

초보자를 위한 C++ 강의

씹어먹는 C++

이재범

v.2020.06.17

Copyright © 2019-2020 이재범

이 책은 모두의 코드에 연재된 씹어먹는 C++ 강좌를 책으로 옮긴 것입니다. 해당 강좌는 <https://mODOOCODE.COM>에서 볼 수 있습니다.

차례

차례	3
제 1 장 C++ 을 시작하기에 앞서	1
제 2 장 C++ 시작하기	6
첫 C++ 프로그램 분석하기	15
C++ 와 C 언어의 공통 문법 구조	22
제 3 장 C++ 의 참조자 (레퍼런스)	30
제 4 장 C++ 의 세계로	46
제 5 장 객체지향프로그래밍의 시작	57
생성자와 함수의 오버로딩	69
복사 생성자와 소멸자	84
const, const, const!	109
내가 만들어보는 문자열 클래스	138
클래스의 explicit 과 mutable 키워드	179
제 6 장 연산자 오버로딩(overloading)	188
잡다한 연산자들의 오버로딩	219
N 차원 배열 만들기 프로젝트	240
제 7 장 클래스의 상속	270
가상 함수와 다형성	298
상속에 관련된 잡다한 내용들	322
제 8 장 C++ 표준 입출력 라이브러리	340
C++ 파일 입출력	356

제 9 장 엑셀 만들기 프로젝트	374
엑셀 만들기 프로젝트 2부	394
제 10 장 C++ 템플릿	410
가변 길이 템플릿	445
템플릿 메타프로그래밍 1 부	459
템플릿 메타프로그래밍 2 부	473
제 11 장 C++ 표준 라이브러리 (컨테이너와 알고리즘)	494
C++ 표준 컨테이너	494
C++ 의 표준 연관 컨테이너들	522
C++ 표준 알고리즘 라이브러리	556
C++ 문자열의 모든 것 (string 과 string_view)	591
제 12 장 C++ 에서의 예외 처리	613
제 13 장 우측값과 이동 연산	630
우측값 래퍼런스와 이동 생성자	630
move 문법과 완벽한 전달	649
제 14 장 스마트 포인터	671
여러 객체가 소유할 수 있는 포인터	688
제 15 장 함수 객체	705
제 16 장 C++ 쓰레드	719
뮤텍스와 조건변수	739
atomic 객체와 명령어 재배치	766
비동기 연산을 위한 도구들	791
ThreadPool 만들기	807
제 17 장 C++ 유니폼 초기화	826
제 18 장 컴파일 타임 상수 constexpr	837
제 19 장 decltype 와 std::decay	852
제 20 장 다양한 C++ 표준 라이브러리 소개	866
type_traits 라이브러리, SFINAE, enable_if	866
정규 표현식(<regex>) 라이브러리 소개	894
난수 생성(<random>)과 시간 관련 라이브러리(<chrono>) 소개	909
C++ 파일 시스템(<filesystem>) 라이브러리 소개	921
C++ 유틸리티 라이브러리 소개	938

제 21 장 강의를 마무리하면서	956
제 22 장 C++ 개발자가 알면 좋을 것들	959
Make 사용 가이드 (Makefile 만들기)	959

C++ 을 시작하기에 앞서

안녕하세요 여러분~ 이제 C 언어에 이어서 C++ 강좌를 연재하게 된 이재범입니다. 저의 C++ 강좌는 여러분이 C 언어를 충분히 이해하고 사용하고 있다는 것을 가정으로 진도를 나갈 것입니다.

물론 C 언어를 굳이 배우지 않은 상태에서 C++ 을 첫 언어로 배워도 상관 없습니다. C 언어와 C++ 은 아예 다른 언어기 때문에 무엇을 먼저 배워야 하냐 라는 의미는 크게 중요하지 않습니다. 하지만 C++ 이 C 의 기초적인 문법을 그대로 사용하고 있고, 제가 C 언어 강의를 먼저 완성하였기 때문에 독자 타겟을 C 언어 문법을 어느 정도 아는 사람 으로 잡았을 뿐입니다.

다시 말해 구질 구질하게 `for` 문 사용법, 포인터와 같은 것들은 C++ 강좌에서 다루지 않겠다는 의미입니다.

간혹 C++ 이 C 언어 확장팩(?) 개념이라고 생각하시는 분들이 있는데 이건 역시 사실이 아닙니다. 물론 초기에는 C++ 이름이 *C with classes* 였을 정도로 그냥 C 언어에 몇 가지 정도를 더 얹은 정도였습니다. 하지만 이제는 C++ 과 C 언어가 둘이 같은 언어야? 라고 말할 수 있을 정도로 매우 달라졌습니다.

왜 C++ 을 배우나

C++ 은 전세계에서 가장 사랑 받는 언어들 중 하나입니다. 그 사용 예시를 따지면 수도 없이 많은데 몇 가지 꼽아보자면

- 많은 게임들과 게임 엔진들 (Unity, Unreal 등등)
- 대부분의 컴파일러 (gcc, clang 등)
- 동영상 및 오디오 처리
- 운영체제 (대표적으로 마이크로소프트의 윈도우즈가 C++ 로 쓰여져 있습니다)
- 상용 프로그램들 (예를 들어 포토샵)

- 크롬 브라우저
- 딥러닝 프레임워크 (Tensorflow, PyTorch 모두 파이썬은 인터페이스 일 뿐 내부적으로 모두 C++ 로 만들어져 있습니다.)
- 서버 프로그램 (구글의 검색 서버도 C++ 로 만들어져 있습니다.)
- 그 외 금융쪽이나 많은 연산이 필요한 경우들...

등등 수도 없이 많습니다. 특히 위와 같이 프로그램의 성능이 중요한 부분에서 널리 쓰이고 있죠.

다만 많은 사람들이 하나같이 손꼽아서 이야기하는 C++ 의 큰 단점이 있는데 바로 쓰기 어렵다 입니다. 맞습니다. 정말로 C++ 을 제대로 공부하지 않는다면 쓸어지는 수 많은 컴파일 오류들과 복잡한 템플릿 문법, 그리고 우측값 좌측값이니 하는 값 카테고리 등등, 덕분에 많은 좌절감을 느끼실 것입니다.

하지만 제 생각으로는 이 문제들은 모두 C++ 을 제대로 공부하지 않아서 발생하는 문제라고 생각하고 굳이 언어를 배우지 않는 시점에서 너무 두려워 할 필요가 없음을 알려드리고 싶습니다. 특히 제 강좌를 통해서 C++ 이 생각했던 것 보다는 괜찮다 라는 것을 전해드리고 싶네요. 위 많은 프로그램들이 C++ 로 쓰여져 있는 것은 분명히 이유가 있을 테니까요!

강의 가이드라인

아무래도 이 강좌를 보러오시는 분들이 다 같은 출발 선상에 있는 것이 아닐 테니 어떤 강좌들을 먼저 보면 좋을지에 대해 아래와 같은 가이드라인을 제안해드립니다. 본인의 현재 상황에 맞게 적절한 강좌를 찾아 보세요.

- 나는 컴퓨터 프로그래밍을 한 번도 해본적이 없다 : 아무래도 이 강의는 적합하지 않은 것 같습니다. 일단 기본적으로 C 언어의 문법을 알고 있다고 가정하고 시작하거든요. 만일 C 언어를 먼저 배우고 싶다면 [이 강의](#) 을 먼저 보고 와주세요
- 얼마 전에 C 언어를 대충 공부했는데 C++ 을 배워보고 싶다 : 좋습니다! 처음부터 차차히 보시기를 바랍니다.
- C++ 문법을 대충 아는데 좀 더 심도 있게 공부하고 싶다: 이 분들도 환영입니다. 중간 중간 내용은 스kip해도 되고, 템플릿을 본격적으로 다루는 [9 - 1 강의](#) 부터 보시면 좋을 것이라 생각합니다.
- 프로그래밍 대회를 위해 C++ 자료 구조를 복습하고 싶다: C++ 의 여러가지 자료구조들을 (`vector`, `map`, `unordered_map` 등등) 빠르게 짚고 넘어가고 싶다면 [10 - 1, 2, 3](#) 강의를 보시는 것을 추천합니다.

- 최근 나온 C++ 기능들에 대해 알아보고 싶다 : [9 - 2 강의](#) 부터 거의 대부분 C++ 11 에 도입된 내용들입니다. 만일 C++ 17 에 새로 추가된 내용들을 공부하고 싶다면 [17 - 4, 5](#) 를 보시면 됩니다.

C++ 강의를 시작하기에 앞서



이 아저씨가 바로 C++ 의 아버지인 Bjarne Stroustrup 입니다

C++ 은 역사가 매우 긴 언어입니다. 1979년에 컴퓨터 과학자인 Bjarne Stroustrup 이 C 언어에 클래스라는 개념을 적립한 *C with Classes*라는 언어가 C++ 의 전신인데, 1982년에 해당 언어를 조금 발전 시켜서 C++ 이라는 이름을 붙였습니다. 그리고 1998년에 C++ 의 첫 번째 표준이 공개되었는데 이를 C++ 98 이라고 부릅니다. 2003년에 표준안에 작은 몇 가지 개정이 있었지만 언어 자체는 크게 바뀌지 않았습니다. 씹어먹는 C++ 강좌 초반에 9 장 까지 내용들이 바로 이 시절에 추가된 개념들입니다.

그 동안 언어가 정체 상태에 있다가 비로소 2011년이 되서야 흔히 말하는 대격변 패치를 받고 여러 가지 새로운 개념들이 추가되었습니다. 이 버전의 C++ 을 C++ 11 이라고 하는데, 우리 강좌 9

장부터 해당 내용들을 다루게 됩니다. 참고로 C++ 11 부터 **Modern C++** 이라고 부릅니다.¹⁾

그 후로 3 년마다 C++ 언어의 새로운 표준안이 공개되고 또 컴파일러들에게 반영이 되고 있습니다. C++ 14 와 C++ 17 에서는 비교적 작은 변화만 있었는데, C++ 20 에서 또다른 대규모의 업그레이드를 받을 예정입니다.

참고로 C++ 은 이전 버전과 호환성이 꽤나 괜찮은 언어 중에 하나입니다. 따라서 새로운 버전이 나오더라도 기존의 코드를 거의 대부분의 경우 무리 없이 컴파일 할 수 있습니다. 쉽게 말해 C++ 11 컴파일러로 C++ 98 코드를 무리 없이 컴파일 할 수 있다는 의미입니다.²⁾

강좌 수칙

제 강좌를 들으시면서 지켜야 할 수칙들은 다음과 같습니다.

1. 강좌는 적어도 한 번 꼭 정독해보기
2. 모르는 것은 꼭 답글 달기
3. 답글로 질문하기 꺼린 내용은 kev0960@gmail.com 으로 메일 보내기!
4. '생각 해보기'에 적어도 30 분 이상은 투자하기
5. 이전 강좌를 완벽히 이해했다 싶지 않으면 다음 강좌로 넘어가지 말기

입니다. 위 수칙 중에서 무엇보다도 중요한 것은 바로 모르는 내용은 꼭 답글을다는 것입니다 :) 세상에서 어리석은 질문은 없습니다. 여러분들이 궁금하거나 이상한 내용, 마음에 와닿지 않는 내용들을 질문하지 않고 다음강좌로 넘어가는 한 여러분의 실력을 결코 향상될 수 없을 것입니다.

꼭 궁금한 것들은 댓글로 남겨주시고, 공개적으로 남기기 꺼리면 비밀글로 남기거나 정 그렇다면 제 메일로 보내주시면 감사하겠습니다. 그러면 제가 확인하는대로 최대한 빨리 답글로 보내드릴께요 ㅎ

참고로 이 강좌는 Visual Studio 2017 커뮤니티 버전을 사용하고 있습니다. 여기를 클릭해서 사용법을 미리 숙지하는게 좋겠습니다.

그렇다면 이제 강좌를 보러갈 시간이 되었네요~

1) 재미있는 사실은 원래 이 개정안이 2010 년 안에 마무리 될 줄 알고 C++ 0x 라고 불리었습니다. 하지만 너무나 많은 변화가 필요하였기에 시간 내에 마무리 하지 못하고 C++ 11 이 되버렸습니다.

2) 물론 이전 버전 호환성이 언어 차원에서 과격한 변화를 가져오기 힘들다는 문제도 있습니다. C++ 과 반대되는 예시가 Python 을 들 수 있는데 Python2 에서 Python3 으로 가면서 언어 간의 호환성을 버리는 대신에 많은 개선을 할 수 있었죠.

C++ 강좌 총 목록

앞의 차례를 참고하시기 바랍니다.

부록

이 부분 부터는 C++ 언어 자체와는 직접 관련은 없지만 실제로 C++ 을 프로그래밍 하기 위해서 필요한 지식들과 여러가지 유용한 도구들에 대해서 이야기 하고자 합니다

C++ 시작하기

안녕하세요~ 여러분. C++ 의 세계에 오신 것을 진심으로 환영합니다. 사실 C 언어를 접해본 여러분들이 생각하기에 이름에 낚여서 C++ 은 C 의 단순한 확장판 정도라 생각하는 분들이 있는데 이는 결코 사실이 아닙니다.

스타크래프트로 따지면 오리지날과 브로드워 개념이 아니라 스타 1 과 스타 2 정도의 차이 일까요. C++ 은 말 그대로 C 언어의 문법만을 차용한 새로운 언어라고 보시면 됩니다. 왜냐하면 언어를 만들어낸 기본 개념이 다르기 때문이지요. 이 기본 개념이 어떻게 다른 지에 대해서는 나중에 설명하도록 하겠습니다.

자 그렇다면, 신나는 C++ 의 세계로 떠나볼까요~

준비물

이전처럼 C++ 을 배우기 위해서는 다음과 같은 준비물들이 필요합니다.

1. 인터넷이 되는 컴퓨터. 특히 [이 사이트](#)가 수월하게 들어가져야 함
2. 노
3. 개념
4. 씹어먹는 C 언어를 다 배움으로써 얻을 수 있는 지식
5. 컴파일러

여기서 가장 중요한 것은 4 번 씹어먹는 C 언어를 다 배움으로써 얻을 수 있는 지식인데, 왜냐하면 이 강좌는 오직 전적으로 여러분이 C 를 충분히 알고 있다는 전제 하에서 진행될 것이기 때문입니다.

물론 C++ 을 배우기 위해서 반드시 C 언어를 먼저 배워야 하는 것은 아닙니다. 하지만 C++ 이 C 의 기본적인 문법을 그대로 따르고 있고 제가 이미 C 강좌를 작성한 관계로 문법에 대한 설명(for

문, if 문 등등)은 대부분 생략할 것이기에 C 의 기초 문법들을 어느정도 아는 상태로 강의를 보시는 것을 추천합니다.

만일 프로그래밍이 처음이시거나 C 언어 문법을 잘 모르시는 분들의 경우 [어서 여기를 누르셔서](#) C 언어 부터 정복 하고 오시기 바랍니다. 그럼 이제 필요한 것은 5 번, 컴파일러가 되겠네요.

주의 사항

참고로 아래 설치하는 비주얼 스튜디오는 2010 년 버전이고 현재 마이크로소프트에서 2017 년 버전을 무료로 제공하고 있습니다. [여기](#) 를 눌러서 따라 설치하시면 됩니다.

사실 여러분들은 C++ 컴파일러를 이미 다 컴퓨터에 설치하고 계실 것입니다. 왜냐구요? 이전에 C 언어를 배울 때 설치하였던 Visual Studio 2008 에 C 및 C++ 컴파일러가 모두 다 들어있기 때문이지요. ¹⁾

하지만 제가 씹어먹는 C 언어 강좌를 썼을 때가 바야흐로 2 년 전인 2009 년으로 최신 버전이 2008 이였지만 이제는 2011 년으로 최신 버전이 2010 으로 올라갔습니다.

2008 하고 2010 의 버전 차이는 하늘과 땅 차인데, 무엇보다도 코딩 하기가 매우 편리해 져서 여러분들이게 설치하기를 강력 추천 합니다. (왠지 이 강좌를 끝낼 즘에 2012 버전이 곧 나올 듯한 안 좋은 느낌이 드네요ㅎ)

일단 [여기](#) 로 들어갑니다.

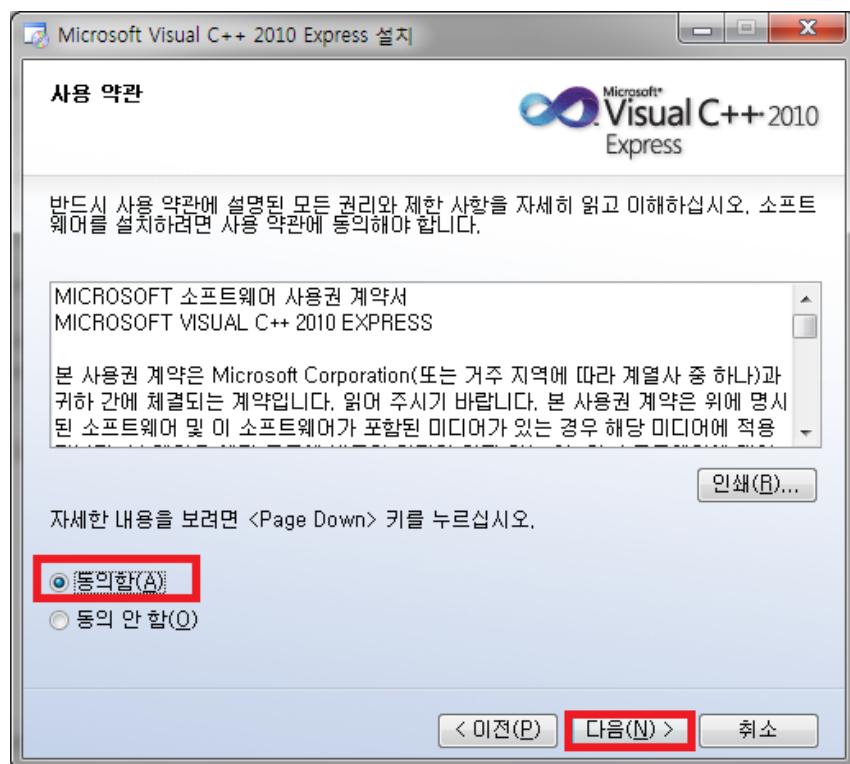


그리고 왼쪽에 DOWNLOAD 를 누르신 뒤 오른쪽에 언어를 Korean 으로 선택하시고 INSTALL NOW 를 누르시면 됩니다.

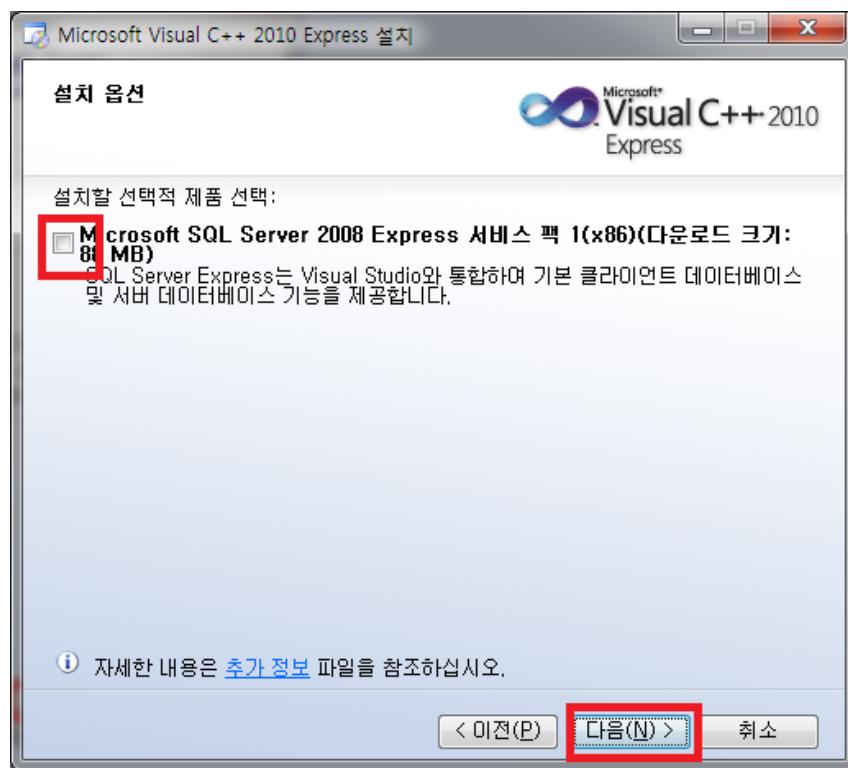
1) 이 글만 보아도 이 강좌의 세월을 느낄 수 있습니다. 제가 C++ 강의를 시작한 것이 2011 년입니다. 하지만 2019년 현재 아직도 지속적으로 업데이트 하고 있습니다.



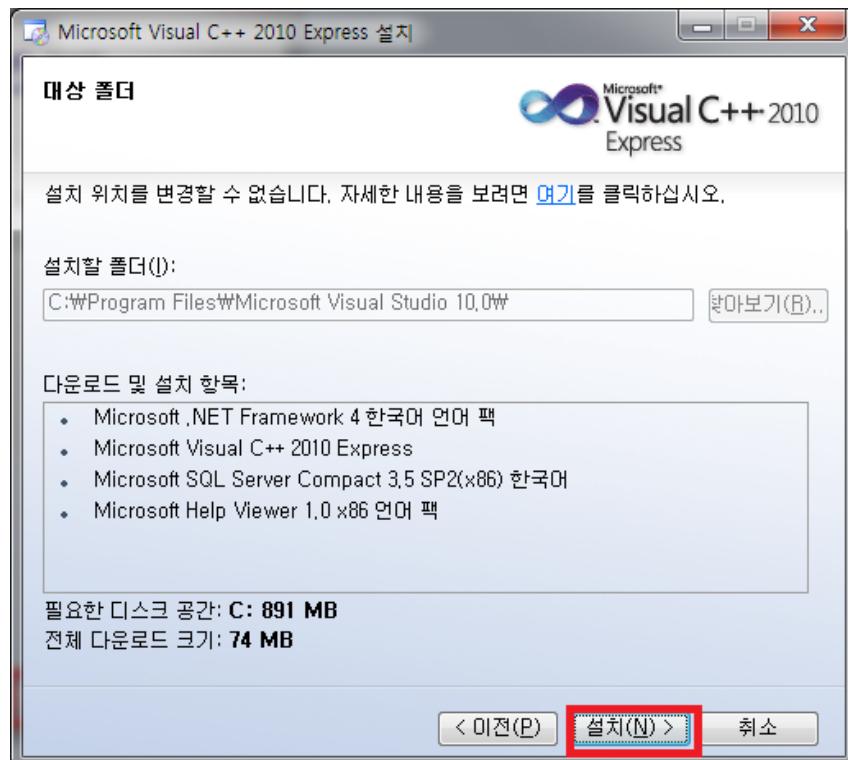
다음을 누르시고,



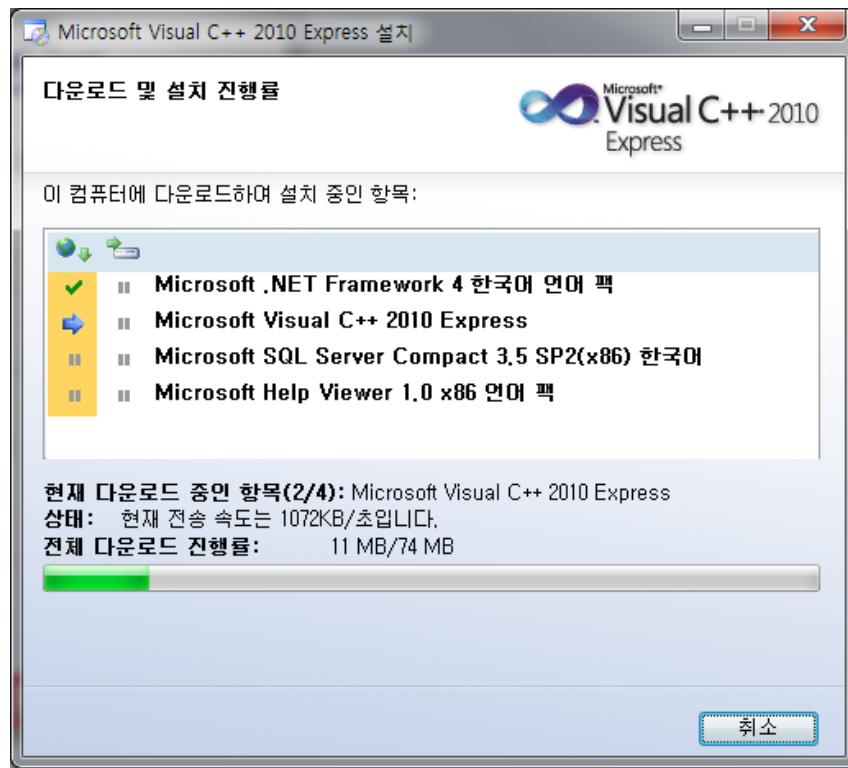
동의를 누른 뒤, 다시 다음을 누르고



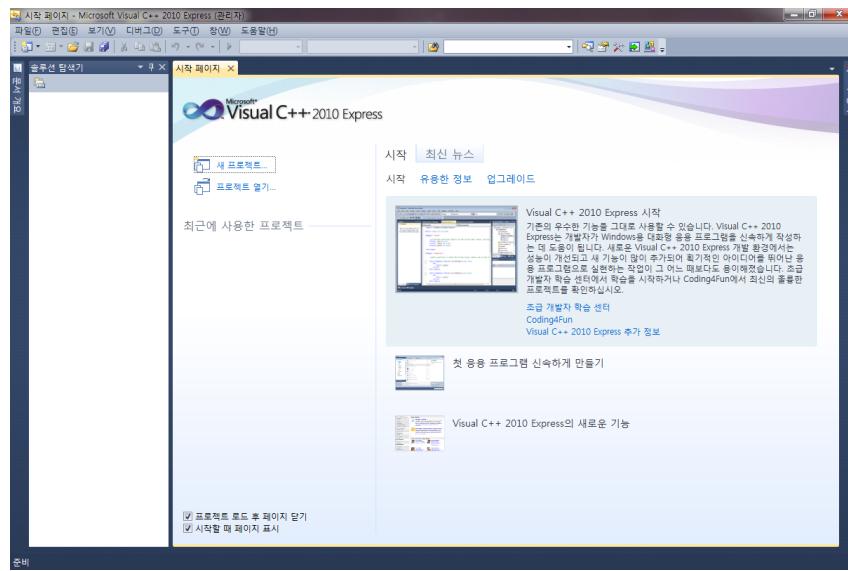
이것은 우리가 앞으로 배울 수준의 프로그래밍에서는 결코 필요한 것이 아니기 때문에 체크를 해제하고 다음을 누르고



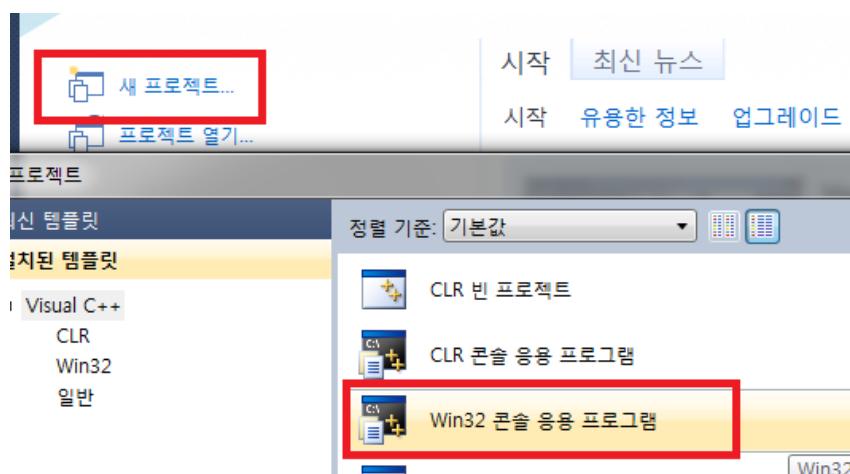
역시 다음..



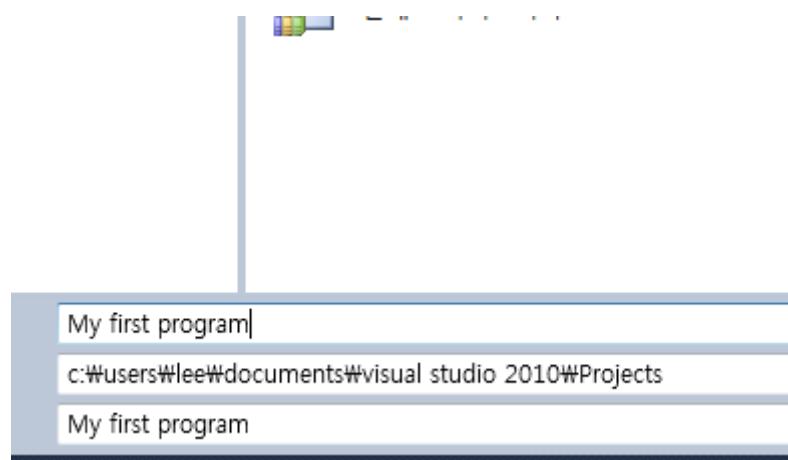
자 이제, 위와 같이 온라인 상으로 전체 프로그램을 다운받게 됩니다. 그리고 조금만 기다리다 보면 완료되었다는 표시가 납니다. 그렇다면 이제 실행해봅시다.



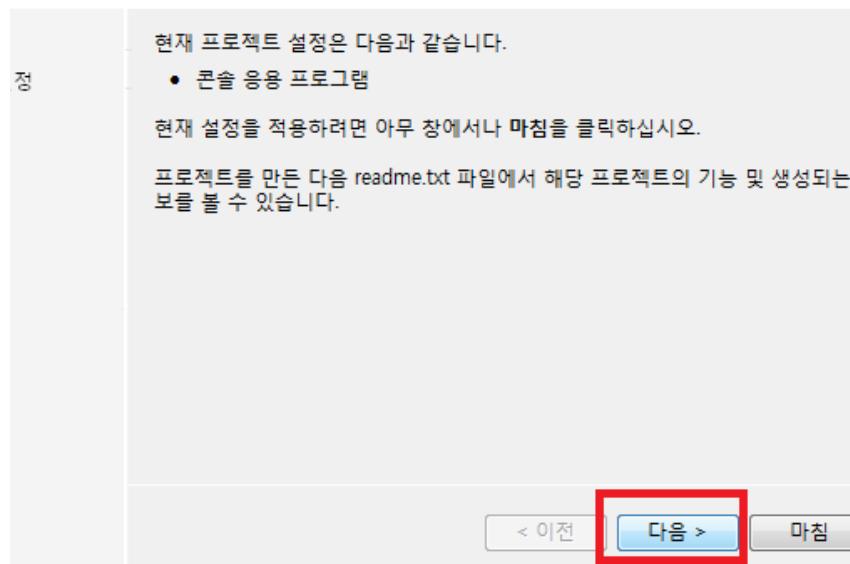
아주 멋있네요~



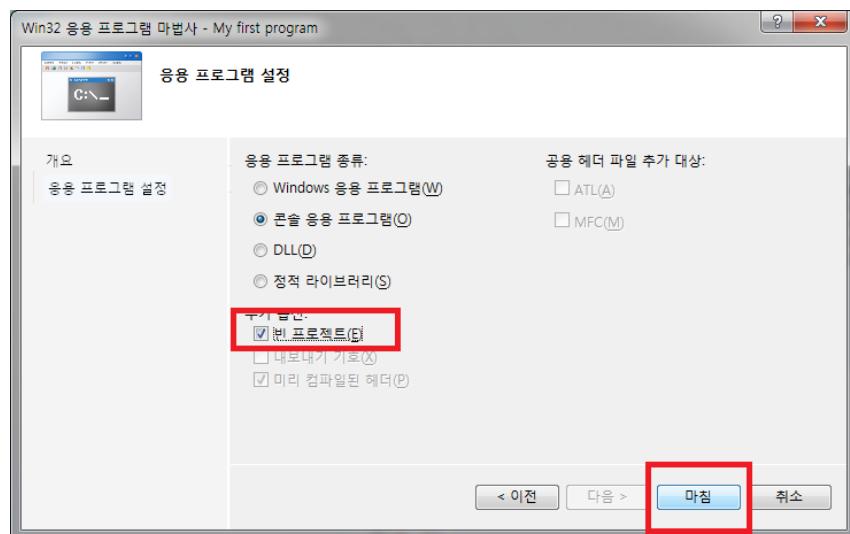
이제 새 프로젝트를 누른 후, 이전에 2008 때 처럼 Win32 콘솔 응용 프로그램을 눌러줍니다. 만일 다른 것을 눌렀을 경우 예상치 못한 오류들이 나오게 됩니다.



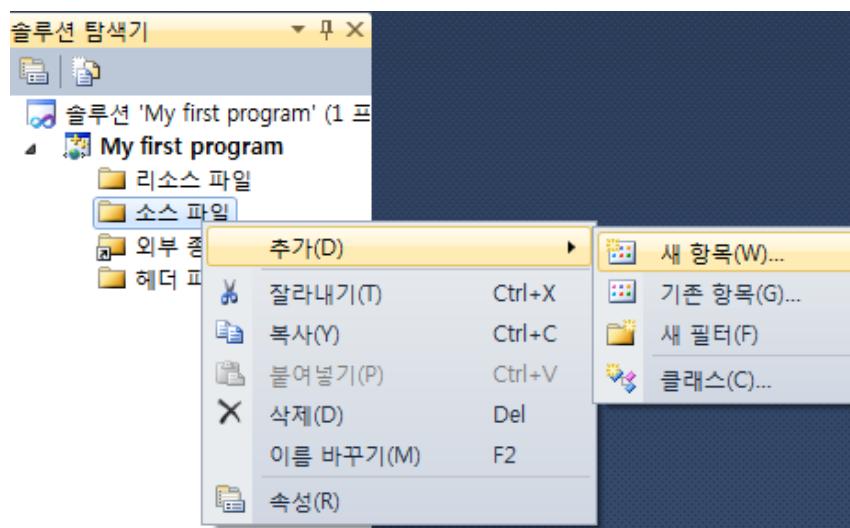
그리고 아래쪽에 이름을 아무거나 씁니다. 저의 경우 `My first program`이라고 적어주었습니다.



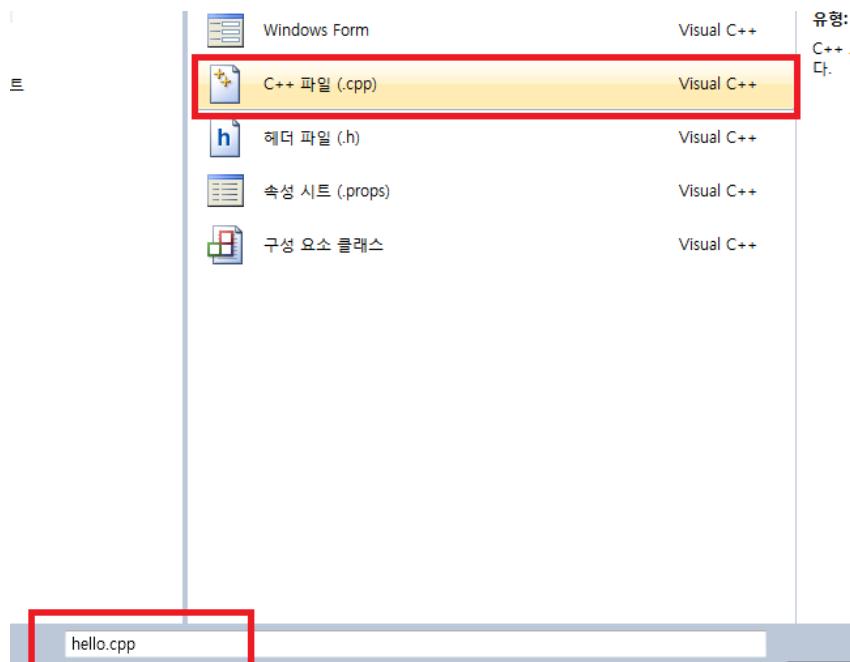
다음을 누르시고



빌드 프로젝트에 체크를 한 뒤, 마침을 누르면 마침내 새로운 프로젝트가 만들어지게 됩니다.



소스파일을 누른 후 마우스 오른쪽 클릭을 한 후, 추가로 들어가서 새 항목을 누르시면 아래와 같이 나옵니다.



여기서 C++ 파일을 선택한 후, 아래에 원하는 이름.cpp로 적으면 됩니다. 저의 경우 hello.cpp라 적었습니다. 이전에 C 언어에서는 원하는 이름.c로 적었던 것이 기억이 나지요? 파일의 확장자를 c로 하면 C 컴파일러가, cpp로 하면 C++ 컴파일러가 프로그램을 컴파일 해줍니다.

