든다. 마지막에 const키워드를 붙이는 것이 그것이다. 이는 비트수준 상수성을 제공한다.*

비트수준 상수성(=물리적 상수성)

어떤 바이트도 수정할 수 없게 만든다. 아주 그 자체의 의미와 적합해 보이지만 함정이 있다. 포인터로 접근하면 내부 값을 변경하는데 아무런 제약이 없다는 점이다.

이 문제를 어떻게 해결하지? 🡪 논리적 상수성

바뀔 수 있지만, 사용자측에서 알아채지 못하게 설계한다. 하지만 상수 멤버 함수 안에서는 어떠한 변수도 수정할 수 없게 되어있으므로 이가 어려울 수 있는데 *mutable키워드로 이를 상수 객체에서도 값을 변경하도록 해준다.*

###### 상수 멤버 함수를 만들어보자.

위에서 언급한대로 효율성을 위해 상수 객체 참조를 이용해야 하고 이에 대해 상수용과 비상수용이 필요하다.

**class** **TextBlock**{

char text[500];

**public**:

**const** char& **operator**[](std::size\_t position) **const**{

*//이러쿵저러쿵 검사*

**return** text[position];

}

char& **operator**[](std::size\_t position){

*//이러쿵저러쿵 검사*

**return** text[position];

);

}

};

이렇게 작성하면 자연스레 코드 중복이 발생할 수 있다. 어떻게 하면 중복없이 한 개의 operator[]만 적절하게 설계할 수 있을까?

const를 날려버린다.

상수든 비상수든 const를 날리고 비상수 함수만 호출한다는 말인데, 통념적으로 좋지 않다. - [항목27] 참조

비상수 함수가 상수 함수를 호출하게 만든다.

상수 함수는 최대한 건드리지 않는 것이 좋다.

**const\_cast**는 해당 대상의 const성질을 벗길 수 있다.  
**static\_cast**는 해당 대상의 const를 붙일 수 있다.

|  |
| --- |
| **class** **TextBlock**{  char text[500];  **public**:  *//operator함수를 오버로딩합니다.*  *//상수용 함수입니다.*  **const** char& **operator**[](std::size\_t position) **const**{  **return** text[position];  }  *//비상수용 함수입니다. 반드시 비상수용에서 상수용을 호출해야 합니다.*  char& **operator**[](std::size\_t position){  *//비상수 함수에서 상수 함수를 반환합니다.*  *//비상수용이므로 값을 변경하기 위해 const제약을 해제합니다.*  **return**  **const\_cast**<char&>(  *//값을 변경하고 다시 상수화시킨다.*  *//\*this는 현재 이 함수를 가리킨다.*  *//지금은 캐스팅 이후이므로 상수용 함수가 되고*  *//상수용 operator을 불러올 수 있다.*  **static\_cast**<**const** TextBlock&>(\***this**)[position]  );  }  };  중복 방지 🡪 비상수가 상수 호출 🡪 상수 해제 🡪 값 변경 🡪 상수화 🡪 반환 |

반대로 상수 함수가 비상수 함수를 호출하는 것은 어떨까?

상수 멤버 함수는 내부를 변경하지 않겠다는 보장([선언4])이 있다. 만약 비상수 객체(\*this)에서 상수 함수를 호출한다 가정하자. 여기서 굳이 변경을 하겠다면 \*this에 붙은 const를 붙였다가 떼어내야 한다. 위의 제약을 어김으로써 안전성에 문제가 크다. 반면 비상수 멤버 함수는 그런 것이 없다. 아무런 제약도 없으며 위에서처럼 여기에 const를 붙이는 것 정도는 안전하다.

***const를 붙여 선언하면 컴파일러가 에러를 잡아내는데 도움을 줍니다.***

***컴파일러는 물리적 상수성을 지키지만 사용자는 논리적 상수성을 지키며 프로그래밍 해야 합니다.***

***상수 멤버와 비상수 멤버 함수를 만들 때 코드 중복을 방지합니다.***

[항목 4] 객체를 반드시 초기화!

초기화: 어떤 객체에 최초의 값을 부여하는 것

# C++에서의 초기화

초기화하지 않으면 심각한 문제를 일으킨다. 원하지도 않는 결과를 보여줌에도 어떠한 오류도 없이 보여준단 말이다. 그래서 우리는 반드시 초기화를 해주어야 하는데 중요한 점은 C++에서는 언어 자체에서 객체에 대한 초기화를 보장하지 않는다는 점이다. 몇 가지 규칙이 있기는 하다. 아래에서 보자.

#### C++에서의 초기화 규칙은?

C부분만을 쓰고 있으며 초기화에 런타임 비용이 소모될 수 있는 상황이라면 값이 초기화된다는 보장이 없다.  
🡪 이 말은 실행 중의 생성되는 객체는 초기화 보장이 없다는 것이다. 함수에서 만든 객체에 이상한 값이 들어가는 반면 전역 변수는 확실히 초기화를 보장됨이 그것이다.   
그렇지만 C가 아닌 부분에서는 그러한 보장을 가질 수도 있다(배열 vs. 벡터).

#### 그러므로 C++을 사용하는 우리는 항상 초기화를 직접 해주어야 한다.

# C++에서 초기화 하기

그냥 값을 넣는다고 모두 초기화가 완성되진 않는다. **우리가 반드시 기억해야 할 몇 가지 요소가 있다.**

#### [요소1] 기본 제공 타입 초기화

|  |
| --- |
| *// 기본 제공 타입(int, 포인터)의 초기화*  int x = 0;  **const** char \* text = "YTK is programming genius.";  double d;  std::cin >> d; *// 입력 스트림에서 읽음으로써 초기화 수행* |

위와 같은 경우를 제외한 나머지 초기화는 모두 *[요소2]*에 해당한다.

#### [요소2] 객체 초기화 = 생성자

반드시 생성자와 대입의 차이를 구별해야 한다. 다음에서 둘의 차이를 코드로 보여준다.

|  |
| --- |
| int masterPlan = 1;  **class** **Archer**{  **public**:  Archer(**const** std::string& name, **const** int rank);  **private**:  std::string myName;  int myRank;  **const** bool isMan;  int& weapon;  };  *//아래 함수는* ***대입****하고 있습니다.*  *//상수와 참조자는 반드시 대입이 아닌 생성자로 만들어야 합니다. 아래 식은 오류입니다.*  Archer::Archer(**const** std::string& name, **const** int rank){  myName = name;  *//isMan = true;*  *//weapon = masterPlan*  }  *//아래 함수는* ***초기화****하고 있습니다. 멤버 초기화 리스트를 사용합니다.*  Archer::Archer(**const** std::string& name, **const** int rank)  : myName(name), myRank(rank), isMan(true), weapon(masterPlan)  {}  위의 함수(대입)은 myName에 임의의 값으로 초기화 한 다음에 내가 원하는 값으로 대입하는 모습이므로 2번의 복사가 일어난다.  반면 아래 함수(초기화)는 곧바로 값을 집어넣으므로 1번의 복사가 일어난다. 그러므로 훨씬 효율적이라고 볼 수 있다. |

#### [요소3] 상수와 참조자는 반드시 초기화 리스트에 넣어주어야 한다.

위의 코드에서 볼 수 있다.

#### [요소4] 기본 클래스와 파생 클래스

상속관계에 있는 객체라면 초기화 순서는 어떨까? 기본 클래스가 먼저 초기화된다. 이후 파생 클래스가 초기화된다.

#### [요소5] 클래스 데이터 멤버(클래스 안에서)는 선언된 순서대로 초기화된다.

초기화 리스트에 넣어진 순서와 상관없이 선언된 순서대로다.

#### [요소6] 비지역 정적 객체의 초기화 순서는 개별 번역 단위에서 정해진다.

정적 객체, 번역 단위, 비지역의 의미에 대해서 살펴본다.

###### 정적 객체(static object): 생성된 시점부터 프로그램이 끝날 때까지 살아있는 객체. 스택이나 힙 기반의 객체는 정적 객체가 될 수 없다.

객체는 아래와 같이 5가지의 형태로 존재한다.  
1. 전역 객체  
2. 네임 스페이스(namespace) 유효범위에서 정의된 객체  
3. 클래스 안에서 static으로 선언된 객체  
4. 함수 안에서 static으로 선언된 객체  
5. 파일 유효범위(extern)에서 static으로 정의된 객체

1, 2, 3, 5 🡪 ***비지역(non-local) 정적 객체***  
4 🡪 *지역(local) 정적 객체*

아래는 위의 5가지 방법으로 정적 객체를 선언한 모습이다.

**class** **DarkWorld**{

**public**:

**static** int counter;

void countUp() { counter++; }

std::size\_t numPlayer() **const**;

};

std::size\_t DarkWorld::numPlayer() **const** { **return** counter; }

*//static 멤버는 아래와 같이 따로 초기화합니다.*

int DarkWorld::counter = 0;

*//비지역 정적 객체*

DarkWorld dw1;

**namespace** night{

DarkWorld dw2;

}

**class** **Black**{

**public**:

**static** DarkWorld dw3;

**static** DarkWorld& getInstance() { **return** dw3; }

};

DarkWorld Black::dw3;

**extern** DarkWorld\* dw5 = **new** DarkWorld;

*//자기 자신을 지역 정적 객체로 만들어 반환합니다.*

DarkWorld& dw4(){

**static** DarkWorld dw;

**return** dw;

}

void isAvail(DarkWorld& dw){

dw.countUp();

std::cout << "dw counter: " << dw.numPlayer() << '\n';

}

int main(){

isAvail(dw1);

**using** **namespace** night;

isAvail(dw2);

isAvail(Black::getInstance());

isAvail(\*dw5);

isAvail(dw4());

*/\*output*

*counter = 1*

*counter = 2*

*counter = 3*

*counter = 4*

*counter = 5*

*\*\*\*\*\*\*\*\*\*\*\*\*/*

**return** 0;

}

정적 객체의 몇 가지 특징을 살펴본다.  
1. 한 프로그램에서 하나의 하나의 객체만 만들어진다. 그러므로 여러 개의 변수로 선언했지만 데이터는 모두 하나로 공유된다.  
2. 선언했다고 해서 전체 프로그램에 대해 초기화가 보장된 것은 없다. *- [요소7]*3. [항목9]에서 static에 대해 약간의 의미를 더 설명한다.

###### 번역 단위: 컴파일을 통해 하나의 목적 파일을 만드는 바탕이 되는 소스 코드. 즉, 하나의 소스 파일을 말한다.

소스파일과 목적파일, 컴파일, 링크

이 낱말들의 의미와 관계를 이해해야 위의 말을 제대로 이해했다고 볼 수 있다.

<http://ith.kr/chair/cpp/cpp103.html>

1.3.소스파일과 컴파일, 링크

1.3.1. 어떤 과정을 거쳐 프로그램이 만들어지는가?

언어 프로그램을 이용하여 실행파일인 \*.COM \*.EXE 파일을 만듭니다

이제 **컴파일러라고 부르는 언어프로그램을 이용해서 컴퓨터프로그램을 만드는 과정**을 간단하게 설명하겠습니다. 컴퓨터프로그램을 만든다는 이야기는 실행파일에 해당하는 \*.COM 파일과 \*.EXE 파일을 만든다는 뜻입니다. 물론 그외의 부속 파일들도 만들어야 하지만 가장 중요한 파일인 실행파일을 만드는 것이 제일 중요합니다. 컴퓨터를 사용하는 분들은 \*.COM 파일과 \*.EXE 파일을 어떻게 만들고 그것이 어떻게 동작하는지 매우 궁금할 겁니다. 간단하게 언어프로그램을 이용해서 일반적인 프로그램을 짜는 과정을 정리하면 아래와 같습니다. 

실행파일 만들기 1: C++언어 문법에 맞게 작문합니다. 다른 말로 **소스파일**을 짠다고 합니다.

편집기라고 부르는 에디터라는 프로그램을 이용해서 C++언어의 약속에 맞게 프로그램을 짜기 시작합니다. 즉 C++언어 문법에 맞는 문장을 작문하는 겁니다. 마치 한국인이 영어작문을 하는 것과 같습니다. 단지 영어 대신 C++언어라는 것으로 언어가 바뀌었을 뿐입니다. 이때 C++언어 문법에 맞게 작성한 문서파일을 원시파일 또는 소스파일(Source File)이라고 말하고 에디터를 이용해서 C++언어 문법에 맞도록 작문하는 것을 흔히 프로그램을 작성한다, 또는 프로그램을 짠다고 표현합니다. 

실행파일 만들기 2: 소스파일을 컴파일러 프로그램을 이용해 기계어파일로 변환합니다

소스파일의 내용에 문법적인 문제가 없다면 소스파일을 컴파일러라는 프로그램을 이용해서 컴파일합니다. 컴파일은 번역한다는 뜻을 지니므로 컴파일러는 번역기에 해당합니다. 무엇을 무엇으로 번역할까요? 사람들이 알아볼 수 있는 소스파일을 기계가 알아볼 수 있는 기계어파일로 번역하는 겁니다. 

실행파일 만들기 3: 컴파일 과정을 통해 만들어진 **목적파일**들을 링커를 이용해 연결합니다

컴파일러라는 프로그램으로 소스파일을 컴파일하면 목적파일, 또는 중간파일이라는 \*.OBJ 파일이 생성됩니다. 이렇게 생성된 목적파일을 링커라는 프로그램을 이용해서 연결시킵니다.   
  
링커를 이용해서 목적파일을 연결시키면 마침내 우리가 사용하는 실행파일인 \*.COM이나 \*.EXE 파일이 하나 만들어집니다. 이 파일은 실행이 되는 프로그램이므로 파일의 이름을 치면 다른 프로그램과 마찬가지로 실행이 됩니다. 

\*\*요약: 컴퓨터에서 사용하는 실행 프로그램은 C++의 문법에 맞게 짠 소스파일을 컴파일하고 링크해서 만듭니다.

1.3.2. 컴파일러란?

컴파일러는 소스파일을 기계어로 번역해주는 프로그램입니다.

앞서도 몇 차례에 걸쳐서 설명했지만 컴파일러란 사람이 C++언어 문법에 맞추어서 쓴 명령어들을 컴퓨터가 이해할 수 있는 기계어로 번역해주는 프로그램입니다. 그리고 보통 언어프로그램이라고 하면 흔히 이런 컴파일러를 가리킵니다.   
  
다시 말하지만 컴퓨터라는 기계는 0과 1의 이진수밖에는 모릅니다. ON과 OFF 아니면 +, -밖에는 모르는 단순한 기계입니다. 반면 사람은 숫자나 문자로만 모든 것을 이해합니다. 때문에 사람은 자신들이 알아볼 수 있도록 숫자와 문자로 명령을 기록하는데, 이를 기계가 이해할 수 있는 언어인 기계어로 번역해주는 프로그램이 컴파일러입니다. 즉 ABC를 0과 1의 조합으로 바꾸어주는 것입니다. 

\*\*요약: 컴파일러란 사람이 짠 소스파일을 기계가 알아들을 수 있는 기계어로 바꾸어주는 일을 하는 프로그램입니다.

1.3.3. 목적파일 \*.obj 파일을 만들어 사용하는 이유

C로 프로그램을 짜면 컴파일러로 컴파일을 해서 목적파일(Object File)인 \*.obj 파일을 만들어줍니다. 그러면 다시 링커라는 프로그램을 이용해서 우리가 원하는 \*.exe 파일을 만듭니다. 그러면 왜 한번에 바로 소스파일을 \*.exe 파일로 만들지 않고 \*.obj라는 중간파일을 만드는 것일까 하는 의문이 들겁니다.   
  
\*.obj라는 파일을 만드는 이유는 여러 가지 이유가 있지만 한 마디로 설명하면 프로그램을 좀더 편하게 만들기 위해서입니다. 프로그램을 직접 만들다보면 왜 \*.obj 파일이 필요한지 알 수 있지만 간단하게 예를 들겠습니다. 

목적파일이 필요한 이유 1: 소스파일의 보안을 유지하면서 기능만 제공할 수 있습니다.

(가)라는 사람이 시계 프로그램을 C++언어로 짠 뒤에 이를 clock.exe라는 실행 프로그램으로 만들었다고 합시다. 그리고 (나)라는 사람이 사람은 달력프로그램을 만들어서 calendar.exe로 만들었다고 합시다. (나)는 (가)가 만든 시계프로그램과 자신의 달력프로그램 합쳐서 cc.exe라는 새로운 달력프로그램을 만들고 싶습니다. 가장 좋은 방법은 clock.exe의 C++언어 소스파일을 제공받아서 자신의 달력프로그램에 합친 후에 이를 컴파일하고 링크해서 cc.exe를 만드는 방법입니다.   
  
그러나 소스파일이라는 것은 그 사람이 지닌 고도의 기술과 경험에 해당하며 일종의 회사기밀입니다. 프로그램의 소스 자체가 가장 중요한 재산이므로 프로그래머는 함부로 이를 공개하지 않습니다. 그래서 (가)는 소스파일 대신에 중간파일인 clock.obj를 (나)에게 제공합니다. 그럼 (나)는 clock.obj와 calendar.obj를 링커 프로그램으로 연결(linking)해서 cc.exe 파일을 만들 수 있습니다. 보안성도 유지되고 서로 각기 다른 프로그램을 만들어서 합칠 수 있으므로 효율적입니다. 

목적파일이 필요한 이유 2: 분업화, 모듈화가 가능합니다

꼭 보안성 문제 때문만은 아닙니다. obj파일을 이용하면 하나의 프로그램을 짤 때 열 명이 각기 다른 기능을 가진 프로그램을 짜도 됩니다. (가)는 시계프로그램을 짜고, (나)는 달력프로그램을 (다)는 문서편집프로그램을 짠 뒤에 각기 짠 프로그램을 \*.obj 파일로 만들어서 총기획자에게 보내주면 작업이 끝납니다. 프로그램 기획자는 (가) (나) (다)로부터 받은 세 개의 파일을 링크프로그램으로 연결만 하면 되기 때문입니다.   
  
따라서 하나의 프로그램을 혼자서 낑낑대며 짤 필요가 없습니다. 여러 명에게 나누어 준 뒤에 나중에 합치기만 하면 되므로 인력을 효율적으로 관리할 수 있으며 프로그램 개발에 걸리는 시간이 줄어듭니다. 만약 하나의 소스로 방대한 프로그램을 짜야 한다면 한 사람이 방대한 소스파일의 내용을 모두 파악하고 있어야 하며, 한 사람이 모두 짜야 하므로 시간이 무척 오래 걸리게 됩니다.   
  
또한 프로그램을 만드는 실무자들은 자신이 맡은 부분만 담당하게 되므로 프로그램 전체의 기밀을 유지하는데도 도움이 됩니다. 

목적파일이 필요한 이유 3: 컴파일 시간 등을 줄일 수 있습니다

보안과 분업화라는 기업적인 측면의 장점 외에도 개인에게 도움이 되는 장점이 있습니다. 목적파일을 만들어 사용하면 컴파일 시간을 줄일 수 있다는 점입니다.   
  
요즘은 컴퓨터와 언어프로그램이 워낙 좋아져서 컴파일하는 시간이 많이 안걸리지만 예전에는 프로그램 하나 컴파일하는데 하루 종일 걸렸습니다. 지금도 조금 덩치가 큰 프로그램을 컴파일할 때는 몇 십분에서 몇 시간씩 걸립니다. 소스 길이가 몇 만 줄, 몇 십 만 줄이나 되는 덩치 큰 프로그램을 개발한다고 생각해보시면 이해가 될 겁니다. 몇 십 만 줄에서 한 줄 고치고 컴파일하면서 몇 십분 또는 몇 시간씩 기다려야 한다면 얼마나 시간낭비입니까?   
  
그래서 문제가 없는 프로그램이나 함수 부분을 컴파일해서 \*.obj 파일로 만들어놓습니다. 그리고 새로 고친 부분이나 새롭게 추가한 부분만 컴파일을 합니다. 그러면 몇 십분씩 걸리는 컴파일 시간이 몇 초로 줄어듭니다. 이렇게 해서 이상이 없으면 다시 \*.obj 파일로 만들어놓고, 또 새로운 함수나 프로그램을 만듭니다. 그리고 나중에 최종적으로 그동안 만든 \*.obj 파일을 하나로 링크시키기만 하면 됩니다. 이렇게 \*.obj 파일을 이용해서 컴파일을 하면 컴파일 시간이 획기적으로 줄어듭니다.   
  
만약 \*.obj 파일을 사용하지 않고 바로 \*.exe 파일을 만들어야 한다면 단 한 줄을 고치고도 수 십 만 줄이나 되는 소스파일을 전부 컴파일하느라고 많은 시간을 낭비해야 합니다. 또한 한꺼번에 수 십 만 줄을 읽을 수 있는 비싼 에디터를 써야 하며, 빠른 컴파일을 위해 컴퓨터도 매우 높은 사양의 기종으로 갖추어야 합니다. 

목적파일이 필요한 이유 4: 확장성과 이식성을 높일 수 있습니다

그외에도 obj 파일을 이용할 때의 장점은 많습니다. 시계프로그램을 clock.obj로 한 번 만들어두었다면 다른 프로그램을 짤 때는 시계프로그램을 다시 짤 필요가 없습니다. 나중에 링크를 할 때 clock.obj 파일만 연결시켜주면 간단하게 해결되기 때문입니다. 따라서 이식성이 매우 높아집니다.   
  
또한 유용한 함수나 소스파일을 컴파일하여 목적파일로 만든 다음에 목적파일만 합쳐놓은 라이브러리 파일을 만들어 사용할 경우 다양한 기능의 함수를 손쉽게 확장시킬 수 있습니다. 대표적인 것이 한글 라이브러리입니다. 한글 라이브러리를 이용하면 따로 한글을 구현하는 방법을 직접 구현하지 않더라도 다른 사람이 만들어놓은 라이브러리를 이용해서 손쉽게 한글을 구현할 수 있습니다. 이와 같은 확장성 때문에 프로그램을 만들 때 목적파일을 만들어 사용하는 것입니다. 

\*\*요약: 목적파일을 만들어 사용하는 이유는 소스파일의 보안, 작업의 분업화, 프로그램의 모듈화, 이식성과 확장성을 높이기 위해서입니다.

1.3.4. 링크와 링커 프로그램

목적파일을 연결해 실행파일로 만드는 일을 **링크**라고 하며, 링크작업을 해주는 프로그램을 링커라고 합니다

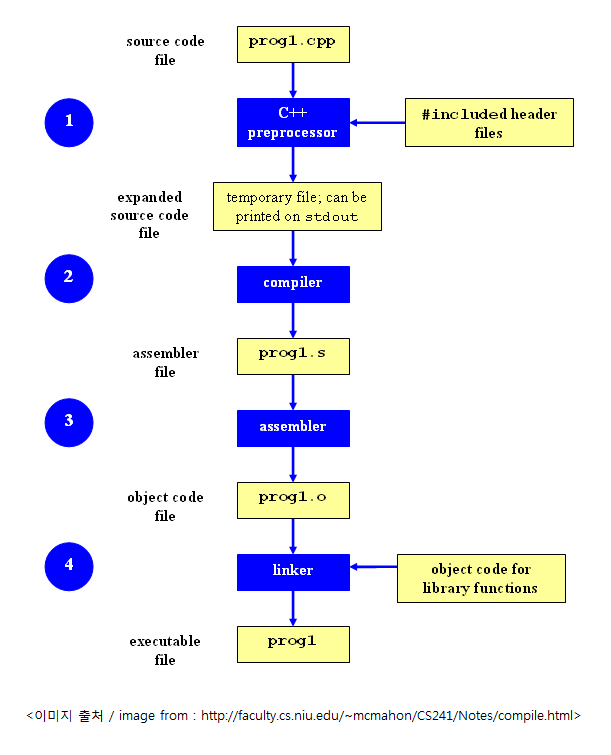
중간중간 소스파일을 컴파일해서 \*.obj 파일로 만들어주는 프로그램을 컴파일러(번역기)라 하고, 목적 파일로 만드는 과정을 '컴파일한다'고 표현합니다. 그리고 중간 목적파일인 obj 파일을 연결해서 최종적으로 하나의 실행파일로 만들어주는 프로그램을 링커(Linker)라고 하고, 실행파일로 만드는 과정을 '링크시킨다(Linking)'라고 말합니다. 이렇게 obj 파일을 링커 프로그램으로 링크시키면 비로소 실행파일이 만들어지는 것입니다. 

다양한 링커 프로그램이 있으며 프로그램의 성격에 맞는 링커를 선택해 사용합니다

컴파일러 프로그램이 여러 종류인 것처럼 링커 프로그램 역시 수 십 종류가 있습니다. 마이크로소프트C에서 사용하는 MS링크, 터보C에서 사용하는 터보링크, 클리퍼 프로그램에서 사용하는 RT링크와 B링크 프로그램 등으로 링커 프로그램은 많은 종류가 있습니다.   
  
이처럼 다양한 링커 프로그램이 있는 이유는 링커마다 성능과 기능이 조금씩 다르기 때문입니다. 초보자에게는 어렵지만 조금 수준이 높아지면 자신에게 맞는 링커 프로그램을 이용하게 됩니다.   
  
과거에는 컴파일을 한 다음에 따로 링킹 과정을 거쳤습니다. 그러나 통합개발환경이 된 지금은 통합개발 프로그램 안에서 컴파일과 링킹 과정을 하나의 과정으로 연속 처리해줍니다. 따라서 별도로 링커 프로그램을 구할 필요도 없으며, 따로 링킹시킬 필요도 없습니다. 물론 라이브러리 파일이라는 것을 만들 때에는 별도로 링커 프로그램을 이용해야 합니다. 

\*\*요약: 컴파일러로 컴파일을 하면 \*.obj 파일이 만들어지는데 이를 연결(link)해서 실행파일인 \*.exe 파일로 만들어주는 프로그램을 링커(linker)라고 합니다.

컴파일 과정



static변수(정적 변수)의 의미

* 블럭 안에서만 유효한 값을 가지지만 자동변수와 같이 없어지지 않고 블럭으로 다시 돌아왔을 때 이전 값을 다시 이용 가능(스택이 아닌 정적데이터 영역에 저장됨)
* 초기화를 생략하면 0으로 자동 초기화
* 정적 데이터영역에 할당
* static은 사용되는 위치에 따라 의미가 달라지니 주의해서 사용할 것.

static class Archer  
{

public:

//정적 객체를 스택에 생성한 후 정적 객체 반환

static Archer getInstance()

{

static Archer inst;

inst++;

printPointerAddr("inst", &inst);

return inst;

}

static Archer\* getDynamicInstance()

{

if (pInst == nullptr)

{

pInst = new Archer();

printPointerAddr("pInst", &pInst);

}

return pInst;

}

//객체의 고유성을 판단하기 위한 data출력 함수

Archer& operator++(int)

{

data++;

std::cout << data << '\n';

return \*this;

}

private:

static Archer\* pInst;

int data{0};

} global;

static Archer global2;

static Archer global3;

static Archer global4;

Archer\* Archer::pInst;

void func()

{

static Archer funcInst;

funcInst++;

printPointerAddr("func", &funcInst);

}

int main()

{

//test 1

static Archer c;

static Archer d;

c++++++++;

d++;

c++;

printPointerAddr("c", &c);

printPointerAddr("d", &d);

//이는 둘 다 1로 나온다.

//static으로 선언할 때마다 객체는 재생성되는 것이 아니다.

//왜냐하면 d를 호출한 다음 c를 호출하면 이전 값이 이어진다.

//둘은 다른 객체이다. 데이터를 공유하지 않는다.

printPointerAddr("Archer::getInstance()", &Archer::getInstance());

Archer::getInstance()++;

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

func(); //1

func(); //값이 2이다.

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

//메인 함수에서의 static Archer와

//func에서의 static Archer은 분명 다른 객체이다.

Archer::getInstance()++;

//static으로 선언된 기존의 객체가 살아있다.

//일반 변수는 프로시저가 끝나면 객체는 사라지지만 이는 그렇진 않다.

(\*Archer::getDynamicInstance())++;

//위와 다른 객체이다. 카운팅을 다시 한다.

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

//printPointerAddr("global", &global);

global++;

global2++;

global3++;

global4++;

//전부 1이다. 모두 서로 다른 객체이다.

return 0;

}

정리

* static은 단 하나만 존재하는 객체는 아니다. 여러 개 존재할 수 있다.
* 데이터 영역에 저장되어 프로그램이 종료되기 전까지 지속되는 것이다. 이 저장된 객체를 사용하려면 선언된 블록 내애서만 유효하다는 것을 기억해야 한다.

###### 결론은 정적 객체의 초기화 순서는 하나의 변역 단위에서만 예측 가능하다. 문제는 아래와 같다.

#### [요소7] 서로 다른 번역 단위에서 정의된 비지역 정적 객체들 사이의 상대적인 초기화 순서는 정해져 있지 않다.

|  |
| --- |
| *//1.cpp*  **class** **darkWorld**{  **public**:  std::size\_t numPlayer() **const**;  };  **extern** darkWorld dw;  *//2.cpp*  Archer::Archer(**const** std::string& name, int rank, bool sex, int& weapon)  : myName(name), myRank(rank), mySex(sex), myWeapon(weapon){  *//dw가 먼저 초기화되리란 보장이 없으므로 심각한 문제를 야기할 수 있습니다.*  otherPlayers = dw.numPlayer();  } |

위와 같은 문제를 [디자인 패턴: 싱글턴]으로 아래와 같이 해결할 수 있다.

|  |
| --- |
| *//1.cpp*  **class** **darkWorld**{  **public**:  std::size\_t numPlayer() **const**;  };  *//자기 자신을 지역 정적 객체로 만들어 반환합니다.*  darkWorld& dw(){  **static** darkWorld dw;  **return** dw;  }  *//2.cpp*  Archer::Archer(**const** std::string& name, int rank, bool sex, int& weapon)  : myName(name), myRank(rank), mySex(sex), myWeapon(weapon){  *//dw()는 지역 정적 객체를 만들어 반환하므로 반드시 먼저 초기화되어 있습니다.*  otherPlayers = dw().numPlayer();  }  **원리는 비지역 정적 객체를 지역 정적 객체로 바꿔준 것이다.** 이로서 초기화 순서가 보장된다. Question. 확실히 초기화는 보장되며 어렵지 않다. 하지만 지역 정적 객체로 바꿔준 것이면 그 함수에서 비정적 객체를 생성하는 것과 무엇이 다른가? 싱글톤에서는 하나의 객체만이 존재하는 정적 객체를 이용한다. 그래서 static으로 선언되어야 하며 위에서도 그렇다. 이는 프로그램 시작과 동시에 만들어져 끝날 때까지 존재하며 여러 개의 static 객체를 생성할 수 없다. 런타임 중에 생성되고 사라지는 비정적 객체와 구별되는 싱글톤에서의 객체는 이와 구별된다. |

#### [요소8] 다중 스레드에서 정적 객체의 문제

다중 스레드에서의 모든 비상수 정적 객체는 심각한 오류를 야기할 잠재력을 지닌다. 위와 같은 참조자 반환 함수는 내부적으로 정적 객체를 쓰기 때문에 여러 스레드가 동시에 접근하여 자원을 엉터리로 만들 수 있다.  
🡪 이에 대한 해결책으로 다중 스레드로 진입하기 전에 미리 참조자 반환 함수를 불러주는 방식을 사용할 수도 있다.

#### [정리] 초기화에 대한 고찰

프로그램은 어떤 자료를 저장하고 이를 ‘어떤 순서’로 연산하느냐를 의미한다. 자료를 저장하기 위해 컴파일러에게 알려준다. 사용할 메모리와 이름이다. 그런데 한 가지 더 알려주어야 할 것은 초기값이다. 처음의 값이 임의의 값이라면 동작도 임의대로 흘러갈 것이 뻔하다. 그래서 위에서 우리가 저장할 모든 자료, 특히 사용자 정의 클래스나 정적 객체에 대해 초기화하는 방법들을 나열했다.  
하지만 내가 정의한 초기화들이 모두 동시에 일어날 수는 없다. 어떠한 흐름에 의해 차례대로 초기화되며, 만약 내가 원하는 순서가 아니라면 골치아프다. 이는 다중 스레드가 갖는 의미와 비슷하다. 이를 해결하기 위해서 [요소7]에서는 함수 안에서 객체를 선언했다. 함수 안에서의 시퀀스는 내가 완벽히 통제할 수 있다는 것이다. 초기화도 일종의 연산이다. ‘어떤 순서’라는 것은 이러한 의미를 갖는다. 그리고 그것을 내가 통제할 수 있도록 설계해야 한다.

# [디자인 패턴: 싱글턴]

개념: 클래스 안에 자기 자신의 정적 객체를 만들어 사용하는 기법입니다.

#### C++에서 싱글톤 구현하기

참고: <http://egloos.zum.com/Agbird/v/4730538>

 작년에 다른 팀에 면접 지원을 나간적이 있습니다. 윈도우 프로그래밍 경력자를 뽑고 있었는데 그 팀에는 윈도우 프로그래밍 경험을 가지신 분들이 없었기 때문이죠. 면접을 위해 윈도우 프로그래밍과 C++ 문법, 그리고 알고리즘 질문을 각각 준비했었는데 그 중 C++ 언어 관련 질문으로 제가 준비한 것은 다음과 같습니다.

C++ 에서 싱글톤 패턴을 구현하는 방법들을 아는데로 나열하고 각각의 장/단점을 말해보세요.

 전 이전 회사에서부터 면접 때 항상 이 질문을 하곤 했습니다. 왜냐하면 싱글톤을 구현하는 방법에는 C++ 에서 필수적으로 알아야 하는 생성/소멸자, 권한, static의 특성 등 기본적인 문법 사항을 고루 담고 있기 때문입니다. 그런데 비교적 해묵은 주제임에도 불구하고 면접을 보신 분 중 한 분도 제대로 대답을 못해 좀 의외였습니다.  따라서 한번쯤 공유차원에서 정리해봐야겠다고 벼르고 있었는데 생각난 김에 지금 정리해 봅니다.

 C++ 에서 싱글톤을 구현하는 방법에는 우선 다음과 같은 방법이 있습니다.

// .h

class Singleton {

  private:

    Singleton() {}  
    Singleton(const Singleton& other);  
    static Singleton inst;

  public:

**static Singleton& getInstance() { return inst; }**

};

// .cpp

**Singleton Singleton::inst;**

 위처럼 생성자를 private으로 하고 static 멤버 변수를 하나 생성해서 그 객체를 반환하도록 하면 외부에서는 해당 전역 객체만을 참조할 수 있습니다. 간단하지요. 클래스 접근 권한과 클래스 내에서의 static 지시 한정자의 역할을 이해하고 있다면 충분히 구현할 수 있는 방법입니다.

**그런데 위 방식은 단순한 반면 몇 가지 단점이 있습니다.** static 클래스 멤버 변수는 static 전역 변수처럼 프로그램 시작 시 main() 함수 호출 이전에 초기화됩니다.  따라서 위 객체는 만약 프로그램의 진행 상황에 따라 필요가 없는 경우에도 무조건 생성되기 때문에 때에 따라서 비효율적입니다.   
 게다가 위와 같은 정적 객체는 **다른 전역 객체의 생성자에서 참조하고 싶은 경우 문제가 발생할 수 있습니다.** 왜냐하면 C++표준에서는 전역 객체들의 생성 순서에 대해서 명확하게 정의하고 있지 않기 때문입니다. 그저 main() 함수가 실행하기 전에만 생성되면 될 뿐입니다. 따라서 어떤 전역 객체의 생성자에서 위 싱글톤 객체를 참조하려고 하는 경우 싱글톤 객체가 미처 생성되기 전인 경우가 발생할 수 있습니다. 결국 객체의 생성 시점을 조절할 필요가 있죠.   
 아마 effective 시리즈 류의 책을 보신 분들이라면 **늦은 초기화**에 대해 들어 보셨을 겁니다. 위의 문제점을 피하기 위해선 늦은 초기화 방법을 사용해 다음과 같이 동적 생성을 하면 됩니다.

// .h

class DynamicSingleton {

  private:  
    DynamicSingleton() {}  
    DynamicSingleton(const DynamicSingleton& other);  
    **~DynamicSingletone()** {}    // 외부에서 싱글톤 객체를 강제 delete 하는 것을 막기 위해 필요함

    static DynamicSingleton\* inst;

  public:

    static DynamicSingleton\* getInstance() {

**if (inst == 0) inst = new DynamicSingleton();**

      return inst;  
    }

};

// .cpp

DynamicSingleton\* DynamicSingleton::inst;

 이렇게 하면 최초 getInstance()를 호출하는 시점에 객체가 생성되므로 상황에 따라(한번도 해당 객체를 사용하지 않으면) 생성이 되지 않기 때문에 자원을 효율적으로 사용할 수 있을뿐더러 물론 다른 전역 객체의 생성자에서 참조하는 것도 가능합니다.   
 여기서 **'동적 생성한 객체는 그럼 언제 해제하나요?'** 라는 질문을 던질 수 있습니다. 그러나 프로그램이 종료되는 순간 동적 객체는 자동으로 해제되기 때문에 굳이 명시적으로 해제할 필요가 없습니다. 메모리 릭 문제는 지속적으로 메모리 할당이 일어나는데 해제는 안 되는 상황에서 발생하는 문제이지 이 객체처럼 한번만 생성되어 프로그램 종료 시까지 유지되는 객체는 문제가 되지 않습니다.   
 **물론 명시적으로 해제해야 하는 경우도 있습니다.** 가령 위 객체가 반드시 프로그램 종료 시 반납해야 하는 외부 시스템 자원을 사용하는 경우가 그렇습니다. 이를 위해서는 atexit() 함수에 해제 함수를 등록하거나 혹은 다른 전역 객체의 소멸자를 이용해야 합니다. 각각의 구현 방법은 아래와 같습니다.  
  
// atexit() 이용 방법  
class DynamicSingleton {  
    ...  
  private:  
    static void destroy() { delete inst; }

  public:

    static DynamicSingleton\* getInstance() {

**if (inst == 0) {  
       inst = new DynamicSingleton();  
       atexit(destroy);  
     }**

      return inst;  
    }

};  
  
// 전역 객체의 소멸자 이용 방법  
// .h  
class \_SingletonDestroyer;  
class DynamicSingleton {  
    ...  
    **friend \_SingletonDestroyer;**};  
  
// .cpp  
static class \_SingletonDestroyer {  
  public:  
    ~\_SingletonDestroyer() {  
      **delete DynamicSingleton::getInstance();**    }  
} destroyer;   
  
보시다시피 좀 귀찮습니다. 따라서 **이런 명시적인 해제 작업을 피하기 위해서는 static 지역 객체를 사용하면 됩니다**. 방법은 아래와 같습니다.

class LocalStaticSingleton {

  public:

    static LocalStaticSingleton& getInstance() {

**static LocalStaticSingleton inst;**

      return inst;

    }

  private:

    LocalStaticSingleton() {}  
    LocalStaticSingleton(const LocalStaticSingleton& other);  
};

[힙으로 정적 객체를 선언한 경우 외부 자원을 사용하는 객체에 대해서 프로그램 종료 시 스스로 소멸되지 않는다. 하지만 지역 객체로 선언한 경우라면 소멸된다. 왜나하면 프로그램 종료 시, 지역 객체는 더 이상 참조할 리가 없지만 동적 객체는 어디서 참조하는지

지역 static 객체는 전역 객체와 달리 해당 함수를 처음 호출하는 시점에 초기화됩니다. 따라서 위 객체를 한번도 사용하지 않으면 생성도 되지 않습니다. 그러면서도 static 객체이기 때문에 프로그램 종료 시까지 객체가 유지되며 종료시에는 자동으로 소멸자가 호출됩니다. 따라서 소멸자에서 자원 해제를 하도록 구현해놓으면 자원 관리도 신경쓸 필요가 없습니다.

 하지만 **위 세번째 구현에도 문제가 하나 있습니다. 만약 저 싱글톤 객체를 다른 전역 객체의 소멸자에서 사용하려고 하면 문제가 발생합니다**. 왜냐하면 C++ 표준에서는 전역 객체들의 생성 순서만 명시하지 않은 것이 아니라 소멸 순서에 대해서도 명시해 놓지 않았기 때문입니다. 따라서 어떤 전역 객체가 소멸자에서 저 싱글톤 객체를 사용하려고 할 때 싱글톤 객체가 먼저 소멸했다면(이것을 참조 무효화 현상이라고 합니다) 문제가 발생합니다.  
  
 이 문제를 해결하기 위해선 다소 고난이도 방법이 필요합니다. 그 중 재밌는 것이 Andrei Alexandrescu가 쓴 [**Modern C++ Design**](http://kangcom.com/common/bookinfo/bookinfo.asp?sku=200303030004) 이라는 책에 나오는 피닉스 싱글톤입니다. 이 싱글톤은 우선 싱글톤 참조 시 해당 객체의 소멸 여부를 파악하고 만약 소멸되었다면 다시 되살립니다. 구현 코드는 아래와 같습니다.  
  
// .h  
class PhoenixSingleton {  
  public:

    static PhoenixSingleton& getInstance() {  
      if (destroyed) {  
        **new(pInst) PhoenixSingleton; // 2)**        atexit(killPhoenix);  
        destroyed = false;  
      } else if (pInst == 0) {  
        create();  
      }

      return \*pInst;

    }

  private:

    PhoenixSingleton() {}  
    PhoenixSingleton(const PhoenixSingleton & other);  
    ~PhoenixSingleton() {  
      **destroyed = true;  // 1)**    }  
  
    static void create() {

**static PhoenixSingleton inst;**

**pInst = &inst;**    }  
  
    static void killPhoenix() {  
      **pInst->~PhoenixSingleton();  // 3)**  
    }  
  
    **static bool destroyed;**    **static PhoenixSingleton\* pInst;**};  
  
// .cpp   
bool PhoenixSingleton::destroyed = false;  
PhoenixSingleton\* PhoenixSingleton::pInst = 0;  
  
 갑자기 굉장히 복잡해졌는데 핵심만 간단히 설명하자면(자세한 내용은 위에 소개한 책을 참조하세요) 정적 객체가 소멸되면 **1) 소멸자에 의해 destroyed 변수가 true가 되면서 소멸 여부를 알 수 있습니다.** 그리고 소멸 후에 getInstance() 함수를 통해 해당 객체를 참조하려 하면 **2) replacement new 를 이용해서 해당 객체의 생성자를 재호출해서 객체를 되살립니다.** 이것이 가능한 이유는 컴파일러는 전역 객체 소멸 시에 해당 메모리를 초기화하지 않기 때문에 해당 메모리를 재 사용해서 객체의 생성자만 다시 호출하면 객체를 재 사용할 수 있기 때문입니다. 그 후 atexit() 함수에 killPhoenix() 함수를 등록해서 **3) 프로그램 종료 시에 PhoenixSingleton 객체의 소멸자를 호출해서 리소스 해제를 합니다.**  
  
 물론 마지막에 소개한 PhoenixSingleton 방법은 상당히 tricky 하며 실제로는 거의 쓸일이 없습니다. 제 경우는 예전에 어떤 윈도우용 프로그램에서 딱 한번 어쩔 수 없이 사용했습니다. **실제 중요한 것은 static 객체의 생성/소멸 시점에 대해 정확히 파악해서 싱글톤 객체를 전역 객체의 생성/소멸자에서 마구잡이로 참조하는 일이 없도록 주의해서 프로그래밍하는 것입니다.**  
  
p.s. 물론 구두 면접에서 이 정도까지 상세한 답을 기대하진 않았습니다...  
p.p.s. 역시나 실전에 별 쓸일은 없지만 난이도 있는 다른 문제를 하나 내보겠습니다. C++에서는 자바의 final 처럼 상속을 막는 키워드가 아쉽게도 없습니다. 그렇다면 C++에서는 클래스의 상속을 막기 위해서 어떤 방법을 사용할 수 있을까요? 힌트는 위의 코드들에 나온 문법 중에 하나를 사용하면 된다는 것입니다.

#### C++에서 상속을 방지하는 방법

아래 코드를 보면 클래스 A의 foo()라는 함수는 private로 선언이 되어 있습니다.

class A

**{**

private**:**

void foo**();**

**};**

class B **:** public A

**{**

public**:**

void bar**()**

**{**

foo**(); *// 'A::foo' : cannot access private member declared in class 'A'***

**}**

**};**

당연한 얘기이지만 본 foo()라는 함수는 class A 내부에서만 사용을 할 수가 있을 뿐, 그 상속 클래스(B)에서도 사용을 하지 못합니다. 원래 private라는 놈이 그런 이유로 만들어 진 거죠.

그런데 constructor와 destructor를 private로 선언하게 되면 조금 다른 역할까지 하게 됩니다.

class A

**{**

private**:**

A**();**

virtual **~**A**();**

**};**

class B **:** public A *// 'A::A' : cannot access private member declared in class 'A'*

*// 'A::~A' : cannot access private member declared in class 'A'*

**{**

**};**

대부분의 OOP language에서는 constructor 및 destructor는 다음과 같아 호출이 되게 됩니다.

**객체가 생성이 될 때에는 상위 클래스(A)에서부터 하위 클래스(B) 순으로 constructor가 불려 져야 한다.**

**객체가 해제가 될 때에는 하위 클래스(B)에서부터 상의 클래스(A) 순으로 destructor가 불려 져야 한다.**

즉 하위 클래스에서는 반드시 상위 클래스의 constructor 및 destructor의 visibility가 반드시 보장되어야 합니다. 이것이 바로 일반 member function이랑 다른 점입니다. 상위 클래스에서 constructor 및 destructor를 private으로 선언을 하게 되면 하위 클래스에서는 상위 클래스의 constructor와 destructor의 접근권을 가지지 못하기 때문에 하위 클래스를 만들 수 없다는 결론을 도출할 수가 있습니다.

여기에서 재미있는 예외 상황을 보도록 하겠습니다. destructor를 빼고 constructor만 private으로 선언을 합니다.

class A

**{**

private**:**

A**();**

**};**

static A**\*** a **=** 0**;**

class B **:** public A

**{**

public**:**

B**() :** A**(\***a**) {}**

**};**

A 클래스에서는 constructor를 private으로 선언해 주었음에도 불구하고 B 클래스는 A 클래스를 상속받을 수가 있게 되어 버립니다. 자세히 보면 B 클래스의 constructor에서는 A 클래스의 copy constructor를 호출하고 있는 것을 알 수가 있습니다. 즉 B 클래스의 constrcutor에서는 A 클래스의 nullary constructor는 호출을 못하지만, A 클래스의 copy constructor를 호출함으로 해서 constructor 호출을 대신하고 있음을 알 수 있습니다.

**클래스 설계시에 nullary constructor(copy constructor 포함) 및 destructor는 명시를 하지 않으면 기본적으로 public입니다.** non-nullary constructor는 명시하지 않으면 존재하지 않기 때문에 패스~~~.

결론적으로 상속을 막는 방법은 다음과 같이 결론지을 수 있습니다.

**1. destructor를 private로 선언을 한다.**

**혹은**

**2. 모든 constructor(non-nullary, nullary, copy)를 private로 선언을 한다.**

1번 방법이 더 간단하겠죠? ^^

***기본 제공 타입의 객체는 직접 손으로 초기화해야 합니다. 경우에 따라 저절로 되기도 하고 안 되기도 하기 때문입니다.***

***생성자에서는, 대입문을 생성자 본문 내부에 넣는 방법으로 멤버를 초기화하지 말고 멤버 초기화 리스트를 사용합니다.***

***초기화 순서는 리스트 순서가 아닌 멤버가 선언된 순서에 좌우됩니다.***

***여러 번역 단위에 있는 비지역 정적 객체들의 초기화 순서 문제는 비지역 정적 객체를 지역 정적 객체로 바꿔서 피할 수 있습니다.***

[항목 5] 자동으로 호출된 C++함수

*복사 생성자, 복사 대입 연산자, 소멸자*에 대하여 주의해야 할 점

# C++(컴파일러)이 저절로 선언하는 함수들

#### 기본 생성자, 소멸자

컴파일러에게 “배후의 코드”를 깔 수 있는 자리를 마련한다.

#### 복사 생성자, 복사 대입 연산자

원본 객체를 복사하여 사본 객체 쪽으로 집어넣는다.

|  |
| --- |
| **class** **Archer**{  **public**:  *//우리가 선언하지 않으면 밑의 함수는 저절로 생성됩니다.*  *//public, inline으로 생성됩니다.*  *//기본 생성자, 복사 생성자*  Archer();  Archer(**const** Archer& a);  *//소멸자*  ~Archer();  *//복사 대입 연산자*  Archer& **operator**=(**const** Archer& a);  }; |

# 위의 자동 생성 함수들에 관하여 문제는 없는가?

#### 이미 있다면 번복하지 않는다.

안에 어떠한 형태로든 생성자(대입 연산자)가 선언되어 있다면 컴파일러는 생성자(대입 연산자)를 따로 만들지 않는다. 아래 모든 요소들은 이를 바탕으로 생각할 수 있다.

#### 소멸자도 신경써야 한다. - 가상 소멸자

**상속한 기본 클래스의 소멸자가 비가상 소멸자면 역시 비가상 소멸자로 만들어진다. -[항목7]**

#### 대입 연산자의 동작

이는 동작하려면 적법해야 하고 이치에 닿아야만 한다. 어떠한 경우에 그렇지 않은가?  
대입은 객체를 복사하여 새로운 객체에 집어넣는다.

###### 만약 멤버 변수에 참조자, 상수가 있을 경우

원래 자신이 참조하고 있는 객체에서 바뀔 수 없고 상수도 바뀔 수 없으므로 컴파일 거부가 된다.   
🡪 위와 같이 참조자나 상수를 멤버 변수로 갖는 경우에는 대입 연산자를 직접 정의해주어야 한다.

###### 복사 대입 연산자가 private선언한 기본 클래스로부터 파생된다면, 이 클래스는 암시적 복사 연산자를 가질 수 없다.

기본 클래스가 private선언되어 있다면 암시적으로 복사 대입 연산자를 가질 수 없다. – [항목12]

왜냐하면 private상속은 ~을 써서 구현된이라는 의미이므로 그의 함수를 가진다라는 의미이기 때문이다. – [항목39]

# [참고] 연산자 오버로드

|  |
| --- |
| Assignment operator The assignment operator (operator=) has special properties: see [copy assignment](http://en.cppreference.com/w/cpp/language/copy_assignment) and [move assignment](http://en.cppreference.com/w/cpp/language/move_assignment) for details. To summarize, the canonical "unified assignment operator" implementation is  T& T::operator=(T arg) // copy/move constructor is called to construct arg  {  swap(arg); // resources are exchanged between \*this and arg  return \*this;  } // destructor of arg is called to release the resources formerly held by \*this  When there are resources that can be reused in assignment, for example, if the class owns a heap-allocated array, then copy-assignment between arrays of the same size can avoid allocation and deallocation:  T& operator=(const T& other) // copy assignment  {  if (this != &other) { // self-assignment check expected  if (*/\* storage cannot be reused (e.g. different sizes) \*/*)  {  delete[] mArray; // destroy storage in this  */\* reset size to zero and mArray to null, in case allocation throws \*/*  mArray = new int[*/\*size\*/*]; // create storage in this  }  */\* copy data from other's storage to this storage \*/*  }  return \*this;  }  T& operator=(T&& other) // move assignment  {  [assert](http://en.cppreference.com/w/cpp/error/assert)(this != &other); // self-assignment check not required  delete[] mArray; // delete this storage  mArray = [std::exchange](http://en.cppreference.com/w/cpp/utility/exchange)(other.mArray, nullptr); // leave moved-from in valid state  return \*this;  }  대입 연산자는 자기 자신을 반환해야 하며, 자기 대입을 주의해야 한다. 뒤 항목에서 언급된다. |
| Increment and decrement When the postfix increment and decrement appear in an expression, the corresponding user-defined function (operator++ or operator--) is called with an integer argument 0. Typically, it is implemented as T operator++(int), where the argument is ignored. The postfix increment and decrement operator is usually implemented in terms of the prefix version:  struct X  {  X& operator++()  {  // actual increment takes place here  return \*this;  }  X operator++(int)  {  X tmp(\*this); // copy  operator++(); // pre-increment  return tmp; // return old value  }  };  Although canonical form of pre-increment/pre-decrement returns a reference, as with any operator overload, the return type is user-defined; for example the overloads of these operators for [std::atomic](http://en.cppreference.com/w/cpp/atomic/atomic) return by value. Binary arithmetic operators Binary operators are typically implemented as non-members to maintain symmetry (for example, when adding a complex number and an integer, if operator+ is a member function of the complex type, then only complex+integer would compile, and not integer+complex). Since for every binary arithmetic operator there exists a corresponding compound assignment operator, canonical forms of binary operators are implemented in terms of their compound assignments:  class X  {  public:  X& operator+=(const X& rhs) // compound assignment (does not need to be a member,  { // but often is, to modify the private members)  */\* addition of rhs to \*this takes place here \*/*  return \*this; // return the result by reference  }    // friends defined inside class body are inline and are hidden from non-ADL lookup  friend X operator+(X lhs, // passing lhs by value helps optimize chained a+b+c  const X& rhs) // otherwise, both parameters may be const references  {  lhs += rhs; // reuse compound assignment  return lhs; // return the result by value (uses move constructor)  }  }; |
| Relational operators Standard algorithms such as [std::sort](http://en.cppreference.com/w/cpp/algorithm/sort) and containers such as [std::set](http://en.cppreference.com/w/cpp/container/set) expect operator< to be defined, by default, for the user-provided types, and expect it to implement strict weak ordering (thus satisfying the [Compare](http://en.cppreference.com/w/cpp/concept/Compare)concept). An idiomatic way to implement strict weak ordering for a structure is to use lexicographical comparison provided by [std::tie](http://en.cppreference.com/w/cpp/utility/tuple/tie):  struct Record  {  [std::string](http://en.cppreference.com/w/cpp/string/basic_string) name;  unsigned int floor;  double weight;  friend bool operator<(const Record& l, const Record& r)  {  return [std::tie](http://en.cppreference.com/w/cpp/utility/tuple/tie)(l.name, l.floor, l.weight)  < [std::tie](http://en.cppreference.com/w/cpp/utility/tuple/tie)(r.name, r.floor, r.weight); // keep the same order  }  };  Typically, once operator< is provided, the other relational operators are implemented in terms of operator<.  inline bool operator< (const X& lhs, const X& rhs){ */\* do actual comparison \*/* }  inline bool operator> (const X& lhs, const X& rhs){ return rhs < lhs; }  inline bool operator<=(const X& lhs, const X& rhs){ return !(lhs > rhs); }  inline bool operator>=(const X& lhs, const X& rhs){ return !(lhs < rhs); }  Likewise, the inequality operator is typically implemented in terms of operator==:  inline bool operator==(const X& lhs, const X& rhs){ */\* do actual comparison \*/* }  inline bool operator!=(const X& lhs, const X& rhs){ return !(lhs == rhs); } Array subscript operator User-defined classes that provide array-like access that allows both reading and writing typically define two overloads for operator[]: const and non-const variants:  struct T  {  value\_t& operator[]([std::size\_t](http://en.cppreference.com/w/cpp/types/size_t) idx) { return mVector[idx]; }  const value\_t& operator[]([std::size\_t](http://en.cppreference.com/w/cpp/types/size_t) idx) const { return mVector[idx]; }  };  If the value type is known to be a built-in type, the const variant should return by value.  Where direct access to the elements of the container is not wanted or not possible or distinguishing between lvalue c[i] = v; and rvalue v = c[i]; usage, operator[] may return a proxy. see for example[std::bitset::operator[]](http://en.cppreference.com/w/cpp/utility/bitset/operator_at).  To provide multidimensional array access semantics, e.g. to implement a 3D array access a[i][j][k] = x;, operator[] has to return a reference to a 2D plane, which has to have its own operator[] which returns a reference to a 1D row, which has to have operator[] which returns a reference to the element. To avoid this complexity, some libraries opt for overloading operator() instead, so that 3D access expressions have the Fortran-like syntax a(i, j, k) = x;  마지막의 다차원 배열의 오버로딩은 포인터를 반환하면 각각의 원소에 접근할 수 있다. |

#### 클래스 내의 동적할당

추가적으로 클래스 내에서 동적 할당을 한다면, 복사 연산에 주의해야 한다. 얕은 복사를 하면 클래스 내의 자료를 공유할 수 있고, 다른 한 쪽에서 해제할 수 있기 때문이다.

Person::Person(**const** Person& p) *// 복사생성자 타입*

{

name = **new** char[strlen(p.name)+1];

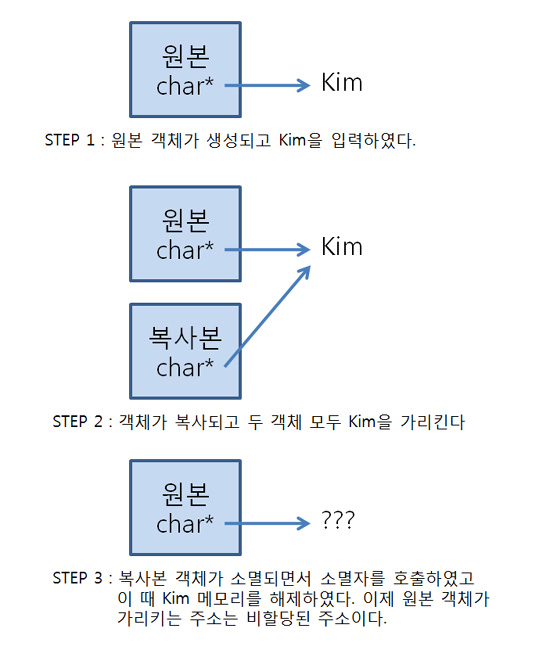
strcpy(name, p.name); *// 원본 내용을 그대로 복사한다.*

phone = **new** char[strlen(p.phone)+1];

strcpy(phone, p.phone);

age = p.age;

}



***컴파일러는 경우에 따라 클래스에 대해 기본 생성자, 복사 생성자, 복사 대입 연산자, 소멸자를 암시적으로 만들어 놓을 수 있습니다.***

[항목6] 컴파일러의 함수 금지하기

# 함수 금지가 무슨 말이냐?

[항목5]에서 컴파일러는 사용자의 의지와 관계없이 자동으로 생성하는 함수가 있다. 만약 이 함수가 필요 없다면? 아니, 심지어 금지해야만 한다면?

#### 어떤 함수가 금지되어야 할까?

기본 생성자나 소멸자는 금지될 리가 없다. 반면 복사는 금지해야 하는 경우가 있다. 그럴 경우, 복사 생성자와 복사 대입 연산자가 금지되어야 하겠다.

# 함수 호출을 어떻게 금지한단 말인가?

위에서 언급한대로 복사 생성자와 복사 대입 연산자를 금지시킬 것이다. 다음을 따라 해보자. 원하는 함수에 대해 아래와 같이 시행하면 된다.

#### [해결1] private멤버로 선언한다.

일단 비공개(private) 접근성을 가지므로 외부에서 호출할 수 없다. 하지만 friend함수에서 호출할 수 있다는 점은 여전히 온전치 않다.

#### [해결2] 정의를 안 한다.

위에서처럼 접근성을 차단해도 어디에선가(friend) 호출될 수 있다면, **[기법]** **정의를 안 하는 것이다.** 만약 정의되지 않은 함수를 호출한다면 링크 시점에서 에러를 맛볼 것이다.

#### [해결3] private상속을 이용한다.

링크시점에서의 에러를 컴파일시점의 에러로 옮긴다. private선언된 복사 생성자와 복사 대입 연산자를 별도의 기본 클래스에 넣어 이것을 파생시킨다.   
위에서 제시한 세가지 해결책을 따라 적용한 코드는 아래에서 볼 수 있다.

|  |
| --- |
| **class** **Uncopyable**{  **protected**: *//파생된 객체에 대하여 생성자와 소멸자를 허용*  Uncopyable(){};  ~Uncopyable(){};  **private**: *//복사 생성, 대입은 금지*  Uncopyable(**const** Uncopyable&);  Uncopyable& **operator**=(**const** Uncopyable&);  };  **class** **Archer** : **private** Uncopyable{  *//복사 생성과 대입을 선언할 수 없다.*  };  위에서 복사를 시도하려고 한다면, Uncopyable(**const** Uncopyable&);에 접근할 수 없다는 오류가 뜬다. 이는 복사를 시도하면 컴파일러는 현재 클래스에서 복사 생성자(복사 대입 연산자도 마찬가지)를 만든다. 만든 이 함수는 대응되는 기본 클래스의 함수에 먼저 접근하는데 여기에 private선언으로 접근할 수 없으므로 오류를 맛보게 된다. |

###### 위의 class: Uncopyable에 대하여

몇 가지 알려줄 사실이 있다. 이후에 제시할 항목과 연관이 있다.  
1. 상속은 public일 필요가 없다. private선언을 하자. 각각의 의미가 있는데 ***public의 의미는 ‘is-a’이며 private은 ‘~의 일부를 사용한다’이다. 의미가 서로 다르다.*** – [항목32, 39]  
2. 소멸자는 가상 소멸자가 아니어도 된다. – [항목7]  
3. 데이터 멤버가 없으므로 **[컴파일러 수준의 최적화 기법]** **공백 기본 클래스 최적화(EBO: Empty Base class Optimization)**를 적용할 수 있다. – [항목39] 하지만 다중상속을 주의해야 한다. – [항목40]  
4. boost라이브러리에 같은 역할인 *‘noncopyable’*이 있다.

***컴파일러에서 자동으로 제공하는 기능을 허용치 않으려면, 대응되는 멤버 함수를 private로 선언한 후에 구현은 하지 않은 채로 두십시오. Uncopyable과 비슷한 기본 클래스를 쓰는 것도 한 방법입니다.***

[항목7] 다형성과 가상 소멸자 사용

다형성을 가진 클래스여, 잘 듣거라.