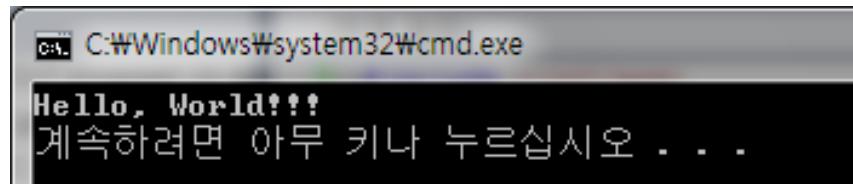
자 그럼 왼쪽에 생긴 hello.cpp를 클릭한 뒤, 나타나는 창에 아래와 같이 코드를 씁니다.

나의 첫 C++ 프로그램

```
#include <iostream>

int main() {
    std::cout << "Hello, World!!!" << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면 (이전과 똑같이 **Ctrl + F5** 를 누르면 컴파일 후 빌드까지 하여 프로그램을 출력해줍니다)



우와!!

여러분은 드디어 첫번째 C++ 프로그램을 작성하였습니다. 위 소스가 어떠한 의미를 가지고 있는지는 다음 강좌에서 다루어 보도록 하겠습니다.

첫 C++ 프로그램 분석하기

안녕하세요 여러분. 씹어먹는 C++ 두번째 강좌입니다. 지난번에는 아마도 여러분 인생 최초의 C++ 프로그램을 만들어 보았을 텐데요, 이번 강좌에서는 소스 코드를 따라가면서 분석을 하는 시간을 갖도록 하겠습니다.

사실, 지금 제 강좌를 보고 계시는 분들 중에서는 막 C 언어 공부를 끝내고 오신 분들도 많으실 텐데요, 무언가 초심자의 마음으로 돌아간 것 같지 않으세요?

C 언어에서 막 어려운 프로그래밍 하다가 C++ 오니 다시 맨 밑바닥 부터 화면에 출력하는 것을 하니 답답한 마음이 들 것도 같네요. 하지만 조금만 기다려보세요. 곧 놀라운 C++ 의 세계가 펼쳐질 것입니다.

```
#include <iostream>

int main() {
    std::cout << "Hello, World!!" << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

Hello, World!!

와 같이 나옵니다.

위 코드가 바로 지난 강좌에서 사용하였던 코드입니다. 일단 C 언어와 비슷한 점들부터 찾아보도록 합시다. 일단 맨 위에

```
#include <iostream>
```

을 보면 아하! `iostream` 이라는 헤더파일을 `include` 하고 있구나 라는 생각이 머리속에 번뜩이 셔야 합니다. 그렇지 않다면 C 언어를 다시 공부하도록 하세요! ([이 강좌를 보시면 됩니다](#))

`iostream` 헤더 파일은 C++ 에서 표준 입출력에 필요한 것들을 포함하고 있습니다. 예를 들면 아래에서 사용되는 `std::cout`이나 `std::endl`과 같은 것들을 말이지요. C 언어에서의 `stdio.h` 와 비슷하다고 보시면 됩니다. (그리고 C 와 하나 다른 점은 헤더 파일 이름 뒤에 `.h` 가 붙지 않습니다!)

```
int main()
```

네. `main` 함수를 정의하는 부분입니다. C 와 마찬가지로 C++ 에서의 `main` 함수는 프로그램이 실행될 때 가장 먼저 실행되는 함수입니다.

그리고 그 함수의 몸체를 보면

```
std::cout << "Hello, World!!" << std::endl;  
return 0;
```

와 같은 내용이 있네요. 화면에 대충 출력된 것을 보아 `std::cout` 은 화면에 무언가 출력시켜주는 것 같은데, `printf` 와 다르게 사용된 것을 보니 함수 같지는 않네요. 그리고 화면에 출력된 것을 대충 보면 "계속하려면 아무 키나 누르세요" 가 한 줄 개행되어서 나온 것을 보니 `std::endl` 은 한 줄 엔터를 쳐서 나타내라는 표시 같습니다.

그리고 마찬가지로 `main` 함수에서도 `return` 을 해주고요. 이렇게 대략 살펴보면 기존의 C 언어와 크게 다른 점은 없는 것 같습니다.

하지만 미스테리로 남아있던 부분부터 살펴보도록 합시다.

이름 공간(namespace)

먼저 `cout` 앞에 붙어 있는 `std` 의 정체부터 알아봅시다. `std` 는 C++ 표준 라이브러리의 모든 함수, 객체 등이 정의된 이름 공간(namespace)입니다.²⁾

그렇다면 이름 공간이란 것이 정확히 무엇일까요? 이름 공간은 말그대로 어떤 정의된 객체에 대해 어디 소속인지 지정해주는 것과 동일합니다.

코드의 크기가 늘어남에 따라, 혹은 다른 사람들이 쓴 코드를 가져다 쓰는 경우가 많아지면서 중복된 이름을 가진 함수들이 많아졌습니다. 따라서 C++ 에서는 이를 구분하기 위해, 같은 이름이라도, 소속된 이름 공간 이 다르면 다른 것으로 취급하게 되었습니다.

예를 들어서 같은 철수라고 해도, 서울 사는 철수와 부산 사는 철수와 다르듯이 말이지요.

```
std::cout
```

위의 경우 `std` 라는 이름 공간에 정의되어 있는 `cout` 을 의미 합니다. 만약에 `std::` 없이 그냥 `cout` 이라고 한다면 컴파일러가 `cout` 을 찾지 못합니다. 서울에 사는 철수인지 부산에 사는 철수인지 알 길이 없기 때문이지요.

이름 공간을 정의하는 방법은 아래와 같습니다. 예를 들어서 두 헤더파일 `header1.h` 와 `header2.h` 를 생각해봅시다.

2) "표준 라이브러리" 나 "객체" 가 무엇인지 아직 몰라도 괜찮습니다. 그냥 쉽게 생각하자면 `stdio.h` 가 C 에서 제공하는 라이브러리듯이 `iostream` 도 C++ 에서 제공하는 출력을 위한 표준 라이브러리 입니다.

```
// header1.h 의 내용
namespace header1 {
int foo();
void bar();
}
```

```
// header2.h 의 내용
namespace header2 {
int foo();
void bar();
}
```

위 코드에서 `header1` 에 있는 `foo` 는 `header1` 라는 이름 공간에 살고 있는 `foo` 가 되고, `header2` 에 있는 `foo` 의 경우 `header2` 라는 이름 공간에 살고 있는 `foo` 가 됩니다.

자기 자신이 포함되어 있는 이름 공간 안에서는 굳이 앞에 이름 공간을 명시하지 않고 자유롭게 부를 수 있습니다. 예를 들어서

```
#include "header1.h"

namespace header1 {
int func() {
    foo(); // 알아서 header1::foo() 가 실행된다.
}
} // namespace header1
```

`header1` 이름 공간안에서 `foo` 를 부른다면 알아서 `header1::foo()` 를 호출하게 됩니다. 그렇다고 해서 `header1` 의 이름 공간 안에서 `header2` 의 `foo` 를 못 호출하는 것은 아닌데 그냥 아래와 같이 간단하게

```
#include "header1.h"

namespace header1 {
int func() {
    foo(); // 알아서 header1::foo() 가 실행된다.
    header2::foo(); // header2::foo() 가 실행된다.
}
} // namespace header1
```

반면에 어떠한 이름 공간에도 소속되지 않는 경우라면 아래와 같이 명시적으로 이름 공간을 지정해야 합니다.

```
#include "header1.h"
#include "header2.h"
```

```
int func() {
    header1::foo(); // header1 이란 이름 공간에 있는 foo 를 호출
}
```

하지만 만일 위 같은 `foo` 을 여러번 반복적으로 호출하게 되는 경우 앞에 매번 `header1::` 을 붙이기가 상당히 귀찮을 것입니다.

그래서 아래와 같이 '나는 앞으로 `header1` 이란 이름 공간에 들어있는 `foo` 만 쓸거다!' 라고 선언할 수 있습니다.

```
#include "header1.h"
#include "header2.h"

using header1::foo;
int main() {
    foo(); // header1 에 있는 함수를 호출
}
```

뿐만 아니라, 그냥 기본적으로 `header1` 이름 공간안에 정의된 모든 것들을 `header1::` 없이 사용하고 싶다면

```
#include "header1.h"
#include "header2.h"

using namespace header1;
int main() {
    foo(); // header1 에 있는 함수를 호출
    bar(); // header1 에 있는 함수를 호출
}
```

아예 위와 같이 `using namespace header1` 과 같이 명시하면 됩니다.

물론 이 경우 역시 `header2` 에 있는 함수를 못 사용하는 것은 아니고 다음과 같이 명시적으로 써주면 됩니다.

```
#include "header1.h"
#include "header2.h"
using namespace header1;

int main() {
    header2::foo(); // header2 에 있는 함수를 호출
    foo();         // header1 에 있는 함수를 호출
}
```

그렇다면 다시 원래 예제를 살펴보도록 합시다.

```
int main() {
    std::cout << "Hello, World!!" << std::endl;
    return 0;
}
```

여기서 cout 과 endl 은 모두 iostream 헤더파일의 std 라는 이름 공간에 정의되어 있는 것들입니다. std 를 붙이기 귀찮은 사람의 경우에는 그냥

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!!" << endl;
    return 0;
}
```

로 써도 됩니다.

주의 사항

참고로 `using namespace std;` 와 같이 어떠한 이름 공간을 사용하겠다라고 선언하는 것은 권장되지 않습니다. 왜냐하면 std 에 이름이 겹치는 함수를 만들게 된다면, 오류가 발생하기 때문이지요.

게다가 C++ 표준 라이브러리는 매우 매우 거대하므로, 정말 수 많은 함수들이 존재하고 있습니다. 자칫 잘못하다가 이름을 겹치게 사용한다면, 고치느라 시간을 많이 잡아먹을 것입니다. 게다가 std 에는 매번 수 많은 함수들이 새롭게 추가되고 있기 때문에 C++ 버전이 바뀔 때마다 기존에 잘 작동하던 코드가 이름 충돌로 인해 동작하지 않게되는 문제가 발생할 수 있습니다.

따라서 권장하는 방식은 `using namespace std;` 같은 것은 사용하지 않고, `std::` 를 직접 앞에 붙여서 std 의 이름공간의 함수이다 라고 명시해주는 것이 좋습니다. 또한, 여러분이 작성하는 코드는 여러분 만의 이름 공간에 넣어서 혹시 모를 이름 충돌로 부터 보호하는 것이 중요합니다.

그렇다면 cout 은 무엇일까요? 정확히 무엇인지 말하자면 ostream 클래스의 객체로 표준 출력(C 언어에서의 stdout 에 대응됩니다) 을 담당하고 있습니다.

무슨 말인지 모르겠다고요? 괜찮습니다. 이 것이 정확히 무슨 의미인지는 나중 강좌에서 알아보도록 하겠고, 그냥 다음과 같이 쓴다는 것만 알아두시면 됩니다.

```
std::cout << /* 출력할 것 */ << /* 출력할 것 */ << ... << /* 출력할 것 */;
```

그리고 endl 은 화면에 출력해주는 '함수' 입니다. 놀라셨지요? 하지만 그냥

```
std::cout << std::endl;
```

이라 쓰면 화면에 엔터를 하나 출력해주는 것으로 기억하시면 됩니다. 물론 `endl`에 대해서도 나중에 다루어 보도록 하겠습니다 :)

이름 없는 이름 공간

잠깐 짚고 넘어가자면, C++에서는 재미있게도 이름 공간에 굳이 이름을 설정하지 않아도 됩니다.

이 경우 해당 이름 공간에 정의된 것들은 해당 파일 안에서만 접근할 수 있게 됩니다. 이 경우 마치 `static` 키워드를 사용한 것과 같은 효과를 냅니다.

```
#include <iostream>

namespace {
    // 이 함수는 이 파일 안에서만 사용할 수 있습니다.
    // 이는 마치 static int OnlyInThisFile() 과 동일합니다.
    int OnlyInThisFile() {}

    // 이 변수 역시 static int x 와 동일합니다.
    int only_in_this_file = 0;
} // namespace

int main() {
    OnlyInThisFile();
    only_in_this_file = 3;
}
```

예를 들어서 위 경우 `OnlyInThisFile` 함수나 `only_in_this_file` 변수는 해당 파일 안에서만 접근할 수 있습니다. 헤더파일을 통해서 위 파일을 받았다 하더라도 (물론 `main` 함수 부분은 무시하고), 저 익명의 `namespace` 안에 정의된 모든 것들은 사용할 수 없게 됩니다.

생각 해보기

문제 1

화면에 출력되는 것들을 바꾸어보자.

문제 2

아래 문장은 화면에 어떻게 출력될까요?

```
std::cout << "hi" << std::endl  
      << "my name is "  
      << "Psi" << std::endl;
```

C++ 와 C 언어의 공통 문법 구조

안녕하세요 여러분~ Psi 입니다. 저의 C++ 세번째 강좌 이네요. 이번 강좌에는 여러분과 많이 친숙할 듯 한데요, 왜냐하면 C++ 이 C 언어에서 빌려온 여러가지 문법들을 어떻게 사용하는지 살펴볼 것입니다.

사실 C 언어에서 작성된 코드를 그대로 C++ 에 붙여 넣기 해도 큰 문제가 없다고 말해도 과언이 아닌 만큼 C++ 은 C 언어의 문법을 거의 완전하게 포함하고 있습니다.³⁾

```
// 변수의 정의
#include <iostream>

int main() {
    int i;
    char c;
    double d;
    float f;

    return 0;
}
```

일단 가장 기초적인 부분으로 변수를 정의하는 것부터 봅시다. 사실 위 코드를 볼 때 의 C 언어에서 작성한 코드라고 말해도 똑같이 생각할 것입니다. 변수를 정의하는 부분에서 만큼은 C 언어때와 달라진 것이 없습니다.

물론 변수 명 이름 작성 규칙도 바뀐 것이 없습니다. 변수명도 C 언어 때와 마찬가지로 알파벳과 _ 기호, 숫자들을 사용할 수 있고 그 외의 것들은 사용할 수 없습니다.⁴⁾ 또한 변수 이름의 맨 앞부분에는 숫자가 오면 안됩니다.

Google 의 C++ 변수 이름 짓기 가이드에 따르면 [여기](#)에서 보실 수 있습니다. 변수 이름을 지을 때 아래와 같은 점들을 고민하는 것을 권장합니다.

먼저 변수의 이름은 변수의 이름만을 딱 보았을 때 무엇을 하는지 확실히 알 수 있어야 합니다.

```
int number_of_people; // OK
double interest_rate; // OK
```

```
int num_of_ppl; // BAD
double intrst_rt; // BAD
```

3) 물론 모든 C 코드가 C++ 코드에 포함되는 것은 아닙니다. C 컴파일러로 컴파일 되지만 C++ 에서는 되지 않는 요소들이 있습니다.

4) 컴파일러에 따라서 한글 변수명도 사용 가능하지만 권장하지 않습니다.

맨 위의 두 변수 이름들은 딱 보았을 때 '아, 사람의 인원수이고 아래는 이자율 이구나' 라는 느낌이 확 들지만 아래의 두 변수 이름을 보았을 때에는 그러한 느낌을 받기 힘듭니다. 따라서 변수 이름이 조금 길더라도 확실히 이해할 수 있는 변수 이름을 짓는 것이 매우 중요합니다.

둘째로 변수 이름의 띠어쓰기에 관한 규칙인데, 보통 변수 이름을 지을 때

```
int number_of_people; // OK
int NumberOfPeople; // OK
```

위 처럼 두 가지 방법을 사용하는데 하나는 이름의 띠어쓰기 부분에 `_`를 넣는 것이고 다른 하나는 띠어쓰기 부분에 대문자로 구분하는 것인데, 저의 경우 전자의 방법을 선호합니다. 물론 이는 사람마다 개인차가 있겠지만, 가장 중요한 것은 한 코드 안에 위 두 방식을 혼용하지 않는 것입니다. 예를 들어서

```
int NumberOf_People; // BAD
```

는 전혀 권장할 것이 못됩니다. 저는 앞으로 제 소스에서 전자의 방법을 취할 것입니다.⁵⁾

아무튼 제 C++ 강좌를 보고 계실 여러분들의 실력은 이미 상당한 수준(아마도 C 언어 정도는 다룰 줄 아실 분들) 이실 테니 이러한 내용들도 중간에 짬짬히 이야기 하고 지나갈 것입니다 ㅎㅎ

이렇게 해서 변수의 정의는 C 나 C++ 이 차이가 없다는 것을 보실 수 있으셨을 것입니다. 마찬가지로 배열이나 포인터를 정의하는 방법도 C 나 C++ 이 동일합니다. 물론 포인터의 경우 C 에서 * 와 & 가 하였던 역할을 C++ 에서도 그대로 물려 받았습니다.

예를 들어 C 에서

```
int arr[10];
int *parr = arr;

int i;
int *pi = &i;
```

게 했던 것들을 C++ 에서는 어떻게 할까요? 답은 간단합니다. 똑같이

```
int arr[10];
int *parr = arr;

int i;
int *pi = &i;
```

5) 참고로 구글의 경우 변수의 이름 내에 띠어쓰기를 모두 `_`로 구분하는 전자의 방식을 취하고 있습니다. 반면에 함수의 이름의 경우 대문자를 사용하는 후자의 방식을 사용합니다.

쓰면 됩니다. 쉽지요? 어떠한 것들을 선언하는 방법은 정말로 C++ 이나 C 가 차이가 하나도 없음을 알 수 있습니다.

그렇다면 다른 문법 구조들은 어떻까요. 반복문(`for`, `while`)이라던지 조건문(`if`, `else`, `switch`)이라던지.. 일단 `for` 부터 살펴보도록 합시다.

```
// C++ 의 for 문
#include <iostream>

int main() {
    int i;

    for (i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
0
1
2
3
4
5
6
7
8
9
```

와우! 정말로 똑같습니다. C 언어 때와 `for` 문은 달라진 것이 없군요. 그렇다면 C++에서 `for` 문을 이용해 1부터 10 까지 더하는 문장을 어떻게 만들까요.

```
/* 1 부터 10 까지 합*/
#include <iostream>

int main() {
    int i, sum = 0;

    for (i = 1; i <= 10; i++) {
        sum += i;
```

```
}

std::cout << "합은 : " << sum << std::endl;
return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
합은 : 55
```

역시 기존의 C 언어 때와 동일합니다.

사실 **for** 문은 C++에서도 그 문법이 바뀌지 않았습니다. 그냥 여러분이 기억하시는대로 사용하시면 됩니다. 한 가지 달라진 점이 있다면 변수의 선언이 반드시 최상단에 있어야 되는 것은 아닙니다. 기존의 C에서는 변수를 정의할 때 언제나 소스 맨 위부분에 선언을 하였습니다. 예를 들어

```
int i, sum = 0;

for (i = 1; i <= 10; i++) {
    sum += i;
}
```

와 같이 말이지요. 하지만 C++에서는 변수를 사용하기 직전 어느 위치에서든지 변수를 선언할 수 있게 됩니다. 예를 들어서 다음과 같이 해도 상관이 없습니다.

```
/* 변수는 변수 사용 직전에 선언해도 된다.*/
#include <iostream>

int main() {
    int sum = 0;

    for (int i = 1; i <= 10; i++) {
        sum += i;
    }

    std::cout << "합은 : " << sum << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

합은 : 55

로 동일한 결과를 보입니다. 그렇다면 `while` 문은 어떨까요. 역시 동일합니다

```
/* while 문 이용하기 */
#include <iostream>

int main() {
    int i = 1, sum = 0;

    while (i <= 10) {
        sum += i;
        i++;
    }

    std::cout << "합은 : " << sum << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

합은 : 55

와 같이 우리가 기존에 알고 있었던 `while` 문과 동일한 결과를 나타냅니다.

C++에서 `if - else` 문 역시 C와 동일한 문법 구조로 되어 있습니다. 아래의 예제를 볼까요.

```
/* 행운의 숫자 맞추기 */
#include <iostream>

int main() {
    int lucky_number = 3;
    std::cout << "내 비밀 수를 맞추어 보세요~" << std::endl;

    int user_input; // 사용자 입력

    while (1) {
        std::cout << "입력 : ";
        std::cin >> user_input;
        if (lucky_number == user_input) {
            std::cout << "맞추셨습니다~~" << std::endl;
            break;
    }
}
```

```

    } else {
        std::cout << "다시 생각해보세요~" << std::endl;
    }
}
return 0;
}

```

성공적으로 컴파일 했다면

실행 결과

```

내 비밀 수를 맞추어 보세요~
입력 : 5
다시 생각해보세요~
입력 : 6
다시 생각해보세요~
입력 : 3
맞추셨습니다~~

```

와 같이 역시 우리가 C에서 생각했던 대로 동일하게 나옵니다. 위 코드에서 살펴볼 부분은 바로

```

std::cout << "입력 : ";
std::cin >> user_input;

```

입니다. 일단 cout은 앞에서 배웠지만 << 를 이용하여 출력을 시키지요. 그리고, 이미 예상했다 싶이 cin은 사용자로부터 입력을 받아서 >> 를 통해 user_input에 넣습니다. cin도 마찬가지로 std에 정의되어 있기에 std::cin과 같이 사용해야 합니다.

scanf에서는 &를 붙였는데 C++에서는 편리하게도 앞에 & 연산자를 붙일 필요가 없습니다. 심지어, scanf에서는 int 형태로 입력받을지 아니면 char인지에 따라서 %d 냐 %c 냐로 구분하였는데 여기서는 그냥 변수를 보고 cin이 알아서 처리해 줍니다. 매우 편리합니다.

아직까지 여러분은 cin이 뭔지, cout이 뭔지 이게 도대체 함수인건지 변수인건지 구조체인건지, 기본의 쉬프트 연산자로 사용되었던 <<나 >> 는 뭔지 도저히 감이 잡히지 않을 것입니다. 그래도 상관은 없습니다. 일단 사용하세요! 사용하시고 편리하게 될 쯤에는 제 강좌에서 뭔지 배우실 것입니다.

```

if (lucky_number == user_input) {
    std::cout << "맞추셨습니다~~" << std::endl;
    break;
} else {
    std::cout << "다시 생각해보세요~" << std::endl;
}

```

C 에서와 마찬가지로 C++ 에서도 `if` 문은 동일하게 사용함을 알 수 있습니다. 그렇다면 `switch` 문은 어떨까요?

```
// switch 문 이용하기
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main() {
    int user_input;
    cout << "저의 정보를 표시해줍니다" << endl;
    cout << "1. 이름" << endl;
    cout << "2. 나이" << endl;
    cout << "3. 성별" << endl;
    cin >> user_input;

    switch (user_input) {
        case 1:
            cout << "Psi !" << endl;
            break;

        case 2:
            cout << "99 살" << endl;
            break;

        case 3:
            cout << "남자" << endl;
            break;

        default:
            cout << "궁금한게 없군요~" << endl;
            break;
    }
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

저의 정보를 표시해줍니다

1. 이름

2. 나이

3. 성별

1

Psi !

으로 switch 문이 잘 작동하는 것을 보실 수 있습니다.

```
switch (user_input) {  
    case 1:  
        cout << "Psi ! " << endl;  
        break;  
  
    case 2:  
        cout << "99 살" << endl;  
        break;  
  
    case 3:  
        cout << "남자" << endl;  
        break;  
  
    default:  
        cout << "궁금한게 없군요~" << endl;  
        break;  
}
```

위를 보면 기존의 C에서 사용하였던 switch 문과 다를 바 없다는 것을 아실 수 있습니다. 그렇습니다. 늘 말해왔듯이 C와 C++은 기본적인 문법 구조(조건문; if, else, switch, 제어문; for, while, break, continue 등등)는 똑같습니다.

자 그럼 이번 강좌에서는 이것으로 마치도록 하겠습니다:) C에서 기본적으로 다뤘던 예제들을 C++로 바꿔 보는 것도 재밌는 작업 일 것 같습니다.

C++ 의 참조자 (레퍼런스)

안녕하세요 여러분! 오랜만에 찾아온 Psi 입니다. 사실 이전 강좌에서부터 강조해왔지만 C 언어에서 되던 것이 C++에서는 거의 100% 된다고 보셔도 무방합니다.

즉 기초적인 문법이 거의 똑같다는 것이지요. 이전 강좌에서는 기본적인 구문들, 예를 들어 변수의 정의 방법이나, 조건문(if, else, switch), 반복문(for, while, do-while) 등등을 살펴 보았는데요, 이번 강좌에서는 C++ 에 새로 도입된 새로운 개념인 참조자 (혹은 레퍼런스라고도 많이 합니다.)에 대해서 다루어 볼 것입니다.

참조자의 도입

```
#include <iostream>

int change_val(int *p) {
    *p = 3;

    return 0;
}

int main() {
    int number = 5;

    std::cout << number << std::endl;
    change_val(&number);
    std::cout << number << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과
5

3

와 같이 나옵니다.

저의 C 언어 강좌를 잘 따라오신 분이라면 위 코드를 무리없이 이해하실 수 있을 것입니다.

`change_val` 함수의 인자 `p`에 `number`의 주소값을 전달하여, `*p`를 통해 `number`를 참조하여 `number`의 값을 3으로 바꾸었습니다.

C 언어에서는 어떠한 변수를 가리키고 싶을 땐 반드시 포인터를 사용해야만 했습니다. 그런데 C++에서는 다른 변수나 상수를 가리키는 방법으로 또 다른 방식을 제공하는데, 이를 바로 참조자(레퍼런스 - reference)라고 부릅니다.

```
#include <iostream>

int main() {
    int a = 3;
    int& another_a = a;

    another_a = 5;
    std::cout << "a : " << a << std::endl;
    std::cout << "another_a : " << another_a << std::endl;

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a : 5
another_a : 5
```

와 같이 나옵니다.

```
int a = 3;
```

먼저 우리는 위와 같이 간단히 `int` 형 변수인 `a`를 정의하였고 그 안에 3이란 값을 넣어주었습니다.

```
int& another_a = a;
```

그 후에 우리는 `a`의 참조자 `another_a`를 정의하였습니다. 이 때 참조자를 정하는 방법은, 가리키고자 하는 타입 뒤에 &를 붙이면 됩니다.

위 처럼 `int` 형 변수의 참조자를 만들고 싶을 때에는 `int&` 를, `double` 의 참조자를 만드려면 `double&` 로 하면 됩니다. 심지어 `int*` 와 같은 포인터 타입의 참조자를 만드려면 `int*&` 로 쓰면 됩니다.

위와 같이 선언함으로써 우리는 `another_a` 는 `a` 의 참조자다! 라고 공표하게 되었습니다. 이 말은 즉슨 `another_a` 는 `a` 의 또다른 이름 이라고 컴파일러에게 알려주는 것입니다. 따라서 `another_a` 에 어떠한 작업을 수행하든 이는 사실상 `a` 에 그 작업을 하는 것과 마찬가지 입니다.

```
another_a = 5;
std::cout << "a : " << a << std::endl;
std::cout << "another_a : " << another_a << std::endl;
```

따라서 위처럼 `another_a` 에 5 를 대입하였지만 실제로 `a` 의 값을 확인해보면 5 로 바뀌었음을 확인할 수 있습니다.

어뜨게 보면 참조자와 포인터는 상당히 유사한 개념입니다. 포인터 역시 다른 어떤 변수의 주소값을 보관함으로써 해당 변수에 간접적으로 연산을 수행할 수 있기 때문이죠. 하지만 레퍼런스와 포인터는 몇 가지 중요한 차이점이 있습니다.

레퍼런스는 반드시 처음에 누구의 별명이 될 것인지 지정해야 합니다.

레퍼런스는 정의 시에 반드시 누구의 별명인지 명시 해야 합니다. 따라서

```
int& another_a;
```

와 같은 문장은 불가능 합니다. 반면의 포인터의 경우

```
int* p;
```

는 전혀 문제가 없는 코드입니다.

레퍼런스가 한 번 별명이 되면 절대로 다른 이의 별명이 될 수 없다.

레퍼런스의 또 한 가지 중요한 특징으로 한 번 어떤 변수의 참조자가 되버린다면, 이 더이상 다른 변수를 참조할 수 없게 됩니다.¹⁾

예를 들어서

1) 학창시절 별명이 무덤까지 가는 것과 비슷하다고 보시면 됩니다 :)

```
int a = 10;
int &another_a = a; // another_a 는 이제 a 의 참조자!

int b = 3;
another_a = b; // ??
```

아래와 같은 코드를 살펴봅시다. 마지막에 `another_a = b;` 문장은 어떤 의미 일까요? `another_a` 보고 다른 변수인 `b` 를 가리키라고 하는 것일까요? 아닙니다! 이는 그냥 a에 b의 값을 대입하라는 의미입니다. 앞서 말했듯이 `another_a` 에 무언가를 하는 것은 사실상 `a` 에 무언가를 하는 것과 동일하다고 했으므로 이 문장은 그냥 `a = b` 와 동치입니다.

참고로

```
&another_a = b;
```

요건 어떤가요? 라고 물어보실 수도 있는데 위 문장은 그냥 `&a = b;` 가 되어서 말이 안되는 문장이 됩니다.

반면에 포인터는 어떨까요.

```
int a = 10;
int* p = &a; // p 는 a 를 가리킨다.

int b = 3;
p = &b // 이제 p 는 a 를 버리고 b 를 가리킨다
```

위와 같이 누구를 가리키는지 자유롭게 바꿀 수 있습니다.

레퍼런스는 메모리 상에 존재하지 않을 수 도 있다.

포인터의 경우를 생각해봅시다. 우리가 아래와 같이 포인터 `p` 를 정의 한다면

```
int a = 10;
int* p = &a; // p 는 메모리 상에서 당당히 8 바이트를 차지하게 됩니다.
```

`p` 는 당당히 메모리 상에서 8 바이트를 차지하는 녀석이 됩니다 (물론 32 비트 시스템에서는 4바이트겠죠!) 그런데 레퍼런스의 경우를 생각해봅시다.

```
int a = 10;
int &another_a = a; // another_a 가 자리를 차지할 필요가 있을까?
```

만일 내가 컴파일러라면 another_a 위해서 메모리 상에 공간을 할당할 필요가 있을까요? 아니죠! 왜냐하면 another_a 가 쓰이는 자리는 모두 a로 바꿔치기 하면 되니까요. 따라서 이 경우 레퍼런스는 메모리 상에 존재하지 않게 됩니다. 물론 그렇다고 해서 항상 존재하지 않은 것은 아닙니다. 아래 예제를 보실까요.

함수 인자로 레퍼런스 받기

```
#include <iostream>

int change_val(int &p) {
    p = 3;

    return 0;
}

int main() {
    int number = 5;

    std::cout << number << std::endl;
    change_val(number);
    std::cout << number << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
5
3
```

위 코드는 앞서 포인터를 사용해서 number 를 change_val 안에 전달한 코드를 참조자를 이용해서 바꿔본 것입니다.

먼저 가장 중요한 부분으로

```
int change_val(int &p) {
```

와 같이 함수이 인자로 참조자를 받게 하였습니다. 여기서

아까 int& p 는 안된다고 하지 않으셨나요?

라고 물을 수 있는데 사실 p 가 정의되는 순간은 change_val(number) 로 호출할 때 이므로 사실상 int& p = number 가 실행된다고 생각하면 됩니다. 따라서 전혀 문제가 없죠.

```
change_val(number);
```

아무튼 위와 같이 참조자 p 에게 너는 앞으로 **number** 의 새로운 별명이야 라고 알려주게 됩니다. 여기서 중요한 점은 포인터가 인자일 때와는 다르게 **number** 앞에 & 를 붙일 필요가 없다는 점입니다. 이는 참조자를 정의할 때 그냥 **int& a = b** 와 같이 한 것과 일맥상통합니다.

```
int change_val(int &p) {
    p = 3;

    return 0;
}
```

그 후 **change_val** 안에서 **p = 3;** 이라 하는 것은 **main** 함수의 **number**에 **number = 3;** 을 하는 것과 정확히 같은 작업입니다.

자 보세요. 어느 방식이 좀 더 깔끔하신 것 같나요?

여러가지 참조자 예시들

```
// 참조자 이해하기

#include <iostream>

int main() {
    int x;
    int& y = x;
    int& z = y;

    x = 1;
    std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;

    y = 2;
    std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;

    z = 3;
    std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
x : 1 y : 1 z : 1
x : 2 y : 2 z : 2
x : 3 y : 3 z : 3
```

예상하고 계셨던 결과 인가요?

```
int x;
int& y = x;
```

먼저 위와 같이 x 의 참조자로 y 를 정의하였습니다. 이제 y 는 x 의 또 다른 별명이 됩니다.

```
int& z = y;
```

그렇다면 다음 문장을 봅시다. 간혹 아래와 같이 고개를 가우뚱 할 수 도 있습니다.

아까 어떤 타입 T 의 참조자 타입은 T& 래매. 그런데 여기서 y 가 int& 니까 y 의 참조자 타입은 int&& 가 되야 하지 않을까?

좋은 질문입니다. 하지만 참조자의 참조자라는 말의 의미를 생각해보면 사실 말이 안된다는 것을 알 수 있습니다. 굳이 별명의 별명을 만들 필요는 없으니까요! 실제로 C++ 문법 상 참조자의 참조자를 만드는 것은 금지되어 있습니다.

```
int& z = y;
```

즉, 위 문장은 결국 x 의 참조자를 선언해라와 같은 의미가 되서, z 역시 x 의 참조자가 될 것입니다. 따라서 y 와 z 모두 x 의 참조자가 됩니다.

```
x = 1;
std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;

y = 2;
std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;

z = 3;
std::cout << "x : " << x << " y : " << y << " z : " << z << std::endl;
```

결과적으로 위 문장들은 모두 1,1,1 과 2,2,2 와 3,3,3 을 출력합니다.

아무래도 처음에 참조자를 접하시는 분들은 왜 굳이 포인터로 할 수 있는 것을 왜 참조자로 해야 하냐고 물을 수 있습니다. 하지만 참조자를 사용하게 되면 불필요한 & 와 * 가 필요 없기 때문에 코드를 훨씬 간결하게 나타낼 수 있습니다.

예를 들어서 지난 강좌에서 변수 입력시 배웠던 `cin` 을 기억하시나요? 아마 사용자로부터 변수에 값을 입력 받을 때 다음과 같이 했었을 것입니다.

```
std::cin >> user_input;
```

그런데 무언가 이상하지 않으세요? 예전에 `scanf` 로 이용할 때 분명히

```
scanf("%d", &user_input);
```

와 같이 항상 주소값을 전달해 주었는데 말이죠. 왜냐하면 어떤 변수의 값을 다른 함수에서 바꾸기 위해서는 항상 포인터로 주소값을 전달해야하기 때문이니까요. 하지만 여기서는 `cin` 이라는 것에 그냥 `user_input` 을 전달했는데 잘 작동합니다.

왜 그럴까요? 바로 `cin` 이 레퍼런스로 `user_input` 을 받아서 그렇습니다. 따라서 구질 구질하게 & 를 `user_input` 앞에 붙일 필요가 없게 되는 것입니다.

상수에 대한 참조자

```
#include <iostream>

int main() {
    int &ref = 4;

    std::cout << ref << std::endl;
}
```

위와 같은 소스를 살펴봅시다. 일단 컴파일 해보면 아래와 같은 오류가 나타날 것입니다.

컴파일 오류

```
error C2440: 'initializing' : cannot convert from 'int' to 'int &'
```

왜 오류가 나타날까요? 아마 여러분들은 다 알고 계시겠지요. 위 상수 값 자체는 리터럴 이기 때문에 (리터럴이 무엇인지 모르겠으면 여기로) 만일 위와 같이 레퍼런스로 참조한다면

```
ref = 5;
```

로 리터럴의 값을 바꾸는 말도 안되는 행위가 가능하게 됩니다. 따라서 C++ 문법 상 상수 리터럴을 일반적인 레퍼런스가 참조하는 것은 불가능하게 되어 있습니다.

물론 그 대신에;

```
const int &ref = 4;
```

상수 참조자로 선언한다면 리터럴도 참조 할 수 있습니다. 따라서

```
int a = ref;
```

는 `a = 4;` 와는 문장과 동일하게 처리됩니다

레퍼런스의 배열과 배열의 레퍼런스

먼저 레퍼런스의 배열이 과연 가능한 것인지에 대해 부터 생각해봅시다. 앞서 말했듯이 레퍼런스는 반드시 정의와 함께 초기화를 해주어야 한다고 했습니다. 따라서 여러분의 머리속에는 다음과 같이 레퍼런스의 배열을 정의하는 것을 떠올렸을 것입니다.

```
int a, b;
int& arr[2] = {a, b};
```

컴파일을 해보면

컴파일 오류

```
error C2234: 'arr' : arrays of references are illegal
```

레퍼런스의 배열을 불법(illegal) 이라고 합니다. 왜 불법인지 한 번 C++ 규정을 찾아 보면, 표준안 8.3.2/4 를 보면 놀랍게도

There shall be no references to references, no arrays of references, and no pointers to references
레퍼런스의 레퍼런스, 레퍼런스의 배열, 레퍼런스의 포인터는 존재할 수 없다.

정말로 언어 차원에서 불가능 하다고 못 박아버렸습니다. 그러면 도대체 왜 안될까요? 웬지 위에서

```
int& arr[2] = {a, b};
```

로 해서 *arr[0]* 는 *a* 를 의미하고 *arr[1]* 은 *b* 를 의미하고.. 로 만들면 안될까요.

이와 같은 주장을 하기 전에 먼저 C++ 상에서 배열이 어떤 식으로 처리되는지 생각해봅시다. 문법상 배열의 이름은 (arr) 첫 번째 원소의 주소값으로 변환이 될 수 있어야 합니다. 이 때문에 *arr[1]* 과 같은 문장이 ** (arr + 1)* 로 바뀌어서 처리될 수 있기 때문이죠.

그런데 주소값이 존재한다라는 의미는 해당 원소가 메모리 상에서 존재한다 라는 의미와 같습니다. 하지만 레퍼런스는 특별한 경우가 아닌 이상 메모리 상에서 공간을 차지하지 않습니다. 따라서 이러한 모순 때문에 레퍼런스들의 배열을 정의하는 것은 언어 차원에서 금지가 되어 있는 것입니다.

그렇다고 해서 그와 반대인 배열들의 레퍼런스 가 불가능 한 것은 아닙니다.

```
#include <iostream>

int main() {
    int arr[3] = {1, 2, 3};
    int (&ref)[3] = arr;

    ref[0] = 2;
    ref[1] = 3;
    ref[2] = 1;

    std::cout << arr[0] << arr[1] << arr[2] << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

231

먼저 가장 중요한 첫 두줄을 살펴봅시다.

```
int arr[3] = {1, 2, 3};
int (&ref)[3] = arr;
```

위와 같이 *ref* 가 *arr* 를 참조하도록 하였습니다.

따라서 *ref[0]* 부터 *ref[2]* 가 각각 *arr[0]* 부터 *arr[2]* 의 레퍼런스가 됩니다. 포인터와는 다르게 배열의 레퍼런스의 경우 참조하기 위해선 반드시 배열의 크기를 명시해야 합니다.

따라서 int (&ref)[3] 이라면 반드시 크기가 3인 int 배열의 별명이 되어야 하고 int (&ref)[5] 라면 크기가 5인 int 배열의 별명이 되어야 합니다.

```
int arr[3][2] = {1, 2, 3, 4, 5, 6};
int (&ref)[3][2] = arr;
```

역시 일차원 배열을 했을 때와 동일합니다.

레퍼런스를 리턴하는 함수

먼저 아래 코드를 살펴봅시다.

```
int function() {
    int a = 2;
    return a;
}

int main() {
    int b = function();
    return 0;
}
```

아마 여기 까지 따라 오신 분들이라면 무리 없이 이해할 수 있겠죠. 제가 주목하고 싶은 부분은 바로 이 부분입니다.

```
int b = function();
```

여기서 무슨 일이 일어났을까요?

이미 잘 아시겠지만, `function` 안에 정의된 `a`라는 변수의 값이 `b`에 복사 되었습니다. 여기서 주목할 점은 복사 되었다는 점입니다.

`function`이 종료되고 나면 `a`는 메모리에서 사라지게 됩니다. 따라서 더 이상 `main` 안에서는 `a`를 만날 길이 없습니다.

지역변수의 레퍼런스를 리턴?

그 다음 예시를 살펴봅시다.

```
int& function() {
    int a = 2;
```

```

    return a;
}

int main() {
    int b = function();
    b = 3;
    return 0;
}

```

만일 컴파일 한다면 아래와 같은 경고가 나오고 (컴파일 오류는 아닙니다.)

컴파일 오류

```

test.cc: In function ‘int& function()’:
test.cc:3:10: warning: reference to local variable ‘a’ returned
  ← [-Wreturn-local-addr]
  3 |     return a;
      ^
test.cc:2:7: note: declared here
  2 |     int a = 2;
      ^

```

실제로 실행해보면

실행 결과

```
[1] 7170 segmentation fault (core dumped) ./test
```

위와 같이 런타임 오류가 발생하게 되었습니다.

과연 뭐가 문제였을까요?

```

int& function() {
    int a = 2;
    return a;
}

```

function 의 리턴 타입은 `int&` 입니다. 따라서 참조자를 리턴하게 됩니다. 그런데 문제는 리턴하는 `function` 안에 정의되어 있는 `a` 는 함수의 리턴과 함께 사라진다는 점입니다.

```

int b = function();

```

위 문장은 사실상

```
int& ref = a;

// 근데 a 가 사라짐
int b = ref; // !!!
```

와 같은 의미 인데, `function` 이 레퍼런스를 리턴하면서 원래 참조하고 있던 변수가 이미 사라져버렸으므로 오류가 발생하게 됩니다. 쉽게 말해 본체는 이미 사라졌지만 별명만 남아 있는 상황입니다.

이와 같이 레퍼런스는 있는데 원래 참조 하던 것이 사라진 레퍼런스를 맹글링 레퍼런스 (Dangling reference) 라고 부릅니다. *Dangling* 이란 단어의 원래 뜻은 약하게 결합대서 달랑달랑 거리는 것을 뜻하는데, 레퍼런스가 참조해야 할 변수가 사라져서 혼자서 덩그러니 남아 있는 상황과 유사하다고 보시면 됩니다.

주의 사항

따라서 위처럼 레퍼런스를 리턴하는 함수에서 지역 변수의 레퍼런스를 리턴하지 않도록 조심해야 합니다.

외부 변수의 레퍼런스를 리턴

그렇다면 이 경우는 어떨까요?

```
int& function(int& a) {
    a = 5;
    return a;
}

int main() {
    int b = 2;
    int c = function(b);
    return 0;
}
```

이 `function` 역시 레퍼런스를 리턴하고 있습니다. 하지만 아까와의 차이점은

```
int& function(int& a) {
    a = 5;
    return a;
}
```

위와 같이 인자로 받은 레퍼런스를 그대로 리턴 하고 있습니다.

`function(b)` 를 실행한 시점에서 `a` 는 `main` 의 `b` 를 참조하고 있게 됩니다. 따라서 `function` 이 리턴한 참조자는 아직 살아있는 변수인 `b` 를 계속 참조 합니다.

```
int c = function(b);
```

결국 위 문장은 그냥 `c` 에 현재의 `b` 의 값인 5 를 대입하는 것과 동일한 문장이 됩니다.

그렇다면 이렇게 참조자를 리턴하는 경우의 장점이 무엇일까요? C 언어에서 엄청나게 큰 구조체가 있을 때 해당 구조체 변수를 그냥 리턴하면 전체 복사가 발생해야 해서 시간이 오래걸리지만, 해당 구조체를 가리키는 포인터를 리턴한다면 그냥 포인터 주소 한 번 복사로 매우 빠르게 끝납니다.

마찬가지로 레퍼런스를 리턴하게 된다면 레퍼런스가 참조하는 타입의 크기와 상관 없이 딱 한 번의 주소값 복사로 전달이 끝나게 됩니다. 따라서 매우 효율적이죠!

참조자가 아닌 값을 리턴하는 함수를 참조자로 받기

이번에는 반대로 함수가 값을 리턴하는데 참조자로 받는 경우를 생각해봅시다.

```
int function() {
    int a = 5;
    return a;
}

int main() {
    int& c = function();
    return 0;
}
```

컴파일 하였다면 아래와 같은 오류가 발생합니다.

실행 결과

```
test.cc: In function ‘int main()’:
test.cc:7:20: error: cannot bind non-const lvalue reference of
     type ‘int&’ to an rvalue of type ‘int’
    7 |     int& c = function();
          |~~~~~^~
```

컴파일 오류를 읽어보면 상수가 아닌 레퍼런스가 `function` 함수의 리턴값을 참조할 수 없다는 의미가 되겠습니다.²⁾

2) 컴파일 메세지를 자세히 보면 lvalue, rvalue 이야기나 나오는데, 나중 강좌에서 lvalue 나 rvalue 가 뭔지 자세히 다룰 테니 아직은 궁금한 것을 참아주세요.

```
int& c = function();
```

왜 c 는 function 의 리턴값을 참조할 수 없는 것일까요? 이는 아까전 상황과 마찬가지로 함수의 리턴값은 해당 문장이 끝난 후 바로 사라지는 값이기 때문에 참조자를 만들게 되면 바로 다음에 랭글링 레퍼런스가 되버리기 때문입니다. 따라서 만약에

```
int& c = function();
c = 2;
```

와 같은 작업을 하게 된다면 앞서 보았던 런타임 오류를 보시게 될 것입니다.

하지만 C++ 에서 중요한 예외 규칙이 있습니다. 바로 다음 코드를 살펴보시죠.

```
#include <iostream>

int function() {
    int a = 5;
    return a;
}

int main() {
    const int& c = function();
    std::cout << "c : " << c << std::endl;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

c : 5

와 같이 나옵니다.

```
const int& c = function();
```

이번에도 역시 function() 의 리턴값을 참조자로 받았습니다. 그런데, 이 const 참조자로 받았더니 문제없이 컴파일 되었습니다.

```
std::cout << "c : " << c << std::endl;
```

그리고 심지어 그 리턴값도 제대로 출력됩니다.

원칙상 함수의 리턴값은 해당 문장이 끝나면 소멸되는 것이 정상입니다. 따라서 기존에 `int&` 로 받았을 때에는 컴파일 자체가 안되었습니다. 하지만 예외적으로 상수 레퍼런스로 리턴값을 받게 되면 해당 리턴값의 생명이 연장됩니다. 그리고 그 연장되는 기간은 레퍼런스가 사라질 때 까지입니다.

이번 강좌에서 다룬 것이 상당히 많은 데 간단히 정리해보자면 다음과 같습니다.

	함수에서 값 리턴 (<code>int f()</code>)	함수에서 참조자 리턴 (<code>int& f()</code>)
값 타입으로 받음(<code>int a = f()</code>)	값 복사됨	값 복사됨. 다만 지역 변수의 레퍼런스를 리턴하지 않도록 주의
참조자 타입으로 받음 (<code>int& a = f()</code>)	컴파일 오류	가능. 다만 마찬가지로 지역 변수의 레퍼런스를 리턴하지 않도록 주의
상수 참조자 타입으로 받음 (<code>const int& a = f()</code>)	가능	가능. 다만 마찬가지로 지역 변수의 레퍼런스를 리턴하지 않도록 주의

자 이렇게 C++ 상에서 레퍼런스를 사용하는 방법에 대해서 간단히 다루어보았습니다. 다음 강좌에서는 본격적으로 C++ 의 객체 지향 프로그래밍 개념에 대해서 다루어보겠습니다.

생각해보기

문제 1

레퍼런스가 메모리 상에 반드시 존재해야 하는 경우는 어떤 경우가 있을까요? 그리고 메모리 상에 존재할 필요가 없는 경우는 또 어떤 경우가 있을까요? (난이도 : 上)

4

C++ 의 세계로

안녕하세요~ 여러분. 오랜 공백기간을 끊고 찾아온 Psi 입니다. 그동안 많이 기다리셨죠? 이제부터 본격적으로 이전의 C 에서 탈피하여 C++ 의 세계로 인도해드릴 것입니다.

메모리를 관리하는 문제는 언제나 중요한 문제입니다. 프로그램이 정확하게 실행되기 위해서는 컴파일 시에 모든 변수의 주소값이 확정되어야만 합니다. 하지만, 이를 위해서는 프로그램에 많은 제약이 따르기 때문에 프로그램 실행 시에 자유롭게 할당하고 해제할 수 있는 힙(heap)이라는 공간이 따로 생겼습니다.

하지만 이전에 컴파일러에 의해 어느정도 안정성이 보장되는 스택(stack) 과는 다르게 힙은 사용자가 스스로 제어해야 하는 부분인 만큼 책임이 따릅니다. [위 문단이 이해되지 않는 분이라면 이 글을 읽어보도록 합시다](#)

C 언어에서는 `malloc` 과 `free` 함수를 지원하여 힙 상에서의 메모리 할당을 지원하였습니다. C++에서도 마찬가지로 `malloc` 과 `free` 함수를 사용할 수 있습니다.

하지만, 언어 차원에서 지원하는 것으로 바로 `new` 와 `delete` 라고 할 수 있습니다. `new` 는 말 그대로 `malloc` 과 대응되는 것으로 메모리를 할당하고 `delete` 는 `free` 에 대응되는 것으로 메모리를 해제합니다. 그럼 한 번 어떻게 이를 사용하는지 살펴보겠습니다.

```
/* new 와 delete 의 사용 */
#include <iostream>

int main() {
    int* p = new int;
    *p = 10;

    std::cout << *p << std::endl;

    delete p;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

10

위와 같이 int 영역이 잘 할당 되어서 10 이 출력이 되었음을 알 수 있습니다.

```
int* p = new int;
```

먼저 위와 같이 int 크기의 공간을 할당하여 그 주소값을 p에 집어 넣었음을 알 수 있습니다. new를 사용하는 방법은

```
T* pointer = new T;
```

와 같습니다. T에는 임의의 타입이 들어가겠지요. 그리고 이제 p 위치에 할당된 공간에

```
*p = 10;
```

를 통해서 값을 집어넣었고 이를 출력하였습니다. 마지막으로 할당된 공간을 해제하기 위해서 delete를 사용하였는데

```
delete p;
```

위와 같이 delete p를 하게 되면 p에 할당된 공간이 해제됩니다. 물론 delete로 해제할 수 있는 메모리 공간은 사용자가 new를 통해서 할당한 공간만 가능합니다.

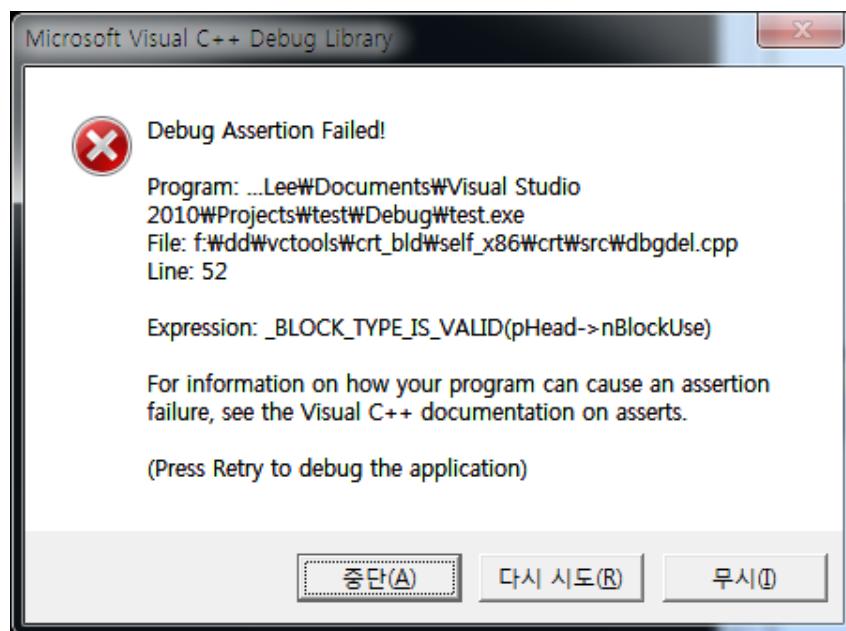
```
/* 지역 변수 delete 하기 */
#include <iostream>

int main() {
    int a = 5;

    delete &a;

    return 0;
}
```

만일 위처럼 지역 변수를 무리하게 delete로 해제해버리려 한다면



위와 같이 Heap 이 아닌 공간을 해제하려고 한다는 경고 메세지가 나타나게 됩니다.

new 로 배열 할당하기

```
/* new 로 배열 할당하기 */

#include <iostream>

int main() {
    int arr_size;
    std::cout << "array size : ";
    std::cin >> arr_size;
    int *list = new int[arr_size];
    for (int i = 0; i < arr_size; i++) {
        std::cin >> list[i];
    }
    for (int i = 0; i < arr_size; i++) {
        std::cout << i << "th element of list : " << list[i] << std::endl;
    }
    delete[] list;
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
array size : 5
1
4
2
6
8
0th element of list : 1
1th element of list : 4
2th element of list : 2
3th element of list : 6
4th element of list : 8
```

위 소스에는 많은 새로운 내용이 담겨 있으니 차근 차근 살펴보도록 합시다.

```
int arr_size;
std::cout << "array size : ";
std::cin >> arr_size;

int *list = new int[arr_size];
```

먼저 위와 같이 배열의 크기를 잡을 `arr_size`라는 변수를 정의하였고 그 값을 입력 받았습니다. 그리고 `list`에 `new`를 이용하여 크기가 `arr_size`인 `int` 배열을 생성하였습니다. 배열을 생성할 때에는 `[]`를 이용해 배열의 크기를 넣어주면 되는데,

```
T* pointer = new T[size];
```

`T`를 임의의 타입이라 하면 위와 같이 하면 됩니다. 따라서 `list`는 이제 크기가 `arr_size`인 `int` 배열을 가지게 됩니다. 사실 여기서 한 가지 놀라운 점이 있는데 C 에선 변수의 선언을 모두 최상단에 몰아서 해야 했지만 C++은 그렇지 않다는 점입니다. C++에서는 편리하게도 소스의 아무대서나 변수를 선언할 수 있으며, 그 변수는 그 변수를 포함하고 있는 중괄호를 빠져 나갈 때 소멸됩니다. 예를 들어서 아래와 같은 코드를 봅시다.

```
// 생략
{
    int a = 4;
    std::cout << "안에서 a : " << a;
}

std::cout << "밖에서 a : " << a;
```

만일 여러분이 위와 같이 중괄호 안에서 변수 a 를 선언하였다면 변수 a 의 사용 범위는 그 중괄호 안 뿐입니다. 즉 안에서 a 는 4 로 잘 출력이 되겠지만 "밖에서 a :" 문장은 오류가 나게 될 것입니다. 왜냐하면 변수 a 가 그 곳에서는 존재하지 않기 때문이죠. 따라서 여러분은 이 점을 항상 유의하셔야겠습니다. 또한 한 가지 더 재미있는 점은, 어떤 변수를 사용할 때 컴파일러는 그 변수를 가장 가까운 범위(scope) 부터 찾게 됩니다. 예를 들어 아래의 코드를 보세요

```
int a = 4;
{
    std::cout << "외부의 변수 1" << a << std::endl;
    int a = 3;
    std::cout << "내부의 변수" << a << std::endl;
}

std::cout << "외부의 변수 2" << a << std::endl;
```

외부의 변수 1 의 출력 결과를 보면 자명하게 4 가 될 것입니다. 왜냐하면 그 때까지 정의된 변수 a 는 앞서 정의한 `int a = 4` 하나 거든요. 하지만 그 아래에서 새롭게 `int a = 3;` 으로 정의한 후 (분명히 이 변수는 위에서의 `a = 4` 와 다른 변수입니다) 내부의 변수를 출력해보면 3 이 나옵니다. 이는 앞서 말한 '가장 가까운 범위에서 찾는다' 라는 원칙 하에서 내부의 변수 `<< a` 에서 a 를 사용할 때 가장 가까운 범위 내에 있는 변수는 바로 같은 중괄호 내에 있는 `int a = 3;` 이므로 3 이 출력되는 것이지요. 바깥 범위에 있는 `a = 4` 의 a 는 내부에 있는 `a = 3` 의 a에 의해 가려집니다. 그리고 중괄호를 지나면서 이 내부 변수는 소멸됩니다.

이제 다시 외부의 변수 2 를 출력할 때에는 `a = 4` 에서의 a 가 출력되어 4 가 나오게 되는 것입니다. 하지만 아래와 같이 같은 범위 안에 동일한 변수를 선언하는 것은 허용되지 않습니다.

```
int a;
a = 3;
int a;
```

왜냐하면 그 다음에 a 를 사용하였을 때 둘 다 같은 범위 안에 있기 때문에 컴파일러는 어떠한 a 를 사용할 지 모르기 때문이지요. 한 가지 당부하고 싶은 말은 결코 위와 같은 변수의 선언 범위를 고려할 만큼 조zano하게 변수 이름을 짓지 말자 입니다.

사람의 눈은 컴파일러가 아니기 때문에 위와 같이 변수 이름을 중복해서 사용한다면 큰 혼동이 있을 뿐더러 나중에 디버깅시 곤란해질 수 있으니 항상 변수 이름은 다르게 짓는 습관을 들이는 것이 좋습니다.

이제 다시 본론으로 돌아와서 원래 코드를 살펴봅시다.

```
for (int i = 0; i < arr_size; i++) {
    std::cin >> list[i];
}
```

```
for (int i = 0; i < arr_size; i++) {
    std::cout << i << "th element of list : " << list[i] << std::endl;
}
```

그림과 같이 `for` 문 안에서 `int i` 를 선언하여 `cin` 을 이용하여 `list` 를 받았습니다. 이렇게 `for` 문 초기식에서 정의된 `i` 는 과연 `for` 문 안에서 정의된 것일까요. `for` 문 밖에서 정의된 것일까요? 즉 `i` 를 `for` 문 밖에서도 사용할 수 있을까요?

답은 안에서 정의된 것입니다. 즉 `i` 는 밖에서 사용할 수 없지요. 이렇게 `for` 문 초기식에 `i` 를 정의해버리면 좋은 점이 설사 밖에 `i` 를 다른 용도로 사용했더라도 for 문 안에서는 i 를 카운터 (counter) 로 사용할 수 있기 때문에 오류가 발생할 가능성이 줄어듭니다.

아무튼 이렇게 해서 `list` 의 각 원소들을 입력받고 또 이를 출력할 수 있었습니다.

```
delete[] list;
```

마지막으로 살펴볼 부분은 `delete` 하는 부분으로 앞서 `new []` 를 이용해서 할당 하였으면 아래에서는 delete [] 를 통해서 해제하면 됩니다. 즉 `new - delete` 가 짝을 이루고 `new []` 와 `delete []` 가 짝을 이루는 것이지요.

돌아온 마이펫

아마도 예전에 저의 C 언어 강좌를 보신 분들이라면 `switch` 문을 배우면서 간단하게 만들어보았던 마이펫을 기억하실 것입니다. 이번에는 그 때 기억을 살려서 동물 관리 프로그램을 간단하게 만들 어보았습니다. 소스를 보기 전에 여러분들도 간단히 만들어보시는 것도 좋을 것 같습니다. 일단 조건은 다음과 같습니다.

- 동물(`struct Animal`) 이라는 구조체를 정의해서 이름(`char name[30]`), 나이(`int age`), 체력(`int health`), 배부른 정도(`int food`), 깨끗한 정도의(`int clean`) 값을 가진다.
- 처음에 동물 구조체의 포인터 배열(`struct Animal* list[30]`)을 만들어서 사용자가 동물을 추가할 때마다 하나씩 생성한다.
- `play`라는 함수를 만들어서 동물의 상태를 변경하고 `show_stat` 함수를 만들어서 지정하는 동물의 상태를 출력한다.
- 1 턴이 지날 때마다 동물의 상태를 변경한다.

대략 이 정도로만 하고 저는 한번 아래와 같이 소스를 짜보았습니다.

```

#include <iostream>

typedef struct Animal {
    int age; // 나이
    char name[30]; // 이름
    int food; // 먹임 글자수
    int clean; // 청결 글자수
    int health; // 건강 글자수
} Animal;

void create_animal(Animal *animal) {
    std::cout << "이름은 무엇인가요? ";
    std::cin >> animal->name;
    std::cout << "나이는 몇 살인가요? ";
    std::cin >> animal->age;
    std::cout << "건강 상태는 어떤가요? ";
    std::cin >> animal->health;
    std::cout << "먹임 상태는 어떤가요? ";
    std::cin >> animal->food;
    std::cout << "청결 상태는 어떤가요? ";
    std::cin >> animal->clean;
}

void play(Animal *animal) {
    animal->health += 10;
    animal->food -= 20;
    animal->clean -= 30;
    animal->age += 10;
}

void one_day_pass(Animal *animal) {
    animal->health -= 10;
    animal->food -= 20;
    animal->clean -= 30;
    animal->age += 1;
}

void show_start(Animal *animal) {
    std::cout << "이름 : " << animal->name << "나이 : " << animal->age << "건강 : " << animal->health << "먹임 : " << animal->food << "청결 : " << animal->clean << std::endl;
}

int main() {
    Animal *list[10];
    int animal_num = 0;
    for (int i = 0; i < 10; i++) {
        list[i] = new Animal();
    }
    std::cout << "1. 물고기 2. 고양이 3. 개 ";
    std::cin >> list[animal_num];
    list[animal_num] = new Animal();
    std::cout << "동물 이름을 입력하세요 ";
    std::cin >> list[animal_num]->name;
    std::cout << "동물 나이를 입력하세요 ";
    std::cin >> list[animal_num]->age;
    std::cout << "동물 청결 상태를 입력하세요 ";
    std::cin >> list[animal_num]->clean;
    std::cout << "동물 건강 상태를 입력하세요 ";
    std::cin >> list[animal_num]->health;
    std::cout << "동물 먹임 상태를 입력하세요 ";
    std::cin >> list[animal_num]->food;
    std::cout << "동물을 출발시킵니다 ";
    std::cout << std::endl;
    std::cout << "동물 출발 후 청결 상태는 ";
    std::cout << list[animal_num]->clean << "이며 ";
    std::cout << "동물 출발 후 건강 상태는 ";
    std::cout << list[animal_num]->health << "이며 ";
    std::cout << "동물 출발 후 먹임 상태는 ";
    std::cout << list[animal_num]->food << "이며 ";
    std::cout << "동물 출발 후 나이는 ";
    std::cout << list[animal_num]->age << "입니다 ";
    std::cout << std::endl;
}

```

```

std::cin >> input;

switch (input) {
    int play_with;
    case 1:
        list[animal_num] = new Animal;
        create_animal(list[animal_num]);

        animal_num++;
        break;
    case 2:
        std::cout << "누구랑 놀게? : ";
        std::cin >> play_with;

        if (play_with < animal_num) play(list[play_with]);

        break;

    case 3:
        std::cout << "누구껄 보게? : ";
        std::cin >> play_with;
        if (play_with < animal_num) show_stat(list[play_with]);
        break;
}

for (int i = 0; i != animal_num; i++) {
    one_day_pass(list[i]);
}
}

for (int i = 0; i != animal_num; i++) {
    delete list[i];
}
}
}

```

성공적으로 컴파일 하였다면

실행 결과

누구껄 보게? : 0

pig의 상태

체력 : 70

배부름 : -40

청결 : -10

1. 동물 추가하기

2. 놀기

3. 상태 보기

그림과 같이 잘 작동됨을 알 수 있습니다. 사실 위 코드에는 그다지 특별한 것이 없습니다. 일단 주요 부분을 살펴볼까요.

```
typedef struct Animal {
    char name[30]; // 이름
    int age; // 나이

    int health; // 체력
    int food; // 배부른 정도
    int clean; // 깨끗한 정도
} Animal;
```

위와 같이 Animal 구조체를 만들어서 **typedef** 를 통해 **struct Animal** 을 Animal 로 간추렸습니다. 그리고,

```
list[animal_num] = new Animal;
create_animal(list[animal_num]);
```

위와 같이 Animal 을 new 로 생성하면 create_animal 함수를 통해서 Animal 의 각 값을 초기화 해주었고요, 사용자가 놀기를 요청하면

```
if (play_with < animal_num) play(list[play_with]);
```

위 처럼 play 함수를 호출해서 놀기를 수행하였습니다. 마지막으로 사용자가 각 동물의 상태를 보기 원한다면

```
if (play_with < animal_num) show_stat(list[play_with]);
```

show_stat 함수를 호출해서 사용자가 지정한 동물의 상태를 출력하도록 하였습니다. 사실 매우 간단한 이야기입니다. 그런데 무언가 상당히 낭비 같지 않으세요? 사용자가 play 를 호출하면 list[play_with] 를 전달해야만 했습니다.

하지만 그러면 어떨까요? Animal 구조체 자체에 함수를 만들어서, 각 구조체 변수가 각각 자신의 함수를 가지게 되는 것입니다. 그러면 list[play_with]->play() 와 같이 "각 변수 자신의 함수" 를 호출하여 자신의 데이터를 이용해서 처리하게 되는 것이지요.

이렇게 할 수 만 있다면 play 함수에 귀찮게 인자를 전달할 필요도 없고 또 함수 내부에서도

```
void play(Animal *animal) {
    animal->health += 10;
    animal->food -= 20;
    animal->clean -= 30;
}
```

위와 같이 귀찮게 `animal->` 을 앞에 붙여가면서 작업할 필요도 없습니다. 왜냐하면 `list[play_with]->play()` 라고 했을 때 `play` 는 '자기 자신의 함수' 이기 때문에

```
health += 10;
food -= 20;
clean -= 30;
```

이렇게 해도 된다는 것입니다. 왜냐하면 `list[play_with]->play()` 이라 했을 때 `health`, `food`, `clean` 이 의미하는 것이 `list[play_with]` 의 것이기 때문입니다. 상당히 괜찮은 생각 아닌가요? 위 소스에서 불편한 점은 이것만이 아닙니다. `new` 를 통해 새로운 동물을 할당하는 부분을 살펴봅시다.

```
list[animal_num] = new Animal;
create_animal(list[animal_num]);
```

`new Animal` 을 통해 동물을 생성한 다음에 반드시 `create_animal` 함수를 호출해야만 했습니다. 왜냐하면 `new Animal` 을 통해 새로운 `Animal` 을 할당한 상태라면 `health`, `food` 등 변수에 아무런 값이 들어가 있지 않기 때문이죠. 다시 말해서 만일 프로그래머가 실수로 `Animal` 을 생성한 후 `create_animal` 을 호출하지 않는다면 나중에 `play` 함수 등을 호출 할 때 끔찍한 오류가 발생하게 됩니다. 초기화 되지 않는 값에 연산을 수행하는 오류이지요.

그렇다면 만일 `new` 로 새로운 `Animal` 을 생성할 때 자동으로 호출되는 함수가 있으면 어떨까요. 즉 `new` 가 알아서 호출해주는 그런 함수. 그렇게 된다면 사용자는 귀찮게 `create_animal` 을 호출할 필요도 없고, 자동으로 호출되는 함수에서 멤버 변수들 (`health`, `food`, ...) 들을 초기화 준다면 나중에 초기화 되지 않아서 생기는 오류도 막을 수 있을 것입니다.

자 이제. 여러분은 위 동물 프로그램이 크나큰 인기를 얻어서 확장팩을 제작하게 되었습니다. `Animal` 이라 단순하게 분류하였던 것을 조금 더 세분화 해서 `Bird`, `Fish` 등으로 나누어서 처리 하려고 합니다. `Bird` 와 `Fish` 는 기본적으로 `Animal` 과 유사하지만 `Bird` 에는 현재 날고 있는 고도를 나타내는 변수인 `int height;` 가 새로 추가되고, `Fish` 에는 현재 잠수하고 있는 수심을 나타내고 있는 변수인 `int deep;` 이 추가되었습니다.

그러면 여러분은 아래와 같이 소스를 짤 것입니다.

```
typedef struct Bird {
    char name[30]; // 이름
    int age; // 나이

    int health; // 체력
    int food; // 배부른 정도
    int clean; // 깨끗한 정도

    // 여기까지는 Animal 과 동일하다.
```

```
int height; // 나는 고도

} Bird;

typedef struct Fish {
    char name[30]; // 이름
    int age; // 나이

    int health; // 체력
    int food; // 배부른 정도
    int clean; // 깨끗한 정도

    // 여기까지는 Animal 과 동일하다.
    int deep; // 현재 깊이

} Fish;
```

와 정말로 시간 낭비가 아닐 수 없었습니다. `Animal` 과 거의 똑같지만 조금조금씩 달라진 것 때문에 구조체를 새로 두 개나 만들어야 한다는 말입니다. 그냥 `Animal` 과 동일한 부분은 가져다 쓰고 새로 추가된 부분만 살포시 추가해 주면 안될까요?

그런데 문제는 이 뿐만이 아닙니다. 여러분은 더이상 `Animal*` 배열 하나로 살 수 없게 됩니다. 이제 `Animal*` 따로, `Fish*` 따로, `Bird*` 따로 만들어서 관리해야 될 뿐더러 `play` 함수, `show_stat` 함수도 모두 `Animal`, `Fish`, `Bird`에 맞게 각각 새로 작성해야 합니다. 다시 말해서 고작 `int height` 나 `int deep` 변수 하나 추가한 덕분에 여태까지 짠 코드 양의 2 배를 써야 하는 위기 상황에 처했습니다.

정말 말이 안되지요. 하지만 C 언어의 세계에 살고 있던 여러분은 이 모든 것을 꿋꿋히 해내고 있었을 것입니다.

그리고 이제. 이곳을 탈출할 때가 온 것 같습니다.

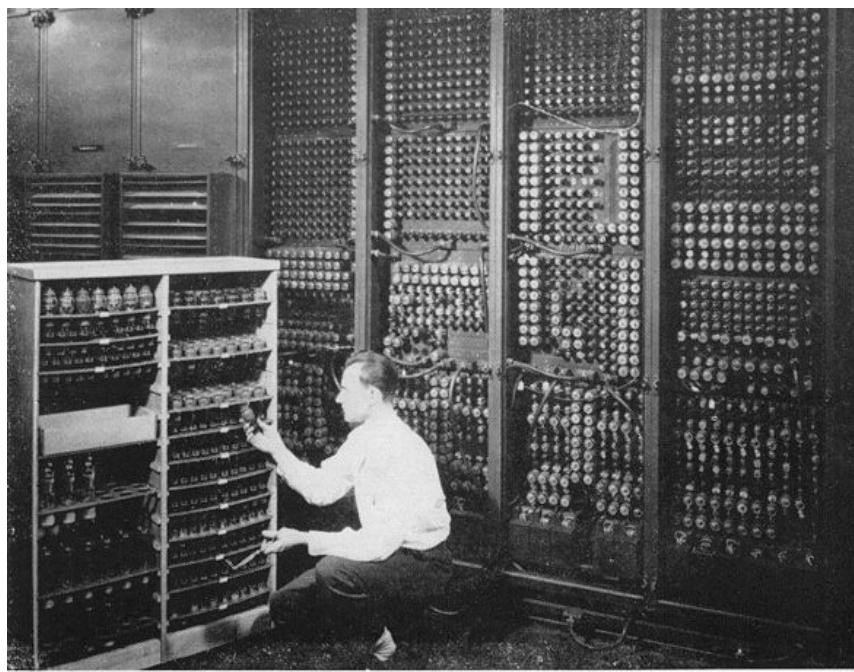
여러분,

객체지향프로그래밍의 세계로 오신것을 환영합니다.

객체지향프로그래밍의 시작

안녕하세요 여러분~ 그간 침묵을 깨고 오래간만에 C++ 강좌를 이어 나가고자 합니다. 앞선 3 강을 읽으셨던 분들은 다 느껴셨겠지만 기존의 C 언어를 통해서 대형 프로젝트를 개발하기 위해서는 많은 어려움들이 있기 마련입니다. (그래도 아직도 많은 수의 프로그램이 C로 쓰여지고 있습니다)

사실 컴퓨터 프로그래밍 언어는 이러한 난관을 뚫고서 발전해 나갔습니다. 초기의 컴퓨터는 이름만 들어도 유명한 에니악(ENIAC)과 같이 거대한 크기를 자랑하였습니다. 이러한 컴퓨터를 어떻게 프로그래밍 했냐고요? 아래 사진 오른쪽에 보이는 수 많은 진공관들 사이의 전선 연결을 바꾸어 가며 전기 신호를 전달했다고 합니다.



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

즉, 우리가 컴퓨터 상에서 소스 코드를 치는 것을 직접 손으로 조작했다고 보시면 됩니다. 정말로 끔찍한 일이 아닐 수 없지요. 이것을 '컴퓨터 언어'라고 보기에는 조금 무리가 있을 것 같습니다. 아무튼 진공관 상에서 단순한 전선 연결 배치로 명령을 했던 언어가 1세대 컴퓨터 언어입니다.

참고로 위 그림 아래에 써있는 말이 무슨 말이냐면, 이와 같이 '컴퓨터 프로그래밍' 을 수행하였는데 어딘가에서 오류가 났을 때 어떠한 진공관이 고장났다는 의미인데 (아니면 전선 배치를 잘못했거나), 이를 수정하기 위해서 19000 여개의 달하는 진공관들을 확인해야 했다고 하네요.