# 다형성을 가진 클래스?

객체지향 언어(OOP)에서 다음과 같이 정의된다.

###### **다형성**: 서로 다른 객체가 동일한 메시지에 대하여 서로 다른 방법으로 응답하는 성질

기본 클래스는 하나의 ***가상 함수***를 갖게 된다. 그리고 그것을 상속하는 파생 클래스는 그 가상 함수를 ***오버라이딩***하여 다른 기능을 갖는 함수, 즉 같은 함수지만 다른 방법으로 응답하는 기능이 되겠다. – 자세한 내용은 [항목41] 참고

객체지향에서의 다형성

객체 중심으로 진행되는 객체지향에서의 다형성은 같은 객체가 다양한 성질을 갖게 한다. 모든 객체마다 각각의 정보를 가질 필요가 없다는 것이다. **어떠한 형태(class)에서 어떤 객체(instance)가 만들어지면 그 객체의 정보는 실행 도중에 결정될 수 있게 한다.** 이로써 다양한 객체를 간단하게 만들어준다.

###### **가상 소멸자**: 소멸할 때 호출되는 함수. 가상(virtual)은 다형성을 표현한다.

즉, 다형성을 가진 소멸자이다. *생성자는 기본 생성자 🡪 파생 생성자를 불러오며(아래 코드 참고) 소멸자는 파생 소멸자 🡪 기본 소멸자를 기대해야 한다.* ***하지만 지금의 객체가 기본 클래스 포인터를 통한다면(펙토리 함수) 기본 소멸자만 사라진다(부분 소멸).***

Archer\* a = **new** BladeMaster("YTK");

위의 코드처럼 **기본 클래스에 파생 클래스를 담는다**라는 개념이다. 이러한 경우가 위에 해당한다.

**이를 막기 위하여 가상 소멸자(virtual~)를 선언해야 한다.** 이는 소멸자를 호출하기 전에 파생된 관계를 따지기 때문이다.

###### **상속**: 기존의 클래스를 토대로 새로운 클래스를 만드는 것

상속은 클래스간 관계를 자명하며 3가지 유형으로 사용된다.

가상 함수와 가상 소멸자에 대해서 아래의 코드로 이해해보자.

|  |
| --- |
| #include *<cstdio>*  #include *<iostream>*  #include *<string>*  #include *<functional>*  #include *<vector>*  **using** **namespace** std;  bool onBattle = 1;  **class** **Archer**{  **public**:  *//함수 포인터를 받을 변수 선언(콜백 함수)*  function<void(int&)> callBackFunc;  *//함수 포인터를 변수에 저장(콜백 함수 등록)*  void registFunc(function<void(int&)> func){  *//완벽한 함수 전달을 위하여 move함수가 준비되어 있다.*  callBackFunc = move(func);  }  *//저장한 콜백 함수 실행*  void executeCallBackFunc(){  callBackFunc(d);  }  Archer(**const** string \_name) : name(\_name), hp(100), d(10){  cout << name << " is coming..." << endl;  }  *//가상 소멸자를 선언한다.*  ***//여기에 가상 소멸자를 선언하지 않으면***  ***//기본 클래스 포인터의 객체는 파생 클래스의 소멸자는 거치지 않는다.***  **virtual** ~Archer(){  cout << name << " is dead!" << endl;  onBattle = 0;  }  *//가상 함수로 선언된 기본 클래스 함수*  **virtual** void attack(Archer& op){  cout << name << " is attaking..." << endl;  op.hit(d);  }  **virtual** void heal(**const** int value){  cout << name << "'s hp is up!" << endl;  hp += value;  }  **virtual** void hit(int d){  cout << name << " is hurt!" << endl;  hp -= d;  **if** (hp <= 0) **delete** **this**;  }  **private**:  **const** string name;  int hp;  int d;  };  **class** **BladeMaster** : **public** Archer{  **public**:  *//파생 클래스 객체 생성 시, 기본 클래스의 기본 생성자를 먼저 호출하는데*  *//기본 생성자가 없으면 컴파일 오류*  *//아래는 기본 클래스의 인자있는 생성자를 직접 불러온다.*  BladeMaster(**const** string \_name): Archer(\_name){  cout << "my sword is cool." << endl;  }  *//여기는 virtual선언을 안해도 되네?*  ~BladeMaster(){  cout << "my sword is fade away..." << endl;  }  **private**:  *//가상 함수 -> 호출 시 파생 클래스의 함수를 불러와야 한다.*  **virtual** void attack(Archer& op){  Archer::attack(op);  *//만약 위의 함수에서 op객체가 죽었다면 런타임 에러가 생긴다.*  *//죽은 함수에 NULL객체를 넣는 방법도 있다. 아래 주소 참고*  // <http://devsw.tistory.com/88>  **if** (onBattle) skill\_headCut(op);  }  void skill\_headCut(Archer& op){  op.hit(1000);  }  };  **class** **DarkKnight**: **public** Archer{  **public**:  DarkKnight(**const** string \_name) : Archer(\_name){  cout << "it's darkness." << endl;  }  ~DarkKnight(){  cout << "i'll be back." << endl;  }  **private**:  **virtual** void attack(Archer& op){  Archer::attack(op);  skill\_houndShild();  }  void skill\_houndShild(){  Archer::heal(50);  cout << "i'll never die." << endl;  }  };  **class** **MasterPlan**{  **public**:  MasterPlan(Archer\* master){  master->registFunc(bind(&MasterPlan::masterPlanEffect, **this**, placeholders::\_1));  }  void masterPlanEffect(int& d){  cout << "I'm having masterPlan!" << endl;  d += 500;  }  };  int main(){  vector<Archer\*> fighter;  Archer\* a = **new** BladeMaster("YTK");  Archer\* b = **new** DarkKnight("TOZ");  fighter.push\_back(a), fighter.push\_back(b);  *//Archer a가 무기 m을 획득했다.*  MasterPlan\* m = **new** MasterPlan(a);  *//콜백 함수로 a가 m의 masterPlanEffect함수를 호출할 수 있다.*  a->executeCallBackFunc();  cout << "battle start!" << endl;  cout << "=============" << endl;  char input;  int time = 0;  **while** (cin >> input){  *//my turn*  **if** ((++time % 2)){  **if** (input == 'a'){  *//기본 클래스를 가리키지만 가상 함수의 선언으로*  *//실제 객체인 파생 클래스의 함수를 불러온다.*  a->attack(\*b);  }  }  *//your turn*  **else** b->attack(\*a);  **if** (!onBattle) **break**;  }  cout << "battle end!" << endl;  cout << "=============" << endl;  **delete** m;  } |

# 가상 소멸자를 사용하자!

위와 같이 ‘기본 클래스’에 가상 소멸자를 선언하면 됩니다.

#### 무조건 가상 소멸자만 선언하면 문제가 없겠군!

그렇지 않다. 가상 함수를 구현하려면 C++에서는 별도의 자료구조(vtbl)를 생성한다.

###### 가상 함수가 1개라도 있는 클래스에 대하여 가상 함수 테이블(vtbl)을 가리키는 가상 함수 테이블 포인터(vptr)가 있어야 한다. – [항목34] 참조

**이는 객체의 크기를 커지게 하며 다른 언어와의 호환성도 없어지게 한다. 그러므로 가상 소멸자를 선언하는 것은 가상 함수가 하나라도 있는 경우에만 하자.**

#### STL 컨테이너 타입(vector, list, set, unordered\_map 등)은 가상 소멸자가 없다.

위의 타입들을 기본 클래스로 하는 행동은 자원을 세게 한다. 그러므로 결코 상속해서는 안 되며 이처럼 다형성을 갖지 못하도록 하려면 가상 소멸자를 선언하지 않는다.

#### 기본 클래스로는 쓰이지만 다형성은 갖지 않도록 설계된 클래스도 있다.

기본 클래스의 인터페이스를 통한 파생 클래스의 조작이 안 된다는 의미다. Uncopyable, input\_iterator\_tag 등이 있다.

#### 순수 가상 소멸자를 사용하자.

순수 가상 함수가 들어있는 클래스는 객체를 만들 수 없는 추상 클래스로 만든다. 하지만 추상 클래스는 만들고 싶은데 넣고 싶은 함수가 없다면? 순수 가상 소멸자를 쓰면 된다. 하지만 **여기에는 반드시 정의가 들어가야 한다는 것**을 기억하자.

|  |
| --- |
| **class** **Item**{  **public**:  **virtual** ~Item() = 0;  };  *//아래 정의가 없으면 링크가 안 됩니다.*  Item::~Item() {} |

###### 순수 가상 함수, 순수 가상 소멸자

**순수 가상 함수?**  
  
virtual void func() = 0 과 같이 선언하는 순수 가상 함수의 의미는 '구현을 하지 않는다'는 뜻이 아니라, 진짜 뜻은 다음과 같다.

이 클래스는 추상클래스이고,  
이 클래스를 상속한 모든 구체 클래스는 이 함수를 구현해야 한다.

보통 순수 가상 함수는 구현하지 않는 것이 보통이지만, 순수 가상 소멸자는 예외이다.   
  
**순수 가상 소멸자?**순수 가상 함수로 만들만한 멤버 함수가 하나도 없는 아주 드문 경우가 있다. 이러한 경우에는 보통 소멸자를 순수 가상함수로 선언해서 해결한다.   
  
주의할 점은 파생 클래스의 소멸자가 호출될 때 결국 이 순수 가상 소멸자도 호출되기 때문에 반드시 구현해야 한다.

|  |
| --- |
| class AbstractAnimal {  protected:          AbstractAnimal & operator=(const AbstractAnimal & rhs);  public:          virtual ~AbstractAnimal() = 0 {          }  };    class Animal : public AbstractAnimal {  public:          Animal & operator=(const Animal & rhs);  };    class Lizard : public AbstractAnimal {  public:          Lizard & operator(const Lizard & rhs);  };    class Chicken : public AbstractAnimal {  public:          Chicken & operator(const Chicken & rhs);  }; |

출처: <http://codemuri.tistory.com/433> [Code Muri]

# 그 외 알아야 할 용어

###### 콜백 함수: ‘다른 곳’에서 ‘이 곳’의 함수를 전달하여 호출해주는 함수

위의 코드에서는 Class::Archer가 Class::MasterPlan을 장착하여 MasterPlan의 함수를 불러올 수 있게 한다.

###### 펙토리 함수: 새로 생성된 파생 클래스에 대한 기본 클래스 포인터를 반환하는 함수

간단하게 객체를 생성하는 함수다. 포인터는 기본 클래스로 융통성을 가진다.

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

std::shared\_ptr<Archer> Barrack<T>::createObject(int level){

**return** std::shared\_ptr<Archer>(**new** T(level));

}

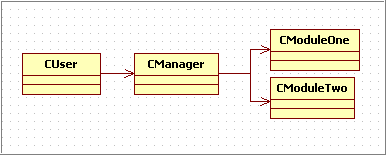
이를 이용하여 펙토리 함수만 묶어서 객체로 관리하는 패턴을 펙토리 메서드 패턴이라고 한다. 예제는 아래에서 확인한다.

# [참고] 전방 선언에 대해서

참고: <http://greenfrog7.egloos.com/1363680>

전방 선언은

**불필요한 헤더 파일이 복잡하게 포함되는 것을 방지하며, Compile 속도를 향상 시켜줍니다.**



현재 상황: CUser 클래스는 CManager 클래스만 알면 된다. 이하 필요없는 코드들까지 컴파일할 필요는 없다.

#### 전방 선언을 하지 않고 모든 헤더파일을 포함한 코드

#pragma once  
  
#include "ModuleOne.h"  
#include "ModuleTwo.h"

class CManager  
{  
public:  
 CManager(void);  
 ~CManager(void);  
};  
  
아래는 CManager클래스의 헤더파일을 포함하고 있는 CUser클래스의 cpp 내용입니다.  
  
#include ".\user.h"  
#include "Manager.h"

CUser::CUser(void)  
{  
}

CUser::~CUser(void)  
{  
}

모두 포함한 상태에서 CModuleTwo 클래스를 수정했을 경우   
CManager 클래스만 함께 컴파일 되면 되는데 이를 직접적으로 사용하고 있지 않음에도 불구하고 CManager클래스의 헤더에 CModuleTwo클래스의 헤더가 포함되어 있기 때문에 이는 **곧 컴파일 속도를 저해하는 원인**이 됩니다.

#### 전방선언으로 고쳐주면 어떻게 될까요?

아래는 CManager클래스의 헤더파일 내용입니다.  
  
#pragma once

class CModuleOne;  
class CModuleTwo;

class CManager  
{  
public:  
 CManager(void);  
 ~CManager(void);  
};  
  
아래는 CManager클래스의 헤더파일을 포함하고 있는 CUser클래스의 cpp 내용입니다.  
  
#include ".\user.h"  
#include "Manager.h"

CUser::CUser(void)  
{  
}

CUser::~CUser(void)  
{  
}

이는 **컴파일 속도의 향상**으로 이어집니다.

그러므로 컴파일 의존도를 최소화해야 합니다. 자세한 내용을 [항목31]에서 봅시다.

# **[디자인 패턴: 펙토리 메소드]**

펙토리 메서드 패턴에 대해 이해할 수 있는 코드는 아래와 같다.

|  |
| --- |
| **class** **application**  {  **public**:  *//순수 가상 함수: 이 클래스는 객체를 생성할 수 없다.*  **virtual** void new\_doc(char \* file\_name) = 0;  };  **class** **hwp\_application** : **public** application  {  **public**:  void new\_doc(char \* file\_name)  {  *//같은 document의 성질을 가지는데*  *//객체마다 각각 구현하는 것은 비효율적이다.*  hwp\_document \* doc = **new** hwp\_document;  doc->open(file\_name);  *// do something*  }  };  **class** **word\_application** : **public** application  {  **public**:  void new\_doc(char \* file\_name)  {  word\_document \* doc = **new** word\_document;  doc->open(file\_name);  *// do something*  }  };  **class** **document**  {  **public**:  **virtual** int open(char \* file\_name) = 0;  };  **class** **hwp\_document** : **public** document  {  **public**:  int open(char \* file\_name)  {  printf("open hwp\_document(%s)**\n**", file\_name);  }  };  **class** **word\_document** : **public** document  {  **public**:  int open(char \* file\_name)  {  printf("open word\_document(%s)**\n**", file\_name);  }  };  **class** **application**  {  **public**:  void new\_doc(char \* file\_name)  {  *// create\_doc 을 호출하고 document 라는 대표 문서객체를 받는다.*  *// 즉, 이 class 에서는 문서의 형식을 신경 쓰면 안 된다.*  ***// 기본 클래스를 포인터로 실제 파생 클래스를 생성한 함수가 보인다.***  ***// 이것이 펙토리 함수이다.***  document \* doc = create\_doc();  doc->open(file\_name);  }  *// 실제로 생성되는 문서 객체는 각 응용에 맞는 것을 생성해야 하므로*  *// 하위 class 로 구현을 넘긴다.*  **virtual** document \* create\_doc() = 0;  };  **class** **hwp\_application** : **public** application  {  **public**:  document \* create\_doc()  {  **return** **new** hwp\_document;  }  };  **class** **word\_application** : **public** application  {  **public**:  document \* create\_doc()  {  **return** **new** word\_document;  }  };  ***//템플릿을 사용함으로써 기본 클래스가 같은 두 형태의 클래스를***  ***//하나의 함수로 요약할 수 있다.***  ***//위의 객체는 필요 없는 것이다.***  **template** <**class** **doc\_type**>  **class** **concrete\_application** : **public** application  {  **public**:  document \* create\_doc()  {  **return** **new** doc\_type;  }  };  int main()  {  *//hwp\_application hwp;*  *//word\_application word;*  *//위의 선언보다 아래 템플릿 선언이 훨씬 간결하다.*  concrete\_application<hwp\_document> hwp;  concrete\_application<word\_document> word;  hwp.new\_doc((char \*)"aa.txt");  word.new\_doc((char \*)"bb.txt");  }  펙토리 메서드 기본 틀  왼쪽에는 문서 자체의 클래스이며, 오른쪽은 문서를 생성하는 프로그램 클래스이다. **모든 문서 생성은 프로그램이 담당한다.**  한글과 엑셀 프로그램이 있다고 할 때, 각각의 프로그램은 각각의 파일들을 생성해야 할 것이다. 그 구체적인 생성 부분들은 각각 하위의 클래스 내에 문서생성() 함수를 통해서 각각의 문서 생성이 가능하고, 인터페이스에서는 생성() 함수만 제공하여 객체 생성의 유연성을 확장 시킨 구조이다. 펙토리 메서드 적용 코드 설명 위의 코드는 이러한 형식을 적용시킨 것이다. 각각의 문서 생성 함수를 만들 필요도 없이 템플릿을 적용하여 하나의 클래스에서 처리할 수 있다. **concrete\_application**이 **doc\_type** 타입 문서 생성을 담당한다. 여러 개의 객체 생성 함수가 있을 필요가 없다. 템플릿을 적용했기 때문이다. |

***다형성을 가진 기본 클래스에서는 반드시 가상 소멸자를 선언해야 합니다. 즉, 어떤 클래스가 가상 함수를 하나라도 갖고 있으면, 이 클래스의 소멸자도 가상 소멸자이어야 합니다.***

***기본 클래스로 설계되지 않았거나 다형성을 갖도록 설계되지 않은 클래스에는 가상 소멸자를 선언하지 말아야 합니다.***

[항목8] 소멸자에서 예외는 안 된다.

예외에 관한 자세한 내용은 [항목29] 참조

# 왜 소멸자인가?

물론 예외가 좋다고 하는 곳은 없다. 하지만 소멸자안에서 생기는 예외는 위험성이 더욱 크다. 소멸자가 죽으면 C++은 프로그램을 종료하지 않고 미정의 동작을 보인다. 소멸자가 재대로 처리되지 않으면 관리하는 자원에 대해서 제대로 처리가 안 됨을 뜻하며, 객체가 사라진 이후이므로 뒷처리도 어렵다.

class Archer

{

public:

Archer() {}

//소멸자에서 예외가 발생하면 프로그램이 죽어버린다.

~Archer()

{

//throw std::exception();

//그래서 소멸자에서 예외가 발생할 수 있는 상황이면

//예외를 넘길 수 없고 아래처럼 내부에서 잡아야 한다.

try

{

throw std::exception();

}

catch (std::exception& e)

{

std::cout << "inner Archer.\n";

std::cout << e.what() << std::endl;

}

}

};

bool what()

{

try

{

//Archer group[10];

Archer a;

}

catch (std::exception& e)

{

std::cout << e.what() << std::endl;

return false;

}

return true;

}

int main()

{

if (what()) std::cout << "success.\n";

else std::cout << "fail.\n";

return 0;

}

위의 예제에서 보면 배열은 물론이고 한 개의 객체에서도 예외가 발생하면 프로그램이 바로 죽는 것으로 나온다. 어쨌든 소멸자 내부에서의 소멸자는 문제가 된다는 것을 알 수 있다.

# 소멸자 내부에서의 예외

DB를 닫아주는 함수(close)가 있다고 하자. 이에 대한 코드는 아래와 같다.

|  |
| --- |
| **class** **DBConnection**{  **public**:  **static** DBConnection create();  void close(); *//close를 호출해야 객체를 소멸한다.*  };  **class** **DBConn**{  **public**:    DBConn(DBConnection \_db): db(\_db){  }  ~DBConn(){  db.close();  }  **private**:  bool closed;  DBConnection db;  };  int main(){  *//위에서 소멸자에 close함수를 넣어준 덕에*  *//생성만 하면 소멸은 저절로! 너무 간편해*  *//close에서 예외만 안 나면...*  DBConn dbc(DBConnection::create());  } |

# 소멸자 예외를 피하려면?

#### 프로그램을 바로 끝낸다.

abort()함수로 프로그램을 끝내버린다. 미정의 동작은 안 볼 수 있다.

#### 예외를 삼켜버린다.

예외 삼키기가 빛을 보려면 발생한 예외를 무시한 뒤라도 프로그램이 신뢰성 있게 실행을 지속할 수 있어야 한다.

|  |
| --- |
| **class** **DBConn**{  **public**:  ~DBConn(){  *//사용자의 몰지각함에 close호출을 까먹는 경우에 대해*  *//소멸시 자동으로 객체를 닫아줍니다.*  **try**{  db.close();  }  **catch** (exception e){  *//여기서 프로그램을 종료하거나*  *//abort();*  *//예외를 삼킬 수 있습니다.*  }  }  **private**:  DBConnection db;  }; |

#### [가장 좋은 해결책] 예외를 다른 함수가 담당한다.

사용자에게 에러를 처리할 수 있는 기회를 주는 것이다. 다음과 같다.

|  |
| --- |
| **class** **DBConnection**{  **public**:  **static** DBConnection create();  void close(); *//close를 호출해야 객체를 소멸한다.*  };  **class** **DBConn**{  **public**:  ***//DBConn에서 DBConnection의 객체를 닫아주는 별도의 다른 함수***  void close(){  *//사용자의 몰지각함에 close호출을 까먹는 경우에 대해*  *//소멸시 자동으로 객체를 닫아줍니다.*  db.close();  closed = true;  }  ~DBConn(){  **if** (!closed)  **try**{  db.close();  }  **catch** (exception e){  *//여기서 프로그램을 종료하거나*  *//abort();*  *//예외를 삼킬 수 있습니다.*  }  }  **private**:  bool closed;  DBConnection db;  };  **중요한 점은 예외는 소멸자가 아닌 다른 곳에서 발생해야 한다는 것.** |

***소멸자에서 예외가 빠져나가면 안 됩니다. 만약 소멸자 안에서 호출된 함수가 예외를 던질 가능성이 있으면 어떤 예외든지 소멸자에서 모두 받아낸 후에 삼켜 버리든지 프로그램을 끝내든지 해야 합니다.***

***어떤 클래스의 연산이 진행되다가 던진 예외에 대하여 사용자가 반응해야 할 필요가 있다면, 해당 연산을 제공하는 함수는 반드시 보통의 함수(소멸자가 아닌)이어야 합니다.***

[항목9] 생성, 소멸에 가상함수 금지

객체 생성, 소멸 과정 중에는 절대로 가상 함수를 호출하면 안 된다!

# 왜 가상 함수를 쓰면 안 되는데?

상속 받고 있는 객체의 생성 순서를 살펴보자.

#### 생성 순서: 기본 클래스 생성자(기본 클래스 초기화) 🡪 파생 클래스 생성자(파생 클래스 초기화) 🡪 여러 가지 행위 🡪 파생 클래스 소멸자(파생 클래스 제거) 🡪 기본 클래스 소멸자(기본 클래스 제거)

보아라, 기본 클래스 생성자에서는 파생 클래스를 초기화하기 전이다. **즉, 파생 클래스 객체는 기본 클래스 생성자가 실행되는 동안에 객체 타입은 기본 클래스다.** 마찬가지로 소멸자에서도 파생 클래스 소멸자가 호출된 뒤에는 더 이상 파생 클래스 객체가 아니라 기본 클래스 객체 타입이다.   
이제 예제를 보자.

|  |
| --- |
| #include *<iostream>*  #include *<string>*  **using** **namespace** std;  **class** **Archer**{  **public**:  **explicit** Archer(**const** string& characterInfo);  *//순수 가상 함수 -> 이 클래스는 객체를 만들 수 없다.*  **virtual** void attack() **const** = 0;  *//비가상 함수입니다.*  void attack(**const** string& characterInfo) **const**;  **private**:  string myInfo;  };  Archer::Archer(**const** string& characterInfo){  myInfo = characterInfo;  cout << myInfo << endl;  *//기본 생성자 호출 중에는 파생 클래스가 아니므로*  *//가상 함수를 불러올 수 없다!*  *//attack();*  ***//그래서 비가상 함수를 호출합니다.***  attack(characterInfo);  }  **class** **BladeMaster** : **public** Archer{  **public**:  BladeMaster(int value);  **virtual** void attack() **const**;  **private**:  *//기본 클래스의 생성자 쪽으로 넘길 값을 생성한다.*  *//static이므로 미초기화된 멤버를 건드릴 염려가 없다.*  **static** string createCharacterInfo(int value);  };  BladeMaster::BladeMaster(int value)  *//파생 클래스에서 기본 클래스를 불러서 비가상 함수를 호출한다.*  ***//생성과 동시에 파생 클래스의 무언가가 호출되어야 한다면***  ***//파생 클래스에서 파생 클래스의 인자(create…(value))를***  ***//기본 클래스의 비가상 함수로 넘겨서 호출하자!***  : Archer(createCharacterInfo(value)){  cout << "create BladeMaster." << endl;  }  void BladeMaster::attack() **const**{  cout << "fast blade." << endl;  }  string BladeMaster::createCharacterInfo(int value){  **return** "BladeMaster value: " + to\_string(value);  }  int main(){  BladeMaster bm(1);  } |

# 파생 클래스의 함수가 필요하지만 가상 함수는 안 쓰는 방법?

위의 예제를 참고하기 바란다.

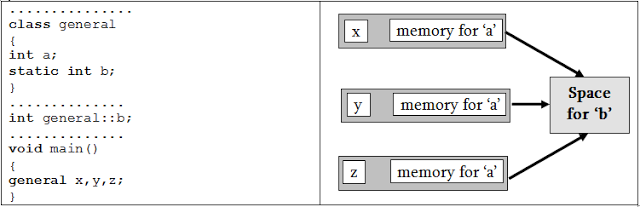
#### 가상 함수를 비가상 함수로 바꾼다.

기본 클래스 생성자에서 비가상 함수를 호출하게 만든다. 그리고 파생 클래스 생성자에서 기본 클래스의 생성자를 호출하는데 인자를 비가상 함수로 넘긴다. 그러면 파생 클래스의 함수처럼 실행할 수 있다.

###### static함수로 복잡한 초기화 인자를 생성할 수 있는 점을 기억하자.

미초기화된 데이터 멤버는 정의되지 않은 상태에 있다. 그러므로 static함수로 선언한다.

###### Question. 왜 static이면 초기화에서 위험이 없을까?

**static멤버변수:** 객체가 1억개여도 클래스 당 1개뿐이다. 그렇다면 이것은 어떤 객체의 소유인가? 어떤 객체의 것도 아니며 프로그램 초기에 생성된다.  
 static 멤버변수가 private으로 선언되면 객체들만 접근할 수 있는데 객체에 속해있다고 착각할 수 있다. 그렇게 생각해서는 안 된다. 객체들은 static 멤버변수에 접근할 수 있는 권한을 가지고 있는 것이지 static 멤버변수를 가지고 있는 것은 아니다. 다시 돌아가서 static 멤버변수를 public으로 선언하면 어떻게 되겠는가? 그러면 클래스의 namespace 혹은 객체의 이름, 참조자, 포인터 등으로 전역 변수처럼 사용할 수 있다.

**static멤버함수:** static멤버변수와 특징이 유사하다.   
static으로 선언하면 멤버변수든 멤버함수든 객체를 생성하지 않아도 먼저 메모리에 할당된다. 만약 위의 클래스의 static 멤버함수인 Adder() 함수를 객체를 생성하지 않은 상태에서 호출하면 어떻게 될까? static 멤버변수인 num2는 메모리에 존재하지만 num1은 아직 메모리에 존재하지 않는 상태이다. 그런데 Adder() 함수를 호출하면 메모리에 존재하지 않는 num1을 참조하게 되므로 에러를 발생시킨다. 그래서 컴파일 에러가 발생한다. 그러므로 static 멤버함수는 static 멤버변수, static 멤버함수만 호출하도록 제한한다.

***생성자 혹은 소멸자 안에서 가상 함수를 호출하지 마세요. 가상 함수라고 해도, 지금 실행 중인 생성자나 소멸자에 해당되는 클래스의 파생 클래스 쪽으로는 내려가지 않아요.***

[항목10] 대입 연산자 만들기

대입 연산자는 \*this의 참조자를 반환하게 해야 한다.

# 대입연산자를 구현해보자.

#### 우측 연관 연산

우리가 사용하는 대입연산자는 이렇게 해석된다.

해석: 15가 z에 대입되고 그 대입 연산 결과(갱신된 z)가 y에 대입된 후에, y에 대입된 결과가 x에 대입된 것이다.

이렇게 보면 우변의 항을 복사하여 좌변에 넣고 있다. 연산자는 무엇을 반환해야 대입에 대입을 할까? 객체 자체, 참조자를 반환해야지. 그래서 다음과 같이 구현한다.

|  |
| --- |
| **class** **Stone**{  **public**:  *//해당 클래스 참조자를 반환한다. 인자 역시 참조자이다.*  Stone& **operator**=(**const** Stone& rs){  *//대입한 후에, 자기 자신의 참조자를 반환한다.*  **return** \***this**;  }  };  ***이렇게 해야 다시 참조자를 인자로 넘기고 하는 방식이 가능해지지!*** |

#### 모든 형태의 대입 연산자에서 지켜져야 합니다.

+=, -= 등에도 마찬가지로 적용한다. 그리고 인자가 정수인 경우에도 동일하게 적용합니다.

***대입 연산자는 \*this의 참조자를 반환하도록 하세요.***

[항목11] 대입 연산에서의 오류

operator=에서의 자기 대입에 대한 처리가 필요하다는데 무엇일까?

# 자기 대입

자기 대입이란, 어떤 객체가 자기 자신에 대해 대입 연산자를 적용하는 것을 말합니다.

#### 그런 대입을 왜 하지?

사용자가 직접 자기 대입이 필요해서 하는 경우는 없지만 여러 곳에서 하나의 객체를 참조할 때 생긴다. 이것을 **중복참조**라고 한다. 이 **중복참조**는 같은 타입의 여러 객체를 참조자나 포인터로 물어놓고 동작하는 코드에서 발생할 수 있다.

중복 참조가 생기면 어떻게 되는지 아래 코드에서 확인한다.

|  |
| --- |
| **class** **Weapon**{  **private**:  int type;  };  **class** **Archer**{  **public**:  Archer& **operator**=(**const** Archer& ra);  **private**:  Weapon \*w;  };  *//대입연산자를 실행합니다. 자기 대입이면 ra와 this는 같습니다.*  Archer& Archer::**operator** = (**const** Archer& ra){  *//현재 객체의 w를 삭제합니다. ra의 w도 같이 삭제되겠군요.*  **delete** w;  *//삭제된 \*ra.w을 생성하는군요.*  w = **new** Weapon(\*ra.w);  **return** \***this**;  }  자기 대입을 하면 하나의 객체(ra=this)를 함수 자체에서와 인자에서 둘 다 쓰고 있다. 이것이 **중복참조**이며 삭제된 w를 생성하는 심각한 문제를 일으킨다. |

#### 중복 참조를 해결하는 방법

자기 자신인지 체크한다. 이것을 일치성 검사(identity test)라 한다. 하지만 약간의 문제가 있다. 코드 비대화, 처리 흐름에 분기를 일으켜서 CPU명령어 선행인출, 실행 시간 증가, 캐시, 파이프파이닝 등의 효과가 떨어질 수 있다.

|  |
| --- |
| Archer& Archer::**operator** = (**const** Archer& ra){  *//자기 대입 검사 - 자기 자신이면 무시합니다.*  **if** (&ra == **this**) **return** \***this**;  **delete** w;  w = **new** Weapon(\*ra.w);  **return** \***this**;  } |

정말로 참고만 한다. 비트맵 파일을 읽고 써야 할 때, 다음을 시행한다.

|  |
| --- |
| #include *<stdio.h>*  #include *<windows.h>*  #include *<iostream>*  #include *<iomanip>*  #include *<fstream>*  **using** **namespace** std;  int main(void){  BITMAPFILEHEADER bf;  BITMAPINFOHEADER bi;  ifstream fin;  fin.open("a.bmp", ios\_base::in | ios\_base::binary); *//a.bmp파일을 바이트로 읽어 들인다..*  fin.read((char\*)&bf, **sizeof**(BITMAPFILEHEADER)); *// 사이즈..*  FILE \*fp = fopen("a.bmp", "rb");  fread(&bf, **sizeof**(unsigned char), **sizeof**(BITMAPFILEHEADER), fp);*// 사이중 헤더에 사이즈 14바이트..*  fread(&bi, **sizeof**(unsigned char), **sizeof**(BITMAPINFOHEADER), fp);*// 나머지 정보 40바이트 저장..*  unsigned char\* data = (unsigned char\*)malloc(bi.biSizeImage);  fread(data, **sizeof**(unsigned char), bi.biSizeImage, fp);    int r, g, b, mean;  **for**(int x=0; x<bi.biHeight; x++){  **for**(int y=0; y<bi.biWidth\*3; y=y+3){  *//printf("%d\t", x \*(bi.biWidth\*3) + y);*  *//printf("%d\t", x \*(bi.biWidth\*3) + y+1);*  *//printf("%d\t", x \*(bi.biWidth\*3) + y+2);*  b = data[x \*(bi.biWidth\*3) + y];*// \*(data+x \*(bi.biWidth\*3) + y);*  g = data[x \*(bi.biWidth\*3) + y+1];  r = data[x \*(bi.biWidth\*3) + y+2];    mean = (r+g+b)/3;  data[x \*(bi.biWidth\*3) + y] = mean;  data[x \*(bi.biWidth\*3) + y+1] = mean;  data[x \*(bi.biWidth\*3) + y+2] = mean;  }  }  printf("%d**\n**", bi.biWidth\*3\*bi.biHeight);  printf("%d**\n**", bi.biSizeImage);    *//평균*  int mean;  **for**(int i=0; i<bi.biSizeImage; i = i+3){  *//data[i] = 255-data[i];*  mean = (data[i] + data[i+1] + data[i+2])/3;  data[i] = mean;  data[i+1] = mean;  data[i+2] = mean;  }  fclose(fp);    *// b.bmp파일에 다시 기록..*  fp = fopen("b.bmp", "wb");  fwrite(&bf, **sizeof**(unsigned char), **sizeof**(BITMAPFILEHEADER), fp);  fwrite(&bi, **sizeof**(unsigned char), **sizeof**(BITMAPINFOHEADER), fp);  fwrite(data, **sizeof**(unsigned char), bi.biSizeImage, fp);  fclose(fp);    free(data);  **return** 0;  } |

# 다른 방법은 없을까?

약간 문제가 있는 *일치성 검사* 말고 더 좋은 방법은 없을까? 그것은 **예외처리**다. 예외에 안전하게 구현하면 예외도 없으며 동시에 자기 대입도 처리할 수 있다.

#### 예외에 안전하게 구현하기

여기서 다루는 내용은 예외처리가 아니다. 바로 아래가 핵심이다.

*“문장의 순서를 세심하게 바꾸는 것만으로 예외에 안전한 코드가 만들어진다.”*

###### 방법1: 포인터가 가리키는 객체를 복사한 뒤 삭제

|  |
| --- |
| Archer& Archer::**operator** = (**const** Archer& ra){  *//자기 대입 검사 - 자기 자신이면 무시합니다.*  **if** (&ra == **this**) **return** \***this**;  *//원래의 w를 어딘가에 기억해둔다. \*pOrig은 원래 w이다.*  Weapon \*pOrig = w;  *//w은 \*ra.w의 사본을 가리키고 있다. 바뀌었다.*  w = **new** Weapon(\*ra.w);  *//원래 w을 삭제합니다.*  **delete** pOrig;  **return** \***this**;  }  먼저 지우면 안 된다. 이와 같은 방식은 객체 대입을 성공시킨 뒤 삭제를 하고 있다. 적어도 w가 이상한 곳을 가리킬 일은 없겠군요. |

###### 방법2: swap() 함수 사용

[기법: 복사 후 맞바꾸기(copy and swap)]이라고 불린다. 예외 안정성과 자기 대입 안정성을 동시에 지닌다. 어떻게 구현하는지 살펴보자.

|  |
| --- |
| **class** **Archer**{  **public**:  Archer& **operator**=(**const** Archer& ra);  Archer& **operator** = (Archer ra);  *//함수를 만들어주어야 한다.*  void swap(**const** Archer& ra);  **private**:  Weapon \*w;  };  *//참조 전달*  Archer& Archer::**operator** = (**const** Archer& ra){  *//사본을 만들어야 하네?*  Archer temp(ra);  swap(temp);  **return** \***this**;  }  *//값에 의한 전달*  Archer& Archer::**operator** = (Archer ra){  *//값에 의한 전달은 사본이 생긴다는 점을 이용하여 코드를 간결하게*  swap(ra);  **return** \***this**;  }  값에 의한 전달에서 ra는 분명 사본이다. 이는 저절로 옮겨 담을 접시를 가져다준 셈이다. 따로 사본을 만들 필요가 없다. |

***operator=을 구현할 때, 어떤 객체가 그 자신에 대입되는 경우를 제대로 처리하도록 만듭시다. 원본 객체가 복사대상 객체의 주소를 비교해도 되고, 문장 순서를 적절히 조정할 수도 있으며, 복사 후 맞바꾸기 기법을 써도 됩니다.***

***두 개 이상의 객체에 대해 동작하는 함수가 있다면, 이 함수에 넘겨지는 객체들이 사실 같은 객체인 경우에 정확하게 동작하는지 확인하세요.***

[항목12] 제대로 된 복사 구현

복사함수는 객체의 모든 부분을 빠짐없이 복사해야 한다.

복사 생성자와 복사 대입 연산자를 **복사함수**라고 부른다.

# 복사함수 생성시 문제가 될 수 있는 부분

물론 컴파일러는 저절로 복사함수를 만들어 주지만 그것이 마음에 안 들면 우리가 직접 만들어준다고 이미 배웠다. 우리는 문제없이 만드는데 성공할 수 있을까?

#### [오류1] 데이터 멤버 추가

데이터 멤버를 추가하면 생성자를 비롯하여 모든 operator=과 연관된 함수를 갱신해야 합니다. 번거롭겠죠?

###### [오류1]의 해결책

어쩔 수 없어요. 추가한 데이터 멤버에 대해 복사하도록 다시 만들어줘야 하죠.

#### [오류2-치명적인] 클래스 상속

이름만 들어서는 와 닿지 않기에 코드로 설명해보자.

|  |
| --- |
| **class** **Archer**{  **public**:  void levelUp();  void printBaseInfo();  Archer& **operator** = (**const** Archer& a);  **private**:  string name;  int level = 0;  };  void Archer::levelUp(){  level++;  }  void Archer::printBaseInfo(){  cout << "level: " << level << endl;  }  Archer& Archer::**operator** = (**const** Archer& a){  name = a.name;  level = a.level;  **return** \***this**;  }  **class** **BladeMaster** : **public** Archer{  **public**:  void setForce(int value);  void printInfo();  BladeMaster& **operator**=(**const** BladeMaster& bm);  **private**:  int force = 0;    };  void BladeMaster::setForce(int value){  force += value;  }  void BladeMaster::printInfo(){  printBaseInfo();  cout << "force: " << force << endl;  }  BladeMaster& BladeMaster::**operator** = (**const** BladeMaster& bm){  force = bm.force;  **return** \***this**;  }  int main(){  BladeMaster \*bm1 = **new** BladeMaster();  BladeMaster \*bm2 = **new** BladeMaster();  bm1->printInfo();  cout << "==========" << endl;  bm2->printInfo();  cout << "==========" << endl;  *//bm1의 레벨은 3입니다.*  bm1->levelUp();  bm1->levelUp();  bm1->levelUp();  *//* *bm1=bm2와 \*bm2 = \*bm1은 다르다는 것을 잊지 마세요!*  \*bm2 = \*bm1;  bm1->printInfo();  cout << "==========" << endl;  bm2->printInfo();  cout << "==========" << endl;  *//복사함에도 불구하고 bm1과 bm2의 레벨이 다릅니다.*  }  기본 클래스 부분은 Archer부분은 복사되지 않는다. 이를 어떻게 해결할 것인가? |

###### [오류2]의 해결책

파생 클래스 복사함수에서 기본 클래스 복사함수를 호출해주어야 합니다.

|  |
| --- |
| *//복사 생성자: 기본 클래스 복사 생성자 함께 호출*  BladeMaster::BladeMaster(**const** BladeMaster& bm) : Archer(bm){  }  *//operator= 복사 연산자: 기본 클래스의 해당 함수 함께 호출*  BladeMaster& BladeMaster::**operator** = (**const** BladeMaster& bm){  Archer::**operator**=(bm);  force = bm.force;  **return** \***this**;  }  이제 기본 클래스까지 완전한 복사를 이루었다. |

# 위의 문제를 고려하여 복사함수를 만들면 완전하군요.

#### 혹시 다른 방법이 있나요? – 중복을 피해보자.

한 쪽에서 다른 쪽(복사 생성자, 복사 대입 연산자)을 호출하고 싶다고? 말이 안 된다. 하지 마세요. 생성자와 대입 연산자는 역할이 다르다. 새로 만들어 내는 것(복사 생성자)과 이미 존재하는 객체에 값을 주는 행위(복사 대입 연산자)는 너무나도 다르다.

다만 양쪽에서 겹쳐지는 부분을 별도의 멤버 함수로 분리하여 그 함수를 호출하게 할 수는 있다(private멤버 함수).

***객체 복사 함수는 주어진 객체의 모든 데이터 멤버 및 모든 기본 클래스 부분을 빠뜨리지 말고 복사해야 합니다.***

***클래스의 복사 함수 2개를 구현할 때, 한 쪽에서 다른 쪽을 구현하려는 시도는 절대로 하지 마세요. 그 대신 공통된 동작을 제3의 함수에다 분리해 놓고 양쪽에서 이것을 호출하게 만들어서 해결합시다.***

[항목13] 자원관리는 객체 스스로

자원관리는 객체 스스로가 하게 합시다!

# 객체 스스로가 자원 관리하는 방법

힙에서 동적으로 할당된 객체가 소멸자 안에서 delete를 호출하는 것이다. 이러한 것을 가능하게 해주는 포인터(Smart Pointer)들이 몇 가지 있다.

#### auto\_ptr(C++11에서 deprecated)

|  |
| --- |
| 사용 방법 |
| auto\_ptr<class type> ptr\_name(가리키는 대상);  class Archer;  class ArchMage : Archer{  public:  Archer\* goodMorningArcher();  void func();  };  *//파생된 클래스에서 기본 클래스의 객체가 필요하여,*  *//파생 클래스의 기본 클래스의 객체를 만들어내는 함수가 필요하군요.*  *//이것은 팩토리 함수입니다.*  Archer\* ArchMage::goodMorningArcher(){  return new Archer();  }  void ArchMage::func(){  *//스마트포인터를 사용합니다.*  *//Archer\* pA = goodMorningArcher();와 같다.*  *//팩토리 함수에서 만든 객체를 바로 auto\_ptr로 넘기는 모습이 보입니다.*  auto\_ptr<Archer> pA(goodMorningArcher());  *//delete를 pA 소멸자 내부에서 스스로 진행한다.*  *//delete pA;*  } |
| 특징 |
| 기본적인 스마트 포인터 특징을 가진다. 1. 자원을 획득하여 바로 자원 관리 객체(smart pointer)에게 넘깁니다. 이것을 자원 획득 즉 초기화(RAII)라고 합니다. 2. 소멸자가 자원을 삭제합니다. auto\_ptr만의 특징 1. 어떤 객체를 가리키는 auto\_ptr의 개수가 2개 이상이면 안 됩니다. 기존의 ptr은 null로 만들어버립니다(객체를 가리키는 유일한 포인터가 됩니다.). 🡪 소유권의 문제가 생김: 복사 생성자와 할당 연산자 구현이 멤버 데이터에 대한 깊은 복사 대신 얕은 복사(제대로 된 복사가 아닌 전달만 하는 – 단순히 포인터 대입만 하는 것)를 하도록 되어 있기 때문이다.    2. 이것을 컨테이너에 넣으면 null을 가리키는 auto\_ptr로 가득 찹니다.  std::auto\_ptr<int> temp = new int;  *//COAP: container of auto\_ptr*  std::vector< std::auto\_ptr<int> > COAP;  COAP.push\_back(temp); //이 컨테이너에 포인터를 삽입할 수 없다.  왜냐하면 표준 위원회에서 COAP를 막고자 컨테이너의 삽입 멤버 함수의 인자로 상수 참조(const T& arg)를 받도록 정의하는 트릭을 썼다고 한다. 복사 생성자에 상수 참조가 넘어가게 되는데, auto\_ptr은 소유권 이전을 하기 때문에 복사 생성자에 상수 참조를 받지 못한다. 그래서 복사 생성자를 찾을 수 없다는 컴파일 에러를 내게 된다.  즉, 컨테이너 안에서 auto\_ptr을 못쓰게 막아놓았다.  auto\_ptr의 선언  // 20.4.5 Class template auto\_ptr  namespace std {  template <class Y> struct auto\_ptr\_ref {};  template<class X> class auto\_ptr {  public:  typedef X element\_type;  // 20.4.5.1 construct/copy/destroy:  explicit auto\_ptr(X\* p =0) throw();  auto\_ptr(auto\_ptr&) throw();  template<class Y> auto\_ptr(auto\_ptr<Y>&) throw();  auto\_ptr& operator=(auto\_ptr&) throw();  template<class Y> auto\_ptr& operator=(auto\_ptr<Y>&) throw();  auto\_ptr& operator=(auto\_ptr\_ref<X> r) throw();  ~auto\_ptr() throw();  // 20.4.5.2 members:  X& operator\*() const throw();  X\* operator->() const throw();  X\* get() const throw();  X\* release() throw();  void reset(X\* p =0) throw();  // 20.4.5.3 conversions:  auto\_ptr(auto\_ptr\_ref<X>) throw();  template<class Y> operator auto\_ptr\_ref<Y>() throw();  template<class Y> operator auto\_ptr<Y>() throw();  };  } |
| 어떻게 사용하면 좋을까? |
| 호출하는 쪽에서 자원을 쓰고 난 뒤, 책임져야 할 때 사용한다.  역시 C++11에서 거의 사용되지 않는다. unique\_ptr이 이를 대체한다. |

#### unique\_ptr

|  |
| --- |
| 사용 방법 |
| unique\_ptr<class type> ptr\_name(가리키는 대상);  #include *<iostream>*  #include *<memory>*  #include *<string>*  int main(void)  {  *// 정수 5를 가리키는 포인터 타입 p1을 만듬*  std::unique\_ptr<int> p1(new int(5));  *// 복사 허용안함, 주석 해제하면 컴파일 에러 발생한다.*  *//std::unique\_ptr<int> p2 = p1;*  *// move() 함수를 이용하며 p1의 소유권을 p3로 이전*  std::unique\_ptr<int> p3 = std::move(p1);  *// get() 함수를 이용하면 포인터의 주소 값을 얻을 수 있음.*  std::cout << p1.get() << std::endl;  std::cout << p3.get() << std::endl; *// 주소 반환*  *// dereferencing를 통해 포인터가 가리키는 값을 가져옴*  std::cout << \*p3 << std::endl;  *// C++11의 auto 키워드를 통해 값 할당*  auto a = \*p3;  std::cout << a << std::endl;  *// auto 키워드는 값을 할당 받을떄도 사용 가능*  auto& a2 = p3;  std::cout << \*a2 << std::endl;  *// p1이 가리키는 값은 이미 소유권이 p3로 이전되었기 때문에*  *// p1을 역참조 하려고 하면 런타임에러가 발생*  *//auto b = \*p1;*  *//std::cout << b << std::endl;*  *// reset 함수를 통해 p3가 가졌던 메모리 영역 초기화 가능.*  p3.reset();  p1.reset();  return 0;  }  이동 연산자 std::move()에 대해서는 밑의 설명 참고 |
| 특징 |
| 기본적인 스마트 포인터 특징 이전과 동일 unique\_ptr만의 특징 unique\_ptr역시 auto\_ptr처럼 하나의 주소에 대해 유일한 소유권을 갖는다. 하지만 다른 곳으로 주소를 옮길 때 복사가 아닌 다른 개념이 적용된다.  C++ 11 표준에서는 Copy Semantics에 추가로 새로운 개념인 Move Semantics가 등장하였다. Move Semantics는 Copy Semantics 처럼 두 객체 사이에 복사를 수행하는 대신 객체의 데이터 필드를 이전 시키는 역할을 수행한다. [정리] 복사 연산자 vs. 이동 연산자 어떤 객체를 생성해야 할 때는 일반적으로 클래스의 생성자를 호출해 만드는데, 때로는 이미 만들어진 객체의 복사는 할당을 통해서(이동 생성자) 새로운 객체를 만들기도 한다. 다른 예로 STL의 백터나 리스트와 같은 컨테이너의 경우 크기가 두 배씩 늘어나게 된다. 이때 메모리 내부에서는 대량의 복사가 발생한다. 그런데 STL 컨테이너에서의 문제는 복사 후 원본과 사본 모두를 사용하는 게 아니라 원본 파괴하고 사본만 사용하므로 불필요한 복사 동작을 하여 오버헤드를 내게 되는데, 이로 인해 쓸데없는 객체를 생성하거나 사용하지 않는 원본 객체를 파괴한다는 점 등 이런 동작은 C++ 성능저하의 주범이라는 문제점이 있었다.  그래서 이를 위해서 복사보다는 이동이 낫겠다고 판단하여 Move Semantics를 도입하게 되었다. 이를 위해 이동 생성자 라는 개념을 도입하였다.  굉장히 마법 같은 일이다. 어떻게 복사하지 않으면서 원본을 훼손하지 않고 값을 옮길 수 있을까? - emc++[항목0] 참고  아래는 우리가 알고 있는 복사 생성자의 모범 답안이다.  class MemoryBlock  {  public:  *// Simple constructor that initializes the resource.*  explicit MemoryBlock(size\_t length)  : \_length(length)  , \_data(new int[length])  {  std::cout << "In MemoryBlock(size\_t). length = "  << \_length << "." << std::endl;  }  *// Destructor.*  ~MemoryBlock()  {  std::cout << "In ~MemoryBlock(). length = "  << \_length << ".";  if (\_data != nullptr)  {  std::cout << " Deleting resource.";  *// Delete the resource.*  delete[] \_data;  }  std::cout << std::endl;  }  *// Copy constructor.*  MemoryBlock(const MemoryBlock& other)  : \_length(other.\_length)  , \_data(new int[other.\_length])  {  std::cout << "In MemoryBlock(const MemoryBlock&). length = "  << other.\_length << ". Copying resource." << std::endl;  std::copy(other.\_data, other.\_data + \_length, \_data);  }  *// Copy assignment operator.*  MemoryBlock& operator=(const MemoryBlock& other)  {  std::cout << "In operator=(const MemoryBlock&). length = "  << other.\_length << ". Copying resource." << std::endl;  if (this != &other)  {  *// Free the existing resource.*  delete[] \_data;  \_length = other.\_length;  \_data = new int[\_length];  std::copy(other.\_data, other.\_data + \_length, \_data);  }  return \*this;  }  *// Retrieves the length of the data resource.*  size\_t Length() const  {  return \_length;  }  private:  size\_t \_length; *// The length of the resource.*  int\* \_data; *// The resource.*  };  이번에는 이동 연산자를 만들어봅시다.  *// Move constructor.*  MemoryBlock(MemoryBlock&& other)  : \_data(nullptr)  , \_length(0)  {  std::cout << "In MemoryBlock(MemoryBlock&&). length = "  << other.\_length << ". Moving resource." << std::endl;  *// Copy the data pointer and its length from the*  *// source object.*  \_data = other.\_data;  \_length = other.\_length;  *// Release the data pointer from the source object so that*  *// the destructor does not free the memory multiple times.*  other.\_data = nullptr;  other.\_length = 0;  }  *// Move assignment operator.*  MemoryBlock& operator=(MemoryBlock&& other)  {  std::cout << "In operator=(MemoryBlock&&). length = "  << other.\_length << "." << std::endl;  if (this != &other)  {  *// Free the existing resource.*  delete[] \_data;  *// Copy the data pointer and its length from the*  *// source object.*  \_data = other.\_data;  \_length = other.\_length;  *// Release the data pointer from the source object so that*  *// the destructor does not free the memory multiple times.*  other.\_data = nullptr;  other.\_length = 0;  }  return \*this;  }  <https://msdn.microsoft.com/ko-kr/library/dd293665.aspx>  Move Semantics 라는 개념을 스마트 포인터에도 적용하려니 auto\_ptr의 내부 구현이 이동 시맨틱을 지원할 수 있도록 업그레이드 하기엔 기술적 제약이 걸려있고 이미 사용되던 기존 auto\_ptr에 대한 호환성을 해칠 수도 없었기 때문에 unique\_ptr이라는 이름의 새로운 단일 포인터 타입을 구현하게 되었다.  unique\_ptr내부에는 복사 생성자와 할당 연산자가 아예 구현되어 있지 않는다. unique\_ptr은 객체가 복사가 원천 봉쇄되어 있어 std::move를 사용해서 이동만 가능하다. |
| 어떻게 사용하면 좋을까? |
| 하나의 객체가 2개 이상의 포인터를 가리킬 수 없거나 해선 안될 때 |

#### shared\_ptr

|  |
| --- |
| 사용 방법 |
| shared\_ptr<class type> ptr\_name(가리키는 대상);  int main(void)  {  shared\_ptr<Archer> pA1(new Archer());  *//operator\*이나 get함수로 현재 포인터가 참조하는 객체를 얻을 수 있다.*  *//shared\_ptr을 포인터로서 쓰고 싶으면(인자 전달 등의) \*p나get() 사용*  *//그러므로 \*p는 \*get()과 일치합니다.*  Archer& a = \*pA1;  *//use\_count함수는 현재 해당 포인터가 참조하는 참조 카운트를 보여줍니다.*  cout << pA1.use\_count() << endl;  shared\_ptr<Archer> pA2(pA1);  cout << pA1.use\_count() << endl;  *//결과: 1 2*  *//reset으로 포인터를 비우거나 새로운 포인터를 가리킬 수 있습니다.*  pA2.reset();  return 0;  } shared\_ptr이 가리키는 형을 변환할 수 있습니다. 구현과 사용예시를 확인합니다.  template<class \_Ty1, class \_Ty2>  shared\_ptr<\_Ty1> static\_pointer\_cast(const shared\_ptr<\_Ty2>& \_Other)  {  *// return shared\_ptr object holding static\_cast<\_Ty1 \*>(\_Other.get())*  return (shared\_ptr<\_Ty1>(\_Other, \_Static\_tag()));  }    template<class \_Ty1, class \_Ty2>  shared\_ptr<\_Ty1> const\_pointer\_cast(const shared\_ptr<\_Ty2>& \_Other)  {  *// return shared\_ptr object holding const\_cast<\_Ty1 \*>(\_Other.get())*  return (shared\_ptr<\_Ty1>(\_Other, \_Const\_tag()));  }    template<class \_Ty1, class \_Ty2>  shared\_ptr<\_Ty1> dynamic\_pointer\_cast(const shared\_ptr<\_Ty2>& \_Other)  {  *// return shared\_ptr object holding dynamic\_cast<\_Ty1 \*>(\_Other.get())*  return (shared\_ptr<\_Ty1>(\_Other, \_Dynamic\_tag()));  }  class Car {...};  class Truck : public Car {...};    *// Truck 타입의 객체를 Car 타입의 객체를 참조하는 shared\_ptr에 초기화*  shared\_ptr<Car> pCar( new Truck() );    *// shared\_ptr<Car>가 참조하고 있던 객체를 Truck 타입으로 static\_cast하여 대입.*  *// 대입 하였기에 참조 카운트는 '2'*  shared\_ptr<Truck> pTruck = static\_pointer\_cast<Truck>(pCar);    *// 위처럼 대입하지 않고 스스로 형변환만 하여도 상관없음.*  *// 참조 카운트는 당연히 변화가 없다.*  static\_pointer\_cast<Car>(pCar); deleter를 사용하자 참조 카운트가 0이 되었을 때, 수행할 함수를 선택합니다. 디폴트 값은 객체를 삭제하는 반면, 이 함수를 정해놓으면 원하는 동작을 할 수 있습니다.  *//정의 부분입니다.*  *//인자를 2개로 받을 시, 생성자로*  *//2번째 인자에 deleter로 사용할 함수를 넣습니다.*  template<class \_Ux, class \_Dx>  shared\_ptr(\_Ux \*\_Px, \_Dx \_Dt)  { *// construct with \_Px, deleter*  \_Resetp(\_Px, \_Dt);  }  *//사용자 정의 삭제자 함수(인자에 넣고자 하는 객체를 받아야 합니다.)*  void delete(Archer\* pa){  *//왜 포인터로 제거가 안 될까?*  *//포인터가 아닌 객체(get())을 delete한다.*  delete spI.get();  delete i2;  };  void test(){  shared\_ptr<Archer> sp(new Archer, deleter);  }  참조자와 포인터의 차이점과 연관된 스마트 포인터 동작 - *[항목21]*  참고: <http://sweeper.egloos.com/2826435> |
| 특징 |
| 참조 카운팅 방식 스마트 포인터(RCSP) 위의 스마트 포인터들과 달리 한 객체를 여러 포인터에서 참조가 가능하다. 그래서 STL 컨테이너 환경에서 사용하기 적합하다.  이는 shared\_ptr의 참조 구조가 어떻게 되어 있는지 상세히 보여준다. 오른쪽은 2개 이상일 때 별도의 객체에서 카운트를 관리하고 있음을 확인할 수 있다. 아래 그림에서 다시 한 번 간단하게 살펴보자.    위에서는 하나의 객체를 참조하는데 관리자 객체가 따로 존재하며 그 객체를 참조하는 포인터를 카운트로 관리한다. Question. weak\_ptr은 뭐지? shared\_ptr의 객체만 참조할 뿐, shared\_ptr의 reference count를 올리지 않는 것이다.  사실 weak\_ptr이 shared\_ptr을 참조할 때 shared\_ptr의 weak reference count는 증가시킨다. 다만 객체의 생명 주기에 관여하는 strong reference count를 올리지 않는 것뿐이다. 위의 그림을 보면 이해가 쉽다. weak\_ptr을 써보자! 객체의 수명과 관계없는 포인터를 사용해야 할 때가 있다. shared\_ptr에는 다음과 같은 문제가 있다. Circular references 레퍼런스 카운팅 기반이기에 순환 참조에 대한 잠재적인 문제가 있을 수 있다.  즉, A와 B가 서로에 대한 shared\_ptr을 들고 있으면 레퍼런스 카운트가 0이 되지 않아 메모리가 해제되지 않는다. 아래 예제는 shared\_ptr을 서로 참조하여 발생하는 문제를 보여준다.  class Archer{ };  class Weapon{  public:  Weapon(const string& n) : name(n) {};  string getWeaponInfo();  private:  const string name;  };  string Weapon::getWeaponInfo(){  return name;  }  *//public 상속입니다.*  class ArchMage : public Archer{  public:  Archer\* goodMorningArcher();  void addWeapon(const shared\_ptr<Weapon>& w);  void showWeapon();  private:  shared\_ptr<Weapon> weapon;  };  void ArchMage::addWeapon(const shared\_ptr<Weapon>& w){  weapon = w;  }  void ArchMage::showWeapon(){  cout << "my Weapon is ";  cout << weapon->getWeaponInfo() << endl;  }  class MasterPlan : public Weapon{  public:  MasterPlan(const string& n) : Weapon(n) {};  void setMaster(const shared\_ptr<Archer>& a);  private:  shared\_ptr<Archer> master;  };  void MasterPlan::setMaster(const shared\_ptr<Archer>& a){  master = a;  }  Archer\* ArchMage::goodMorningArcher(){  return new Archer();  }  void inSpace(shared\_ptr<ArchMage>& player){  cout << "in Some Space..." << endl;  *//무기를 획득하였습니다.*  *//객체를 생성하고 그것을 가리키는 포인터를 만들어서 참조 카운터 1입니다.*  shared\_ptr<Weapon> dropItem(new MasterPlan("MasterPlan"));  *//플레이어의 weapon이 dropItem을 가리킵니다.*  *//이로써 dropItem의 참조 카운터는 2입니다.*  player->addWeapon(dropItem);  *//이제 나의 무기는 MasterPlan입니다.*  player->showWeapon();  *//이 아이템의 소유주는 플레이어입니다.*  *//이 함수는 예제용입니다.*  *//파생 클래스로 형변환하여 파생 클래스의 함수를 호출합니다.*  *//본래의 포인터와 dropItem이 가리키는 포인터까지*  *//player의 참조 카운터는 2입니다.*  static\_pointer\_cast<MasterPlan>(dropItem)->setMaster(player);  *//이 공간을 벗어나면 아이템은 사라지는 겁니다.*  cout << "exit from the Space!" << endl;  }  int main(void)  {  *//플레이어 등장!*  shared\_ptr<ArchMage> player(new ArchMage());  *//공간에서 무슨 일이 일어나는 거지?*  inSpace(player);  *//공간에서 득템했어요.*  *//여전히 나의 무기는 MasterPlan입니다.*  *//공간을 벗어났지만 참조 카운터(1)가 남아있으므로 무기는 그대로입니다.*  player->showWeapon();  *//함수가 종료되어도 객체는 사라지지 않습니다.*  *//본래의 player의 포인터가 사라져도 아까 dropItem의 참조가 그대로 남아있고*  *//dropItem도 계속해서 그 player포인터가 참조하고 있기 때문입니다.*  *//이렇게 서로 참조하면 객체를 소멸해도 제거되지 않습니다.*  return 0;  }  위 코드에서 생긴 문제를 weak\_ptr로 바꾼 아래 코드가 보인다. 비교해보자.  *//public 상속입니다.*  class ArchMage : public Archer{  public:  Archer\* goodMorningArcher();  *//void addWeapon(const shared\_ptr<Weapon>& w);*  void addWeapon(const weak\_ptr<Weapon>& w);  void showWeapon();  private:  *//shared\_ptr<Weapon> weapon;*  weak\_ptr<Weapon> weapon;  };  */\*void ArchMage::addWeapon(const shared\_ptr<Weapon>& w){*  *weapon = w;*  *}\*/*  void ArchMage::addWeapon(const weak\_ptr<Weapon>& w){  weapon = w;  }  void ArchMage::showWeapon(){  cout << "my Weapon is ";  *//weak\_ptr<\_Ty>는 \_Ty 포인터에 대해 직접 access가 불가능*  *//cout << weapon->getWeaponInfo() << endl;*  *//그래서 위와 같은 코드는 안 되고*  *//아래처럼 shared\_ptr에 lock()을 이용해 복사하여 객체에 접근한다.*  *//weak\_ptr은 shared\_ptr로만 복사, 대입이 가능하다.*  shared\_ptr<Weapon> pW = weapon.lock();  cout << pW->getWeaponInfo() << endl;  }  void inSpace(shared\_ptr<ArchMage>& player){  cout << "in Some Space..." << endl;  *//무기를 획득하였습니다.*  *//객체를 생성하고 그것을 가리키는 포인터를 만들어서 참조 카운터 1입니다.*  shared\_ptr<Weapon> dropItem(new MasterPlan("MasterPlan"));  *//플레이어의 weapon이 dropItem을 가리킵니다.*  *//이로써 dropItem의 참조 카운터는 강한 참조1, 약한 참조1입니다.*  player->addWeapon(dropItem);  *//이제 나의 무기는 MasterPlan입니다.*  player->showWeapon();  static\_pointer\_cast<MasterPlan>(dropItem)->setMaster(player);  *//이 공간을 벗어나면 아이템은 사라지는 겁니다.*  cout << "exit from the Space!" << endl;  }  int main(void)  {  *//플레이어 등장!*  shared\_ptr<ArchMage> player(new ArchMage());  *//공간에서 무슨 일이 일어나는 거지?*  inSpace(player);  *//공간에서 득템했어요.*  *//내 무기가 사라졌어요. 공간을 벗어났기 때문이죠.*  player->showWeapon();  return 0;  }    위처럼 weak\_ptr은 생명과는 연관이 없는 참조이다. 남아있어도 소멸되면 사라지는데 영향을 주지 않는다. 차이점을 분명히 알고 사용해야 한다. 멀티스레드 안정성 레퍼런스 카운트에 대해서만 동기화를 해서 멀티스레드에서의 안정성을 얻는다. shared\_ptr 배열로 구현하기 *//기본적으로 shared\_ptr은 new, delete만 호출한다.*  *//delete[]를 호출하지 않기 때문에 아래 코드 문제 발생*  std::shared\_ptr<int> spi( new int[1024] );  *//아래와 같이 벡터로 구현해야 한다.*  std::vector< std::shared\_ptr<int> > spVec;  spVec.push\_back( std::shared\_ptr<int>( new int(3) ) );  boost의 scoped\_array나 shared\_array를 사용할 수도 있다.  deleter연산자를 이용할 수도 있다. 예시는 아래와 같다.  *// deleter 클래스 정의*  template<typename T>  struct ArrayDeleter  {  void operator () (T\* p)  {  delete [] p;  }  };    *// shared\_ptr 생성시 두 번째 인자로 deleter class를 넘기면...*  *// 아무런 문제없이 객체 배열도 제대로 delete [] 처리가 된다.*  std::shared\_ptr<int> spi( new int[1024], ArrayDeleter<int>() ); |

***자원 누출을 막기 위해, 생성자 안에서 자원을 획득하고 소멸자에서 그것을 해제하는 RAII객체를 사용합니다.***

***일반적으로 널리 쓰이는 RAII클래스는 shared\_ptr과 auto\_ptr입니다. 이 둘 가운데 shared\_ptr이 복사 시의 동작이 직관적이기 때문에 대개 더 좋습니다. 반면 auto\_ptr은 복사되는 객체(원복 객체)를 null로 만들어버립니다.***

[항목14] 자원 관리 클래스의 복사

위에서 언급한 자원 획득 즉 초기화(RAII)에 대해 명확히 알았을 것이다. 지금까지 힙에서 생성되는 객체에 대해 알아보았다.

# 모든 자원이 힙에서 생기지는 않는다?

스마트 포인터로 힙에 생성된 객체는 참조 카운팅을 한다든지, 하나의 포인터만 갖게 한다든지로 객체를 관리한다. 그리고 문제없이 메모리 반납이 가능하게 한다.

하지만 스택에 저장되는 자원, 데이터 영역에 저장되는 전역 변수도 있다. 스택? 스택 내의 자원은 스스로 소멸하지 않나? 하지만 자원 관리라는 것이 꼭 소멸 시 메모리 반납만을 말하는 것은 아닐 테니 말이다.

#### 스마트 포인터는 일반적으로 힙에서 생기는 자원을 관리한다.

그러므로 나머지는? **필요성에 따라 우리가 직접 ‘자원 관리 객체’를 만들어주어야 한다.**

# 직접 자원 관리 객체를 만들어보자!

자원 관리 클래스를 직접 만들고, 그 클래스의 동작에 대해 진지하게 고민해보자.

#### 일단 만들어 봅시다.

교재에 있는 뮤텍스 예제를 본떠서 만들 것이다.

|  |
| --- |
| #include *<iostream>*  #include *<memory>*  #include *<string>*  #include *<thread>*  #include *<mutex>*  **using** **namespace** std;  *//뮤텍스 생성*  mutex m;  *//뮤텍스 잠금과 잠금해제를 저절로 해주는 객체*  *//unlock을 내가 호출할 필요가 없잖아!*  **class** **Lock**{  **public**:  **explicit** Lock(mutex \*pm) : mutexPtr(pm) {  mutexPtr->lock();  }  ~Lock(){ mutexPtr->unlock(); }  **private**:  *//만약 shared\_ptr이나 unique\_ptr로 선언하면 런타임 에러가 생긴다.*  *//추정컨대 이놈들이 뮤텍스를 마음대로 제거해버리므로*  *//이후에 뮤텍스에 접근할 때 혹은 해제할 때 문제가 생기는 듯 하다.*  *//shared\_ptr<mutex> mutexPtr;*  *//unique\_ptr<mutex> mutexPtr;*  *//이 뮤텍스는 본문 뮤텍스를 참조하는 뮤텍스다.*  mutex \*mutexPtr;  };  void print\_thread\_id(int id) {  *//critical section (exclusive access to cout signaled by locking mtx)*  *//뮤텍스에 잠금을 겁니다. 스스로 해제도 합니다.*  *//스레드들이 차례대로 들어옵니다. 만약 Lock이 없다면 뒤죽박죽 출력이 되겠죠?*  Lock m1(&m);  *//복사를 시도합니다???*  *//Lock m2(m1);*  std::cout << "thread #" << id << '\n';  }  int main(){  *//스레드 10개 생성*  std::**thread** threads[10];  **for** (int i = 0; i<10; ++i)  threads[i] = std::**thread**(print\_thread\_id, i + 1);  **for** (**auto**& **th** : threads) th.join();  cout << "test for mutex." << endl;  **return** 0;  }  위에서 자원관리 클래스의 생성은 성공적인가? 물론 unlock을 충실하게 수행해 줄 것이다. **그러나 복사가 된다면?** **모든 자원들은 복사와 대입에 관한 연산에 대한 대비가 되어야 한다. 언제나 일어날 수 있기 때문이다.** |

###### Question. 뮤텍스(mutex)가 무엇인가요?

**뮤텍스**: 뮤텍스는 화장실에 들어가기 위한 열쇠로 비유할 수 있습니다. 즉 화장실에 들어갈 수 있는 열쇠를 한 사람이 갖고 있다면, 한 번에 열쇠를 갖고 있는 그 한 사람만이 들어갈 수 있습니다. 화장실에 열쇠를 갖고 있는 사람이 들어가 볼일을 다 본 후에는 줄을 서서 기다리고 있는(대기열-큐) 다음 사람에게 열쇠를 주게 됩니다.

공식적인 정의(심비안 개발자 라이브러리에서 발췌): 뮤텍스는 한 번에 하나의 쓰레드만이 실행되도록 하는 재 입장할 수 있는 코드 섹션에 직렬화된 접근이 가능하게 할 때 사용됩니다. 뮤텍스 객체는 제어되는 섹션에 하나의 쓰레드만을 허용하기 때문에 해당 섹션에 접근하려는 다른 쓰레드들을 강제적으로 막음으로써 첫 번째 쓰레드가 해당 섹션을 빠져나올 때까지 기다리도록 합니다.

**세마포어**: *뮤텍스는 값이 1인 세마포어입니다.* 세마포어는 빈 화장실 열쇠의 개수라고 보면 됩니다. 즉, 네 개의 화장실에 자물쇠와 열쇠가 있다고 한다면 세마포어는 열쇠의 개수를 계산하고 시작할 때 4의 값을 갖습니다. 이 때는 이용할 수 있는 화장실 수가 동등하게 됩니다. 이제 화장실에 사람이 들어갈 때마다 숫자는 줄어들게 됩니다. 4개의 화장실에 사람들이 모두 들어가게 되면 남은 열쇠가 없게 되기 때문에 세마포어 카운트가 0이 됩니다. 이제 다시 한 사람이 화장실에서 볼일을 다 보고 나온다면 세마포어의 카운트는 1이 증가됩니다. 따라서 열쇠 하나가 사용가능하기 때문에 줄을 서서 기다리고 있는 다음 사람이 화장실에 입장할 수 있게 됩니다.

공식적인 정의(심비안 개발자 라이브러리에서 발췌): 세마포어는 공유 리소스에 접근할 수 있는 최대 허용치만큼 동시에 사용자 접근을 할 수 있게 합니다. 쓰레드들은 리소스 접근을 요청할 수 있고 세마포어에서는 카운트가 하나씩 줄어들게 되며 리소스 사용을 마쳤다는 신호를 보내면 세마포어 카운트가 하나 늘어나게 됩니다.

#### 복사에 관한 처리를 합시다.

완성된 RAII 객체를 만들려면 복사에 대해서도 대비가 되어야 합니다. 앞에서 보았던 스마트 포인터들은 각각 복사에 대해 다른 특징을 가지고 있었습니다. 우리가 만든 객체 관리 클래스는 필요에 따라 구현할 줄 알아야겠습니다.

**[추천] 복사 금지**

|  |
| --- |
| *//기억하는가? 복사금지 private상속*  **class** **Uncopyable**{  **protected**:  Uncopyable(){};  ~Uncopyable(){};  **private**: *//접근 불가능 + 정의 안 됨*  Uncopyable(**const** Uncopyable&);  Uncopyable& **operator**=(**const** Uncopyable&);  };  **class** **Lock** : **private** Uncopyable{  };  복사 금지법은 앞에서 다룬 내용이다(**[항목6]**). |

**[추천] 관리하고 있는 자원에 참조 카운팅을 수행합니다.**

**shared\_ptr**이 생각나는군요. 이를 이용하여 mutex를 참조합시다. 하지만 이는 제대로 작동하지 않습니다. 참조 카운트0이 되면 뮤텍스를 지워버리거든요. 잠금 해제만 해야 하는데 지워버리면 다음에 못쓰겠네요. 그래서 우리는 **deleter(항목13에서 언급됨)**을 지정하여 우리에게 맞는 shared\_ptr을 사용합시다. 다음과 같이요.

|  |
| --- |
| *//삭제자 함수는 객체를 받아야 합니다.*  void deleter(mutex\* pm){  pm->unlock();  };  **class** **Lock** : **private** Uncopyable{  **public**:  **explicit** Lock(mutex \*pm)  ***//삭제자로 unlock을 지정합니다.***  : mutexPtr(pm, deleter) {  mutexPtr->lock();  }  *//~Lock(){ mutexPtr->unlock(); }*  **private**:  shared\_ptr<mutex> mutexPtr;  *//mutex \*mutexPtr;*  }; |

관리하고 있는 자원을 진짜로 복사합니다.

**깊은 복사(deep copy)**를 의미한다. 앞에서 언급되었지만 얕은 복사는 포인터만 옮기는 과정이며 깊은 복사는 또 하나의 데이터 공간을 할당하여 원본과 똑같이 만들어내는 것이다.

관리하고 있는 자원의 소유권을 옮깁니다.

**auto\_ptr**의 동작을 말한다. 포인터를 옮기고 기존의 객체는 null로 만들어서 해당 객체를 가리키는 포인터를 유일하게 만든다.

***RAII객체의 복사는 그 객체가 관리하는 자원의 복사 문제를 안고 가기 때문에, 그 자원을 어떻게 복사하느냐에 따라 RAII 객체의 복사 동작이 결정됩니다.***

***RAII클래스에 구현하는 일반적인 복사 동작은 복사를 금지하거나 참조 카운팅을 해주는 선에서 마무리하는 것입니다. 그 이외의 방법도 존재합니다.***

[항목15] 자원 관리 클래스의 접근

자원 관리 클래스의 자원은 외부에서 접근할 수 있게 합시다.

# 자원 관리 클래스의 자원이란?

해당 클래스(스마트 포인터)가 가리키는 대상(포인터가 가리키는 실제 객체)을 말한다. 당연히 해당 객체에 접근할 일은 생기기 않겠는가?

이에 접근하려면 어떻게 해야 할까?

#### 명시적 변환으로 객체 접근과 암시적 변환으로 객체 접근

|  |
| --- |
| **class** **Archer**{  **public**:  void intro();  };  void Archer::intro(){  }  Archer\* makePlayer(){  **return** **new** Archer();  }  void tutorial(**const** Archer\* a){  }  void darkWorld(){  shared\_ptr<Archer> pA(makePlayer());  *//아래는 인자가 포인터므로 shared\_ptr을 넘기면 에러가 발생합니다.*  *//tutorial(pa);*  *//아래와 같이 실제 자원(즉, pA가리키는 주소)을 넘겨야 합니다.*  *//get()으로 포인터를 객체 자체를 얻습니다(명시적 변환).*  tutorial(pA.get());  *//아래와 같이 \*, ->연산자도 사용할 수 있습니다(암시적 변환).*  (\*pA).intro();  pA->intro();  *//명시적 변환은 번거롭군요.*  pA.get()->intro();  }  int main(){  darkWorld();  }  해당 자원에 접근하기 쉽게 하기 위한 명시적 변환과 암시적 변환을 보여준다. |

# 자원 관리 클래스에서 자원 접근하기

#### 암시적 변환으로 자원 접근

즉, 명시하지 않아도 스마트 포인터나 혹은 자원을 갖는 클래스를 저절로 자원으로 바꿔주는(Player->Archer) 방법이다. 아래 코드와 같다.

**class** **Player**{

**public**:

Player(**const** shared\_ptr<Archer>& a) : pA(a) {};

*//oerator Archer()은 player클래스를 이의 자원인 Archer로 바꿉니다.*

*//바로 암시적으로요.*

**operator** Archer() **const**{ **return** \*pA; }

**private**:

shared\_ptr<Archer> pA;

};

Archer\* makeArcher(){

**return** **new** Archer();

}

void tutorial(**const** Archer& a){ }

void darkWorld(){

shared\_ptr<Archer> pA(makeArcher());

tutorial(\*pA);

*//pa는 p1의 자원입니다.*

Player p1(pA);

*//Archer의 참조를 인자로 받지만 Player도 허용됩니다. 암시적 변환이니까요.*

tutorial(p1);

}

int main(){

darkWorld();

}

#### 명시적 변환으로 자원 접근

어딘가가 좀 불편해 보인다.

**class** **Player**{

**public**:

Player(**const** shared\_ptr<Archer>& a) : pA(a) {};

Archer& get() **const** { **return** \*pA; }

**private**:

shared\_ptr<Archer> pA;

};

void darkWorld(){

tutorial(p1.get());

}

#### 무엇을 선호해야 할까?

암시적 변환이 편하기는 하지만 이는 장점만 있는 것은 아니다. 실수로 원하지 않는 복사가 일어날 수 있다.

###### **“맞게 쓰기는 쉽게, 틀리게 쓰기는 어렵게”** 라는 원칙을 지킨다**([항목18]**).

특정한 용도와 사용 환경에 따라 만들어주어야 할 것이다.

###### 캡슐화에 대한 고찰

자원을 공개하는 것은 캡슐화에 위배된다. 하지만 shared\_ptr도 참조 카운팅 메커니즘에 필요한 장치들은 캡슐화하고 있지만 자신의 포인터는 쉽게 공개하는 것처럼, ***애초에 클래스가 가진 목적, 고객의 편의성***을 보면 그리 나쁘지만은 않다.

***실제 자원을 직접 접근해야 하는 기존 API들도 많기 때문에, RAII클래스를 만들 때는 그 클래스가 관리하는 자원을 얻을 수 있는 방법을 열어 주어야 합니다.***

***자원 접근은 명시적 변환 혹은 암시적 변환을 통해 가능합니다. 안전성만 따지면 명시적 변환이 대체적으로 낫지만, 고객 편의성을 놓고 보면 암시적 변환이 괜찮습니다.***

[항목16] new와 delete사용 시 형태

# new🡪delete, new배열🡪delete[]

단일 객체의 메모리와 객체 배열에 대한 메모리 구조가 다르기 때문이다. 객체의 배열에는 메모리에 첫 공간에 배열 크기가 들어간다.

#### 이렇게 해야 하는 이유

출처: <http://blog.daum.net/jsg3001kr/48>

new연산자에 대한 자세한 고찰은 [항목49] 참고

**CRT Library에서 동적메모리를 할당해제하는 함수는 malloc free다.** new는 내부적으로malloc를 호출하고 delete도 free를 내부적으로 호출한다.

즉, new = malloc(free) + 생성자, 소멸자 호출

malloc은 클래스객체를 동적할당할경우 해당메모리덩어리를 가리키는 void\*반환하기때문에 어떤 클래스타입의 객체가 생성되는지알수없다. free역시 오직인자로 받은 포인터의 주소를 가지고 힙관리자에게 해당포인터가 가리키는 메모리블록을 해제하는일만 하기때문이다.

new뒤에 생성하고자 하는 객체의 타입이 들어가는 이유이다. 그 타입으로malloc을 한다. 컴파일러는 대상 포인터의 타입만을 체크하여 해당타입과 일치하는 클래스의 소멸자를 호출한 뒤에 메모리영역을 해제하는 어셈블리를 작성한다.

DEF\* pA = new DEF[2]; //배열요소 2개의 객체배열 동적할당

delete [] pA ;

컴파일러는 어떻게 생성자와 소멸자가 두번씩 호출해야된다는걸 알까?

생성자야 2라고 써넣엇으니 2번호출하는게 당연하다. 근데 소멸자는 [] 밖에 안쓰였는데 어떻게 2번 호출될까?

힙관리자는 메모리할당내역을 관리하는데 할당내역이란 메모리주소와 크기정보이다. 컴파일러는 할당내역을 통해서 검색, 추가, 삭제하는 과정을 한다. 즉new[] delete[]는 할당내역의 변화를 일으켜야 한다. (배열이니까) **new 와 delete는 malloc과 free를 포함한다고 했는데 할당내역의 변화는 내부에서 호출되는 malloc과 free이다!**

단일타입먼저 자세히살펴보자

기본타입 int에대해서 new를 할경우 내부적으론malloc이 호출되고 malloc은 힙관리자에게 메모리할당을 요청하며 힙관리자가 관리하고있는 힙에서 적절히 4바이트만큼의 메모리블록을 찾은뒤에 해당 블록의 주소를 반환할 것이다. 동시에 힙관리자는 해당블록의 주소를 해당내역에 추가한다. malloc을 통해서 할당받은 메모리블록의 주소를 new는 그대로 다시 반환한다. 그래서 int타입포인터는 힙관리자가 할당해준 메모리블록주소를 가지게된다

그렇다면 배열객체를 new할경우에는 어떻게될까?

pA는 int형 타입을 하나갖고있는데 배열로 생성했으니 당연히 필요한 메모리공간은 8바이트라고 생각할수있다 (int \* 2 니까 8바이트)  
**하지만 다르다 new[]로 할경우엔 4바이트가 더 할당된다. 즉 12바이트의 메모리 할당을 요청한다**

왜 4바이트를 더 추가적으로 생성할까?  
**바로 소멸자를 배열갯수만큼 호출하기위해서다!! 추가로할당된 4바이트엔 배열요소갯수가 저장되며 여기선 2가 저장된다.**

즉, delete[]를 할경우 해당객체가 배열타입할당임을 알고 뒤에 추가로 할당된 4바이트에서 배열 요소 갯수를 알고 이 4바이트는 없앤다.

그리고 그 갯수만큼 반복하면서 소멸자를 호출한다. 물론 delete는 free한테 4바이트를 뺀값을 인자로 넘긴다. 힙관리자는정확히 12바이트를 해제한다.

만일 실수로 배열타입한테 []없이 delete만할경우 소멸자는 한번만호출될꺼고 pA의 주소를 그대로 free에 인자로 넘길것이다. 그러나 힙관리자는 할당내역에서 pA의 주소를 찾을수 없어 메모리는 제대로 해제되지않게되며 즉시 오류가 발생이안될수도있기때문에 이 부분은 조심해야한다

그냥 기본타입의 배열이나 소멸자가없는 클래스타입의 배열을 new[]로하면어떻게될까?

소멸자를 부를필요가없기때문에 추가로 할당되는 4바이트는 불필요해 할당되지않는다

단일 객체할당을 위한 new와 크게다를바없으며 delete delete[]의 구분이 무의미해지는것이다

즉 소멸자가없으면 new[]를 해도 delete만 해도되는것이다

그러니까 배열객체로 new[]를 하고 delete를 []없이 써도 컴파일에러는 나지 않는다. 소멸자가없으니 의미가 없다

즉 new[] 썼을떄 추가로 할당되는 4바이트는 소멸자를 위해 존재하는것이며 delete를 써도되긴하다

하지만 좋지않다. 유지보수과정중에 언제클래스가 추가될지모르는 일이기때문이다 new[]는 delete[]가 좋다

소멸자가없는 클래스타입?

위에서 소멸자가없는 클래스타입이라고했는데 우리가 소멸자함수를 정의하지않으면 소멸자함수는 없는거다

암시적으로 컴파일러가 만들이유도 없고 말이다.

그렇다고 소멸자를 정의안했다고 항상 소멸자가없다고 보장할순 없다 (지금하는 이야기는 꼭 new delete뿐만아니라 소멸자자체에대한설명)

만약 상속관계의 경우 (따로 코드를 치진않겠다. 부모클래는 소멸자가정의되있고 유도클래스는 소멸자가없다고 가정)

자식클래스를 동적할당하고 해제할경우 해제할떈 부모클래스의 생성자와 소멸자가 호출되야된다.

**부모클래스의 소멸자가 정의되어있기떄문에 해제하기위해선 자식클래스에 소멸자가없더라도 암시적으로**

**자식클래스소멸자를생성한다**

###### void\*에 대한 고찰

malloc이 반환값으로 갖는 void\*에 대해 생각해보자.

**void 타입**은 C에서 파생된 여러 언어에서 정상적으로 반환하지만 호출자에게 결과 값을 제공하지 않는 함수의 결과를 위한 타입이다. 보통 이러한 함수들은 몇몇 작업을 수행하거나 자신의 결과 파라미터를 쓰는것 같은 그것의 [부작용](https://ko.wikipedia.org/wiki/%EB%B6%80%EC%9E%91%EC%9A%A9_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99))을 위해 호출된다. 이러한 문맥에서 void 타입의 사용은 [비주얼 베이직](https://ko.wikipedia.org/wiki/%EB%B9%84%EC%A3%BC%EC%96%BC_%EB%B2%A0%EC%9D%B4%EC%A7%81)의 [서브루틴](https://ko.wikipedia.org/wiki/%ED%95%A8%EC%88%98_(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D))과 [파스칼](https://ko.wikipedia.org/wiki/%ED%8C%8C%EC%8A%A4%EC%B9%BC_(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%96%B8%EC%96%B4))의[함수](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EC%8B%9C%EC%A0%80)를 정의하는 통사적 구조체의 그것과 차별화된다. 또한 [함수형 프로그래밍](https://ko.wikipedia.org/wiki/%ED%95%A8%EC%88%98%ED%98%95_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)과 [타입 이론](https://ko.wikipedia.org/w/index.php?title=%ED%83%80%EC%9E%85_%EC%9D%B4%EB%A1%A0&action=edit&redlink=1)에서 사용되는 [유닛 타입](https://ko.wikipedia.org/w/index.php?title=%EC%9C%A0%EB%8B%9B_%ED%83%80%EC%9E%85&action=edit&redlink=1)과 비슷하지만 void 타입은 값이 없는 빈 타입이라는 점에서 허용되는 사용법에 몇몇 차이점이 있다.

C와 [C++](https://ko.wikipedia.org/wiki/C%2B%2B) 또한 **void type에 대한 포인터** ( void \*로 명시된)를 제공하지만, 이것은 개념과는 관련없다. 이 타입의 변수들은 명시되지 않은 타입의 데이터에 대한 [포인터 (프로그래밍)](https://ko.wikipedia.org/wiki/%ED%8F%AC%EC%9D%B8%ED%84%B0_(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D))여서, 이 문맥의 경우 void는 일반적인 또는 [탑 타입](https://ko.wikipedia.org/w/index.php?title=%ED%83%91_%ED%83%80%EC%9E%85&action=edit&redlink=1)처럼 행동한다. 프로그램은 아마 어느 데이터의 타입에 대한 포인터라도([함수 포인터](https://ko.wikipedia.org/wiki/%ED%95%A8%EC%88%98_%ED%8F%AC%EC%9D%B8%ED%84%B0) 제외) void에 대한 포인터로 변환할 수 있으며, 정보를 잃는것 없이 원래의 타입으로 돌아갈 수 있는데, 이러한 포인터들은 [다형성](https://ko.wikipedia.org/wiki/%EB%8B%A4%ED%98%95%EC%84%B1) 함수들에 유용하다. C 언어 표준은 다른 포인터 타입들이 같은 크기를 갖는다고 보장하지 않는다.

**void\*는 일반적인 포인터 타입으로 모든 타입을 캐스팅 없이 받아들일 수 있다.** 이는 참조하거나 연산할 수는 없는데, 다른 타입으로 전환된 후에 그것이 가능하다. 같은 코드 안에 타입에 따른 연산을 하고 싶을 때 유리하다. 아래 예시가 있다.

void qsort(void \*base, size\_t nmemb, size\_t size,

int (\*compar)(const void \*, const void \*));

base is the address of an array, nmemb is the number of elements in the array, size is the size of each element, and compar is a pointer to a function that compares two elements of the array. It gets called like so:

int iArr[10];

double dArr[30];

long lArr[50];

...

qsort(iArr, sizeof iArr/sizeof iArr[0], sizeof iArr[0], compareInt);

qsort(dArr, sizeof dArr/sizeof dArr[0], sizeof dArr[0], compareDouble);

qsort(lArr, sizeof lArr/sizeof lArr[0], sizeof lArr[0], compareLong);

The array expressions iArr, dArr, and lArr are implicitly converted from array types to pointer types in the function call, and each is implicitly converted from "pointer to int/double/long" to "pointer to void".

The comparison functions would look something like:

int compareInt(const void \*lhs, const void \*rhs)

{

const int \*x = lhs; // convert void \* to int \* by assignment

const int \*y = rhs;

if (\*x > \*y) return 1;

if (\*x == \*y) return 0;

return -1;

}

By accepting void \*, qsort can work with arrays of any type.

The disadvantage of using void \* is that you throw type safety out the window and into oncoming traffic. There's nothing to protect you from using the wrong comparison routine:

qsort(dArr, sizeof dArr/sizeof dArr[0], sizeof dArr[0], compareInt);

compareInt is expecting its arguments to be pointing to ints, but is actually working with doubles. There's no way to catch this problem at compile time; you'll just wind up with a mis-sorted array.

###### typedef *사용법*

typedef 선언은 typedef를 저장소 클래스로 사용하는 선언입니다. 선언자는 새 형식이 됩니다.

|  |
| --- |
| **구조체**에서의 사용법  **typedef** **struct**{  int data;  } S1;  *// S1이라고 타입 정의합니다. C와 C++에서 문제없이 동작합니다.*  **struct** S2{  int data;  };  *//S2라고 타입 정의합니다.*  **struct**{  int data;  }S3;  int main(){  S1 mine1; *// 문제없이 동작합니다. 이때 S1은 typedef입니다.*  S2 mine2; *// 역시 문제없이 동작합니다. 이때 S2도 역시 typedef입니다.*  S3 mine3; *// 문제 발생! S3는 typedef가 아니라 변수입니다.*  S1.data = 5; *// 에러가 발생합니다. S1은 변수가 아닌 typedef입니다.*  S2.data = 5; *// 위와 동일*  S3.data = 5; *// 잘 동작합니다. S3자체가 변수이기 때문입니다.*  }  **함수 포인터**에서의 사용법  **typedef** void (\*ptrfunc)(int);    void testprint(int n){  printf("Number : %d**\n**", n);  }    int main(){  testprint(100);  ptrfunc elecs;  elecs = testprint;  elecs(77);    **return** 0;  } |

***new표현식에 []에 썼으면, 대응되는 delete 표현식에도 []를 써야 합니다. 마찬가지로 new표현식에 []를 안 썼으면, 대응되는 delete 표현식에도 []를 쓰지 말아야 합니다.***

[항목17] new생성객체시 주의사항

바로 생성한 객체를 스마트 포인터에 저장하는 코드는 별도의 한 문장으로 만들자.

# 인자에서 객체를 생성한다면 주의하자.

밑의 코드에서 문제점을 발견하기 바란다.

void battle(shared\_ptr<Archer> pw, int level){

cout << "on Battle!" << endl;

}

int getLevel(){

int a; cin >> a;

**return** a;

}

int main(){

*//아래처럼 포인터 형식(new로 생성된 객체)가 shared\_ptr형으로 바뀔 수 없다.*

*//battle(new Archer(), 1);*

*//아래와 같이 명시적으로 객체를 새로 만들어서 넘겨주는 것도 방법이다.*

battle(shared\_ptr<Archer>(**new** Archer()), getLevel());

*//여기서 3가지를 호출하게 된다.*

***//객체 생성: new Archer***

***//getLevel() 함수 호출***

***//shared\_ptr 생성자 호출***

***//이 3가지의 순서가 일정치 않으므로 객체를 생성하고 예외가 생겨***

***//프로그램이 죽을 수도 있다.***

}

#### 문제점:

new연산자로 Archer객체 생성 후 자원을 관리해주는 **shared\_ptr에 저장되기도 전에 다른 함수에서 예외가 끼어들면서** 결국 메모리가 낭비되면서 끝나버릴 것이다.

#### 해결책

new객체 생성과 shared\_ptr에 담는 코드를 따로 빼준다.

int main(){

*//new로 생성한 객체를 스마트포인터에 담는 코드를 독립적으로 둔다.*

shared\_ptr<Archer> pa(**new** Archer());

battle(pa, getLevel());

}

###### 왜 이런 문제가 생길까?

어떻게 저장하고 어떤 순서로 계산할것인가. 사용자는 사용할 데이터를 정의한다. 그럼 그 정보를 메모리에 저장하고 필요하다면 변수에 저장하여 여러 번 사용할 수 있다. 그리고 모두 사용했다면 메모리를 해제한다. 때로는 직접 해제해야 할 수도 있다.

때로는 잠깐만 필요할 수도 있다. 하나의 표현식에서 어떤 정보(객체)를 생성하지만 저장할 필요는 없는 것이다.   
battle(shared\_ptr<Archer>(**new** Archer()), getLevel());  
가 그 예가 되겠다.

그런데 문제는 뭐냐면 하나의 실행 단위는 여러 개의 실행 단위로 묶여 있다는 점이다. 그리고 사용자는 각 실행 단위를 모두 통제할 수 없다. 예외가 발생할 수 있고, 그 예외는 다른 단위에 영향을 주어서는 동작을 통제할 수 없다. 그러므로 하나의 실행 단위에 대해 성공을 보장해야 하며, 그렇지 못하다면 기존의 정보가 날라가지 않게 저장해두어야 한다. 혹은 그에 맞는 사전처리가 필요하다. 그냥 기본적인 문서작성 이야기다.

***new로 생성한 객체를 스마트 포인터로 넣는 코드는 별도의 한 문장으로 만듭시다. 이것이 안 되어 있으면, 예외가 발생할 때, 디버깅하기 힘든 자원 누출이 초래될 수 있습니다.***

[항목18] 인터페이스 설계

제대로 쓰기에는 쉽게, 엉터리로 쓰기에는 어렵게 설계해야 한다.

# 함수, 클래스, 템플릿 모두 인터페이스입니다.

일반화하면, 코드와 소통하기 위한 모든 수단이다.

# 제대로 쓰기에는 쉽고 엉터리로 쓰기에는 어렵게 설계하려면?

#### [제안1] 사용자들이 할만한 실수를 고려한다.

###### 매개변수를 잘못 사용할 수 있다.

여러 개의 매개변수가 있어 섞어 쓰거나 오타를 만들 때 에러를 잡아줄 수 있어야 한다. 이는 새로운 타입을 들여와 인터페이스를 강화할 수 있다. 적절한 타입이 준비되어 있는 경우에는 각 타입 값에 static함수로 제약을 가하면 더욱 안전해진다.

**struct** Human{

**explicit** Human(int value) : data(value){};

int data;

};

**struct** Elf{

**explicit** Elf(int value) : data(value){};

int data;

};

**class** **Dark**{

**public**:

*//사용자가 1번이 blood라는 사실을 외워야 한다면 실수가 잦겠지?*

**static** Dark blood(){ **return** Dark(1); }

**static** Dark gold(){ **return** Dark(2); }

**static** Dark role(){ **return** Dark(3); }

**private**:

*//엉뚱한 다른 값이 들어가는 것을 막기 위해* ***private에 선언***

**explicit** Dark(int value) : data(value){};

int data;

};

**class** **Potion**{

**public**:

*//이와 같은 선언은 각 매개변수에 잘못된 숫자를 넣기가 쉽다.*

Potion(int human, int elf, int dark);

***//아래처럼 새로운 타입으로 선언하면 실수를 막습니다.***

Potion(Human h, Elf e, Dark d);

};

Dark.blood를 반환하여 더 간단하게 해야지

[항목4]를 보고오기 바란다. 정적 객체의 초기화를 믿고 나갈 수 없다. 함수로 순서를 제어하는 편이 더 낫다.

###### 사용자 마음대로 어떠한 타입을 바꾸거나 대입할 수 있다.

const를 붙여서 어떤 타입에 제약을 부여하거나 다른 일을 못하도록 묶어버릴 수 있다. 어떤 타입에 대해 할 수 있는 일을 한정시켜 주는 것이다.

#### [제안2] 일관성 있는 인터페이스를 제공한다.

특별한 경우가 아니라면, 사용자 정의 타입은 기본 정의 타입과 동작이 똑같게 만들어야 합니다. 컨테이너에서 똑 같은 길이를 구하는 함수가 size, length 등 여러 가지라면 개발자는 고통을 겪을 수 있다.

#### [제안3] 사용자한테 시키지 말고 최대한 객체 스스로 모든 것을 담당해라.

사용자가 외워서 사용해야 하는 인터페이스는 좋지 않다.

###### shared\_ptr사용: 자원 관리를 사용자가 선택하게 하지 말아라.

*//어딘가에 포함된 포션을 생성하는 펙토리 함수입니다.*

*//포인터를 반환합니다.*

Potion\* createPotion(){

**return** **new** Potion(Human(1), Elf(1), Dark::blood());

}

*//스마트 포인터를 반환합니다.*

shared\_ptr<Potion> createPotionBySmart(){

**return** shared\_ptr<Potion>(**new** Potion(Human(1), Elf(1), Dark::gold()));

}

int main(){

*//아래와 같은 코드는 위험합니다.*

*//사용자가 메모리 해제를 안 할 수도 있으며 delete를 2번 사용할 수도 있다.*

*//스마트 포인터로 해결할 수 있지만*

*//사용자가 깜빡하고 스마트 포인터를 사용하지 않을 수도 있다.*

Potion\* redPotion = createPotion();

***//이제 스마트 포인터로 사용해야만 하므로 까먹을 일이 없습니다.***

**shared\_ptr<Potion> bluePotion = createPotionBySmart();**

}

###### shared\_ptr사용: 삭제자를 함께 묶어서 반환하자.

void RemovePotion(Potion\* pP){

**delete** pP;

}

int main(){

shared\_ptr<Potion> bluePotion = createPotionBySmart();

***//아래는 삭제자를 가진 null shared\_ptr을 생성한다.***

shared\_ptr<Potion> eatenPotion(**static\_cast**<Potion\*>(0), RemovePotion);

***//아래처럼 대입만 하면 삭제자가 지정된 포인터를 사용하네요!***

eatenPotion = bluePotion;

}

###### shared\_ptr사용: [교차 DLL문제]를 걱정하지 않아도 된다.

shared\_ptr은 생성된 DLL과 동일한 DLL을 쓰도록 해준다. 초기에 만들어질 때의 삭제자를 사용하도록 하기 때문입니다.

[교차 DLL문제](cross-dll-problem)

객체 생성 시, 어떤 **동적 링크 라이브러리(dynamically linked library: DLL)**의 new를 썼는데 그 객체를 삭제할 때에는 이전의 DLL과 다른 DLL을 쓰는 경우 생기는 문제이다.

###### [참고] DLL사용하기

#include *<windows.h>*

#include *<stdio.h>*

*// 두 개의 수를 추가하는 함수 가져오기*

**extern** "C" **\_\_declspec**(dllimport) int AddNumbers(int a, int b);

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

int result = AddNumbers(1, 2);

printf("결과는: %f**\n**", result);

**return** 0;

}

###### extern “C” 알아보기

참고: <http://egloos.zum.com/jimbo73/v/1486292>

컴파일러는 링커가 링킹작업시 오브젝트간 함수 이용 및 위치를 파악할 수 있도록, 컴파일시 사용된 함수에 관련한 정보를 오브젝트 파일에 기록하며, 이러한 정보를 **linkage**라고 한다.

그런데 C 와 C++을 혼합하여 사용 시, 함수이용에 문제가 발생할수 있는데, 이는 두 언어간의 linkage방식이 틀리는데 기인한다.  
  
**linkage**이란 컴파일 시 함수이름 앞 또는 뒤에 '\_' 등의 심볼을 덧붙이는 것을 말하는것으로, C 와 C++은 컴파일시 오브젝트 파일에 함수명, 변수명등에 심볼을 기록하는 방식이 다르다.

C에서는 함수의 이름이 유일하기 때문에 함수 앞에 '\_' 등의 심볼만을 붙이면 된다. 하지만 C++에서는 overloading을 지원하기 때문에 같은 이름의 함수를 여러개 가질 수 있으므로, 인자의 개수와 데이터형에 대한 정보까지 넣게 되어(이러한 방식을 mangled name이라한다.), linkage 정보가 상이하게 된다.

그러므로 C와 C++ 을 혼합하는 프로그램에서는, link시 함수명을 각각의 방식으로 찾을 수 있도록 C인지, C++인지를 명시해주어야 하며, 이때 ***extern "C"*** 는 linkage에 대한 지시자 역할을 한다.

#ifdef \_\_cplusplus

**extern** "C" {

#endif

int func1();

int func2();

#ifdef \_\_cplusplus

}

#endif

***좋은 인터페이스는 제대로 쓰기는 쉽고 엉터리로 쓰긴 어렵습니다. 인터페이스를 만들 때는 이 특성을 항상 고려합시다.***

***인터페이스의 올바른 사용을 이끄는 방법으로는 인터페이스 사이의 일관성 잡아주기, 그리고 기본제공 타입과의 동작 호환성 유지하기가 있습니다.***

***사용자의 실수를 방지하는 방법으로는 새로운 타입 만들기, 타입에 대한 연산을 제한하기, 객체의 값에 대한 제약 걸기, 자원 관리 작업을 사용자 책임으로 놓지 않기가 있습니다.***

***shared\_ptr은 사용자 정의 삭제자를 지원합니다. 이 특징 때문에 shared\_ptr은 교차 DLL문제를 막아 주며, 뮤텍스 등을 자동으로 잠금 해제하는 데 쓸 수 있습니다.***

[항목19] 클래스 설계: 기본 공식

# 클래스를 만들어보자!

c++에서 클래스를 정의한다 함은 하나의 새로운 타입을 정하는 것과 같다.

그러니 신중해야 합니다. 메모리 제어, 함수 및 연산자 중복적재, 재정의, 초기화 등…

# 무엇을 고려해야 할까?

문법이 자연스럽고, 의미 구조가 직관적이며, 효율적인 구현이 가능해야 한다.

#### 생성과 소멸

**생성자와 소멸자**를 어떻게 만들지 설계합니다.   
메모리 할당 함수(operator new, delete, new[], delete[])를 고려합니다.

#### 초기화와 대입 결정

**초기화와 대입**의 차이점을 인지하여 설계해야 한다.

#### 객체가 값에 의해 전달될 경우의 의미 결정

**참조**에 의한 전달 = 주소 전달 or  
**값**에 의한 전달 = **복사 생성자**

#### 새로운 타입이 가지는 적법한 값에 대한 제약

***불변객체(immutable object)***는 생성 후 그 상태를 바꿀 수 없는 객체를 말한다.   
반대 개념으로는 *가변(mutable)* 객체로 생성 후에도 상태를 변경할 수 있다.   
즉, const선언된 멤버이며, 다음 함수에 영향을 준다.  
 **- 생성자, 대입:** 대입하므로 값이 바뀐다.  
 **- 쓰기(setter)함수:** 값을 쓰는 함수이므로 영향을 준다.  
 - 그리고 **예외**에도 영향을 줄 수 있다.

#### 클래스 상속 계통망

상속을 시킬 시, 고려할 점이다.  
**멤버함수가 가상인지 비가상인지**를 결정된다.   
특히 **소멸자**는 더욱 중요하다.

#### 타입 변환 허용

**암시적**이거나 **명시적(explicit)**을 복사 생성자에 선언하여 어떤 타입의 변환을 허용할 지 결정할 수 있다.

#### 클래스 안에 어떤 함수와 연산자를 넣을까?

넣고 싶은 것을 넣어!

#### 허용하지 말 표준 함수를 정한다.

**금지할 함수**를 private으로 선언한다.

#### 각 멤버에 대한 접근 권한을 정하자.

어떤 멤버를 **public, private, protected** 영역에 둘 지를 정한다. 프렌드함수도 고려한다.

#### ‘선언되지 않은 인터페이스’로 무엇을 둘 것인가?

우리가 만들 타입이 제공할 보장은 어떤 종류일까?   
**수행 성능, 예외 안전, 자원 사용(잠금, 동적 메모리 등)**이 있습니다.

#### 얼마나 일반적인가?

**동일 계열 타입군**을 원한다면 템플릿 설계를 고려해야 한다.

#### 정말로 꼭 필요한 타입인가?

**기존의 클래스에서 간단한 비멤버함수나 템플릿으로 추가**하여 정의할 수 있는지를 고려한다.

#### 구현

실제 구현에서 어떻게 고려할 지 코드와 함께 이미지를 그려본다.

**class** **DarkKnight**;

**class** **BladeMaster**;

**class** **Person**{

**public**:

**virtual** void overridefunc1() = 0;

};

*//10. 이 클래스는 어떤 보장을 가지는가?*

*//12. 다른 클래스에서 비멤버 함수나 템플릿으로 해결할 수는 없나?*

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

**class** **Archer** : **public** Person{

**public**:

*//1. 생성 및 소멸*

Archer(){}

~Archer(){}

*//2. 초기화 및 대입*

Archer(**const** Archer&);

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

*//3. 참조 전달 및 값 전달*

Archer func1(Archer);

Archer& func2(**const** Archer&);

*//5. 상속 고려*

**virtual** void overridefunc1() **override**; *// : public Person*

*//6. 타입 변환 허용 여부*

**explicit** Archer(**const** DarkKnight&); *// 저절로 안 변함*

Archer(**const** BladeMaster&); *// 저절로 변함*

*//7. 내가 필요한 함수, 연산자 넣기*

void niceFunc() **const**;

*//11. 템플릿으로 일반화할 수 있는가?*

void genericFunc(T obj);

*//9. 접근 권한 설정*

**protected**:

int barrier;

**private**:

*//4. 불변속성*

**const** std::string name;

void setter(std::string&); *// 불변속성은 값을 set할 수 없다.*

*//8. 표준 함수 중 허용치 말아야 할 것*

void swap() = **delete**;

};

***클래스 설계는 타입 설계입니다. 새로운 타입을 정의하기 전에, 이번 항목에 나온 모든 고려사항을 빠짐없이 점검해 보십시오.***

[항목20] 참조자에 의한 전달 사용

**대부분**의 경우에는 값에 의한 전달보다 **참조자에 의한 전달**이 낫다.

# 간단히 말해 이렇게 전달하라는 뜻이다.

**class** **Weapon**;

**class** **Archer**;

*//****a는 값에 의한 전달****: 객체를 그대로 복사한다.*

*//매개변수를 받는 순간, 객체 a를 싹다 복사하므로*

*//생성소멸(함수가 끝날 때 반환하므로)+string::name의 생성소멸*

*//+string::info생성소멸+Weapon 포인터까지 복사 및 호출:* ***총6번***

bool isStrongArcher(Archer a);

*//****sa는 참조자에 의한 전달****: 객체의 포인터를 넘긴다.*

*//const는 값을 고정하므로 값 전달처럼 쓰이면서 효율적이다.*

bool isStrongArcher(**const** Archer& sa);

# 왜 참조자를 전달해야 하는데?

#### 생성자와 소멸자가 전혀 호출되지 않으므로 효율적이다.

앞서 본 것처럼 객체 안에 무엇이 들어있느냐에 따라 상당히 많은 생성자를 만들어낼 수 있다. 이것을 방지하는 것은 상당히 효율적이다.

#### 복사 손실 문제를 방지한다.

복사 손실에 관한 코드를 아래에 게시한다.

**class** **BladeMaster** : **public** Archer;

*//여기서 인자를 받으면 BladeMaster의 Archer부분만 복사되어*

*//나머지부분은 흔적도 없이 사라지고 a는 그냥 Archer가 된다.*

void battle(Archer a){

a.attack();

}

*//참조자를 넘김으로서 복사 손실을 방지할 수 있으므로*

***//a의 Archer부분만이 아닌 객체 그대로를 사용할 수 있다.***

void powerBattle(**const Archer &a**){

a.attack();

}

int main(){

Archer a;

BladeMaster bm;

*//결코 bladeMaster의 강력한 공격을 보여줄 수 없다.*

*//왜냐하면 battle함수의 인자를 보라.*

battle(bm);

***//이제 강력한 공격이 가능하다.*** *2번째 함수를 보라.*

powerBattle(bm);

}

# 값에 의한 전달을 사용해야 할 때

물론 대부분의 경우에 참조자 전달을 사용하지만 예외가 있다.

#### 기본제공 타입

참조자 전달이란 포인터를 복사하여 전달한다는 의미이다. 하지만 기본제공 타입은 이보다 가벼우므로 값을 넘기는 편이 낫다.

###### **그러나 크기가 작다고 해서 무조건 값을 전달하지 않는다.**

사용자 정의 타입은 절대로 값을 전달하지 않는다. 이유는 다음과 같다.  
1. 포인터 멤버가 있는 경우 그가 가리키는 대상까지 복사해야 한다.  
2. 컴파일러가 레지스터에 넣어주지 않는 경우가 많다. 하지만 포인터는 확실히 레지스터에 들어간다.

레지스터 변수에 대하여

|  |
| --- |
| 일단 2가지 변수에 대해 기억하자.  **register 변수**  레지스터 변수의 수가 제한되기 때문에 운이 좋으면 레지스터 변수는 CPU 레지스터 또는 더 일반적으로 가장 빠른 메모리에 저장되는데, **더 빠르게 접근하고 계산될 수 있다.** 이외의 부분들은 자동변수와 똑같다. 컴파일러는 레지스터 수와 선언된 변수의 수를 가중치에 따라 조정하기 때문에 레지스터 변수로 선언했지만 자동 변수로 되는 경우도 있을 수 있다. 그리고 레지스터 변수로 선언될 수 있는 형도 제한이 있다. 또한 & 연산자도 레지스터 변수에는 적용할 수 없다. 왜냐하면 &는 메모리를 참조하는 연산자이기 때문이다. 결국 아래와 같이 정의된다.  **register** int pvar = \*p;  이 pvar은 메모리가 아닌 레지스터에 존재하게 된다.  **volatile 변수**  최적화를 수행하지 않는다. 변수를 register가 아닌 메모리에 저장한다. 아무리 많은 register 변수를 선언하더라도 CPU의 register 개수에는 한계가 있기 때문에 register 선언이 효과를 지니지 않고 메모리 변수로 되는 경우도 있다.  참고: <https://salkuma.wordpress.com/2010/06/13/register-volatile-keyword/> Question. 도데체 더 느린 volatile은 왜 쓰는가? 가능한 register을 쓰는게 최선인데? **중요한 점은 외부에서 값이 변경될 수 있는 경우 volatile을 사용해야 한다는 것이다.** 왜냐하면 원본이 바뀌었으면 이를 갱신해야 하는데 register은 이것을 할 수 없다. 실제 메모리를 참조해야 실시간 바뀐 값을 알기 때문이다. 자세한 내용은 링크 참고 |

#### STL의 반복자

반복자는 값으로 전달되도록 설계되어 왔다. 반복자에 대해 조금 알아보자.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **반복자란?**  **순회 방법을 일반화하기 위해STL에서 사용하는 개념이 바로 반복자이다**. 반복자는 다음과 같은 기능을 가지는데 이는 포인터를 흉내내면서 모든 요소를 순서대로 가리킬 수 있다. 종류에 따라서는 일부 기능이 제외되기도 한다.  **① 컨테이너의 요소 하나를 가리키는 기본적인 역할을 한다.**  **② 가리키는 지점의 요소를 읽고 쓸 수 있다. 내용을 읽는\* 연산자가 정의된다**.  **③ 증감에 의해 주변 요소로 이동할 수 있다. ++, -- 등의 연산자가 정의된다.**  **④ 반복자끼리 대입, 비교 가능해야 한다. 대입, 비교 연산자가 정의된다**.  **반복자를 사용해보자.**  vector<int> vi;  vector<int>::iterator it;  *//이와 같은 순회는 모든 STL컨테이너가 일관되어 있다.*  **for** (it = vi.begin(); it != vi.end(); it++) {  printf("%d**\n**", \*it);  }  **STL관련 주요 함수(include <algorithm>)**   |  |  | | --- | --- | | **함수** | **설명** | | **count** | 조건에 맞는 요소의 **개수를** 센다. | | **for\_each** | 각 요소에 대해 **지정한 작업을** 한다. | | **equal** | 구간이 **일치하는지** 비교한다. | | **search** | 일치하는 부분 구간을 **검색한다**. | | **copy** | 구간끼리 **복사한다**. | | **fill** | 일정한 값으로 지정 구간을 **채운다**. | | **reverse** | 구간의 요소들을 반대로 **뒤집는다**. | | **random\_shuffle** | 요소들을 **무작위로 섞는다**. | | **swap** | 컨테이너를 **교환한다**. | | **binary\_search** | **이분 검색한다**. | | **merge** | 구간을 **병합하여** 새로운 구간으로 복사한다. | | **accumulate** | 구간의 값을 모두 **합한다**. |   **왜 반복자는 값으로 전달해야 할까?**  아마도 자체가 포인터이기 때문에 값의 전달이 곧 참조라고 볼 수 있겠다. |

#### 함수 객체

|  |
| --- |
| **함수 객체**가 무엇이지?  함수처럼 사용하는 객체를 뜻한다. 이는 operator()를 오버로딩하여 사용할 수 있다.  **함수 객체**를 사용해볼까?  **class** **Equal**{  **public**:  int **operator**()(int a, int b){  **return** a == b;  }  };  int main(){  Equal cmp;  cout << "pls(5, 5): " << cmp(5, 5) << endl;  cout << "cmp.operator()(10, 20): " << cmp.**operator**()(10, 20) << endl;  cout << "Equal()(10, 10): " << Equal()(10, 10) << endl;  **return** 0;  }  **함수 객체**를 왜 쓸까? 1. 속성을 가지고 있어 다양한 동작을 가능하게 한다(누적 값 저장 등). 2. 일반 함수보다 빠르게 작동한다.  **함수 객체**는 값을 전달해야 한다고?  앞서 본 **반복자와 함수 객체**는 값에 의해 전달되도록 설계되어 왔습니다. 그러므로  **1. 복사 효율을 높일 것 2. 복사손실 문제에 노출되지 않도록 만드는 것**  이 2가지를 지키면서 만들어야 합니다. |

***값에 의한 전달보다는 상수 객체 참조자에 의한 전달을 선호합시다. 대체적으로 효율적일 뿐만 아니라 복사손실 문제까지 막아 줍니다.***

***이번 항목에서 다룬 법칙은 기본제공 타입 및 STL 반복자, 그리고 함수 객체 타입에는 맞지 않습니다. 이들에 대해서는 값에 의한 전달이 더 적절합니다.***

[항목21] 객체 반환은 반드시 값으로

[항목20]에서 대부분의 경우에서 참조자를 반환해야 효율적이라고 배웠다. 하지만 함수에서 객체를 반환해야 하는 경우에는 참조자를 반환하려고 들지 말자.

# 모든 것을 참조자로 넘겨버리자!

즉, 어떠한 객체는 매개변수로 사용되기 위해 넘겨질 수도 있으며, 함수 내에서 일을 끝내고 반환할 때 넘겨지는 객체도 있다. “위의 2가지를 포함하여 넘겨지는 모든 것은 **‘참조에 의한 전달’**만이 효율적이겠지?”이런 생각은 **금물**이라는 말이다.

# 반환되는 객체는 생성하여 값으로 전달하자.

#### 반환되는 객체는 참조자로 넘길 수 없다.

참조자란 그 객체의 주소이자 이름이다. 이미 존재하는 것을 가리킬 뿐이지 존재하는 곳이 없다면 정말 좋지 않은 일이 일어난다. 그러므로 우리는 이렇게 해야만 한다.

**class** **Archer**{

};

**class** **Weapon**{

**public**:

Weapon(int d = 0, int a = 1) : damage(d), armor(a) {}

**private**:

int damage;

int armor;

*//****[항목3]:*** *값이 대입되어 객체 값이 변하는 것을 방지하도록 const선언*

*//****프랜드 함수 선언:*** *관련하여 아래에서 다시 설명합니다.*

**friend** **const** Weapon **operator**\*(**const** Weapon& lw, **const** Weapon& rw);

};

**const** Weapon **operator**\*(**const** Weapon& lw, **const** Weapon& rw){

*//damage, armor 둘 다 friend함수 안이기에 접근 가능한 겁니다.*

***//여하튼 이렇게 객체를 만들어서 반환해야만 합니다.***

**return** Weapon(lw.damage\*rw.damage, lw.armor\*rw.armor);

}

int main(){

Weapon w1(5, 1), w2(100, 75), w3;

w3 = w1\*w2;

}

#### friend함수에 대하여 알아봅시다.

|  |
| --- |
| **friend함수**가 무엇인가요? MSDN 정의: friend 함수는 클래스 멤버로 간주되지 않으며, 클래스의 private접근 가능한 일반 외부 함수입니다.  friend는 클래스 범위에 포함되지 않으며, 다른 클래스의 멤버가 아닌 경우 멤버 선택 연산자(. 및 –>)를 사용하여 호출되지 않습니다. friend 함수는 액세스 권한을 부여하는 클래스에서 선언됩니다.  friend 선언은 클래스 선언의 어느 곳에나 배치될 수 있으며, 액세스 제어 키워드의 영향을 받지 않습니다. 이는 extern선언과 비슷한 효과를 가집니다.  **정리: *friend함수*는 비멤버 함수지만 임의의 클래스의 private접근을 가능하게 해주는 함수이다.** **friend함수**는 어떻게 써야 하나요? **class** **Archer**{  ***//Archer은 Weapon을 친구로 인정합니다. 즉, 선언된 것입니다.***  ***//Weapon클래스에서는 Archer의 private접근이 가능합니다.***  **friend** **class** **Weapon**;  void drainAnotherWeapon(**const** Weapon& w);  **private**:  string name;  int power;  };  void Archer::drainAnotherWeapon(**const** **Weapon**& w){  *//아래의 Weapon클래스를 보면 이 함수를 friend로 선언한 모습이 보인다.*  ***//이는 해당 함수에서 선언된 곳(Weapon)의 private접근이 가능하게 해준다.***  power = w.damage + w.armor;  }  **class** **Weapon**{  *//이 선언은 Archer가 Weapon에게 접근하도록 해줍니다.*  **friend** **class** **Archer**;  **public**:  Weapon(int d = 0, int a = 1) : damage(d), armor(a) {}  void setMaster(**const** Archer& a);  **private**:  int damage;  int armor;  *//프랜드 함수 선언: 이 함수는 class Weapon의 함수가 아닙니다.*  ***//다만 Weapon내부에 선언되어 Weapon의 private접근이 가능한 함수가 됩니다.***  **friend** **const** Weapon **operator**\*(**const** Weapon& lw, **const** Weapon& rw);  ***//외부 클래스 멤버 함수를 선언하여 그 함수에서 지금 클래스의 private접근허용***  **friend** void Archer::drainAnotherWeapon(**const** Weapon& w);  };  void Weapon::setMaster(**const** Archer& a){  *//위에서 언급한대로* ***Archer에서 Weapon을 친구로 선언했으므로***  cout << "my master is " << a.name << endl;  }  **const** Weapon **operator**\*(**const** Weapon& lw, **const** Weapon& rw){  ***//damage, armor 둘 다 friend함수 안이기에 접근 가능한 겁니다.***  **return** Weapon(lw.damage\*rw.damage, lw.armor\*rw.armor);  }  int main(){  Weapon w1(5, 1), w2(100, 75), w3;  w3 = w1\*w2;  } |

# 만약 참조자를 반환한다면?

만약 위에 대로 하지 않고, 반환되는 객체를 참조에 의한 전달로 인해 무슨 일이 벌어지는지 살펴보자.

#### 참조자를 넘기려면 객체가 이미 존재해야 한다.

연산을 하고 나면 그의 값을 가지는 새로운 객체를 가질 겁니다. 그러므로 일단 그 새로운 객체를 만들어야겠죠? **이 최소한의 생성 비용을 생성하지 않고 참조로 절약할 수 있을까요?**

###### 스택에 만든다면

객체를 스택에 만들면 생성자 호출한다. 중요한 점은 스택은 함수가 끝나면 사라지므로 이 참조 전달된 이 객체는 쓸 수가 없다.

###### 그럼 힙에 만들어야지

힙에 만들기 위해 여전히 한 번의 생성자를 호출한다. 중요한 것은 delete처리가 곤란하다는 것이다. 반환 값은 어디로든 갈 수 있기 때문이다.

###### **참조자 vs. 포인터**

**공통점:** 객체에 직접 접근하지 않고 주소를 저장하여 간접적으로 참조한다.  
**차이점:** 포인터는 주소다. 어디든지 가리킬 수 있으며 바뀔 수 있다. 그 객체가 있는 곳의 주소를 저장한다. 반면 참조자는 객체의 주소를 가리키지만 바뀔 수 없다. 그리고 반드시 초기화가 되어야 한다.

[약간의 의문] 포인터나 참조자나 이후에 가리키는 참조로 메모리를 해제할 수 있는 것으로 확인되었다. 다만 포인터는 스마트 포인터로 저절로 관리가 되지만 참조자는 사용자가 까먹을 수 있으므로 위험하다는 의미로 받아들인다.

**그리고 스마트 포인터와 관련하여 다음과 같은 일이 발생한다.**

shared\_ptr<Item> createItem(int n){

cout << "create Item: " << n << endl;

shared\_ptr<Item> spI(**new** Item(n));

**return** spI;

}

*//Item의 스마트 포인터를 반환합니다.*

===================================================

int main(){

shared\_ptr<Item> spI(createItem(1));

Item &rI = \*spI;

*//만약 포인터를 저장하지 않고 반환 값을 바로 참조하면*

Item &rI2 = \*createItem(2);

*//객체는 이미 소멸하고 없습니다.*

*//이유는 간단합니다.* ***스마트 포인터는 객체를 포인터로 가리켜야 하고***

***//참조는 그렇지 않습니다. 참조할 뿐입니다.***

*//그러므로 더 이상 참조 카운터가 없으므로 소멸합니다.*

showItem(rI);

**return** 0;

}

###### 그렇다면 전역이나 static객체를 사용해야겠다.

역시 생성자를 호출한다. 게다가 스레드 안전성 문제까지 얽혀있다. 한 객체를 공유해서 쓰므로 서로 다른 객체의 값을 비교하는데 항상 같아지는 괴이한 현상까지 일어난다.

하지만 [항목4]에서도 보았듯이, 반환 객체를 한번만 쓴다는 보장이 있다면 정적 참조로 반환할 수 있다.

###### static을 배열로 만들어서 구분을 지어놓자.

생성을 무지막지하게 하므로 최적화가 아닌 부적화가 된다.

***지역 스택 객체의 참조 전달, 힙 객체의 참조 전달, 지역 정적 객체의 참조자를 반환하는 일은 객체가 2개 이상 생긴다면 절대로 하지 마세요.***

[항목22] 데이터 멤버는 private

데이터 멤버가 private이어야 하는 이유는 무엇일까?

# 데이터 멤버가 public이면 안 되는 이유

#### 문법적 일관성

데이터 멤버에 접근할 수 있는 유일한 수단은 멤버 함수이어야 한다. 데이터로 접근했다 함수로 접근했다 하는 행동은 좋지 못합니다.

#### 정교한 제어

함수의 사용은 데이터로 직접 접근하는 것보다 더 정교한 제어가 가능하다. 함수를 사용함으로 접근 불가, 읽기 전용, 읽기 쓰기 접근 등을 직접 제어할 수 있다.