1950 년대 이후 컴퓨터 내장 메모리가 만들어지고, 실질적으로 '컴퓨터 프로그래밍' 을 할 수 있게 되자, 2 세대 컴퓨터 언어가 등장하였는데요, 가장 첫번째로 나온 것이 어셈블리어 (Assembly language) 입니다. 언어라고 하기에는 컴퓨터에 직접 명령을 내리는 기계어 (0 과 1 로 이루어 짐) 에 사람들이 보기 쉽게 문자열을 대응 시킨 것에 가까운 형태였습니다.

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32 PROC      ; procedure begins here
    CMP AX,97   ; compare AX to 97
    JL  DONE    ; if less, jump to DONE
    CMP AX,122  ; compare AX to 122
    JG  DONE    ; if greater, jump to DONE
    SUB AX,32   ; subtract 32 from AX
    DONE: RET    ; return to main program
    SUB32 ENDP   ; procedure ends here
```

FIGURE 17. Assembly language

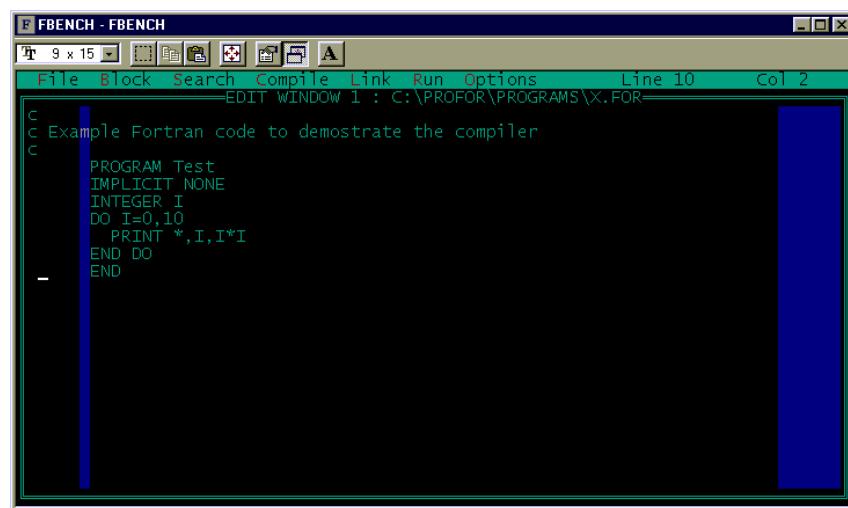
위 그림은 IBM PC 어셈블리어로 쓰여진 것을 캡쳐한 것입니다. 어셈블리어는 말그대로 기계어와 1 : 1 대응 되어 있기 때문에, 할 수 있는 동작이 매우 단순합니다. 즉, 우리가 "1 + 1 을 계산해봐!" 라고 C 언어에서 쉽게 말하는 것을 어셈블리어에서는 "음, 1 을 어디에 저장하고, 또 1 을 어디에 저장하고, 이제 CPU 에 메모리 어디에서 1 을 불러오고, 또 CPU 에 메모리 어디에서 1 을 불러오고, 이들에 덧셈을 수행해!" 라고 말하는 것과 동일한 것입니다.

참으로 노가다가 아닐 수 없지요. 위 캡쳐한 사진도 사실 보면, 어떠한 값이 97 과 122 사이에 있다면 32 를 빼라는 의미 인데, 이를 일일히 지정하고 있는 것을 볼 수 있습니다. 여기까지의 컴퓨터 언어는 저급 언어 (Low level) 이라 부르며, 수준이 낮다는 것이 아니라, 조금 더 기계어에 근접해 있다는 것을 의미합니다. 그리고, 이 때부터 초보적인 수준의 고급 언어(High level) 가 등장하게 되었는데, 은행과 같은 금융 쪽에서 많이 쓰였던 COBOL 이나, 수치 계산용으로 쓰이는 FORTRAN 이 등장하게 됩니다. 그리고 꽤나 쉬운 언어였던 BASIC 도 한 몷 하게 되지요.

```

000017      03 EMP-MNAME          PIC X(10).
000018      03 EMP-LNAME          PIC X(10).
000019      *                   .
000020      02 EMP-ADDRESS         PIC X(26).
000021      *                   .
000022      02 EMP-ADDRESS-DETAILS REDEFINES EMP-ADDRESS.
000023          03 EMP-STREET        PIC X(10).
000024          03 EMP-CITY         PIC X(10).
000025          03 EMP-PINCODE       PIC X(6).
000026      *                   .
000027      02 EMP-CONTACT        PIC 9(18).
000028      *                   .
000029      02 EMP-CONTACT-DETAILS REDEFINES EMP-CONTACT.
000030          03 EMP-PHONE-1      PIC 9(09).
000031          03 EMP-PHONE-2      PIC 9(09).
000032      *                   .
000033      PROCEDURE DIVISION.

```



위의 FORTRAN 코드를 살작 보면 알겠지만 어셈블리어를 통해서는 매우 복잡한 명령 (예를 들어서 화면에 출력한다던지) 을 단순하게 처리하고 있음을 알 수 있습니다. 이런 2 세대 초기의 언어들은 어셈블리어에 비해서는 획기적인 발전이 있을 수 있었습니다. 하지만 문제는 데이터 타입이나, 프로그램 문법 구조가 완전하지가 않아서 복잡한 데이터 타입을 단순히 모두 배열로 처리한다던지, 논리 구조를 모두 `goto` 문으로 처리한다던지의 문제가 있었습니다.

이렇게 하게 되면 오류가 발생하여도 찾기가 굉장히 힘든, 소위 말하는 스파게티 코드가 만들어지지요. 스파게티코드란, 스파게티 처럼 프로그램의 논리 구조가 뒤엉킨 상태를 의미합니다. 그래서 유명한 컴퓨터 과학자 다익스트라 (Dijkstra) 가 *Go to statement considered harmful* 이라는 유명한 글을 남기게 됩니다. 아무튼 이 때문에 조금 더 체계적인 프로그래밍 언어가 크게 필요로 해졌습니다.

그래서 짠하고 나타난 것이 3 세대 프로그래밍 언어, 절차 지향 언어 (Procedural programming language) 라고 불리는 파스칼(Pascal) 언어와 그 뒤를 이어서 C 언어가 등장하게 됩니다.

```

File Edit Run Compile Options Debug Break/watch
Line 15 Col 39 Insert Indent Unindent * D:NONAME.PAS
program KenLovesTurboPascal;
uses
  crt;
var
  age: Integer;
  name: String;
  message: String;
begin
  ClrScr;
  name := 'Ken Egozi';
  age := 30;
  if age < 10 then
    message := ' loves Turbo Pascal'
  else
    message := ' loved Turbo Pascal';
  write (name);
  writeln (message);
end.

```

위 그림은 파스칼 언어로, 우리가 지금 아는 C 언어와 상당히 유사합니다. 함수라는 개념이 완성되었고, (이전에도 있었지만 불완전한 면이 있었다), 변수의 타입 (위 그림의 var 부분을 보면 알 수 있지만 정수형, 문자열 등등) 이 정립되어서 기존의 배열을 남발하던 것에서 벗어날 수 있게 됩니다.

절차(?)를 지향한다는 말이 이해가 잘 안가실 텐데, 영어로 보면, *Procedure*를 지향하는 언어, 즉 프로시저 (함수)를 지향한다는 것입니다. 다시 말해 프로그램을 설계할 때 중요한 부분을 하나의 프로시저로 만들어서 쪼개어 처리한다는 것입니다. 물론 기존의 언어들에서도 프로시저라는 것은 존재하였지만, 함수의 인자와 같은 개념이 없었고, 비로소 이 때야 완전한 함수라는 것이 만들어지게 되는 것입니다.

그렇게 해서 절차 지향 언어로 몇십년을 벼텨왔습니다. 그러나, 프로그램의 크기가 예전보다 상상도 할 수 없을 만큼 거대해 지자 새로운 패러다임이 필요하게 되었는데요, 그것이 바로 **객체 지향 언어(Object oriented language)**입니다. 이를 사용하는 언어는 C++ 을 비롯한 Java, Python, C# 등등 아마 90년대 이후에 생긴 언어들은 대부분 객체 지향 언어 일 것입니다. ¹⁾

객체란?

그럼, 절차 지향적 언어의 뭐가 부족해서였는지 객체 지향 언어를 필요로 하게 된 것일까요? 먼저 지난 강좌의 Animal 구조체를 가져와서 살펴 봅시다. 우리는

```

typedef struct Animal {
  char name[30]; // 이름
}

```

1) 물론 C++ 을 단순히 객체 지향 프로그래밍 언어 이다라고 단정 짓는 것은 어폐가 있습니다. 나중에 템플릿을 이용해서 제너릭(Generic)한 프로그램을 작성할 수도 있고 모던 C++ 에 들어서는 함수형 프로그래밍을 할 수 있습니다. 물론 기존의 C 스타일로 (비록 권장하지는 않지만) 절차 지향적인 프로그래밍을 할 수도 있겠고요. 즉, C++ 은 멀티 패러다임 (Multi paradigm) 언어입니다.

```

int age;           // 나이

int health;      // 체력
int food;         // 배부른 정도
int clean;        // 깨끗한 정도
} Animal;

```

위와 같이 `Animal` 구조체를 정의한 후, `animal` 변수를 만들어서 이를 필요로 하는 함수들에게

```
play(list[play_with]);
```

이와 같이 전달해 주었습니다. 그런데, 곰곰히 생각해 보면 `Play` 함수에 인자로 전달하는 것이 매우 불필요해보입니다. 이 상황을 그림을 생각하면, 마치 **러시아식 유머**처럼 "Play 가 `Animal` 을 합니다!" 라고 볼 수 있는데, 사실은 "`Animal` 이 `Play` 를 한다" 가 더 맞기 때문이지요.

다시 말해서 `Animal` 자체가 `Play` 를 하는 것이지, `Play` 가 `Animal` 을 해주는 것이 아닙니다. 만일 `Animal` 자체가 `Play` 를 한다 라는 개념을 생각하게 된다면, 다음과 같이 생각할 수 있을 것입니다.

```

Animal animal;

// 여러가지 초기화 (생략)

animal.play();    // 즉 내가 (animal 이) Play 를 한다!
animal.sleep();   // 내가 sleep 을 한다!

```

이렇게 하면 `play` 함수에 `animal` 을 인자로 주지 않아도 됩니다. 왜냐하면 내가 `play` 하는 것이기 때문에 내 정보는 이미 `play` 함수가 다 알고 있기 때문입니다. `play` 함수는 나의 상태들, 예를 들어서 체력이나, 배고픔 정도나 피곤한 정도 등을 모두 알 수 있기 때문에 나에 대한 적절한 처리를 할 수 있게 되는 것입니다. 즉, `animal` 은 자신의 상태를 알려주는 변수(variable) 과, 자신이 하는 행동들 (play, sleep 등등) 을 수행하는 함수(method) 들로 이루어졌다고 볼 수 있습니다.

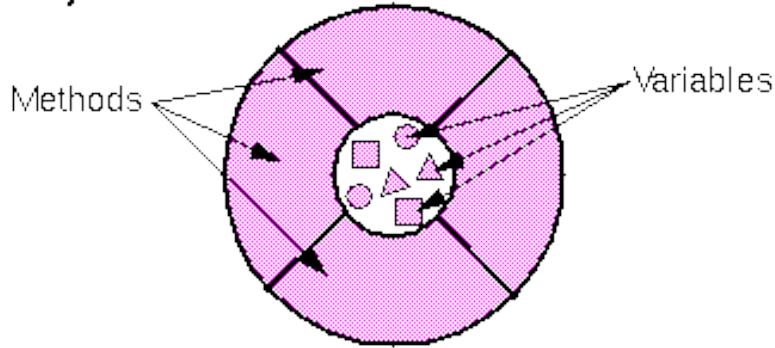
결과적으로 객체는 다음과 같이 정의됩니다.

객체란, 변수들과 참고 자료들로 이루어진 소프트웨어 덩어리 이다.

이 때 객체가 현실 세계에서의 존재하는 것들을 나타내기 위해서는 **추상화(abstraction)** 라는 과정이 필요합니다. 컴퓨터 상에서 현실 세계를 100% 나타낼 수 없는 것이기 때문에, 적절하게 컴퓨터에서 처리할 수 있도록 바꾸는 것인데, 예를 들어서 핸드폰의 경우 '전화를 한다', '문자를 보낸다' 와 같은 것들은 '핸드폰이 하는 것' 이므로 함수로 추상화시킬 수 있고, 핸드폰의 상태를 나타내는 것들, 예를 들어서 자기 자신의 전화 번호나 배터리 잔량 같은 것은 변수로 추상화시킬 수 있습니다.

아래 그림은 흔히 객체를 나타내기 위한 그림입니다.

An Object



<http://journals.ecs.soton.ac.uk/java/tutorial/java/objects/object.html>에서 인용하였습니다.

이와 같이 어떠한 객체는 자기 만의 정보를 나타내는 변수들과, 이를 가지고 어떠한 작업을 하는 함수들로 둘러싸고 있다고 보시면 됩니다. 참고로, 이러한 객체의 변수나 함수들을 보통 인스턴스 변수(instance variable) 와 인스턴스 메소드(instance method) 라고 부르게 되는데, 그냥 알고 계시는 변수와 함수와 동일한 것으로 생각하시면 됩니다. 누군가 인스턴스 메소드라고 하면 "아 그냥 객체에 정의되어 있는 함수구나" 라고 생각하시면 됩니다.

그림을 메소드가 변수들을 감싸고 있는 것 처럼 그리는 이유는 진짜로 변수들이 외부로 부터 '보호' 되고 있기 때문입니다. 다시 말해, 외부에서 어떠한 객체의 인스턴스 변수의 값을 바꾸지 못하고 오직 객체의 인스턴스 함수를 통해서만 가능하다는 것이지요 (물론 항상 이렇게 극단적으로 불가능 한 것은 아니고 사실 사용자가 조절할 수 있습니다) 이를 단순히 코드로 표현한다면, 예컨대 Animal의 food 를 바꾼다고 할 때

```
Animal animal;
// 초기화 과정 생략

animal.food += 100;           // --> 불가능
animal.increase_food(100);   // --> 가능
```

이렇게 된다는 것입니다. 일단 animal.food += 100; 자체는 외부에서 animal이라는 '객체'의 '인스턴스 변수'에 '직접' 접근하는 것이기 때문에 불가능한 것이고, 아래의 animal.increase_food(100); 의 경우 animal 객체의 '인스턴스 함수'를 통해서 값을 수정하는 것이기 때문에 가능한 것이지요. 이와 같이 외부에서 직접 인스턴스 변수의 값을 바꿀 수 없고 항상 인스턴스 메소드를 통해서 간접적으로 조절하는 것을 캡슐화(Encapsulation) 라고 부릅니다.

이 개념을 처음 들었을 때 이게 왜 필요하냐고 생각하시는 분들이 많습니다. 저도 캡슐화를 굳이 해야될 이유를 못 찾았거든요. 그냥, animal.food += 100; 하나 animal.increase_food(100); 하나 거기서 거기이지라는 생각을 말이죠.

일단 여기서는 캡슐화의 장점에 대해서는 나중에 설명하겠지만 간단하게 말하자면, "객체가 내부적으로 어떻게 작동하는지 몰라도 사용할 줄 알게 된다" 라고 볼 수 있습니다. 예컨대 animal.increase_food(100); 을 하면 내부적으로 food 변수 값이 100 증가하는 것 뿐만 아니라 몸무게도 바뀔

수 있고, 행복도도 올라갈 수 있고 등등 여러가지 작업들이 일어나겠지요. 만일 `increase_food` 함수를 사용하지 않았다면

```
animal.food += 100;
animal.weight += 10;
//... 여러가지 처리
```

여러가지 처리를 프로그래머가 직접 해주어야 합니다. 하지만 이것은 프로그래머가 `food` 를 100 늘리는 과정에서 정확히 어떠한 일들이 일어나는지 알아야지만 가능하다는 것입니다. 이는 상당히 피곤한 작업이겠지요. 더군다나, 대형 프로젝트에서는 객체들을 한 사람이 설계하는 것이 아니기 때문에 다른 사람이 작성한 것을 읽고 완벽히 이해해야만 합니다. 짜증나는 일이겠지요. 하지만 인스턴스 메소드를 이용하면 "food 를 늘리려면 `increase_food` 를 이용하세요~ (나머지는 우리가 다 알아서 할께요)" 라는 것만 알아도 `increase_food(100)` 해버리면, 객체 내부적으로 알아서 처리되기 때문에 이를 사용하는 프로그래머가 굳이 이해하지 않아도 됩니다.

"내부적으로 어떻게 처리되는지는 알 필요가 없다!" 라는 말이 조금 못마땅 하다고 생각하시는 분들이 있을 것입니다. 사람이 그렇게 무책임 해서도 되나 말이죠. 하지만 곰곰히 생각해보면 우리가 접하는 모든 전자 기기들은 캡슐화 되어 있다고 볼 수 있습니다. 노트북의 경우도, 화면에 글자 'a' 를 띠우기 위해서 우리는 컴퓨터 내부에서 어떠한 연산이 처리되는지 알 필요 없습니다. 단순히 우리가 하는 일은 '키보드의 a 를 누른다' 라는, 마치 `my_computer.keyboard_hit('a');` 라는 메소드를 호출하는 것과 동일한 작업이지요. 만일 노트북이 캡슐화 되어 있지 않다면요? 그건 여러분의 상상에 맡기겠습니다.

클래스

자 그러면 객체는 C++ 상에서 어떻게 만들어낼까요. 이를 위해 C++ 에서 객체를 만들 수 있는 장치를 준비하였습니다. 쉽게 말하면 객체의 '설계도' 라고 볼 수 있지요. 바로 클래스(class)입니다.



위와 같이 안의 내용은 차있지 않고 빈 껍질로만 생각할 수 있습니다. 그리고 우리는 이 객체의 설계도를 통해서 실제 객체를 만들게 되지요. C++ 에서 이와 같이 클래스를 이용해서 만들어진 객체를

인스턴스(instance) 라고 부릅니다. 앞서 객체의 변수와 메소드를 왜 인스턴스 변수와 인스턴스 메소드라고 했는지 아시겠죠?

```
#include <iostream>

class Animal {
private:
    int food;
    int weight;

public:
    void set_animal(int _food, int _weight) {
        food = _food;
        weight = _weight;
    }
    void increase_food(int inc) {
        food += inc;
        weight += (inc / 3);
    }
    void view_stat() {
        std::cout << "이 동물의 food : " << food << std::endl;
        std::cout << "이 동물의 weight : " << weight << std::endl;
    }
}; // 세미콜론 잊지 말자!

int main() {
    Animal animal;
    animal.set_animal(100, 50);
    animal.increase_food(30);

    animal.view_stat();
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
이 동물의 food : 130
이 동물의 weight : 60
```

와 같이 나오게 됩니다. 새로운 개념들이 왕창 많이 등장했으니 코드를 천천히 살펴봅시다.

```
Animal animal;
```

먼저 `main` 함수에서 `Animal` 클래스의 인스턴스를 어떻게 생성하였는지 살펴 봅시다. 기존의 구조체에서 구조체 변수를 생성할 때와 동일한데, 구조체의 경우 앞에 `struct` 를 명시 했어야 했지만

여기서는 그러지 않아도 됩니다. 그냥 `int` 나 `char`처럼 `Animal`이라고 써주면 됩니다. 이와 같이 `Animal animal;` 을 했으면 `Animal` 클래스의 인스턴스 `animal`을 만들게 된 것이지요. 이제 본격적으로 클래스가 어떻게 되어 있는지 살펴봅시다.

```
class Animal {
private:
    int food;
    int weight;

public:
    void set_animal(int _food, int _weight) {
        food = _food;
        weight = _weight;
    }
    void increase_food(int inc) {
        food += inc;
        weight += (inc / 3);
    }
    void view_stat() {
        std::cout << "이 동물의 food : " << food << std::endl;
        std::cout << "이 동물의 weight : " << weight << std::endl;
    }
};
```

위는 `Animal`이라는 클래스를 나타낸 것으로 `Animal` 클래스를 통해서 생성될 임의의 객체에 대한 설계도라고 볼 수 있습니다. 즉, `Animal` 클래스를 통해서 생성될 객체는 `food`, `weight`라는 변수가 있고, `set_animal`, `increase_food`, `view_stat`이라는 함수들이 있는데, `Animal` 클래스 상에서 이들을 지칭할 때 각각 멤버 변수(member variable) 과 멤버 함수(member function) 라고 부릅니다.

즉, 인스턴스로 생성된 객체에서는 인스턴스 변수, 인스턴스 함수, 그리고 그냥 클래스 상에서는 멤버 변수, 멤버 함수라고 부르는 것입니다. 멤버 변수와 멤버 함수는 실재 하는 것이 아니지요. 인스턴스가 만들어져야 비로소 세상에 나타나는 것입니다. 즉, 설계도 상에 있다고 해서 아파트가 실제로 존재하는 것이 아닌 것처럼 말이지요.

```
private:
    int food;
    int weight;
```

먼저 멤버 변수들을 정의한 부분을 봅시다. 처음 보는 키워드가 있지요? 이러한 키워드를 '접근 지시자'라고 하는데, 외부에서 이러한 멤버들에 접근을 할 수 있느냐 없느냐를 지시해주는 것입니다. `private` 키워드의 경우, 아래에 쓰여진 것들은 모두 객체 내에서 보호되고 있다라는 의미이지요. 즉, 앞서 객체 그림을 떠올리면 멤버 변수들이 안에서 보호 받고 있던 것 기억하시죠? `private` 되고 있는 모든 것들은 자기 객체 안에서만 접근할 수 있을 뿐 객체 외부에서는 접근할 수 없게 됩니다. 다시 말해

```
void set_animal(int _food, int _weight) {
    food = _food;
    weight = _weight;
}
```

와 같이 같은 객체 안에서 `food` 와 `weight`에 접근하는 것은 가능한 일이지만

```
int main() {
    Animal animal;
    animal.food = 100;
}
```

처럼 객체 밖에서 인위적으로 `food`에 접근하는 것은 불가능 하다는 것입니다. (실제로 컴파일 해보면 오류가 발생합니다) 반면에 `public` 키워드의 경우,

```
public:
void set_animal(int _food, int _weight) {
    food = _food;
    weight = _weight;
}
void increase_food(int inc) {
    food += inc;
    weight += (inc / 3);
}
void view_stat() {
    std::cout << "이 동물의 food : " << food << std::endl;
    std::cout << "이 동물의 weight : " << weight << std::endl;
}
```

이와 같이 멤버 함수들을 `public`으로 지정하였습니다. `public`이라는 것은 말 그대로 공개된 것으로 외부에서 마음껏 이용할 수 있게 됩니다. 그래서 `main` 함수에서도 이들을

```
animal.set_animal(100, 50);
animal.increase_food(30);
animal.view_stat();
```

처럼 마음껏 접근할 수 있었습니다. 만일 멤버 함수들을 `private`로 설정해버렸다면 어떨까요. `public` 키워드를 지워봅시다.

그냥 컴파일 해보면

컴파일 오류

```
'Animal::set_animal' : cannot access private member declared in
→ class 'Animal'
```

위와 같은 오류가 3 개 정도 등장하게 됩니다. 다시 말해, `Animal` 의 `private` 멤버 함수에 접근할 수 없다는 의미겠지요. 결과적으로 외부에서 접근을 할 수 없는 객체는 그냥 아무짝에도 쓸모 없는 덩어리로 남게 됩니다.

참고로 키워드 명시를 하지 않았다면 기본적으로 `private` 로 설정됩니다. 즉, 맨 위의 `private` 키워드를 지워도 상관이 없다는 것이지요. 그냥 `private` 없이

```
class Animal {
    int food;
    int weight;
    // ... 생략
```

이렇게 해도 `food` 와 `weight` 는 알아서 `private` 로 설정 됩니다.

만일 멤버 변수들도 `public` 으로 공개해버리면 어떨까요. 그러면 `main` 함수에서 마치 예전에 구조체를 사용했던 것처럼

```
animal.food = 100;
```

로 손쉽게 접근할 수 있게 됩니다. 이제 멤버 변수에 대해 조금 더 자세히 살펴 보도록 합시다.

```
void set_animal(int _food, int _weight) {
    food = _food;
    weight = _weight;
}
```

위는 각 멤버 변수들의 값을 설정하는 부분인데요, 여기서 `food` 와 `weight` 는 누구의 것일까요? 당연하게도, 객체 자신의 것입니다. 그렇기 때문에 `food` 와 `weight` 가 누구 것인지 명시할 필요 없이 그냥 `food`, `weight` 라고 사용하면 됩니다. `set_animal` 을 호출한 객체의 `food` 와 `weight` 값이기 때문이지요. 마찬가지로 `increase_food` 를 살펴보면

```
void increase_food(int inc) {
    food += inc;
    weight += (inc / 3);
}
```

이와 같이 얼마나 food 를 증가시킬 지 입력 받은 다음에 내부적으로 food 와 weight 를 모두 처리해주게 됩니다.

이번 강좌는 여기서 마치도록 하겠습니다. 한 가지 꼭 기억하실 점은, 객체가 무엇인지, 그리고 클래스가 무엇인지 꼭 명심해 두시기 바랍니다. 또 앞으로 계속 나올 인스턴스, 인스턴스 변수, 인스턴스 함수, 멤버 변수, 멤버 함수 와 같은 용어들을 잘 파악하고 있어야지 뒤에가서 헷갈리지 않겠죠.

생각 해볼 문제

문제 1

여러분은 아래와 같은 Date 클래스를 디자인 하려고 합니다. SetDate 는 말그대로 Date 함수 내부를 초기화 하는 것이고 AddDay, AddMonth, AddYear 는 일, 월, 년을 원하는 만큼 더하게 됩니다. 한 가지 주의할 점은 만일 2012 년 2 월 28 일에 3 일을 더하면 2012 년 2 월 31 일이 되는 것이 아니라 2012 년 3 월 2 일이 되겠지요? (난이도 : 상)

```
class Date {  
    int year_;  
    int month_; // 1 부터 12 까지.  
    int day_; // 1 부터 31 까지.  
  
    public:  
        void SetDate(int year, int month, int date);  
        void AddDay(int inc);  
        void AddMonth(int inc);  
        void AddYear(int inc);  
  
        void ShowDate();  
};
```

생성자와 함수의 오버로딩

안녕하세요 여러분. 이제 본격적으로 객체 지향 프로그래밍을 시작 하도록 하겠습니다. 아마도 지난 번 생각해보기를 열심히 하셨던 분들이라면, `Date` 클래스에서 `add_day` 함수는 조금 어렵더라도, `add_month` 나 `add_year` 정도는 가뿐하게 만드셨을 것이라고 생각합니다. 사실, `add_day` 도 `add_month` 함수를 응용해서 만들면 쉬웠을 것입니다. 이번 강좌에서는 저와 함께 다시 새로운 방법으로 `Date` 클래스를 만들어가면서 객체 지향 프로그래밍에 조금 더 친숙해져 보도록 하겠습니다.

함수의 오버로딩 (Overloading)

본격적으로 객체 지향 프로그래밍을 시작하기에 앞서 C++ 에 C 와는 다른 새로운 기능을 잠시 살펴 보도록 하겠습니다. 바로 '함수의 오버로딩' 이라고 하는 것인데요, 사실 오버로드를 사전에서 찾아보면 다음과 같은 뜻이 나옵니다.

1. 과적하다
2. sb (with sth) 너무 많이 주다[부과하다]
3. (컴퓨터·전기 시스템 등에) 과부하가 걸리게 하다

음.. 그렇다면 함수의 오버로딩이라는 것은 '함수에 과부하를 주는 것' 인가라는 생각도 드실 텐데요, 사실 맞는 말씀입니다. 사실 C 언어에서는 하나의 이름을 가지는 함수는 딱 1 개만 존재할 수 밖에 없기에 과부하라는 말 자체가 성립이 안됬지요.

`printf` 는 C 라이브러리에 단 한 개 존재하고, `scanf` 도 C 라이브러리에 단 1 개만 존재합니다. 하지만 C++ 에서는 같은 이름을 가진 함수가 여러개 존재해도 됩니다. 즉, 함수의 이름에 과부하가 걸려도 상관이 없다는 것이지요!

그렇다면 도대체 C++ 에서는 같은 이름의 함수를 호출했을 때 구분을 어떻게 하는 것일까요. 물론 단순합니다. 함수를 호출 하였을 때 사용하는 인자를 보고 결정하게 됩니다.

```
/* 함수의 오버로딩 */
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
void print(char x) { std::cout << "char : " << x << std::endl; }
void print(double x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1;
    char b = 'c';
    double c = 3.2f;
```

```

print(a);
print(b);
print(c);

return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

int : 1
char : c
double : 3.2

```

일단 위 소스를 보게 된다면 이름이 `print` 인 함수 3 개가 정의가 되었음을 알 수 있습니다. 고전적인 C 컴파일러에서는 오류가 발생했겠지만 C++에서는 함수의 이름이 같더라도 인자가 다르면 다른 함수 라고 판단하기 때문에 오류가 발생하지 않는 것입니다.

```

void print(int x);
void print(char x);
void print(double x);

```

위와 같이 정의된 함수들을 `main`에서 아래와 같이 호출하게 됩니다.

```

int a = 1;
char b = 'c';
double c = 3.2f;

print(a);
print(b);
print(c);

```

여기서 한 가지 눈여겨 보아야 할 점은 `a`는 `int`, `b`는 `char`, `c`는 `double` 타입이라는 것인데, 이에 따라 각각의 타입에 맞는 함수들, 예를 들어 `print(b)`는 `b`가 `char` 이므로 `char` 형의 인자를 가지는 두 번째 `print` 가 호출 된 것입니다.

C 언어였을 경우 `int`, `char`, `double` 타입에 따라 함수의 이름을 제각각 다르게 만들어서 호출해 주어야 했던 반면에 C++에서는 컴파일러가 알아서 적합한 인자를 가지는 함수를 찾아서 호출해 주게 됩니다.

```
/* 함수의 오버로딩 */
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
void print(double x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1;
    char b = 'c';
    double c = 3.2f;

    print(a);
    print(b);
    print(c);

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
int : 1
int : 99
double : 3.2
```

이번에는 조금 특이한 경우입니다. 일단 함수는

```
void print(int x)
void print(double x)
```

`int` 타입의 인자나 `double` 타입의 인자를 하나 받는 함수 하나 밖에 없습니다. 하지만 `main`에서 각기 다른 타입의 인자들 (`int`, `char`, `double`)로 `print` 함수를 호출하게 됩니다. 물론 `a`나 `c`의 경우 각자 자기를 인자로 하는 정확한 함수들이 있어서 성공적으로 호출 될 수 있겠지만, `char`의 경우 자기와 정확히 일치하는 인자를 가지는 함수가 없기 때문에 '자신과 최대로 근접한 함수'를 찾게 됩니다.

C++ 컴파일러에서 함수를 오버로딩하는 과정은 다음과 같습니다.

1 단계

자신과 타입이 정확히 일치하는 함수를 찾는다.

2 단계

정확히 일치하는 타입이 없는 경우 아래와 같은 형변환을 통해서 일치하는 함수를 찾아본다.

- Char, unsigned char, short 는 int 로 변환된다.
- Unsigned short 는 int 의 크기에 따라 int 혹은 unsigned int 로 변환된다.
- Float 은 double 로 변환된다.
- Enum 은 int 로 변환된다.

3 단계

위와 같이 변환해도 일치하는 것이 없다면 아래의 좀 더 포괄적인 형변환을 통해 일치하는 함수를 찾는다.

- 임의의 숫자(numeric) 타입은 다른 숫자 타입으로 변환된다. (예를 들어 float -> int)
- Enum 도 임의의 숫자 타입으로 변환된다 (예를 들어 Enum -> double)
- 0 은 포인터 타입이나 숫자 타입으로 변환된 0 은 포인터 타입이나 숫자 타입으로 변환된다
- 포인터는 void 포인터로 변환된다.

4 단계

유저 정의된 타입 변환으로 일치하는 것을 찾는다 (이 부분에 대해선 나중에 설명!)(출처)

만약에 컴파일러가 위 과정을 통하여라도 일치하는 함수를 찾을 수 없거나 같은 단계에서 두 개 이상이 일치하는 경우에 모호하다 (ambiguous) 라고 판단해서 오류를 발생하게 됩니다.

그렇다면 우리의 소스 코드에서

```
print(b);
```

는 어떻게 될까요. 1 단계에서는 명백하게도 char 타입의 인자를 가진 print 가 없기에 2 단계로 넘어오게 됩니다. 그런데 2 단계에서는 char 이 int 로 변환된다면 print (int x) 를 호출할 수 있기 때문에 결국 print (int x) 가 호출되게 되는 것이지요.