#### 캡슐화

캡슐화를 위하여 private으로 두어야 한다. 데이터로는 접근을 제한하고 오로지 멤버 함수를 통한다.

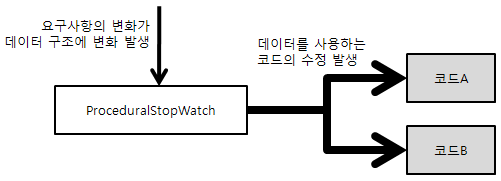
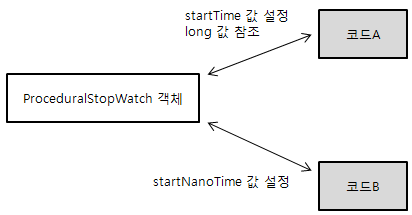
###### 캡슐화(encapsulation)란?

객체의 속성(data fields)과 행위(메서드, methods)를 하나로 묶고*,* 실제 구현 내용 일부를 외부에 감추어 은닉한다. 캡슐화를 왜 해야 할까? 아래를 보고 이해를 돕는다.

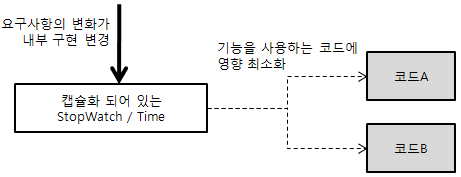
###### 캡슐화의 사용

기본적인 객체지향 내용은 [항목1]에서 다루었다.

데이터는 자기 자신의 멤버 함수 외에는 절대로 건드리지 않는다. 아래는 비캡슐화이다.



다음은 캡슐화된 모습(객체지향)이다.



위와 같은 상황을 코드로 살펴본다. **Archer은 캡슐화**이며 **CrazyArcher은 비캡슐화**이다.

**class** **Archer**{

**public**:

Archer() : hp(100), power(1) {}

void levelUp(){ setPower(5); }

void setPower(int delta){ power += delta; }

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

void attack(T&& target){ target.hit(power); }

void hit(int delta){ hp -= delta; }

**private**:

int hp;

int power;

};

**struct** **CrazyArcher**{

CrazyArcher() : hp(100), power(1) {}

int hp;

int power;

};

int main(){

Archer a1;

CrazyArcher a2;

*//무기장착*

int masterPlan = 1000;

a1.setPower(masterPlan);

a2.power += masterPlan;

*//레벨 업*

a1.levelUp();

a2.power += 5;

*//아프다*

a1.hit(99);

a2.hp -= 99;

**return** 0;

}

아무리봐도 똑같다. 캡슐화를 도대체 왜 쓰는거지? 아무짝에 쓸모없다는 생각이 들만한 코드다. 오히려 비캡슐화가 코드도 간단하고 편해보인다.

이제 유지보수에 들어간다.

새로운 데이터가 추가되어 power와 함께 spell이 증가하도록 변경한다.

*//캡슐화가 된 경우*

void Archer::setPower(int delta){

power += delta;

**if** (power > 10) spell += 1;

}

이것만 바꿔주면 된다.

*//비캡슐화의 경우*

a2.power += masterPlan;

**if** (power > 10) spell += 1;

a2.power += 5;

**if** (power > 10) spell += 1;

*//...*

코드에 쓰인 모든 power을 다 찾아서 갱신해야 할 것이다. 만약 power자체가 바뀐다면 이건 그것을 사용하고 있는 외부 대상까지 수정해야 하니 답도 없다. 규모가 조금만 커지면 인생의 패배자가 될 것이다.

# 데이터 멤버가 protected이면 안 되는 이유

이 내용은 public과 같다. protected라고 해서 public보다 나은 점은 없다.

#### 일관성 + 정교성 + 캡슐화

public보다는 많이 가려져 있겠지? 하지만 어쨌든 의존도는 존재하기 때문에 데이터를 변경할 시 많은 부분이 망가질 수 있는 점은 동일하므로 public가 사실상 같다.

***데이터 멤버는 private멤버로 선언합시다. 이를 통해 클래스 제작자는 문법적으로 일관성 있는 데이터 접근 통로를 제공할 수 있고, 필요에 따라서는 세밀한 접근 제어도 가능합니다. 클래스의 불변속성 강화와 내부 구현의 융통성도 발휘합니다.***

***protected는 public보다 더 많이 ‘보호’받고 있는 것이 아닙니다.***

[항목23] 비멤버 비프렌드 함수 사용

멤버 함수보다 비멤버 비프렌드 함수와 더 가까워지자.

# 멤버 함수 vs. 비멤버 함수

각각이 갖는 의미란 무엇일까?

#### 멤버 함수

클래스 안의 함수: 객체가 가지는 기능을 말한다. ‘나는 이러한 기능을 가진다.’

#### 비멤버 함수

전역 함수: 두루두루 어디에서나 사용할 수 있는 편의용 함수다.

# 캡슐화를 시키려면 무조건 멤버 함수를 사용해야 하는군.

그렇지 않다.

#### 캡슐화

앞에서 언급했듯이, ***캡슐화****란 데이터를 감춰놓는 것이다.* 감춰놓으면 그것들을 변경할 수 있는 여유도 많아진다. 왜냐하면 *바꿔도 영향을 주는 정도가 최소화*되기 때문이다.

###### 데이터를 감추려면 어떻게 해야 할까?

역시 데이터에 직접 접근하는 코드가 적어야 한다. 그래서 [항목22]에서처럼 데이터를 private선언하는 이유다. 객체 내부에서야 해당 데이터에 접근할 수 있는 개수가 예측된다지만 만약 객체 밖에서도 접근이 된다면 어마어마하게 개수가 많아지면서 어디에서 접근하는지 예측하기 어렵다.

#### 캡슐화와 멤버 함수

결국 객체 내에서만 데이터에 접근해야 영향을 예측할 수 있다. 그것이 멤버 함수인 것이다. 멤버 함수는 그 객체의 데이터에 접근 가능하고 **곧 멤버 함수가 많아지면 데이터에 접근하는 영향도가 커져 캡슐화가 작아진다고 본다.**

#### 멤버 함수 vs. 비멤버 비프렌드 함수

둘 중 어느 것을 골라야 할까? 위에서 언급한대로 **멤버 함수**는 데이터에 접근한다. 하지만 **비멤버 함수**는? 그저 객체 내의 공개된 기능인 함수를 쓸 뿐이다. 데이터에 접근할 수 없다. **그러므로 비멤버 비프렌드 함수를 사용하면 같은 기능을 사용하면서 *데이터에 영향을 주지 않는 캡슐화가 더 높다***고 볼 수 있는 것이다.

###### 비프렌드 함수여야 하는 이유는?

당연하다. 위에서 데이터에 영향을 주지 않는 것이 **캡슐화**의 포인트다. 그런데 프렌드 함수는 private데이터에 접근하므로 의미가 없는 것이다.

#### 결국, 중요한 것은 비멤버 함수가 아니라, 데이터에 영향을 주면 안 된다는 점이다.

**class** **Archer**{

**public**:

void move();

void attack();

void casting\_skill1();

*//멤버 함수로 여러 함수를 하나로 묶었다.*

void bash\_pattern(){

move();

attack();

casting\_skill1();

}

};

*//비멤버 함수로 여러 함수를 하나로 묶었다.*

*//여기서는 a의 private으로 접근할 수 없다는 보장이 있다.*

void bash\_pattern(Archer& a){

a.move();

a.attack();

a.casting\_skill1();

}

# 위와 같은 사항을 고려하며 함수를 만들어볼까?

위에서 비멤버 함수는 캡슐화에 영향을 주지 않음을 보였다. C++에서는 이를 더 효율적으로 사용할 수 있는 방법이 존재한다.

#### namespace

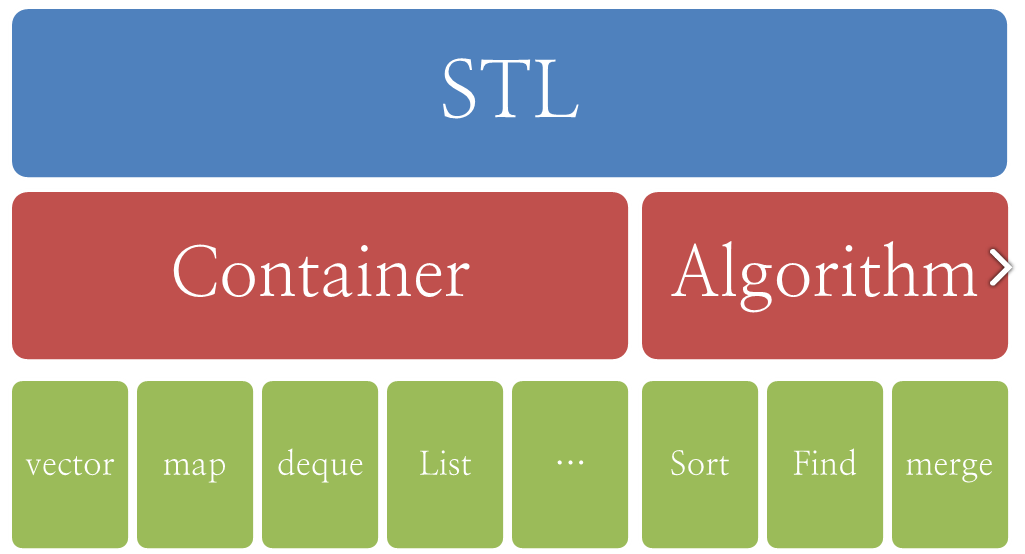
**네임스페이스**는 변수 이름이나 함수 이름과 같이 명칭을 사용하는 공간으로 소속을 나타낸다. using namespace 선언은 말 그대로 소속을 알리는 역할을 한다. 그러니까

***[namespace]::[변수 혹은 함수]***

의 꼴로 사용된다. **네임스페이스**라는 **소속 공간에 따라서 변수나 함수가 같은 이름임에도 다른 식으로 구분이 될 수 있다**.

#### namespace만의 특징

namespace에서 중요한 특징은 여러 개의 소스 파일에 나뉠 수 있다는 것이다. 이러한 특징이 잘 드러난 곳이 ***<C++ Standard Library: STL>***이다.

커다란 STL이란 namespace안에 <vector>, <algorithm>, <list>등 여러 가지 헤더파일이 들어있다. STL안에 필요한 부분만 헤더 파일로 포함하여 사용할 수 있다. 물론, 헤더 파일 안에 STL이란 namespace안에 포함되어 있는 것만 사용할 수 있다.

#### namespace를 사용하여 캡슐화하자.

멤버 함수는 객체가 가져야 하는 기능이 필요할 때만 사용한다. 이외에 사용자가 편의를 위해서 연산이라든지 정리된 함수 등의 관리는 비멤버 함수에서 이루어져야 한다. 그리고 그 비멤버 함수를 namespace에 담는다. 그리고 비멤버 함수를 특정 주제로 구분 지어 헤더파일로 나눈 다음에 필요한 기능만 헤더파일을 추가하여 부분적으로 사용하고 관리할 수 있다. 아래 예제를 보자.

*//item.h: 아이템 객체와 핵심을 모아놓았어요.*

**namespace** item{

*//namespace와 class이름이 같으면 못 써요.*

**class** **Item**{

**public**:

Item(int n) : item\_num(n) {};

**virtual** ~Item();

*//뒤에 const를 붙이지 않으면 상수용으로 쓸 수 없다.*

*//참조 전달에서 사용하려면 const가 있어야 한단 말.*

int getNumber() **const**;

**private**:

int item\_num;

};

**class** **Weapon** :**public** Item;

*//비멤버 함수입니다.*

shared\_ptr<Item> createItem(int n){

cout << "create Item: " << n << endl;

shared\_ptr<Item> spI(**new** Item(n));

**return** spI;

}

void showItem(**const** Item& i);

}

*//item\_shop.h: 아이템 상점과 연관되어 있죠.*

*//다른 헤더파일에서는 class::Item를 사용할 수 없구나.*

#include *"Item.h"*

**namespace** item{

*//여기서 인자를 복사하게 되면 하나를 더 참조하고*

*//그렇지 않으면 포인터 그 자체다. 이로써 객체를 삭제할 수 있다.*

void sellItem(shared\_ptr<item::Item>& i){

cout << "sell Item: " << i->getNumber() << endl;

i = **nullptr**;

}

}

*//=============================================================================*

#pragma once

#include *<iostream>*

#include *"Item.h"*

#include *"Item\_shop.h"*

**using** **namespace** std;

int main(){

shared\_ptr<item::Item> spI(item::createItem(1));

showItem(\*spI);

cout << "===shop===" << endl;

*//포인터를 넘겨서 객체를 삭제해요.*

item::sellItem(spI);

*//객체를 팔아버렸죠.*

***//이렇게 namespace안에서 필요한 함수들만 모여있는 헤더파일만 추가해요.***

**return** 0;

}

***멤버 함수보다는 비멤버 비프렌드 함수를 자주 쓰도록 합시다. 캡슐화 정도도 높아지고 패키징 유연성도 커지며, 기능적인 확장성도 늘어납니다.***

[항목24] 비멤버 함수와 암시적 변환

타입 변환이 모든 매개변수에 대해 적용되어야 한다면 비멤버 함수를 사용해야 한다.

# 명시적 변환 vs. 암시적 변환

일반적으로는 **명시적 변환**(***explicit***)을 사용해야 한다. 왜냐하면 의도치 않은 자동 형 변환을 방지하기 위해서다.

#### [예외] 숫자 타입을 만들 때

계산할 때마다 int, long long, double등 숫자를 캐스팅해야 한다면 꽤 불편할 것이다.

# 수치 연산자 만들기

어떠한 함수로 만들어야 할까? 고민이 된다.

#### 멤버 함수

연산자는 객체의 기능이라기 보다 사용자가 객체를 다루는 방법이라서 멤버 함수는 적절치 않지만, 여하튼 다음과 같이 만들 수 있다.

**class** **Rational**{

**public**:

Rational(int v) : value(v) {};

**//멤버 연산 함수**

**const** Rational **operator**\*(**const** Rational& rhs) **const**;

**private**:

int value;

};

**const** Rational Rational::**operator**\*(**const** Rational& rhs) **const**{

**return** Rational(rhs.value \* value);

}

int main(){

Rational r1(5), r2(10), r3(1);

*//아래의 식은 잘 됩니다. 3개의 문장은 동일합니다.*

*//2가 Rational타입으로 자동 형 변환이 일어납니다. 3번째처럼요.*

r3 = r1 \* 2;

r3 = r1.**operator**\*(2);

r3 = r1.**operator**\*(Rational(2));

*//아래의 식은 오류입니다.*

*//2는 함수를 가지고 있지 않아요.*

r3 = 2 \* r1;

r3 = 2.**operator**\*(r1);

}

왜 뒤에 있는 인자만 형 변환이 일어날까? **자동 형 변환은 매개변수 리스트에 들어있어야만 한다.** 그러므로 해결책은 간단하다. 둘 다 매개변수로 집어넣는다.

#### 비멤버 함수

연산자는 비멤버 함수로 만들기 적절하다. 게다가 위에서처럼 두 개의 객체를 매개변수로 넣어야만 자동 형 변환이 생김을 알았다. 다음과 같다.

**class** **Rational**{

**public**:

Rational(int v) : value(v) {};

int getValue() **const** { **return** value; };

**private**:

int value;

};

**const** Rational **operator**\*(**const** Rational &lhs, **const** Rational &rhs){

**return** Rational(lhs.getValue() \* rhs.getValue());

}

int main(){

Rational r1(5), r2(10), r3(1);

*//비멤버 함수로 둘 다 가능하군요.*

r3 = r1 \* 2;

r3 = 2 \* r1;

}

**그러므로 매개변수에 암시적 변환이 필요한 부분(특히 숫자 연산)에서는 비멤버 함수가 아주 적합하다.**

###### 프렌드 함수에 관하여

위에서도 살펴보았지만 역시 최대한 안 쓰는 편이 좋다. 위에서는 private에 접근할 일이 없기에 쓰지 않았고 써서 간단하게 연산할 수 있지만 안 쓸 수 있으면 캡슐화에 더욱 좋다.

***어떤 함수에 들어가는 모든 매개변수(this 포인터가 가리키는 객체도 포함해서)에 대해 타입 변환을 해 줄 필요가 있다면, 그 함수는 비멤버이어야 합니다.***

[항목25] 표준 함수의 특수화

예외를 던지지 않는 std::swap()에 대한 지원도 생각해보자.

# 표준 swap

**namespace** std{

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

void swap(T& a, T& b){

T temp(a);

a = b;

b = temp;

}

}

아주 심플하다! 그런데 복사가 3번이나 일어나네. **이러한 동작이 너무 비효율적이면?**

# 특수화 swap

이는 특정 클래스(위와 같이 손해를 많이 볼 클래스면 좋겠다.)에 한하여 그에 적합하게 사용할 swap을 만드는 것이다.

#### 템플릿 특수화에 관하여

###### 템플릿 특수화가 뭐죠?

특정 객체에 한하여 다른 동작을 하게 만든다. 아래 예제를 보자.

1. **template**<**typename** T>
2. **class** CFoo
3. {
4. **public**:
5. T Add(T lhs, T rhs)
6. {
7. **return** lhs + rhs;
8. }
9. };
11. // char\* 타입용 특수화 함수
12. **template**<>
13. **char**\* CFoo<**char**\*>::Add(**char**\* lhs, **char**\* rhs)
14. {
15. **char** \*sz = **new** **char**[128];
16. memset(sz, 0, **sizeof**(sz));
17. strcat(sz, lhs);
18. strcat(sz, rhs);
19. **return** sz;
20. }

###### 완전 템플릿 특수화 vs. 부분 템플릿 특수화

위에서 어떤 함수를 특수화시킨 모습을 보고 있다. 그런데 함수 전체에서 특수화하고자 하는 타입이 적용된다. 이것이 ***‘함수에 완전 템플릿 특수화하였다’***라고 한다.

**부분 템플릿 특수화**는? 여러 개의 템플릿의 값이 있다면 이들 중 일부분만 일치하는 경우에 대하여 정의한다. 아래를 보자.

**template** <**class** **Key**, **class** **Value**> **class** **Dictionary** {

Key\* keys;

Value\* values;

int size;

int max\_size;

**public**:

Dictionary(int initial\_size) : size(0) { }

void add(Key key, Value value);

void print();

};

*// Template partial specialization: Key is specified to be int.*

*// dictionary가 <int, value>인 경우에 한하여*

**template** <**class** **Value**> **class** **Dictionary**<int, Value> {

int\* keys;

Value\* values;

int size;

int max\_size;

**public**:

Dictionary(int initial\_size) : size(0) { }

void add(Key key, Value value);

void print();

};

부분 일치하는 것은 위에서처럼 2개의 값 중 한 개라든지, 포인터나 값으로서 객체의 템플릿도 마찬가지다.

이렇게 특수화는 클래스와 함수에 2가지로 적용할 수 있으며 **클래스**에는 완전, 부분적으로 가능하지만 **함수**에는 완전으로만 특수화할 수 있다.

#### 어떻게 특수화시키지?

멍할 것이다. Archer타입에 대해서만 특별하게 동작하는 swap을 만들 수 있다는 것을 알았다. 하지만 어떻게 만드는가? 여기서 이에 적합한 관용구를 소개한다.

###### **Pimpl Idiom**

|  |
| --- |
| **Pimpl(Pointer to Implementation) Idiom**이란 include가 필요한 유저 define 타입의 멤버변수들을 해당 멤버변수들을 포함한 구조체의 포인터로 대체하는 방법이다. 즉, 어떤 객체에 구조체의 포인터를 넣고 그 구조체에 여러 데이터를 담아 구분하는 방식이 되겠다. 아래와 같은 방식이다.  //in header Widget.h  class Widget{  public:  Widget();  ...  private:  struct Impl;  std::unique\_ptr<Impl> pImpl //unique\_ptr로 대체  };  //in implementation Widget.cpp  #include "Widget.h"  #include "Gadget.h"  #include <string>  #include <vector>  struct Widget::Impl{  std::string name;  std::vector<double> data;  Gadget g1, g2, g3;  };  Widget::Widget() //초기화 리스트에서 unique\_ptr생성  :pImpl(std::make\_unique<Impl>())  {}  위에서는 unique\_ptr로 객체 관리하는 C++11에서 추가된 기능까지 적용시켜 놓은 모습인데 참고 바란다.  **이러한 패턴은 왜 쓰는 걸까?** include가 필요한 멤버변수들을 모아 Impl 구조체에 저장하고 그 포인터를 멤버변수로 들고 있는다. Impl구조체의 정의는 구현파일(cpp)로 미룬다. 그러면 해당 멤버변수에 필요한 #include를 cpp에서 해주면 되기 때문에, 헤더파일에는 어떤 #include도 필요가 없다. **cpp파일의 #include는 오직 해당 cpp파일에만 영향을 미치기에 컴파일 시간을 최적화** 할 수 있다.  또한 아래에서도 보겠지만 여기서 이 패턴을 설명하는 이유는 **모든 데이터를 묶어놓으면 이동하기에 가볍기 때문이다.** 복사시에는 모두 복사해야되지만 이동할때에는 포인터만 바꾸면 되므로 효율적이다. |

###### **Pimpl관용구를 적용하면 swap을 최적화시킬 수 있다.**

위에서 살펴본 Archer의 구조를 Pimpl관용구를 이용하여 바꿀 것이다. 정리하면 Archer은 상당히 많은 내용을 포인터로 가리키고 있다. 포인터로 되어있으므로 포인터만 바꾸면 간단한데 안의 모든 데이터를 복사하려면 상당한 손해이다. **그러므로 이러한 클래스에 대해서만 사용할 특별한 swap()을 만들어 주면 아주 적절하다.**

#### 특수화된 swap의 구현

이는 std::swap에 내가 원하는 클래스에만 Pimpl관용구에 적합한 버전으로 특수화시킨 모습이다.

**namespace** std{

*//이 함수는 비멤버 함수입니다.*

*//std를 개조하여 타입 Archer에 대하여 swap을 완전 특수화시킨 모습*

**template**<>

void swap<Archer>(Archer& a, Archer& b){

*//비멤버 함수에서는 private에 접근할 수 없습니다.*

swap(a.pd, b.pd);

}

}

안 된다. 왜 그런가? 위의 주석을 보라. 비멤버 함수에서 어떻게 private을 접근하겠는가?

우리는 2가지 선택을 할 수 있다.  
**1. friend함수 만들기**: 이는 비멤버처럼 작동하지만 private에 접근할 수 있다. 하지만 이는 표준 템플릿이 쓰인 모습과 다르므로 적합하지 않다.  
**2. 멤버 함수로 구현한다**. 그리고 다시 위에서처럼 namespace std안에서 swap특수화버전에서 멤버 함수를 호출하면 그만이다. 이를 아래에서 살펴본다.

# 멤버 접근을 해결하는 멤버 swap

위에서 Pimpl관용구를 적용하여 데이터를 가볍게 옮기는 최적화를 적용하여 멤버 swap을 구현할 것이다.

#### 멤버 함수 swap의 구현

*//물론 pimpl 관용구라면 이 클래스는 다른 cpp파일에 있어야 합니다.*

**class** **PlayerData**{

**public**:

**private**:

*//이 엄청난 데이터를 복사한다면 큰일납니다.*

vector<string> data;

};

**class** **Archer**{

**public**:

void swap(Archer &other);

**private**:

PlayerData \*pd;

};

*//멤버 함수 swap입니다.*

void Archer::swap(Archer &other){

***//이 문장은 이제부터 std의 swap을 쓰겠다는 말입니다.***

**using** std::swap;

***//포인터만 맞바꿉니다. 데이터는 private이므로 반드시 멤버 함수이어야 합니다.***

swap(pd, other.pd);

}

**이것을 만든 이유는 위에 주석으로 써져있듯이, 데이터가 private이기 때문에 다른 함수에서 이를 불러올 것이다.**

#### 멤버 함수 swap -> 특수화된 std::swap

위에서 만든 **멤버 함수 swap를 namespace std안에서 특수화된 swap을 하나 더 구현하여 그 안에서 이를 호출한다.** 이렇게 비멤버처럼 만들어주어야 표준처럼 작동하기 적합할 것이다. 그리고 클래스 내부의 private에도 접근할 수 있다. 멤버 함수를 통하기 때문이다.

*//멤버 함수 swap입니다.*

void Archer::swap(Archer &other){

*//이 문장은 이제부터 std의 swap을 쓰겠다는 말입니다.*

**using** std::swap;

*//포인터만 맞바꿉니다.*

swap(pd, other.pd);

}

**namespace** std{

***//std의 타입 Archer에 대하여 swap을 완전 특수화시킨 모습***

**template**<>

void swap<Archer>(Archer& a, Archer& b){

a.swap(b);

}

}

***이로써 임의의 클래스에 대하여 아주 잘 작동하는 특수화된 swap을 만들었습니다. 이는 std함수를 특수화시킨 것이지요.***

#### Question. 하지만 Archer가 템플릿으로 되어있다면 어떻게 되는 거죠?

*//모든 클래스가 템플릿화 됩니다.*

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

**class** **PlayerData**{

**public**:

**private**:

vector<string> data;

};

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

**class** **Archer**{

**public**:

void swap(Archer &other);

**private**:

PlayerData<T> \*pd;

};

*//멤버 함수 swap입니다.*

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

void Archer<T>::swap(Archer<T> &other){

**using** std::swap;

swap(pd, other.pd);

}

**namespace** std{

*//템플릿화된 타입 Archer에 대하여 swap을 부분 특수화시킨 모습*

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

void swap<Archer<T> >(Archer<T>& a, Archer<T>& b){

a.swap(b);

}

}

이렇게 함수를 부분 특수화시키면 컴파일이 안 된다. 클래스 템플릿의 부분 특수화는 허용하지만 **함수 템플릿의 부분 특수화는 허용하지 않는다.** 여하튼 안 된다.

###### 이게 왜 부분 특수화지?

잘 보면 swap은 T타입을 받게되어 있다. 그런데 여기서는 많은 T타입 중 Archer<T>에 대해서만 특수화를 요구하고 있다. T는 BladeMaster, Item등 다양한 객체가 될 수 있지만 Archer<T>에 대해서만 요구한다는 의미다.

# 템플릿 버전을 해결하는 비멤버 swap

이를 해결할 수 있는 것이 **오버로드 비멤버 swap**이다. 다음과 같이 만든다.

#### std::swap을 오버로드하기

또 하나의 핵심 원리를 기억해야 한다. **특정 타입의 매개변수에만 작동하게 하는 다형성에 관한 도구로 오버로드를 생각할 수 있다는 것이다!** Archer에 대해서의 오버로드 버전을 생성하면 특수화가 아니므로 템플릿을 적용할 수 있다. 템플릿을 조작하려 하지 말고 그냥 오버로드하여 특정 객체에 반응하도록 추가해버리자!

**namespace** std{

*//특수화랑 다른 점은 swap위에 <Archer<T> >가 없음으로써 오버로드됩니다.*

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

void swap(Archer<T>& a, Archer<T>& b){

a.swap(b);

}

}

이는 완벽하게 실행되지만 **미정의동작**이다. 다름이 아니라 std가 우리를 엿먹이려 설계한 모습이라 알려졌다. **std함수에 새로운 템플릿을 추가하는 것은 안 된다**는 std의 규칙이 있다. 이제 어떻게 해야 할까?

#### std밖에서 비멤버 swap함수 만들기

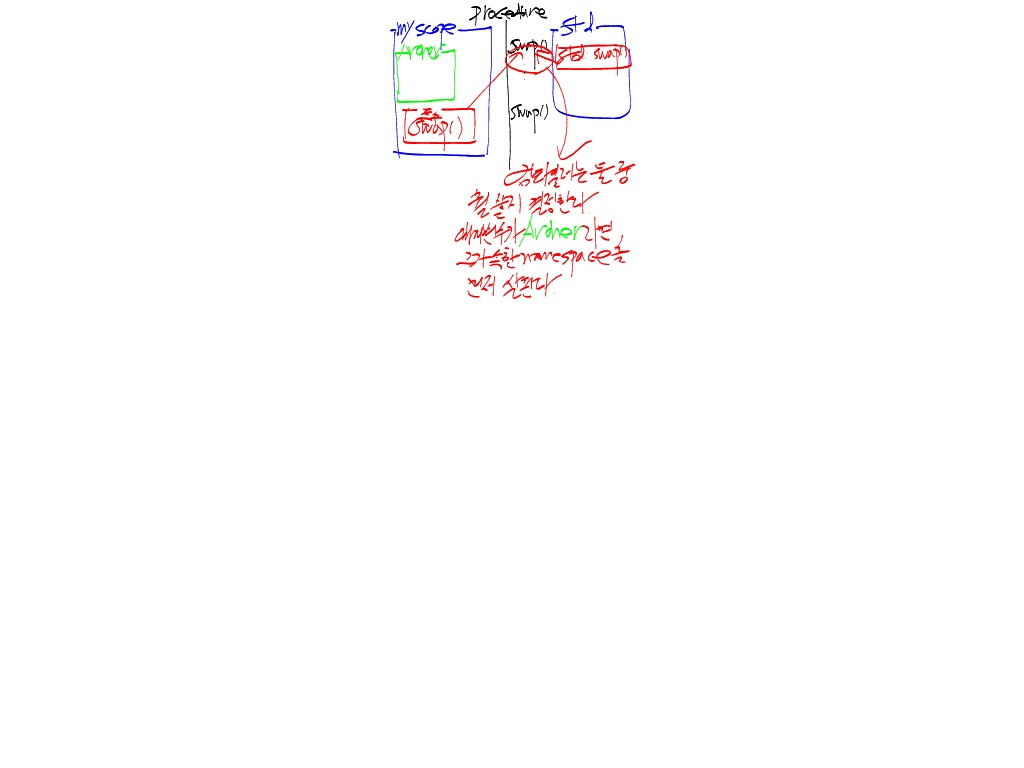
~~아마 std안에서 쓰이는 함수의 템플릿을 만들면 범위넓게 쓰이는 std이므로 오류가능성이 높다고 판단하지 않았을까?~~

###### std안에서의 swap의 의미

여하튼 오버로드에 템플릿을 적용시킴은 변함이 없다. 그런데 왜 우리는 std안에 swap을 만드려고 했을까? 내가 진행하는 프로그래밍 범위에서 이 특수화 버전만 쓸 것은 아니지 않은가? std::swap도 써야하고 이 특수화 버전도 써야한다. 그래서 오버로드가 컴파일 타임에 다양한 매개변수에 대해 적합한 타입을 추론해내는 원리를 우리는 사용하려고 한 것이다.

###### 사용자정의 namespace를 이용한 오버로드 swap

우리가 원하는 것은 Archer<T>는 이 특수화 버전 swap을 사용하게 하는 일이다. 이를 가능하게 해주는 일이 있다. 사용자정의 namespace로 Archer와 그의 swap을 묶어주는 것이다. 사용 범위를 만들어주는 것이다. 그림으로 이해해보자.



쾨니그 탐색(ADL)

위에서처럼 이름찾는 규칙을 쾨니그 탐색이라 한다. 어떤 함수에 어떤 타입의 인자가 있으면, 그 함수의 이름을 찾기 위해 해당 타입의 인자가 위치한 네임스페이스 내부의 이름을 먼저 탐색한다는 간단한 규칙이다.

###### namespace + 오버로드 + 비멤버 swap구현

**namespace** game

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

**class** **PlayerData**{

**public**:

**private**:

vector<string> data;

};

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

**class** **Archer**{

**public**:

void swap(Archer &other);

**private**:

PlayerData<T> \*pd;

};

*//멤버 함수 swap입니다.*

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

void Archer<T>::swap(Archer<T> &other){

**using** std::swap;

swap(pd, other.pd);

}

*//std에 없습니다.*

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

void swap(Archer<T>& a, Archer<T>& b){

a.swap(b);

}

}

***이로써 특수한 템플릿 클래스에 대해서도 아주 잘 작동하는 swap을 만들었습니다.***

# using으로 원하는 swap사용하기

상황에 따라 필요한 swap이 다를 것이다. 이것을 고르는 방법이 using이다.

#### 매개변수가 속한 namespace에 swap이 없으면?

위에서 쾨니그 탐색에 의해 매개변수가 속한 namespace내에서 함수를 찾는다. 하지만 이 범위내에 해당 함수가 없다면 어떻게 될까? 모든 네임스페이스를 뒤질 일은 없다는 것은 잘 알고 있을 것이다. 본인이 속한 namespace와 전역에 없다면, swap을 호출할 수 없다.

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

void power\_change(T& a, T& b)

이런 경우는 상당히 난감하다. 어떤 타입이 올지 모르며, T타입의 swap이 없다면 std:: swap을 호출하고 싶은데 방법이 없을까?

#### using선언으로 융통성있는 swap호출

**using**은 타입 T전용 버전이 있으면 그것이 호출되고, 없다면 다른 곳(using선언)의 버전이 호출될 수 있다.

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

void power\_change(T& a, T& b){

*//T가 속한 네임스페이스 혹은 전역에 swap이 없을 시,*

*//using 선언된 swap을 찾을 것이다.*

**using** std::swap;

swap(a, b);

*//다음 강제적인 선언이다. 해서는 안 된다.*

*//std::swap(a, b);*

}

###### 컴파일러의 이름찾기 규칙

컴파일러가 *‘어떤 swap함수를 쓸까?’*라고 생각할 때,   
1. **타입과 동일한 네임스페이스, 전역 공간에서 사용 가능한 swap**이 있는지 찾습니다.   
2. 전용 swap이 없으면 **using선언된 함수**를 검토합니다(위에서 std).  
이는 번호처럼 1번이 먼저 고려되며, 2번은 그 다음이다.

# 표준과 특수화에 대한 정리

위는 물론 swap에만 한정된 이야기가 아니다. **핵심은 보편적이면서 동시에 내가 원하는 부분에만의 특수화 진행이다.** 그 대상이 표준일 경우는 더 까다로울 뿐이다.  
1. 표준 혹은 정해져 있는 함수가 충분하다면 만족해라.  
2. 그렇지 않다면 그 타입에 대한 특수화가 진행되어야 한다. **기본 원리는 사용자 정의 함수가 표준 함수를 호출하게 한다.**3. 클래스의 **private멤버를 사용**해야 한다면, **멤버 함수를 구현**해야 할 것이다. 그리고 비멤버 함수에서 멤버 함수를 호출하도록 설계한다.  
4. 이제 비멤버 함수를 구현한다. 필요하다면 3에서의 멤버 함수를 호출한다. 여기서 특수화를 원하는 T타입에 대해서, 같은 범위로 묶어주어야 한다. 그러므로 이 **특수화 비멤버 함수와 T타입을 같은 namespace에 묶어준다.**  
5. **using std::swap**으로 빠짐없는 swap호출을 대비해야 한다.  
6. swap은 절대로 예외를 던져서는 안 된다. 왜냐하면 swap으로 임의의 클래스를 강력한 예외 안정성 보장을 제공하도록 할 수 있기 때문이다. 이를 제공하려면 멤버 버전의 swap에서 예외를 절대로 던져서는 안 된다. 비멤버 버전은 복사와 대입을 기반으로 하므로 예외 발생이 허용되지만, 멤버 함수는 더 신경써야 한다.

간단한 팁은 기본제공타입을 이용하는 것이다. 기본제공타입은 절대로 예외를 던지지 않으며, 대부분 효율적이다.

자세한 내용은 [항목29]에서 다룰 것이다.

***std::swap이 여러분의 타입에 대해 느리게 동작할 여지가 있다면 swap 멤버 함수를 제공합시다. 이 멤버 swap은 예외를 던지지 않도록 만듭니다.***

***멤버 swap을 제공했으면, 이 멤버를 호출하는 비멤버 swap도 제공합니다. 클래스(템플릿이 아닌)에 대해서 작동하는 특수화된 std::swap도 제공합시다.***

***사용자 입장에서 swap을 호출할 때에는, std::swap에 대한 using선언을 넣어 준 후에 네임 스페이스 한정 없이 swap을 호출합시다.***

***사용자 정의 타입에 대한 std템플릿을 완전 특수화하는 것은 가능합니다. 그러나 std에 어느 것이라도 새로 ‘추가’하려고 들지는 마세요.***

[항목26] 변수 정의를 늦추자

# 변수 정의

임의의 변수에 대해 구체적인 사항을 컴파일러에게 알려준다. 그리고 이것은 다음과 같은 비용을 물게 된다.

생성자 호출 🡪 (유효 범위) 🡪 소멸자 호출

# 변수 정의는 최대한 늦게 한다.

핵심은 필요없는 생성자를 호출하지 않는다. 이에 접근하는 설계를 해야 한다.

###### 예외가 생긴다면, 사용하지도 않았는데 생성과 소멸에 대한 비용을 지불해야 한다.

###### 대입보다는 정의와 동시에 초기화하면서 값을 주는 것이 효율적이다.

###### 루프에서는 생성자와 대입의 비용을 비교하여 효율적인 방법을 고른다.

int main(){

*//이 방법은 변수가 밖에 있으므로 이해도와 유지보수에서 불리하다.*

*//생성자 소멸자1번 호출에 대입을 100000번 호출한다.*

*//대입이 가벼우면서 성능이 중요하면 이를 선택한다.*

Archer a;

**for** (int i = 0; i < 100000; i++){

a = i;

}

*//생성자와 소멸자를 100000번 호출한다.*

*//특별히 성능과 연관이 없다면 보편적으로 이를 선택한다.*

**for** (int i = 0; i < 100000; i++){

Archer a(i);

}

}

***변수 정의는 늦출 수 있을 때까지 늦춥시다. 프로그램이 더 깔끔해지며 효율도 좋아집니다.***

[항목27] C++ 캐스팅

캐스팅은 절약하고 또 절약해야 한다.

# C++: 어떤 일이 있어도 타입 에러가 생기지 않도록 보장한다.

이를 가볍게 배대뒤치기할 녀석이 캐스트입니다. 왜 캐스트는 위험할까요? 캐스트에 대해서 알아볼까요?

# C스타일 캐스트: 구질구질 구형 캐스트

*: T타입으로 캐스팅합니다.*

*: T타입으로 캐스팅합니다. 함수 호출문 같습니다.*

# C++스타일 캐스트: 신형 캐스트

**const\_cast<T>:** 객체의 상수성을 없애줍니다.

**dynamic\_cast<T>:** 안전한 다운캐스팅(어떤 객체가 상속 계통에 속한 특정 타입인자 아닌지를 결정할 때)에 쓰입니다.

**reinterpret\_cast<T>:** 포인터를 int로 바꾸는 등의 하부 수준의 캐스팅에 쓰입니다. C스타일의 캐스팅과 가장 유사합니다.

**static\_cast<T>:** 암시적 변환을 강제로 진행할 때 사용합니다.

구질구질한 구형 캐스트보다는 신형 캐스트를 선호합니다. 이유는 아래와 같습니다.  
1. 타입 시스템이 어디서 망가졌는지 찾아보기 쉽습니다.  
2. 컴파일러쪽에서 사용 에러를 진단할 수 있습니다.

# 캐스팅을 왜 줄여야 하나요?

가장 큰 핵심은 성능 저하와 미정의 동작이다. 아래에서 캐스팅이 우리를 어떻게 괴롭히는지 살펴본다.

#### 타입 변환은 런타임에 실행된다.

###### int 🡪 double변환

**타입 변환보다 새로운 double타입 변수에 초기화**하는 편이 더 낫다.

###### 기본 클래스 포인터 🡪 파생 클래스 주소

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

**class** **Builder**{};

**class** **BladeMaster** : **public** Archer, **public** Builder{};

int main(){

BladeMaster bm;

*//파생 클래스의 주소를 확인합니다.*

cout << "bm: " << &bm << endl;

Archer \*pA = &bm;

cout << "pA: " << pA << endl;

Builder \*pB = &bm;

cout << "pB: " << pB << endl;

*//다중 상속인 경우에는 기본클래스의 포인터가 항상 다르다.*

*//하지만 단일 상속인 경우에도 가끔은 pA와 bm이 다를 수 있다.*

*//bm: 0027FE1F*

*//pA: 0027FE1F*

*//pB : 0027FE20*

}

이게 무엇일까? 같은 객체를 나타내는데 주소 값이 다르다. 주소 값이 다르면 어떻게 되는 것일까? 한 객체에서 양 쪽의 주소를 오가기 위하여 포인터의 변위(포인터 값의 차이)를 계산한다. 이 변위를 찾는 동작이 런타임에 이루어진다. **본인의 포인터를 찾고서도 연산을 또 하는 것이다.**

**비효율적인 시간뿐만 아니라 미정의 동작을 낳을 수도 있다.** 예를 들어, std::string객체의 주소 값을 전달하는데 char\*로 캐스팅하여 전달한다면 이 값이 달라질 수 있다는 것이다. 이렇게 되면 미정의 급행열차에 올라서게 되는 것이다.

#### static\_cast<T>: 클래스에서 캐스팅이 갖는 의미

**class** **Archer**{

**public**:

Archer(int n) : playerNumber(n), counter(0) {}

**virtual** void attack(){

count();

cout << playerNumber << "'s Base shot! <" << counter << ">" << endl;

}

***//공격을 할 때마다 해당 객체의 카운터는 상승합니다.***

void count(){ counter++; }

**private**:

int playerNumber;

int counter;

};

**class** **Builder**{

};

**class** **BladeMaster** : **public** Archer, **public** Builder{

**public**:

BladeMaster(int n) : Archer(n) {}

**virtual** void attack(){

*//1. 현재의 파생 클래스를 기본 클래스로 형변환을 시도합니다.*

**static\_cast**<Archer>(\***this**).attack();

*//2. 해당 클래스의 기본 클래스를 불러옵니다.*

Archer::attack();

cout << "Critical blade!" << endl;

*//1번과 2번에서의 카운터는 1로 같습니다. 증가하지 않지요.*

}

};

int main(){

BladeMaster bm(3);

bm.attack();

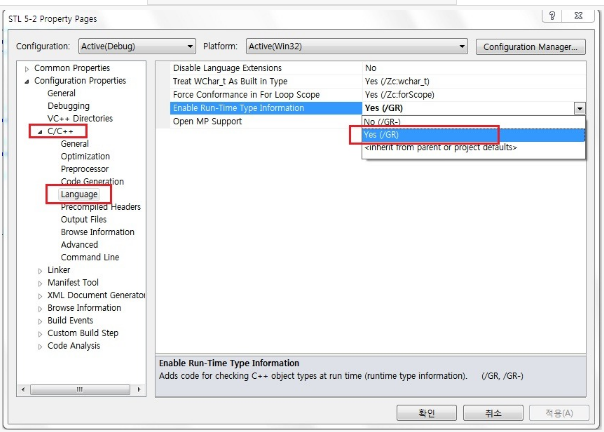
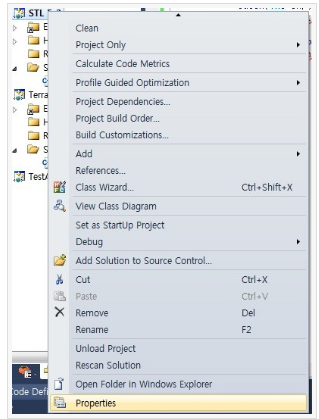
}

위에서 우리가 원하는 동작은 ‘***해당 클래스의 기본 클래스 호출***’이다. 하지만 1번의 **캐스팅 동작에서는 \*this에 대한 사본이 임시적으로 만들어지게 되는데 해당 객체가 아닌 사본의 객체에서 함수를 호출하게 된다.** 이는 원하는 동작이 나오지 않는다는 것이다. 그러므로 원하는 동작을 이루려면 2번과 같이 직접 기본 클래스를 호출해주어야 한다.  
위의 코드가 예시다. 코드에서 공격을 하게되면 카운터가 증가하는 구조다. 1번을 호출하면 카운터가 1이 되는데 2번을 호출해도 여전히 1이다. 이것은 1번과 2번을 호출한 객체가 서로 다르다는 의미이다. 이미 언급한대로 1번은 사본의 카운터이며 2번이 진짜 객체의 카운터다.

#### dynamic\_cast<T>: 성능 문제

###### RTTI(Runtime Type Information)

RTTI는 실행시간에 객체의 타입에 대한 정보를 얻게 해준다. dynamic\_cast를 사용하려면 이를 활성화시켜야 하며 방식은 아래와 같다.



이는 어떻게 작동할까? 컴파일러는 RTTI와 연결하기 위해 가상함수 포인터 테이블을 사용한다. c++언어는 이 앞에 4byte를 붙여 RTTI로 구분지어 사용한다.

**다이나믹 캐스트**는 파생 클래스의 함수를 호출하고 싶은데 그 객체를 조작할 방법이 기본 클래스밖에 없을 경우에 사용하고 싶을 것이다. 이는 성능면에서 상당히 문제가 많기로 유명하다. 왜냐하면 원하는 파생 클래스를 찾기 위해서 strcmp로 모든 문자열을 비교하는데 깊이가 깊어질수록 비용이 증가하기 때문이다. 동적 링크 지원(내부에서 다른 프로그램의 기능을 연결하는)을 동반해야 하기 때문에 이렇게 구현되었다고 한다.  
아래에서는 다이나믹 캐스트를 사용한 상황와 **이를 대처하는 방안을 제시**한다.

*//dynamic\_cast로 기본 클래스를 파생 클래스로 전환합니다.*

**if** (BladeMaster \*pBm =

**dynamic\_cast**<BladeMaster\*>(iter->get()))

*//해당 객체의 본체가 파생 클래스라면 공격할 것입니다.*

pBm->attack();

이런 것이 바람직하지 않다는 것이다.

###### 파생 클래스에 대한 포인터를 컨테이너에 담아두기

상황은 위에서와 같이 파생 클래스를 가지고 있는데 기본 클래스로써 사용하고 싶은 경우다.

**class** **BladeMaster** : **public** Archer, **public** Builder{

**public**:

BladeMaster(int n) : Archer(n) {}

**typedef** vector<shared\_ptr<Archer> > VPA;

VPA arcPtrs;

**typedef** vector<shared\_ptr<BladeMaster> > VPB;

VPB blmPtrs;

void arc\_command\_attack();

};

void BladeMaster::arc\_command\_attack(){

*//****해결1:******파생 클래스의 배열****이므로 캐스팅이 필요 없습니다.*

**for** (VPB::iterator iter = blmPtrs.begin();

iter != blmPtrs.end(); iter++)

(\*iter)->attack();

}

**해결1:** 기본 클래스와 파생 클래스에 대한 컨테이너를 따로 마련해두는 것이다. 그러면 기본 클래스든 파생 클래스든 꺼내쓸 수 있다. 하지만 문제가 좀 많다. 용량이 남아도는가?

###### 의미없는 가상 함수 만들기

**class** **BladeMaster** : **public** Archer, **public** Builder{

**public**:

BladeMaster(int n) : Archer(n) {}

**virtual** void attack(){

Archer::attack();

cout << "Critical blade!" << endl;

}

*//****해결2:*** *만약 파생 클래스의 함수만 불러오고 싶다면*

*//****해당 함수를 기본 클래스에서 정의를 안 하고 virtual을 선업****합니다.*

*//캐스팅을 피할 수 있습니다.*

**for** (VPA::iterator iter = arcPtrs.begin();

iter != arcPtrs.end(); iter++)

(\*iter)->attack();

};

**해결2:** 기본 클래스의 가상 함수를 불러오면 각각의 객체에 맞는 함수를 불러올 것이다. 여기서 기본 클래스에 해당 함수를 구현없이 선언하면 기본 클래스 포인터로도 파생 클래스의 함수를 불러오는 꼴이 된다.

모든 상황에서 가능하지는 않지만, 파생 클래스와 기본 클래스 관계를 파악하고 적절하게 캐스팅을 대체할 수 있는 예시이다.

###### 폭포식 dynamic\_cast

유지보수성 -100으로써, 반드시 피해야 할 형태이다.

void BladeMaster::arc\_command\_attack(){

**for** (VPA::iterator iter = arcPtrs.begin();

iter != arcPtrs.end(); iter++){

*//dynamic\_cast로 기본 클래스를 파생 클래스로 전환합니다.*

*//여러 가지 파생 클래스가 존재할 수 있으므로 모두 검사합니다.*

**if** (BladeMaster \*pBm =

**dynamic\_cast**<BladeMaster\*>(iter->get()))

pBm->attack();

else **if** (ArchMage \*pAm =

**dynamic\_cast**<ArchMage\*>(iter->get()))

pAm->attack();

else **if** (DarkKnight \*pDk =

**dynamic\_cast**<BladeMaster\*>(iter->get()))

pDk->attack();

*//이런...*

}

}

# 캐스팅은 최대한 격리하자.

하지만 캐스팅을 발본색원할 수는 없다. 캐스팅을 해야 하는 코드를 내부 함수 속에 몰아 놓고, 그 안에서 일어나는 천한 일들은 외부에서 알 수 없도록 인터페이스로 막아두는 방법으로 해결할 수 있다. **즉, 오류가 생길 수 있는 여지가 밖으로 퍼져나가지 않음을 보장해야 한다. 캡슐화 같은 것이다.**

***다른 방법이 있다면 캐스팅은 피하십시오. 특히 수행 성능에 민감한 코드에서 dynamic\_cast는 몇 번이고 다시 생각하십시오. 설계 중에 캐스팅이 필요해졌다면, 캐스팅을 쓰지 않는 다른 방법을 시도해 보십시오.***

***캐스팅이 어쩔 수 없이 필요하다면, 함수 안에 숨길 수 있도록 해 보십시오. 이렇게 하면 최소한 사용자는 자신의 코드에 캐스팅을 넣지 않고 이 함수를 호출할 수 있게 됩니다.***

***구형 캐스팅을 쓰려거든 C++ 스타일의 캐스트를 선호하십시오. 발견하기도 쉽고, 설계자가 어떤 역할을 의도했는지가 더 자세하게 드러납니다.***

[항목28] 내부요소에 대한 핸들 반환

어떤 객체의 내부요소(private멤버)에 대한 핸들을 반환하는 코드는 되도록 피해야 한다.

# 핸들(handle)

핸들은 윈도우즈 프로그래밍에서 각각 프로그램 창에 대해 식별하게 해주는 객체로서 많이 쓰이지만 ***일반적으로는 다른 객체에 손을 댈 수 있게 하는 매개자라고 정의한다.***

포인터, 참조자, 반복자가 핸들이다. 내부 객체를 직접 바꿀 수 있지 않은가?

# [핸들을 반환 문제1] 캡슐화가 깨진다.

**class** **Weapon**{

**public**:

Weapon(int t, int p) : type(t), power(p) {}

void setType(int t) { type = t; }

void setPower(int p) { power = p; }

**private**:

int type;

int power;

};

**class** **Archer**{

**public**:

void setWeapon(shared\_ptr<Weapon> pW)

{ pWeapons.push\_back(pW); };

Weapon& upgradeWeapon(int n) **const**

{ **return** \*pWeapons[n].get(); };

**private**:

*//나의 무기들은 private에 보호되어 변경할 수 없습니다.*

vector<shared\_ptr<Weapon> > pWeapons;

};

int main(){

Archer player1;

*//player1이 6개의 무기를 장착합니다.*

**for** (int i = 0; i < 6; i++)

player1.setWeapon(shared\_ptr<Weapon>(**new** Weapon(i, 100)));

*//접근할 수 없는 나의 무기를 마음대로 변경하는데 성공했어요.*

*//이는 캡슐화가 깨짐을 의미합니다. 위험할 수 있어요.*

player1.upgradeWeapon(1).setPower(1000);

}

내부요소는 private의 멤버를 의미한다. private멤버는 바꿀 수 없어야 한다. 하지만 Weapon&을 보라. 객체 복사본이 아닌 그 자체를반환하면 당연히 이에 접근이 가능하고 캡슐화가 깨지지 않겠는가?

#### 객체 참조 반환이 시사하는 바

**참조된 객체의 함수의 접근도에 따라 캡슐화 정도가 정해진다.** 데이터 멤버가 private이라고 해도 객체를 반환한다면 함수로 이를 접근하기에 public과 다름없다. 그리고 어떤 객체에서 호출한 상수 멤버 함수의 참조자 반환 값의 실제 데이터가 그 객체의 바깥에 저장되어 있다면, 이 함수의 호출부에서 그 데이터의 수정이 가능하다. 즉, 자신 밖에 있는 주소로 접근하면 수정이 가능하다.

#### 반환한 핸들을 지키는 방법

간단하다. 참조한 값을 바꾸지 못하게 함수 앞에 const를 붙인다.

**const** Weapon& upgradeWeapon(int n) **const**;

# [핸들을 반환 문제2] 무효참조 핸들이 생길 수 있다.

반환한 핸들을 지킨다 해도, 문제가 발생할 수 있다.

#### 무효참조 핸들(dangling handle)

핸들이 있기는 하지만 그 핸들을 따라갔을 때, 실제 객체의 데이터가 없는 것입니다.

#### 유효 범위의 문제

int main(){

Archer player1;

*//player1이 6개의 무기를 장착합니다.*

**for** (int i = 0; i < 6; i++)

player1.setWeapon(shared\_ptr<Weapon>(**new** Weapon(i, 100)));

*//새로운 무기를 꺼내어 옮깁니다.*

***//앞에 반드시 const를 붙여야 참조할 수 있습니다.***

**const** Weapon \*pNewWeapon = &(player1.upgradeWeapon(1));

*//새로운 무기보다 플레이어가 먼저 죽는다면?*

**delete** &player1;

*//심각한 오류가 생기는군요.*

pNewWeapon->getInfo();

}

이와 같은 오류는 앞에서 언급한 무효참조 떨어져 나간 핸들이 참조한 핸들보다 오래 살기 때문이다. 이미 죽은 핸들을 참조할 방법이 없으므로 오류가 생긴다.

#### 임시 객체는 곧 사라진다.

위의 경우는 상당히 직관적이지만 아래를 보라.

const StrongStrong& StrongMissile::getStrong()

{

return strong;

}

const StrongMissile createArrow(const Quiver& q)

{

return StrongMissile();

}

int main(){

Quiver niceQuiver;

const StrongStrong \*strong = &(createArrow(niceQuiver).getStrong());

return 0;

}

이러한 문제가 생기는 본질적인 문제는 임시 객체라는 점이다. 해당 변수에 저장한 형태가 아니다. 저 문장이 지나면 사라질 객체다. 이는 객체를 반환하는 식에서 많이 볼 수 있다. 포인터가 아닌 객체 사본을 반환한다면 임시적인지 확인해야 한다.  
하지만 핵심은 객체를 반환하지 말라는 것이 아니다. 반환한 객체를 저장해서 사용하면 아무런 문제가 없다. 참조를 반환하는 getStrong이 문제라는 것이다.

***어떤 객체의 내부요소에 대한 핸들(참조자, 포인터, 반복자)을 반환하는 것을 피하세요. 캡슐화를 높이고 상수성을 유지하며 무효참조 핸들이 생기지 않게 합니다.***

[항목29] 예외 안전성 확보하기

참고: <http://www.slideshare.net/ssuser239f08/tcpl-14>

# 예외 처리의 기본

모든 프로그램 과정이 내가 의도한 대로 흘러가면 최상이다. 하지만 항상 그렇지 않다.

#### 예외에 대하여

*‘시스템의 특정 부분에 어떤 요청이 들어왔는데 거기서 그것을 처리할 수 없는 경우’*

이를 **예외가 발생**했다고 한다.

**예외 처리**(例外 處理) 또는 **오류 처리**는 일반적인 실행의 흐름을 바꾸는 몇 가지 조건을 처리하도록 설계한 프로그래밍 언어의 개념이나 컴퓨터 하드웨어 구조를 말한다.

일반적으로 프로그램이 처리되는 동안 특정한 문제가 일어났을 때 처리를 중단하고 다른 처리를 하는 것을 예외 처리라고 한다.

오류가 발생했다면 어떻게 하는가? 오류가 생겼는데 무시할 것인가? 이 ‘어떻게’가 예외 처리인 셈이다.

#### 어떻게 예외를 처리할 것인가

일반적으로 다음과 같이 할 수 있다.  
1. 프로그램을 종료하거나   
2. 에러를 나타내는 값을 반환하거나   
3. 정상적인 값을 반환하고 비정상적인 상태로 프로그램을 끝내든가  
4. 에러의 경우에 호출되도록 만들어 둔 함수를 호출한다.  
5. 예외가 절대로 생기지 않음을 보장한다.

###### 기본 형태: 예외가 발생(try)하면 던져서(throw) 받는다(catch).

class mathErr

{

public:

virtual void debug\_error() const

{

cerr << "math error.\n";

}

};

class divideZero : public mathErr

{

public:

virtual void debug\_error() const

{

cerr << "divide zero error.\n";

}

};

int calc(int a, int b)

{

if (b == 0) throw divideZero();

return a / b;

}

int main()

{

try

{

cout << calc(9999, 0) << endl;

}

catch (mathErr& e)

{

e.debug\_error();

}

return 0;

}

상속을 이용하면 기본 클래스에 속해있으면서 그에 맞는 예외를 맞춰서 보여준다.

###### 프로그램을 종료한다.

try

{

cout << calc(9999, 0) << endl;

}

catch (mathErr& e)

{

e.debug\_error();

abort();

}

가장 단순하면서 무식한 방법처럼 보인다.

###### 에러를 나타내는 값을 반환한다.

적절한 메시지를 보여주면 되겠다. 상속, 예외 재전송을 이용하여 유연한 대처를 하자.

예외 재전송(예외 전파)

catch 한 예외를 다시 throw 할 수 있다. 재 throw 된 예외는 본래의 타입으로 던져진다.

void varification(int a, int b, int ret)

{

if (ret\*b != a) throw mathErr();

}

int calc(int a, int b)

{

try

{

//던진 예외는 가장 가까운 catch가 받습니다.

//zero예외도 "re throw."가 출력됩니다.

if (b == 0) throw divideZero();

int ret = a / b;

//여기에 try ~ catch구문이 없어도

//varification의 예외는 main으로 전파됩니다.

varification(a, b, ret);

return ret;

}

catch (mathErr& e)

{

cout << "re throw.\n";

throw;

}

return -1;

}

int main()

{

try

{

cout << calc(9999, 5) << endl;

}

catch (mathErr& e)

{

e.debug\_error();

}

return 0;

}

###### 정상적인 값을 반환하고 비정상적인 상태로 프로그램을 끝낸다.

그냥 무시하라는 의미다.

###### 에러의 경우에 호출되도록 만들어 둔 함수를 호출한다.

virtual void debug\_error()에 적절한 함수를 구현하면 되겠다. 혹은 catch안에서 함수를 호출하라.

###### 예외가 절대로 생기지 않음을 보장한다.

noexcept는 말 그대로 이 함수에서는 예외가 발생하지 않는다는 명세다. VS2013에는 noexcept를 사용할 수 없다. noexcept의 경우, 이러한 명세임에도 불구하고 만약 예외가 튀어나오면 그 예외는 무시할 수 있으며 스택이 풀릴수도 있고 아닐 수도 있다. 자세한 것은 emc++’s [항목14] 참고

int calc(int a, int b) \_NOEXCEPT // noexcept

{

int ret = a / b;

varification(a, b, ret);

return ret;

}

int main()

{

try

{

int b = 0;

while (b == 0) cin >> b;

cout << calc(9999, b) << endl;

}

catch (mathErr& e)

{

e.debug\_error();

}

return 0;

}

#### 예외 처리에 대한 세부사항

특별한 경우의 예외에 대해서 생각해보자.

###### 스택 풀기

예외가 발생했을 때 예외를 처리해 줄 처리자를 찾아 스택을 거슬러 올라가 탐색하는 과정을 말한다. 스택 풀기가 일어나면, 지역적으로 생성된 객체는 모두 다 소멸자가 호출된다.   
스택 풀기시 소멸자가 호출 될 때, 다시 예외가 발생하면? terminate() 함수가 호출되어 프로그램이 종료된다.

class CalcMaster

{

public:

~CalcMaster()

{

cout << "CalcMaster destroy.\n";

}

};

class PencilPrince

{

public:

~PencilPrince()

{

cout << "PencilPrince destroy.\n";

}

};

void varification(int a, int b, int ret)

{

if (ret\*b != a) throw mathErr();

}

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;

}

###### 소멸자 내부에서의 예외

소멸자에서 예외가 발생하면 해당 객체는 이미 사라진 이후이므로 복구할 방법이 없다. 많은 함수에서 종료되거나 예외가 발생하면 소멸자를 호출하게 된다. 여기에 문제가 생기면 많은 부분에 문제가 생김을 예약하는 것과 같다.   
또한, 스택풀기 중에 소멸자 예외가 발생하면 프로그램이 종료된다. - [항목8] 참고

###### 생성자, 복사 생성자 내부에서의 예외

생성되기도 전이므로 이는 객체가 없는 것이다. 그러므로 소멸자는 호출되지 못한 채로 메모리가 누출된다.   
생성 중 예외 방지는 다음과 같다.

class CalcMaster

{

public:

CalcMaster(int a) try : core(init())

{

core = a;

}

catch (mathErr& e)

{

e.debug\_error();

}

~CalcMaster()

{

cout << "CalcMaster destroy.\n";

}

private:

int core;

};

###### 복사 대입 연산자에서의 예외

예외 발생시, 소멸자를 호출하게 된다.

###### 예외를 불가 명세임에도 불구하고 예외가 발생하면?

지정한 예외가 아닌 다른 예외를 던지면 std:: unexpected() 함수가 호출된다. set\_unexpected()로 핸들러를 설정할 수 있으며, 이 핸들러에서 새 예외 처리를 throw 하면 원하는 예외 처리를 할 수 있다.

// set\_unexpected example

#include <iostream> // std::cerr

#include <exception> // std::set\_unexpected

void myunexpected() {

std::cerr << "unexpected called\n";

throw 0; // throws int (in exception-specification)

}

void myfunction() throw (int) {

throw 'x'; // throws char (not in exception-specification)

}

int main(void) {

std::set\_unexpected(myunexpected);

try {

myfunction();

}

catch (int) { std::cerr << "caught int\n"; }

catch (...) { std::cerr << "caught some other exception type\n"; }

return 0;

}

# 바른 프로그래밍을 위한 고수의 조언

1. 에러 처리에는 예외를 사용하자.   
2. 지역적 흐름 제어 구조로도 충분한 곳에는 예외를 사용하지 말 것.   
3. 자원 관리에는 "자원획득 즉 초기화"(Resource Acquisition is Initialization, RAII) 기법을 사용하자.   
4. 모든 프로그램에 예외 안전성을 부여할 필요는 없다.   
5. 불변속성을 유지하는 데도 "자원획득 즉 초기화" 기법 및 예외 처리자를 사용 하자.   
6. try 블록의 사용은 최대한 자제하자, 직접 에러 처리자 코드를 두기 보다는 "자원획득 즉 초기화" 방법을 적극적으로 활용하자.   
7. 모든 함수에서 모든 가능한 에러를 도맡아 처리할 필요는 없다.   
8. 생성자가 수행되다 오동작이 나면, 예외를 던져서 이것을 알릴 것.   
9. 대입 연산 중에 예외를 던지기 전에, 모든 연산자가 유효한 상태에 있도록 하자.   
10. 소멸자에서는 예외 발생을 피하자.   
11. 모든 예외를 받아 보고하는 코드는 main()에 두도록.   
12. 에러 처리와 상관 없는 코드와 에러 처리 코드는 격리 시킬 것.   
13. 생성자에서 예외를 던지려 할 때 미리 생성자에서 자원이 획득된 상태라면, 해당 자원을 모두 해제하자.   
14. 자원 관리는 단계적으로 진행되도록 하자.   
15. 주로 사용하게 될 인터페이스에 대해서는 예외 지정 기능을 사용하자.   
16. new에 의해 할당되었는데 예외 발생으로 인해 해제되지 않은 메모리가 메모리 누수를 일으킨다는 사실을 잊지말자.   
17. 함수에서 발생할 수 있는 예외는 언젠가 발생할 것이란 가정을 항상 빠뜨리 지 말도록.   
18. 무릇 예외라면 exception 클래스에서 파생되어야 한다는 편견을 버리자.   
19. 라이브러리는 자신의 판단만으로 바로 프로그램을 끝내 버리게 만들면 안 된다. 예외를 발생시켜 라이브러리 호출자에게 판단을 넘기자.   
20. 라이브러리는 최종 사용자를 염두에 둔 진단 출력 메시지를 만들어서도 안 된다. 예외를 발생시켜 호출자에게 판단을 넘기자.   
21. 예외 처리 전략은 설계 단계에서 세우자.

# 예외 안전성(exception safety)

예외에서 안전해야 하는 것은 당연하다. 만일 아니라면, 이건 쓰레기 프로그램이다. 비정상적인 결과를 보여줄 것이기 때문이다. 그러므로 어떻게든 비정상적인 결과가 아닌 그러한 결과는 예외로 처리하여 정상적인 결과가 나오게 해야 한다.

위에서도 언급되지만 try ~ catch구문으로 던진 예외를 받는 방식은 그리 좋은 발상이 아니다. 좀 거추장스러운 조건문이 생겼을 뿐이다. 여러 개의 클래스를 상속한 상태라면 깔끔하게 예외를 던지기란 정말 어려울 것이다.

위에서 예외를 처리하는 5가지 방법에 대해 논의했다. **여기서는 그 중 실제로 사용해야 하는 3가지를 제시한다.** 가장 좋은 방법은 *예외가 절대로 없다는 보장*이다. 인터페이스 설계 시, 예외가 발생하지 않도록 설계하는 것이다. 하지만 이는 쉽지 않으므로 앞으로 *언급할 우회책인 강력한 보장*이나 최소한의 수준의 *기본적인 보장*을 생각하여 예외에 대해 안전하게 객체를 생성해야 할 것이다.

예외 안전성을 확보하려면 2가지의 요구사항을 만족해야 한다.

#### 자원이 세지 않도록 만든다.

예외가 발생했을 때, 어떠한 자원이 잡힌채로 함수가 사라지면 안 된다.

#### 자료구조가 더렵혀지는 것을 허용하지 않는다.

의도하지 않은 어떠한 자료가 변하거나 삭제될 수 없다.

각각의 요구사항을 만족하려면 어떻게 설계해야 할까?

# [해결] 자원 누출 문제

스마트 포인터를 사용한다. 물론 스스로 관리하는 RAII객체를 스스로 만들어도 적절하다. **위에서 그렇게 중요하다고 외친 RAII를 사용하라는 의미이다.** 아래는 Lock클래스를 통해 자원 누출 문제를 해결한 코드이다. **- [항목13]**

**class** **Lock**{

**public**:

Lock(mutex\* m) : mutexPtr(m){ mutexPtr->lock(); }

~Lock(){ mutexPtr->unlock(); }

**private**:

mutex\* mutexPtr;

};

**class** **Skill**{

**public**:

Skill(int p) : power(p) {}

**private**:

int power;

};

**class** **Archer**{

**public**:

Archer(int n) : number(n), level(1), skill(**new** Skill(1)) {}

void moreStrongArcher(Skill& s);

**private**:

*//뮤텍스를 가진 아처입니다.*

mutex m;

int number;

int level;

Skill\* skill;

};

void Archer::moreStrongArcher(Skill& s){

*//스레드들이 순서대로 들어오게 해줍니다.*

*//m.lock(); 이는 예외가 발생하면 메모리가 누출됩니다.*

***//이는 자동 잠금 장치를 가지게 됩니다.***

Lock lm(&m);

*//기존의 스킬을 삭제하고 새로운 스킬을 배웁니다.*

**delete** skill;

skill = **new** Skill(s);

level++;

cout << "Upgrade is complete." << endl;

}

void t\_func(int id){

}

int main(){

Archer a(1);

vector<Skill> s;

**for** (int i = 0; i < 10; i++) s.push\_back(Skill(i + 1));

**thread** t[10];

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

*//일반적인 방법은 전역 함수나 static함수만 가능하다.*

*//t[i] = thread(t\_func, i);*

*//클래스 멤버 함수같은 경우에는 다음과 같이 선언한다.*

t[i] = **thread**([&](){ a.moreStrongArcher(s[i]); });

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

t[i].join();

**return** 0;

}

# [해결] 자료구조 보존

예외가 발생하여 자료를 엉터리로 만들어놓고 죽어버릴 수 있다. 이를 보존하기 위하여 3가지 중 하나를 선택하여 설계해야 한다.

#### 기본적인 보장

모든 것을 유효한 상태로 유지하겠다는 보장이다. **어떠한 객체나 자료구조는 동작의 일관성(어떤 부분만 변하지 않는 것)을 갖게 되지만 그 상태가 정확히 어떤지는 예측할 수 없다.** 예를 들어 위에서 강력한 업그레이드를 실행하는데 있어서 예외가 발생하면 기본적인 보장의 경우라면 계속해서 업그레이드를 시도할 수 있고, 아니면 그 전 상태로 되돌릴 수도 있다는 의미이다. 위의 코드를 기본적인 보장을 적용하여 작성한 코드를 살펴보자.

**class** **Archer**{

**public**:

Archer(int n) : number(n), level(1), skill(**new** Skill(1)) {}

void moreStrongArcher(Skill& s);

**private**:

*//뮤텍스를 가진 아처입니다.*

mutex m;

int number;

int level;

shared\_ptr<Skill> skill;

};

void Archer::moreStrongArcher(Skill& s){

Lock lm(&m);

***//reset을 이용한 스마트포인터 갱신이 보입니다.***

skill.reset(**new** Skill(s));

level++;

cout << "Upgrade is complete." << endl;

}

객체가 스마트포인터에 의해 관리되므로 skill에 대한 안전성이 보장된다.

하지만 매개변수에서 예외가 발생할 수 있으므로 강력한 보장으로 보완할 수 있다.

#### 강력한 보장

프로그램의 상태를 절대로 변경하지 않겠다는 보장이며 **한마디로 되돌리기다.** 가장 쓰기 쉬우며 효율성이 좋은 보장이다. 강력한 보장은 [기법: 복사 후 맞바꾸기]를 이용하여 작성한다. 다음과 같다.

*//struct로 선언하여도 Archer안에 private으로 들어가므로 캡슐화가 된다.*

**struct** AImpl{

AImpl(int n) : number(n), level(1), skill(**new** Skill(1)) {}

int number;

int level;

shared\_ptr<Skill> skill;

};

**class** **Archer**{

**public**:

Archer(int n) : pImpl(**new** AImpl(n)) {}

void moreStrongArcher(Skill& s);

**private**:

*//뮤텍스를 가진 아처입니다.*

mutex m;

*//모든 데이터를 따로 구조체로 빼놓았습니다.*

***//pimpl관용구 입니다.***

shared\_ptr<AImpl> pImpl;

};

void Archer::moreStrongArcher(Skill& s){

Lock lm(&m);

***//기존의 정보를 복사하여 저장해둡니다.***

**shared\_ptr<AImpl> pNew(new AImpl(\*pImpl));**

***//사본을 수정합니다.***

pNew->skill.reset(**new** Skill(s));

pNew->level++;

***//서로 맞바꿉니다. 지금은 pNew를 갖게 되군요.***

swap(pImpl, pNew);

cout << "Upgrade is complete." << endl;

}

이는 함수 안에서의 강력한 예외 안전성을 보장한다. 원리는 일단 해놓고, 문제가 없으면 실제로 적용한다는 것이다. 선테스트 후적용인 셈이다. 하지만 기본적인 보장보다 메모리도 많이 차지하며 코드도 길기 때문에 시간이나 공간이 여유로워 실용성이 보장될 때만 사용하는 것이 좋다.

#### 예외불가 보장

예외를 절대로 던지지 않겠다는 보장이다. 함수 뒤에 **noexcept(C++98: throw())**가 **예외를 던지지 않겠다**는 보장이라는 뜻이다. 만약 ()안에 타입이 있다면 그 타입에 대해서 예외를 던지며 unexpected함수를 호출한다. 이 함수는 사용자가 set\_unexpected()로 지정할 수 있으며 지정된 함수가 없으면 terminate()를 호출한다. 다음은 예외 던지기 활용에 대한 내용이다.

참고: <http://www.gpgstudy.com/forum/viewtopic.php?t=18428>

*//쓰지 않는 것을 권장한다. VC++에서는 무시한다.*

double myfunction(char param) **throw** (int);

*//표준적인 예외 활용. 다음과 같이 활용한다.*

#include *<exception>*

**class** **myexception** : **public** exception

{

**virtual** **const** char\* what() **const** **throw**()

{

**return** "My exception happened";

}

} myex;

int main(){

**try**{

int a;

cin >> a;

*//예외가 생기면 던진다(throw).*

**if** (!a) **throw** a;

**else** **throw** myex;

}

*//던진 값을 받는다.*

**catch** (int e){

cout << "no accept " << e << endl;

}

**catch** (exception& e)

{

cout << e.what() << '\n';

}

**return** 0;

}

이는 어떤 구현이냐에 따라 예외가 보장되는 정도가 따른다. 곧, 예외를 던진다고 해서 절대로 예외불가 보장이 되는 것은 아니다.

#### 함수의 부수효과

한 함수에 2개 이상의 함수가 들어있다면 다른 1개의 함수의 예외가 전체 함수를 예외에서 위험하게 만들 수 있다. 즉, 하나의 함수가 예외에서 완벽하더라도, 나머지 하나에서 예외가 터지면 기존의 함수를 복구해야 하는데 이미 바뀌어버린 이후다. 전체 함수는 그렇지 못한 것이다. 이를 **함수의 부수효과**라고 부르며 **비지역 데이터**에 영향을 주는 경우에는 더욱 그렇다. **그러므로 전체 프로그램에서 하나라도 예외에 안전하지 않은 함수가 있다면 이는 전부가 예외에서 위험하다고 볼 수 있다.**

***예외 안전성을 갖춘 함수는 실행 중 예외가 발생하더라도 자원을 누출시키지 않으며 자료구조를 더럽힌 채로 내버려 두지 않습니다. 이런 함수들이 제공할 수 있는 예외 안전성 보장은 기본적인 보장, 강력한 보장, 예외 금지 보장이 있습니다.***

***강력한 예외 안전성 보장은 복사 후 맞바꾸기 기법을 써서 구현할 수 있지만, 모든 함수에 대해 강력한 보장이 실용적인 것은 아닙니다.***

***어떤 함수가 제공하는 예외 안전성 보장의 강도는, 그 함수가 내부적으로 호출하는 함수들이 제공하는 가장 약한 보장을 넘어서지 않습니다.***

[항목30] 인라인 함수의 이해

코드 최적화 참고: <http://egloos.zum.com/msbfox/v/76290>

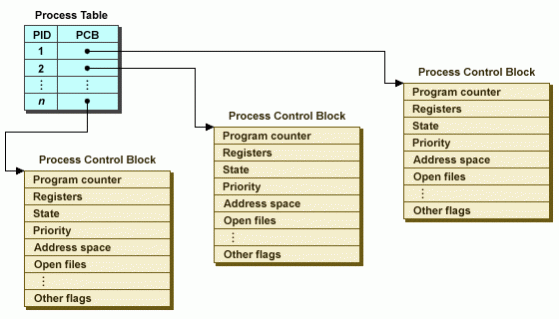
# 함수 호출 과정

함수가 어떻게 호출되는지 과정을 이해해야 한다.

#### 함수 호출의 시작: process를 담은 PCB

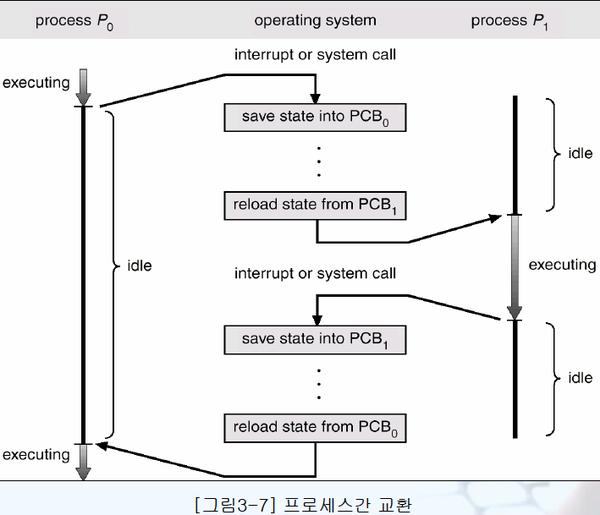
프로그램의 시작은 PCB이다. 이는 **process table**와 다음과 같은 관계를 맺는다.

The **process table** is a data structure maintained **by the operating system** to facilitate *context switching* and *scheduling*, and other activities.



###### process의 실행

이렇게 만들어진 여러 개의 process들은 CPU에 의해 실행될 수 있다. 어떤 프로세스가 어떤 순서로 실행되는가? 이는 PCB의 정보로 운영체제가 정해준다. 운영체제가 적절하게 끼어들어서 아래의 그림처럼 여러 개의 process를 교환하는 모습이 보인다. 이를 *context switching*이라고 한다.

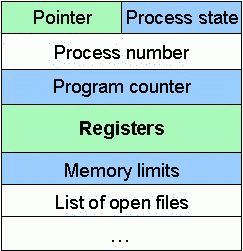


###### Process Control Block (PCB)

**PCB(Process Control Block):** CPU에게 주는 해당 Process에 대한 정보를 담은 블록

프로세스가 실행되면(exe가 실행되면) 운영체제에서 PCB를 할당해준다.

Each and every process state is represented as a process control block by the operating system. This is also called as **task control block.** The PCB contains many pieces of information associated with the specific process.



[**Process State**](https://cssimplified.wordpress.com/2015/11/23/process-states/)**:**The state may be new, ready, running, wait and halted.

**Program Counter:** The counter indicates the address of the next instructions to be executed for the process.

**CPU Registers:** The registers vary in number and type, depending on the computer architecture. These registers include accumulator, stack pointers, index registers.

These information are important and saved when an interrupt occurs, to allow the process to be continued correctly afterward.

**CPU scheduling information:** This information includes process priority, pointers to scheduling queues and any other scheduling parameters.

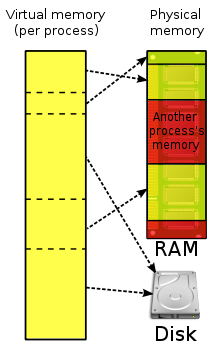
**I/O status information:** This includes information such as the list of allocated I/O devices, list of opened files and so on.

#### 해당 프로세스에 메모리 할당

위에서 보는 것처럼, PCB에는 여러가지 정보가 있으며, **그 중 프로그래밍에 가장 중요한 프로세스에 할당된 메모리도 포함된다.** 그리고 여기에 스택프레임이 포함된다.

###### 메모리 구조: 코드 / 전역 / 스택 / 힙

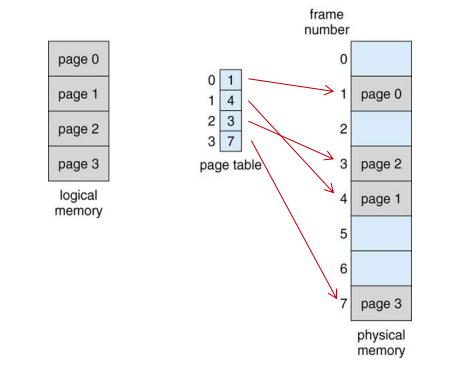
프로그램을 실행시키면 로더(디스크에서 주기억장치로 가져옴)에 의해 프로그램이 **주 기억 장치인 메모리(Random Access Memory, RAM)에 적재**된다. 이렇게 적재된 프로그램은 메모리상에서 **스택(First In, Last Out)의 형태로 구현**된다.

가상 메모리

32비트 시스템에서 프로세스(실행 중인 프로그램) 생성시 4GB의 메모리를 할당받을 수 있는데, 이는 램만으로는 충당하기엔 턱없이 부족하다.

그래서 운영체제는 램과 하드디스크를 하나로 묶어 ***가상 메모리***로 관리한다. 대부분의 시스템에서는 주로 ***페이징(Paging)***이라는 기법으로 가상 메모리를 관리한다.

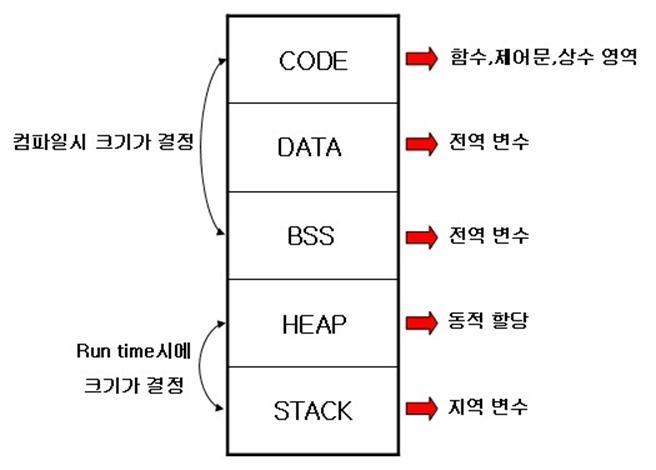
실제로 프로세스에서 사용하는 메모리는 곧 가상 메모리가 되는 것이다.

페이징

페이징이란, 테이블을 관리하여 물리적 메모리의 주소를 매핑하여 논리적 메모리를 관리한다. 이는 연속되지 않은 물리적 메모리도 효율적으로 사용하게 해준다.

이 기법은 하드디스크의 메모리를 가상 메모리로 올리기 위한 메모리 관리 기법으로도 사용되는 것이다.

**그럼 이제 이렇게 할당된 가상 메모리가 어떻게 구성되어 있는지 살펴본다.**

  
1. **CODE**: 프로그램의 소스코드가 저장된다.

2-1. **DATA**: 전역변수(global), 정적변수(static), 배열(array), 구조체(structure) 등이 저장된다. 초기화 된 데이터는 data 영역에 저장된다.

2-2. **BSS**: 초기화 되지 않은 데이터는 BSS (Block Stated Symbol) 영역에 저장된다.

3. **HEAP**: 크기가 가변한다. 동적 할당된 메모리가 적재된다. 프로그래머의 필요에 따라 할당하여 사용할 수 있다. 위에서부터 채워져 내려옴.

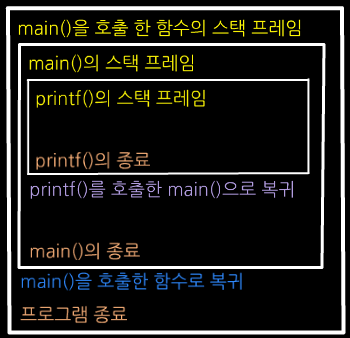
4. **STACK**: 크기가 가변한다. 지역변수가 저장된다. 스택에는 여러 개의 스택 프레임이 존재한다. 데이터 용량의 불확실성을 가지므로 밑에서부터 채워 올림, 또한 커널 영역을 침범하지 않게 하기 위해 밑에서부터 채워 올림 (스택 프레임에 대한 설명은 아래 참고)

#### 함수의 적재 과정: 스택프레임

어떤 함수가 호출되었을 때 그 함수가 가지는 공간 구조이다.

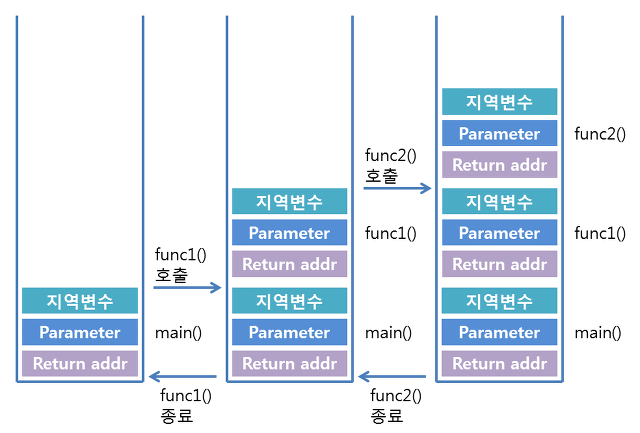
예를 들어  
void main() {  
 printf("Hellow");  
}

위와 같은 프로그램을 만들었다면 프로그램이 실행 되었을 때 main()이라는 함수가 호출되고 그 다음에 printf()라는 함수가 호출 된다.



스택프레임은 코딩의 순서에 맞춰 LIFO 구조로 차곡차곡 쌓여진다.

stack에서 주로 사용되는 세 개의 레지스터로는 EBP, ESP, EIP가 있습니다.  
EBP는 Stack frame의 시작점을 가리키고,  
ESP는 Stack의 TOP을 가리키고,  
EIP는 다음에 실행할 명령의 offset을 가지고 있습니다.



#### 스택에 어떻게 함수를 넣을 것인가?

호출규약에 따른다. -> 함수 호출 규약

**함수 호출 규약**이란 함수가 호출 되었을 때 전달 되었던 파라미터들에 대하여 함수 호출이 끝난 후 스택을 처리하는 방법에 대한 약속이다.

*즉, ESP를 정리하는 방법이다.*  


위처럼 비주얼베이직에서 속성을 통해 함수 호출 규칙을 직접 정할 수도 있다. 그러면 각 방법은 무엇이 다를까?

caller: 함수를 호출한 쪽 / callee : 호출 당한 함수

1**. cdcel**

- 주로 C언어에서 사용 된다.

- caller(함수를 호출한 쪽)에서 스택을 정리한다.

- 장점: 가변 길이 파라미터를 전달할 수 있다.

2. **stdcall**

- Win32 API에서 사용됨

- callee(호출 당한 함수)에서 스택을 정리 -> 가변인자 불가능

- 함수의 호출이 끝나고 존재하는 스택 정리 코드가 없어서 비교적 빠름.

3. **fastcall(C++)**

-이름에서 알 수 있듯이 좀 더 빠른 함수 호출을 위해 고안된 방식이다. CPU가 주 기억장치에서 인자를 불러다 쓸 경우보다 레지스터에서 바로 떙겨 쓰는 것이 훨씬 빠르기 때문에 피라미터의 마지막 2개를 레지스터(ECX, EDX)를 이용해 전달한다. (나머지는 스택에 전달)

- 장점: 좀더 빠른 함수 호출 가능

- 단점: ECX, EDX에 대하여 백업이 필요하거나, 함수에서 ECX, EDX를 다른 용도로 써야 할 경우, 파라미터를 따로 저장해야 함

# 인라인 함수가 뭡니까?

해당 코드 자체를 그대로 삽입하라고 요구하는 한정어(inline)를 사용한 함수

즉, 함수를 호출하는 구조가 아닌 컴파일 시 실행 과정에 그대로 붙여서 원래 하나의 문장이었던 것처럼 사용하게 해준다.

#### vs.아웃라인(outline) 함수

일반적으로 사용되는 함수를 말한다. 이는 **함수 호출 비용**이라는 것이 존재한다.   
보통의 함수 호출의 정확한 비용은 구현에 종속적이다. 그것은 보통 현재 스택 상태를 저장하고 스택으로 함수의 인수들을 푸시하고 그것들을 초기화하며 함수의 명령들을 포함하는 메모리 주소로 점프하는 것을 포함한다. 그 이후에만 함수가 실행을 시작한다.  
함수가 리턴할 때 역순의 동작들이 발생한다.

# 인라인 함수의 이점

함수를 호출하지 않고 코드를 그대로 삽입한다는 점 자체는 성능 이득이다.

###### [*이점1]* 매크로보다 안전하고 쓰기 좋다. – **[항목2]**

###### *[이점2] 오버헤드 걱정이 없다.*

위에서 언급한바와 같이 런타임에 함수 주소로 이동하는 것이 아닌 컴파일 타임에 본문에 그대로 붙여버리므로 오버헤드(함수 호출 비용)가 없다.

###### ***[이점3] 컴파일러가 함수 본문에 대해 문맥별 최적화를 걸기 유리하다.***

대체적으로 컴파일러 최적화는 함수 호출이 없는 코드가 연속적으로 이어지는 구간에 적용되도록 되어있다. 이는 인라인 함수에 적용되기 적절하며 일반적인 함수에는 적용되지 않는다.

그 외 코드 최적화에 관한 내용 참고: <http://egloos.zum.com/msbfox/v/76290>

# 인라인 함수의 단점

###### 목적코드가 커진다.

**목적코드**: 컴파일러나 어셈블러가 소스 코드 파일을 컴파일 또는 어셈블해서 생성하는 파일이다. 목적 파일들은 기계어나 혹은 이에 준하는 RTL과 같은 이진 코드로 이루어져 있다. 링커는 여러 개의 목적 파일을 묶어 커널과 연결함으로써 실행 파일을 만들거나, 혹은 라이브러리를 만들어내는 데에 쓰인다.

당연하다. 컴파일 시 본문에 붙여버리므로 함수 본문이 죄다 목적코드의 길이가 되는데 이는 메모리 면에서 치명적일 수 있다. 10번 반복되는 인라인 함수라면 10번 모두 목적코드에 포함되게 된다. 그에 따라 페이징 횟수가 늘어나고 캐시 적중률도 떨어지게 된다.

**[오히려 장점]** 하지만 반대로 함수 크기가 작다면 되려 함수 호출에 필요한 코드보다 짧아져서 메모리도 절약하며 페이징 횟수나 적중률에서 이득을 볼 수 있다.

# 인라인 함수 사용하기

암시적으로와 명시적으로 사용할 수 있으며, 템플릿이 만드는 모든 함수가 인라인으로 사용하고 싶다면 이와 함께 사용할 수 있다. 참고로 템플릿 인스턴스화(어떠한 객체로 만들 것인지 컴파일러가 결정하는 작업)와 완전히 별개이다. 그러므로 무조건 템플릿을 인라인으로 선언하면 안 되고 아래의 주의사항을 고려한다. - **[항목44]**

**class** **Archer**{

**public**:

***//암시적으로 인라인 함수가 됩니다.***

*//헤더파일의 클래스 정의 내부에서 정의됩니다.*

int getPower() **const** { **return** power; }

**private**:

int power;

};

***//명시적으로 인라인 함수를 요청합니다.***

*//템플릿 또한 컴파일 타임에 시행되므로 인라인과 함께*

*//헤더파일에 있어야 하며 컴파일 타임에 시행된다.*

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

**inline** **const** T& std::max(**const** T& a, **const** T& b){

**return** a < b ? **a** : b;

}

# 때로는 인라인 요구를 무시할 수 있다. – 인라인 적용 조건

inline은 컴파일러 선에서 무시할 수 있는 요청이다.

###### 재귀 함수, 반복문

이 외에 코드가 복잡하다면 무시할 수 있다.

###### **가상함수**

**virtual**의 의미는 런타임 중, 어떤 함수를 호출할 지 결정하라는 의미이므로 컴파일 타임에 작업하는 인라인과는 완전히 상반된다.

###### 함수 포인터

함수 포인터는 함수가 있는 주소를 가리켜야 하므로 인라인 될 수 없을 것이다.

int f(){ **return** 0; }

*//pf는 함수f를 가리키는 함수 포인터입니다. 인라인 되지 않습니다.*

int(\*pf)() = f;

###### 생성자, 소멸자

이는 인라인을 거절한다기 보다 인라인 하기에 좋지 않은 함수들이다. 왜냐하면 비어있는 생성자라 할지라도 컴파일러가 보이지 않는 코드들을 생성하여 호출하기 때문이다.

# 인라인 기본 전략

인라인 함수에 대해 따진다면 다음과 같은 전략을 세운다.

###### 일단, 아무것도 인라인하지 않는다.

###### 정말 단순한 함수에 대해서만 인라인 함수로 선언한다.

###### 디버거를 제대로 쓰도록 만들고 정말로 필요한 부분에 인라인 함수를 놓는다.

#### 핵심은 컴파일 동작 성질을 이용한 최적화이다.

적절한 상황 고려로 적용한 인라인은 컴파일 타임의 장점을 극대화시키며 성능을 향상시킬 수 있다.

***함수 인라인은 작고, 자주 호출되는 함수에 대해서만 하는 것으로 묶어둡시다. 이렇게 하면 디버깅 및 라이브러리의 바이너리 업그레이드가 용이해지고 코드 부풀림 현상이 최소화되며, 프로그램의 속력이 더 빨라질 수 있는 여지가 최고로 많아집니다.***

***함수 템플릿이 대개 헤더 파일에 들어간다는 일반적인 부분만 생각해서 이들을 inline으로 선언하면 안 됩니다.***

[항목31] 컴파일 의존성

파일 간의 컴파일 의존성을 최대한 줄이자.

# 컴파일 의존성이란?

C++에서의 클래스 정의에서는 인터페이스와 구현이 깔끔하게 분리되어있지 않다. 즉, 섞여있을 수도 있다는 말이다.

**class** **Archer**{

**public**:

*//아래는 선언한 인터페이스일 뿐입니다.*

Archer();

void move() **const**;

void attack() **const**;

string info() **const**;

**private**:

*//아래는 구현 세부사항입니다.*

*//어떻게 정의되었는지 알아야만 합니다.*

string name;

vector<Skill> skills;

int power;

};

string, vector가 어떻게 정의되었는지 모르면 컴파일되지 않습니다. 그래서 이 구현 내용이 들어있는 다른 파일이 필요하겠죠? #include로 그것을 가능하게 합니다. 그리고 **우리는 이렇게 불러온 파일들과 의존관계가 맺어지는데 이를 ‘컴파일 의존성(compilation dependency)’라고 부릅니다.**

# 컴파일 의존성이 일으키는 문제

파일 A, B, C가 컴파일 의존성 관계에 있다고 하자. 여기서 A의 일부분만 수정을 하였다. 하지만 컴파일러는 ABC모두 다시 컴파일을 진행할 것이다. 왜냐하면 컴파일 의존성 관계에 있기 때문이다. 컴파일러는 해당 변경한 부분의 구현과 연관이 있으면 모두 검사해야 한다는 것이다.

# 컴파일 의존성 최소화하기

가장 기본적으로 전방선언을 한다. 객체는 찾을 수 있으므로 일단 큰 위기는 넘길 수 있다.

#### 전방 선언을 이용한 컴파일 의존도 최소화

참고: <http://egloos.zum.com/sweeper/v/2827565>

###### #include 의존성 최소화

큰 프로젝트를 컴파일하는 데 걸리는 시간은 #include를 얼마나 많이 그리고 얼마나 깊은 수준으로 사용하고 있느냐에 따라 달라진다.  
1. 거대한 헤더 포함 집합체 헤더 파일 제거  
2. 중복 헤더포함 제거  
**3. 전방 선언**

###### 전방 선언의 의미

위와 같이 전방선언을 하는 것은 선언된 클래스/구조체의 이름을 **간단히 컴파일러에게 심볼 테이블에 추가하라고만 이야기하는 것이 아니라, 실제로 이 클래스가 필요할 때 그 정의를 함께 제공하겠노라고 약속**하는 것이다.

따라서, C.h에 대응하는 .cpp 파일 중 전방선언한 클래스/구조체에 접근하는 모든 cpp 파일에서는 반드시 #include "A.h" 구문을 선언해야 한다.

#### 전방 선언만으로 해결할 수 없다.

위에서 볼 수 있지만, 단순히 전방선언을 한다고 해서 구현이 필요없는 것은 아니다. 단순하게 이름만 참조하는 경우가 아니라면 전방선언으로 해결할 수 없다. 조금 더 구체적으로 살펴보자.

###### 타입 동의어는 전방선언할 수 없다.

string이 그런 경우다. 실제로는 basic\_string<char>를 typedef한 것인데, string이란 class는 존재하질 않으니 전방 선언이란 것이 존재할리 없다.

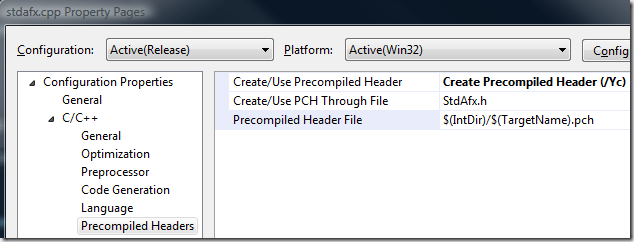
precompiled header

참고: <http://minjang.egloos.com/1956289>

윈도우 프로그래머라면 **precompiled header**가 익숙할 것이다. ***stdafx.h*** 바로 그 녀석이다. C/C++ 컴파일러의 컴파일 단위는 무조건 .c/.cpp 하나씩이다. 그래서 개개의 .c/.cpp 파일을 컴파일할 때 #include를 하는 모든 헤더를 당연히 매번 파싱 해야한다. Precompiled header는 이런 비용을 줄이기 위해 나온 것으로 **특정 헤더 파일을 미리 파싱한 결과물을 .pch 파일로 덤프시킨다.** 그리고 다른 파일들은 이 .pch를 단순히 사용함으로써 중복된 파싱 작업을 없앨 수 있다.

VC++는 디폴트로 이 precompiled header를 사용한다. Precompiled header가 작동하는 원리는 다음과 같다.

1. 덩치가 크거나 거의 바뀌지 않는 정의 같은 것을 모두 stdafx.h에 넣는다.
2. stdafx.cpp는 /Yc "stdafx.h" 옵션을 줘서 컴파일 한다. (Create)
3. 다른 .cpp 파일들은 /Yu "stdafx.h" 옵션을 준다. (Use)



이를 잘 사용하면 표준 라이브러리 헤더는 컴파일 시 큰 병목요인이 되지 않는다.

###### 컴파일 도중에 객체의 크기를 알아야 한다.

사용자 정의 객체의 크기는 정의를 보지 않는다면 알 수가 없다. 이 객체를 위한 공간을 얼마나 할당할까? 컴파일러는 모른다.

#### 전방 선언 + 포인터

자바 같은 경우는 모든 클래스가 포인터로 정의되어 있다. 이는 컴파일러 입장에서 포인터의 크기만큼만 컴파일시키면 되므로 세부사항을 알 필요가 없다. 이를 이용한 설계가 pimpl관용구이다. 이전에서 살펴본 바가 있다. 다음과 같다.

*//전방 선언을 합니다. 안에 구현내용은 다른 곳으로 구분지어야 합니다.*

**class** **ArcherImpl**;

*//클래스 안에서 사용하는 것들에 대한 전방 선언*

**class** **Skill**;

*//이제 이 파일만으로 컴파일이 가능합니다.*

**class** **Archer**{

**public**:

Archer();

void move() **const**;

void attack() **const**;

string info() **const**;

**private**:

*//구현 세부사항의 포인터만 가지고 있습니다.*

shared\_ptr<ArcherImpl> pImpl;

};

**‘한쪽은 인터페이스만 제공하고(위의 코드같이) 또 한쪽은 인터페이스의 구현을 맡도록 한다.’** *이는* ***정의부****에 대한 의존성을* ***선언부****에 대한 의존성으로 바꾸는* ***컴파일 의존성을 최소화하는 핵심****이다.*

1. // 전방선언
2. class Archer;
3. class RangeAttack
4. {
5. public:
6. // 포인터 형식으로 Archer 이름만 참조
7. RangeAttack(const **Archer\*** archer);
8. // 참조 형식으로 Archer 이름만 참조
9. RangeAttack(const **Archer&** archer);

#### 할 수 있으면 클래스 정의 대신 클래스 선언에 최대한 의존하도록 만듭시다.

함수를 선언할 때에 사용한 객체의 정의는 필요없다. 전방 선언으로 클래스의 정의 없이도 컴파일이 가능하게 한다.

**매개변수나 리턴 타입**을 위한 이름만 참조할 경우

이 경우 컴파일러가 Archer 클래스의 크기를 알아야 한다고 생각할 수 있지만,

사실 메써드를 구현하는 코드와 그 메써드를 호출하는 코드에서만 클래스의 크기를 요구한다.

1. // 전방선언
2. class Archer;
3. class RangeAttack
4. {
5. public:
6. void SetArcher(**Archer** archer);
7. **Archer** GetArcher() const;
8. }

#### 선언부와 정의부에 대해 별도의 헤더 파일을 제공합시다.

선언부만의 헤더 파일과 정의부만의 헤더 파일을 별도로 제공합니다. c++ 라이브러리의 <iosfwd>는 입출력 관련 함수들의 선언만 모아둔 것이다. 선언부의 헤더파일을 만드는데 <iostream>의 내용이 필요하다면 사용할 수 있겠죠? 선언부만의 헤더파일의 예시가 되며 여기에는 템플릿에 대한 선언도 구경할 수 있다.

**namespace** std {

**template**<**class** **charT**> **struct** char\_traits;

**template**<> **class** **char\_traits**<char>;

**template**<> **class** **char\_traits**<char16\_t>;

**template**<> **class** **char\_traits**<char32\_t>;

**template**<> **class** **char\_traits**<wchar\_t>;

**template**<**class** **T**> **class** **allocator**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_ios**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_streambuf**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_istream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_ostream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_iostream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT>, **class** **Alloc**=allocator<charT> >

**class** **basic\_stringbuf**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT>, **class** **Alloc**=allocator<charT> >

**class** **basic\_istringstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT>, **class** **Alloc**=allocator<charT> >

**class** **basic\_ostringstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT>, **class** **Alloc**=allocator<charT> >

**class** **basic\_stringstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_filebuf**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_ifstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_ofstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **basic\_fstream**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **istreambuf\_iterator**;

**template** <**class** **charT**, **class** **traits**=char\_traits<charT> > **class** **ostreambuf\_iterator**;

**template**<**class** **stateT**> **class** **fpos**;

**typedef** basic\_ios<char> ios;

**typedef** basic\_streambuf<char> streambuf;

**typedef** basic\_istream<char> istream;

**typedef** basic\_ostream<char> ostream;

**typedef** basic\_iostream<char> iostream;

**typedef** basic\_stringbuf<char> stringbuf;

**typedef** basic\_istringstream<char> istringstream;

**typedef** basic\_ostringstream<char> ostringstream;

**typedef** basic\_stringstream<char> stringstream;

**typedef** basic\_filebuf<char> filebuf;

**typedef** basic\_ifstream<char> ifstream;

**typedef** basic\_ofstream<char> ofstream;

**typedef** basic\_fstream<char> fstream;

**typedef** basic\_ios<wchar\_t> wios;

**typedef** basic\_streambuf<wchar\_t> wstreambuf;

**typedef** basic\_istream<wchar\_t> wistream;

**typedef** basic\_ostream<wchar\_t> wostream;

**typedef** basic\_iostream<wchar\_t> wiostream;

**typedef** basic\_stringbuf<wchar\_t> wstringbuf;

**typedef** basic\_istringstream<wchar\_t> wistringstream;

**typedef** basic\_ostringstream<wchar\_t> wostringstream;

**typedef** basic\_stringstream<wchar\_t> wstringstream;

**typedef** basic\_filebuf<wchar\_t> wfilebuf;

**typedef** basic\_ifstream<wchar\_t> wifstream;

**typedef** basic\_ofstream<wchar\_t> wofstream;

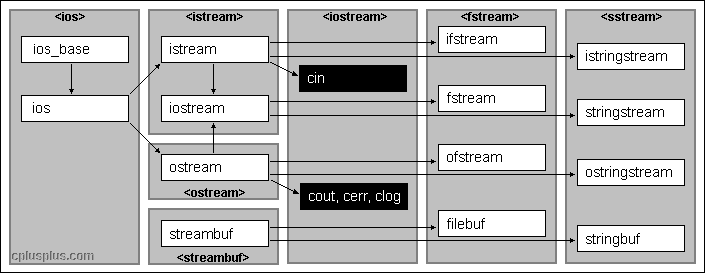
**typedef** basic\_fstream<wchar\_t> wfstream;

**typedef** fpos<mbstate\_t> streampos;

**typedef** fpos<mbstate\_t> wstreampos;

}

참고: <iostream>내부의 상속 관계입니다.

**

###### **[방법1] 핸들 클래스**를 이용합시다.

아까 선보인 pimpl관용구를 사용하는 Archer와 같은 클래스를 핸들 클래스라고 한다. 어떤 객체가 가지고 있는 객체에 행위를 전해준다. 이런 관계를 합성되었다 하며, 이러한 관계로 전해주는 행위를 위임이라고 한다. 컴파일 의존도에 유리할 뿐만 아니라, 캡슐화에도 상속보다 유리하여 고려해볼 가치가 있다. – [항목38] 참고

다음과 같이 사용한다.

*//Archer.h*

**class** **ArcherImpl**;

**class** **Archer**{

**public**:

Archer(**const** string& n, int p);

void move() **const**;

void attack() **const**;

string info() **const**;

**private**:

*//구현 세부사항의 포인터만 가지고 있습니다.*

shared\_ptr<ArcherImpl> pImpl;

};

*//ArcherImpl.h*

**class** **Skill**;

**class** **ArcherImpl**{

**public**:

ArcherImpl(**const** string& n, int p);

string name() **const**;

**private**:

string name;

int power;

vector<Skill> skills;

};

*//Archer.cpp*

*//선언부의 헤더파일이 필요합니다.*

#include *"Archer.h"*

#include *"ArcherImpl.h"*

Archer::Archer(**const** string& n, int p)

: pImpl(**new** ArcherImpl(n, p))

{}

string Archer::info() **const** {

**return** pImpl->name();

}

###### **[방법2] 인터페이스 클래스**를 이용합시다.

순수 가상 함수로 만들어진 **인터페이스 클래스 + 펙토리 함수**(**[항목 13]**)를 만들어서 설계하자.

*//순수 가상 함수*

*//인터페이스 클래스입니다. 그 자체로 객체를 만들 수 없습니다.*

**class** **Archer**{

**public**:

**virtual** ~Archer();

**virtual** void move() **const** = 0;

**virtual** void attack() **const** = 0;

**virtual** string info() **const** = 0;

***//객체를 못 만드므로 객체 생성 수단: static을 이용할 수 있습니다.***

**static** shared\_ptr<Archer> create();

};

*//모든 구현은 인터페이스의 명세만 물려받은 파생 클래스에서 됩니다.*

**class** **ArchMage** : **public** Archer{

**public**:

ArchMage() {}

**virtual** ~ArchMage() {}

void move() **const**;

void attack() **const**;

string info() **const**;

**private**:

string name;

int power;

};

*//static 함수 정의합니다.*

shared\_ptr<Archer> Archer::create(){

*//Archer은 생성할 수 없으므로 파생된 구체 클래스를 생성합니다.*

*//[의문]이 구체 클래스의 구현은 파생 클래스에 있습니다?*

**return** shared\_ptr<Archer>(**new** ArchMage());

}

인터페이스 클래스를 사용하여 선언과 정의를 분리할 수 있다. 인터페이스에는 정의가 없다. 그러므로 선언과 정의를 분리하기에 적합할 것이다. 그런데 해당 인터페이스로 객체를 만들어 의존도를 최소화시킬 목적인데 그 자체로는 객체를 만들 수 없다.

이럴때 유용한 것이 펙토리 함수다. 펙토리함수는 기본 클래스 타입의 생성을 파생 클래스에서 담당한다. 즉, 기본 클래스에서 파생된 모든 것들을 이 펙토리 함수로 기본 클래스 타입으로 통일시켜 만들어준다는 것이다. 결국, 위의 인터페이스로도 객체를 만들 수 있게 되었다.

위의 파생 클래스(구체 클래스 역할)은 구현부에 들어가게 되어 선언부에서는 필요 없습니다. 객체를 생성할 때에는 구체 클래스를 생성하여 사용하게 됩니다.

***컴파일 의존성을 최소화하는 작업의 배경이 되는 가장 기본적인 아이디어는 ‘정의’대신에 ‘선언’에 의존하게 만들자는 것입니다. 이 아이디어에 기반한 두 가지 접근 방법을 핸들 클래스와 인터페이스 클래스입니다.***

***라이브러리 헤더는 그 자체로 모든 것을 갖추어야 하며 선언부만 갖는 형태여야 합니다. 템플릿 유무 상관없이 모두 적용되어야 합니다.***

[항목32] public상속: is-a

public상속이 갖는 의미를 정확하게 해야 한다.

# public상속의 의미

***“is – a”: ~는 ~의 일종이다.***

# is-a의 의미에 대한 고민

~의 일종이다라는 직관적인 의미와 어긋나는 경우가 있다.

#### 명확치 않은 자연어로 인한 오해

일반적으로 새의 날 수 있다. 하지만 모두 그런 것은 아니다. 예를 들어 어떠한 클래스가 새의 클래스를 상속한다면 “새의 일종이다.”라는 의미를 가지지만 날 수 있다는 능력을 부여할 때 조심해야 한다.

void error(**const** string& s){

cout << s << endl;

}

**class** **Archer**{

**public**:

**virtual** void bow();

};

**class** **ArchMage** : **public** Archer{

**public**:

*//아크메이지는 bow()라는 능력이 없습니다.*

**virtual** void bow(){

***//런타임에서 에러를 출력합니다.***

error("you don't have bow.");

}

***//하지만 아예 bow함수를 없애버리면 컴파일 타임에 에러 검출***

***//항상 런타임보다는 컴파일타임에 에러를 검출하자.***

};

#### ~의 일종이지만 서로 다른 특징이 있다.

직관적인 의미와 달리 완전히 포함되지 않는다는 것이다. 정사각형과 직사각형이 그 예시가 되겠다. 분명 정사각형은 직사각형의 일부지만 직사각형의 어떠한 성질은 정사각형에 적용되지 않으므로 우리가 생각하는 public상속이라고 볼 수 없다. 예를 들어, 직사각형은 넓이를 변경할 때, 직사각형에서 사용하는 넓이 변경은 사용할 수 없다. “***public상속****은 기본 클래스 객체가 가진* ***모든 것****들이 파생 클래스 객체에도 적용된다고 단정하는 상속이다.”*

###### 단정문에 대하여

#include *<assert.h>*

void print\_number(int\* myInt) {

***//단정문은 제시한 문장이 실패할 일이 없다고 확신하는 것입니다.***

assert(myInt != NULL);

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

}

int main()

{

int a = 10;

int \* b = NULL;

int \* c = NULL;

b = &a;

print\_number(b);

print\_number(c);

**return** 0;

}

***public상속의 의미는 “is-a(~는 ~의 일종이다)”입니다. 기본 클래스에 적용되는 모든 것들이 파생 클래스에 그대로 적용되어야 합니다. 왜냐하면 모든 파생 클래스 객체는 기본 클래스 객체의 일종이기 때문입니다.***

[항목33] 이름의 유효범위

함수나 변수의 이름의 유효범위를 고려하여 동작을 이해해야 합니다.

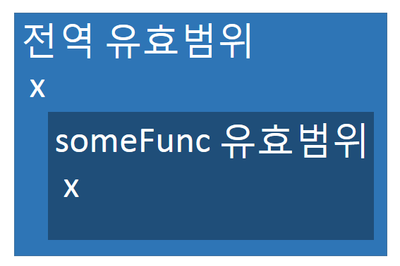
# 유효범위(local scope)

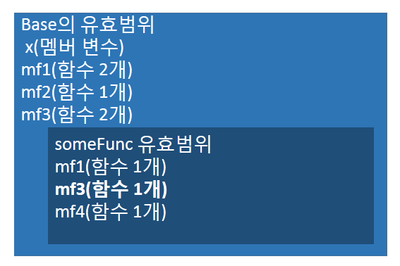
어떠한 이름이 유효한 범위를 말한다. 함수 내일 수도 있으며 전역일 수도 있다. 그리고 기본 클래스와 파생 클래스의 유효범위도 존재한다. 곧, 어떠한 이름의 정보는 유효범위 안에 있는 것이다. 그리고 범위에서 이름을 찾아야 할텐데 순서는 다음과 같다.

지역🡪현재 클래스🡪기본 클래스🡪네임 스페이스🡪전역

만약 이 유효범위가 겹치게 된다면, 가까운 유효범위를 먼저 탐색하고 없다면 위의 순서대로 찾을 것이다. 하지만 그 안에서 찾으면 오버로드든 뭐든 더 이상 찾지 않는다.

# 이름이 가려질 수 있다.

someFunc에서 전역 x는 없는 것이다.

Base의 mf3는 없는 것이다.

상속에서도 마찬가지인 것이다. 물론 파생 🡪 기본 클래스 순으로 탐색한다.

#### [문제] 가려진 이름

유효범위 밖에 함수의 이름은 가려질 수 밖에 없다.

**class** **Archer**{

**public**:

**virtual** void attack();

void move();

*//move가 오버로드되었군요.*

void move(double);

void eat();

};

***//Archer을 public상속하므로***

***//BM은 A의 일부이며 모든 기능을 사용해야 합니다.***

**class** **BladeMaster** : **public** Archer{

**public**:

**virtual** void attack();

*//이 함수가 있다면 move(double)은 실행될 수 없다.*

*//유효범위가 가까이 있는 이름이 가려버리기 때문이다.*

void move();

void strongBlade();

};

가까운 유효범위 밖에 있는 함수는 호출할 수 없는 괴이한 현상이 생기는 것이다. 여기서 진짜 문제는 public상속이라는 점이다. 모든 함수를 그대로 사용해야 의미를 그대로 거스른다. 물론 가상함수는 제외이다. 적절하게 유효범위를 바꾸는 방법이 필요하다.

#### [해결] 사용할 유효범위를 정한다.

이는 위에서 **[항목25]**에서도 사용했었다. using을 선언함으로써 원하는 범위를 지정할 수 있다. 이름 가리기를 무시해버리는 방법을 다음에서 살펴보자.

**class** **BladeMaster** : **public** Archer{

**public**:

*//이 문장은 Archer의 함수도 볼 수 있게 해준다. 오버로드 가능*

*//물론 파생 클래스의 move를 우선 사용한다.*

**using** Archer::move;

**virtual** void attack(){

*//파생 클래스에서 기본 클래스의 이름이 같은 함수를 쓰고 싶다면*

*//아래처럼 기본 클래스의 함수를 직접 호출한다.* ***전달 함수가*** *된다.*

Archer::attack();

}

void move();

void strongBlade();

};

***파생 클래스의 이름은 기본 클래스의 이름을 가립니다. public상속에서는 이런 이름 가림 현상은 바람직하지 않습니다.***

***가려진 이름을 다시 볼 수 있게 하는 방법으로, using선언 혹은 전달 함수를 쓸 수 있습니다.***

[항목34] public상속에서 함수 의미

인터페이스 상속과 구현 상속의 차이를 제대로 이해하고 구별하자.

# 가상 함수(virtual function)

기본 클래스로부터 상속받는 파생 클래스가 있을 때, 기본 클래스의 함수를 물려받거나 그 형태를 그대로 쓰고 싶은 경우에 가상 함수를 정의한다. 이것은 일반 함수와 어떻게 다른가? 기본적인 원리인 함수를 연결하는 과정, 바인딩에서 출발한다.

#### 바인딩(binding)

변수와 어떠한 속성을 연결하는 일. 변수에 값을 넣을 수도 있으며 함수를 지정(함수 호출 시, 그 함수의 정보가 있는 메모리로 연결시켜 주는 일)할 수도 있다.

###### **정적 바인딩(=선행 바인딩)**

**컴파일 타임**에 일어나며 변하지 않는다. 변수에 메모리 주소나 값을 집어넣는 과정, 함수 호출 부분과 그 함수 내용과 연결되는 과정이 정적 바인딩이라고 볼 수 있다.

###### **동적 바인딩(=지연 바인딩)**

**런타임**에 일어나므로 실행 중에 결정된다. 함수 호출 부분이 어떤 함수를 호출할 지 미리 결정하지 않고 런타임에 결정하여 임의의 함수 내용과 연결하는 과정을 동적 바인딩이라 한다. 동적 바인딩은 가리키는 포인터가 하나를 더 거쳐서 연결하는 과정 때문에 메모리와 동작 시간이 더 필요하다.

링크 바인딩

역시 동적 바인딩인데, 런타임 중 일어나는 바인딩뿐만 아니라, 링크 과정에서 생기는 바인딩도 동적 바인딩으로 분류한다. 어떤 것이 그러한가? extern과 static변수를 비교해보자. **extern은 다른 파일끼리도 변수를 공유한다. 링크 과정에서 속성을 결정하기에 가능한 일이다. 하지만static은 하나의 파일에서만 유효하다. 이 또한 정적으로 바인딩하기에 가능한 일이다.**

#### 가상 함수에서의 바인딩

우리가 함수를 사용할 때에는 어떠한 메모리와 연결되게 된다. 이 연결은 함수 포인터로 이루어지며 이것이 바인딩이 되는데 일반적으로 사용하는 함수는 정적 바인딩이다. 정해져 있는 것이다. 하지만 모두 그래서는 안 된다. 왜냐하면 가상 함수는 대상이 어떠한 타입을 가리키느냐에 따라 호출될 함수가 달라지기 때문이다. 그러므로 **가상 함수**는 런타임에 적절한 함수를 파생 클래스와 그가 상속하는 클래스에서 찾아서 연결되어야 한다. 상황이 이러므로 **동적 바인딩**이 필요한 셈이다.

#### 바인딩 과정

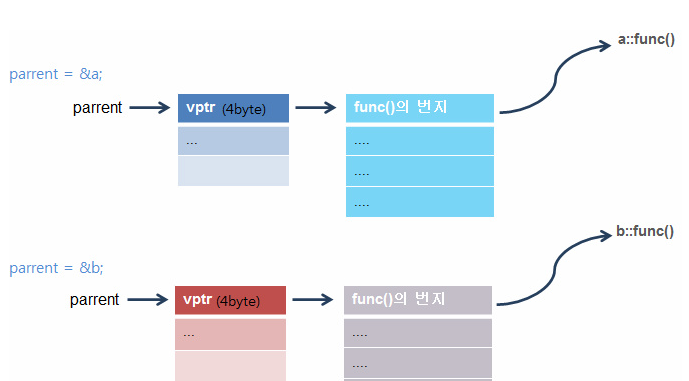
우리가 무언가의 함수를 호출할 때, 어떠한 함수를 연결할 지 결정하는 과정이다. 여기서는 클래스 내의 가상 함수를 찾을 것이다. 이를 가리키기 위해서 2가지 재료가 필요하다.

###### vptr(가상 함수 포인터)

동적 연결을 담당하는 포인터다. 뒤에 나올 vtbl의 시작 주소를 가리킬 것이다. ***이는 객체에 들어있으며 가상 함수를 포함한 객체를 생성하면 함께 포함됨을 기억해야 한다.*** 4byte가 필요하다.

###### vtbl(가상 함수 테이블)

이곳에는 가상 함수들의 포인터 주소가 배열로 담겨져있다. 이는 객체 밖에 있다. 이들의 연결 관계는 아래와 같다.

과정은 다음과 같다.  
1. 가상 함수를 1개라도 가진 객체는 vptr을 가지고 태어납니다. 그리고 vptr은 vtbl을 가리키고 있습니다.  
**2. 함수가 호출되면 vptr은 vtbl에서 적절한 함수를 고릅니다. 이 과정은 일반 함수 호출 과정에는 없습니다.**  
3. 함수를 호출하게 됩니다.

가상 함수 테이블(vtable)은 클래스당 1개만 존재한다.

The vtable contains function pointers that point to the virtual functions in that class. **It’s important to note that there can only be one vtable per class, and all objects of the same class will share the same vtable.** This means that in the example above, the Animal and Tiger classes will each have their very own vtable, and any objects of the Animal or Tiger classes will use their respective class’s vtables.

가상함수가 1개라도 존재한다면, 가상함수 포인터가 생긴다. 이 포인터가 가상함수테이블을 가리키므로 가상함수 테이블도 함께 생기는 셈이다. **이는 동일 클래스에서 생성된 모든 인스턴스가 공유한다는 의미다.** 반대로 포인터가 사라지면 가상함수 테이블도 있을 이유가 없다. 이는 std::shared\_ptr을 연상시키지만, 동적으로 일어나지는 않으므로 신경쓰지 않아도 될 듯 하다.

# public상속에서 사용할 함수들

public상속은 우리가 잘 알다시피 is-a관계를 뜻한다. 이 관계가 적절하려면 기본 클래스에 적절한 함수 선언이 필요하다.

#### 순수 가상 함수

그 함수의 **인터페이스**만을 물려줍니다. “이 함수는 여러분이 직접 제공하시오. 난 아무런 생각이 없다네.”라고 말하는 것과 같다네요.

#### 비순수 가상 함수

그 함수의 **인터페이스**와 **기본 구현**까지 물려받는다. 그러므로 재정의를 할 수도 있으며 하지 않아도 기본 구현을 사용할 수 있다. 재정의하게 되면 파생 기능만 사용하며 기본 구현도 사용하려면 별도 호출이 필요하다. “이 함수는 여러분이 지원해야 합니다. 굳이 새로 만들 생각이 없다면 그냥 기본 함수를 쓰세요.”

파생 클래스에서 기본 클래스를 사용하려면

void BladeMaster::attack(Archer& target) **const**{

**Archer::attack(target);**

cout << "power blade!" << endl;

}

###### 가상 함수를 쓰지 않고 그냥 죄다 기본 함수로 덮어버리면 어떨까?

가상 함수는 설계에서 무슨 의미를 갖는 것일까? 이런 궁금증을 위해 둘을 비교한 코드를 제시한다.

class Archer

{

public:

virtual void attack(){ std::cout << "shot\n"; }

void doubleShot() { std::cout << "shotshot\n"; }

};

class BladeMaster : public Archer

{

public:

virtual void attack(){ std::cout << "blade\n"; }

void doubleShot() { std::cout << "no data\n"; }

};

int main()

{

Archer\* a = new Archer;

Archer\* bm = new BladeMaster;

a->attack();

bm->attack();

a->doubleShot();

bm->doubleShot();

return 0;

}

attack은 실제 객체의 기능을 수행한다. 그러므로 a와 bm은 다르다. 하지만 비가상함수인 doubleshot은 a와 bm이 똑 같은 작업을 수행한다. 여기서 다형성의 의미가 두드러진다. **똑 같은 기능을 사용하면 같은 결과를 보여야 한다. 하지만 virtual은 같은 기능을 사용해도 다른 결과를 보여준다.**

#### 비가상 함수

파생과 상관없이 변하지 않는 동작이며 **인터페이스와 더불어 그 함수의 필수적인 구현**을 물려받게 합니다. “이 객체에서 파생된 모든 객체는 식별자를 내어 주는 함수를 갖게 되겠지.”