```
// 모호한 오버로딩
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
```

```

void print(char x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1;
    char b = 'c';
    double c = 3.2f;

    print(a);
    print(b);
    print(c);

    return 0;
}

```

위 소스를 컴파일 하였다면 오류가 발생함을 알 수 있습니다. 오류를 살짝 보자면

컴파일 오류

```

error C2668: 'print' : ambiguous call to overloaded function
could be 'void print(char)'
or          'void print(int)'
while trying to match the argument list '(double)'

```

와 같이 나오는데요, 왜 오류가 발생하였는지 살펴보도록 합시다. 일단 위 소스에서는 함수가 `print(int x)` 와 `print (char x)` 밖에 없으므로 관건은 `print(c);` 를 했을 때 어떠한 함수가 호출되어야 하는지 결정하는 것인데요, `print(c)` 를 했을 때 1 단계에서는 명백하게 일치하는 것이 없습니다.

2 단계에서는 마찬가지로 `double` 의 캐스팅에 관련한 내용이 없기에 일치하는 것이 없고 비로소 3 단계로 넘어오게 됩니다. 3 단계에서는 '임의의 숫자 타입이 임의의 숫자 타입'으로 변환되서 생각되기 때문에 `double` 은 `char` 도, `int` 도 변환 될 수 있게 되는 것입니다.

따라서 같은 단계에 두 개 이상의 가능한 일치가 존재하므로 오류가 발생하게 되는 것이지요.

위와 같은 C++ 오버로딩 규칙을 머리속에 숙지 하는 일은 매우 중요한 일입니다. 왜냐하면 나중에 복잡한 함수를 오버로딩할 때 여러가개 중복되서 나온다면 눈물없이 볼 수 없는 오류의 향연을 만날 수 있을 것입니다!

Date 클래스

```

#include<iostream>

class Date {

```

```
int year_;
int month_; // 1 부터 12 까지.
int day_; // 1 부터 31 까지.

public:
void SetDate(int year, int month, int date);
void AddDay(int inc);
void AddMonth(int inc);
void AddYear(int inc);

// 해당 월의 총 일 수를 구한다.
int GetCurrentMonthTotalDays(int year, int month);

void ShowDate();
};

void Date::SetDate(int year, int month, int day) {
    year_ = year;
    month_ = month;
    day_ = day;
}

int Date::GetCurrentMonthTotalDays(int year, int month) {
    static int month_day[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if (month != 2) {
        return month_day[month - 1];
    } else if (year % 4 == 0 && year % 100 != 0) {
        return 29; // 윤년
    } else {
        return 28;
    }
}

void Date::AddDay(int inc) {
    while (true) {
        // 현재 달의 총 일 수
        int current_month_total_days = GetCurrentMonthTotalDays(year_, month_);

        // 같은 달 안에 들어온다면;
        if (day_ + inc <= current_month_total_days) {
            day_ += inc;
            return;
        } else {
            // 다음달로 넘어가야 한다.
            inc -= (current_month_total_days - day_ + 1);
            day_ = 1;
            AddMonth(1);
        }
    }
}
```

```
void Date::AddMonth(int inc) {
    AddYear((inc + month_ - 1) / 12);
    month_ = month_ + inc % 12;
    month_ = (month_ == 12 ? 12 : month_ % 12);
}

void Date::AddYear(int inc) { year_ += inc; }

void Date::ShowDate() {
    std::cout << "오늘은 " << year_ << " 년 " << month_ << " 월 " << day_
        << " 일입니다 " << std::endl;
}

int main() {
    Date day;
    day.SetDate(2011, 3, 1);
    day.ShowDate();

    day.AddDay(30);
    day.ShowDate();

    day.AddDay(2000);
    day.ShowDate();

    day.SetDate(2012, 1, 31); // 윤년
    day.AddDay(29);
    day.ShowDate();

    day.SetDate(2012, 8, 4);
    day.AddDay(2500);
    day.ShowDate();
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
오늘은 2011 년 3 월 1 일입니다
오늘은 2011 년 3 월 31 일입니다
오늘은 2016 년 9 월 20 일입니다
오늘은 2012 년 2 월 29 일입니다
오늘은 2019 년 6 월 9 일입니다
```

위의 코드는 간단히 만들어본 Date 클래스입니다. 그런데, 이상한 것이 있죠? 클래스 내부에 아래 코드와 같이

```

void SetDate(int year, int month, int date);
void AddDay(int inc);
void AddMonth(int inc);
void AddYear(int inc);

// 해당 월의 총 일 수를 구한다.
int GetCurrentMonthTotalDays(int year, int month);

void ShowDate();

```

함수의 정의만 나와 있고, 함수 전체 몸통은

```

void Date::ShowDate() {
    std::cout << "오늘은 " << year_ << " 년 " << month_ << " 월 " << day_
        << " 일입니다 " << std::endl;
}

```

처음 밖에 나와 있습니다. `Date::` 을 함수 이름 앞에 붙여주게 되면 이 함수가 "Date 클래스의 정의된 함수" 라는 의미를 부여하게 됩니다. 만일 그냥

```
void ShowDate() { // ... }
```

와 같이 작성하였다면 위 함수는 클래스의 멤버 함수가 아니라 그냥 일반적인 함수가 됩니다. 보통 간단한 함수를 제외하면 대부분의 함수들은 클래스 바깥에서 위와 같이 정의하게 됩니다. 왜냐하면 클래스 내부에 쓸 경우 클래스 크기가 너무 길어져서 보기 좋지 않기 때문이죠.²⁾

```

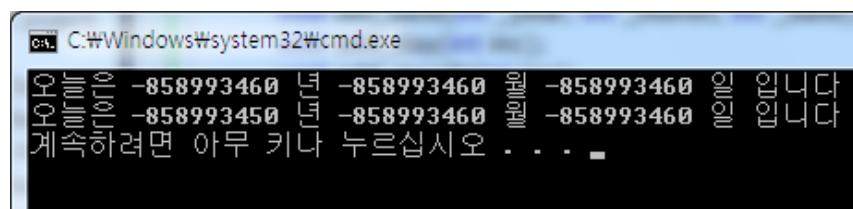
Date day;
day.SetDate(2011, 3, 1);
day.ShowDate();

day.AddDay(30);
day.ShowDate();

```

그럼 이제 `main` 함수를 살펴 봅시다. 위처럼 `day` 인스턴스를 생성해서 `SetDate`로 초기화 한 다음에 `ShowDate`로 내용을 한 번 보여주고, 또 `AddDay`을 해서 30일을 증가 시킨뒤 다시 새로운 날짜를 출력하도록 하였습니다. 여기서 가장 중요한 부분은 무엇일까요? 당연하게도, 처음의 `SetDate` 부분입니다. 만일 `SetDate`를 하지 않았더라면 초기화 되지 않은 값들에 덧셈 과 출력 명령이 내려져서

2) 다만 예외적으로 나중에 배울 템플릿 클래스의 경우 모두 클래스 내부에 작성하게 됩니다.



위 처럼 이상한 쓰레기 값이 출력되게 되거든요. 그런데 문제는 이렇게 `SetDate` 함수를 사람들이 꼭 뒤에 써주지 않는다는 말입니다. 물론 훌륭한 프로그래머들은 생성 후 초기화를 항상 숙지하고 있겠지만 간혹 실수로 생성한 객체를 초기화하는 과정을 빠트린다면 끔찍한 일이 벌어지게 됩니다. 다행으로 C++에서는 이를 언어 차원에서 도와주는 장치가 있는데 바로 생성자(constructor)입니다.

생성자(Constructor)

```

#include <iostream>

class Date {
    int year_;
    int month_; // 1 부터 12 까지.
    int day_; // 1 부터 31 까지.

public:
    void SetDate(int year, int month, int date);
    void AddDay(int inc);
    void AddMonth(int inc);
    void AddYear(int inc);

    // 해당 월의 총 일 수를 구한다.
    int GetCurrentMonthTotalDays(int year, int month);

    void ShowDate();

    Date(int year, int month, int day) {
        year_ = year;
        month_ = month;
        day_ = day;
    }
};

// 생략

void Date::AddYear(int inc) { year_ += inc; }

void Date::ShowDate() {
    std::cout << "오늘은 " << year_ << " 년 " << month_ << " 월 " << day_
}

```

```

        << " 일입니다 " << std::endl;
}

int main() {
    Date day(2011, 3, 1);
    day.ShowDate();

    day.AddYear(10);
    day.ShowDate();

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

오늘은 2011 년 3 월 1 일입니다

오늘은 2021 년 3 월 1 일입니다

위와 같이 초기화가 잘 되서 출력됨을 알 수 있습니다.

생성자는 기본적으로 "객체 생성시 자동으로 호출되는 함수" 라고 볼 수 있습니다. 이 때 자동으로 호출 되면서 객체를 초기화 해주는 역할을 담당하게 됩니다. 생성자는 아래와 같이 정의합니다.

```
// 객체를 초기화 하는 역할을 하기 때문에 리턴값이 없다!
/* 클래스 이름 */ /* 인자 */ {}
```

예를 들어서 위 경우 저는 아래와 같이 Date 의 생성자를 정의하였습니다.

```
Date(int year, int month, int day)
```

이렇게 정의가 된 생성자는 객체를 생성할 때 다음과 같이 위 함수에서 정의한 인자에 맞게마치 함수를 호출하듯이 써준다면 위 생성자를 호출하여 객체를 생성할 수 있게 됩니다. 즉, 우리의 경우 아래와 같이 객체를 생성하였지요.

```
Date day(2011, 3, 1);
```

이는 곧 **Date** 클래스의 **day** 객체를 만들면서 생성자 **Date(int year, int month, int day)** 를 호출한다 라는 의미가 됩니다. 따라서 Date 의 객체를 생성할 때 생성자의 인자 year, month, day 에 각각 2011, 3, 1 을 전달하여 객체를 생성하게 되는 것이지요. 참고로

```
Date day = Date(2012, 3, 1);
```

위 문장 역시 생성자 `Date(2012, 3, 1)` 을 호출해서 이를 토대로 객체를 생성하라는 의미입니다. 각각의 방식에 대해 이름이 붙어 있는데,

```
Date day(2011, 3, 1);           // 암시적 방법 (implicit)
Date day = Date(2012, 3, 1);   // 명시적 방법 (explicit)
```

마치 함수를 호출하듯이 사용하는 것이 암시적 방법, 명시적으로 생성자를 호출한다는 것을 보여주는 것이 명시적 방법인데 많은 경우 암시적 방법으로 축약해서 쓸 수 있으므로 이를 선호하는 편입니다.

디폴트 생성자 (Default constructor)

그런데 한 가지 궁금증이 생겼습니다. 맨 처음에 단순히 `SetDate` 함수를 이용해서 객체를 초기화하였을 때 우리는 생성자를 명시하지 않았습니다. 즉 처음에 생성자 정의를 하지 않은 채 (`SetDate` 함수를 사용했던 코드)

```
Date day;
```

로 했을 때 과연 생성자가 호출 될까요? 답은 Yes 입니다. 생성자가 호출됩니다. 그런데, 우리가 생성자를 정의하지도 않았는데 어떤 생성자가 호출이 될까요? 바로 디폴트 생성자(**Default Constructor**)입니다. 디폴트 생성자는 인자를 하나도 가지지 않는 생성자인데, 클래스에서 사용자가 어떠한 생성자도 명시적으로 정의하지 않았을 경우에 컴파일러가 자동으로 추가해주는 생성자입니다.³⁾ 물론 컴파일러가 자동으로 생성할 때에는 아무런 일도 하지 않게 되지요. 그렇기에 맨 처음에 `SetDate` 를 하지 않았을 때 쓰레기 값이 나왔던 것입니다.

물론 여러분이 직접 디폴트 생성자를 정의할 수도 있습니다. 아래와 같아요.

```
// 디폴트 생성자 정의해보기
#include <iostream>

class Date {
    int year_;
    int month_; // 1 부터 12 까지.
    int day_;   // 1 부터 31 까지.

public:
    void ShowDate();
```

3) 사용자가 어떤 다른 생성자를 추가한 순간 컴파일러는 자동으로 디폴트 생성자를 삽입하지 않는다는 것을 명심하세요!

```

Date() {
    year_ = 2012;
    month_ = 7;
    day_ = 12;
}
};

void Date::ShowDate() {
    std::cout << "오늘은 " << year_ << " 년 " << month_ << " 월 " << day_
        << " 일입니다 " << std::endl;
}

int main() {
    Date day = Date();
    Date day2;

    day.ShowDate();
    day2.ShowDate();

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

오늘은 2012 년 7 월 12 일입니다

오늘은 2012 년 7 월 12 일입니다

와 같이 나오게 됩니다. 여러분은 아래와 같이 디폴트 생성자 Date() 를 정의하였습니다.

```

Date() {
    year = 2012;
    month = 7;
    day = 12;
}

```

즉 year 에는 2012, month 에는 7, day 에는 2 를 대입합니다.

```

Date day = Date();
Date day2;

```

그래서 사용하게 되면 위와 같이 디폴트 생성자를 이용해서 day 와 day2 를 추가할 수 있게 되는 것입니다. 한 가지 주의할 점은 위에서 인자가 있는 생성자에서 적용했던 것처럼

```
Date day3();
```

와 하면 `day3` 객체를 디폴트 생성자를 이용해서 초기화 하는 것이 아니라, 리턴값이 `Date`이고 인자가 없는 함수 `day3` 을 정의하게 된 것으로 인식합니다. 이는 암시적 표현으로 객체를 선언할 때 반드시 주의해 두어야 할 사항입니다.

절대로 인자가 없는 생성자를 호출하기 위해서 `A a()` 처럼 하면 안됩니다. 해당 문장은 `A` 를 리턴하는 함수 `a` 를 정의한 문장입니다. 반드시 그냥 `A a` 와 같이 써야 합니다.

명시적으로 디폴트 생성자 사용하기

C++ 11 이전에는 디폴트 생성자를 사용하고 싶을 경우 그냥 생성자를 정의하지 않는 방법 밖에 없었습니다. 하지만 이 때문에 그 코드를 읽는 사용자 입장에서 개발자가 깜빡 잊고 생성자를 정의를 안한 것인지, 아니면 정말 디폴트 생성자를 사용하고파서 이런 것인지 알길이 없겠죠.

다행이도 C++ 11 부터 명시적으로 디폴트 생성자를 사용하도록 명시할 수 있습니다.

```
class Test {
public:
    Test() = default; // 디폴트 생성자를 정의해라
};
```

바로 위처럼 생성자의 선언 바로 뒤에 `= default` 를 붙여준다면, `Test` 의 디폴트 생성자를 정의하라고 컴파일러에게 명시적으로 알려줄 수 있습니다.

생성자 오버로딩

앞서 함수의 오버로딩에 대해 잠깐 짚고 넘어갔는데, 생성자 역시 함수 이기 때문에 마찬가지로 함수의 오버로딩이 적용될 수 있습니다. 쉽게 말해 해당 클래스의 객체를 여러가지 방식으로 생성할 수 있게 되겠지요.

```
#include <iostream>

class Date {
    int year_;
    int month_; // 1 부터 12 까지.
    int day_; // 1 부터 31 까지.

public:
    void ShowDate();
```

```

Date() {
    std::cout << "기본 생성자 호출!" << std::endl;
    year_ = 2012;
    month_ = 7;
    day_ = 12;
}

Date(int year, int month, int day) {
    std::cout << "인자 3 개인 생성자 호출!" << std::endl;
    year_ = year;
    month_ = month;
    day_ = day;
}
};

void Date::ShowDate() {
    std::cout << "오늘은 " << year_ << " 년 " << month_ << " 월 "
        << " 일입니다 " << std::endl;
}

int main() {
    Date day = Date();
    Date day2(2012, 10, 31);

    day.ShowDate();
    day2.ShowDate();

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

기본 생성자 호출!
인자 3 개인 생성자 호출!
오늘은 2012 년 7 월 12 일입니다
오늘은 2012 년 10 월 31 일입니다

```

와 같이 적절히 오버로딩이 되서 사용자가 원하는 생성자를 호출할 수 있게 됩니다.

이것으로 생성자에 대해 간단히 설명을 마치겠습니다. 물론 아직 생성자에 대해 이야기 할 거리는 무궁무진하게 남아 있지만 일단 오늘은 함수의 오버로딩과 생성자에 대한 입문으로 충분히 머리가 아플 테니 생각해 보기로 머리를 식히도록 합시다!

생각 해보기

문제 1

Date 클래스에 여러가지 생성자들을 추가해보세요 (난이도 : 下)

문제 2

수학 관련 소프트웨어를 만드는 회사에서 의뢰가 들어왔습니다. 중학생용 기하학 소프트웨어를 만드는 것인데요, 클래스는 총 두 개로 하나는 Point로 점에 관한 정보를 담는 것이고 다른 하나는 Geometry로 점들을 가지고 연산을 하는 클래스입니다. 즉 아래와 같은 두 클래스의 함수들을 모두 정의하세요 (난이도 : 上)

```
class Point {
    int x, y;

public:
    Point(int pos_x, int pos_y);
};

class Geometry {
    // 점 100 개를 보관하는 배열.
    Point* point_array[100];

public:
    Geometry(Point **point_list);
    Geometry();

    void AddPoint(const Point &point);

    // 모든 점들 간의 거리를 출력하는 함수입니다.
    void PrintDistance();

    // 모든 점들을 잇는 직선들 간의 교점의 수를 출력해주는 함수입니다.
    // 참고적으로 임의의 두 점을 잇는 직선의 방정식을  $f(x, y) = ax+by+c = 0$ 
    // 이라고 할 때 임의의 다른 두 점  $(x_1, y_1)$  과  $(x_2, y_2)$  가  $f(x, y)=0$  을 기준으로
    // 서로 다른 부분에 있을 조건은  $f(x_1, y_1) * f(x_2, y_2) \leq 0$  이면 됩니다.
    void PrintNumMeets();
};
```

복사 생성자와 소멸자

안녕하세요 여러분. 지난 강좌에서 생성자에 대해 처음 알게 된 이후로 생성자의 위력에 대해 감탄하지 않으셨나요? 생성자를 통해 이전에 C 프로그래밍 시에 변수 초기화를 하지 않아서 생겼던 수많은 오류들을 효과적으로 없앨 수 있었습니다.

뿐만 아니라 C++ 에서 새로 도입된 함수 오버로딩 덕분에 함수 이름을 일일히 따로 지정하지 않더라도 성공적으로 인자들의 타입에 따라 원하는 함수들만 호출 할 수 있게 되었습니다. 실제로 C 언어 였다면 인자의 타입에 따라서 함수의 이름들을 외워야 했지만 C++ 에서는 그럴 필요가 전혀 없게 되었다는 것이지요.

스타크래프트 만들기



사실 제가 오늘 이 강좌에서 진행하고 싶었던 것은 바로 '스타크래프트 만들기' 이었습니다. 아니, 스타크래프트를 만들겠다니요. 그게 말이 됩니까? 네. 말이 됩니다. 저는 앞으로 C++ 강좌를 진행해나가면서 '스타크래프트'의 뼈대를 차근 차근 만들어 나가보고자 합니다. 그럼, 여러분 모두 준비 되셨나요?



스타크래프트라는 거대한 프로젝트를 진행하기에 앞서서 일단, 유닛 하나 부터 만들어 보도록 할 것입니다. 위에 조그만 사진에 있는 총들고 서 있는 사람은 스타크래프트의 마린이라는 유닛입니다. (테란 유저로써 제가 가장 사랑하는 유닛 중 하나라고 볼 수 있죠) 위 유닛은 스타크래프트 유닛 중에서 가장 단순하고 기본이 되는 유닛이라고 할 수 있습니다. 그렇다면 한 번, 이 마린을 코드상에서 구현해보도록 합시다.

```
#include <iostream>

class Marine {
    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    int damage; // 공격력
    bool is_dead;
```

```
public:  
Marine(); // 기본 생성자  
Marine(int x, int y); // x, y 좌표에 마린 생성  
  
int attack(); // 데미지를 리턴한다.  
void be_attacked(int damage_earn); // 입는 데미지  
void move(int x, int y); // 새로운 위치  
  
void show_status(); // 상태를 보여준다.  
};  
Marine::Marine() {  
hp = 50;  
coord_x = coord_y = 0;  
damage = 5;  
is_dead = false;  
}  
Marine::Marine(int x, int y) {  
coord_x = x;  
coord_y = y;  
hp = 50;  
damage = 5;  
is_dead = false;  
}  
void Marine::move(int x, int y) {  
coord_x = x;  
coord_y = y;  
}  
int Marine::attack() { return damage; }  
void Marine::be_attacked(int damage_earn) {  
hp -= damage_earn;  
if (hp <= 0) is_dead = true;  
}  
void Marine::show_status() {  
std::cout << " *** Marine *** " << std::endl;  
std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) " << std::endl;  
std::cout << " HP : " << hp << std::endl;  
}  
  
int main() {  
Marine marine1(2, 3);  
Marine marine2(3, 5);  
  
marine1.show_status();  
marine2.show_status();  
  
std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;  
marine2.be_attacked(marine1.attack());  
  
marine1.show_status();
```

```
    marine2.show_status();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 50
```

마린 1 이 마린 2 를 공격!

```
*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 45
```

어때요? 우리는 일단 위 소스코드에서 아주 초보적으로 작동하는 마린을 구현하였습니다. 한번 살펴볼까요?

```
class Marine {
    int hp;                      // 마린 체력
    int coord_x, coord_y;        // 마린 위치
    int damage;                  // 공격력
    bool is_dead;

public:
    Marine();                   // 기본 생성자
    Marine(int x, int y);      // x, y 좌표에 마린 생성

    int attack();                // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y);     // 새로운 위치

    void show_status();          // 상태를 보여준다.
};
```

위는 마린을 구현한 클래스입니다. 즉, 위 클래스의 객체들이 바로 개개의 마린들이 되는 것이지요. 이전 강좌에서도 이야기 하였지만, 보통 어떠한 객체의 내부적 성질, 상태 등에 관련된 변수들은 모두 **private** 범주에 두고, 그 객체가 외부에 하는 행동들은 함수로써 구현하여 **public**에 두면 된다고 하였습니다.

그렇다면, 마린의 경우, 마린의 상태에 관련된 것들 - 예를 들어서, 마린의 현재 **hp** 라던지, 위치, 공격력, 그리고 생존 여부 등은 **private** 범주에 두어서 관리하고, 마린이 하는 행동들 - 즉, 이동한다던지 공격한다던지, 혹은 외부로 부터 공격 받는 등에 관련된 것들은 메소드로 만들어서 **public**에서 범주로 두면 좋을 것 같습니다.

따라서 위와 같이 코드를 구성하였습니다. 사실 나머지 함수들은 그 구현이 너무 간단해서 굳이 따로 집어서 살펴볼 필요는 없을 것 같습니다. 그래서 바로 **main** 함수의 코드들을 살펴보도록 합시다.

```
Marine marine1(2, 3);
Marine marine2(3, 5);
```

먼저, 위 두개의 **marine1**과 **marine2**라는 이름의 **Marine** 객체들을 생성하였습니다. 물론 생성자 오버로딩에 의해 각각 (2,3), (3,5)에 위치한 마린들이 생성되었지요.

```
marine1.show_status();
marine2.show_status();
```

이제 위 함수들을 통해서 각각의 마린의 상태를 출력한 뒤에,

```
std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
marine2.be_attacked(marine1.attack());
```

마린 2 가 마린 1 로 부터 공격을 받는 상황을 그렸습니다. 어때요? 정말 단순한 코드이지요?

그런데 사실 위 코드에는 약간의 문제가 있습니다 (스타에서의 진짜 마린의 비해 너무 빈약한거 아니냐?? 라는 지적 말고) 만약에 실제 게임에서 처럼 수십 마리의 마린들이 서로 둉여켜 싸우기라도 하면 어떨까요.

그럴 때는 **marine1**, **marine2** 와 같이 일일히 이름 붙이기도 벅찰 뿐더러, 사용자가 몇 개의 마린을 만들겠다라고 컴파일 시점에 정해버리는 것도 아니기 때문에 수십개의 **marine1**, **marine2**...를 미리 만들 수도 없는 격입니다. 그럼 어떡할까요? 답은 단순합니다. **marine** 들을 배열로 정해버리면 되지요.

```
/* int main 전 까지 내용은 동일 */
int main() {
    Marine* marines[100];
```

```

marines[0] = new Marine(2, 3);
marines[1] = new Marine(3, 5);

marines[0]->show_status();
marines[1]->show_status();

std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;

marines[0]->be_attacked(marines[1]->attack());

marines[0]->show_status();
marines[1]->show_status();

delete marines[0];
delete marines[1];
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 50

마린 1 이 마린 2 를 공격!
*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 45

```

로 동일하게 나옵니다.

예전에, `new` 와 `delete` 에 대해서 배울 때 `malloc` 과의 차이점에 대해서 잠깐 언급 했던 것이 기억 나나요? 그 때는 아직 내용을 다 배우지 못해서, `new` 와 `malloc` 모두 동적으로 할당하지만 '무언가' 다르다고 했었는데, 위 코드에서 여러분들은 아마 눈치 채셨을 것이라 생각됩니다. 바로 `new` 의 경우 객체를 동적으로 생성하면서와 동시에 자동으로 생성자도 호출해준다는 점입니다.

```
marines[0] = new Marine(2, 3);
marines[1] = new Marine(3, 5);
```

위와 같이 `Marine(2,3)` 과 `Marine(3,5)` 라는 생성자를 자동으로 호출해주세요. 이것이 바로 C++에 맞는 새로운 동적 할당이라고 볼 수 있습니다.

```
marines[0]->show_status();
marines[1]->show_status();
```

물론 `Marine`들의 포인터를 가리키는 배열이기 때문에 메소드를 호출할 때 . 이 아니라 `->` 를 사용해줘야 되겠지요. 마지막으로, 동적으로 할당한 메모리는 언제나 해제해 주어야 된다는 원칙에 따라

```
delete marines[0];
delete marines[1];
```

를 해주어야 하겠지요.

소멸자 (Destructor)



알고 보니 각각의 마린에도 이름을 지정할 수 있었습니다. 그래서, 우리는 만들어놓은 `Marine` 클래스에 `name`이라는 이름을 저장할 수 있는 또 다른 인스턴스 변수를 추가하도록 합시다.

```
// 마린의 이름 만들기
#include <string.h>

#include <iostream>

class Marine {
    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    int damage; // 공격력
    bool is_dead;
```

```
char* name; // 마린 이름

public:
Marine(); // 기본 생성자
Marine(int x, int y, const char* marine_name); // 이름까지 지정
Marine(int x, int y); // x, y 좌표에 마린 생성

int attack(); // 데미지를 리턴한다.
void be_attacked(int damage_earn); // 입는 데미지
void move(int x, int y); // 새로운 위치

void show_status(); // 상태를 보여준다.
};

Marine::Marine() {
    hp = 50;
    coord_x = coord_y = 0;
    damage = 5;
    is_dead = false;
    name = NULL;
}

Marine::Marine(int x, int y, const char* marine_name) {
    name = new char[strlen(marine_name) + 1];
    strcpy(name, marine_name);

    coord_x = x;
    coord_y = y;
    hp = 50;
    damage = 5;
    is_dead = false;
}

Marine::Marine(int x, int y) {
    coord_x = x;
    coord_y = y;
    hp = 50;
    damage = 5;
    is_dead = false;
    name = NULL;
}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return damage; }
void Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;
}

void Marine::show_status() {
    std::cout << " *** Marine : " << name << " ***" << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
```

```
    std::cout << " HP : " << hp << std::endl;
}

int main() {
    Marine* marines[100];

    marines[0] = new Marine(2, 3, "Marine 2");
    marines[1] = new Marine(1, 5, "Marine 1");

    marines[0]->show_status();
    marines[1]->show_status();

    std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;

    marines[0]->be_attacked(marines[1]->attack());

    marines[0]->show_status();
    marines[1]->show_status();

    delete marines[0];
    delete marines[1];
}
```

성공적으로 컴파일 하였다면

실행 결과

```
*** Marine : Marine 2 ***
Location : ( 2 , 3 )
HP : 50
*** Marine : Marine 1 ***
Location : ( 1 , 5 )
HP : 50

마린 1 이 마린 2 를 공격!
*** Marine : Marine 2 ***
Location : ( 2 , 3 )
HP : 45
*** Marine : Marine 1 ***
Location : ( 1 , 5 )
HP : 50
```

와 같이 나옴을 알 수 있습니다.

그런데 사실, 위 코드에는 또 다른 문제점이 있습니다.

```
Marine::Marine(int x, int y, const char* marine_name) {
    name = new char[strlen(marine_name) + 1];
    strcpy(name, marine_name);
    coord_x = x;
    coord_y = y;
    hp = 50;
    damage = 5;
    is_dead = false;
}
```

우리는 분명히 위 코드에서 `name`에 우리가 생성하는 마린의 이름을 넣어줄 때, `name`을 동적으로 생성해서 문자열을 복사하였는데요, 그럼, 이렇게 동적으로 할당된 `char` 배열에 대한 `delete`는 언제 이루어지는 것인가요?

안타깝게도, 우리가 명확히 `delete`를 지정하지 않는 한 자동으로 `delete`가 되는 경우는 없습니다. 다시 말해서 우리가 동적으로 할당했던 저 `name`은 영원히 메모리 공간 속에서 둑둥 떠다닌다는 말이지요. 사실 몇 바이트 정도 밖에 되지 않을 것이지만 위와 같은 `name`들이 쌓이고 쌓이게 되면 메모리 누수 (Memory Leak) 이라는 문제점이 발생하게 됩니다 (가끔 몇몇 프로그램들이 비정상적으로 많은 메모리를 점유하는 것 보이시지 않나요?)

그렇다면, 만일 `main` 함수 끝에서 `Marine`이 `delete`될 때, 즉 우리가 생성했던 객체가 소멸될 때 자동으로 호출되는 함수 - 마치 객체가 생성될 때 자동으로 호출되었던 생성자처럼 소멸될 때 자동으로 호출되는 함수가 있다면 얼마나 좋을까요? 놀랍게도 이미 C++에서는 이 기능을 지원하고 있습니다. 바로 소멸자(Destructor) 이죠.

```
#include <string.h>

#include <iostream>

class Marine {
    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    int damage; // 공격력
    bool is_dead;
    char* name; // 마린 이름

public:
    Marine(); // 기본 생성자
    Marine(int x, int y, const char* marine_name); // 이름까지 지정
    Marine(int x, int y); // x, y 좌표에 마린 생성
    ~Marine();

    int attack(); // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y); // 새로운 위치
```

```
void show_status(); // 상태를 보여준다.  
};  
Marine::Marine() {  
    hp = 50;  
    coord_x = coord_y = 0;  
    damage = 5;  
    is_dead = false;  
    name = NULL;  
}  
Marine::Marine(int x, int y, const char* marine_name) {  
    name = new char[strlen(marine_name) + 1];  
    strcpy(name, marine_name);  
  
    coord_x = x;  
    coord_y = y;  
    hp = 50;  
    damage = 5;  
    is_dead = false;  
}  
Marine::Marine(int x, int y) {  
    coord_x = x;  
    coord_y = y;  
    hp = 50;  
    damage = 5;  
    is_dead = false;  
    name = NULL;  
}  
void Marine::move(int x, int y) {  
    coord_x = x;  
    coord_y = y;  
}  
int Marine::attack() { return damage; }  
void Marine::be_attacked(int damage_earn) {  
    hp -= damage_earn;  
    if (hp <= 0) is_dead = true;  
}  
void Marine::show_status() {  
    std::cout << " *** Marine : " << name << " ***" << std::endl;  
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "  
        << std::endl;  
    std::cout << " HP : " << hp << std::endl;  
}  
Marine::~Marine() {  
    std::cout << name << " 의 소멸자 호출 ! " << std::endl;  
    if (name != NULL) {  
        delete[] name;  
    }  
}  
int main() {  
    Marine* marines[100];
```

```

marines[0] = new Marine(2, 3, "Marine 2");
marines[1] = new Marine(1, 5, "Marine 1");

marines[0]->show_status();
marines[1]->show_status();

std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;

marines[0]->be_attacked(marines[1]->attack());

marines[0]->show_status();
marines[1]->show_status();

delete marines[0];
delete marines[1];
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine : Marine 2 ***
Location : ( 2 , 3 )
HP : 50
*** Marine : Marine 1 ***
Location : ( 1 , 5 )
HP : 50

마린 1 이 마린 2 를 공격!
*** Marine : Marine 2 ***
Location : ( 2 , 3 )
HP : 45
*** Marine : Marine 1 ***
Location : ( 1 , 5 )
HP : 50
Marine 2 의 소멸자 호출 !
Marine 1 의 소멸자 호출 !

```

와 같이 나오게 됩니다.

생성자가 클래스 이름과 똑같이 생겼다면 소멸자는 그 앞에 ~ 만 붙여주시면 됩니다.

~(클래스의 이름)

우리의 `Marine` 클래스의 소멸자의 경우

```
~Marine();
```

위와 같이 생겼지요. 생성자와 한 가지 다른 점은, 소멸자는 인자를 아무것도 가지지 않는다는 것입니다. 생각해보세요. 소멸하는 객체에 인자를 넘겨서 무엇을 하겠습니까? 다시 말해, 소멸자는 오버로딩도 되지 않습니다.

우리의 소멸자의 내용을 살펴보자면

```
Marine::~Marine() {
    std::cout << name << " 의 소멸자 호출 ! " << std::endl;
    if (name != NULL) {
        delete[] name;
    }
}
```

위와 같이 `name`이 `NULL`이 아닐 경우에 (즉 동적으로 할당이 되었을 경우에)만 `delete`로 `name`을 삭제하는 것을 알 수 있습니다. 참고로 `name` 자체가 `char`의 배열로 동적할당 하였기 때문에 `delete` 역시 `delete [] name`, 즉 `[]`를 꼭 써주어야만 합니다.

```
delete marines[0];
delete marines[1];
```

객체가 소멸될 때 소멸자가 호출된다고 출력하도록 했는데, 실제로 위 코드가 실행 시 소멸자 호출 메세지가 뜬다는 것을 확인할 수 있습니다.

```
// 소멸자 호출 확인하기
#include <string.h>
#include <iostream>

class Test {
    char c;

public:
    Test(char _c) {
        c = _c;
        std::cout << "생성자 호출" << c << std::endl;
    }
    ~Test() { std::cout << "소멸자 호출" << c << std::endl; }
};

void simple_function() { Test b('b'); }
int main() {
    Test a('a');
```

```
    simple_function();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
생성자 호출 a
생성자 호출 b
소멸자 호출 b
소멸자 호출 a
```

와 같이 나옵니다. 위 코드에서 여러분은 '객체가 파괴될 때 호출되는 소멸자'를 확실하게 확인할 수 있었을 것입니다.

```
class Test {
    char c;

public:
    Test(char _c) {
        c = _c;
        std::cout << "생성자 호출 " << c << std::endl;
    }
    ~Test() { std::cout << "소멸자 호출 " << c << std::endl; }
};
```

`Test` 클래스는 매우 간단한데, 생성자와 소멸자 호출 때 어떤 객체의 것이 호출되는지 확인하기 위해 `char c` 를 도입하였습니다.

```
int main() {
    Test a('a');
    simple_function();
}
```

일단 가장 먼저 `main` 함수에서 `a` 객체를 생성하였으므로 `a` 의 생성자가 호출됩니다. 그리고 `simple_function` 을 실행하게 되면,

```
void simple_function() { Test b('b'); }
```

`simple_function` 안에서 또 `b` 객체를 생성하므로 `b` 의 생성자가 호출되지요. 하지만 `b` 는 `simple_function` 의 지역 객체이기 때문에 `simple_function` 이 종료됨과 동시에 `b` 역시 소멸되게 됩니다. 따라서 끝에서 `b` 의 소멸자가 호출되지요.

```
int main() {
    Test a('a');
    simple_function();
}
```

`simple_function` 호출 후, 이제 `main` 함수가 종료될 때 마찬가지로 `main` 함수의 지역 객체였던 `a` 가 소멸되면서 `a` 의 소멸자가 호출됩니다. 자, 이제 그러면 왜 출력 결과가 `a - b - b - a` 순으로 나타났는지 이해가 되셨나요?

소멸자가 뭐 별거 있어? 라고 생각하시는 분들도 있겠지만, 사실은 소멸자의 역할은 상당히 중요합니다. 이 세상에 태어나는 일이 중요한 일이지만, 그 보다 더 중요한 일은 이 세상을 떠날 때 얼마나 깔끔하게 떠나는 지가 더욱 중요한 일이 듯이, 객체가 다른 부분에 영향을 끼치지 않도록 깔끔하게 소멸되는 일은 매우 중요한 일입니다.

소멸자가 하는 가장 흔한 역할은 위에서도 나타나 있지만, 객체가 동적으로 할당받은 메모리를 해제하는 일이라고 볼 수 있습니다. 그 외에도 (아직 배우진 않았지만) 쓰레드 사이에서 `lock` 된 것을 푸는 역할이라던지 등의 역할을 수행하게 됩니다.

참고로 우리가 따로 생성자를 정의하지 않더라도 디폴트 생성자가 있었던 것처럼, 소멸자도 디폴트 소멸자(**Default Destructor**)가 있습니다. 물론, 디폴트 소멸자 내부에선 아무런 작업도 수행하지 않습니다. 만일 소멸자가 필요 없는 클래스라면 굳이 소멸자를 따로 써줄 필요는 없습니다.

복사 생성자

사실 스타 유즈맵을 조금이나마 해본 사람이라면 아래 그림과 같은 '포토캐논 겹치기' 정도는 한 번 접해보셨을 것입니다.



사실 위에 나타나 있는 포토캐논의 모습은 한 개가 아니라 수십 개의 포토캐논이 서로 겹친 모습입니다. 다시 말해 같은 포토캐논들이 수 백개 '복사' 되었다고 볼 수 있지요. 위와 같이 동일한 포토캐논을 만들어 내는 방법은 각각의 포토캐논을 일일히 생성자로 생성 할 수도 있지만, 1 개만 생성해 놓고, 그 한 개를 가지고 나머지 포토캐논들은 '복사 생성' 할 수도 있는 것입니다.

```
// 포토캐논
#include <string.h>

#include <iostream>

class Photon_Cannon {
    int hp, shield;
    int coord_x, coord_y;
    int damage;

public:
    Photon_Cannon(int x, int y);
    Photon_Cannon(const Photon_Cannon& pc);

    void show_status();
};

Photon_Cannon::Photon_Cannon(const Photon_Cannon& pc) {
    std::cout << "복사 생성자 호출 !" << std::endl;
    hp = pc.hp;
    shield = pc.shield;
    coord_x = pc.coord_x;
    coord_y = pc.coord_y;
    damage = pc.damage;
}

Photon_Cannon::Photon_Cannon(int x, int y) {
```

```

    std::cout << "생성자 호출 !" << std::endl;
    hp = shield = 100;
    coord_x = x;
    coord_y = y;
    damage = 20;
}

void Photon_Cannon::show_status() {
    std::cout << "Photon Cannon" << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
}

int main() {
    Photon_Cannon pc1(3, 3);
    Photon_Cannon pc2(pc1);
    Photon_Cannon pc3 = pc2;

    pc1.show_status();
    pc2.show_status();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

생성자 호출 !
복사 생성자 호출 !
복사 생성자 호출 !
Photon Cannon
Location : ( 3 , 3 )
HP : 100
Photon Cannon
Location : ( 3 , 3 )
HP : 100

```

와 같이 나옵니다.

먼저 우리가 제작한 복사 생성자 (**copy constructor**) 부터 살펴보도록 합시다.

```
Photon_Cannon(const Photon_Cannon& pc);
```

사실 위는 복사 생성자의 표준적인 정의라고 볼 수 있습니다. 즉, 복사 생성자는 어떤 클래스 T 가 있다면

```
T(const T& a);
```

라고 정의됩니다. 즉, 다른 T의 객체 a를 상수 레퍼런스로 받는다는 이야기입니다. 여기서 a가 **const**이기 때문에 우리는 복사 생성자 내부에서 a의 데이터를 변경할 수 없고, 오직 새롭게 초기화 되는 인스턴스 변수들에게 '복사'만 할 수 있게 됩니다. 다시 말해,

```
Photon_Cannon::Photon_Cannon(const Photon_Cannon& pc) {
    std::cout << "복사 생성자 호출 !" << std::endl;
    hp = pc.hp;
    shield = pc.shield;
    coord_x = pc.coord_x;
    coord_y = pc.coord_y;
    damage = pc.damage;
}
```

위와 같이 복사 생성자 내부에서 pc의 인스턴스 변수들에 접근해서 객체의 **shield**, **coord_x**, **coord_y** 등을 초기화 할 수는 있지만

```
pc.coord_x = 3;
```

처럼 pc의 값 자체는 변경할 수 없다는 이야기입니다. (왜냐하면 **const** 레퍼런스로 인자를 받았기 때문이죠! 아직도 이해가 안되시면 이전에 [포인터에서 const의 용법](#)을 떠올려보시기 바랍니다. 정확히 하는 동작이 동일합니다.)

한 가지 중요한 점은 함수 내부에서 받은 인자의 값을 변화시키는 일이 없다면 꼭 **const**를 붙여주시기 바랍니다. 위와 같이 복사 생성자의 경우도, 인자로 받은 pc의 값을 변경할 일이 없기 때문에 아예 처음부터 **const** 인자로 받았지요. 이렇게 된다면 후에 발생 할 수 있는 실수들을 효과적으로 막을 수 있습니다. (예를 들어 **pc.coord_x = coord_x**로 쓴다던지)

주의 사항

인자로 받는 변수의 내용을 함수 내부에서 바꾸지 않는다면 앞에 **const**를 붙여 주는 것이 바람직합니다.

이제 위와 같이 정의된 복사 생성자를 실제로 어떻게 이용하는지 살펴보도록 합시다.