**class** **Archer**{

**public**:

Archer(**const** string& n) : name(n) {}

*//****비순수 가상 함수****로 선언합니다.*

**virtual** void attack(Archer& target) **const**;

*//함부로 base함수를 물려받지 못하게 하려면 순수 가상 함수로 선언*

**virtual** void move() **const** = 0;

*//비가상 함수입니다.*

**const** string getName() **const** { **return** name; }

**private**:

string name;

};

void Archer::attack(Archer& target) **const**{

cout << name << " attack to " << target.getName() << endl;

}

*//순수 가상 함수에 정의를 넣음으로써 파생에서 반드시 재정의를 해야하군요.*

void Archer::move() **const** {

cout << "normal move" << endl;

}

*//Archer을 public상속하므로* ***Archer의 기능을 온전히 사용해야 합니다.***

**class** **BladeMaster** : **public** Archer{

**public**:

BladeMaster(**const** string& n) : Archer(n) {}

*//비순수 가상 함수 재정의를 해도 되고 안 해도 그만*

void attack(Archer& target) **const**;

*//순수 가상 함수는 반드시 재정의를 해야합니다.*

void move() **const**;

*//비가상 함수는 재정의하는 것이 아니라 반드시 포함되어 있는 기능이지요.*

};

void BladeMaster::attack(Archer& target) **const**{

cout << "power blade!" << endl;

}

void BladeMaster::move() **const** {

cout << "faster" << endl;

}

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

**public**:

DarkKnight(**const** string& n) : Archer(n) {}

*//비순수 가상 함수 attack은 재정의하지 않아도 됩니다.*

void move() **const**;

};

void DarkKnight::move() **const** {

cout << "shadow" << endl;

}

int main(){

BladeMaster bm("YTK");

DarkKnight dk("SS");

bm.attack(dk); *// 재정의한 함수 결과만 호출*

bm.move();

**return** 0;

}

***인터페이스 상속은 구현 상속과 다릅니다. public상속에서, 파생 클래스는 항상 기본 클래스의 인터페이스를 모두 물려받습니다.***

***순수 가상 함수는 인터페이스 상속만을 허용하며 단순 가상 함수는 기본 구현의 상속도 가능하도록 지정합니다.***

***비가상 함수는 인터페이스 상속과 더불어 필수 구현의 상속도 가하도록 지정합니다.***

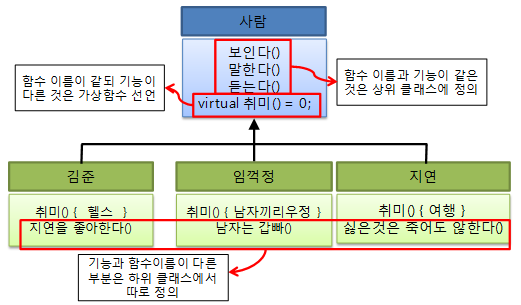
[항목35] 가상함수를 대체할 무기

가상 함수를 대신 쓸 것을 준비해 두는 자세, 그 생각하는 힘이 필요하다. 가상 함수를 사용하기 전에 반드시 아래 4가지, 혹은 그 이상을 고려해야 한다.

# 비가상 인터페이스 관용구(non-virtual interface, NVI)

템플릿 메서드라 불리는 고전 디자인 패턴을 C++식으로 구현한 것

#### 템플릿 메서드 패턴



behavioral pattern이라고 부르기도 한다. C++의 template와는 아무런 상관이 없다. 상위 클래스, 즉 base 클래스는 기본적인 메서드만 제공하고 **base 클래스로부터 파생된 클래스에서 해당 메서드를 제정의 해서 사용한다.** 만약 해당 메서드를 재정의 하지 않는다면, 기본 메서드를 그대로 사용한다.

**class** **Base**{

**public**:

void execute(){

a(); ph1(); c(); ph2(); e();

}

**private**:

void a() { cout << "a "; }

void c() { cout << "b "; }

void e() { cout << "c "; }

**virtual** void ph1() = 0;

**virtual** void ph2() = 0;

};

*//기본 클래스의 private멤버 함수를 재정의합니다.*

**class** **One** : **public** Base{

void ph1(){ cout << "d "; }

void ph2(){ cout << "e "; }

};

**class** **Two** : **public** Base{

void ph1(){ cout << "2 "; }

void ph2(){ cout << "4 "; }

};

int main(){

Base \*array[] = { &One(), &Two() };

**for** (int i = 0; i < 2; i++){

*//다양한 타입에 대응하여 실행할 수 있다.*

array[i]->execute();

cout << '\n';

}

}

**같은 타입**의 객체를 바탕으로 **같은 동작(execute)**에 대해 **다른 행동**을 할 수 있다. **일반적인 가상 함수와 다른 점은, 파생 클래스에서 재정의한 가상 함수를 호출하는 것이 아닌, 기본 클래스의 비가상 함수를 호출하며, 그 안에서 파생 클래스만의 기능인 재정의한 가상 함수를 호출한다.** 이러한 점을 이용하여 가상 함수를 대신하여 런타임에 적절한 알고리즘을 선택한다.

#### NVI사용법

위의 템플릿 메서드 패턴가 그대로 사용된다. **동작은 비가상 함수**로 변하지 않으며 **실질적인 행위가 private가상 함수**로 구현된다.

**class** **Archer**{

**public**:

*//기본적인 가상함수*

*//virtual void upgrade(int level) const;*

*//아래의* ***비가상*** *함수와* ***private가상*** *함수가* ***NVI구조****입니다.*

void upgrade(int level) **const**; *//이 함수가* ***wrapper****가 됩니다.*

**private**:

**virtual** void doUpgrade(int level) **const**;

**virtual** bool isAvailUpgrade(int level) **const**;

};

void Archer::upgrade(int level) **const**{

*//업그레이드 수행 전의 사전 작업을 수행합니다.*

*//업그레이드 시행의 조건에 만족한지 확인 등의 작업이 있습니다.*

**if** (isAvailUpgrade(level))

*//사전 작업 수행 후, 실제 작업을 수행합니다.*

doUpgrade(level);

*//사후 작업을 수행합니다.*

*//업그레이드 후에 수행할 작업, 사용한 객체를 닫는 등의 작업이 있습니다.*

}

upgrade라는 함수를 각 파생 클래스마다 다르게 만들고 싶을 때, 기본적인 virtual 함수를 사용할 수 있다. 하지만 NVI관용구를 이용하면 실제 사용하는 함수는 비가상 함수로 두고 가상 함수를 private함수에 옮긴다. 그리고 비가상 함수에서 이 private함수를 수행하는 것이다. 그러면 나는 비가상 함수의 호출로 업그레이드를 수행하는 것이다.

#### NVI는 왜 사용하는가?

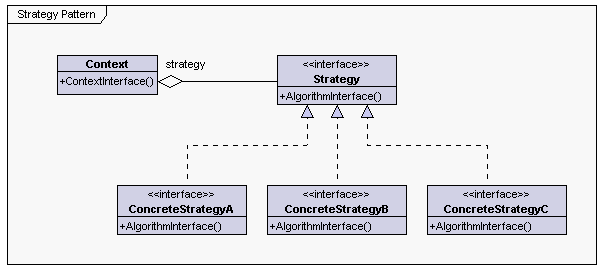
**NVI의 이점**  
1. 위의 코드에도 있듯이, 사전 작업과 실제 작업, 사후 작업을 하나로 묶을 수 있다. 실제로 사용자가 업그레이드 함수를 사용한다고 할 때, 미리 확인해야 하는 사항이 있다고 하자. 그러면 업그레이드 함수를 사용하기 전에 확인 작업의 함수(isAvailUpgrade)를 수행해야 하는데 이것을 깜빡할 수도 있다. 그러기에 하나의 함수로 묶어주면 깔끔한 구현을 만들 수 있다.  
2. 이는 곧 사용자에게 제공해야 할 인터페이스와 개발자가 적용해야 할 인터페이스를 구분짓는 것입니다. 사용자 입장에서는 하나의 함수만 불러오면 업그레이드를 수행할 수 있고, 개발자는 수정하기가 용이해집니다.

#### private가상 함수

private가상 함수를 파생 클래스에서 재정의한다. 가상 함수를 재정의하는 일은 어떤 동작을 **어떻게 구현할 것인가**를 지정하며, 가상 함수를 호출하는 일은 그 동작이 **수행될 시점**을 지정하는 것이다. 이 둘은 관계가 없으며, **어떻게 구현할지**는 파생 클래스가 담당하며 **수행될 시점**은 기본 클래스가 담당한다.

# 전략 패턴을 이용한 설계

#### 전략 패턴이란?

동적으로 알고리즘을 교체할 수 있는 구조이다. 상황에 따라서 내가 원하는 함수나 객체를 사용하여 동작을 구현할 수 있다. 코드는 이후에 차례로 제시한다.  


#### 고전적인 전략 패턴

가장 기본적인 전략 패턴을 사용한다고 생각하면 되겠다. 클래스에 가상 함수가 존재하며 다양한 클래스를 위 그림처럼 포함한다고 생각할 수 있다.

**class** **HealthCalcClass**{

**public**:

HealthCalcClass(int a) : armor(a) {}

**virtual** int calc(**const** Archer& a) **const** { **return** 1; }

**private**:

int armor;

};

*//다양한 체력 계산 클래스를 만듭니다.*

*//그 안의 함수는 파생 클래스에서 조절하도록 가상 함수로 만듭니다.*

HealthCalcClass normalHealthCalc(1);

HealthCalcClass masterPlanHealthCalc(1000);

**class** **Archer**{

**public**:

***//함수 객체의 포인터를 받아서 그 객체에서 함수를 실행하게 합니다.***

***//이는 다양한 알고리즘을 제공하면서 객체 밖에서 체력 계산을 합니다.***

**explicit** Archer(HealthCalcClass \*hcc = &normalHealthCalc)

: pHealthCalc(hcc) {}

int healthValue() **const** { **return** pHealthCalc->calc(\***this**); }

**private**:

HealthCalcClass \*pHealthCalc;

};

다양한 객체의 포인터를 품는 가장 표준적인 전략 패턴 설계이다.

#### 함수 포인터로 구현한 전략 패턴

함수 포인터에 관해서는 *typedef사용법*을 참조한다. 이는 체력 계산 함수를 클래스 밖에서 가져와 대입하여 사용하는 구조이다. 원하는 함수를 런타임에 가져와 바꿀 수 있으므로 같은 객체에 대하여 다양한 구현이 가능하다.

*//전방 선언*

**class** **Archer**;

*//외부에 채력 계산 함수가 정의되어 있습니다.*

***//내가 원하는 체력계산함수를 골라 쓸 수 있다.***

int defaultHealthCalc(**const** Archer& a);

int shieldHealthCalc(**const** Archer& a);

int curseHealthCalc(**const** Archer& a);

**class** **Archer**{

**public**:

*//이제 함수를 객체처럼 사용할 수 있습니다.*

**typedef** int(\*HealthCalcFunc)(**const** Archer& a);

***//나의 healthFunc함수를 지정할 수 있습니다.***

**explicit** Archer(HealthCalcFunc hcf = defaultHealthCalc) : healthFunc(hcf) {}

**private**:

HealthCalcFunc healthFunc;

};

###### 전략 패턴의 장점

같은 캐릭터 타입의 객체라도 체력 계산을 다르게 할 수 있다. 상황에 따라서 런타임에 변경도 가능하다.

###### 전략 패턴의 단점

체력을 계산할 때, 객체의 private멤버로는 접근할 수 없다. 이렇게 되면 어쩔 수 없이 캡슐화를 약화시켜야 한다.

#### std::function으로 구현한 전략 패턴

사실 이것은 위의 함수 포인터와 거의 유사하다. 하지만 더 일반화된 함수 포인터를 물게 됨으로써 여러 가지 융통성을 발휘할 수 있다. 아래에서 사용법을 설명하면서 이에 대한 구현도 함께 예시로써 제시한다.

###### std::function에 대하여

**std::function사용법**

하나의 객체로서 역할한다. 위에서 함수가 하나의 객체처럼 작동한다고 되어 있는데 이는 실제로 객체가 된다. 함수 객체라고도 하는데 원리는 이렇다. 하나의 함수를 인자로 받아서 이를 감싸서 wrapper가 된다. 이는 곧 **‘함수를 품은 객체’**가 되는 것이다. 결국 객체가 함수처럼 작동하는 함수 포인터를 함수 객체로 만들어주는 것과 동일하다.

*//헤더 파일이 필요합니다.*

#include *<functional>*

*//함수 객체를 사용합니다.*

**typedef** **function**<int(**const** Archer&)> HealthCalcFunc;

이 코드는 그 위에서 선보인 함수 포인터와 동일한 동작을 한다. function의 타입에 2가지가 있는데 int는 객체화하고자 하는 함수의 반환 값이며 그 뒤에는 매개변수가 이어진다.

**std::function이점**

그러면 함수 포인터대신 객체를 사용하여 얻을 수 있는 이점은 무엇인가? 융통성이다. **std::function계열의 객체는 함수호출성 객체(함수 포인터, 함수 객체, 멤버 함수 포인터)를 가질 수 있고**, 이로써 같은 function객체는 위의 3가지 객체를 모두 받을 수 있으므로 여러 가지 객체를 받을 수 있는 것이다. 게다가 반환 값이나 매개 변수가 달라도 암시적 변환이 가능하면 받을 수 있다. **이것은 호환되는 모든 객체는 받을 수 있다는 의미다.** 즉, 다음과 같이 사용할 수 있다.

*//전방 선언*

**class** **Archer**;

**class** **BladeMaster**;

*//내가 원하는 체력계산함수를 골라 쓸 수 있다.*

int defaultHealthCalc(**const** Archer& a);

int shieldHealthCalc(**const** Archer& a);

long long curseHealthCalc(**const** Archer& a);

long long swordCurseHealthCalc(**const** BladeMaster& a);

*//****함수 객체****도 고를 수 있습니다.*

**class** **holyHealthCalc**{

**public**:

int **operator**()(**const** Archer& a){ **return** 0; }

};

**class** **Archer**{

**public**:

*//typedef int(\*HealthCalcFunc)(const Archer& a);*

*//함수 객체를 사용합니다.*

**typedef** function<int(**const** Archer&)> HealthCalcFunc;

*//나의 healthFunc함수를 지정할 수 있습니다.*

**explicit** Archer(HealthCalcFunc hcf = defaultHealthCalc) : healthFunc(hcf) {}

void setHealthFunc(HealthCalcFunc hcf) { healthFunc = hcf; }

**private**:

HealthCalcFunc healthFunc;

};

**class** **BladeMaster** :**public** Archer{

};

int main(){

Archer a;

*//****함수 포인터***

a.setHealthFunc(shieldHealthCalc);

*//****함수 객체***

a.setHealthFunc(holyHealthCalc());

*//****반환 타입이 다른*** *함수 포인터*

a.setHealthFunc(curseHealthCalc);

*//****매개 변수도 암시적 변환****이 가능합니다.*

a.setHealthFunc(swordCurseHealthCalc);

***//위의 코드가 모두 가능케 함으로써 융통성을 발휘합니다.***

}

그럼 **멤버 함수 포인터**를 여러 개의 인자와 함께 사용하려면 어떻게 해야 할까? 아래의 bind함수를 이용해야 한다.

**std::bind()**

bind는 일련의 인자들을 함수에 바인딩시킨 function 객체를 반환하는 함수 템플릿이다. 즉, bind 함수는 함수의 포인터를 받고, 인자 목록을 전달받아 그것들을 시그너처로 하는 function 객체를 만들어 낸다.

void show\_text(**const** string& t)

{

cout << "TEXT: " << t << endl;

}

*// show\_text 함수에 "Bound function"이라는 문자열을 바인딩시킨*

*// function<void ()> 타입을 반환한다.*

function <void ()> f = std::bind(show\_text, "Bound function");

std::placeholders

위에서는 1개의 매개변수만 존재했는데 여러 여러 개의 인자가 있다면 혹은 어떤 인자는 고정하고 어떤 인자는 자유롭게 호출하고 싶다면 다음과 같이 **placeholders**를 이용한다.

**using namespace std::placeholders;**

void show(**const** string& a, **const** string& b, **const** string& c)

{

cout << a << "; " << b << "; " << c << endl;

}

int main()

{

*// y의 첫번째 인자는 show 함수의 세번째 인자*

*// y의 두번째 인자는 show 함수의 첫번째 인자*

*// 이렇게 인자가 서로 뒤바뀔 수도 있다.*

**auto** y = bind(show, \_3, \_1, \_2);

*// function z의 경우 show 함수의 첫번째 인자는 "hello"로* ***고정****된 상태*

*// z의 첫번째 인자는 show 함수의 세번째 인자*

*// z의 두번째 인자는 show 함수의 두번재 인자*

**auto** z = bind(show, "hello", \_2, \_1);

y("one", "two", "three"); //show(“three”, “one”, “two”)

z("one", "two"); // show(“hello”, “two”, “one”)

**return** 0;

}

이것을 이용하여 **멤버 함수**를 집어넣어보자.

일단 멤버 함수는 왜 bind를 사용해야 할까? 매개 변수가 2개 이상이기 때문이다. 객체가 들어가므로 객체를 함께 묶어주어야 하는데 이 역할을 하는 함수가 bind이다. 결국 위의 placeholders를 이용하여 응용할 수 있다. 다음과 같다.

*//bind의 \_1, \_2, \_3, ...\_N placeholder를 사용하기 위한 namespace*

**using** **namespace** std::placeholders;

**class** **shieldHealthCalc**{

**public**:

shieldHealthCalc(int sl = 1) : shieldLevel(sl) {}

float healthCalc(**const** Archer& a) {

cout << shieldLevel \* 0.5 << endl;

**return** shieldLevel \* 0.5;

}

**private**:

int shieldLevel;

};

**class** **Archer**{

**public**:

**typedef** function<int(**const** Archer&)> HealthCalcFunc;

void startHealthOperation(**const** Archer& target) { healthFunc(target); };

int getLevel() { **return** level; }

**private**:

HealthCalcFunc healthFunc;

int level;

};

int main()

{

Archer target;

shieldHealthCalc ophc(10);

*//멤버 함수를 집어넣으려면 객체와 함께 넣어주어야 하므로 매개 변수가 2개*

***//bind로 2개의 매개 변수를 묶어준다.***

Archer a(bind(&**shieldHealthCalc**::healthCalc, ophc, target));

*//startHealthOperation의 매개 변수가 어떻건 bind에 묶여있는대로 실행된다.*

a.startHealthOperation(target);

***//\_1을 이용하여 첫 번째 매개 변수는 내 마음대로 대입할 수 있다.***

Archer c(bind(&**shieldHealthCalc**::healthCalc, ophc, **\_1**));

c.startHealthOperation(target);

**return** 0;

}

**이로써 멤버 함수나 인자가 여러 개인 함수 또한 적절한 자유 인자 설정으로 std::function객체에 담을 수 있게 되었다.**

***가상 함수 대신에 쓸 수 있는 다른 방법으로 NVI관용구 및 전략 패턴을 들 수 있습니다. NVI관용구는 템플릿 메서드 패턴의 한 예입니다.***

***객체에 필요한 기능을 멤버 함수로부터 클래스 외부의 비멤버 함수로 옮기면, 그 비멤버 함수는 그 클래스의 public멤버가 아닌 것들을 접근할 수 없다는 단점이 생깁니다.***

***function객체는 일반화된 함수 포인터처럼 동작합니다. 이 객체는 주어진 대상 시그니처와 호환되는 모든 함수호출성 객체를 지원합니다.***

[항목36] 비가상 함수는 재정의 금지

상속받은 비가상 함수를 파생 클래스에서 재정의하는 것은 미친 짓이다.

# 비가상 함수 재정의

#### 비가상 함수를 재정의하면 어떻게 되는가?

어떠한 모습인지 아래 코드에서 확인한다.

**class** **Archer**{

**public**:

void attack() **const**;

};

void Archer::attack() **const**{

cout << "shot!" << endl;

}

**class** **BladeMaster** : **public** Archer{

**public**:

*//가상함수가 아닌데 재정의?*

void attack() **const**;

};

void BladeMaster::attack() **const**{

cout << "cool Blade." << endl;

}

int main(){

BladeMaster bm;

Archer \*pA = &bm;

pA->attack(); *// shot!*

BladeMaster \*pBm = &bm;

pBm->attack(); *// cool Blade.*

*//같은 bm객체인데 불리는 함수가 다르면 안 됩니다.*

*//비가상 함수거든요.*

}

#### 같은 객체에서 불리는 함수가 다르다?

뭐, 상황에 따라서 괜찮지 않을까? 라고 생각할 수도 있다. 안타깝게도 아주 멍청한 생각이다. 비가상 함수라는 것은 파생이든 기본이든 공통적으로 그 객체가 가지고 있는 기능인데 같은 객체에서 그 기능이 때에 따라 다르다면 난감하게 된다.

###### 가상 함수 vs. 비가상 함수의 동작 차이

가상 함수에서는 pA든 pBm이든 같은 객체에 대하여 같은 함수가 호출된다. 이러한 **차이의 이유는 비가상 함수는 정적 바인딩으로 묶이며 가상 함수는 동적 바인딩으로 묶이기 때문**이다. ***– [항목34]*정적 바인딩**은 선언과 동시에 묶어버린다. pA로 선언하면 pA타입으로 고정되버린 것이다. 반면 **동적 바인딩**은 가상 함수 포인터로 적절한 함수를 찾기 때문에 진짜 객체가 무엇인지 찾을 수 있다.   
**그러므로 비가상 함수는 무엇을 가리키느냐에 따라 함수가 바뀌며(실제 객체를 파악하지 못함), 가상 함수는 실제 객체를 파악하여 항상 고정된 함수를 불러온다.**

#### 비가상 함수의 재정의가 갖는 의미

public상속은 앞서 본바와 같이, ~의 일종이다는 의미이며 상속받은 비가상함수는 파생 클래스가 반드시 가지고 있어야 할 기능이다. 그러나 이것을 덮어버리면 못쓰게 되므로 상속의 의미가 깨져버린다. 소멸자 또한 반드시 가상 소멸자로 두어야 하는 의미도 이와 통한다. ***– [항목7]***

***상속 받은 비가상 함수를 재정의하는 일은 절대로 하지 맙시다.***

[항목37] 상속받은 기본 매개변수

어떤 함수에 대해서도 상속받은 기본 매개변수 값을 절대로 재정의하지 말자.

# 기본 매개변수?

이런 것이다. 아래를 보라.

*//num2, num3은 없어도 디폴트 값으로 처리한다.*

void Test( int nNum1, int nNum2=20, int nNum3=30 );

# 기본 매개변수를 가진 재정의된 가상 함수

일단 비가상함수는 절대로 재정의할 수 없으므로(**[항목36]**) 가상 함수에 한정된다.

**class** **Archer**{

**public**:

**enum** **class** **AttackType** { Sword, Bow, Shotgun };

*//기본 매개변수: Bow*

**virtual** void attack(AttackType type = AttackType::Bow) **const** = 0;

};

**class** **BladeMaster** : **public** Archer{

**public**:

*//기본 매개변수가 달라졌어요!*

**virtual** void attack(AttackType type = AttackType::Shotgun) **const**;

};

void BladeMaster::attack(AttackType type = AttackType::Shotgun) **const**{

cout << "my weapon type is " << (int)type << endl;

}

**class** **ArchMage** : **public** Archer{

**public**:

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

};

void ArchMage::attack(AttackType type) **const**{

cout << "my weapon type is " << (int)type << endl;

}

int main(){

***//선언된 타입을 정적 타입이라고 합니다.***

Archer \*pA;

Archer \*pBm = **new** BladeMaster;

Archer \*pAm = **new** ArchMage;

***//그 객체가 진짜로 무엇이냐에 의해 결정되는 타입을 동적 타입이라 합니다.***

*//아래처럼 동적 타입은 실행 중에 바뀔 수 있다. pA의 동적 타입은 BM이다.*

pA = pBm;

pA = pAm;

*//2개의 값이 같다. 왜냐하면 기본 매개변수는 정적 타입에서 결정되기 때문이다.*

*//하지만 현재 기본 매개변수가 다르면 링크 에러가 나는 것으로 확인된다.*

pBm->attack();

pAm->attack();

}

결론: 가상함수에 기본 매개변수가 있으면 예상할 수 없는 일이 발생할 수 있다!

#### 정적 타입 vs. 동적 타입

선언할 때 부여한 타입이 **정적 타입**이며 이는 변하지 않는다.   
**동적 타입**은 실제로 가리키는 객체의 타입이며 런타임 중 언제든지 변할 수 있다.

#### 왜 기본 매개변수는 런타임에 변할 수 없을까?

런타임 효율. 지금의 매커니즘은 속도 유지와 구현 간편성에 더 무게를 두어 이와 같은 구현으로 기본 매개변수 값을 결정할 방법을 컴파일러 쪽에서 결정할 수 없게 되어 있다.

#### 기본 매개변수를 가질 때, 올바른 설계법

상속된 가상함수의 매개변수와 기본 가상함수의 매개변수는 같아야 한다.

**class** **Archer**{

**public**:

**enum** **class** **AttackType** { Sword, Bow, Shotgun };

*//기본 매개변수: Bow*

**virtual** void attack(AttackType type = AttackType::Bow) **const** = 0;

};

**class** **BladeMaster** : **public** Archer{

**public**:

*//기본 매개변수가 달라졌어요!*

**virtual** void attack(AttackType type = AttackType::Bow) **const**;

};

이런, 코드중복이군요. 동작에 문제는 없지만 모양세에서 심각한 냄새가 납니다.

# 코드 중복 피하기: 더 좋은 방법을 찾고 싶다면?

NVI관용구를 이용한다. ***– [항목35]***

**class** **Archer**{

**public**:

**enum** **class** **AttackType** { Sword, Bow, Shotgun };

*//비가상 함수*

void attack(AttackType type = AttackType::Bow) **const**{

doAttack(type);

};

**private**:

*//진짜 작업이 이루어집니다.*

**virtual** void doAttack(AttackType type) **const**;

};

**class** **BladeMaster** : **public** Archer{

**private**:

*//private에 있습니다.* ***기본 매개변수는 Bow로 고정됩니다. 고정된점이 중요***

**virtual** void doAttack(AttackType type) **const**;

};

void BladeMaster::doAttack(AttackType type) **const**{

cout << "my weapon type is " << (int)type << endl;

}

***상속 받은 기본 매개변수 값을 재정의하지 않습니다. 왜냐하면 기본 매개변수 값은 정적으로 바인딩되는 반면, 가상 함수(재정의할 수 있는 유일한 함수)는 동적으로 바인딩되기 때문입니다.***

[항목38] 객체 합성에 대하여

“has-a”혹은 “is-implemented-in-terms-of”를 모형화할 때는 객체 합성을 사용하자.

# (객체)합성(composition)이란?

C++설계에서 합성이란, 어떤 타입의 객체들이 그와 다른 타입의 객체들을 포함하고 있을 경우 성립하는 그 타입들 사이의 관계를 말한다. 다른 말로 layering, containment, aggregation, embedding등이 있다.

**class** **Archer**{

**public**:

void attack() **const**;

**private**:

*//string, weapon객체를 가짐: 객체 합성*

string name;

Weapon w;

};

# 객체 합성과의 비교

설계에서 어떠한 구조를 사용해야 할까? 각각의 갖는 의미를 명확히 하고 그 의미에 맞는 적절한 구조를 사용해야 한다.

#### 상속 vs. 객체 합성

**public상속**: “is-a(~의 일부)”  
**private상속**: “is-implemented-in-terms-of(~을 써서 구현됨)” – *[항목39]***객체 합성**: 응용 영역(실생활을 본뜬)에서는 “has-a(~을 가짐)”, 시스템 영역(순수히 구현만을 위한 인공물)에서는 “is-implemented-in-terms-of(~을 써서 구현됨)”

###### is-implemented-in-terms-of(~을 써서 구현됨)

~을 써서 구현됨이라 함은, 해당 클래스의 일부 기능을 사용한다는 것이다. 즉, 모든 부분이 아닌 일부분의 기능만 사용한다는 것이다. 인터페이스를 따라한다는 의미와는 관련이 없다.

# 상속의 한계

상속이란 객체들간의 관계를 구축한다. 그리고 public상속은 is-a라는 관계를 구축하는 것이다. 이는 생각보다 굉장히 강력한 결합이라는 것이다. is-a는 같다는 뜻 아닌가? 이 의미를 정확하게 이해할 수 있어야 한다.

#### ***상속은 friend다음으로 강한 결합이다.***

상속을 하게 되면 완전히 내것이 된다. 어떠한 객체가 생겨나든 그것은 본래의 기능과 같다. 예를 들어, 핸드폰이라는 클래스가 있다. 그리고 이 핸드폰에는 카메라의 기능을 가지고 있다. 만약 이를 구현한다면, 카메라의 기능을 재정의하여 핸드폰의 기능으로 사용하기 편하므로 상속이 적절해 보인다. 하지만 그렇지 않을 가능성이 높다는 것이다.

핸드폰은 카메라인가? 아니다. 카메라의 기능을 사용할 뿐이지 만약 상속이 되려면 핸드폰은 카메라의 버튼부터 모든 기능, 데이터를 완전히 가져야 한다.

만일 이를 어긴다면, 대가를 치를 것이다. 핸드폰이 상속받은 카메라의 모든 기능은 외부에서 접근될 수 있다. 어디서 접근될 지 모르며, 의도치 않게 핸드폰에 없어야 할 데이터, 기능으로 인해 변경될 수도 있으며, 데이터와 기능이 어지럽혀져 핸드폰이 가지는 사진 촬영이라는 기능의 캡슐화는 깨져버릴 것이다. 상속은 캡슐화를 산산조각내기에 충분하다.

어떠한 객체(기능)을 물려받고 싶은데, 객체 내부에서만 사용하고 싶다. 즉, 기능은 물려받으면서 외부의 접근을 철저히 통제하고 싶다. **조금 더 느슨한 결합이 되는 것이다. 이런 경우에 객체 합성이 적합하다.**

#### 상속은 클래스간의 정적인 구조이다.

상속은 클래스의 계층 관계이다. **인스턴스 관계가 아니다. 클래스는 기능 명세 틀이다. 객체 관계와는 다르다.** 만일 정해진 개수의 객체에 대해서만 기능을 추가하고 싶다면 상속해서는 안 된다. 의미상 맞지도 않으며, 혼돈 그 자체가 된다. 이런 경우에 객체 합성이라는 개념이 적합하다.

#### 결론

최대한 객체 합성을 이용하는 것이 좋다. 불명확한 관계 표현, 상속 납용할 바에는 안 쓰는 것이 낫다. **객체 합성은 어떠한 객체에 기능을 위임하기에 가장 적절하다. (has-a)**

***객체 합성의 의미는 public상속과 완전히 다릅니다.***

***응용 영역에서는 has-a이며 구현 영역에서는 is-implemented-in-terms-of입니다.***

[항목39] private상속

private상속은 심사숙고해서 구사하자.

# private상속의 의미

앞에서 언급한대로, 객체 합성과 의미가 같다. 의미가 같으면 왜 구분되어야 하는가? 둘의 차이점을 알아보고 적절한 사용 시기를 파악한다.

상속 형태에 따라서 각 멤버들은 다음과 같이 받아들여진다.



# private상속 vs. 객체 합성

같은 코드를 2가지 방법으로 설계한 모습을 관찰하겠다. 2가지는 같은 것이다.

**class** **Weapon**{

**public**:

**explicit** Weapon(int p) : power(p) {}

**virtual** void effect() **const**;

**private**:

int power;

};

*//private상속입니다.*

*//Weapon을 가지고 있다는 의미입니다.*

**class** **Archer** : **private** Weapon{

**private**:

*//private상속이므로 private멤버가 되었다.*

*//상속하므로 재정의할 수 있습니다.*

**virtual** void effect() **const**;

};

*//(public)상속 + 객체 합성*

**class** **BladeMaster**{

**private**:

*//private안에 클래스를 정의합니다.*

*//public상속하고 가상 함수를 재정의합니다.*

*//그 안에서 재정의함. 밖에서는 불가능(내부에서만 클래스 사용 가능)*

**class** **MasterPlan** : **public** Weapon{

**public**:

**virtual** void effect() **const**;

};

MasterPlan mp;

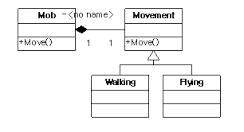
};

우리는 다양한 설계와 이렇게 생각할 힘을 길러야 한다.

#### (public상속) + 객체 합성을 언제 쓸까?

###### Question. (public)상속은 왜 했을까요?

위의 코드에서는 Weapon라는 객체를 가지고 있다. 그런데 Weapon이 한 종류만 있는 것은 아니다. **여러 기능으로 재정의되어 다양하게 쓰일려면 public상속 형태가 유용한 것이다.** 반면, private상속은 재정의가 되므로 이럴 필요가 없다.

좌측과 같은 패턴을 기억한다.

**가능하면 private상속보다 객체 합성을 쓴다.**   
1. 가상 함수 재정의를 막을 수 있다. C#의 sealed역할처럼 파생 클래스에서 가상 함수를 재정의할 수 없다. 위의 코드에서 BladeMaster클래스에서는 effect()함수를 재정의할 수 없는 것이다. 왜냐하면 effect함수가 없지 않은가.  
2. 컴파일 의존성을 최소화할 수 있다. 굳이 MasterPlan의 정의를 필요로 하지 않는다. 전방선언만 하면 컴파일이 가능하다.

*//만약 Weapon에서 파생된다면, Weapon의 구현 내용이 필요합니다.*

**class** **Archer** : **private** Weapon{

**private**:

**virtual** void effect() **const**;

};

*//객체 상속에서 포인터로 선언하면 구현 내용이 필요 없습니다.*

**class** **BladeMaster**{

**private**:

*//MasterPlan의 구현 내용을 빼고*

*//포인터만 갖도록 해줍니다.*

**class** **MasterPlan;**

MasterPlan **\*mp**;

};

#### private상속은 언제 쓸까?

꼭 private상속을 사용해야 하는지 검토해본다.   
1. 비공개나 protected멤버로 접근해야 하는 경우  
2. 가상 함수를 재정의해야 하는 경우  
3. ***공백 기본 클래스 최적화(EBO)***를 사용하는 경우

###### **공백 기본 클래스 최적화(EBO)**

C++에서 **독립 구조**의 객체는 반드시 크기가 0을 넘어야 한다라는 규칙이 있다. 그래서 공백 클래스를 포함한다면 크기가 1이 된다. 하지만 이를 private상속시킨다면 독립 구조가 아니므로 크기를 0으로 만들 수 있다. 이는 단일 상속에서만 가능하다.

공백 클래스는 왜 사용하는가? 비정적 데이터는 안 갖고 있지만 typedef, enum등 기술적으로 공백 처리된 클래스가 많다. 저장된 자료를 이용하는 목적이 아닌 틀을 만들 때 이용할 수 있겠다. 그리고 사용자 정의 함수 객체 생성에 제공될 수 있다. 다음과 같이 사용된다. 예로 std::<functional>의 binary\_function이다.

**template** <**class** **Arg1**, **class** **Arg2**, **class** **Result**>

**struct** binary\_function {

**typedef** Arg1 first\_argument\_type;

**typedef** Arg2 second\_argument\_type;

**typedef** Result result\_type;

};

# [참고] 프로파일(profile)

프로그램이 실행되면서 호출되는 함수들의 순차적 리스트 및 각 함수의 실행 시간과 전체 시간에 대한 실행 시간 비율을 측정하는 작업 혹은 그 분석 결과.

여러 가지 프로파일러가 있다. 가장 간단한 것은 VS자체 프로파일링 기능이다. 분석 탭에서 프로파일링에서 새 세션을 추가하여 실행한다.



# [참고] 수행시간 측정 함수

|  |
| --- |
|  |
|  |
|  |

#### C++11: std::chrono를 이용한 시간 측정

OS가 아닌 C++표준에서 제공하는 시간 측정이 등장했다.

###### 기본 시간 측정법

#include <chrono>

using namespace std;

using namespace chrono;

// Debug 모드로 재 보니, i5-750에서 9.87초나 걸린다

void BigLoop()

{

for (int i = 1; i <= 50000; i++)

{

for (int j = 1; j <= 50000; j++)

{

int x = (i + j) \* 2 + (i / j);

}

}

}

int main()

{

// now() 함수를 통해 현재 시간값을 구한다.

system\_clock::time\_point start = system\_clock::now();

BigLoop();

system\_clock::time\_point end = system\_clock::now();

// 초 단위 (소수점으로 표현)

duration<double> sec = end - start;

}

###### 제공되는 시간 단위

typedef duration<int, ratio<3600> > hours;

typedef duration<int, ratio<60> > minutes;

typedef duration<long long> seconds;

typedef duration<long long, milli> milliseconds;

typedef duration<long long, micro> microseconds;

typedef duration<long long, nano> nanoseconds;

이를 이용하여 다양한 시간 단위로 시간을 측정한다.

system\_clock::time\_point start = system\_clock::now();

BigLoop();

system\_clock::time\_point end = system\_clock::now();

// 시간 단위

hours hour = duration\_cast<hours>(end - start);

// 분 단위

minutes min = duration\_cast<minutes>(end - start);

// 초 단위

duration<double> sec = end - start;

// 밀리 초 단위 (1/1000)

milliseconds milliSec = duration\_cast<milliseconds>(end - start);

// 마이크로 초 단위 (1/1000000)

microseconds microSec = duration\_cast<microseconds>(end - start);

// 나노 초 단위 (1/1000000000)

nanoseconds nanoSec = end - start;

***private상속의 의미는 is-implemented-in-terms-of입니다. 대개 객체 합성과 비교해서 쓰이는 분야가 많지 않지만, 파생 클래스 쪽에서 기본 클래스의 protected멤버에 접근해야 할 경우 혹은 상속받은 가상 함수를 재정의해야 할 경우에는 private상속이 나름대로 의미가 있습니다.***

***객체 합성과 달리, private상속은 공백 기본 클래스 최적화(EBO)를 활성화시킬 수 있습니다. 이 점은 객체 크기를 가지고 고민하는 라이브러리 개발자에게 꽤 매력적인 특징이 되기도 합니다.***

[항목40] 다중 상속

다중 상속은 심사숙고해서 사용하자.

# 다중상속 문제점1: 모호성

상속받은 2개의 클래스(다중 상속)에 같은 함수가 있다면 어느 것을 호출할 지 모호해진다.

#### 해결책

호출할 클래스의 이름을 직접 호출해준다.

**class** **Archer**{

**public**:

void move() **const**;

};

**class** **Builder**{

**public**:

void move() **const**;

};

**class** **Player** : **public** Archer, **public** Builder{

**public**:

void dance() **const**{

**for** (int i = 0; i < 5; i++)

*//move(); //모호합니다.*

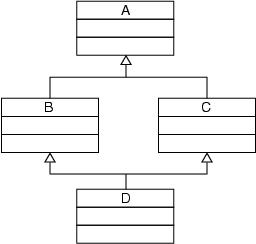
**Builder::**move(); *// 어디서 호출할 지 지정하는군요.*

}

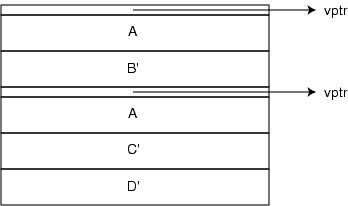
};

# 다중 상속의 문제점2: 중복 생성

다음은 죽음의 마름모꼴이라 불리는 형태이다.



위와 같은 구조에서 상속하게 되면 그 경로(B-A, D-C-A)의 데이터 멤버를 모두 생성해야 하는데 심각한 낭비다. 다음과 같은 메모리 구조를 갖게 된다.

A클래스가 2번 담기므로 데이터 멤버도 2개씩 포함

#### 해결책

가상 상속을 사용한다.

참조: <http://www.drdobbs.com/cpp/multiple-inheritance-considered-useful/184402074>

###### 가상 상속의 사용

**class** **B** : **public** **virtual** A {

**public**:

**virtual** ~B();

**virtual** void foo();

};

**class** **C** : **public** **virtual** A {

**public**:

**virtual** ~C();

**virtual** void foo();

**virtual** void foobar();

};

**class** **D** : **public** **virtual** B, **public** **virtual** C {

**public**:

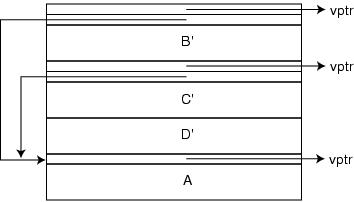
**virtual** ~D();

**virtual** void foo();

**virtual** void foobar();

};

다음은 메모리에 저장된 동작 원리이다. 가상 상속 역시 포인터를 이용해 적절한 클래스를 찾는 동작이 보인다.

가상 상속을 이용하여 1개의 A만 생성하였다.

###### 가상 상속의 문제와 해결

**class** **Player** : **virtual** **public** Archer, **virtual** **public** Builder{

**public**:

*//...*

};

[문제1] 가상 상속으로 만들어진 객체의 크기는 일반 상속보다 크며 접근 속도도 느리다.   
[문제2] 가상 기본 클래스의 초기화 규칙은 일반 규칙과 다르다. 그 규칙은 파생 클래스는 거리와 상관없이 기본 클래스의 존재를 염두에 두고 있어야 하며, 기존의 클래스 계통에 파생 클래스를 새로 추가할 때에도 그 파생 클래스는 기본 클래스의 초기화를 떠맡아야 한다. 결국, 초기화, 대입 연산이 복잡해진다.

**[해결]** 이러한 문제를 가진 가상 상속이므로 단순히 가상 상속의 사용은 다중 상속을 해결해 줄 수 없다. 다음과 같이 사용해야 한다.  
1. 최대한 사용하지 말기. 단일 상속으로 해결할 방법이 있을 수도 있다.  
2. **인터페이스(interface)**처럼 사용하기

#### 가상 상속을 사용한 예시

다음은 가상 상속을 사용하지 않아서 데이터 중복이 생긴 코드이다. 각 함수들은 가상 함수와 비가상 함수를 하나씩 가지고 있다.

class Item

{

public:

virtual void getInfo() const

{

std::cout << "item id: " << id << '\n';

}

void setIdentify(int \_id)

{

id = \_id;

}

protected:

int id;

};

class Sword : public Item

{

public:

virtual void getInfo() const

{

std::cout << "sword id: " << id << '\n';

}

void blade()

{

std::cout << "blade!\n";

}

};

class Armor : public Item

{

public:

virtual void getInfo() const

{

std::cout << "armor id: " << id << '\n';

}

void powerProtect()

{

std::cout << "protect.\n";

}

};

class MasterPlan : public Armor, public Sword

{

public:

virtual void getInfo() const

{

//여기서 id의 정체가 모호해지는 문제가 발생하므로

//Sword::id처럼 구체적으로 명시해주어야 한다.

std::cout << "masterPlan id: " << Sword::id << '\n';

}

void megaBlade()

{

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

}

};

int main()

{

MasterPlan mp;

//기본 클래스의 함수인 setIdentify가 모호해져서 호출할 수 없다.

//mp.setIdentify(9999);

mp.blade();

mp.powerProtect();

mp.megaBlade();

mp.getInfo();

return 0;

}

아래는 가상 상속을 사용한 경우이다.

class Item

{

public:

virtual void getInfo() const

{

std::cout << "item id: " << id << '\n';

}

void setIdentify(int \_id)

{

id = \_id;

}

protected:

int id;

};

class Sword : virtual public Item

{

public:

virtual void getInfo() const

{

std::cout << "sword id: " << id << '\n';

}

void blade()

{

std::cout << "blade!\n";

}

};

class Armor : virtual public Item

{

public:

virtual void getInfo() const

{

std::cout << "armor id: " << id << '\n';

}

void powerProtect()

{

std::cout << "protect.\n";

}

};

class MasterPlan : public Armor, public Sword

{

public:

virtual void getInfo() const

{

//id의 정체에 아무런 문제가 없음

std::cout << "masterPlan id: " << id << '\n';

}

void megaBlade()

{

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

}

};

int main()

{

MasterPlan mp;

//가상 상속으로 기본 클래스를 상속받으면 정상적으로 사용 가능

mp.setIdentify(9999);

mp.blade();

mp.powerProtect();

mp.megaBlade();

mp.getInfo();

return 0;

}

# 다중 상속의 적절한 사용

#### 인터페이스 public상속과 private상속을 조합하는 경우

가상 상속은 아니다. 데이터가 없으므로 중복 생성되어도 문제가 안 된다는 것인가?

*//인터페이스, 순수 가상 함수*

**class** **IArcher**{

**public**:

*//반드시 가상 소멸자 - [항목7]*

**virtual** ~IArcher();

**virtual** void attack() **const** = 0;

};

**class** **Weapon**{

**public**:

**virtual** ~Weapon();

**virtual** void effect() **const**;

};

**class** **BladeMaster** : **public** **IArcher**, **private** **Weapon**{

**public**:

*//펙토리 함수: IArcher를 반환하기 위해 인터페이스 상속*

**static** **IArcher**\* createArcher()

{

return **new** **BladeMaster**;

}

*//인터페이스의 함수 정의, 재정의 필수*

**virtual** void attack() **const**{

cout << "blade!" << endl;

}

*//Weapon의 가상 함수를 재정의*

**virtual** void effect(){

cout << "power up for only blade!" << endl;

}

};

그 이외에 여러 가지 경우가 있을 수 있다.

#### [Question] 뭐가 더 있을까?

데이터가 없는 인터페이스라면 다중 상속도 괜찮을 것 같다. 기능을 명세하는 용도로 사용하거나 해당 포인터 타입으로 융통성있는 객체들을 만들 수 있을 것이다.

또 찾아보세요.

***다중 상속은 단일 상속보다 확실히 복잡합니다. 새로운 모호성 문제를 일으킬 뿐만 아니라 가상 상속이 필요해질 수도 있습니다***

***가상 상속을 쓰면 크기 비용, 속도 비용이 늘어나며, 초기화 및 대입 연산의 복잡도가 커집니다. 따라서 가상 기본 클래스에는 데이터를 두지 않는 것이 현실적으로 기징 실용적입니다.***

***다중 상속을 적법하게 쓸 수 있는 경우가 있습니다. 여러 시나리오 중 하나는, 인터페이스 클래스로부터 public상속을 시킴과 동시에 구현을 돕는 클래스로부터 private상속을 시키는 것입니다.***

[항목41] 템플릿 프로그래밍의 뿌리

암시적 인터페이스와 컴파일 타임 타형성에서 시작한다.

# 객체 지향 vs. 템플릿

명시적 인터페이스 vs. 암시적 인터페이스  
런타임 다형성 vs. 컴파일 타임 다형성  
**객체 지향 프로그래밍**은 *명시적 인터페이스*와 *런타임 다형성*에 의해 설계된다. 반면 **템플릿 프로그래밍**에서는 *암시적 인터페이스*와 *컴파일 타임 다형성*이 활개친다. 각각의 차이를 살펴보자.

# 명시적 인터페이스 vs. 암시적 인터페이스

이게 각각 뭘 의미하는지 정확하게 알 필요가 있다.

#### 명시적 인터페이스

곧, 함수 시그니처 모임이 **명시적 인터페이스**다. 다음 코드를 보자.

**class** **Archer**{

**public**:

*//시그니처:* ***함수의 이름, 매개변수 타입, 반환 타입****이 포함된다.*

Archer();

**virtual** void move() **const**;

**virtual** void attack(Archer& target) **const**;

**private**:

*//데이터 멤버는 시그니처에 들어가지 않는다.*

int level;

int power;

};

#### 암시적 인터페이스

**암시적 인터페이스**는 보이지 않는 인터페이스다. 아래 코드에서 설명해보자.

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

void doSomething(T &a){

*//size()함수와 !=연산자 함수를 지원해야 합니다.*

**if** (a.size() > 10 && a != ytk)

cout << "it's stupid." << endl;

*//...*

}

위 코드은 정당한가? 아직 모른다. 성립하기 위해서는**, T가 size와 !=연산자 함수를 지원해야 한다. 이와 같은 조건(표현식)이 위에서 보이는가?** 안 보인다. 하지만 암시적으로 반드시 지켜져야 하는 인터페이스다. 이러한 표현식의 집합이 암시적 인터페이스다.

# 런타임 다형성 vs. 컴파일 타임 다형성

***다형성(polymorphism)***이란 모습은 같은데 다른 형태를 갖는 성질이다. 다시 말하자면, 타입이 다른 객체가 똑같은 기능을 수행해도 서로 다른 동작을 하는 것이다.

예를 들어, void attack이라는 함수가 있다고 하자. 그리고 이 기능을 가진 Archer라는 클래스가 있다. 그리고 이에서 파생된 BladeMaster라는 클래스가 있다. 각각에 대해 attack기능을 수행시킬 때, 어떻게 반응해야 할까? 같은 기능이므로 똑 같은 반응을 해야 할까? 그렇지 않을 수도 있다는 것이 다형성이 가지는 의미다.

대표적으로 위와 같은 작업을 수행해주는 virtual선언이 있다. 하지만 이것만 있는 것은 아니다. C++에는 4가지 종류의 다형성이 존재한다.

# C++의 4가지 다형성 (The Four Polymorphisms in C++)

* 서브타입 다형성은 **런타임 다형성(Runtime Polymorphism)**으로 알려져 있다.
* 매개 변수 다형성은 **컴파일 타임 다형성(Compile-Time Polymorphism)**으로 알려져 있다.
* 임시 다형성은 **오버로딩(Overloading)**으로 알려져 있다.
* 강제 다형성은 (암시적 또는 명시적) **캐스팅(Casting)**으로 알려져 있다.

## 서브타입 다형성 (런타임 다형성)

서브타입 다형성은 C++에서 "다형성"을 이야기할 때 모든 사람들이 이해하고 있는 의미의 다형성이다. Base 클래스의 포인터 또는 참조형을 통해 Derived 클래스를 사용하는 기능을 말한다. 여기에 예제가 있다. 고양이과에 속하는 다양한 종류의 고양이들을 가지고 있다고 가정해 보자.

모두 생물학적으로 고양이과에 속하고 "야옹"이라고 할 수 있기 때문에, Felid라는 Base 클래스로부터 상속을 받고 meow라는 순수 가상 함수를 오버라이딩하는 클래스들로 나타낼 수 있다.

// file cats.h

class Felid {

public:

virtual void meow() = 0;

};

class Cat : public Felid {

public:

void meow() { std::cout << "Meowing like a regular cat! meow!\n"; }

};

class Tiger : public Felid {

public:

void meow() { std::cout << "Meowing like a tiger! MREOWWW!\n"; }

};

class Ocelot : public Felid {

public:

void meow() { std::cout << "Meowing like an ocelot! mews!\n"; }

};

이제 메인 프로그램에서 Felid (Base 클래스) 포인터를 통해 Cat, Tiger, Ocelot를 번갈아 가며 사용할 수 있다.

#include <iostream>

#include "cats.h"

void do\_meowing(Felid \*cat) {

cat->meow();

}

int main() {

Cat cat;

Tiger tiger;

Ocelot ocelot;

do\_meowing(&cat);

do\_meowing(&tiger);

do\_meowing(&ocelot);

}

메인 프로그램은 Cat, Tiger, Ocelet을 가리키는 포인터를 do\_meowing 함수로 전달한다. 전달되는 포인터들은 모두 Felid이기 때문에, 프로그램은 객체마다 올바른 meow 함수를 호출하며 출력 결과는 다음과 같다.

Meowing like a regular cat! meow!

Meowing like a tiger! MREOWWW!

Meowing like an ocelot! mews!

다형성 함수의 호출 결정은 런타임에 가상 테이블을 통한 **간접 참조**를 통해 일어난다. 좀 더 쉽게 설명하자면, 컴파일러가 컴파일 타임 때 호출될 함수의 주소를 찾는 것이 아니라 프로그램을 실행할 때 **가상 테이블에 있는 오른쪽 포인터를 역참조**해 함수를 호출하는 것이다. – [항목34] 참고

타입 이론에서는 서브타입 다형성을 **포함 다형성(Inclusion Polymorphism)**이라고도 부른다.

## 매개 변수 다형성 (컴파일 타임 다형성)

매개 변수 다형성은 **어떤 타입에 대해 동일한 코드**를 실행하기 위한 수단을 제공한다. 이는 **컴파일 타임**에 일어나며, C++에서 매개 변수 다형성은 **템플릿**을 통해 구현할 수 있다.

가장 간단한 예제 중 하나는 두 개의 인수 중에서 큰 값을 찾는 일반화된 max 함수다.

#include <iostream>

#include <string>

template <class T>

T max(T a, T b) {

return a > b ? a : b;

}

int main() {

std::cout << ::max(9, 5) << std::endl; // 9

std::string foo("foo"), bar("bar");

std::cout << ::max(foo, bar) << std::endl; // "foo"

}

max 함수는 **타입 T에 따라서 다양한 형태**가 될 수 있다. 하지만, 포인터를 통한 비교는 내용이 아닌 메모리 위치를 비교하기 때문에 포인터 타입에 대해서는 동작하지 않는다. 포인터 타입에 대해서 동작하게 만들고 싶다면 포인터 타입에 대해 템플릿 특수화해야 되는데, 그렇게 되면 더 이상 매개 변수 다형성이 아닌 임시 다형성이 될 것이다.

## 임시 다형성 (오버로딩)

임시 다형성은 같은 이름을 가진 함수가 **각 타입에 따라 다르게 행동**할 수 있도록 해준다. 예를 들어, int 타입의 변수 2개와 +연산자가 주어졌다면, 두 변수를 더한다. 반면, std::string 타입의 변수 2개와 + 연산자가 주어졌다면, 두 변수를 연결하게 된다. 이를 **오버로딩**이라고도 부른다.

int와 std::string에 대해 add 함수를 구현한 구체적인 예제가 있다.

#include <iostream>

#include <string>

int add(int a, int b) {

return a + b;

}

std::string add(const char \*a, const char \*b) {

std::string result(a);

result += b;

return result;

}

int main() {

std::cout << add(5, 9) << std::endl;

std::cout << add("hello ", "world") << std::endl;

}

C++에서 임시 다형성은 템플릿 특수화를 할 때도 나타난다. max 함수를 다룬 예제로 돌아가, char 포인터 타입의 변수 2개에 대한 max 함수를 작성하는 방법을 살펴보자.

template <>

const char \*max(const char \*a, const char \*b) {

return strcmp(a, b) > 0 ? a : b;

}

이제 두 문자열 "foo"와 "bar" 중 큰 값을 찾기 위해 ::max("foo", "bar")를 호출할 수 있다.

## 강제 다형성 (캐스팅)

강제 다형성은 객체 또는 기본 타입이 다른 객체 또는 기본 타입으로 변환할 때 일어난다. 이는 같은 객체가 다른 두 타입으로 존재할 수 있다는 면에서 다형성의 의미를 갖는다. 예를 들어,

float b = 6; // int는 (암시적으로) float으로 승격된다.

int a = 9.99 // float은 (암시적으로) int로 강등된다.

명시적 캐스팅은 (unsigned int)나 (int)와 같이 C 스타일의 타입 캐스팅 표현식을 사용하거나 C++의 static\_cast, const\_cast, reinterpret\_cast, dynamic\_cast를 사용할 때 일어난다.

또한 강제 다형성은 클래스의 생성자가 explicit로 선언되어 있지 않을 때에도 일어나는데, 예를 들어,

#include <iostream>

class A {

int foo;

public:

A(int ffoo) : foo(ffoo) {}

void giggidy() { std::cout << foo << std::endl; }

};

void moo(A a) {

a.giggidy();

}

int main() {

moo(55); // 55를 출력

}

만약 A의 생성자를 explicit로 선언했다면, 위 예제는 더 이상 실행이 되지 않는다. 변환과 관련된 실수를 피하기 위해서는 생성자를 explicit로 선언하는 것이 좋다.

또한 클래스에 타입 T에 대한 변환 연산자를 정의하는 경우, T 타입을 알 수 있는 곳이라면 어디든 사용할 수 있다.

예를 들어,

class CrazyInt {

int v;

public:

CrazyInt(int i) : v(i) {}

operator int() const { return v; } // CrazyInt에서 int로 변환

};

CrazyInt는 int 타입에 대한 변환 연산자를 정의했다. 이제 int를 인수로 받는 print\_int 함수가 있다면, CrazyInt 타입의 객체도 전달할 수 있게 된다.

#include <iostream>

void print\_int(int a) {

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

}

int main() {

CrazyInt b = 55;

print\_int(999); // 999를 출력

print\_int(b); // 55를 출력

}

실제로 앞에서 설명한 서브타입 다형성은 Derived 클래스를 Base 클래스 타입으로 변환하기 때문에 강제 다형성이기도 하다.

# 객체 지향 프로그래밍의 다형성

객체 지향 프로그래밍에서 런타임 다형성이 작동하는 과정을 살펴보자.

#### 다형성이 왜 필요한가?

객체는 계층관계나 포함관계를 가질 수 있다. 이는 객체간 관계를 알기 쉽게 해준다. 만약 다른 객체간의 아무런 관계가 없다고 한다면 다형성따위는 필요없다. 그런데 그런 경우는 거의 없다. 아래와 같은 경우를 보자.

class Archer

{

public:

void move() { std::cout << "move\n;"; }

};

class BladeMaster : public Archer

{

public:

void move() { std::cout << "move\n;"; }

};

이건 분명 멍청한 짓이다. 프로그래머라 할 수 없다. is-a관계라면 public상속하여 기본 클래스 함수를 그대로 쓰면 된다. 이로써 코드는 절약된다.

문제는 다른 기능을 수행하고 싶을 때이다. Archer와 BladeMaster의 공격은 다른데 각기 함수를 구현해야 할까? **같은 기능을 수행하라 명령해도 다르게 수행할 수는 없을까? 이것이 다형성이며, virtual로 이를 가능하게 한다.**

class Archer

{

public:

virtual void attack(){ std::cout << "shot\n"; }

};

class BladeMaster : public Archer

{

public:

virtual void attack(){ std::cout << "blade\n"; }

};

결론은 보기 좋은 인터페이스 설계를 위해, 이를 유연하게 해주는 것이 다형성이다.

#### 왜 런타임 다형성인가?

어떠한 실행 시점에서 이 객체는 어떤 함수를 실행할 것인가? **각 시점마다 다르며, 이 객체의 타입을 확인한 후에 결정할 수가 있다.** 어떻게 확인하는가? 시그니처(명시적 인터페이스)이다. 이는 시점마다 다르므로 런타임에 수행할 수 밖에 없다.

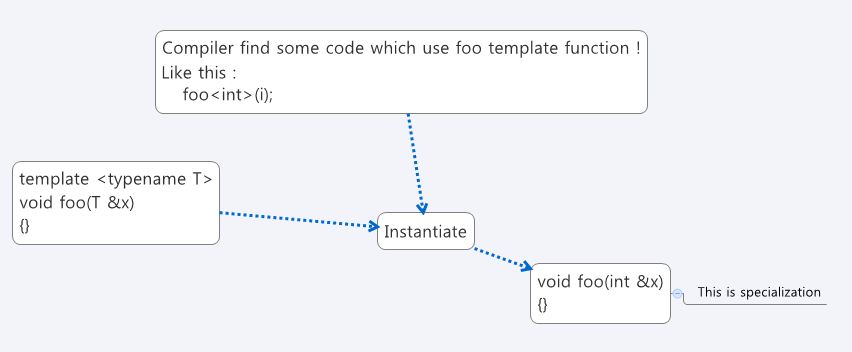
# 템플릿 프로그래밍의 다형성

템플릿은 일반화 프로그래밍을 만들어낸다. 일반화란, 클래스, 함수 등이 다양한 타입에서 실행 가능하게 되는 과정이라 볼 수 있다. 다양한 타입에서 실행되니 각 타입마다 실행될 다형성이 필수적이다.

#### 컴파일 타임 다형성의 동작

왜 컴파일 타임 다형성일까? 템플릿의 타입은 컴파일 타임에 정해져야 한다. 그래야 표현식(암시적 인터페이스)를 판단할 수 있다. 즉, 템플릿이 선언된 클래스 혹은 함수에 대해 유효한 인스턴스를 만들어내야 한다. 이는 컴파일 타임에 다양하게 나올 수 있는 다형성의 성질을 가진다.

###### 템플릿 인스턴스화



컴파일러가 템플릿 코드가 사용된 것을 감지하고 이를 구체화된 코드로 만들어 내는 것을 ***인스턴스화*** 라고 합니다. 인스턴스화는 해당 템플릿이 사용되지 않으면 인스턴스화 되지 않습니다. 그리고 이렇게 인스턴스화된 구체화된 결과물 (함수든 클래스든..) 을 특수화라고 합니다. 특수화의 경우 이렇게 인스턴스화를 통해서만이 아니라 명시적 특수화, 부분 특수화를 통해서도 가능합니다. – [항목26] 근데 보통은 템플릿의 특수화 라고 하면 명시적 특수화나 부분 특수화를 말합니다.

***클래스 및 템플릿은 다형성을 지원합니다.***

***클래스의 경우, 인터페이스는 명시적이며 함수의 시그니처를 중심으로 구성되어 있습니다. 다형성은 프로그램 실행 중에 가상 함수를 통해 나타납니다.***

***템플릿 매개변수의 경우, 인터페이스는 암시적이며 유효 표현식에 기반을 두어야 합니다. 다형성은 컴파일 중에 템플릿 인스턴스화와 함수 오버로딩 모호성 해결을 통해 나타납니다.***

[항목42] typename 사용법

typename의 2가지 의미를 제대로 이해하자.

# typename의 첫 번째 의미

템플릿 타입 매개변수를 선언하는 것. 참고로 여기서 typename과 class의 차이는 없다.

**template** <**typename** T> **class** **Archer**;

**template** <**class** **T**> **class** **Archer**;

# typename의 두 번째 의미

“나는 변수가 아닙니다. 타입입니다.”라고 명시적으로 선언하는 것

#### 용어 정리

template<typename T>

void find(const T& target, typename T::const\_iterator iter)

{

T power;

Archer<T> a1;

typename Archer<T>::Nasted a2;

int dd = 2;

}

이런 표현식이 있다고 생각해보자.

###### 의존 이름

템플릿 매개변수에 종속된 것. T에 따라 달라지는 변수의 타입  
🡪 const T&, T power, Archer<T> a1

###### 중첩 의존 (타입) 이름

어떤 클래스 안에 중첩된 의존 타입 이름. ::이 있으면 중첩된다는 의미다.  
🡪 typename T::const\_iterator iter, typename Archer<T>::Nasted a2;

###### 비의존 이름

템플릿과 연관 없는 것. int와 같은 기본 매개변수가 속한다.  
🡪 int dd

정리하자면, T만 사용한 타입은 의존 이름이며 T안의 혹은 T를 이용한 타입인 경우에는 중첩 의존 이름이라 하며 T와 연관없다면 비의존 이름이다.

#### typename을 언제 사용할까?

typename은 이 **중첩 의존 이름에 타입을 명시할 때** 사용한다. 아래와 같다.

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

**class** **MyTypeClass**

{

**public**:

**typedef** T A;

**static** int B;

**class** **C**{};

**static** void D(void){}

};

template<typename T>

void foo(){

MyTypeClass<int>::A \*E; *//A는* ***비의존******타입****이므로 변수 E를 선언합니다.*

MyTypeClass<int>::B \* F; *//B는 변수이므로 F와 곱합니다.*

MyTypeClass<int>::C \*G; *//C* ***비의존 타입****의 변수 G를 선언*

MyTypeClass<int>::D \* H; *//함수 D에 H를 곱합니다.*

*//****가리키는 것이 중첩 의존 타입일 경우****, 다음과 같이 한다.*

**typename** MyTypeClass<T>::A \*I;

}

#### typename을 쓰면 안 되는 경우

다시 한번 강조하지만, 중첩 의존 이름**’에만’** typename을 사용한다.

###### 중첩 의존 이름이 아니라면 사용할 수 없다.

T는 그냥 의존 이름이다. 의존이름에는 쓰지 말거라.

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

void find(**const** T& target, *//typename을 쓸 수 없습니다.*

**typename** T::const\_iterator iter) ***//typename이 필요합니다.***

###### 중첩 의존 이름임에도 불구하고 사용할 수 없는 경우

기본 클래스의 리스트에 있거나, 멤버 초기화 리스트 내의 기본 클래스 식별자로서 있을 경우에는 typename을 사용할 수 없다.

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

**class** **Archer**{

**public**:

**class** **Nasted** { Nasted(int x){} };

};

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

*//기본 클래스 리스트에는 typename이 없다.*

**class** **BladeMaster** : **public** Archer<T>::Nasted{

**public**:

*//멤버 초기화 리스트에는 typename을 쓸 수 없다.*

**explicit** BladeMaster(int x) : Archer<T>::Nasted(x){

***//중첩 의존 이름이며 위의 경우가 아니라면 typename이 필요***

**typename** Archer<T>::Nasted temp;

}

};

#### 중첩 의존 타입의 예제

template<typename iterT>

void findNobleBlood(iterT iter)

{

//value의 타입: iter이 실제로 가리키는 타입

typename std::iterator\_traits<iterT>::value\_type value1(\*iter);

using value\_t = std::iterator\_traits<iterT>::value\_type;

value\_t value2(\*iter); //value1과 타입이 같다.

}

template<class \_Ty>

struct iterator\_traits<\_Ty \*>

{ // get traits from pointer

typedef random\_access\_iterator\_tag iterator\_category;

typedef \_Ty value\_type;

typedef ptrdiff\_t difference\_type;

typedef ptrdiff\_t distance\_type; // retained

typedef \_Ty \*pointer;

typedef \_Ty& reference;

};

위의 value\_type에서 실제로 가리키는 타입을 나타낼 수 있는 것이다.

***템플릿 매개변수를 선언할 때, class 및 typename은 서로 바꾸어 써도 무방합니다.***

***중첩 의존 타입 이름을 식별하는 용도에는 반드시 typename을 사용합니다. 단, 중첩 의존 이름이 기본 클래스 리스트에 있거나 멤버 초기화 리스트에 기본 클래스 식별자로 있는 경우는 예외입니다.***

[항목43] 템플릿과 기본 클래스

템플릿으로 만들어진 기본 클래스 안의 이름에 접근하는 방법을 알아 두자.

# 템플릿화된 기본 클래스

기본 클래스가 템플릿이 포함되었다면, 주의해서 사용해야 한다. 원하는 대로 함수를 호출할 수 없을 수도 있다. 아래 코드를 보자.

class Item;

class MasterPlan : public Item;

class Potion : public Item;

class Archer

{

public:

void getItem(Item& item) \_NOEXCEPT

{

backPack.push\_back(item);

}

private:

std::vector<Item> backPack;

};

template<typename Item>

class Shop

{

public:

Shop()

{

for (int i = 0; i < 10; i++) makeItem();

}

void makeItem()

{

store.push\_back(Item());

}

void sellItem(Archer& buyer) \_NOEXCEPT

{

if (store.size() == 0)

{

std::cout << "sold out.\n";

return;

}

Item item = store.back();

store.pop\_back();

buyer.getItem(item);

std::cout << "sell the item.\n";

}

private:

std::vector<Item> store;

};

template <typename Item>

class ExtremeShop : public Shop<Item>

{

public:

ExtremeShop() : Shop<Item>() {}

//비가상 함수이므로 기본 클래스 함수와 이름이 다릅니다.

void specialSellItem(Archer& buyer) \_NOEXCEPT

{

sellItem(buyer);

std::cout << "it's special edition!\n";

}

};

int main()

{

Archer a1;

Shop<MasterPlan> mpShop;

mpShop.sellItem(a1);

ExtremeShop<MasterPlan> empShop;

empShop.specialSellItem(a1);

return 0;

}

#### 템플릿이 포함된 기본 클래스의 함수를 호출

현재 상황은 파생클래스에서 기본 클래스(템플릿이 있는)의 함수를 호출하는 것이다. ExtremeShop 의 비가상 함수 specialSellItem 에서 Shop 의 비가상 함수 sellItem 를 불러오고 있다. 이는 적법한 일이다. 하지만 템플릿에서라면 문제가 될 수 있다.

위 상황만 놓고 보면은 아무런 문제가 없다. 분명히public상속하므로 모든 기능을 사용한다는 보장이 있고 실제로 구현되어 있다. (실제로 이는 컴파일 오류도 없고 함수 실행이 아주 잘 된다.)

#### 기본 클래스에 완전 템플릿 특수화 버전이 있다면?

이런 상황에 위에서 말한 문제가 발생하는 것이다.

template<>

class Shop<Potion>

{

public:

Shop()

{

for (int i = 0; i < 10; i++) makeItem();

}

void makeItem()

{

store.push\_back(Potion());

}

void sellPotion(Archer& buyer) \_NOEXCEPT

{

if (store.size() == 0)

{

std::cout << "sold out.\n";

return;

}

Item item = store.back();

store.pop\_back();

buyer.getItem(item);

buyer.getItem(item);

std::cout << "sell the potion.\n";

}

private:

std::vector<Potion> store;

};

ExtremeShop<Potion> epShop;

epShop.specialSellItem(a1);

완전 템플릿 특수화를 이용해서 포션전용 샵을 만들어냈다. 여기에서도 익스트림은 물건을 팔 수 있는데 문제가 생겼다. sellItem 을 불러와야 하는데 없다. 완전 특수화 버전에는 이 함수가 없기 때문이다.

중요한 것은 기본 클래스의 typename의 형태를 모르므로 어떤 함수가 있는지 모른다. 그래서 컴파일이 안 된다는 점이다.

#### 본질적인 문제

기본 클래스의 기능을 사용하려면 컴파일러가 유효한지를 확인한다. 그런데 템플릿은 어떤 매개변수가 들어가느냐에 따라 이 유효판단이 달라진다. 위의 예제인 템플릿 특수화에 의해서 있는 함수가 없게 되는 것처럼 말이다. 결국, 컴파일러는 이른 진단(early diagnose)를 선호할 수 밖에 없다.

# 템플릿화된 기본 클래스 사용하기

C++ “난 템플릿화된 기본 클래스를 멋대로 뒤지지 않겠다.”

위에서 언급한 본질적인 문제에 의해 이는 때로 문제가 될 수 있다. 어떻게 하면 이를 무시하고 템플릿화된 기본 클래스의 멤버를 사용할 수 있을까?

#### this->사용하기

기본 클래스 함수 호출문 앞에 “this->”를 붙입니다.

void specialSellItem(Archer& buyer) \_NOEXCEPT

{

this->sellItem(buyer);

std::cout << "it's special edition!\n";

}

#### using선언을 사용하기

using선언을 하여 기본 클래스에 해당 함수가 있다고 알려줍니다. 그러면 컴파일러가 using선언한 곳을 찾으러 갈 것입니다.

void specialSellItem(Archer& buyer) \_NOEXCEPT

{

using Shop<Item>::sellItem;

sellItem(buyer);

std::cout << "it's special edition!\n";

}

#### 기본 클래스 함수 명시적 지정

기본 클래스를 직접 함수 앞에 써준다. 호출하는 함수가 가상 함수일 경우에는 가상 함수 바인딩이 무시되고 기본 클래스의 함수로 호출하므로 주의해야 한다.

void specialSellItem(Archer& buyer) \_NOEXCEPT

{

Shop<Item>::sellItem(buyer);

std::cout << "it's special edition!\n";

}

위의 3가지는 작동 원리는 동일하다. 컴파일러의 이른 진단이 필요 없다고 명시하고 싶다. 그러면 일종의 약속이 필요하다. 결국 기본 클래스 템플릿이 이후 어떻게 특수화되더라도 이렇게 인터페이스를 제공할 것이다라고 약속하는 것이다. 그리고 사용자는 그 약속에 맞게 특수화된 클래스에서 없는 함수를 호출하지 않도록 해야 할 것이다.

***파생 클래스 템플릿에서 기본 클래스 템플릿의 이름을 참조할 때는, “this->”를 접두사로 붙이거나 기본 클래스 한정문을 명시적으로 써 주는 것으로 해결합시다.***

[항목44] 템플릿에서 코드 분리

매개변수에 독립적인 코드는 템플릿으로부터 분리시키자.

# 비타입 매개변수로 인한 템플릿의 코드 비대화

수 많은 객체에 대하여 중복 코드 없이 만들 수 있게 해주는 템플릿. 함수에 코드 중복이 존재하는 것처럼 템플릿에도 똑 같은 코드가 중복될 수 있다. 다음을 보자.

***//함수의 코드 중복: 명시적입니다.***

int funcA(int a){

**for** (int i = 0; i < 1000; i++){

cout << "!" << endl;

}

cout << a << "is nice!" << endl;

**return** a;

}

int funcB(int b){

**for** (int i = 0; i < 1000; i++){

cout << "!" << endl;

}

cout << b << "is good!" << endl;

**return** b;

}

**class** **Fire**{};

**class** **Water**{};

***//템플릿 코드 중복: 암시적입니다.***

*//타입이름 T와* ***비타입 매개변수 n****이 추가된 것을 잘 보세요.*

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

**class** **CrossFire**{

**public**:

void shakeFire();

};

*//****템플릿 클래스에서 함수 선언****은 아래처럼 합니다.*

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

void CrossFire<T, n>::shakeFire(){

}

int main(){

*//매개변수 Fire, 10에 대한 코드 생성*

CrossFire<Fire, 10> cf\_f1;

cf\_f1.shakeFire();

*//또 매개변수 Fire, 5에 대한 코드 생성*

*//shakeFire에 대한 코드가 중복 생성되었군요.*

CrossFire<Fire, 5> cf\_f2;

cf\_f2.shakeFire();

}

#### [해결책] 중복되는 부분은 기본 클래스로 구분하자.

위에서 매개변수로 인해서 여러 개의 코드가 중복생성되었다. 그렇다면 매개변수에 영향을 받지 않는 부분(독립적인 부분)은 묶어서 기본 클래스로 빼주어 이를 사용하면(private상속) 코드 비대화를 피할 수 있을 것이다.

###### 기본 클래스로 중복 코드 분리하기

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

**class** **CrossFireBase**{

**protected**:

*//코드 복제를 피할 목적으로 만들어졌으므로 protected*

CrossFireBase(size\_t n, T \*pB) : size(n), pBomb(pB) {}

void shakeFire(size\_t n);

**private**:

size\_t size;

T \*pBomb;

};

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

void CrossFireBase<T>::shakeFire(size\_t n){

cout << n << " Base Fire." << endl;

}

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

*//private상속: Base의 일부가 아니라 Base를 사용할 뿐이다.*

**class** **CrossFire** : **private** CrossFireBase<T>{

**private**:

*//using* ***or*** *this->: 기본 클래스의 shakeFire이 가려지는 것 방지*

**using** CrossFireBase<T>::shakeFire;

**public**:

***//데이터를 넘겨주기 위해서 포인터를 인자로 생성자를 통해 올린다.***

***//기본 클래스의 생성자를 부를 때 이와 같이 하는 것을 다시 상기***

**CrossFire() : CrossFireBase<T>(n, bombs)** {}

void shakeFire();

**private**:

***//실제 데이터는 파생 클래스가 가지고 있다.***

T bombs[n \* n];

};

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

void CrossFire<T, n>::shakeFire(){

**this**->shakeFire(n);

}

이로써 크기 n에 따라서 새로운 객체를 생성하기는 하지만, 공통된 함수는 중복 생성하지 않는다. 기본 클래스로 빼버렸기 때문이다.

###### [참고] 데이터를 힙에 할당해보는 연습을 하자.

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

**class** **CrossFireBase**{

**protected**:

CrossFireBase(size\_t n, T \*pB) : size(n), pBomb(pB) {}

*//포인터를 저장하는 역할의 함수*

void setBombPtr(T \*ptr) { pBomb = ptr; }

void shakeFire(size\_t n);

**private**:

size\_t size;

T \*pBomb;

};

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

**class** **CrossFire** : **private** CrossFireBase<T>{

**public**:

*//힙에 할당하기 위하여 기본 클래스의 포인터는 일단 널포인터로 초기화시킨다.*

*//초기화하면서 데이터를 힙에 생성하고 이를 기본 클래스의 함수로 포인터 지정*

CrossFire() : CrossFireBase<T>(n, 0), ba(**unique\_ptr<T[]>**(**new** T[n\*n]))

{

**this**->setBombPtr(ba.get());

}

void shakeFire();

**private**:

*//boost::scoped\_array 대신 다음처럼 구현한다.*

*//스마트 포인터로 배열을 사용하는 방법이다. 벡터를 이용한 방법 - [항목13]*

**unique\_ptr<T[]>** ba;

};

# 중복해서 쓰기 vs. 기본 클래스로 분리하기

무작정 기본 클래스로 분리하라는 것이 아니다. 무엇을 고려해야 할까?

#### 그냥 중복해서 쓰기

위의 최적화가 공짜는 아니다. 비타입 매개변수는 컴파일 시점에 투입되므로 상수가 된다. 이는 *상수 전파* 등의 최적화가 먹혀 들어갈 수 있다.

###### 상수 전파(Constant Propagation)

고정된 값을 갖는 변수를 상수로 대체한다. 컴파일 시간에 계산이 가능한 연산을 계산함으로써 실행 시간과 코드의 크기를 줄일 수 있다.

#### 기본 클래스로 분리하기

일단 실행 코드가 작아진다. 처음부터 이를 목적으로 만든 것이다. 이렇게 되면 명령어 캐시 내의 ***참조 지역성***(프로세스의 메모리 참조가 특정 시점이나 부분에 집중되는 특성, *시간적 지역성*과 *공간적 지역성*이 있다.)이 향상된다.   
반면 객체의 크기는 늘어난다. 위에서 기본 클래스의 포인터의 크기가 그것이 된다.

# 비타입이 아닌 타입 매개변수로 인한 코드 비대화

T로 인해서도 코드 비대화가 생긴다는 의미다. vector<int>와 vector<long>이 한 예로, int와 long은 이진표현이 같아서 같은 코드를 2번 생성하게 한다.   
또한 포인터 타입을 매개변수로 하는 경우가 그렇다. 포인터의 이진 표현은 모두 같기 때문이다. **그래서 타입 제약이 엄격한 포인터(T\*)를 써서 동작하는 멤버 함수를 구현할 때는 하단에서 타입미정 포인터(void\*)로 동작하는 버전을 호출하는 식으로 만든다.** 아래에서 설명하지만 타입미정 포인터는 어떠한 타입도 형변환하여 사용할 수 있는데 이를 이용하여 모든 타입에 대해서 동작하게 한다. 즉, 하나의 기본 함수를 공용하고 있는 것이다. C++ 표준 라이브러리의 vector, queue, list등의 템플릿에 대해 이렇게 구현되어 있다.

###### 타입미정 포인터(void\*)

void 포인터변수는 가리키는 자료형이 정해져 있지 않으므로 모든 포인터를 저장할 수 있다.

int main(){

int a = 5;

double b = 7.0;

void\* vp;

*//어떤 타입의 주소를 가리키든 가능합니다.*

vp = &a;

vp = &b;

}

그러나 역시 가리키는 자료형에 대한 정보가 없으므로 참조연산이나 정수값 연산이 불가능하다. 참조연산의 경우는 메모리의 특정 번지로 가서 몇 바이트를 어떤 형태로 읽어야 할지 알 수 없으며, 정수값 연산도 얼마를 곱해서 더해야 하는지 알 수 없다.

따라서 void 포인터변수를 사용할 때는 원하는 형태로 변환하여 사용해야 한다.

*//형 변환 후, 값을 참조하여 출력*

printf("%d**\n**", \*(int \*)vp);

###### 함수 포인터

함수 포인터 제대로 쓰기: <http://zeph.tistory.com/155>

위의 void는 함수의 주소를 저장하는데에도 사용된다. 다음과 같이 저장한다.

void func1(){

printf("함수 포인터 예제!**\n**");

}

void func2(int a, int b){

printf("인자가 2개!**\n**");

}

int main(){

void(\*pf1)();

void(\*pf2)(int, int);

pf1 = func1;

pf2 = func2;

*//아래 2개는 같은 것입니다.*

func1();

pf();

}

#### 타입 미정 포인터로 코드 비대화 막기

이를 이용하여 위에 템플릿에서 타입미정 포인터로 동작하는 버전을 호출하게 해보자.

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

**class** **Archer**{

**public**:

*//타입미정 baseAttack을 호출하여 코드 비대화를 줄입니다.*

void attack(T target) { baseAttack((void\*)target); }

void baseAttack(void\* target) { cout << "this call." << endl; }

};

***템플릿을 사용하면 비슷비슷한 클래스와 함수가 여러 벌 만들어집니다. 따라서 템플릿 매개변수에 종속되지 않은 템플릿 코드는 비대화의 원인이 됩니다.***

***비타입 템플릿 매개변수로 생기는 코드 비대화의 경우, 템플릿 매개변수를 함수 매개변수 혹은 클래스 데이터 멤버로 대체함으로써 비대화를 종종 없앨 수 있습니다.***

***타입 매개변수로 생기는 코드 비대화의 경우, 동일한 이진 표현구조를 가지고 인스턴스화되는 타입들이 한 가지 함수 구현을 공유하게 만듦으로써 비대화를 감소시킬 수 있습니다.***

[항목45] 멤버 함수 템플릿

호환되는 모든 타입을 받아들이는 데는 멤버 함수 템플릿이 적절하다.

# 멤버 함수 템플릿(=멤버 템플릿)

멤버 함수 템플릿이란, 클래스 안에 있는 템플릿 함수다.

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

**class** **Factory**{

**public**:

*//Factory<T>라는 class의 멤버 함수다.*

*//그리고 U(모든 타입)를 받아들일 수 있다.*

**template**<**class** **U**>

Factory(**const** Factory<U>& other) {}

};

#### [동기] 이걸 왜 쓰는가?

스마트 포인터가 예로 적절하다.

class Item {};

class MasterPlan : public Item {};

std::shared\_ptr<Item> spI

= std::shared\_ptr<MasterPlan>(new MasterPlan);

분명 기본 클래스 타입으로 파생 클래스 객체를 받고 싶을 때가 있을 것이다. 하지만 같은 클래스라도 타입이 다르면 완전히 다른 클래스라는 것. 이처럼 암시적 변환을 지원하고 싶을 때, 템플릿버전 복사 생성자를 만들어야 한다.

#### [사용법1] 일반화 복사 생성자

*일반화 복사 생성자*는 **타입이 다른 객체로부터 원하는 타입의 객체를 만들어주는 생성자**를 뜻한다. 즉, Factory<U>를 Factory<T>로 만들 수 있다.

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

**class** **BladeMaster** :**public** Archer{};

**class** **DragonKnight** :**public** BladeMaster{};

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

**class** **Factory**{

**public**:

**explicit** Factory(T\* realPtr) : ptr(realPtr) {}

*//Factory<T>라는 class의 멤버 함수다.*

*//그리고 U(모든 클래스)를 받아들일 수 있다.*

**template**<**class** **U**>

Factory(**const** Factory<U>& other) {}

**private**:

T\* ptr;

};

int main(){

*//Archer <- BladeMaster로 변환: 원시 포인터의 암시적 변환*

Archer\* pA1 = **new** BladeMaster;

Archer\* pA2 = **new** DragonKnight;

*//기본적으로 템플릿은 2개의 클래스를 완전히 별개로 본다.*

*//****템플릿 멤버 함수****로 다른 타입의 Factory를 받아들일 수 있다.*

*//Archer <- BladeMaster로 변환: Factory에서도 암시적 변환 가능*

Factory<**Archer**> pF = Factory<**BladeMaster**>(**new** **BladeMaster**);

}

서로 다른 타입이지만 템플릿 안의 템플릿이 2개의 타입을 허용한 것이다.

이는 암시적 변환을 가능하게 한다. 하지만 불가능한 암시적 변환을 허용할 수 없다. 그러므로 문법적인 부분을 검토해주어야 한다. **멤버 초기화 리스트를 이용하면 이를 검토할 수 있다.** 리스트에 원시 포인터를 대입하는 것이다. 아래 코드에 적용된 모습이 보인다.

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

**class** **Factory**{

**public**:

**explicit** Factory(T\* realPtr) : ptr(realPtr) {}

**template**<**class** **U**>

Factory(**const** Factory<U>& other) : ptr(other.get()){}

T\* get() **const** { **return** ptr; }

**private:**

T\* ptr;

};

int main(){

*//암시적 변환 가능*

Factory<Archer> pF = Factory<BladeMaster>(**new** BladeMaster);

*//error C2440: '초기화 중': 'Archer \*'에서 'BladeMaster \*'(으)로 변환할 수 없습니다. 아래의 코드는 컴파일되지 않는다.*

*//원시 포인터를 대입함으로써 문법상 가능한 암시 변환만 가능*

Factory<BladeMaster> pFB = Factory<Archer>(**new** Archer);

}

###### Question. 암시적 변환을 원하는데 explicit을 왜 쓰는 거지?

explicit의 정확한 의미를 다시한번 짚어보자.

**explicit specifier:** 명시적 지정자는 생성자 또는 변환 함수 (C ++ 11 이후)가 암시적 변환 또는 복사 초기화를 허용하지 않도록 지정합니다. 클래스 정의 내에서 그러한 함수 선언의 decl-specifier-seq 내에 만 나타날 수 있습니다. – cppreference.com

struct A

{

A(int) { } // converting constructor

A(int, int) { } // converting constructor (C++11)

operator bool() const { return true; }

};

struct B

{

explicit B(int) { }

explicit B(int, int) { }

explicit operator bool() const { return true; }

};

int main()

{

A a1 = 1; // OK: copy-initialization selects A::A(int)

A a2(2); // OK: direct-initialization selects A::A(int)

A a3{ 4, 5 }; // OK: direct-list-initialization selects A::A(int, int)

A a4 = { 4, 5 }; // OK: copy-list-initialization selects A::A(int, int)

A a5 = (A)1; // OK: explicit cast performs static\_cast

if (a1); // OK: A::operator bool()

bool na1 = a1; // OK: copy-initialization selects A::operator bool()

bool na2 = static\_cast<bool>(a1); // OK: static\_cast performs direct- initialization

// B b1 = 1; // error: copy-initialization does not consider B::B(int)

B b2(2); // OK: direct-initialization selects B::B(int)

B b3{ 4, 5 }; // OK: direct-list-initialization selects B::B(int, int)

// B b4 = {4, 5}; // error: copy-list-initialization does not consider B::B(int,int)

B b5 = (B)1; // OK: explicit cast performs static\_cast

if (b2); // OK: B::operator bool()

// bool nb1 = b2; // error: copy-initialization does not consider B::operator bool()

bool nb2 = static\_cast<bool>(b2); // OK: static\_cast performs direct- initialization

}

template<typename U>

explicit Factory(U\* realPtr)

template<class U>

Factory(const Factory<U>& other)

위의 두 표현식이 말하고자 하는 것은 무엇일까? 포인터를 받는 멤버 함수 템플릿은 explicit이 선언되어 암시적 변환을 막는다. 스마트 포인터가 다른 기본 포인터로 변하는 식의 동작은 막겠다는 의미가 된다. 아래처럼 상대가 같은 스마트 포인터일때만 암시적 변환 혹은 복사 대입이 가능하다는 말이다.

Factory<Archer> pF = new BladeMaster; //이런건 안 됩니다.

Factory<Archer> pF = Factory<BladeMaster>(new BladeMaster); //이런건 가능합니다.

#### [사용법2] 일반화 복사 대입 연산자

위의 다른 타입으로 생성자를 만들 수 있는 것처럼, 대입도 다른 타입에 대입할 수 있어야 한다. 생성자와 마찬가지로 정의해준다.

한가지 주의할 점은 컴파일러는 **일반화 생성/대입 연산자와 비타입 생성/대입 연산자와 구별**한다는 것이다. 그러므로 일반화 버전만 정의해주면 컴파일러가 비타입 버전을 자동으로 만들어내어 곤란하게 할 수 있으므로 비타입 버전도 사용자가 함께 정의해 주어야겠다. 이것을 적용하여 다음은 모든 타입에 대하여 생성/대입이 가능한 완성된 코드이다.

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

**class** **Factory**{

**public**:

**explicit** Factory(T\* realPtr) : ptr(realPtr) {}

**template**<**class** **U**>

Factory(**const** Factory<U>& other) : ptr(other.get()){}

*//복사 대입 연산자: 컴파일러가 허접한거 만들기 전에 우리가 만듭니다.*

Factory& **operator**=(**const** Factory& of){ ptr = of.get(); }

*//일반화 복사 대입 연산자*

**template**<**class** **U**>

Factory& **operator**=(**const** Factory<U>& of){ ptr = of.get(); }

T\* get() **const** { **return** ptr; }

**private**:

T\* ptr;

};

***호환되는 모든 타입을 받아들이는 멤버 함수를 만들려면 멤버 함수 템플릿을 사용합시다.***

***일반화된 복사 생성 연산과 일반화된 대입 연산을 위해 멤버 템플릿을 선언했다 하더라도, 보통의 복사 생성자와 복사 대입 연산자는 여전히 직접 선언해야 합니다.***

[항목46] 템플릿內 비멤버 함수 사용

타입 변환이 바람직할 경우에는 비멤버 함수를 클래스 템플릿안에 정의해 두자.

# 함수 템플릿의 암시적 변환

연산을 적용할 때에는 매개변수에서 암시적 변환이 일어나면 편하다. [항목24]에서 이를 비멤버 함수로 놓았다.

#### 함수 템플릿 vs. 클래스 템플릿

비멤버 함수에 템플릿을 씌우면 함수 템플릿, 클래스에 템플릿을 씌우면 클래스 템플릿이다. 간단하다. 아래는 암시적 변환이 가능한 비멤버 함수를 보여준다.

**class** **Rational**{

**public**:

Rational(int v) : value(v) {};

int getValue() **const** { **return** value; };

**private**:

int value;

};

*//매개변수 안의 요소만 암시적 변환이 가능합니다.*

**const** Rational **operator**\*(**const** Rational &lhs, **const** Rational &rhs){

**return** Rational(lhs.getValue() \* rhs.getValue());

}

int main(){

Rational r1(5), r2(10), r3(1);

*//int->Rational로 암시적 변환이 일어납니다.*

*//암시적 변환으로 int형과 r1이 연산할 수 있습니다.*

r3 = r1 \* 2;

r3 = 2 \* r1;

}

결국, Rational과 int형은 연산이 가능해졌다. 하지만 Rational이 템플릿이어도 형변환이 일어날 수 있을까?

#### 템플릿이 포함된 비멤버 함수(함수 템플릿)의 매개변수에서는 암시적 변환이 일어나지 않는다.

이제 다양한 타입에 대해 유리수를 만들도록 여기에 템플릿을 씌워볼 것이다.

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

**class** **Rational**{

**public**:

Rational(int v) : value(v) {};

int getValue() **const** { **return** value; };

**private**:

int value;

};

*//비멤버 함수입니다.곧, 함수 템플릿이죠.*

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

**const** Rational<T> **operator**\*(**const** Rational<T> &lhs, **const** Rational<T> &rhs){

**return** Rational<T>(lhs.getValue() \* rhs.getValue());

}

int main(){

Rational<long long> r1(5), r2(10), r3(1);

***//컴파일 에러, 암시적 변환이 안 됩니다.***

r3 = r1 \* 2;

r3 = 2 \* r1;

}

왜 템플릿이 포함된다면 암시적 변환이 일어나지 않을까?

# 템플릿 인자 추론

함수 템플릿 호출시 타입을 명시적으로 지정하지 않더라도 **주어진 함수 인자에 기반해 템플릿 파라미터를 결정하는 것을 인자 추론**이라고 합니다.

사실 밑의 2가지 과정 외에도 많은 추론 과정에 관한 내용이 있다.

#### 일반적인 추론 과정

추론과정은 함수 인자의 데이터형과 함수 템플릿의 파라미터형을 비교해 하나 이상의 추론된 파라미터로 해당 인자를 올바르게 치환하는 과정입니다 각 인자-파라미터 쌍은 독립적으로 분석되며 끝에 가서 일치하지 않는다면 추론 과정은 실패합니다.

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

T **const**& max (T **const**& a, T **const**& b)

{

**return** a < b ? **b** : a;

}

int g = max(1, 1.0);

첫번째 호출인자는 int형이므로 원 max() 템플릿의 파라미터 T는 일단 int로 추론됩니다 그런데 두번째 호출인다는 double이므로 여기서는 T가 double이어야 합니다 그러면 두 결론은 서로 충돌합니다. **여기서 "추론과정이 실패했다"라고 하지 "프로그램이 유효하지 않다"라고 말하지 않는다는 것에 유념합시다**. 왜냐하면 **이후 max라는 다른 이름의 템플릿에서 추론 과정이 성공할 수 있기 때문입니다(함수 템플릿도 일반 함수처럼 오버로딩될 수 있다).**

#### 암시적 추론 과정

모든 추론된 템플릿 파라미터가 일관되게 결정되더라도 인자를 치환한 이후 유효하지 않은 함수가 생성된다면 추론과정은 실패합니다.

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

**typename** T::ElementT at (T **const**& a, int i )

{

**return** a[i];

}

void f( int\* p )

{

int x = at( p, 7 ); *//묵시적으로 템플릿파라미터를 인자추론 할 때*

}

여기서 T는 int로 결정된다(T가 나타난 파라미터형은 하나뿐이기 때문에 분석상에서 충돌이 일어날이가 없습니다) 하지만 T자리에 int를 치환한다면 반환형인 T::ElementT는 C++에서 정상적인 값을 가질 수 없으며 결국 추론과정은 실패합니다(SFINAE) 에러 메시지는 at()에 대한 호출과 일치하는 것을 찾을 수 없다고 말할 것입니다.

#### 위의 코드에서의 추론 과정

위의 코드로 돌아가보면, r3 = r1 \* 2;에서 r1과 2의 타입을 추론해야 할 것이다. 참고로 템플릿 인자 추론은 함수 템플릿에서만 생긴다. 그런데 이는 인자 추론(T가 무엇인지)이 아닌 Class<T>클래스의 타입 추론이다. 클래스 템플릿은 타입을 정해놓는데 추론할 이유가 없다. 이러한 함수 템플릿에서 r1은 이미 정의한 바와 같이 long long으로 추론할 수 있지만, **2를 추론하려면 2를 Rational로 미리 암시적 변환**을 수행해야 할 것이다. 하지만 이는 불가능하다. **템플릿 인자 추론 과정에서는 암시적 타입 변환이 고려되지 않는다.**

# 인자 추론 실패를 피하는 함수 템플릿 만들기

템플릿의 인자 추론 이해와 비멤버 함수의 이해를 통하여 만들 수 있다.

#### 프렌드 함수는 함수 템플릿으로서의 성격을 주지 않고 특정한 함수 하나를 나타낼 수 있다.

프렌드 함수는 함수 템플릿이 아니다. 왜냐하면 클래스 안에 있기 때문이다. **프렌드 함수는 클래스 안에서 비멤버 함수를 선언하는 유일한 방법이다.**

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

**class** **Rational**{

**public**:

Rational(T v) : value(v) {};

T getValue() **const** { **return** value; };

*//프렌드 함수는 함수 템플릿의 성격을 가지지 않습니다.*

*//클래스가 인스턴스화될 당시에 타입이 정해집니다.*

*//클래스 템플릿 내부의 Rational에는 <T>가 생략되어 있습니다.*

*//Rational타입이 정해지므로 매개변수 안에서 암시적 변환이 일어날 수 있습니다*

**friend**

*//반드시 정의를 해주어야 합니다. 이는* ***인라인 함수****가 됩니다.*

**const** Rational **operator**\*(**const** Rational& lhs, **const** Rational& rhs){

**return** lhs.getValue()\*rhs.getValue();

}

**private**:

T value;

};

int main(){

Rational<long long> r1(5), r2(10), r3(1);

*//이제 가능합니다.*

r3 = r1 \* 2;

r3 = 2 \* r1;

}

결국, 프렌드 함수는 클래스 내부에서 타입이 정해질 수 있는 보장, 비멤버 함수의 클래스의 생성 없이 매개변수로 받아들이는 성질이 합쳐져서 가능하다고 볼 수 있다.

#### 프렌드 함수의 도우미만 호출하게 만들기

위의 프렌드 함수는 인라인 함수를 호출하고 있다. 하지만 함수의 길이가 길어진다면 다음과 같은 구현이 효과적이다.

*//Rational<T>를 사용하므로 전방 선언이 필요합니다.*

**template**<**typename** T> **class** **Rational**;

*//도우미 함수 템플릿을 정의합니다. 비멤버 함수이며 인라인이 아닙니다.*

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

**const** Rational<T> **doMultiply**(**const** Rational<T>& lhs, **const** Rational<T>& rhs){

**return** lhs.getValue()\*rhs.getValue();

}

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

**class** **Rational**{

**public**:

Rational(T v) : value(v) {};

T getValue() **const** { **return** value; };

**friend**

*//템플릿과 인라인은 같은 헤더파일에 있지만, 서로 관련은 없습니다. - [항목30]*

**const** Rational **operator**\*(**const** Rational& lhs, **const** Rational& rhs){

**return** **doMultiply**(lhs, rhs);

}

**private**:

T value;

};

이는 처음에 비멤버 함수에서 생긴 암시적 변환에 대한 문제를 해결해준다. 왜냐하면 프렌드 함수인 operator\*에서 제대로 곱해질 수 있도록 매개변수를 처리하고 처리한 매개변수를 비멤버 함수로 전해주므로 적절하게 작동할 수 있다.

###### 템플릿과 인라인은 같은 헤더파일에 있지만, 서로 관련은 없습니다. - [항목30]

같은 헤더파일에 있다 해서 반드시 템플릿이 인라인은 아닙니다. 함수 템플릿이 곧 인라인은 아니며, 사용하려면 인라인 선언을 해야하고 또한 인라인으로 강제할 필요도 없다는 의미입니다.

***모든 매개변수에 대해 암시적 타입 변환을 지원하는 템플릿과 관계가 있는 함수를 제공하는 클래스 템플릿을 만들려고 한다면, 이런 함수는 클래스 템플릿 안에 프렌드 함수로서 정의합시다.***

[항목47] 템플릿과 특성정보 클래스

타입에 대한 정보가 필요하다면 특성정보 클래스를 이용하자.

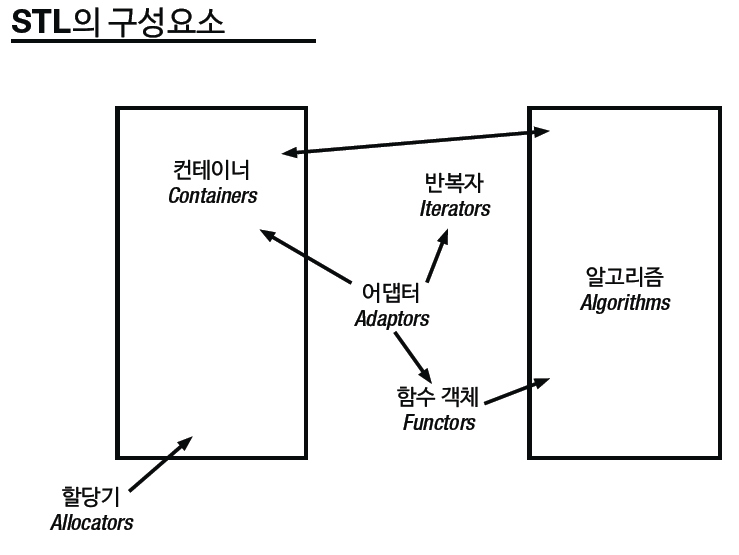
[논의] 효율성을 잃지 않은 추상화, 폰 노이만 구조 그리고 밸류 시멘틱스(value semantics)

# STL(Standard Template Library)

자주 쓰는 **자료 구조**와 **알고리즘**을 **템플릿**을 이용하여 구현한 **C++라이브러리**

#### STL 구성

  
위에는 2가지로 되어 있는 것처럼 보이지만 실제로는 다음과 같다.



각 요소에 대해서 간단하게 알아보겠다.

#### 컨테이너

STL에서 컨테이너의 역할은 간단히 말해 동일한 타입의 객체를 저장해두는 통이라고 보면 된다.

STL의 컨테이너에는 크게 세 가지의 종류가 있다.

*시퀀스 컨테이너(Sequence Container):* 자료를 선형적으로 관리하는 컨테이너이다. 대표적인 컨테이너로는 벡터(vector)와 리스트(list), 데크(deque) 세 가지가 있다.

*연관 컨테이너(Associative Container):* 자료를 일정한 규칙에 따라 관리하여 저장하는 컨테이너이다. 삽입되는 자료들을 항상 일정한 기준에 따라 저장하므로 검색 속도가 빠르다는 장점이 있다. 셋(set), 맵(map)등의 컨테이너가 대표적인 연관 컨테이너이다.

*어댑터 컨테이너(Adapter Container):* 시퀀스 컨테이너를 적절히 변형하여 자료를 미리 정해진 일정한 방식에 따라 관리하는 컨테이너다. 자료를 마음대로 넣고 뺄 수 없으며 컨테이너에서 정해놓은 방식대로만 조작할 수 있다. 대표적인 어댑터 컨테이너에는 큐(queue), 스택(stack), 우선순위 큐(Priority queue) 등이 있다.

#### 알고리즘

자료를 가지고 자주 활용하는 함수들이 포함되어 있다. 내용이 많지만 사전처럼 이용하길 바란다.

알고리즘에 포함된 함수는 모두 4개지로 분류할 수 있습니다.  
\* 변경 불가 순차 알고리즘(nonmutating sequence algorithm)

\* 변경 가능 순차 알고리즘(mutating sequence algorithm)

\* 정렬 관련 알고리즘(sorting-related  algorithm)

\* 범용 수치 알고리즘(generalized numeric algorithm)

변경 불가 순차 알고리즘(nonmutating sequence algorithm)

find / 원하는 요소를 검색합니다.

find\_if / 조건자를 true가 되게 하는 첫 번째 요소를 검색합니다.

adjacent\_find / 동일한 두 개의 원소가 이웃하고 있는 쌍을 검색합니다.

count / 특정 값과 일치하는 요소의 개수를 반환합니다.

count\_if / 조건자를 true가 되게 하는 요소의 개수를 반환합니다.

for\_each / 인자로 넘어간 함수를, 요소 전체에 대해서 호출합니다. for문과 같은 반복문을 대신 해서 사용할 수 있습니다.

equal / 두 개의 구간이 일치하면, true를 반환합니다.

mismatch / 두 개의 구간이 일치하지 않으면, true를 반환합니다.

search / 구간 내에 포함된 하위 구간을 검색합니다.  C 함수의 strstr 함수와 비슷한 역할을 합니다.

변경 가능 순차 알고리즘(mutating seqence algorithm)종류

copy / 특정 구간에 속하는 원소를 다른 구간으로 복사합니다. 구간이 중복될 경우, 왼쪽으로의 이동만 허용합니다.

copy\_backward / 특정 구간에 속하는 원소를 다른 구간으로 복사합니다. 구간이 중복될 경우, 오른쪽으로의 이동만 허용합니다.

fill / 구간에 속하는 모든 원소를 주어진 값으로 채웁니다.

fill\_n / 시작 위치에서부터 n 개의 요소를 주어진 값으로 채웁니다.

generate / 구간에 들어 있는 요소 각각에 대해 인자로 넘어간 함수를 호출하고, 그 결과로 다시 요소를 채웁니다. 구간 내의 요소는 함수의 결과값으로 채워집니다.

partition / 구간 내에서 주어진 값보다 작은 요소는 앞쪽에, 큰 요소는 뒤쪽에 오도록 재배치합니다.

stable\_partition / 구간 내에서 주어진 값보다 작은 요소는 앞쪽에, 큰 요소는 뒤쪽에 오도록 재배치하면서, 최초의 상대적인 위치를 그대로 유지합니다.

random\_shuffle / 구간 내의 요소를 난수로 채웁니다.

remove / 주어진 값과 일치하는 구간 내의 모든 요소를 삭제합니다. 그러나, 실제로 삭제되는 것이 아니고, 가장 뒤쪽으로 자리르 변경합니다. 실제 삭제는 erase 함수를 사용합니다.

replace / 주어진 값과 일치하는 구간 내의 요소들을 지정한 값으로 치환합니다.

reverse / 구간 내의 모든 요소들을 거꾸로 뒤집어서 재배치합니다.

rotate / 구간 내의 모든 요소를 주어진 위치를 중심으로 한 바퀴 돌립니다.

swap / 두 개의 값을 교환합니다.

swap\_ranges / 두 구간에 속하는 값들을 교환합니다.

transfrom / 구간에 속하는 요소 각각에 대하여, 인자로 넘어간 함수를 호출한 결과를 다른 구간에 저장합니다.

unique  / 구간 내에 속한 모든 중복된 요소를 제거함으로써, 해당 요소가 유일하도록 만듭니다. remove 함수처럼, 실제로 삭제가 일어나는 것이 아니고, 구간의 맨 뒤로 이동시켜 놓기 때문에, 컨테이너의 크기에는 변화가 없습니다.

정렬 관련 알고리즘(sorting-related algorithm) 종류

sort / 구간을 정렬합니다. (quick-sort)

stable\_sort / 구간을 정렬합니다. 동등한 요소의 상대적인 위치를 유지합니다. (merge-sort)

partial\_sort / 구간을 정렬합니다. 동등한 요소의 상대적인 위치를 유지하지 않습니다. (heap-sort)

nth\_element / 구간이 정렬되었을 경우, N번째 위치에 놓이게 될 요소만 놓고, 나머지 요소들은 N보다 작은 값들을 왼쪽에, 큰 값들을 오른쪽에 배치한다

binary\_search / 정렬되어 있는 구간으로부터 요소를 검색합니다. (이진 검색)

lower\_bound / 이진 검색을 사용해서 정렬된 구간을 검색한 후, 동일한 값들이 연속되어 있을 경우 맨 앞의 요소를 반환합니다.

upper\_bound / 이진 검색을 사용해서 정렬된 구간을 검색한 후, 동일한 값들이 연속되어 있을 경우 맨 마지막의 요소를 반환합니다.

equal\_range / 이진 검색으로 검색한 값이 동일하게 연속된 구간을 반환합니다. lower\_bound와 upper\_bound 함수의 결과를 합한 결과가 반환됩니다.

merge / 두 개의 정렬된 구간을 병합해서, 중복되지 않는 구간에 결과를 저장합니다.

inplace\_merge 두 개의 정렬된 구간을 병합해서, 중복되지 않는 구간에 결과를 저장합니다.

includes / 주어진 정렬된 구간의 요소들이 다른 구간에 속하는지를 반환합니다.

set\_union / 정렬된 두 개 구간의 합집합을 다른 구간에 저장합니다.

set\_difference / 정렬된 첫 번째 구간에 속한 요소들 중에서 정렬된 두 번째 구간에 속하지 않는 요소들을 다른 구간에 저장합니다.

set\_intersection / 정렬된 두 개의 구간에 공통으로 속한 요소들을 다른 구간에 저장합니다.

set\_symmetric\_defference / 정렬된 두 개의 구간에서, 한 쪽 구간에만 속해있는 요소들을 다른 구간에 저장합니다.

make\_heep / 구간 내의 요소들을 힙(heep)으로 재구축 합니다.

sort\_heep / 힙에 저장된 원소들을 정렬합니다.

push\_heap / 주어진 구간을 힙으로 유지하면서, 새로운 요소를 힙에 추가합니다.

pop\_heap / 주어진 구간을 힙으로 유지하면서, 가장 큰 요소를 삭제합니다.

min / 두 개의 요소 중, 작은 값을 반환합니다.

max / 두 개의 요소 중, 큰 값을 반환합니다.

lexicographical\_compare / 두 개의 구간을 사전 순서로 비교해서 결과를 반환합니다.

next\_permutation / 주어진 구간을 사전 순서에 기초해서, 다음 순열로 변경합니다.

prev\_permutation / 주어진 구간을 사전 순서에 기초해서, 이전 순열로 변경합니다.

범용 수치 알고리즘(generalized numeric algorithm) 종류

accumulate / 주어진 구간에 속하는 요소들의 합계를 구합니다.

partial\_sum / 주어진 구간의 요소를 기초로, 부분합을 구합니다.

adjacent\_difference / 주어진 구간의 요소를 기초로, 이웃하는 요소의 차이를 구합니다.

inner\_product / 주어진 두 구간의 내적(inner product)을 구합니다.

#### 반복자(iterator)

반복자란 컨테이너에 저장되어 있는 모든 원소들을 전체적으로 훑어나갈 때 사용하는, 일종의 포인터와 비슷한 객체라고 할 수 있다.

template<class \_Category,

class \_Ty,

class \_Diff,

class \_Pointer,

class \_Reference,

class \_Base>

struct \_Iterator012

: public \_Base

{ // base type for debugging iterator classes

typedef \_Category iterator\_category;

typedef \_Ty value\_type;

typedef \_Diff difference\_type;

typedef \_Pointer pointer;

typedef \_Reference reference;

};

반복자의 몇 가지 특징을 정리하면 다음과 같다.

###### 움직이면서 어느 지점을 가리킨다.

포인터가 특정 메모리 위치를 참조하는 것처럼, 반복자는 널 값을 가리킬 수도 있으며 컨테이너 안에서 어느 원소를 가리키게 된다. 반복자는 여러 연산자를 통해서 움직일 수 있다. 아래에서 자세히 살펴본다.

###### 구간(Range)을 지정할 때 사용한다.

연속적인 메모리 영역을 두개의 포인터로 나타내듯이, 한쌍의 반복자는 값들의 구간(range)을 설정하는데 사용될 수 있다.

이 구간은 도달 가능(반복자에 증가 연산자를 적용시키면 반복자는 움직일 수 있으며 이렇게 움직여서 반대편으로 도달할 수 있을 때 사용하는 용어)해야 하며, STL에는 구간을 나타내는 begin, end함수가 있다. 이 begin은 시작을 나타내지만 end는 끝의 다음 위치를 가리킨다는 점도 기억해할만 하다.

이 구간을 구현하면 다음과 같을 것이다.

template<long FROM, long TO>

class Range {

public:

// member typedefs provided through inheriting from std::iterator

class iterator : public std::iterator<

std::input\_iterator\_tag, // iterator\_category

long, // value\_type

long, // difference\_type

const long\*, // pointer

long // reference

> {

long num = FROM;

public:

explicit iterator(long \_num = 0) : num(\_num) {}

iterator& operator++() { num = TO >= FROM ? num + 1 : num - 1;

return \*this; }

iterator operator++(int) { iterator retval = \*this; ++(\*this);

return retval; }

bool operator==(iterator other) const { return num == other.num; }

bool operator!=(iterator other) const { return !(\*this == other); }

reference operator\*() const { return num; }

};

iterator begin() { return iterator(FROM); }

iterator end() { return iterator(TO >= FROM ? TO + 1 : TO - 1); }

};

int main() {

// std::find requires a input iterator

auto range = Range<15, 25>();

auto itr = std::find(range.begin(), range.end(), 18);

std::cout << \*itr << '\n'; // 18

// Range::iterator also satisfies range-based for requirements

for (long l : Range<3, 5>()) {

std::cout << l << ' '; // 3 4 5

}

std::cout << '\n';

}

###### 반복자의 종류

알고리즘마다 각기 다른 방식으로 컨테이너를 훑어가기 때문에, 반복자에도 여러가지 종류가 있게 된다. 프로그래머는 표준 라이브러리에서 제공하는 컨테이너들에 알맞는 반복자를 생성할 수 있다. 특정 알고리즘이 어떤 컨테이너와 함께 사용할 수 있는가는 어떤 종류의 반복자를 인자로 요구하느냐에 따라 결정된다.

|  |  |  |  |
| --- | --- | --- | --- |
| 반복자 | 이름 | 약어 | 설명 |
| 입력 | Input Iterator | InIt | 입력만 가능. 쓰는 것은 불가능하다. |
| 출력 | Output Iterator | OutIt | 출력만 가능. 읽는 것은 불가능하다. |
| 순방향 | Forward Iterator | FwdIt | 입출력 모두 가능, ++연산만 가능(뒤로 순회 불가) |
| 양방향 | Bidirectional Iterator | BiIt | 양방향 순회가 모두 가능하다. |
| 임의접근 | Random Iterator | RanIt | 임의의 요소를 참조할 수 있다. |

입력 반복자(Input Iterator)

전진만 가능하며, 한 번에 한 칸씩만 이동하며, 자신이 가리키는 위치에서 읽기만 가능하다. 즉, 이 반복자가 가리키는 원소가 무엇인지 알 수 있다. 횟수는 한 번뿐이다.

결국, 가장 단순한 형태의 반복자이며 기본 3가지 조건을 만족한다.  
1. 반복자는 다른 반복자와의 상등여부를 비교할 수 있어야 한다. 같은 위치를 가리키면 같은 것이고, 그렇지 않다면 다른 것이다.  
2. 반복자는 \* 연산자를 사용하여 반복자가 가리키는 값을 얻을 수 있어야 한다.  
3. 반복자는 ++ 연산자를 사용하여 다음 원소를 가리킬 수 있어야 한다.

사용 예시: 표준 라이브러리의 입력 스트림 반복자(input stream iterator): *istream\_iterator*

출력 반복자(Output Iterator)

입력 반복자와 비슷하며 출력용인 점만 다르다. 즉, 시퀀스에 값을 대입하는데 사용될 수 있지만, 값을 접근하는데는 사용될 수 없다.

사용 예시: *ostream\_iterator*

순방향 반복자(Forward Iterator)

순방향 반복자는 입력 반복자와 출력 반복자의 특징을 결합한 것이다. 값을 접근하고 갱신하는 것이 가능하게 된 것이다. 그것도 여러 번 가능하다. 순방향 반복자를 사용하는 함수로 replace() generic 알고리듬이 있는데, 이는 원하는 값을 찾아 다른 값으로 대치하는 작업을 한다. 이 알고리듬은 다음과 같이 작성될 수 있다.

**template** <**class** **ForwardIterator**, **class** **T**>

void replace (ForwardIterator first, ForwardIterator last,

**const** T& old\_value, **const** T& new\_value)

{

**while** (first != last) {

**if** (\*first == old\_value)

\*first = new\_value;

++first;

}

}

사용 예시: *STL’s 해시 컨테이너*

양방향 반복자(bidirectional iterator)

순방향 반복자와 상당히 비슷하다. 다만, 양방향 반복자는 뒤로도 갈 수가 있다. 예를 들어, 컨테이너 내의 값들의 순서를 뒤집어 새로운 컨테이너로 옮기는 함수에서는 양방향 반복자를 사용한다.

**template** <**class** **BidirectionalIterator**, **class** **OutputIterator**>

OutputIterator reverse\_copy (BidirectionalIterator first,

BidirectionalIterator last,

OutputIterator result)

{

**while** (first != last)

\*result++ = \*--last;

**return** result;

}

사용 예시: *STL’s list, set, multiset, map, multimap*

임의접근 반복자(random access iterator)

양방향 반복자에 “반복자 산술 연산” 수행 기능을 추가한 것이다. 즉, 한 칸씩이 아닌 임의의 거리만큼 상수시간에 할 수 있고 산술 연산처럼 포인터의 위치를 바꿀 수 있다. 기본제공 포인터를 이용하여 만들었다.

사용 예시: *STL’s vector, deque, string*

C++ 표준 라이브러리에는 위와 같은 5가지 반복자를 식별하기 위해 5가지 구조체가 쓰이고 있다.

<xutility.h>

**struct** input\_iterator\_tag

{

*// identifying tag for input iterators*

};

**struct** output\_iterator\_tag

{

*// identifying tag for output iterators*

};

**struct** **forward\_iterator\_tag**

: **public** input\_iterator\_tag, output\_iterator\_tag

{

*// identifying tag for forward iterators*

};

**struct** **bidirectional\_iterator\_tag**

: **public** forward\_iterator\_tag

{

*// identifying tag for bidirectional iterators*

};

**struct** **random\_access\_iterator\_tag**

: **public** bidirectional\_iterator\_tag

{

*// identifying tag for random-access iterators*

};

# 어떤 타입에 대한 정보를 알아야 할 때: 특성정보

위에서 반복자에는 다양한 종류가 있고 능력이 다르므로 구현할 때, 다음과 같은 문제가 생길 수 있다.

#### Problem. 어떤 타입인지 어떻게 알지?

advance() 함수는 하나의 반복자와 하나의 숫자값을 인자로 받아서, 반복자를 주어진 값만큼 이동시킨다.   
   
임의접근 반복자의 경우에, 위 함수는 iter + n; 과 같은 효과를 가진다. 그러나 위 함수는 모든 형태의 반복자에 대해 적용되어야 한다.

문제는 반복자가 어떤 타입이느냐에 따라 구현이 달라지므로 **타입에 대한 정보**를 알 필요가 있다.

#### Solution. 특성정보(traits) 다루기

특성정보란, **컴파일 도중에 어떤 주어진 타입의 정보를 얻을 수 있게 해주는** 객체를 지칭하는 개념이다. – *[항목48]*

###### 특성정보 요구사항

특성정보는 정의된 문법도 아니며 키워드도 아니지만 다음의 표준적인 방법이 존재한다.  
1. 기본제공 타입과 사용자 정의 타입에서 모두 돌아가게 한다.  
2. 해당 특성정보를 템플릿 및 그 템플릿의 1개 이상의 특수화 버전을 넣는다.  
3. 위의 템플릿을 구조체로 표현하며 이를 ***‘특성정보 클래스’***라고 부른다.

###### iterator\_traits에서의 특성 정보 사용

예시: **STL의 iterator\_traits의 구현**은 어떠한지 살펴본다. 위의 요구사항을 지킨 모습을 잘 살펴보아야 한다.

1. 사용자 정의(typedef) 반복자 타입에 대한 구현

다른 사람이 사용하도록 열어 주고 싶은 타입 관련 정보를 확인하고 그 이름을 선택합니다.

예시: 사용자 정의 타입으로 하여금 iterator\_category라는 이름의 타입을 내부에 가질 것을 요구한다.

*// 이터레이터 클래스... \_List\_const\_iterator도 이와 유사하다*

**template**<**class** **\_Mylist**>

**class** **\_List\_iterator** : **public** \_List\_const\_iterator<\_Mylist>

{ *// iterator for mutable list*

**public**:

**typedef** \_List\_iterator<\_Mylist> \_Myiter;

**typedef** \_List\_const\_iterator<\_Mylist> \_Mybase;

**typedef** bidirectional\_iterator\_tag **iterator\_category**;

*// ...*

};

**template**<**class** **\_Ty**, **class** **\_Ax** = allocator<\_Ty> >

**class** **list** : **public** \_List\_val<\_Ty, \_Ax>

{

**public**:

**typedef** \_List\_val<\_Ty, \_Ax> \_Mybase;

*// ...*

*// \_List\_const\_iterator 클래스의 iterator 사용*

*//iterator는* **\_List\_iterator***를 뜻하며 그 안에* **iterator\_category***라고 통일됨*

**typedef** \_List\_const\_iterator<\_Mybase> const\_iterator;

**typedef** \_List\_iterator<\_Mybase> iterator;

*// ...*

};

2. 위에서 사용자 정의한 타입을 똑같이 가진 iterator\_traits 구조체를 정의

지원하고자 하는 타입 정보를 담은 템플릿 구조체를 정의한다.

**template**<**class** **\_Iter**>

**struct** iterator\_traits

{

*// get traits from iterator \_Iter*

**typedef** **typename** **\_Iter::**iterator\_category iterator\_category;

**typedef** **typename** \_Iter::value\_type value\_type;

**typedef** **typename** \_Iter::difference\_type difference\_type;

**typedef** difference\_type distance\_type; *// retained*

**typedef** **typename** \_Iter::pointer pointer;

**typedef** **typename** \_Iter::reference reference;

};

3. 기본 제공 타입에 대해서도 특성정보를 사용할 수 있도록 부분 특수화 버전

2번에서 만든 구조체의 부분 특수화 버전을 만들어서 기본제공 타입도 지원한다.

**template**<**class** **\_Ty**>

**struct** iterator\_traits<\_Ty \*> *// 또는 <const \_Ty\*>*

{

*// get traits from pointer*

*// 포인터는 산술연산이 가능하므로, 바로 random\_access\_iterator\_tag를 지정해준다*

**typedef** random\_access\_iterator\_tag iterator\_category;

**typedef** \_Ty value\_type;

**typedef** ptrdiff\_t difference\_type;

**typedef** ptrdiff\_t distance\_type; *// retained*

**typedef** \_Ty \*pointer;

**typedef** \_Ty& reference;

};

#### 컴파일 도중에 조건처리: 오버로딩

위에서 특성정보 클래스를 이용하여 컴파일 도중에 타입을 확인할 수 있었다. 그런데 확인한 타입에 따라 다른 함수를 적용해야 하는데 이는 어떻게 하는가? 즉, 컴파일 도중에 if문을 적용해야 하는데 이 역할을 **오버로딩**이 담당한다.

각 타입에 따라 어떤 함수를 불러올지 컴파일 타임에 결정된다. 아래 코드는 Advance함수가 각 반복자에 따라 다른 함수를 불러옴을 보여준다.

*// \_Iter 인자의 iterator\_category 객체를 생성하여 반환한다.*

*// 중첩 의존 이름에 붙은 typename을 모르면 곤란하다*

**template**<**class** **\_Iter**> **inline**

**typename** iterator\_traits<\_Iter>::iterator\_category \_Iter\_cat(**const** \_Iter&)

{

*// return category from iterator argument*

**typename** iterator\_traits<\_Iter>::iterator\_category \_Cat;

**return** (\_Cat);

}

*// advance 함수는 \_Iter\_cat으로 구해진 iterator\_tag와 함께 \_Advance 내부 함수를 호출한다. 어떤 반복자이든 각자의 iterator\_category(iterator\_tag)을 반환한다.*

*// 즉, iterator\_tag별 적합하게 선택된 오버로드된 \_Advance를 호출할 수 있는 것이다.*

**template**<**class** **\_InIt**, **class** **\_Diff**>

**inline** void advance(\_InIt& \_Where, \_Diff \_Off)

{

*// increment iterator by offset, arbitrary iterators*=

\_Advance(\_Where, \_Off, \_Iter\_cat(\_Where));

}

/-

각 iterator\_tag로 overload된 함수들

\*-

**template**<**class** **\_InIt**, **class** **\_Diff**>

**inline** void \_Advance(\_InIt& \_Where, \_Diff \_Off, input\_iterator\_tag)

{

*// ...*

}

**template**<**class** **\_FI**, **class** **\_Diff**>

**inline** void \_Advance(\_FI& \_Where, \_Diff \_Off, forward\_iterator\_tag)

{

*// ...*

}

**template**<**class** **\_BI**, **class** **\_Diff**>

**inline** void \_Advance(\_BI& \_Where, \_Diff \_Off, bidirectional\_iterator\_tag)

{

*// ...*

}

**template**<**class** **\_RI**, **class** **\_Diff**>

**inline** void \_Advance(\_RI& \_Where, \_Diff \_Off, random\_access\_iterator\_tag)

{

*// ...*

}

#### 특성 정보 사용하기: Skill타입 확인 후 그에 맞는 버전 호출

위의 특성정보 클래스를 만드는 방법을 배웠다. 이를 적용하여 작성한 코드이며 여기서 오버로딩에 관한 내용은 바로 밑에 있다.