```
Photon_Cannon pc1(3, 3);
Photon_Cannon pc2(pc1);
```

일단 pc1은 **int x**, **int y**를 인자로 가지는 생성자가 오버로딩 되었고, pc2의 경우 인자로 pc1을 넘겼으므로 복사 생성자가 호출되었음을 알 수 있습니다.

```
Photon_Cannon pc3 = pc2;
```

그렇다면 위 코드는 어떻까요? 놀랍게도, 위 코드 역시 복사 생성자가 호출됩니다. C++ 컴파일러는 위 문장을 아래와 동일하게 해석합니다.

```
Photon_Cannon pc3(pc2);
```

따라서 복사 생성자가 호출되게 되는 것입니다. 물론, 위는 아주아주 특별한 경우입니다. 만일 그냥

```
pc3 = pc2;
```

를 했다면 이는 평범한 대입 연산이겠지만, 생성 시에 대입하는 연산, 즉 위에 같이 `Photon_Cannon pc3 = pc2;` 한다면, 복사 생성자가 호출되게 되는 것입니다. 이런식으로 `Photon_Cannon pc3 = pc2;`를 해석함으로써 사용자가 상당히 직관적이고 깔끔한 프로그래밍을 할 수 있습니다.

참고로 한 가지 더 말하자면,

```
Photon_Cannon pc3 = pc2;
```

와

```
Photon_Cannon pc3;
pc3 = pc2;
```

는 엄연히 다른 문장입니다. 왜냐하면 위의 것은 말 그대로 복사 생성자가 1 번 호출되는 것이고, 아래 것은 그냥 생성자가 1 번 호출되고, `pc3 = pc2;`라는 명령이 실행되는 것이지요. 다시 한 번 강조하지만, 복사 생성자는 오직 '생성' 시에 호출된다는 것을 명심하시면 됩니다.

그런데, 사실 디폴트 생성자와 디폴트 소멸자처럼, C++ 컴파일러는 이미 디폴트 복사 생성자 (**Default copy constructor**)를 지원해 주고 있습니다. 위 코드에서 복사 생성자를 한 번 지워 보시고 실행해보면, 이전과 정확히 동일한 결과가 나타남을 알 수 있습니다. 디폴트 복사 생성자의 경우 기존의 디폴트 생성자와 소멸자가 하는 일이 아무 것도 없었던 것과는 달리 실제로 '복사'를 해줍니다.

만일 우리가 위 `Photon_Cannon`의 디폴트 복사 생성자의 내용을 추정해 본다면

```
Photon_Cannon::Photon_Cannon(const Photon_Cannon& pc) {
    hp = pc.hp;
    shield = pc.shield;
    coord_x = pc.coord_x;
```

```

coord_y = pc.coord_y;
damage = pc.damage;
}

```

와 같이 생겼을 것입니다. 대응되는 원소들을 말 그대로 1 대 1 복사해주게 됩니다. 따라서 위와 같이 간단한 클래스의 경우 귀찮게 복사생성자를 써주지 않고도 디폴트 복사 생성자만 이용해서 복사 생성을 쉽게 처리할 수 있습니다.

디폴트 복사 생성자의 한계

이번에도 위의 마린 처럼 포토 캐논의 이름을 지어줄 수 있다는 사실을 알고 클래스 `Photon_Cannon`에 `char *name`을 추가 해주었습니다. 그리고, 복사 생성자는 그냥 위에서처럼 디폴트 복사 생성자를 사용하기로 했죠. 그 코드는 아래와 같습니다.

```

// 디폴트 복사 생성자의 한계
#include <string.h>

#include <iostream>

class Photon_Cannon {
    int hp, shield;
    int coord_x, coord_y;
    int damage;

    char *name;

public:
    Photon_Cannon(int x, int y);
    Photon_Cannon(int x, int y, const char *cannon_name);
    ~Photon_Cannon();

    void show_status();
};

Photon_Cannon::Photon_Cannon(int x, int y) {
    hp = shield = 100;
    coord_x = x;
    coord_y = y;
    damage = 20;

    name = NULL;
}

Photon_Cannon::Photon_Cannon(int x, int y, const char *cannon_name) {
    hp = shield = 100;
    coord_x = x;
    coord_y = y;
}

```

```
damage = 20;

name = new char[strlen(cannon_name) + 1];
strcpy(name, cannon_name);
}

Photon_Cannon::~Photon_Cannon() {
// 0 이 아닌 값은 if 문에서 true 로 처리되므로
// 0 인가 아닌가를 비교할 때 그냥 if(name) 하면
// if(name != 0) 과 동일한 의미를 가질 수 있다.

// 참고로 if 문 다음에 문장이 1 개만 온다면
// 중괄호를 생략 가능하다.

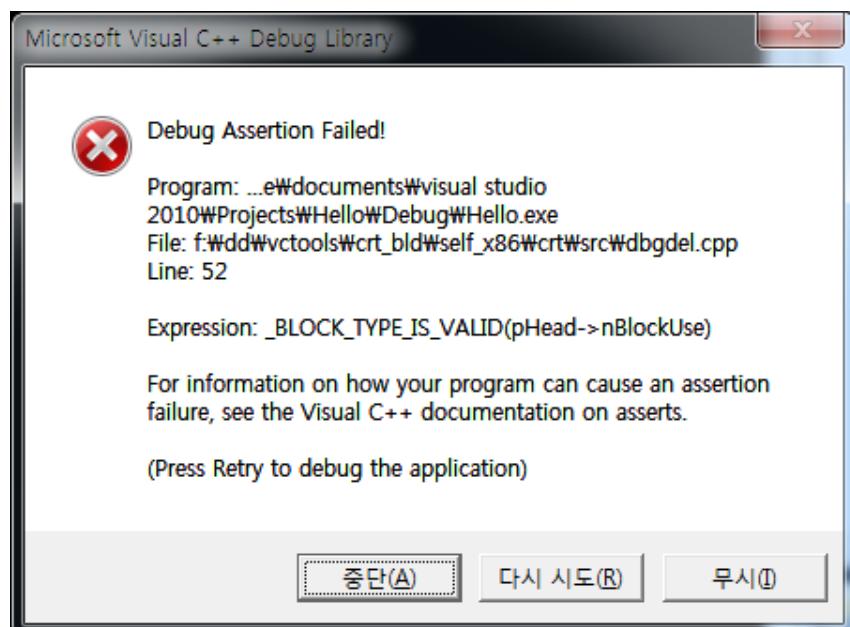
if (name) delete[] name;
}

void Photon_Cannon::show_status() {
std::cout << "Photon Cannon :: " << name << std::endl;
std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
<< std::endl;
std::cout << " HP : " << hp << std::endl;
}

int main() {
Photon_Cannon pc1(3, 3, "Cannon");
Photon_Cannon pc2 = pc1;

pc1.show_status();
pc2.show_status();
}
```

컴파일 후 실행해보면 아래와 같은 오류를 만나게 될 것입니다.

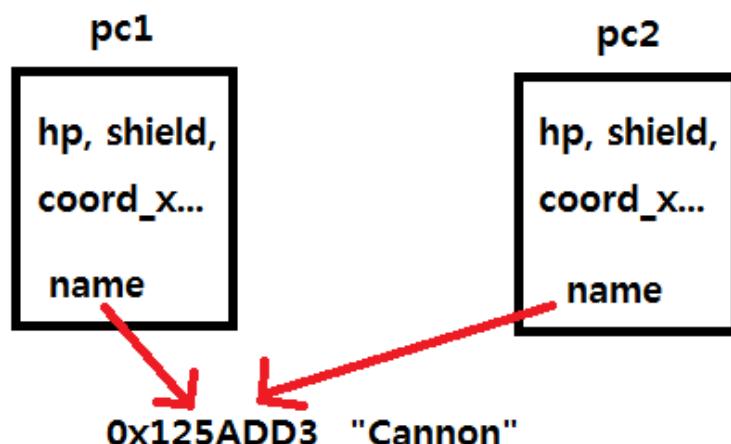


이럴 수가. 오래간만에 보는 런타임 오류입니다. 왜 이런 오류가 발생하였을까요? 분명히 디폴트 복사 생성자는 1 대 1로 원소들 간의 정확한 복사를 수행해 준다고 했었는데 말이죠.

그럼 일단, 여기서 우리의 디폴트 복사 생성자가 어떻게 생겼는지 살펴보도록 합시다. 아마도 추정 칸대, 컴파일러는 솔직하게 1 대 1 복사를 해주는 디폴트 복사 생성자를 아래와 같이 만들어 주었을 것입니다.

```
Photon_Cannon::Photon_Cannon(const Photon_Cannon& pc) {
    hp = pc.hp;
    shield = pc.shield;
    coord_x = pc.coord_x;
    coord_y = pc.coord_y;
    damage = pc.damage;
    name = pc.name;
}
```

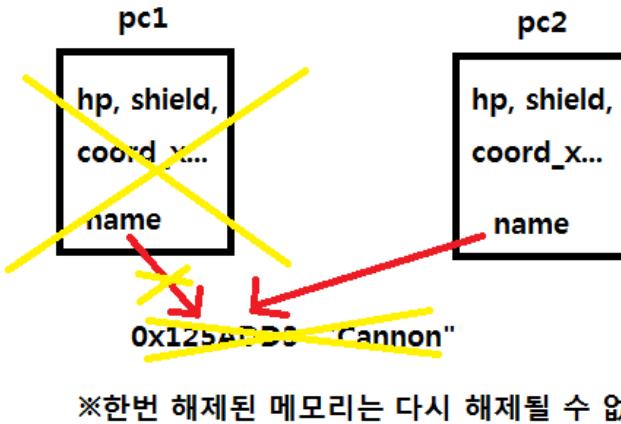
그렇다면 위 복사 생성자를 호출한 뒤에 pc1 과 pc2 가 어떻게 되었는지 살펴보도록 합시다.



당연히도, **hp**, **shield**, ... 그리고 **name** 까지 모두 같은 값을 갖게 됩니다. 여기서 **name** 이 같은 값 - 즉 두 개의 포인터가 같은 값을 가진다는 것은 같은 주소 값을 가리킨다는 말이 됩니다. 즉, 우리는 **pc1** 의 **name** 이 동적으로 할당받아서 가리키고 있던 메모리 ("Cannon"이라는 문자열이 저장된 메모리) 를 **pc2** 의 **name** 도 같이 가리키게 되는 것이지요.

물론 이 상태에서는 별 문제가 안됩니다. 뭐, 같은 메모리를 두 개의 서로 다른 포인터가 가리켜도 되기 때문이죠. 하지만 진짜 문제는 소멸자에서 일어납니다.

main 함수가 종료되기 직전에 생성되었던 객체들은 파괴되면서 소멸자를 호출하게 되죠. 만일 먼저 **pc1** 이 파괴되었다고 해봅시다.

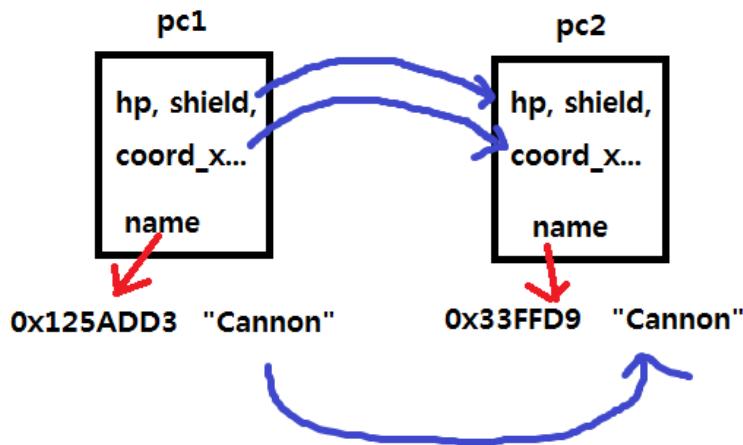


그러면 소멸자는 pc1의 내용을 모두 파괴 함과 동시에 0x125ADD3에 할당한 메모리 까지 delete하게 됩니다. 그런데 문제는 pc2의 name이 해제된 메모리인 0x125ADD3을 가리키고 있다는 것입니다.

```
Photon_Cannon::~Photon_Cannon() {
    if (name) delete[] name;
}
```

pc2에서 일단 name은 NULL이 아니므로 (0x125ADD3이라는 주소값을 가지고 있음) delete [] name이 수행되고, 이미 해제된 메모리에 접근해서 다시 해제하려고 하였기 때문에 (사실 접근한 것 자체만으로 오류) 위 그림과 같이 무서운 런타임 오류가 발생하게 됩니다. 그렇다면 이러한 문제를 막으려면 어떡할까요?

그 답은 간단합니다. 복사 생성자에서 name을 그대로 복사하지 말고 따로 다른 메모리에 동적 할당을 해서 그 내용만 복사하면 되겠지요? 이렇게 메모리를 새로 할당해서 내용을 복사하는 것을 깊은 복사(deep copy)라고 부르며 아까처럼 단순히 대입만 해주는 것을 얕은 복사(shallow copy)라고 부릅니다. 컴파일러가 생성하는 디폴트 복사 생성자의 경우 얕은 복사 밖에 할 수 없으므로 위와 같이 깊은 복사가 필요한 경우에는 사용자가 직접 복사 생성자를 만들어야 합니다.



위 과정을 그림으로 설명하자면 복사 생성자에서 `hp`, `shield` 와 같은 변수들은 얕은 복사를 하지만, `name`의 경우 따로 메모리를 할당해서 그 내용만 복사하는 깊은 복사를 수행하게 되는 것이지요. 그러면 소멸자에서도 메모리 해제시 각기 다른 메모리를 해제하는 것이기 때문에 전혀 문제가 발생하지 않습니다. 이를 바탕으로 복사 생성자를 만들어보면 아래와 같습니다.

```
// 복사 생성자의 중요성
#include <string.h>

#include <iostream>

class Photon_Cannon {
    int hp, shield;
    int coord_x, coord_y;
    int damage;

    char *name;

public:
    Photon_Cannon(int x, int y);
    Photon_Cannon(int x, int y, const char *cannon_name);
    Photon_Cannon(const Photon_Cannon &pc);
    ~Photon_Cannon();

    void show_status();
};

Photon_Cannon::Photon_Cannon(int x, int y) {
    hp = shield = 100;
    coord_x = x;
    coord_y = y;
    damage = 20;

    name = NULL;
}

Photon_Cannon::Photon_Cannon(const Photon_Cannon &pc) {
```

```
std::cout << "복사 생성자 호출! " << std::endl;
hp = pc.hp;
shield = pc.shield;
coord_x = pc.coord_x;
coord_y = pc.coord_y;
damage = pc.damage;

name = new char[strlen(pc.name) + 1];
strcpy(name, pc.name);
}

Photon_Cannon::Photon_Cannon(int x, int y, const char *cannon_name) {
    hp = shield = 100;
    coord_x = x;
    coord_y = y;
    damage = 20;

    name = new char[strlen(cannon_name) + 1];
    strcpy(name, cannon_name);
}

Photon_Cannon::~Photon_Cannon() {
    if (name) delete[] name;
}

void Photon_Cannon::show_status() {
    std::cout << "Photon Cannon :: " << name << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
}

int main() {
    Photon_Cannon pc1(3, 3, "Cannon");
    Photon_Cannon pc2 = pc1;

    pc1.show_status();
    pc2.show_status();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
복사 생성자 호출!
Photon Cannon :: Cannon
Location : ( 3 , 3 )
HP : 100
Photon Cannon :: Cannon
Location : ( 3 , 3 )
HP : 100
```

와 같이 제대로 작동하는 것을 볼 수 있습니다.

자, 이것으로 이번 강좌는 마치도록 하겠습니다. 사실 이 강좌에서 스타크래프트의 0.01% 도 구현하지 못한 것이지만, 차츰 우리는 그 뼈대를 만들어 나갈 것입니다. 자, 모두 화이팅!

주의 사항

혹시 이 강좌만 보고 뒤의 강좌를 안보시는 분들을 위해 노파심에 이야기 하지만, C++에서 문자열을 다룰 때 C 언어 처럼 널 종료 `char` 배열로 다루는 것을 매우 매우 매우 비추합니다. C++ 표준 라이브러리에서 `std::string`이라는 훌륭한 문자열 클래스를 제공하니까, 뒤의 강좌들도 꼭 읽어서 사용법을 숙지하시기 바랍니다.

생각해보기

문제 1

아래와 같은 문자열 클래스를 완성해보세요 (난이도 : 中)

```
class string {
    char *str;
    int len;

public:
    string(char c, int n); // 문자 c 가 n 개 있는 문자열로 정의
    string(const char *s);
    string(const string &s);
    ~string();

    void add_string(const string &s); // str 뒤에 s 를 붙인다.
    void copy_string(const string &s); // str 에 s 를 복사한다.
    int strlen(); // 문자열 길이 리턴
};
```

const, const, const!

안녕하세요 여러분. 무려 5 개월 만의 컴백입니다! 최근 들어서 그동안 바빴던 일이 잘 풀려서 여러 강좌들을 폭풍 업로드 하려 합니다. 아무쪼록 2013년 안으로 저의 씹어먹는 C++ 강좌가 완결될 수 있도록 노력하겠습니다 :) 그 동안 기다려 주셔서 정말로 감사합니다~ . 아무쪼록 2013년 안으로 저의 씹어먹는 C++ 강좌가 완결될 수 있도록 노력하겠습니다 :) 그 동안 기다려 주셔서 정말로 감사합니다!

이번 강좌에서는 지난 강좌에서 만들었던 `Marine` 클래스를 변형하는것 부터 시작하도록 하겠습니다.

생성자의 초기화 리스트(initializer list)

```
#include <iostream>

class Marine {
    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    int damage; // 공격력
    bool is_dead;

public:
    Marine(); // 기본 생성자
    Marine(int x, int y); // x, y 좌표에 마린 생성

    int attack(); // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y); // 새로운 위치

    void show_status(); // 상태를 보여준다.
};

Marine::Marine() : hp(50), coord_x(0), coord_y(0), damage(5), is_dead(false) {}

Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), damage(5), is_dead(false) {}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return damage; }
void Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;
}
```

```

}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
}

int main() {
    Marine marine1(2, 3);
    Marine marine2(3, 5);

    marine1.show_status();
    marine2.show_status();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 50

```

와 같이 됩니다.

예전에 만들었던 `Marine` 클래스와 달라진 것은 딱 하나인데, 바로 생성자에서 무언가 특이한 것을 도입했다는 것입니다. 한 번 살펴보도록 할까요.

```
Marine::Marine() : hp(50), coord_x(0), coord_y(0), damage(5), is_dead(false) {}
```

놀랍게도 함수 본체에는 아무것도 없습니다. 오직, 위에 추가된 이상한 것들이 기존의 생성자가 했던 일과 동일한 작업을 하고 있을 뿐입니다. 기존의 생성자는

```

Marine::Marine() {
    hp = 50;
    coord_x = coord_y = 0;
    damage = 5;
    is_dead = false;
}

```

게 생겼는데, 그 내부에서 하던 멤버 변수들의 초기화 작업들을 새롭게 추가한 것들이 대신해서 하고 있을 뿐입니다.

위와 같이 생성자 이름 뒤에

```
: hp(50), coord_x(0), coord_y(0),
damage(5), is_dead(false) {}
```

로 오는 것을 초기화 리스트 (initializer list) 라고 부르며, 생성자 호출과 동시에 멤버 변수들을 초기화해주게 됩니다.

```
Marine::Marine(int x, int y)
: coord_x(x), coord_y(y), hp(50), damage(5), is_dead(false) {}
```

위에서 coord_x 는 x 로 초기화 되고, is_dead 는 false 로 초기화 되게 됩니다.

멤버 초기화 리스트의 일반적인 꼴은 아래와 같습니다.

```
(생성자 이름) : var1(arg1), var2(arg2) {}
```

여기서 var 들은 클래스의 멤버 변수들을 지칭하고, arg 는 그 멤버 변수들을 무엇으로 초기화 할지 지칭하는 역할을 합니다. 한 가지 흥미로운 점은 var1 과 arg1 의 이름이 같아도 되는데, 실제로 아래의 예제는

```
Marine::Marine(int coord_x, int coord_y)
: coord_x(coord_x), coord_y(coord_y), hp(50), damage(5), is_dead(false) {}
```

정상적으로 작동합니다. 왜냐하면 coord_x (coord_x) 에서 바깥쪽의 coord_x 는 무조건 멤버 변수를 지칭하게 되는데, 이 경우 coord_x 를 지칭하는 것이고, 괄호 안의 coord_x 는 원칙상 Marine 이 인자로 받은 coord_x 를 우선적으로 지칭하는 것이기 때문입니다.

따라서 실제로, 인자로 받은 coord_x 가 클래스의 멤버 변수 coord_x 를 초기화 하게 됩니다. 아래는 당연한 얘기 겠지만

```
Marine::Marine(int coord_x, int coord_y) {
    coord_x = coord_x;
    coord_y = coord_y;
    hp = 50;
    damage = 5;
    is_dead = false;
}
```

컴파일러가 두 coord_x 모두 인자로 받은 coord_x 로 구분해서 오류가 나겠지요.

그렇다면, 왜 도대체 초기화 리스트를 사용해야 되냐고 물을 수 있습니다. 왜냐하면

```
Marine::Marine() {
    hp = 50;
    coord_x = coord_y = 0;
    damage = 5;
    is_dead = false;
}
```

나

```
Marine::Marine() : hp(50), coord_x(0), coord_y(0), damage(5), is_dead(false) {}
```

는 하는 일이 똑같아 보이기 때문이죠. 하지만 실제로 약간의 차이가 있습니다. 왜냐하면, 초기화 리스트를 사용한 버전의 경우 생성과 초기화를 동시에 하게 됩니다.

반면에 초기화 리스트를 사용하지 않는다면 생성을 먼저 하고 그 다음에 대입 을 수행하게 됩니다. 쉽게 말하면 초기화 리스트를 사용하는 것은

```
int a = 10;
```

이라 하는 것과 같고, 그냥 예전 버전의 생성자를 사용하는 것은

```
int a;
a = 10;
```

이라 하는 것과 동일하다는 것입니다. 만약에 int 가 대신에 클래스 였다면, 전자의 경우 복사 생성자가 호출되는데, 후자의 경우 디폴트 생성자가 호출된 뒤 대입이 수행된다는 이야기 이겠지요.

딱 보아도 후자가 조금 더 하는 작업이 많게 됩니다. 따라서 초기화 리스트를 사용하는 것이 조금 더 효율적인 작업이라는 사실을 알 수 있지요. 그 뿐만 아니라, 우리 경험상 반드시 '생성과 동시에 초기화 되어야 하는 것들' 이 몇 가지 있었습니다. 대표적으로 레퍼런스와 상수가 있지요.

앞서 배운 바에 따르면 상수와 레퍼런스들은 모두 생성과 동시에 초기화가 되어야 합니다.

```
const int a;
a = 3;

int& ref; // 이것이 왜 안되는지 기억이 안난다면
ref = c; // 레퍼런스 강좌를 참조
```

모두 컴파일 오류가 나겠지요. 따라서 만약에 클래스 내부에 레퍼런스 변수나 상수를 넣고 싶다면 이들을 생성자에서 무조건 초기화 리스트를 사용해서 초기화 시켜주어야만 합니다.

```
#include <iostream>

class Marine {
    int hp;                      // 마린 체력
    int coord_x, coord_y;        // 마린 위치
    bool is_dead;

    const int default_damage;    // 기본 공격력

public:
    Marine();                  // 기본 생성자
    Marine(int x, int y);      // x, y 좌표에 마린 생성

    int attack();               // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y);    // 새로운 위치

    void show_status();         // 상태를 보여준다.
};

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {}

Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), default_damage(5), is_dead(false) {}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return default_damage; }
void Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;
}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
              << std::endl;
    std::cout << " HP : " << hp << std::endl;
}

int main() {
    Marine marine1(2, 3);
    Marine marine2(3, 5);

    marine1.show_status();
    marine2.show_status();
```

```

    std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
    marine2.be_attacked(marine1.attack());

    marine1.show_status();
    marine2.show_status();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 50

```

마린 1 이 마린 2 를 공격!

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 45

```

이 됩니다.

위 마린 클래스는 프로그래머들의 실수로 마린의 공격력이 이상하게 변하는 것을 막기 위해서 기본 공격력이라는 상수 멤버를 도입해서, 딱 고정 시켜 버리고 마음 편하게 프로그래밍 할 수 있도록 하였습니다. 따라서 이를 위해 생성자에서 초기화 리스트를 도입해서

```

Marine::Marine()
: hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {}

```

와 같이, `default_damage` 를 생성과 동시에 초기화 할 수 있도록 하였습니다. 따라서 우리는 상수인 `default_damage` 를 5 로 초기화 할 수 있고, 이 값은 영원히 바뀌지 않게 됩니다.

```

Marine marine1(2, 3);
Marine marine2(3, 5);

```

```
marine1.show_status();
marine2.show_status();
```

위와 같이 `Marine` 의 객체들 (`marine1`, `marine2`) 를 생성하면서 생성자 오버로딩에 따라 `Marine(int x, int y)` 가 호출되는데, 이를 통해 각각 (2,3) 과 (3,5) 에 위치해 있는 마린 객체들을 만들 수 있게 되었습니다. 그리고, `show_status` 를 호출해 보면 이들이 제대로 정의되어 있다는 사실을 알 수 있습니다.

```
std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
marine2.be_attacked(marine1.attack());
```

마찬가지로 `Marine` 객체들이 서로 공격하는 과정도 잘 실행되고 있음을 알 수 있습니다.

위와 같이 중요한 값을 상수로 처리하는 것은 매우 유용한 일입니다. 다른 프로그래머가 이 클래스를 사용하면서 실수로 `marine` 의 `default_damage` 를 변경하는 명령을 집어 넣더라고 컴파일 오류가 발생하기 때문에 프로그램을 실행해서 지루한 디버깅 과정을 거쳐서 알아 내는 것 보다 훨씬 효율적으로 오류를 발견할 수 있겠지요.

```
#include <iostream>

class Marine {
    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    bool is_dead;

    const int default_damage; // 기본 공격력

public:
    Marine(); // 기본 생성자
    Marine(int x, int y); // x, y 좌표에 마린 생성
    Marine(int x, int y, int default_damage);

    int attack(); // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y); // 새로운 위치

    void show_status(); // 상태를 보여준다.
};

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {}

Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), default_damage(5), is_dead(false) {}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
```

```
    coord_y(y),  
    hp(50),  
    default_damage(default_damage),  
    is_dead(false) {}  
  
void Marine::move(int x, int y) {  
    coord_x = x;  
    coord_y = y;  
}  
int Marine::attack() { return default_damage; }  
void Marine::be_attacked(int damage_earn) {  
    hp -= damage_earn;  
    if (hp <= 0) is_dead = true;  
}  
void Marine::show_status() {  
    std::cout << " *** Marine *** " << std::endl;  
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "  
        << std::endl;  
    std::cout << " HP : " << hp << std::endl;  
}  
  
int main() {  
    Marine marine1(2, 3, 10);  
    Marine marine2(3, 5, 10);  
  
    marine1.show_status();  
    marine2.show_status();  
  
    std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;  
    marine2.be_attacked(marine1.attack());  
  
    marine1.show_status();  
    marine2.show_status();  
}
```

성공적으로 컴파일 하였다면

실행 결과

```
*** Marine ***  
Location : ( 2 , 3 )  
HP : 50  
*** Marine ***  
Location : ( 3 , 5 )  
HP : 50
```

```

마린 1 이 마린 2 를 공격!
*** Marine ***
Location : ( 2 , 3 )
HP : 50
*** Marine ***
Location : ( 3 , 5 )
HP : 40

```

이 예제에서는 생성자 하나를 새로 더 추가하였는데 한 번 살펴보도록 합시다.

```

Marine::Marine(int x, int y, int default_damage)
: coord_x(x),
  coord_y(y),
  hp(50),
  default_damage(default_damage),
  is_dead(false) {}

```

이전에는 `default_damage`에 초기화 리스트로 5를 전달하였는데, 이 생성자의 경우 어떤 값을 전달할지 인자로 받은 다음에 그 내용을 상수에 넣어주었습니다. 마찬가지로 이는

```
const int default_damage = (인자로 받은 default_damage);
```

를 실행한 것과 마찬가지이기 때문에 잘 작동됨을 알 수 있습니다. 그리고, 실제로 5가 아닌 10의 HP가 깎였음을 `show_status`를 통해 확인 할 수 있습니다.

생성된 총 Marine 수 세기 (static 변수)

자, 이번에는 여태까지 만들어지는 총 `Marine`의 수를 알아내기 위해 코드를 짠다고 생각해봅시다. 이를 위해서는 많은 방법이 있겠지만 가장 단순한 두 방식을 생각해본다면

1. 어떠한 배열에 `Marine`을 보관해 놓고, 생성된 마린의 개수를 모두 센다.
2. 어떤 변수를 만들어서 `Marine`의 생성시에 1을 추가하고, 소멸시에 1을 뺀다.

을 생각할 수 있을 것입니다. 첫 번째 방법의 경우, (물론 `vector`라는 자료형을 이용하면 쉽게 할 수 있겠지만 나중에 이야기 하도록 합시다.) 따로 크기가 자유자재로 변할 수 있는 배열을 따로 만들어야 하는 문제점이 있고, 두 번째의 같은 경우 만일 어떠한 함수 내에서 이런 변수를 정의하였다며 다른 함수에서도 그 값을 이용하기 위해 인자로 계속 전달해야 하는 귀찮음이 있습니다.

물론 전역 변수로 만들면 되지 않겠냐고 물을 수도 있겠지만, 전역 변수의 경우 프로젝트의 크기가 커질 수록 프로그래머의 실수로 인해 서로 겹쳐서 오류가 날 가능성이 다분하기에 반드시 필요한 경우가 아니면 사용을 하지 않습니다. (실제로 꼭 필요한 경우가 아니면 전역변수는 사용하지 맙시다) 하지만 C++ 에서는 위와 같은 문제를 간단하게 해결 할 수 있는 기능을 제공하고 있습니다. 마치 전역 변수 같지만 클래스 하나에만 종속되는 변수인 것인데요, 바로 `static` 멤버 변수입니다.

예전에 C 언어에서 어떠한 함수의 `static` 변수 ([여기](#) 참조) 가 지역 변수들처럼 함수가 종료될 때 소멸되는 것이 아니라 프로그램이 종료될 때 소멸되는 것처럼, 어떤 클래스의 `static` 멤버 변수의 경우, 멤버 변수들처럼, 객체가 소멸될 때 소멸되는 것이 아닌, 프로그램이 종료될 때 소멸되는 것입니다.

또한, 이 `static` 멤버 변수의 경우, 클래스의 모든 객체들이 '공유' 하는 변수로써 각 객체 별로 따로 존재하는 멤버 변수들과는 달리 모든 객체들이 '하나의' `static` 멤버 변수를 사용하게 됩니다. 그럼 바로 아래의 예제를 살펴 보도록 합시다.

```
// static 멤버 변수의 사용

#include <iostream>

class Marine {
    static int total_marine_num;

    int hp;           // 마린 체력
    int coord_x, coord_y; // 마린 위치
    bool is_dead;

    const int default_damage; // 기본 공격력

public:
    Marine();           // 기본 생성자
    Marine(int x, int y); // x, y 좌표에 마린 생성
    Marine(int x, int y, int default_damage);

    int attack();           // 데미지를 리턴한다.
    void be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y);           // 새로운 위치

    void show_status(); // 상태를 보여준다.

~Marine() { total_marine_num--; }
};

int Marine::total_marine_num = 0;

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {
    total_marine_num++;
}
```

```
Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
    coord_y(y),
    hp(50),
    default_damage(default_damage),
    is_dead(false) {
    total_marine_num++;
}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return default_damage; }

void Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;
}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
    std::cout << " 현재 총 마린 수 : " << total_marine_num << std::endl;
}

void create_marine() {
    Marine marine3(10, 10, 4);
    marine3.show_status();
}

int main() {
    Marine marine1(2, 3, 5);
    marine1.show_status();

    Marine marine2(3, 5, 10);
    marine2.show_status();

    create_marine();

    std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
    marine2.be_attacked(marine1.attack());

    marine1.show_status();
    marine2.show_status();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 1
*** Marine ***
Location : ( 3 , 5 )
HP : 50
현재 총 마린 수 : 2
*** Marine ***
Location : ( 10 , 10 )
HP : 50
현재 총 마린 수 : 3
```

마린 1 이 마린 2 를 공격!

```
*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 2
*** Marine ***
Location : ( 3 , 5 )
HP : 45
현재 총 마린 수 : 2
```

와 같이 나오게 됩니다.

```
static int total_marine_num;
```

먼저 위와 같이 클래스 `static` 변수를 정의하였습니다. 모든 전역 및 `static` 변수들은 정의와 동시에 값이 자동으로 0 으로 초기화 되기 때문에 이 경우 우리가 굳이 따로 초기화 하지 않아도 되지만 클래스 `static` 변수들의 경우 아래와 같은 방법으로 초기화 합니다.

```
int Marine::total_marine_num = 0;
```

간혹 어떤 사람들의 경우 클래스 내부에서

```
class Marine {
    static int total_marine_num = 0;
```

와 같이 초기화 해도 되지 않냐고 묻는 경우가 있는데, 멤버 변수들을 위와 같이 초기화 시키지 못하는 것처럼 `static` 변수 역시 클래스 내부에서 위와 같이 초기화 하는 것은 불가능 합니다. 위와 같은 꼴이 되는 유일한 경우는 `const static` 변수일 때만 가능한데, 실제로

```
class Marine {
    const static int x = 0;
```

으로 쓸 수 있습니다.

그럼 실제로 `total_marine_num`이 잘 작동하고 있는지 살펴보도록 합시다. 클래스의 편한 점이 생성자와 소멸자를 제공한다는 점인데, 덕분에 `Marine`이 생성될 때, 그리고 소멸될 때 굳이 따로 처리하지 않고도, 생성자와 소멸자 안에 `total_marine_num`을 조작하는 문장을 넣어주면 편하게 처리할 수 있습니다. 그 결과

```
Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
    coord_y(y),
    hp(50),
    default_damage(default_damage),
    is_dead(false) {
    total_marine_num++;
}
```

로 각 생성자 호출 시에 `total_marine_num`을 1씩 증가시키는 문장을 넣었고,

```
~Marine() { total_marine_num--; }
```

소멸 될 때는 1 감소시키는 문장을 넣었습니다.

```
Marine marine1(2, 3, 5);
marine1.show_status();

Marine marine2(3, 5, 10);
marine2.show_status();
```

따라서 위를 실행하면 실제로 총 Marine 의 수가 1, 2 늘어나는 것을 확인할 수 있고, 그 다음에 create_marine 을 실행하였을 때

```
void create_marine() {
    Marine marine3(10, 10, 4);
    marine3.show_status();
}
```

역시 marine3 을 생성함으로써 총 marine 의 수가 3 이 됨을 확인할 수 있는데, marine3 은 create_marine 의 지역 객체이기 때문에 create_marine 이 종료될 때 소멸되게 됩니다. 따라서 다시 main 함수로 돌아와서

```
std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
marine2.be_attacked(marine1.attack());

marine1.show_status();
```

에서 총 마린수를 표시할 때 2 명으로 나오게 됩니다.

그런데 클래스 안에 static 변수만 만들 수 있는 것이 아닙니다. 놀랍게도 클래스 안에는 static 함수도 정의할 수 있는데, static 변수가 어떠한 객체에 종속되는 것이 아니라, 그냥 클래스 자체에 딱 1 개 존재하는 것인 것 처럼, static 함수 역시 어떤 특정 객체에 종속되는 것이 아니라 클래스 전체에 딱 1 개 존재하는 함수입니다.