*//Skill의 tag들입니다. Skill의 속성을 표현합니다.*

**struct** fire\_skill\_tag{};

**struct** water\_skill\_tag{};

**struct** wind\_skill\_tag{};

**struct** **holy\_skill\_tag** : **public** wind\_skill\_tag{};

**struct** **pain\_skill\_tag** : **public** fire\_skill\_tag, water\_skill\_tag{};

**struct** **rainbow\_skill\_tag** : **public** pain\_skill\_tag{};

**class** **Striker**{};

**class** **Builder**{};

*//Archer은 불속성 스킬을 구사합니다.*

**template**<**typename** C>

**class** **Archer**{

**public**:

*//사용자 정의*

**class** **Skill**{

**public**:

*//****skill\_category****라는 이름의 타입을 가지는 것이 요구사항*

**typedef** fire\_skill\_tag **skill\_category**;

};

};

*//블레이드 마스터는 고통속성 스킬 구사*

**template**<**typename** C>

**class** **BladeMaster**{

**public**:

**class** **Skill**{

**public**:

**typedef** pain\_skill\_tag **skill\_category**;

};

};

*//****스킬 특성정보 클래스****입니다.*

*//컴파일 도중에 소유한 스킬이 어떤 타입인지 알아내야 합니다.*

**template**<**typename** Skill>

**struct** skill\_traits{

*//어떠한 Skill타입이든* ***skill\_category****로 통합니다.*

**typedef** **typename** Skill::skill\_category **skill\_category**;

};

*//위와 같은 코드는 사용자 정의 객체는 작동하지만 기본타입에는 작동하지 않습니다.*

*//****부분 템플릿 특수화****를 이용합니다. 타입이 포인터라면 아래 함수로 작동합니다.*

**template**<**typename** Skill>

**struct** skill\_traits<Skill\*>{

**typedef** rainbow\_skill\_tag **skill\_category**;

};

*//스킬을 사용하는 함수입니다.*

*//특성정보 클래스로 어떤 타입의 스킬인지 먼저 확인해야 합니다.*

**template**<**typename** Skill>

void casting(Skill& s){

*//typeid: 타입 이름을 사용하기 위해 쓰는 예약어군요.*

*//이는 런타임에 작동하며 type이 컴파일 타임에 정해지는 점을 감안하면 낭비입니다.*

*/\*if (typeid(typename skill\_traits<Skill>::skill\_category)*

*== typeid(fire\_skill\_tag)) cout << "fire" << endl;\*/*

*//****오버로딩****하면 컴파일 타임에 타입을 검사할 수 있습니다.*

*//타입에 따른 적절한 doCasting을 호출합니다.*

doCasting(s,

*//skill\_category의 본질은 구조체이기 때문에 ()가 존재합니다.*

**typename** skill\_traits<Skill>::**skill\_category**());

}

**template**<**typename** Skill>

void doCasting(Skill& s, fire\_skill\_tag){

cout << "overload fire" << endl;

}

**template**<**typename** Skill>

void doCasting(Skill& s, pain\_skill\_tag){

cout << "overload pain" << endl;

}

**template**<**typename** Skill>

void doCasting(Skill& s, rainbow\_skill\_tag){

cout << "overload rainbow" << endl;

}

int main(){

*//iterator에서의 경우*

*/\*vector<int>::iterator iter;*

*advance(iter, 1);\*/*

*//Archer클래스 내부의 Skill객체를 생성하고 이를 casting합니다.*

Archer<Striker>::Skill archer\_s;

casting(archer\_s); *//overload fire*

BladeMaster<Striker>::Skill blade\_s;

casting(blade\_s); *//overload pain*

int temp = 5;

int \*p = &temp;

*//int\*가 casting을?*

*//포인터이므로 부분 특수화의 함수가 적용됩니다.*

casting(p); *//overload rainbow*

}

***특성정보 클래스는 컴파일 도중에 사용할 수 있는 타입 관련 정보를 만들어냅니다. 또한 특성정보 클래스는 템플릿 및 템플릿 특수 버전을 사용하여 구현합니다.***

***함수 오버로딩 기법과 결합하여 특성정보 클래스를 사용하면, 컴파일 타임에 결정되는 타입별 if…else 점검문을 구사할 수 있습니다.***

[항목48] 템플릿 메타프로그래밍

하지 않겠는가?

# 템플릿 메타프로그래밍(Template MetaProgramming, TMP)

템플릿을 사용하는 프로그래밍 기법이다. 템플릿은 컴파일 타임에 정해지지 않는가? 그렇다. 이는 **컴파일 도중**에 실행되는 **템플릿 기반**의 프로그램을 작성하는 일이다.

#### TMP의 장점

기존에 런타임에 실행할 작업들을 컴파일 시점에 동작을 다 해가지고 오므로 실행 코드가 적어지고 메모리도 효율적이다. 그리고 런타임 중 발생할 에러를 컴파일 타임에 잡을 수 있다.

#### TMP의 단점

컴파일 시간이 길어진다. 가독성이 좋지 않다.

#### TMP의 튜링 완전성

튜링 완전성이란, 튜링 머신과 동일한 계산 능력을 가졌다는 의미인데 튜링 장치가 갖는 역할을 모두 할 수 있어야 한다. 쉽게 말해 다음 2가지가 가능하면 된다. 별 거 없다.  
1. 조건 분기문이 있다. if (...) then goto (...). 또는 branch if zero 등. for, while 등의 루프문은 모두 조건 분기문으로 바꿔 쓸 수 있다.  
2. 메모리의 임의 위치의 값을 바꿀 수 있다.

이를 풀어보면 ***TMP는 위의 2가지가 가능하다는 말***이다. [항목47]에서 if문을 오버로딩으로 컴파일 타임에 조건을 검사할 수 있음을 확인했지만 동작은 런타임이므로 연산을 수행할 수는 없다.

template<int n>

struct Fibo

{

enum

{

MOD = 1000000007,

value = (Fibo<n - 1>::value + Fibo<n - 2>::value) % MOD

};

};

template<>

struct Fibo<0>

{

enum { value = 1 };

};

template<>

struct Fibo<1>

{

enum { value = 1 };

};

###### TMP에서의 조건

**템플릿 특수화**로 가능하게 한다. 예를 들어, 위 코드에서 if(n == 1)임을 확인하고 싶다면, n이 1일 경우에의 특수화하여 Fibo<1>을 호출하게 한다.

###### TMP에서의 반복

TMP에서 진정한 반복의 의미는 없지만 재귀로 루프를 구현하면 된다. 물론, 일반적인 재귀 함수를 호출하는 것이 아닌 ***재귀식 템플릿 인스턴스화***를 해야 한다. 위의 코드에서 Fibo<n> = Fibo<n - 1>::value + Fibo<n - 2>::value 식으로 재귀를 호출하는 부분이 보인다.

#### 기본 예제

재귀식 템플릿 인스턴스화를 이용한 factorial을 구현한다.

*//기본 재귀 함수*

int factorial(int n){

**if** (n == 0) **return** 1;

**return** n \* factorial(n - 1);

}

*//TMP를 이용한* ***템플릿 인스턴스화***

**template**<int n>

**struct** Factorial{

*//****나열자 둔갑술****: 컴파일 타임에* ***변수를 담는 역할****을 한다.*

**enum**{ value = n \* Factorial<n - 1>::value };

};

*//****부분 템플릿 특수화****: 0에 대해서는 예외처리를 한다.* ***if문****에 해당한다.*

**template**<>

**struct** Factorial<0>{

**enum**{ value = 1 };

};

int main(){

*//런타임 없이 10!을 출력*

cout << Factorial<10>::value << endl;

*//런타임 중 10!을 출력*

cout << factorial(10) << endl;

}

다음은 지수를 연산하는 예제이다.

template<unsigned int n, unsigned int e, unsigned int m = e % 2>

struct Exp

{

enum { value = n \* Exp<n, e - 1>::value };

};

template<unsigned int n, unsigned int e>

struct Exp<n, e, 0>

{

enum { temp = Exp<n, e / 2>::value, value = temp \* temp };

};

template<unsigned int n, unsigned int e>

struct Exp<n, e, 1>

{

enum { value = n \* Exp<n, e - 1>::value };

};

template<unsigned int n>

struct Exp<n, 1>

{

enum { value = n };

};

template<unsigned int n>

struct Exp<n, 0>

{

enum { value = 1 };

};

# TMP가 쓰이면 유용한 경우

TMP로 무엇을 할 수 있는지 살펴본다. 크게 3가지가 있다.

#### 치수 단위의 정확성 확인(선행 에러 탐지)

모든 치수 단위의 조합이 제대로 되어있는지 컴파일 동안에 확인할 수 있다. 예를 들어, 속력이 들어갈 자리에 거리 변수를 집어넣을 수 없게 할 수 있다는 것이다. 그리고 분수식 지수 표현이 지원된다고 한다.

현재는 예제를 만들 능력이 없다.

#### 행렬 연산의 최적화(표현식 템플릿)

행렬 연산과 같이 배열을 가진 객체에 대해 복잡한 연산이 필요하다면, TMP를 응용한 ***표현식 템플릿***으로 임시 객체를 없앨 수 있으며 루프도 합칠 수 있다.

###### 표현식 템플릿

*표현식 템플릿*은 숫자 배열 클래스를 지원하기 위해 고안된 템플릿 프로그래밍 기술이다. 배열 연산에서 높은 성능을 지원하기 위해 개발되었습니다.

int main()

{

SArray <double> x(1000), y(1000);

//...

for (int idx = 0; idx < x.size(); ++idx)

{

x[idx] = 1.2\*x[idx] + x[idx] \* y[idx];

}

}

그러나 이런 구현은 두가지 이유 때문에 비효울적으로 알려져 있습니다.  
1. 연산자의 모든 응용은 적어도 하나의 임시 배열을 생성한다.  
2. 연산자의 모든 응용은 인자와 결과 배열에 대해 부가적으로 탐색한다.

위와 같은 배열을 어떻게 효율적으로 연산을 할까요?

template<typename T>

class Scalar

{

public:

Scalar(const T& \_value) : value(\_value) {}

T operator[](int) const { return value; }

private:

const T& value;

};

template<typename T>

struct Traits

{

typedef const T& value\_type;

};

template<typename T>

struct Traits<Scalar<T> >

{

typedef const Scalar<T> value\_type;

};

template<typename T, typename OP1, typename OP2>

class Add

{

public:

Add(const OP1& op1, const OP2& op2) : op1\_inst(op1), op2\_inst(op2) {}

T operator[](int idx) const { return op1\_inst[idx] + op2\_inst[idx]; }

private:

typename Traits<OP1>::value\_type op1\_inst;

typename Traits<OP2>::value\_type op2\_inst;

};

template<typename T, typename OP1, typename OP2>

class Multifly

{

public:

Multifly(const OP1& op1, const OP2& op2) : op1\_inst(op1), op2\_inst(op2) {}

T operator[](int idx) const { return op1\_inst[idx] \* op2\_inst[idx]; }

private:

typename Traits<OP1>::value\_type op1\_inst;

typename Traits<OP2>::value\_type op2\_inst;

};

int main()

{

std::vector<double> x{ 1, 2, 3, 4, 5 }, y{ 6, 7, 8, 9, 10 };

//일반적인 런타임 계산

for (int idx = 0; idx < x.size(); ++idx)

{

x[idx] = 1.2\*x[idx] + x[idx] \* y[idx];

std::cout << x[idx] << '\n';

}

x = { 1, 2, 3, 4, 5 }, y = { 6, 7, 8, 9, 10 };

//표현식 템플릿을 이용

for (int idx = 0; idx < x.size(); ++idx)

{

Add<double,

Multifly<double, Scalar<double>, std::vector<double> >,

Multifly<double, std::vector<double>, std::vector<double> >

> ret(

Multifly<double, Scalar<double>, std::vector<double> >(1.2, x),

Multifly<double, std::vector<double>, std::vector<double> >(x, y)

);

std::cout << ret[idx] << '\n';

}

return 0;

}

핵심은 값을 전달하는 것이 아니라, 식 자체(클래스 타입으로 표현할 수 있음)를 전달하여 하나의 식으로 만드는 것이다.

근데 좀 잔인할 정도로 복잡하다. 간단하게 사용할 수 있도록 wrapper클래스를 만들자.

template<typename T>

class Scalar

{

public:

typedef T value\_type;

Scalar(const T& \_value) : value(\_value) {}

T operator[](int) const { return value; }

private:

const T& value;

};

//…

template<typename OP1, typename OP2>

auto operator\*(const OP1& op1, const OP2& op2)

{

static\_assert(std::is\_same<typename OP1::value\_type,

typename OP2::value\_type>::value,

"you can't matching type.");

return Multifly<typename OP1::value\_type, OP1, OP2>(op1, op2);

}

template<typename OP1, typename OP2>

auto operator+(const OP1& op1, const OP2& op2)

{

static\_assert(std::is\_same<typename OP1::value\_type,

typename OP2::value\_type>::value,

"you can't matching type.");

return Add<typename OP1::value\_type, OP1, OP2>(op1, op2);

}

int main()

{

//…

x = { 1, 2, 3, 4, 5 }, y = { 6, 7, 8, 9, 10 };

//표현식 템플릿을 이용

for (int idx = 0; idx < x.size(); ++idx)

{

auto ret2 = (Scalar<double>)1.2 \* x + x \* y;

std::cout << ret2[idx] << '\n';

}

return 0;

}

#### 맞춤식 디자인 패턴 구현의 생성

TMP를 사용한 ***정책 기반 설계***를 사용하면, 따로따로 마련된 설계상의 선택(policy, 정책)을 나타내는 템플릿을 만들 수 있다. 이는 서로 임의대로 조합되어 사용자 취향에 맞는 동작을 갖는 패턴으로 구현될 수 있다.

무슨 말인지 모르겠다.

***템플릿 메타프로그래밍은 기존 작업을 런타임에서 컴파일 타임으로 전환하는 효과를 냅니다. 따라서 TMP를 쓰면 선행 에러 탐지와 높은 런타임 효율을 손에 거머쥘 수 있습니다.***

***TMP는 정책 선택의 조합에 기반하여 사용자 정의 코드를 생성하는 데 쓸 수 있으며, 또한 특정 타입에 대해 부적절한 코드가 만들어지는 것을 막는 데도 쓸 수 있습니다.***

[항목49] new처리자 동작 원리

new처리자의 동작 원리를 제대로 알자.

# new연산자

new가 C++ 클래스 개체에 메모리를 할당하는 데 사용되면 메모리가 할당된 후 개체의 생성자가 호출됩니다.

#### new연산자 정의

//throw(1)

void\* operator new (std::size\_t size);

//nothrow(2)

void\* operator new (std::size\_t size, const std::nothrow\_t& nothrow\_value) noexcept;

//placement(3)

void\* operator new (std::size\_t size, void\* ptr) noexcept;

일반적으로 C++에서 동적 할당을 위해서 사용하며 delete로 메모리를 해제해야 한다.

int main(){

*//Archer클래스 객체 pA의 동적 할당 및 생성*

Archer \*pA = **new** Archer;

*//죽을 시, delete로 pA 메모리 해제*

**if** (!pA->getState()) **delete** pA;

**return** 0;

}

만약 생성 객체가 배열이면, 해제도 배열로 해주어야 한다. *– [항목16]*

new연산자는 3가지 방법으로 사용할 수 있다.

###### **throwing allocation(1)**

해당 크기의 모든 객체를 나타내도록 적절하게 정렬 된 크기 바이트의 저장소를 **할당**하고, 이 블록의 첫 번째 바이트에 대한 null이 아닌 **포인터를 반환**합니다. 실패하면 bad\_alloc예외가 발생합니다.

###### **nothrow allocation(2)**

**위의 (1)과 동일**합니다. 예외가 발생하면 예외를 throw하는 대신 null 포인터를 반환합니다.

###### **placement(3)**

새로 공간을 할당하지 않고, ptr공간에 생성자를 호출하여 ptr을 반환합니다.

그러나 new-expression에 의해 함수가 호출되면 적절한 초기화가 수행됩니다 (클래스 객체의 경우 기본 생성자 호출이 포함됨).

Question. 기존 객체와 다른 생성자를 호출할 경우

다른 객체도 할당할 수 있다. 하지만 컴파일러는 기존 객체로 인식하여 새로운 함수를 사용할 수는 없다. 하지만 생성자는 정상적으로 호출된다. 메모리 구조는 바뀌었지만 포인터는 그대로이기 때문이다.

**new**(pA)BladeMaster(2, 9); *// 메모리에는 BladeMaster가 있습니다.*

cout << pA->getState() << endl; *// 기존 객체의 함수는 가능*

*//cout << pA->getSword() << endl; // 이건 안 된다.*

예시는 아래와 같다.

int main()

{

Archer\* a1 = new Archer; // 1

Archer\* a2 = new(std::nothrow)Archer; // 2

new (a2) Archer; // 3

// 생성자를 호출하지 않고 공간만 할당한다.

Archer\* a3 = (Archer\*)::operator new (sizeof(Archer));

delete a1, a2, a3;

return 0;

}

###### Question. malloc vs. new

malloc참고: <http://dsnight.tistory.com/51>

생성자 호출 여부 차이다. malloc는 생성자를 호출하지 않고 메모리를 할당해준다. 배열에 할당할 때, new는 디폴트 생성자를 따로 마련해야 하는 반면, malloc는 그런 번거로움을 덜어준다.

int main(){

*//new연산자의 다차원 배열 동적 할당 및 초기화: 생성자를 호출해야 한다.*

Archer \*\*pGA = **new** Archer\*[100];

**for** (int i = 0; i < 100; i++)

pGA[i] = **new** Archer(i);

*//malloc의 다차원 배열 동적 할당 및 초기화*

*//객체 크기(sizeof(BM\*))에 배열 크기(100)를 곱한만큼 메모리 할당*

BladeMaster \*\*pGB = (BladeMaster\*\*)malloc(**sizeof**(BladeMaster\*)\* 100);

**return** 0;

}

#### new연산자가 메모리 할당에 실패할 경우

new연산자에서 메모리 할당에 실패했을 시, 어떻게 처리해야 할까?

위에서도 언급되듯이 C++에서는 이러한 경우, std::bad\_alloc예외를 던지거나(1), 널 포인터를 반환한다(2). *그리고 해당new에 맞는 delete를 호출하여 메모리 할당을 해제하여 안전하게 되돌려야 한다. – [항목52]* **근데 더 중요한 것은 동시에 에러 처리 함수를 불러온다는 점이다. 이것이 new처리자**이며, **std::set\_new\_handler**로 설정할 수 있다.

우리는 이 예외에 대해서 안전하게 처리해야만 안전한 프로그램을 만들 수 있다.

# new처리자

메모리 할당에 실패했을 시, new연산자는 예외를 던지며 할당 에러 처리 함수를 호출한다. 디폴트 값으로 bad\_alloc를 호출하며 사용자 정의 함수가 있다면 이를 우선적으로 호출한다. 이 사용자 정의 함수로써 우리가 할당 에러 처리 함수를 지정해주어야 하는데 이 함수가 ***new처리자(new\_handler)***이다. 그리고 new처리자를 설정하기 위한 함수가 필요한데 이것이 std에서는 ***set\_new\_handler()***로 제공된다.

**namespace** std{

*//typedef로 함수 포인터 재선언*

*//new\_handler라는 함수를 변수처럼 씁니다.*

**typedef** void(\*new\_handler)();

new\_handler set\_new\_handler(new\_handler p) **throw**();

}

위는 실제 new처리자의 핵심 구현 부분이다.

#### new처리자 사용 예시

위의 구현을 이용하여 아래처럼 사용하는 것이다.

#include *<iostream> // std::cout*

#include *<cstdlib> // std::exit*

#include *<new> // std::set\_new\_handler*

void no\_memory() {

std::cout << "Failed to allocate memory!**\n**";

std::exit(1);

}

int main() {

std::set\_new\_handler(no\_memory);

std::cout << "Attempting to allocate 1 GiB...";

char\* p = **new** char[1024 \* 1024 \* 1024];

std::cout << "Ok**\n**";

**delete**[] p;

**return** 0;

}

#### new처리자 적절하게 사용하는 방법

위에서는 프로그램을 끝내버렸다. 하지만 더 좋은 방법을 생각해내야 한다.

###### 사용할 수 있는 메모리를 더 많이 확보합니다.

프로그램이 처음 시작할 때, 메모리 블록을 크게 하나 할당해 놓았다가 new처리자가 가장 처음 호출될 때, 그 메모리를 쓰게 하는 방법이다.

###### 다른 new처리자를 설치합니다.

현재의 new처리자가 메모리 할당을 실패했을 시, set\_new\_handler로 새로운 new처리자를 설치하는 것이다. 그러면 현재와 다른 동작을 하게 될 것이다.

###### new처리자 설치를 제거합니다.

new처리자에 널 포인터를 넘기는 데, 그러면 메모리 할당 실패 시, 예외를 던지게 된다.

###### 예외를 던집니다.

bad\_alloc나 여기서 파생된 타입의 예외를 던집니다.

###### 복귀하지 않습니다.

위의 코드처럼 프로그램을 죽여버리는 것이다.

###### new처리자 사용 예시

결국, 다음처럼 만들 수 있다.

int main()

{

Archer\*\* a = new Archer\*[1];

try

{

std::set\_new\_handler([]()

{

//default값은 bad\_alloc예외를 던지는 것입니다.

//아래랑 똑같습니다.

std::cout << "너무 많아.\n";

throw std::bad\_alloc(); // 예외를 던집니다.

//std::abort(); // 프로그램을 죽일 수도 있습니다.

});

//new(a) Archer\*[100000000]; // 존재할 수 없는 일

//a = new Archer\*[100000000];

a = new Archer\*[100000000];

for (int i = 0; i < 100000000; i++)

a[i] = new Archer;

//a[i] = new(std::nothrow) Archer;

//위처럼 하면 메모리가 부족하면 a[i]는 널포인터가 된다.

//그리고 공간이 생길때까지 메모리 할당을 재시도한다.

}

catch(std::bad\_alloc& e)

{

delete[] a;

std::cout << "조금만 만들자.\n";

Archer\* a1 = new Archer; // 1

Archer\* a2 = new(std::nothrow)Archer; // 2

new (a2) Archer; // 3

Archer\* a3 = (Archer\*)::operator new (sizeof(Archer));

delete a1, a2, a3;

}

return 0;

}

//output

//너무 많아.

//조금만 만들자.

#### 다른 new처리자 만들기

**각 클래스의 타입에 따라서 메모리 할당 실패에 대한 처리를 다르게 하고 싶을 때** 그 클래스만의 new처리자를 만들어야 한다. 순서는 다음과 같다.  
1. 나만의 클래스에 operator new와 set\_new\_handler를 정의한다.  
//기존 것을 반환하고 새로운 것을 저장한다.

static std::new\_handler set\_new\_handler(std::new\_handler p) noexcept {

std::new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

2. 객체만의 new\_handler를 저장해둔다. 이는 모든 객체가 공용하므로 static으로 선언한다.  
static std::new\_handler currentHandler;  
  
3. 나만의 클래스의 new연산자를 정의한다. 여기에서 RAII객체의 핸들러 관리 클래스를 생성한다. 이는 예외가 발생해도 안전하게 핸들을 되돌림을 보장한다.

//new연산자를 재정의합니다.

static void\* operator new(size\_t size) throw(bad\_alloc) {

NewHandlerHolder h(std::set\_new\_handler(currentHandler));

return operator new(size);

}

class NewHandlerHolder {

public:

//현재의 new처리자를 획득합니다.

explicit NewHandlerHolder(std::new\_handler nh) : handler(nh) {}

//전역 처리자로 기존의 처리자를 해제하고 현재의 처리자로 설치합니다.

~NewHandlerHolder() { std::set\_new\_handler(handler); }

private:

std::new\_handler handler;

NewHandlerHolder(const NewHandlerHolder&) = delete;

NewHandlerHolder& operator=(const NewHandlerHolder&) = delete;

};  
  
4. RAII객체의 핸들 교환 과정은 이러하다. 클래스에 저장된 현재 핸들을 std::set\_new\_handler에 설정하고 기존 핸들은 RAII객체에 집어넣는다. 이제 전역 new연산자를 호출한다. 그럼 메모리 부족시, 현재 핸들이 호출된다. 그리고 호출이 끝나면, RAII객체는 예외가 발생하여도 소멸시에 기존 핸들을 다시 std::set\_new\_handler에 설정하게 된다.   
  
**결국, 이 객체에서만 이 객체만의 핸들러를 쓰고, 다시 기존 핸들러를 반환하게 된다.**

*//new처리자를 담아놓는 객체입니다.*

*//자동으로 소멸하면 처리자로 변경합니다.*

**class** **NewHandlerHolder**{

**public**:

*//현재의 new처리자를 획득합니다.*

**explicit** NewHandlerHolder(new\_handler nh) : handler(nh) {}

*//전역 처리자로 기존의 처리자를 해제하고 현재의 처리자로 설치합니다.*

~NewHandlerHolder(){ std::set\_new\_handler(handler); }

**private**:

new\_handler handler;

*//복사를 방지합니다.*

NewHandlerHolder(**const** NewHandlerHolder&);

NewHandlerHolder& **operator**=(**const** NewHandlerHolder&);

};

**class** **Archer**{

**public**:

*//정적으로 set\_new\_handler을 정의합니다. 기존과 동작은 같습니다.*

*//받은 p처리자를 설치하며 old처리자를 반환합니다.*

**static** new\_handler set\_new\_handler(new\_handler p) **throw**(){

new\_handler oldHandler = currentHandler;

currentHandler = p;

**return** oldHandler;

}

*//new연산자를 재정의합니다.*

**static** void\* **operator** **new**(size\_t size) **throw**(bad\_alloc){

*//RAII에 기존 핸들을 보관해두고, 지금은 currentHandler를 사용합니다.*

NewHandlerHolder h(std::set\_new\_handler(currentHandler));

*//전역의 new연산자로 메모리를 할당합니다.*

**return** **operator** new(size);

}

**private**:

**static** new\_handler currentHandler;

};

*//null로 초기화합니다.*

new\_handler Archer::currentHandler = 0;

###### 사용자 new처리자를 재활용하고 싶다.

이렇게 만들어놓은 처리자를 한번쓰고 버리기 아깝다. 다른 클래스에도 그대로 적용되게 하고 싶다. 상속을 이용하는 것이다. (위의 Archer은 Support가 되었다.)

class Archer : public NewHandlerSupport

문제 인식

하지만 문제가 있다. 핸들러는 static으로 선언되어 있으며, 모든 객체가 공유하는 꼴이 된다. 각 클래스끼리만 사용할 수 있는 new처리자를 만들 수 없을까? 각 클래스별로 모두 만들어주어야 하는 것일까?

문제 해결

바로 템플릿 인스턴스화를 이용하는 것이다. 클래스별로 틀을 만들어준다. 바로 이것이다.

template<typename T>

class NewHandlerHolder {

public:

explicit NewHandlerHolder(std::new\_handler nh) : handler(nh) {}

~NewHandlerHolder() { std::set\_new\_handler(handler); }

private:

std::new\_handler handler;

NewHandlerHolder(const NewHandlerHolder&) = delete;

NewHandlerHolder& operator=(const NewHandlerHolder&) = delete;

};

template<typename T>

class NewHandlerSupport {

public:

static std::new\_handler set\_new\_handler(std::new\_handler p) noexcept {

std::new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

static void\* operator new(size\_t size) throw(std::bad\_alloc) {

NewHandlerHolder h(std::set\_new\_handler(currentHandler));

return operator new(size);

}

private:

static std::new\_handler currentHandler;

};

template<typename T>

std::new\_handler NewHandlerSupport<T>::currentHandler = 0;

class Archer : public NewHandlerSupport<Archer>

{

public:

private:

};

What is the curiously recurring template pattern (CRTP)?

묘하게 반복되는 템플릿 패턴이라고 불리는 이 원리가 사용된 것이다.

*싱글톤에서의 응용*

template <class ActualClass>

class Singleton

{

public:

static ActualClass& GetInstance()

{

if(p == nullptr)

p = new ActualClass;

return \*p;

}

protected:

static Actualclass\* p;

private:

Singleton(){}

Singleton(Singleton const &);

Singleton& operator = (Singleton const &);

};

template <class T>

T\* Singleton<T>::p = nullptr;

각 템플릿에 적용되는 싱글톤을 쉽게 만들 수 있다.

*Imagine you want to provide only operator < for your classes but automatically operator == for them!*

you could do it like this:

template<class Derived>

class Equality

{

};

template <class Derived>

bool operator == (Equality<Derived> const& op1, Equality<Derived> const & op2)

{

Derived const& d1 = static\_cast<Derived const&>(op1);//you assume this works

//because you know that the dynamic type will actually be your template parameter.

//wonderful, isnit it?

Derived const& d2 = static\_cast<Derived const&>(op2);

return !(d1 < d2) && !(d2 < d1);//assuming derived has operator <

}

Now you can use it like this

struct Apple:public Equality<Apple>

{

int size;

};

bool operator < (Apple const & a1, Apple const& a2)

{

return a1.size < a2.size;

}

now, you haven't provided explicitly operator == for apple? But you have it! You can write

int main()

{

Apple a1;

Apple a2;

a1.size = 10;

a2.size = 10;

if(a1 == a2) //the compiler won't complain!

{

}

}

This could seem that you would write less if you just wrote operator == for Apple, but imagine that the Equality template would provide not only == but >, >=, <= etc. And you could use these definitions for multiple classes, reusing the code!

신기하다. Equality연산이 모든 객체에 대해 적용될 수 있는 것이다. 자신의 클래스에만 해당되는 특별한 연산이 말이다.

#### 예외가 발생시키지 않는다.

a = new Archer\*[100000000];

for (int i = 0; i < 100000000; i++)

a[i] = new(std::nothrow) Archer;

//위처럼 하면 메모리가 부족하면 a[i]는 널포인터가 된다.

//그리고 공간이 생길때까지 메모리 할당을 재시도한다.

이 부분에서도 알 수 있듯이, 널 포인터를 반환하며 공간이 할당될 때까지 반복한다. 정말로 예외에 대해 안전할 수 있을까?

new(std::nothrow) Archer에 대해서는 예외 안전이지만 생성자에서 발생할 수 있는 예외는 고려하지 못한다. **결국, 예외에 안전하지 못하다.**

#### 결론

위 2가지 대처를 모두 이해하고, 상황에 맞게 사용할 수 있어야 할 것이다. 중요한 것은 메모리 할당 실패에 안전해야 한다는 것이다.

***set\_new\_handler함수를 쓰면 메모리 할당 요청이 만족되지 못했을 때 호출되는 함수를 지정할 수 있습니다.***

***예외불가(nothrow) new는 영향력이 제한되어 있습니다. 메모리 할당 자체에만 적용되기 때문입니다. 이후에 호출되는 생성자에서는 얼마든지 예외를 던질 수 있습니다.***

[항목50] 언제 new, delete를 바꿀까

왜 컴파일러가 작성해준 operator new와 delete를 바꾸고 싶은 것일까?

# new, delete를 바꾸는 이유

#### 잘못된 힙 사용을 탐지하기 위해

발생할 수 있는 문제1: 한번 new한 메모리에 2번 이상 delete한다.

해결: 사용자 정의 new를 사용하면 할당된 메모리 주소 목록을 기록해두면 delete에서 중복처리할 위험을 방지할 수 있다.

발생할 수 있는 문제2: 데이터 오버런(overrun, 할당된 메모리 블록의 끝을 넘어 뒤에 기록하는 것), 데이터 언더런

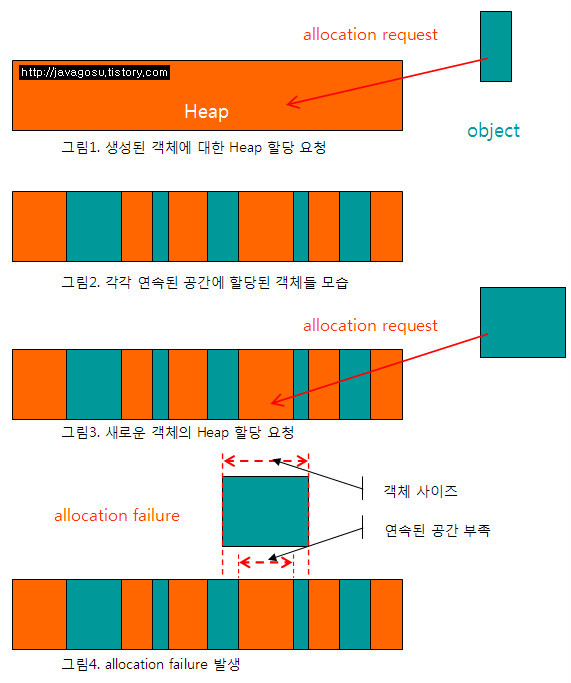
해결: 실제로 사용할 메모리 앞과 뒤에 경계표시를 만들어놓으면 할당된 메모리를 넘어갔을때를 인지할 수 있다.

#### 효율을 향상시키기 위해

기본 버전의 new는 일반적인 모든 상황에 만족해야 하므로 여러 가지 기능들이 들어있다. 하지만 사용 목적에만 맞도록 직접 수정한다면 불필요한 작업들을 많이 줄일 수 있다.

###### 힙 단편화(Heap Fragmentation)

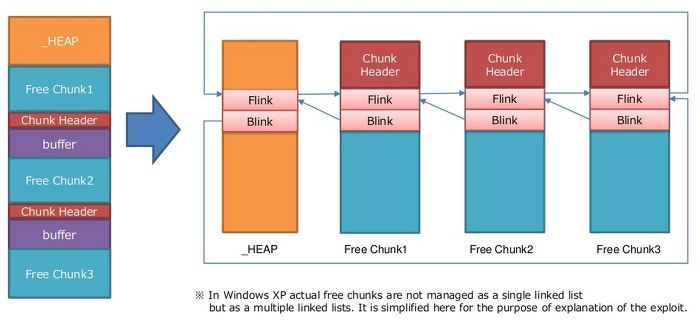
<http://ozt88.tistory.com/30>



메모리를 사용하다 보면 할당, 해제 순서는 뒤죽 박죽이다. 그렇기 때문의 위 그림처럼 쓸 수 있는 공간이 파편화되는 현상이 발생한다. 이것을 Heap Fragmentation이라고 한다. 이러한 현상으로 전체 메모리는 여분이 많지만 중간에 들어갈 메모리 공간이 없어 비효율적인 메모리 관리가 일어난다.

###### Heap Chunk

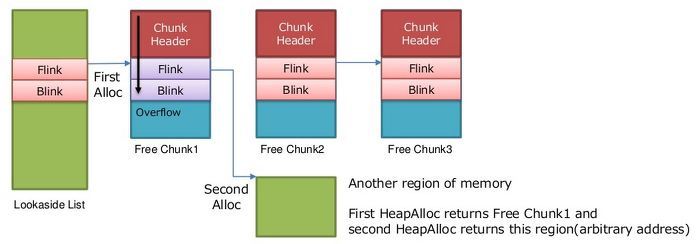
**연속된 메모리의 한 단위를 chunk**라고 부르자. chunk가 사용자에 의해 요청된 후 반환되면, free 상태의 메모리가 된다. **반환된 메모리들은 기존의 데이터를 이전에 반환된 메모리의 주소와 앞으로 연결된 주소를 저장할 공간으로 사용**한다. 운영체제는 반환된 free chunk (bin)에 대한 Linked List를 구성하여 빈 영역에 대해서 관리할 수 있게 장치를 마련해 두었다.

http://www.slideshare.net/ffri/mr201312-history-and-current-state-of-heap-exploit-eng

새로 메모리 할당 요청이 들어오면 OS는 우선적으로 free list를 순회하면서 적절한 메모리 영역을 찾는다. **free list의 chunk들을 이용가능한 사이즈로 sorting**한다면 이 순회작업이 훨씬 효과적일 것이다. **이렇게 최대한 적합한 빈 chunk를 찾은뒤 유저가 원하는 사이즈로 분할하여 반환**하는 방식으로 동작한다. 이 방식의 단점은 적은 단위의 메모리를 매우 빈번하게 할당 / 해제하는 경우에 발생한다. 처음 할당한 메모리의 크기를 가지고 free chunk를 만들었고, 요청에 따라 또다시 chunk를 분할하여 사용한다. 그러면 매우 많고 작은 chunk들이 발생할 것이고, 이것을 원활하게 사용하는 것은 힘든 일이 될 것이다.

###### Low Fragmentation Heap

최신 Windows 운영체제는 위의 Heap Chunk 방식에서 좀더 발전한 Heap 관리 정책을 적용시켰다. 이름부터 패기 넘치는 Low Fragmentation Heap (이하 LFH), 단편화 현상이 적은 힙이다. **아이디어는 미리 힙의 Chunk를 정해진 크기로 나눠서 해당 Chunk영역를 할당/해제 하는 방식으로 직접 메모리를 관리하는 방식**이다. 앞에서 이야기한 free list 방식은 작은 chunk들을 계속해서 분할하여 할당하는 방식을 진행하여 관리하기 어려운 문제가 있었다. **하지만 직접 정해진 사이즈별로 Chunk를 미리 만들어 두고 필요한 메모리에 적합한 최소한의 세그먼트를 할당하는 방식으로 지나친 분할과 관리의 어려움을 극복했다.** 이렇게 미리 정해진 **사이즈로 나뉘어진 Chunk를 Bucket**이라고 부른다. 이런 Bucket 형식이 유용한 부분은 작은 사이즈의 Chunk를 사용하는 경우이다. 이 때 **사용하는 관리자를 front-end heap매니저**라고 한다. 그리고 큰 사이즈의 Chunk들은 기존의 free list방식을 유지할 수 있게 back-end heap manager에게 일을 맡긴다.



http://www.slideshare.net/ffri/mr201312-history-and-current-state-of-heap-exploit-eng

그리고 LFH는 미리 Bucket으로 분할하여 관리하는 것으로 적합한 메모리영역을 찾는 속도를 빠르게 가져간다. 크기별 Free List들을 분할하여 Look-Aside List에 나눠서 관리하는 것이다. 말하자면 Free List에 대한 List를 만드는 것이다. 정해진 사이즈의 Free List들을 관리하는 각각의 BlocksIndex에 정보를 담아서 이 Free List에 원하는 사이즈가 있는지 체크한 뒤에 넘어가는 방식으로 접근 성능을 높였다.

//원하는 사이즈가 현재 인덱스의 최대 크기보다 큰지 체크한다.

while(BlockSize >= BlocksIndex -> ArraySize )

{

//다음번 인덱스(더 큰 단위의 free list)를 찾는다.

if(BlocksIndex -> ExtendedLookup == NULL)

{

//없으면 back-end heap manager에게 패스

}

//있다면 다음번 인덱스로 넘어가서 체크

BlocksIndex = BlocksIndex -> ExtendedLookup;

}

//그리고 해당 Index에 알맞은 free list에서 원하는 Chunk를 찾아서 할당

**더하여 high-end windows의 LFH는 힙의 멀티쓰레딩이슈까지 처리했다고 한다.** 힙은 private힙을 별도로 만들어 주지 않는 이상, 프로세스의 쓰레드들이 공유한다. 그래서 힙 영역을 다루는 API는 힙 할당이 경쟁상태가 되는 경우를 막기위해 atomic한 연산으로 동작한다. 쓰레드들은 힙을 다루는데 비싼 Lock연산을 통해 동작할 수 밖에 없는 문제가 있었다.

LFH는 affinity한 관리자를 제공한다. 자주 사용되는 bucket들을 내부적인 counter를 통해 접근 수준을 파악하고, 빈번한 bucket에 대해서 해당 bucket의 개수를 두배로 늘린다. 그리고 경쟁상태에 있는 bucket이 발생하면(아마도 빈번한 접근 때문에 개수가 많아졌을것), 대기중인 쓰레드를 다른 bucket으로 접근할 수 있게 해준다.

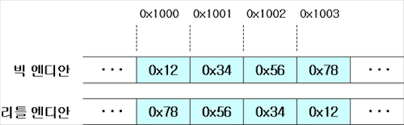
다양한 힙 관리 라이브러리들이 힙의 성능을 최적화하기 위해 이미 노력하였다. 그 중 대표적인 tcmalloc과 jemalloc이 있다. 이렇게 성능면에서 향상을 꾀하자면 사용자 정의 new가 효율적일 수 있다.

#### 적당히 타협한 기본 할당자의 바이트 정렬 동작을 보장하기 위해

바이트 정렬이 무엇이고 왜 하는 것일까?

###### 바이트 정렬

*바이트 정렬(byte ordering):* 메모리에 데이터를 저장할 때의 바이트 순서를 나타내는 용어이다.

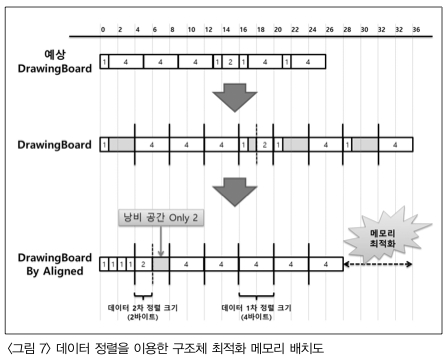


*빅 엔디안(big-endian):* 최상위 바이트(MSB, Most significant Byte)부터 차례로 저장되는 방식

*리틀 엔디안(little-endian):* 최하위 바이트(LSB, Least Significant Byte)부터 차례대로 저장되는 방식

*바이트 정렬을 하는 이유*

int n; 이라는 변수가 있다.   
이변수의 주소는 0x1000 ~ 0x1004 이다. 왜냐 4byte 변수니까 4byte 를 차지하고 있겠지.  
이러면 cpu 에서 메모리 0x1000 ~ 0x1004 의 조소를 한번에 받아 처리 할수가 있다.   
CPU 는 32bit 씩 처리 되니까.  
이러면 아무런 문제가 없다. 그런데 만일 int n;. 의 변수가 0x1001 ~ 0x1005 의 주소에   
존재 한다고 생각해 보라.   
그러면 CPU 는 메모리의 0x1000 ~ 0x1004, 0x1005 ~ 0x1008 의 주소를 받아서 처리해야   
int n; 을 처리할 수 있는것이다.   
  
그러면 쓰잘데기 없이 두번의 일을 하는 결과가 됐다. 그래서 바이트를 정렬해서 효율을 높이자는 것이다.



위와 같이 정렬하여 메모리를 효율적으로 사용하는 것이다.

#### 동적 할당 메모리의 실제 사용에 관한 통계 정보를 수집하기 위해

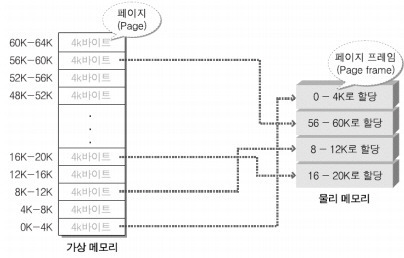
내 프로그램이 동적 메모리를 어떻게 사용하고 있는지를 감시할 수 있다.

#### 임의의 관계를 맺고 있는 객체들을 한 군데에 나란히 모아 놓기 위해

동시에 쓰이는 자료구조를 한 곳으로 모아두어 페이지 부재 발생 횟수를 최소화하고 싶을 경우 유용하다. 이는 위치 지정(placement) new및 위치지정 delete로 구현할 수 있다.

###### 페이징 기법(paging)

**페이징 기법**(paging)은 컴퓨터가 [메인 메모리](https://ko.wikipedia.org/wiki/%EA%B8%B0%EC%96%B5_%EC%9E%A5%EC%B9%98)에서 사용하기 위해 [2차 기억 장치](https://ko.wikipedia.org/wiki/%EA%B8%B0%EC%96%B5_%EC%9E%A5%EC%B9%98)로부터 데이터를 저장하고 검색하는 [메모리 관리](https://ko.wikipedia.org/wiki/%EB%A9%94%EB%AA%A8%EB%A6%AC_%EA%B4%80%EB%A6%AC) 기법이다.즉, [가상기억장치](https://ko.wikipedia.org/wiki/%EA%B0%80%EC%83%81%EA%B8%B0%EC%96%B5%EC%9E%A5%EC%B9%98)를 모두 같은 크기의 [블록](https://ko.wikipedia.org/wiki/%EB%B8%94%EB%A1%9D_(%EC%BB%B4%ED%93%A8%ED%8C%85))으로 편성하여 운용하는 기법이다. 이때의 일정한 크기를 가진 블록을 **페이지**(page)라고 한다. 주소공간을 페이지 단위로 나누고 실제기억공간은 페이지 크기와 같은 [프레임](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A0%88%EC%9E%84_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99))으로 나누어 사용한다.



이렇게 2차 저장 기억 장치를 메인 메모리에서 사용하는 것처럼 하는 기법이다.

#### 그때그때 원하는 동작을 수행하도록 하기 위해

적당한 동작을 각 수행 목적에 맞춰 바꾸고 싶다면 사용자 정의 버전을 만들면 된다.

***개발자가 스스로 사용자 정의 new및 delete를 작성하는 데는 여러 가지 나름대로 타당한 이유가 있습니다. 여기에는 수행 성능을 향상시키려는 목적, 힙 사용 시의 에러를 디버깅하려는 목적, 힙 사용 정보를 수집하려는 목적 등이 포함됩니다.***

[항목51] new, delete작성 방법

new및 delete를 작성할 때 따라야 할 기존의 관례를 잘 알아두자.

# operator new 작성 방법

기존의 관례를 지키면서 기능을 구현한다.

#### 구현

void\* operator new(std::size\_t size) throw(std::bad\_alloc)

{

//size가 0일 경우

if (size == 0) size = 1;

//계속해서 메모리 할당을 시도합니다.

while (true)

{

void\* alloc = malloc(size);

//성공 시, 할당된 포인터를 반환

if (alloc) return alloc;

//실패 시,

else

{

//new처리자를 설정

std::new\_handler globalHandler = std::set\_new\_handler(0);

std::set\_new\_handler(globalHandler);

//new처리자가 유효하면 수행

if (globalHandler) (\*globalHandler)();

//무효하면 std::bad\_alloc반환

else throw std::bad\_alloc();

}

}

}

여기에 몇 가지 고려해야 할 점이 있습니다.  
1. 크기가 0인 할당을 요청하는 경우: 크기를 1로 바꾸어준다. 0을 할당하면 문제가 발생한다.  
2. 할당에 성공할 시, 할당한 메모리의 주소를 반환한다.   
3. 할당에 실패할 시, new처리자 함수를 설정한다. 기존의 new처리자를 globalHandler에 담아두는데, 유효한지 확인하기 위해서다.  
4. new처리자 함수가 유효하다면 수행한다. 유효하지 않다면 std::bad\_alloc를 던진다.  
5. 2-5의 과정은 반드시 무한 루프에서 이루어져야 한다. 즉, new처리자가 유효하다면 계속해서 메모리할당을 시도할 수 있어야 한다.  
6. 반드시 무한 루프에서 빠져나갈 수 있는 수단이 있어야 한다. 위에서는 std::bad\_alloc를 던진다.

#### 세부사항

###### operator new가 기본 클래스의 멤버 함수를 불러온다.

멤버 함수로 new를 구현한 모습도 보인다.

class Archer

{

public:

explicit Archer() { std::cout << "constructor Archer.\n"; }

static void\* operator new(std::size\_t) throw(std::bad\_alloc);

private:

int power;

};

class BladeMaster : public Archer

{

public:

BladeMaster() { std::cout << "constructor BladeMaster.\n"; }

private:

int blade;

};

void\* Archer::operator new(std::size\_t size) throw(std::bad\_alloc)

{

std::cout << "create Archer.\n";

//size가 0일 경우

if (size == 0) size = 1;

//계속해서 메모리 할당을 시도합니다.

while (true)

{

void\* alloc = malloc(size);

//성공 시, 할당된 포인터를 반환

if (alloc) return alloc;

//실패 시,

else

{

//new처리자를 설정

std::new\_handler globalHandler = std::set\_new\_handler(0);

std::set\_new\_handler(globalHandler);

//new처리자가 유효하면 수행

if (globalHandler) (\*globalHandler)();

//무효하면 std::bad\_alloc반환

else throw std::bad\_alloc();

}

}

}

int main()

{

Archer\* a1 = new Archer(); //create Archer.

BladeMaster\* bm1 = new BladeMaster(); //create Archer?

delete a1;

delete bm1;

return 0;

}

이렇게 되면 기본 클래스와 파생 클래스의 크기가 다르므로 문제가 될 수 있다. 그러므로 다음과 같이 처리할 수 있다. 크기를 검사하고 그 크기에 맞는 함수를 다시 호출한다.

if (size != sizeof(Archer)) return ::operator new(size); //표준 new에게 할당을 넘긴다.

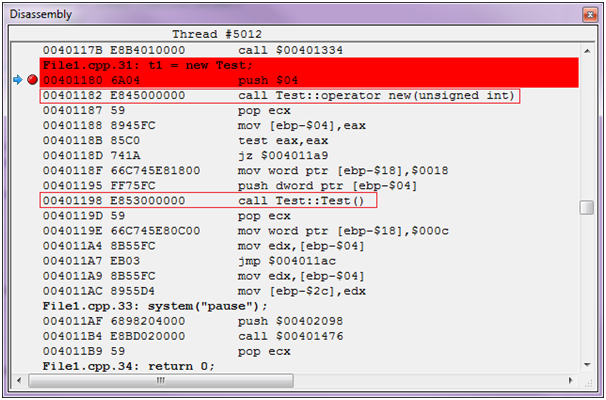
###### operator new[] 구현

배열의 할당은 원시 덩어리를 할당하는 것이 전부다. 다른 설정은 아무것도 할 수 없다.  
왜냐하면 상속 때문에 얼마나 큰지 알 수 없으며, 동적으로 할당된 배열은 그의 개수를 담는 크기가 추가로 들어갈 수 있으므로 정확한 크기와 다를 수 있기 때문이다.

###### operator new의 진실

위에서 사용자 정의 new를 만들어보았다. 하지만 우리는 생성자를 호출한 적이 없지만 호출된다. 무슨 일이 벌어진 거지?

<https://lunapiece.net/Article/3900>



우리는 new 를 호출했을 뿐인데 실제로는 operator new 라는 함수와 생성자를 따로따로 호출 하고 있었네요. **그럼 우리가 수정한 것은 operator new라는 함수 일 뿐 new 연산자 자체가 아니라는 것**을 알 수 있습니다.

심지어 윗줄에서는 할당 할 메모리 크기를 스택에 집어넣어주는 친절함(?) 까지 엿볼 수 있군요. Pop 이 호출부위 아래에 존재한다는 것은 operator new 는cdecl 방식으로 call 되고 있다고 짐작 할 수 있겠습니다.

**그럼 new 는 단지 operator new 와 생성자를 호출 하고 있을 뿐이라면 이 두 함수를 따로 따로 호출하는것도 가능하지 않을까요?**

|  |
| --- |
| class Test;  int main()  {             Test\* t1;               t1 = (Test\*)Test::operator new(sizeof(Test));             t1:Test();               system("pause");             return 0;  } |

그리고 실행 해 보았습니다. 결과는 한번 더 생성자가 호출된다.

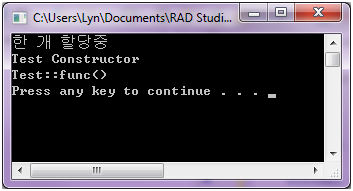
그럼 이렇게 결론 내릴 수 있을까요? **new 연산자는 operator new 함수를 호출 한 후, 생성자를 호출 해 주는 연산자이다.**

그런데 이게 또 아닌거 같습니다. 왜냐면 이 두 함수에는 VMT(Virtual Methor Table) 을 생성 해 주는 부분이 없거든요. 가상함수가 한 개 이상 존재하고 상속관계가 있는 클래스(상속 받았던 상속 했던) 에는 반드시 VMT의 포인터가 존재합니다. 그럼 이 VMT는 언제 등록되었을까요? 살짝 코드를 고쳐보았습니다.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    class Test  {             virtual void func()             {                       printf("Test::func()\n");             }  };    class Test2 : public Test  {             virtual void func()             {                       printf("Test2::func()\n");             }  };  int main()  {             Test\* t1;               t1 = new Test2;               system("pause");             return 0;  } |

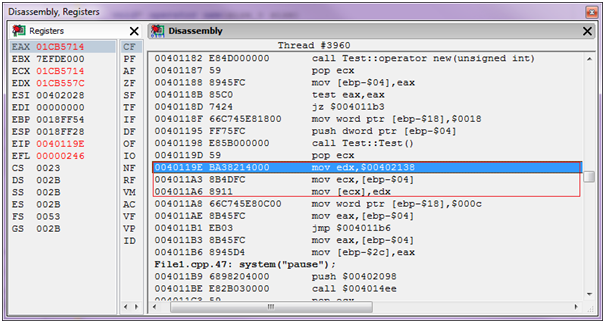
생성자에서 func라는 가상함수를 호출 하고 있습니다.

우리는 Test2를 생성하였으므로 Test2::func() 가 출력 될 거라고 기대 할 수 있겠습니다. 결과를 볼까요?



어라 뭔가 이상합니다. Test::func() 가 출력되었네요. 이게 어찌 된 일일까요?

그 이유는 생성자를 호출 하는 시점에서는 VMT의 포인터가 제대로 등록 되지 않기 때문에, 가상함수 본래의 역할을 제대로 하지 못하는 겁니다. 디스어셈블 한 코드를 보겠습니다.



생성자를 호출 한 다음에도 추가적인 작업을 하고 있습니다.

그 중 빨간 네모로 표시 된 부분이 가상함수가 없을때는 존재하지 않던 부분입니다.

즉 저 부분에서 VMT포인터를 추가하는 작업을 하고 있다고 생각 할 수 있겠습니다.

**이제 new 의 진실이 밝혀졌군요.**

new 연산자를 호출하면

1.     메모리 할당을 위해 operator new 함수를 호출한다

2.     생성자를 호출한다

3.     VMT를 등록해 준다

# operator delete 작성 방법

new보다 훨씬 더 간단하다.

#### 구현

void operator delete(void\* raw)

{

std::cout << "delete Archer.\n";

if (!raw) return;

free(raw);

}

#### 세부사항

멤버 함수에서의 delete + 기본 클래스의 호출 문제 고려

class Archer

{

public:

explicit Archer() { std::cout << "constructor Archer.\n"; }

static void\* operator new(std::size\_t) throw(std::bad\_alloc);

static void operator delete(void\*, std::size\_t) noexcept;

private:

int power;

};

class BladeMaster : public Archer

{

public:

BladeMaster() { std::cout << "constructor BladeMaster.\n"; }

private:

int blade;

};

void Archer::operator delete(void\* raw, std::size\_t size) noexcept

{

if (size != sizeof(Archer))

{

::operator delete(raw); //표준 delete에게 삭제를 맡긴다.

return;

}

std::cout << "delete Archer.\n";

if (!raw) return;

free(raw);

}

***관레적으로, operator new함수는 메모리 할당을 반복해서 시도하는 무한 루프를 가져야 하고, 메모리 할당 요구를 만족시킬 수 없을 때 new처리자를 호출해야 하며, 0바이트에 대한 대책도 있어야 합니다. 클래스 전용 버전은 자신이 할당하기로 예정된 크기보다 더 큰(다른) 메모리 블록에 대한 요구도 처리해야 합니다.***

***operator delete함수는 널 포인터가 들어왔을 때 아무 일도 하지 않아야 합니다. 클래스 전용 버전의 경우에는 예정 크기보다 더 큰 블록을 처리해야 합니다.***

[항목52] 위치지정 new, delete 사용

위치지정 new를 작성한다면 위치지정 delete도 같이 준비하자.

# new연산자의 실패

위에서 언급되었지만 new연산자는 3가지 버전이 있다고 했으며 다음과 같다.

//throw(1)

void\* operator new (std::size\_t size);

//nothrow(2)

void\* operator new (std::size\_t size, const std::nothrow\_t& nothrow\_value) noexcept;

//placement(3)

void\* operator new (std::size\_t size, void\* ptr) noexcept;

new연산자는 2가지 함수를 호출한다. 하나는 메모리 할당이며 다른 하나는 생성자이다.

#### new연산에 실패할 경우

각각의 함수 호출에 대해 실패할 수 있다.

###### 메모리 할당에 실패할 경우

할당한 메모리 여유가 없어 생길 수 있다.

int main()

{

std::vector<Archer\*> va;

int i = 0;

try

{

while(i++ < 1000)

{

va.push\_back(new Archer());

}

}

catch (std::bad\_alloc& e)

{

std::cerr << e.what() << '\n';

if (va[i]) std::cout << "isDone.\n";

}

return 0;

}

위의 코드에서는 518개째의 객체에서 메모리 할당에 실패하여 std::bad\_alloc를 던진다. 그럼 마지막에 시도한 메모리 할당은 어떻게 할까? 당연히 되돌려야 마땅하다. 이는 비어있는 할당자가 null포인터를 반환함으로써 해결할 수 있다.

**결국, new연산이 메모리 할당에 실패하면 할당자가 null을 반환하여 메모리 할당을 되돌릴 수 있다.**

###### 생성자에 실패한 경우

생성자에서 예외가 발생할만한 코드를 만들면 안 되겠지만, 발생할 수는 있다.

int main()

{

Archer\* a = nullptr;

try

{

a = new Archer();

}

catch (std::exception& e)

{

std::cerr << e.what() << '\n';

if (a)

{

std::cout << "isDone.\n";

delete a;

}

}

return 0;

}

위에서는 생성자에서 예외가 발생한다. 그럼 메모리 할당은 어떻게 될까? 이는 C++런타임 시스템이 알아서 delete를 호출해주기 때문에 남아있는 메모리는 없다.

###### Question. 여기서 만약에 delete를 호출하지 못하면 어떻게 될까?

포인터의 반환은 new연산이 끝난 후, 발생한다. 즉, 메모리 할당 -> 생성자 호출 -> 포인터 반환식이다. 여기서 생성자가 예외를 방출하면 반환할 포인터는 더 이상 유효하지 않다. 포인터 a로는 메모리를 복구할 수 없다. 즉, 이미 할당한 메모리는 사용자 입장에서 메모리를 관리할 수 없으며, 런타임 시스템 영역이다. **그러므로 런타임 시스템이 delete를 호출하지 못한다면 사용자 입장에서 복구할 방법은 없다.**

**결국, new연산이 메모리 할당에 실패할 시, C++런타임 시스템은 호출한 new에 맞는 delete를 호출하여 메모리를 되돌린다.**

# new연산자의 실패가 문제가 되는 경우

위에서 보면 new연산의 실패는 어떠한 경우에도 방지되어있는 것처럼 보인다. 하지만 그렇지 않다. 어떠한 경우에 안전하지 않은가?

#### 위치지정 new의 사용

메모리 할당의 경우는 크게 문제가 안 된다. 문제는 생성자에서의 예외이다. **런타임 시스템이 수행한 new에 맞는 delete를 찾을 수 있어야 하는데,** 기본형은 문제가 되지 않지만 그렇지 않으면(위치지정 new일 경우) 문제가 스물스물 스며들게 된다.

###### 위치지정 new란?

이렇게 **추가로 매개변수를 받는 new를 위치지정 new**라고 한다. 기본으로 제공되는 위치지정 new는 위의 2가지 버전(nothrow객체나 포인터를 받는)이 있다. 그 외 사용자가 직접 작성하는 new는 대부분 이에 속할 것이다.

###### new에 짝이 맞는 delete를 찾을 수 없을 때

위치지정new를 사용할 때, 생길 수 있는 문제를 구체적으로 살펴보자.

class SpecialPunch{};

class Archer

{

public:

Archer()

{

throw std::exception();

}

static void\* operator new(std::size\_t size, SpecialPunch sp)

{

std::cout << "special new.\n";

return malloc(size);

}

private:

int data1[1'000'000];

};

int main()

{

Archer\* a = nullptr;

try

{

SpecialPunch sp;

a = new(sp) Archer();

}

catch (std::exception& e)

{

std::cerr << e.what() << '\n';

if (a)

{

std::cout << "isDone.\n";

delete a;

}

}

return 0;

}

위에 코드에서는 특별한 new버전을 호출하게 되어있다. 그리고 여전히 예외는 발생한다. 이러한 상황에서 짝이 맞는 delete를 호출해야 하는데 없다면 호출을 안할까? 컴파일러는 다음과 같은 섬뜩한 경고를 한다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 심각도 | 코드 | 설명 | 프로젝트 | 파일 | 줄 | Suppression State |
| 경고 | C4291 | 'void \*Archer::operator new(::size\_t,SpecialPunch)': 일치하는 operator delete가 없습니다. 초기화할 때 예외가 Throw되지 않으면 메모리가 확보되지 않습니다. | ec++ | e:\ytk\algorithm\_solution\_2013\effective\_c\effective\_c\item52.cpp | 44 |  |

위에서 Question의 고찰로 인하여 런타임 시스템이 delete를 호출하지 못하면 사용자가 직접 메모리를 복구할 방법은 없다. 말 그대로 메모리가 스물스물 세어나갈 것이다.

**결국 위치지정 new는 그에 맞는 delete를 찾아야 하므로 이에 대한 구현도 반드시 이루어져야 예외로부터 안전하다.**

#### 문제 해결: 위치지정 new 사용

간단하다. 매개변수가 일치하는 delete를 직접 정의하면 된다.

class Archer

{

public:

Archer()

{

throw std::exception();

}

static void\* operator new(std::size\_t size, SpecialPunch sp)

{

std::cout << "special new.\n";

return malloc(size);

}

static void operator delete(void\* raw, SpecialPunch sp)

{

std::cout << "special delete.\n";

free(raw);

}

private:

int data1[1'000'000];

};

# new연산자의 가림

위에서 위치지정new를 직접 작성하였다. 이런 new가 하나라도 존재하면 표준 new는 가려지게 된다.

if (a)

{

std::cout << "wow.\n";

delete a; //error: 기본 delete는 사용 못함

}

**지금 우리가 원하는 것은 호출한 new에 대하여 매개변수에 맞는 사용자 정의new나 표준 new를 컴파일러가 알아서 링크하는 것**이다.

class BasicNewDelete

{

public:

static void\* operator new(std::size\_t size)

{

return ::operator new(size);

}

static void operator delete(void\* raw)

{

::operator delete(raw);

}

};

class SpecialPunch{};

class Archer : public BasicNewDelete

{

public:

using BasicNewDelete::operator new;

using BasicNewDelete::operator delete;

// 이하 동일

};

표준 new, delete를 같이 정의함으로써 적절하게 오버로딩할 수 있게 되었다.

***operator new함수의 위치지정 버전을 만들 때는, 이 함수와 짝을 이루는 위치지정 버전의 operator delete함수도 꼭 만들어 주세요. 이 일을 빼먹으면, 찾아내기도 힘드며 또는 생겼다 안 생겼다 하는 메모리 누출 현상을 경험하게 됩니다.***

***new및delete의 위치지정 버전을 선언할 때는, 의도한 바도 아닌데 이들의 표준 버전이 가려지는 일이 생기지 않도록 주의하세요.***