즉, static 이 아닌 멤버 함수들의 경우 객체를 만들어야지만 각 멤버 함수들을 호출할 수 있지만 static 함수의 경우, 객체가 없어도 그냥 클래스 자체에서 호출할 수 있게 됩니다. 그럼, 아래 예제를 살펴볼까요.

```
// static 함수
#include <iostream>

class Marine {
    static int total_marine_num;
    const static int i = 0;

    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
```

```
bool is_dead;

const int default_damage; // 기본 공격력

public:
Marine(); // 기본 생성자
Marine(int x, int y); // x, y 좌표에 마린 생성
Marine(int x, int y, int default_damage);

int attack(); // 데미지를 리턴한다.
void be_attacked(int damage_earn); // 입는 데미지
void move(int x, int y); // 새로운 위치

void show_status(); // 상태를 보여준다.
static void show_total_marine();
~Marine() { total_marine_num--; }
};

int Marine::total_marine_num = 0;
void Marine::show_total_marine() {
    std::cout << "전체 마린 수 : " << total_marine_num << std::endl;
}

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y)
    : coord_x(x), coord_y(y), hp(50), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
    coord_y(y),
    hp(50),
    default_damage(default_damage),
    is_dead(false) {
    total_marine_num++;
}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return default_damage; }
void Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;
}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
```

```
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
    std::cout << " 현재 총 마린 수 : " << total_marine_num << std::endl;
}

void create_marine() {
    Marine marine3(10, 10, 4);
    Marine::show_total_marine();
}

int main() {
    Marine marine1(2, 3, 5);
    Marine::show_total_marine();

    Marine marine2(3, 5, 10);
    Marine::show_total_marine();

    create_marine();

    std::cout << std::endl << "마린 1 이 마린 2 를 공격! " << std::endl;
    marine2.be_attacked(marine1.attack());

    marine1.show_status();
    marine2.show_status();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
전체 마린 수 : 1
전체 마린 수 : 2
전체 마린 수 : 3

마린 1 이 마린 2 를 공격!
*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 2
*** Marine ***
Location : ( 3 , 5 )
HP : 45
현재 총 마린 수 : 2
```

와 같이 나옵니다.

`static` 함수는 앞에서 이야기 한 것과 같이, 어떤 객체에 종속되는 것이 아니라 클래스에 종속되는 것으로, 따라서 이를 호출하는 방법도 (`객체`).(`멤버 함수`) 가 아니라,

```
Marine::show_total_marine();
```

와 같이 (`클래스`)`::(static` 함수) 형식으로 호출하게 됩니다. 왜냐하면 어떠한 객체도 이 함수를 소유하고 있지 않기 때문이죠. 그러하기에, `static` 함수 내에서는 클래스의 `static` 변수만을 이용할 수 밖에 없습니다. 만일 `static` 함수 내에서 아래처럼 그냥 클래스의 멤버 변수들을 이용한다면

```
void Marine::show_total_marine() {
    std::cout << default_damage << std::endl; // default_damage 는 멤버 변수
    std::cout << "전체 마린 수 : " << total_marine_num << std::endl;
}
```

`default_damage` 가 누구의 `default_damage` 인지 아무도 모르는 상황이 발생합니다. 즉, 어떤 객체의 `default_damage` 인지 `static` 변수인 `show_total_marine()` 은 알 수 없겠죠. 왜냐하면 앞에서 계속 말해왔듯이 어떤 객체에도 속해이지 않기 때문이니까요!

this

```
// 자기 자신을 가리키는 포인터 this
#include <iostream>

class Marine {
    static int total_marine_num;
    const static int i = 0;

    int hp; // 마린 체력
    int coord_x, coord_y; // 마린 위치
    bool is_dead;

    const int default_damage; // 기본 공격력
};

public:
    Marine(); // 기본 생성자
    Marine(int x, int y); // x, y 좌표에 마린 생성
    Marine(int x, int y, int default_damage);

    int attack(); // 데미지를 리턴한다.
    Marine& be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y); // 새로운 위치
};
```

```
void show_status(); // 상태를 보여준다.
static void show_total_marine();
~Marine() { total_marine_num--; }
};

int Marine::total_marine_num = 0;
void Marine::show_total_marine() {
    std::cout << "전체 마린 수 : " << total_marine_num << std::endl;
}

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y)
    : coord_x(x),
    coord_y(y),
    hp(50),

    default_damage(5),
    is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
    coord_y(y),
    hp(50),
    default_damage(default_damage),
    is_dead(false) {
    total_marine_num++;
}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() { return default_damage; }

Marine& Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;

    return *this;
}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
    std::cout << " 현재 총 마린 수 : " << total_marine_num << std::endl;
}
```

```

int main() {
    Marine marine1(2, 3, 5);
    marine1.show_status();

    Marine marine2(3, 5, 10);
    marine2.show_status();

    std::cout << std::endl << "마린 1 이 마린 2 를 두 번 공격! " << std::endl;
    marine2.be_attacked(marine1.attack()).be_attacked(marine1.attack());

    marine1.show_status();
    marine2.show_status();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 1
*** Marine ***
Location : ( 3 , 5 )
HP : 50
현재 총 마린 수 : 2

```

마린 1 이 마린 2 를 두 번 공격!

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 2
*** Marine ***
Location : ( 3 , 5 )
HP : 40
현재 총 마린 수 : 2

```

와 같이 나옵니다.

일단 가장 먼저 눈에 띄는 것은 바로 레퍼런스를 리턴하는 함수와 `this`라는 것인데, 차근 차근 살펴보도록 하겠습니다.

```

Marine& Marine::be_attacked(int damage_earn) {

```

```

hp -= damage_earn;
if (hp <= 0) is_dead = true;

return *this;
}

```

일단 `this` 라는 것이 C++ 언어 차원에서 정의되어 있는 키워드인데, 이는 객체 자신을 가리키는 포인터의 역할을 합니다. 즉, 이 멤버 함수를 호출하는 객체 자신을 가리킨다는 것이지요. 따라서, 실제로 위 내용은

```

Marine& Marine::be_attacked(int damage_earn) {
    this->hp -= damage_earn;
    if (this->hp <= 0) this->is_dead = true;

    return *this;
}

```

과 동일한 의미가 됩니다. (구조체 포인터 변수에서 `->` 를 이용해서 구조체 원소들에 접근했던 것을 상기해보세요) 실제로 모든 멤버 함수 내에서는 `this` 키워드가 정의되어 있으며 클래스 안에서 정의된 함수 중에서 `this` 키워드가 없는 함수는 (당연하게도) `static` 함수 뿐입니다.

그러면 이제 `Marine&` 을 리턴한다는 말이 도대체 무엇인지 생각해봅시다. 이전 강좌에서 배운 바에 따르면 레퍼런스라는 것이 어떤 변수의 다른 별명이라고 했습니다. (실제로 레퍼런스를 별명(alias)라고 부르기도 합니다)

그런데, 그 별명을 리턴한다니, 무슨 말일까요? '레퍼런스를 리턴하는 함수'에 대해 알아보기 위해 아래와 같은 짧막한 예제 클래스를 살펴보도록 합시다.

레퍼런스를 리턴하는 함수

```

// 레퍼런스를 리턴하는 함수
#include <iostream>

class A {
    int x;

public:
    A(int c) : x(c) {}

    int& access_x() { return x; }
    int get_x() { return x; }
    void show_x() { std::cout << x << std::endl; }
};


```

```

int main() {
    A a(5);
    a.show_x();

    int& c = a.access_x();
    c = 4;
    a.show_x();

    int d = a.access_x();
    d = 3;
    a.show_x();

    // 아래는 오류
    // int& e = a.get_x();
    // e = 2;
    // a.show_x();

    int f = a.get_x();
    f = 1;
    a.show_x();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

5
4
4
4

```

와 같이 나옵니다. 일단 위 클래스 A 는 아래와 같이 int 와 int 의 레퍼런스를 리턴하는 두 개의 함수를 가지고 있습니다.

```

int& access_x() { return x; }
int get_x() { return x; }

```

access_x 는 x 의 레퍼런스를 리턴하게 되고, get_x 는 x 의 '값' 을 리턴하게 되지요. 실제로 이들이 어떻게 작동하는지 살펴보도록 하겠습니다.

```

int& c = a.access_x();
c = 4;
a.show_x();

```

여기서 레퍼런스 `c` 는 `x` 의 레퍼런스, 즉 `x` 의 별명을 받았습니다. 따라서, `c` 는 `x` 의 별명으로 탄생하게 되는 것이지요. 레퍼런스를 리턴하는 함수는 그 함수 부분을 원래의 변수로 치환했다고 생각해도 상관이 없습니다. 다시 말해서

```
int &c = x; // 여기서 x 는 a 의 x
```

와 동일한 말이라는 것입니다. 따라서 `c` 의 값을 바꾸는 것은 `a` 의 `x` 의 값을 바꾸는 것과 동일한 의미이므로 (`c` 는 단순히 `x` 에 다른 이름을 붙여준 것일뿐!) `show_x` 를 실행 시에 `x` 의 값이 5에서 4로 바뀌었음을 알 수 있습니다. 그렇다면 아래 예도 살펴볼까요.

```
int d = a.access_x();
d = 3;
a.show_x();
```

이번에는 `int&` 가 아닌 그냥 `int` 변수에 '`x` 의 별명' 을 전달하였습니다. 만일 `d` 가 `int&` 였다면 `x` 의 별명을 받아서 `d` 역시 또 다른 `x` 의 별명이 되었겠지만, `d` 가 그냥 `int` 변수 이므로, 값의 복사가 일어나 `d` 에는 `x` 의 값이 들어가게 됩니다. 그리고 당연히, `d` 는 `x` 의 별명이 아닌 또 다른 독립적인 변수 이기에, `d = 3;` 을 해도 `x` 의 값은 바뀌지 않은 채, 그냥 4가 출력되게 되죠.

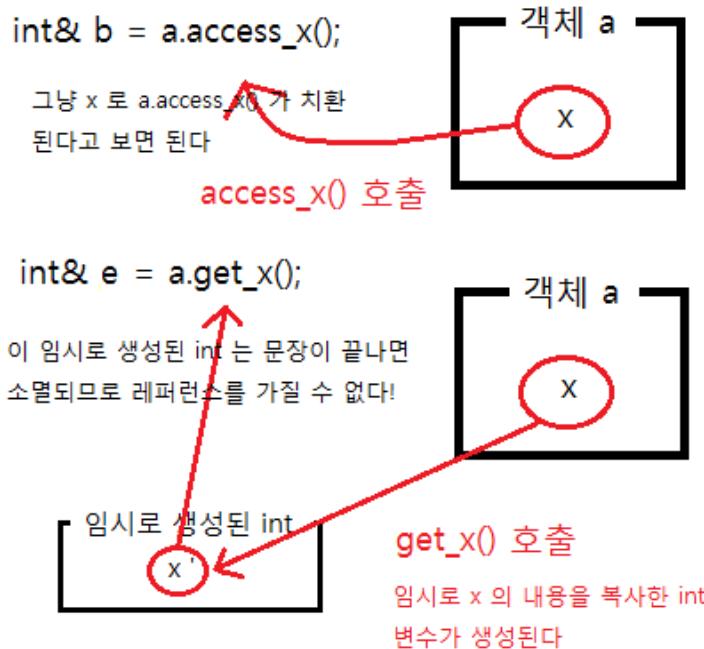
```
// 아래는 오류
// int& e = a.get_x();
// e = 2;
// a.show_x();
```

그럼 주석 처리된 위 예를 살펴봅시다. 주석을 풀면 컴파일이 안되므로 주석 처리 해 놓은 것인데, 실제로 주석을 풀고 컴파일을 해보면

컴파일 오류

```
error C2440: 'initializing' : cannot convert from 'int' to 'int &'
```

아래와 같은 오류가 발생합니다. 그 이유는 레퍼런스가 아닌 타입을 리턴하는 경우는 '값'의 복사가 이루어지기 때문에 임시 객체가 생성되는데, 임시객체의 레퍼런스를 가질 수 없기 때문입니다. (임시객체는 문장이 끝나게 되면 소멸됩니다) 이 과정을 그림으로 그려보면 아래와 같습니다.



`get_x`의 리턴으로 인해 임시로 '복사생성' 된 `int`는 `a.get_x()` 부분을 대체하며 위 그림의 경우

```
int &e = x'
```

과 같이 되는데, `x'`은 문장이 끝날 때 자동으로 소멸되는 임시 객체이기 때문에 레퍼런스를 만들 수 없습니다. 설사 레퍼런스를 만들었다고 해도 '이미 존재하지 않는 것에 대한 별명'이 되므로 이 레퍼런스에 접근하는 것은 오류이겠지요.

아무튼 이러한 이유로 `int`를 리턴하는 `a.get_x`에 대해서는 레퍼런스를 만들 수 없습니다. (정확한 설명을 하자면 `int&`는 좌측값에 대한 레퍼런스이고, `a.get_x()`는 우측값이기 때문에 레퍼런스를 만들 수 없습니다. 좌측값, 우측값 내용은 나중에 더 자세히 다루겠지만 궁금하신 분들은 [이 글을 읽어보세요!](#)!)

```
int f = a.get_x();
f = 1;
a.show_x();
```

마지막으로 위 코드는 익히 보았던 것처럼, 임시로 생성된 `int` 변수(위 그림에서는 `x'`)이 `f`에 복사되는데, 역시 `f = 1`한 것이 실제 객체 `a`의 `x`에게는 아무런 영향을 끼칠 수 없겠지요. 한 가지 재미있는 점은

```
a.access_x() = 3;
```

위 문장이 잘 작동한다는 점인데, 앞에서도 말했지만 '레퍼런스를 리턴하는 함수는 그 함수 부분을 리턴하는 원래 변수로 치환해도 됨다' 라는 말이 명확히 들어맞는다는 점입니다. 즉, 위 문장은 결국

```
a.x = 3;
```

과 동일한 말이 됩니다. 그에 반면, 잘 알고 있듯이

```
a.get_x() = 3;
```

은 역시 오류가 나게 되는데, 왜냐하면 `a.get_x()` 는 `get_x()` 가 리턴하면서 생성되는 임시 객체 (`x`) 으로 치환되며 임시객체에 대입을 하게 되는 모순적인 상황이 발생하게 됩니다.

그럼 이제 다시 예전의 `Marine` 예제로 돌아가보도록 합시다.

```
Marine& Marine::be_attacked(int damage_earn) {
    this->hp -= damage_earn;
    if (this->hp <= 0) this->is_dead = true;

    return *this;
}
```

위 경우 `be_attacked` 함수는 `Marine&` 타입을 리턴하게 되는데, 위 경우, `*this` 를 리턴하게 됩니다. 앞에서도 말했지만 `this` 가 지금 이 함수를 호출한 객체를 가리키는 것은 기억 하시죠? 그렇기 때문에 `*this` 는 그 객체 자신을 의미하게 됩니다. 따라서,

```
marine2.be_attacked(marine1.attack()).be_attacked(marine1.attack());
```

문장의 경우, 먼저 `marine2.be_attacked(marine1.attack())` 이 먼저 실행되고 리턴되는 것이 다시 `marine2` 이므로 그 다음에 또 한 번 `marine2.be_attacked(marine1.attack())` 가 실행된다고 생각할 수 있습니다.

간단하죠? 만일, `be_attacked` 함수의 리턴 타입이 `Marine&` 이 아니라 그냥 `Marine` 이라고 해봅시다. 즉, 만일 `be_attacked` 함수가 아래와 같이 바뀌었다고 가정한다면

```
MarineMarine::be_attacked(int damage_earn) {
    this->hp -= damage_earn;
    if (this->hp <= 0) this->is_dead = true;
```

```

    return *this;
}

```

위로 바뀐 함수를 가지고

```
marine2.be_attacked(marine1.attack()).be_attacked(marine1.attack());
```

를 실행해보면 marine2 는 실제로 두 번 공격이 아닌 1 번 공격으로 감소한 HP 만을 보입니다. (즉 40 이 아닌 45 로 나옴) 이는 앞에서도 설명했듯이 리턴타입이 Marine 이므로, 임시 객체 Marine 을 생성해서, *this 의 내용으로 복사가 되고 (즉, Marine 의 복사 생성자 호출) 이 임시 객체에 대한 be_attacked 함수가 호출되게 되는 것입니다.

따라서 결국 두 번째 be_attacked 는 marine2 가 아닌 영뚱한 임시 객체에 대해 호출되는 것이므로 결국 marine2 는 marine1 의 공격을 1 번만 받게 됩니다.

const 함수

C++ 에서는 변수들의 값을 바꾸지 않고 읽기 만 하는, 마치 상수 같 C++ 에서는 변수들의 값을 바꾸지 않고 읽기 만 하는, 마치 상수 같은멤버 함수를 '상수 함수'로써 선언할 수 있습니다. 아래의 예제를 살펴봅시다.

```

// 상수 멤버 함수
#include <iostream>

class Marine {
    static int total_marine_num;
    const static int i = 0;

    int hp;           // 마린 체력
    int coord_x, coord_y; // 마린 위치
    bool is_dead;

    const int default_damage; // 기본 공격력

public:
    Marine();           // 기본 생성자
    Marine(int x, int y); // x, y 좌표에 마린 생성
    Marine(int x, int y, int default_damage);

    int attack() const; // 데미지를 리턴한다.
    Marine& be_attacked(int damage_earn); // 입는 데미지
    void move(int x, int y); // 새로운 위치

    void show_status(); // 상태를 보여준다.
}

```

```
static void show_total_marine();
~Marine() { total_marine_num--; }
};

int Marine::total_marine_num = 0;
void Marine::show_total_marine() {
    std::cout << "전체 마린 수 : " << total_marine_num << std::endl;
}

Marine::Marine()
    : hp(50), coord_x(0), coord_y(0), default_damage(5), is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y)
    : coord_x(x),
    coord_y(y),
    hp(50),

    default_damage(5),
    is_dead(false) {
    total_marine_num++;
}

Marine::Marine(int x, int y, int default_damage)
    : coord_x(x),
    coord_y(y),
    hp(50),
    default_damage(default_damage),
    is_dead(false) {
    total_marine_num++;
}

void Marine::move(int x, int y) {
    coord_x = x;
    coord_y = y;
}

int Marine::attack() const { return default_damage; }

Marine& Marine::be_attacked(int damage_earn) {
    hp -= damage_earn;
    if (hp <= 0) is_dead = true;

    return *this;
}

void Marine::show_status() {
    std::cout << " *** Marine *** " << std::endl;
    std::cout << " Location : ( " << coord_x << " , " << coord_y << " ) "
        << std::endl;
    std::cout << " HP : " << hp << std::endl;
    std::cout << " 현재 총 마린 수 : " << total_marine_num << std::endl;
}

int main() {
```

```

Marine marine1(2, 3, 5);
marine1.show_status();

Marine marine2(3, 5, 10);
marine2.show_status();

std::cout << std::endl << "마린 1 이 마린 2 를 두 번 공격! " << std::endl;
marine2.be_attacked(marine1.attack()).be_attacked(marine1.attack());

marine1.show_status();
marine2.show_status();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 1
*** Marine ***
Location : ( 3 , 5 )
HP : 50
현재 총 마린 수 : 2

```

```

마린 1 이 마린 2 를 두 번 공격!
*** Marine ***
Location : ( 2 , 3 )
HP : 50
현재 총 마린 수 : 2
*** Marine ***
Location : ( 3 , 5 )
HP : 40
현재 총 마린 수 : 2

```

와 같이 나옵니다. 사실 위 소스는 거의 바뀐 것은 없고, 단순히 예시를 위해 아래와 같이 attack 함수를 살짝 바꾸었습니다.

```
int attack() const; // 데미지를 리턴한다.
```

일단 상수 함수는 위와 같은 형태로 선언을 하게 됩니다. 즉,

(기존의 **함수의** 정의) **const**;

그리고 함수의 정의 역시 **const** 키워드를 꼭 넣어주어야 하는데, 아래와 같이 말이지요.

```
int Marine::attack() const { return default_damage; }
```

그렇게 하였으면 위 **attack** 함수는 '상수 멤버 함수'로 정의된 것입니다. 우리는 상수 함수로 이 함수를 정의함으로써, 이 함수는 다른 변수의 값을 바꾸지 않는 함수라고 다른 프로그래머에게 명시 시킬 수 있습니다. 당연하게도, 상수 함수 내에서는 객체들의 '읽기' 만이 수행되며, 상수 함수 내에서 호출 할 수 있는 함수로는 다른 상수 함수 밖에 없습니다.

사실 많은 경우 클래스를 설계할 때, 멤버 변수들은 모두 **private**에 넣고, 이 변수들의 값에 접근하는 방법으로 **get_x** 함수처럼 함수를 **public**에 넣어 이 함수를 이용해 값을 리턴받는 형식을 많이 사용합니다. 이렇게 하면 멤버 변수들을 **private**에 넣음으로써 함부로 변수에 접근하는 것을 막고, 또 그 값은 자유롭게 구할 수 있게 됩니다.

그럼 이번 강좌는 여기서 마치도록 하겠습니다!

생각해보기

문제 1

아래와 같은 코드에서 복사 생성은 몇 번이나 표시될까요?

```
#include <iostream>

class A {
    int x;

public:
    A(int c) : x(c) {}
    A(const A& a) {
        x = a.x;
        std::cout << "복사 생성" << std::endl;
    }
};

class B {
    A a;

public:
    B(int c) : a(c) {}
```

```
B(const B& b) : a(b.a) {}
A get_A() {
    A temp(a);
    return temp;
}
};

int main() {
    B b(10);

    std::cout << "-----" << std::endl;
    A a1 = b.get_A();
}
```

(난이도 : 上 - 사실 이 글을 잘 읽었더라면 틀리게 답하는 것이 맞습니다. 컴파일러는 불필요한 복사를 막기 위해 *copy elision* 이라는 기술을 사용하고 있는데, 이에 관해서는 추후에 이야기하도록 하겠습니다. 정 궁금하신 분들은 http://en.wikipedia.org/wiki/Copy_elision 를 읽어보시기 바랍니다.)

내가 만들어보는 문자열 클래스

안녕하세요? 여러분. C++ 강좌도 벌써 10 번째 강좌입니다. 그 동안 잘 따라오고 있으셨나요? 이번 강좌는 그 동안 배운 내용을 종합해서 하나의 작은 프로젝트를 진행해보도록 할 것입니다. 이 강좌를 통해 여태까지 배운 C++ 클래스의 중요한 내용들을 복습하고 점검할 수 있는 기회가 되었으면 합니다.

이번 강좌에서는 문자열들을 효율적으로 관리하고 보관할 수 있는 문자열 클래스를 만들어보려고 합니다.

이 강좌를 읽기에 앞서 여태까지 배운 내용들을 복습할 겸 직접 자신만의 문자열 클래스를 만들어보기 바랍니다. 그 문자열 클래스는 아래와 같은 내용들을 지원해야 합니다.

1. 문자(`char`)로 부터의 문자열 생성, C 문자열 (`char *`)로 부터의 생성
2. 문자열 길이를 구하는 함수
3. 문자열 뒤에 다른 문자열 붙이기
4. 문자열 내에 포함되어 있는 문자열 구하기
5. 문자열이 같은지 비교
6. 문자열 크기 비교 (사전 순)

만일 위 내용을 다 만드셨다면 아래 내용을 읽으셔도 되고, 그렇지 않다면, 다시 한 번 도전해보시기 바랍니다!

주의 사항

참고로 C++ 표준에서는 문자열 클래스 (`string`)을 지원하므로 여러분이 직접 문자열 클래스를 만들어서 사용할 일은 거의 없을 것이고, 그 성능 역시 C++에서 제공하는 문자열 클래스를 이용하는 것이 훨씬 빠를 것입니다.

하지만, 페이스북이나 구글 같은 큰 회사들에서는 C++에서 기본적으로 지원하는 표준 문자열 대신에 자체적으로 최적화 된 버전을 만들어서 사용하고 있기는 합니다.

문자열 클래스를 만들자

기존 C 언어에서는 문자열을 나타내기 위해 널 종료 문자열(Null-terminated string)이라는 개념을 도입해서 문자열 끝에 NULL 문자를 붙여 문자열을 나타내는 방식을 사용하였습니다.

하지만 C 언어 문자열을 사용하는데에는 번거로움이 많았는데, 예를 들어서 만들어진 문자열의 크기를 바꾼다던지, 문자열 뒤에 다른 문자열을 붙인다던지 등의 작업들은 상당히 프로그래머 입장에서는 귀찮을 수 밖에 없습니다. 이와 같은 작업들을 문자열 클래스를 따로 만들어서 클래스 차원에서 지원해주면 상당히 편할 텐데 말이지요. 그래서 우리는 직접 문자열 클래스를 만들고자 합니다.

사실 C++에서는 표준 라이브러리로 `string` 클래스를 지원하고 있습니다. (실제로 `<string>` 헤더파일을 `include` 하면 사용할 수 있습니다.)

주의 사항

노파심에 이야기 하지만 C++에서는 정말 웬만하면 `char` 배열을 사용하는 것보다 `string` 을 사용해서 문자열을 다루는 것을 권장합니다. 뒤에 강좌에서 `string` 클래스를 어떻게 사용하는지 자세히 다룹니다.

하지만 이 막강한 `string` 클래스를 사용하기 이전에 우리는 직접 `MyString` 이라는 우리 만의 문자열 클래스를 만들고자 합니다.

일단 간단히 생각해서 우리가 만들 `MyString` 클래스에 멤버 변수로 무엇이 필요할지 생각해봅시다. 아마, 대표적으로 아래 두 개의 데이터들이 필요하다고 볼 수 있습니다.

1. 문자열 데이터가 저장된 공간을 가리키는 포인터
2. 문자열 데이터의 길이

왜 객체에 문자열 데이터를 직접 보관하는 것이 아니라, 그 저장된 공간을 가리키는 포인터를 보관하냐고 물을 수 있습니다. 이렇게 하는 이유는 나중에 문자열 데이터의 크기가 바뀔 때, 저장된 공간을 가리키는 방식으로 하면 그 메모리를 해제한 뒤에, 다시 할당할 수 있지만 직접 보관하면 그럴 수 없기 때문이죠.

또한, 문자열 데이터의 길이를 보관하는 이유는 문자열 길이를 사용할 일이 굉장히 많은데, 그 때마다 계속 길이를 구하는 것은 상당히 불필요한 일이기 때문입니다. 따라서 길이를 한 번 구해놓고 길이가 바뀔 때 까지 변경하지 않는 방법이 유용할 것입니다. 그럼, 위 내용을 바탕으로 한번 `MyString` 을 구성해보도록 하겠습니다.

```
class MyString {  
    char* string_content; // 문자열 데이터를 가리키는 포인터  
    int string_length;    // 문자열 길이  
};
```

일단 위 두 정보는 `private` 으로 설정하였습니다. 왜냐하면 우리는 다른 프로그래머가 저의 `MyString` 을 이용하면서 위와 같은 중요한 정보들에 함부로 접근하기를 원치 않거든요.

프로그래머가 실수로 `string_length` 를 조작하는 명령을 썼다가 자칫 잘못하기라도 하면 어떻겠습니까. 그렇기에 우리는 다른 프로그래머가 저의 `MyString` 을 `string_length` 나 `string_content` 를 직접 만지작거리지 않고도 그들의 원하는 모든 작업들을 수행할 수 있도록 충분한 '함수' 들을 제공해야 할 것입니다.

그럼 생성자들은 어떨까요. 일단, 위에 제가 구현하고자 요구했던 내용들을 충족시키기 위해서는 아래와 같은 생성자들을 만들어야 합니다.

```
// 문자 하나로 생성
MyString(char c);

// 문자열로 부터 생성
MyString(const char* str);

// 복사 생성자
MyString(const MyString& str);
```

위와 같은 생성자들을 만들기 전에, 어떠한 방식으로 문자열을 저장할 것인지에 대해 먼저 생각해보도록 합시다. 과연 그대로 `string_content` 에 C 형식의 문자열(널 종료 문자열) 을 보관하는 것이 좋을까요, 아니면 필요없는 정보들을 빼고 (즉 맨 마지막의 널 문자) 실제 '문자' 만 해당하는 부분만을 넣을까요.

C 형식의 문자열을 그대로 보관한다면, 문자열의 끝 부분을 쉽게 체크할 수 있다는 장점이 있지만 이 문제는 우리가 `string_length` 라는 변수를 같이 도입함으로써 해결할 수 있게 되었습니다. 따라서, 저희 `MyString` 클래스에서는 실제 문자에만 해당하는 내용만을 `string_content` 에 보관하도록 하겠습니다.

```
MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::~MyString() { delete[] string_content; }
```

위와 같이 3 개의 생성자들을 정의하였고, `string_content` 를 동적 할당으로 처리하였기 때문에 반드시 소멸자에서 이를 동적해제하는 것을 처리해줘야만 합니다.

이것이 클래스의 큰 장점이라고도 볼 수 있는데, C 언어에서 구조체같은 것으로 문자열을 구현하였다면 이를 일일히 해제 하는 것도 처리해줘야 했었기 때문입니다. 말 그대로, 클래스를 사용하는 사람은 안에서 어떻게 돌아가는지 전혀 신경쓰지 않고도 사용할 수 있게 되는 것입니다.

그럼 이제 매우 쉽게 문자열의 길이를 구하는 함수를 만들 수 있게 되었습니다. 단순히 `string_length` 만 리턴해 주면 되는 것입니다. 참고로 내부 변수의 내용을 바꾸지 않는다면 꼭 상수 함수로 정의해주는 것이 좋습니다.

주의 사항

내부 멤버 변수의 값을 바꾸지 않는다면 함수를 꼭 상수로 정의하세요.

`length` 함수 역시 `string_length` 의 값을 읽기만 하므로처럼 상수 함수로 정의하였습니다.

```
int MyString::length() const { return string_length; }
```

다만 이러한 방식으로 문자열의 길이를 구한다면, 문자열 조작시에 `string_length` 의 값을 올바른 값으로 설정해야만 합니다. 예를 들어서, 두 문자열을 서로 더해서 새로운 문자열을 만들 때 새로운 문자열의 `string_length` 는 두 문자열의 `string_length` 의 합이 되겠지요. 마찬가지로 부분 문자열을 추출하거나, 문자 하나를 지우는 등 모든 작업을 할 때 `string_length` 값을 정확하게 조정해야만 합니다.

```
void MyString::print() {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}
```

그리고 마지막으로, 우리의 `MyString` 클래스의 내용을 보기 위해서, 문자열을 출력하는 함수 `print` 와 `println` 을 만들었습니다. (단지 마지막에 개행을 하느냐 안하느냐의 차이) 마찬가지로 `print` 와 `println` 모두 문자열을 읽기만 하므로 상수 함수로 만들었습니다.

그럼, 우리의 현재 임시로 만들어 놓은 `MyString` 클래스가 잘 작동하고 있는지 살펴보도록 합시다.

```
#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
```

```
char* string_content; // 문자열 데이터를 가리키는 포인터
int string_length; // 문자열 길이

public:
// 문자 하나로 생성
MyString(char c);

// 문자열로 부터 생성
MyString(const char* str);

// 복사 생성자
MyString(const MyString& str);

~MyString();

int length() const;

void print() const;
void println() const;
};

MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}
```

```

    }

    std::cout << std::endl;
}

int main() {
    MyString str1("hello world!");
    MyString str2(str1);

    str1.println();
    str2.println();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

hello world!
hello world!

```

와 같이 잘 실행됨을 알 수 있습니다.

assign 함수

`assign` 함수는 '지정하다'라는 뜻을 가진 함수로, 우리가 흔히 생각하는 '='과 동일한 역할을 하게 됩니다. 예를 들어서 우리의 `MyString` 변수 `str`에서

```
str.assign("abc");
```

를 하게 된다면 `str`에는 원래 있었던 문자열이 지워지고 `abc`가 들어가게 되겠지요. 그렇다면 우리는 다음과 같은 두 개의 `assign` 함수를 준비할 수 있습니다.

```

MyString& assign(const MyString& str);
MyString& assign(const char* str);

```

물론 이 `assign` 함수들의 구현 자체는 매우 간단하게 할 수 있습니다. 저의 경우 다음과 같이 구현하였습니다.

```

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > string_length) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

```

```

        string_content = new char[str.string_length];
    }
    for (int i = 0; i != str.string_length; i++) {
        string_content[i] = str.string_content[i];
    }

    // 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
    // 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

    string_length = str.string_length;

    return *this;
}

MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > string_length) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}

```

`string`의 크기가 작으면 동적 할당을 수행하는데 큰 시간이 필요하지 않겠지만, 우리의 `MyString` 클래스는 어떤 크기의 문자열에 대해서도 좋은 성능을 보여주어야만 하기 때문에 위처럼 인자로 입력받는 문자열의 크기가, 원래 문자열의 크기 보다 작다면 굳이 동적 할당을 할 필요가 없게 되죠.

따라서 그 경우에는 그냥 그대로 복사하게 됩니다. 하지만, 인자로 입력받는 문자열의 크기가 더 크다면, 현재까지는 이전에 동적으로 할당된 메모리 바로 뒤에 메모리를 추가하는 방법은 없으므로, 새로 동적할당을 해줘야만 합니다.

그런데 이렇게 방식으로 구현하는데에는 약간의 문제가 있습니다. 예를 들어 다음과 같은 상황을 생각해봅시다.

```

MyString str1("very very very long string");
str1.assign("short string");
str1.assign("very long string");

```

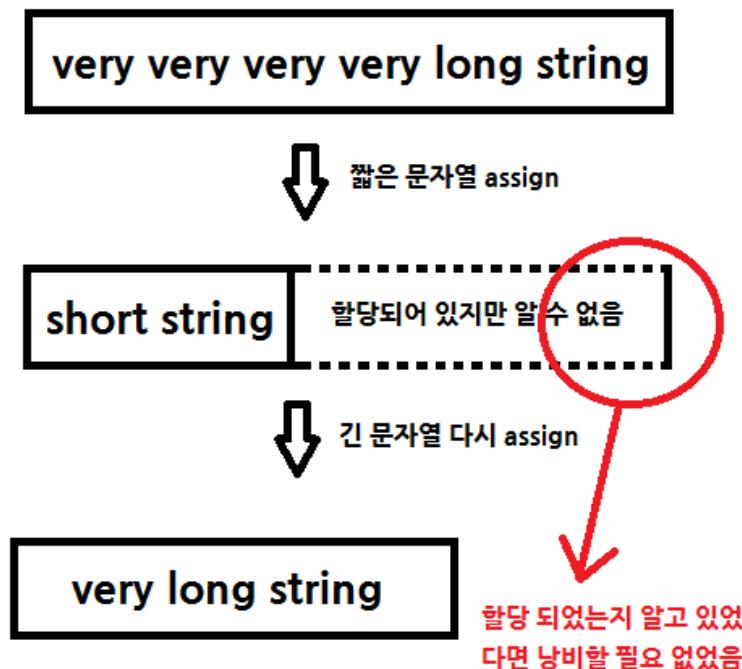
맨 처음에 `str1`에 아주 긴 문자열을 할당하였습니다. 예를 들어서 그 문자열의 길이가 1000 바이트

라고 생각해봅시다. (물론 위 소스에서는 겨우 몇 바이트 이겠지만 아무튼 아주 긴 문자열이라 생각합시다) 그런 다음에 `str1`에 다시 짧은 문자열을 `assign` 하였다고 가정합니다.

우리의 `assign` 함수에 따르면 맨 처음에 아주 긴 문자열의 길이가 짧은 문자열 보다 훨씬 작을 것이므로 `assign` 시에 특별히 동적 할당을 할 필요 없이 그냥 `string_length` 만을 줄인채 짧은 문자열로 덮어 씌우게 됩니다.

그런데 문제는 다시 `str1`에 긴 문자열을 `assign` 시에 발생하게 됩니다 (이번의 긴 문자열은 이전의 아주 긴 문자열 보다는 약간 짧다고 생각합니다). 비록 `str1`에 이미 1000 바이트에 달하는 공간이 할당되어 있는데도 불구하고 현재 짧은 문자열이 있기 때문에 새롭게 긴 문자열을 `assign` 시에 이미 1000 바이트가 할당되어 있다는 사실을 알 수 없게 됩니다.

따라서, `assign` 함수는 문자열에 짧은 문자열을 위한 작은 크기에 공간만이 할당되어 있다고 생각하여 메모리를 해제하고 다시 많은 양의 메모리를 할당하는 매우 비효율적인 작업을 하게 됩니다. 이 과정을 그림으로 나타내면 아래와 같습니다.



따라서 이러한 비효율적인 막기 위해서는 얼마나 많은 공간이 할당되어 있는지 알 수 있는 정보를 따로 보관하는 것이 좋을 것이라 생각됩니다. 이를 위해 `memory_capacity`라는, 현재 할당된 메모리 공간의 크기를 나타내는 새로운 변수를 추가하였습니다.

```

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
  
```

```

// 그러면 다시 할당을 해줘야만 한다.
delete[] string_content;

string_content = new char[str.string_length];
memory_capacity = str.string_length;
}
for (int i = 0; i != str.string_length; i++) {
    string_content[i] = str.string_content[i];
}

// 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
// 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

string_length = str.string_length;

return *this;
}

MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
        memory_capacity = str_length;
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}

```

이렇게 하게 된다면, 앞선 그림에서 나타나는 상황과 같은 문제를 방지할 수 있게 됩니다. 이렇게 `capacity` 를 도입함으로써 여러가지 새로운 함수들을 추가할 수 있게 되었습니다. 예를 들어서, 할당할 문자열의 크기를 미리 예약해 놓는 `reserve` 함수와 현재 문자열의 할당된 메모리 크기를 나타내는 `capacity` 함수를 만들 수 있습니다. 이들은 다음과 같습니다.

```

int MyString::capacity() { return memory_capacity; }
void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char *prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)

```

```

    string_content[i] = prev_string_content[i];

    delete[] prev_string_content;
}

// 만일 예약하려는 size 가 현재 capacity 보다 작다면
// 아무것도 안해도 된다.
}

```

참고로 `reserve` 함수의 경우, 만일 할당하려는 크기가 현재의 할당된 크기보다 작다면 굳이 할당할 필요가 없게 됩니다. 따라서 위와 같이 `size` 가 `memory_capacity` 보다 클 경우에만 할당하도록 처리하였습니다. 과연 잘 작동하는지 살펴볼까요.

```

#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이
    int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
    int capacity() const;
    void reserve(int size);

    void print() const;
    void println() const;

    MyString& assign(const MyString& str);
    MyString& assign(const char* str);
};

MyString::MyString(char c) {
    string_content = new char[1];

```

```
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}
MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    memory_capacity = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }

    std::cout << std::endl;
}

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str.string_length];
        memory_capacity = str.string_length;
    }

    for (int i = 0; i != str.string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}
```

```
// 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화  
// 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.  
  
string_length = str.string_length;  
  
return *this;  
}  
MyString& MyString::assign(const char* str) {  
    int str_length = strlen(str);  
    if (str_length > memory_capacity) {  
        // 그러면 다시 할당을 해줘야만 한다.  
        delete[] string_content;  
  
        string_content = new char[str_length];  
        memory_capacity = str_length;  
    }  
    for (int i = 0; i != str_length; i++) {  
        string_content[i] = str[i];  
    }  
  
    string_length = str_length;  
  
    return *this;  
}  
int MyString::capacity() const { return memory_capacity; }  
void MyString::reserve(int size) {  
    if (size > memory_capacity) {  
        char* prev_string_content = string_content;  
  
        string_content = new char[size];  
        memory_capacity = size;  
  
        for (int i = 0; i != string_length; i++)  
            string_content[i] = prev_string_content[i];  
  
        delete[] prev_string_content;  
    }  
  
    // 만일 예약하려는 size 가 현재 capacity 보다 작다면  
    // 아무것도 안해도 된다.  
}  
int main() {  
    MyString str1("very very very long string");  
    str1.reserve(30);  
  
    std::cout << "Capacity : " << str1.capacity() << std::endl;  
    std::cout << "String length : " << str1.length() << std::endl;  
    str1.println();  
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Capacity : 30
String length : 26
very very very long string
```

와 같이 잘 나옵을 알 수 있습니다.

마지막으로 추가할 함수는 임의의 위치의 문자를 리턴하는 함수로, 이전 C 언어에서 [] 로 구현되었던 것입니다. C 문자열의 경우 구조상 배열의 범위를 벗어나는 위치에 대한 문자를 요구하여도 이를 처리할 수 밖에 없었는데 (이는 결국 심각한 오류로 이루어졌죠) C++ 의 경우 특정 위치의 문자를 얻는 것을 함수로 만들어서 올바르지 않는 위치에 대한 문제를 처리할 수 있게 되었습니다.

```
char MyString::at(int i) const {
    if (i >= string_length || i < 0)
        return NULL;
    else
        return string_content[i];
}
```

위와 같이 i 가 허용되는 범위를 초과한다면 NULL 을 리턴하도록 하였습니다.

자 이것으로 해서, 우리가 직접 제작한 문자열 클래스 **MyString** 의 기본적인 함수들은 모두 제작하였다고 볼 수 있습니다.

- 문자 c 혹은 C 형식 문자열 str 에서 생성할 수 있는 생성자와 복사 생성자
- 문자열의 길이를 리턴하는 함수(length)
- 문자열 대입 함수(assign)
- 문자열 메모리 할당 함수(reserve) 및 현재 할당된 크기를 알아오는 함수(capacity)
- 특정 위치의 문자를 리턴하는 함수(at)

이제 **MyString** 을 사용하는 다른 프로그래머들은 이 최소한의 인터페이스를 이용해서, 여러가지 문자열에 관련한 모든 작업을 수행할 수 있게 됩니다. 하지만 실제로 여러분이 **MyString** 함수를 널리 편하게 사용하고 싶다면, 더 많은 기능을 제공할 수 있어야 하겠습니다.

문자열 삽입하기 (insert)

문자열 처리에서 가장 빈번하게 사용되는 작업으로, 문자열 중간에 다른 문자열을 삽입하는 작업을 들 수 있습니다. 사실 여태까지 만들언 놓은 함수들만을 가지고도 `insert` 작업을 쉽게 구현할 수 있겠지만, 빈번하게 사용되는 작업이다 보니까 미리 만들어 놓아서 인터페이스로 제공하는 것도 나쁘지 않을 것이라 생각됩니다.

```
MyString& MyString::insert(int loc, const MyString& str);
MyString& MyString::insert(int loc, const char* str);
MyString& MyString::insert(int loc, char c);
```

일단 저의 경우 `insert` 작업이 워낙 다양한 용도로 빈번하게 사용되기 때문에 위와 같은 3 개의 `insert` 함수를 준비하였습니다. 참고로 `loc` 을 어떻게 생각할 지 미리 기준을 정해야 하는데, 일반적으로 `insert` 함수에서 입력 위치를 받는 경우, 그 입력 위치 '앞'에 문자를 `insert` 하는 경우가 많습니다.

예를 들어서 `abc` 라는 문자열에 `insert(1, "d")` 를 하게 된다면, 1 의 위치에 있는 `b` 앞에 (참고로 모든 위치는 배열의 인덱스로 생각합니다. 즉 `a` 는 0 의 위치) `d` 가 삽입됩니다.

저는 맨 위의 `MyString` 을 인자로 받는 함수 하나만 제작할 것입니다. 왜냐하면 이 함수만 제대로 제작한다면 나머지 아래의 두 함수는

```
MyString& MyString::insert(int loc, const char* str) {
    MyString temp(str);
    return insert(loc, temp);
}
MyString& MyString::insert(int loc, char c) {
    MyString temp(c);
    return insert(loc, temp);
}
```

와 같이 간단하게 처리할 수 있기 때문이지요. 따라서 우리가 제대로 만들어야 할 함수는 맨 위의 `MyString` 을 인자로 받는 함수입니다.

```
MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i 의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc 라는 문자열에 insert(1, "d") 를 하게 된다면 adbc 가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) return *this;

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.
        memory_capacity = string_length + str.string_length;
```

```

char* prev_string_content = string_content;
string_content = new char[memory_capacity];

// 일단 insert 되는 부분 직전까지의 내용을 복사한다.
int i;
for (i = 0; i < loc; i++) {
    string_content[i] = prev_string_content[i];
}

// 그리고 새롭게 insert 되는 문자열을 넣는다.
for (int j = 0; j != str.string_length; j++) {
    string_content[i + j] = str.string_content[j];
}

// 이제 다시 원 문자열의 나머지 뒷부분을 복사한다.
for (; i < string_length; i++) {
    string_content[str.string_length + i] = prev_string_content[i];
}

delete[] prev_string_content;

string_length = string_length + str.string_length;
return *this;
}

// 만일 초과하지 않는 경우 굳이 동적할당을 할 필요가 없게 된다.
// 효율적으로 insert 하기 위해, 밀리는 부분을 먼저 뒤로 밀어버린다.

for (int i = string_length - 1; i >= loc; i--) {
    // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
    string_content[i + str.string_length] = string_content[i];
}
// 그리고 insert 되는 문자 다시 집어넣기
for (int i = 0; i < str.string_length; i++)
    string_content[i + loc] = str.string_content[i];

string_length = string_length + str.string_length;
return *this;
}

```

제가 만든 `insert` 함수는, 이전의 `assign` 함수처럼 새로 메모리를 할당해야 할 경우와, 굳이 할당할 필요가 없는 경우를 나누어서 처리하도록 하였습니다. 만일 원 문자열의 길이 + 새로 삽입되는 문자열의 길이가, 현재의 할당된 메모리의 크기 보다 크다면 반드시 메모리를 새로 할당해야 하겠지만, 작은 경우에는, 굳이 메모리를 해제하고 재할당하는데 시간을 낭비할 필요가 없게 됩니다. 메모리를 다시 할당해야 하는 경우, 일단 `string_content`에 새로운 할당된 메모리 주소가 들어 가므로, 이전의 메모리 주소를 보관하기 위해 `prev_string_content` 함수를 이용하였습니다. 따라서 이를 이용해서, `string_content`에 삽입된 문자열을 손쉽게 집어 넣을 수 있었습니다.

반면에, 메모리를 다시 할당할 필요가 없는 경우 원래의 문자열 내용을 이용하여 삽입된 문자열을 `string_content` 에 넣어야 하므로 약간의 트릭을 이용하였습니다. 바로, 자리가 바뀌는 문자열들을 먼저 뒤로 밀어버리는 것입니다. 이미 메모리의 할당된 공간은 충분하기 때문에 뒤로 미는 것을 쉽게 수행할 수 있습니다.

```
for (int i = string_length - 1; i >= loc; i--) {
    // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
    string_content[i + str.string_length] = string_content[i];
```

예를 들어서 앞서 abc에서 d를 삽입하는 예에서, 1의 위치에 d를 넣었으므로, 자리가 바뀌는 것들은 bc가 됩니다. 따라서 먼저 bc를 뒤로 밀어버린 다음, 생긴 공간에 d를 집어 넣으면 되는 것입니다. 즉, 위 작업을 수행하면 abc는 abbc가 되고,

```
// 그리고 insert 되는 문자 다시 집어넣기
for (int i = 0; i < str.string_length; i++)
    string_content[i + loc] = str.string_content[i];
```

를 수행하면, `insert` 되는 문자가 밀린 문자열 공간에 들어가면서 abbc에서 adbc가 됩니다. 실제로 실행해보면 아래와 같이 잘 작동함을 알 수 있습니다.

```
#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이
    int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
    int capacity() const;
    void reserve(int size);
```

```
void print() const;
void println() const;

MyString& assign(const MyString& str);
MyString& assign(const char* str);

char at(int i) const;

MyString& insert(int loc, const MyString& str);
MyString& insert(int loc, const char* str);
MyString& insert(int loc, char c);
};

MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    memory_capacity = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}
```

```
    }

    std::cout << std::endl;
}

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str.string_length];
        memory_capacity = str.string_length;
    }
    for (int i = 0; i != str.string_length; i++) {
        string_content[i] = str.string_content[i];
    }

    // 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
    // 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

    string_length = str.string_length;

    return *this;
}
MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
        memory_capacity = str_length;
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}
int MyString::capacity() const { return memory_capacity; }
void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char* prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];
    }
}
```

```
    delete[] prev_string_content;
}

// 만일 예약하려는 size 가 현재 capacity 보다 작다면
// 아무것도 안해도 된다.
}

char MyString::at(int i) const {
    if (i >= string_length || i < 0) {
        return 0;
    } else {
        return string_content[i];
    }
}

MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i 의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc 라는 문자열에 insert(1, "d") 를 하게 된다면 adbc 가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) {
        return *this;
    }

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.
        memory_capacity = string_length + str.string_length;

        char* prev_string_content = string_content;
        string_content = new char[memory_capacity];

        // 일단 insert 되는 부분 직전까지의 내용을 복사한다.
        int i;
        for (i = 0; i < loc; i++) {
            string_content[i] = prev_string_content[i];
        }

        // 그리고 새롭게 insert 되는 문자열을 넣는다.
        for (int j = 0; j != str.string_length; j++) {
            string_content[i + j] = str.string_content[j];
        }

        // 이제 다시 원 문자열의 나머지 뒷부분을 복사한다.
        for (; i < string_length; i++) {
            string_content[str.string_length + i] = prev_string_content[i];
        }

        delete[] prev_string_content;

        string_length = string_length + str.string_length;
        return *this;
    }
}
```

```

// 만일 초과하지 않는 경우 굳이 동적할당을 할 필요가 없게 된다.
// 효율적으로 insert 하기 위해, 밀리는 부분을 먼저 뒤로 밀어버린다.

for (int i = string_length - 1; i >= loc; i--) {
    // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
    string_content[i + str.string_length] = string_content[i];
}
// 그리고 insert 되는 문자 다시 집어넣기
for (int i = 0; i < str.string_length; i++)
    string_content[i + loc] = str.string_content[i];

string_length = string_length + str.string_length;
return *this;
}

MyString& MyString::insert(int loc, const char* str) {
    MyString temp(str);
    return insert(loc, temp);
}

MyString& MyString::insert(int loc, char c) {
    MyString temp(c);
    return insert(loc, temp);
}

int main() {
    MyString str1("very long string");
    MyString str2("<some string inserted between>");
    str1.reserve(30);

    std::cout << "Capacity : " << str1.capacity() << std::endl;
    std::cout << "String length : " << str1.length() << std::endl;
    str1.println();

    str1.insert(5, str2);
    str1.println();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Capacity : 30
String length : 16
very long string
very <some string inserted between>long string

```

와 같이 잘 나옵니다.

훌륭한 MyString 클래스를 만들기 위해서, 한 가지 좀 더 생각해보아야 할 점들이 있습니다. 과연

`insert`를 사용하는 경우는 보통 어떤 경우일까요? 많은 경우 `insert`는 많은 문자열을 한 번에 집어넣는 것이 아니라, 작은 크기의 문자열들을 자주 집어넣는 경우가 많습니다.

즉, 큰 크기의 문자열을 한 번에 `insert`하는 작업 보다는 작은 크기의 문자열들을 여러번 `insert`하는 명령을 많이 수행한다는 뜻이지요. 그런데, 만일 이미 `capacity` 한계에 달한 문자열 클래스에 문자 a를 계속 추가하는 명령을 생각해보도록 합시다.

```
while (some_condition) {
    str.insert(some_location, 'a');
}
```

마치 위와 같은 명령 말이지요. 이미 `str` 가 `capacity` 한계에 도달했다고 가정했으므로, 매 `insert`마다 메모리를 해제하고, 1만큼 큰 메모리를 할당하는 작업을 반복하게 될 것입니다. 이는 `str`의 크기가 크다면 엄청난 작업의 낭비가 아닐 수 없습니다.

즉, 짜잘하게 계속 `insert`하는 명령에서 메모리 할당과 해제를 반복하지 않도록 하기 위해서라면, 통 크게 메모리를 미리 `reserve`해놓는 것이 필요합니다. 물론, 무턱대고 미리 엄청난 크기의 메모리를 할당해 놓을 수도 없는 일이지요. 만일 10 바이트 밖에 사용하지 않는다면, 이와 같이 짜잘하게 `insert`하는 문제를 피하기 위해 1000 바이트를 미리 할당해 놓는다면 소중한 자원의 낭비가 될 것입니다.

따라서 'insert' 작업에서의 잊은 할당/해제를 피하기 위해 미리 메모리를 할당해놓기' 와 '메모리를 할당해 놓되, 많은 자원을 낭비하지 않는다' 라는 두 조건을 모두 만족하는 방법이 있을까요? 물론 있습니다. 메모리를 미리 할당할 경우, 현재 메모리 크기의 두 배 정도를 할당해 놓는다는 것입니다. 이를 코드로 표현하면 아래와 같습니다.

```
MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc라는 문자열에 insert(1, "d")를 하게 된다면 adbc가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) return *this;

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.

        if (memory_capacity * 2 > string_length + str.string_length)
            memory_capacity *= 2;
        else
            memory_capacity = string_length + str.string_length;
    }

    // 생략..
}
```

즉 새로 할당해야 할 메모리 크기(`string_length + str.string_length`)가 현재의 `memory_capacity`의 두 배 이하라면, 아예 `memory_capacity`의 두 배에 달하는 크기를 할당해버리는

것입니다.

그리고 물론 `insert` 되는 문자열의 크기가 엄청 커서 `memory_capacity` 의 두 배를 뛰어 넘어버린다면 그냥 예약을 생각하지 않고 필요한 만큼 할당해버리면 됩니다. 이와 같은 방식으로 처리한다면, 빈번한 메모리의 할당/해제를 막을 수 있고 또 많은 메모리 공간을 낭비하지 않을수 있습니다.

참고로 이러한 방법은 C++에서 동적으로 할당되는 메모리를 처리하는데 매우 빈번하게 사용되는 기법중 하나입니다.

```
#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이
    int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
    int capacity() const;
    void reserve(int size);

    void print() const;
    void println() const;

    MyString& assign(const MyString& str);
    MyString& assign(const char* str);

    char at(int i) const;

    MyString& insert(int loc, const MyString& str);
    MyString& insert(int loc, const char* str);
    MyString& insert(int loc, char c);
};

}
```

```
MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    memory_capacity = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }

    std::cout << std::endl;
}

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str.string_length];
        memory_capacity = str.string_length;
    }

    for (int i = 0; i != str.string_length; i++) {
```

```
        string_content[i] = str.string_content[i];
    }

    // 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
    // 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

    string_length = str.string_length;

    return *this;
}

MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
        memory_capacity = str_length;
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}
int MyString::capacity() const { return memory_capacity; }
void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char* prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];

        delete[] prev_string_content;
    }

    // 만일 예약하려는 size 가 현재 capacity 보다 작다면
    // 아무것도 안해도 된다.
}
char MyString::at(int i) const {
    if (i >= string_length || i < 0) {
        return 0;
    } else {
        return string_content[i];
    }
}
```

```
MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i 의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc 라는 문자열에 insert(1, "d") 를 하게 된다면 adbc 가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) return *this;

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.

        if (memory_capacity * 2 > string_length + str.string_length)
            memory_capacity *= 2;
        else
            memory_capacity = string_length + str.string_length;

        char* prev_string_content = string_content;
        string_content = new char[memory_capacity];

        // 일단 insert 되는 부분 직전까지의 내용을 복사한다.
        int i;
        for (i = 0; i < loc; i++) {
            string_content[i] = prev_string_content[i];
        }

        // 그리고 새롭게 insert 되는 문자열을 넣는다.
        for (int j = 0; j != str.string_length; j++) {
            string_content[i + j] = str.string_content[j];
        }

        // 이제 다시 원 문자열의 나머지 뒷부분을 복사한다.
        for (; i < string_length; i++) {
            string_content[str.string_length + i] = prev_string_content[i];
        }

        delete[] prev_string_content;

        string_length = string_length + str.string_length;
        return *this;
    }

    // 만일 초과하지 않는 경우 굳이 동적할당을 할 필요가 없게 된다.
    // 효율적으로 insert 하기 위해, 밀리는 부분을 먼저 뒤로 밀어버린다.

    for (int i = string_length - 1; i >= loc; i--) {
        // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
        string_content[i + str.string_length] = string_content[i];
    }

    // 그리고 insert 되는 문자 다시 집어넣기
    for (int i = 0; i < str.string_length; i++)
        string_content[i + loc] = str.string_content[i];
```

```
    string_length = string_length + str.string_length;
    return *this;
}
MyString& MyString::insert(int loc, const char* str) {
    MyString temp(str);
    return insert(loc, temp);
}
MyString& MyString::insert(int loc, char c) {
    MyString temp(c);
    return insert(loc, temp);
}
int main() {
    MyString str1("very long string");
    MyString str2("<some string inserted between>");
    str1.reserve(30);

    std::cout << "Capacity : " << str1.capacity() << std::endl;
    std::cout << "String length : " << str1.length() << std::endl;
    str1.println();

    str1.insert(5, str2);
    str1.println();

    std::cout << "Capacity : " << str1.capacity() << std::endl;
    std::cout << "String length : " << str1.length() << std::endl;
    str1.println();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Capacity : 30
String length : 16
very long string
very <some string inserted between>long string
Capacity : 60
String length : 46
very <some string inserted between>long string
```

로 잘 수행됨을 알 수 있습니다.

erase 함수

앞서 `insert` 함수를 만들었으니, 이번에는 정 반대의 역할을 하는 `erase` 함수를 만들어보도록 합시다. `erase` 함수는 `insert` 함수보다 만들기 훨씬 쉬운데, 왜냐하면 기본적으로 데이터의 양이 줄어 드는 것이기 때문에 복잡하게 `capacity` 이런 것들을 생각할 필요가 없기 때문입니다.

```
MyString& erase(int loc, int num);
```

`erase` 함수는 위와 같이 생겼고, `loc`은 `insert`와 동일하게 `loc`의 해당하는 문자 앞 을 의미합니다. 그리고 `num`은 지우는 문자의 수를 의미하죠. 예를 들어서 `abcd`라는 문자열에서 `erase(1, 2);`를 하게 된다면, 1에 해당하는 문자 '`b`'의 앞에서부터 2 문자를 지우게 되어, `bc`가 지워져서 `ad`가 리턴됩니다.

```
MyString& MyString::erase(int loc, int num) {
    // loc 의 앞 부터 시작해서 num 문자를 지운다.
    if (num < 0 || loc < 0 || loc > string_length) return *this;

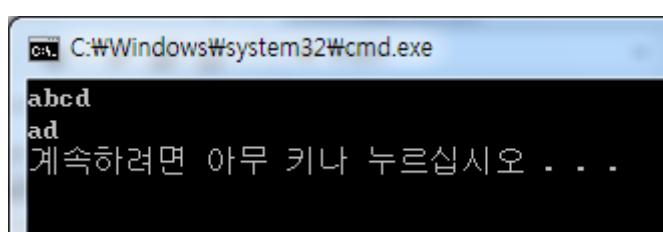
    // 지운다는 것은 단순히 뒤의 문자들을 앞으로 끌고 온다고
    // 생각하면 됩니다.

    for (int i = loc + num; i < string_length; i++) {
        string_content[i - num] = string_content[i];
    }

    string_length -= num;
    return *this;
}
```

위와 같이 간단히 처리할 수 있습니다. `string_length`를 도입하여서 정말 편리한 점이 무엇이냐면, 어차피 `string_length` 뒤에 어떠한 정보가 오든지 간에 별로 신경을 안써도 된다는 점입니다. 위와 같이 앞으로 문자들을 끌고 오면서 뒤의 문자들을 초기화 하지 않았지만, `string_length` 값을 처리하였기 때문에 뒤의 문자들에 신경 쓰지 않아도 됩니다.

물론 실제로 컴파일 해보면



로 아주 잘 작동함을 알 수 있습니다.

find 함수

이제 `insert` 와 `erase` 이외에 매우 빈번하게 사용되는 또 다른 작업으로 `find` 함수가 있습니다. 사실 `insert`, `erase`, `find` 는 문자열 연산의 기초 중의 기초라고 불러도 과언이 아닐 만큼 필수적인 함수입니다. 그렇기 때문에 `find` 함수 자체를 어떻게 구현하느냐에 따라 문자열 클래스의 전반적인 성능이 좌지우지 되는 경우도 있습니다.

왜냐하면 `insert` 와 `erase` 는 사실 연산 시간이 크게 오래 걸리지는 않지만 문자열의 크기가 매우 크다면 `find` 연산은 엄청나게 오래 걸릴 수 있게 될 수 있지요.

문자열을 검색하는 알고리즘은 수 없이 많지만, 어떤 상황에 대해서도 좋은 성능을 발휘하는 알고리즘은 없습니다. (예를 들어 짧은 문자열 검색에 최적화 된 알고리즘과 긴 문자열 검색에 최적화 된 알고리즘들 같이 말입니다) 그렇기에 특별한 알고리즘을 사용하는 경우에는 그 클래스의 사용 목적이 명확해서 그 알고리즘이 좋은 성능을 발휘할 수 있는 경우에만 사용하는 것이 보통입니다. 따라서 우리의 `MyString` 의 경우, 가장 간단한 방법으로 `find` 알고리즘을 구현하기로 하였습니다.

```
int find(int find_from, MyString& str) const;
int find(int find_from, const char* str) const;
int find(int find_from, char c) const;
```

일단 우리는 앞서 `insert` 함수를 구현한 방법처럼, 맨 위의 `MyString` 을 인자로 받는 `find` 만 제대로 구현한 후에, 아래 두 개의 `find` 는 맨 위의 함수를 이용해서 구현하는 방식으로 처리하였습니다.

참고로 `find` 함수는 `find_from` 에서부터 시작해서 가장 첫 번째 `str` 의 위치를 리턴하게 됩니다. 그리고 `str` 이 문자열에 포함되어 있지 않다면, -1 을 리턴하게 되지요. 이러한 방법으로, 어떤 문자열 내에 있는 모든 `str` 들을 찾을 수 있는 `for` 문을 생각할 수도 있을 것입니다.

```
int MyString::find(int find_from, MyString& str) const {
    int i, j;
    if (str.string_length == 0) return -1;
    for (i = find_from; i <= string_length - str.string_length; i++) {
        for (j = 0; j < str.string_length; j++) {
            if (string_content[i + j] != str.string_content[j]) break;
        }
        if (j == str.string_length) return i;
    }
    return -1; // 찾지 못했음
}
```

저의 경우 위와 같이 간단한 방법으로 `find` 함수를 구현하였습니다. `find_from` 부터 시작해서 `string_content` 와 `str` 가 완벽히 일치하는 부분이 생긴다면 그 위치를 리턴하고, 찾지 못할 경우 -1 을 리턴하도록 말이지요. 그럼 잘 작동하는지 살펴보도록 합시다.

```
#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이
    int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
    int capacity() const;
    void reserve(int size);

    void print() const;
    void println() const;

    MyString& assign(const MyString& str);
    MyString& assign(const char* str);

    char at(int i) const;

    MyString& insert(int loc, const MyString& str);
    MyString& insert(int loc, const char* str);
    MyString& insert(int loc, char c);

    MyString& erase(int loc, int num);

    int find(int find_from, const MyString& str) const;
    int find(int find_from, const char* str) const;
    int find(int find_from, char c) const;
};
```

```
MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    memory_capacity = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }

    std::cout << std::endl;
}

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str.string_length];
        memory_capacity = str.string_length;
    }

    for (int i = 0; i != str.string_length; i++) {
```

```
        string_content[i] = str.string_content[i];
    }

    // 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
    // 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

    string_length = str.string_length;

    return *this;
}

MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
        memory_capacity = str_length;
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}
int MyString::capacity() const { return memory_capacity; }
void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char* prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];

        delete[] prev_string_content;
    }

    // 만일 예약하려는 size 가 현재 capacity 보다 작다면
    // 아무것도 안해도 된다.
}
char MyString::at(int i) const {
    if (i >= string_length || i < 0) {
        return 0;
    } else {
        return string_content[i];
    }
}
```

```
MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i 의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc 라는 문자열에 insert(1, "d") 를 하게 된다면 adbc 가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) return *this;

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.

        if (memory_capacity * 2 > string_length + str.string_length)
            memory_capacity *= 2;
        else
            memory_capacity = string_length + str.string_length;

        char* prev_string_content = string_content;
        string_content = new char[memory_capacity];

        // 일단 insert 되는 부분 직전까지의 내용을 복사한다.
        int i;
        for (i = 0; i < loc; i++) {
            string_content[i] = prev_string_content[i];
        }

        // 그리고 새롭게 insert 되는 문자열을 넣는다.
        for (int j = 0; j != str.string_length; j++) {
            string_content[i + j] = str.string_content[j];
        }

        // 이제 다시 원 문자열의 나머지 뒷부분을 복사한다.
        for (; i < string_length; i++) {
            string_content[str.string_length + i] = prev_string_content[i];
        }

        delete[] prev_string_content;

        string_length = string_length + str.string_length;
        return *this;
    }

    // 만일 초과하지 않는 경우 굳이 동적할당을 할 필요가 없게 된다.
    // 효율적으로 insert 하기 위해, 밀리는 부분을 먼저 뒤로 밀어버린다.

    for (int i = string_length - 1; i >= loc; i--) {
        // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
        string_content[i + str.string_length] = string_content[i];
    }

    // 그리고 insert 되는 문자 다시 집어넣기
    for (int i = 0; i < str.string_length; i++)
        string_content[i + loc] = str.string_content[i];
```

```
string_length = string_length + str.string_length;
return *this;
}
MyString& MyString::insert(int loc, const char* str) {
    MyString temp(str);
    return insert(loc, temp);
}
MyString& MyString::insert(int loc, char c) {
    MyString temp(c);
    return insert(loc, temp);
}

MyString& MyString::erase(int loc, int num) {
    // loc 의 앞부터 시작해서 num 문자를 지운다.
    if (num < 0 || loc < 0 || loc > string_length) return *this;

    // 지운다는 것은 단순히 뒤의 문자들을 앞으로 끌고 온다고
    // 생각하면 됩니다.

    for (int i = loc + num; i < string_length; i++) {
        string_content[i - num] = string_content[i];
    }

    string_length -= num;
    return *this;
}

int MyString::find(int find_from, const MyString& str) const {
    int i, j;
    if (str.string_length == 0) return -1;
    for (i = find_from; i <= string_length - str.string_length; i++) {
        for (j = 0; j < str.string_length; j++) {
            if (string_content[i + j] != str.string_content[j]) break;
        }

        if (j == str.string_length) return i;
    }

    return -1; // 찾지 못했음
}

int MyString::find(int find_from, const char* str) const {
    MyString temp(str);
    return find(find_from, temp);
}

int MyString::find(int find_from, char c) const {
    MyString temp(c);
    return find(find_from, temp);
}

int main() {
    MyString str1("this is a very very long string");
    std::cout << "Location of first <very> in the string : " << str1.find(0, "very")
        << std::endl;
```

```
    std::cout << "Location of second <very> in the string : "
        << str1.find(str1.find(0, "very") + 1, "very") << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Location of first <very> in the string : 10
Location of second <very> in the string : 15
```

와 같이 잘 처리되고 있음을 알 수 있습니다. 참고로 첫 번째 `str1.find(0, "very")` 에서는, 맨 처음 부터 시작해서 "very" 를 찾습니다. 그 결과 10 의 위치에 있음을 알 수 있었지요. 그 다음 "very" 를 찾기 위해 이전의 검색 한 위치 바로 다음 부터 시작하여 또 "very" 를 찾습니다. 그 결과 15 의 위치에 있는 "very" 를 찾을 수 있게 됩니다.

크기 비교 함수 compare

마지막으로 만들 함수는 문자열 간의 크기를 비교하는 `compare` 함수입니다. 여기서 '크기' 를 비교한다는 의미는 사전식으로 배열해서 어떤 문자열이 더 뒤에 오는지 판단한다는 의미가 됩니다. 이 함수를 이용해서 문자열 전체를 정렬하는 함수라던지, 기존의 C 언어에서 `strcmp` 함수 등으로 지원하였던 것들을 그대로 사용할 수 있게 됩니다.

```
int compare(const MyString& str) const;
```

일단 함수의 원형은 위와 같이 `*this` 와 `str` 을 비교하는 형태로 이루어집니다.

```
int MyString::compare(const MyString& str) const {
    // (*this) - (str) 을 수행해서 그 1, 0, -1 로 그 결과를 리턴한다
    // 1 은 (*this) 가 사전식으로 더 뒤에 온다는 의미. 0 은 두 문자열
    // 이 같다는 의미, -1 은 (*this) 가 사전식으로 더 앞에 온다는 의미이다.

    for (int i = 0; i < std::min(string_length, str.string_length); i++) {
        if (string_content[i] > str.string_content[i])
            return 1;

        else if (string_content[i] < str.string_content[i])
            return -1;
    }

    // 여기 까지 했는데 끝나지 않았다면 앞 부분 까지 모두 똑같은 것이 된다.
    // 만일 문자열 길이가 같다면 두 문자열은 아예 같은 문자열이 된다.
```

```
if (string_length == str.string_length) return 0;

// 참고로 abc 와 abcd 의 크기 비교는 abcd 가 더 뒤에 오게 된다.
else if (string_length > str.string_length)
    return 1;

return -1;
}
```

참고로 말하면 abc 와 abcd 의 크기를 비교하면 abc 가 abcd 보다 사전식으로 더 앞에 오게 됩니다. 따라서 이에 대한 처리는 뒷부분에서 따로 하게 됩니다. 그리고 한 가지 더 말하자면 `std::min` 과 `std::max` 함수는 `iostream` 를 `include` 하면 사용할 수 있는 함수들 이므로, 굳이 귀찮게 만드실 필요는 없습니다.

```
#include <iostream>

// string.h 는 strlen 때문에 include 했는데, 사실 여러분이 직접 strlen
// 과 같은 함수를 만들어서 써도 됩니다.
#include <string.h>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length;    // 문자열 길이
    int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
    int capacity() const;
    void reserve(int size);

    void print() const;
    void println() const;

    MyString& assign(const MyString& str);
    MyString& assign(const char* str);

    char at(int i) const;
```

```
MyString& insert(int loc, const MyString& str);
MyString& insert(int loc, const char* str);
MyString& insert(int loc, char c);

MyString& erase(int loc, int num);

int find(int find_from, const MyString& str) const;
int find(int find_from, const char* str) const;
int find(int find_from, char c) const;

int compare(const MyString& str) const;
};

MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str[i];
    }
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    memory_capacity = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) {
        string_content[i] = str.string_content[i];
    }
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() const { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}
```

```
    }

    std::cout << std::endl;
}

MyString& MyString::assign(const MyString& str) {
    if (str.string_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str.string_length];
        memory_capacity = str.string_length;
    }
    for (int i = 0; i != str.string_length; i++) {
        string_content[i] = str.string_content[i];
    }

    // 그리고 굳이 str.string_length + 1 ~ string_length 부분은 초기화
    // 시킬 필요는 없다. 왜냐하면 거기 까지는 읽어들이지 않기 때문이다.

    string_length = str.string_length;

    return *this;
}
MyString& MyString::assign(const char* str) {
    int str_length = strlen(str);
    if (str_length > memory_capacity) {
        // 그러면 다시 할당을 해줘야만 한다.
        delete[] string_content;

        string_content = new char[str_length];
        memory_capacity = str_length;
    }
    for (int i = 0; i != str_length; i++) {
        string_content[i] = str[i];
    }

    string_length = str_length;

    return *this;
}
int MyString::capacity() const { return memory_capacity; }
void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char* prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];
    }
}
```

```
    delete[] prev_string_content;
}

// 만일 예약하려는 size 가 현재 capacity 보다 작다면
// 아무것도 안해도 된다.
}

char MyString::at(int i) const {
    if (i >= string_length || i < 0) {
        return 0;
    } else {
        return string_content[i];
    }
}

MyString& MyString::insert(int loc, const MyString& str) {
    // 이는 i 의 위치 바로 앞에 문자를 삽입하게 된다. 예를 들어서
    // abc 라는 문자열에 insert(1, "d") 를 하게 된다면 adbc 가 된다.

    // 범위를 벗어나는 입력에 대해서는 삽입을 수행하지 않는다.
    if (loc < 0 || loc > string_length) return *this;

    if (string_length + str.string_length > memory_capacity) {
        // 이제 새롭게 동적으로 할당을 해야 한다.

        if (memory_capacity * 2 > string_length + str.string_length)
            memory_capacity *= 2;
        else
            memory_capacity = string_length + str.string_length;

        char* prev_string_content = string_content;
        string_content = new char[memory_capacity];

        // 일단 insert 되는 부분 직전까지의 내용을 복사한다.
        int i;
        for (i = 0; i < loc; i++) {
            string_content[i] = prev_string_content[i];
        }

        // 그리고 새롭게 insert 되는 문자열을 넣는다.
        for (int j = 0; j != str.string_length; j++) {
            string_content[i + j] = str.string_content[j];
        }

        // 이제 다시 원 문자열의 나머지 뒷부분을 복사한다.
        for (; i < string_length; i++) {
            string_content[str.string_length + i] = prev_string_content[i];
        }

        delete[] prev_string_content;

        string_length = string_length + str.string_length;
    }
}
```

```
    return *this;
}

// 만일 초과하지 않는 경우 굳이 동적할당을 할 필요가 없게 된다.
// 효율적으로 insert 하기 위해, 밀리는 부분을 먼저 뒤로 밀어버린다.

for (int i = string_length - 1; i >= loc; i--) {
    // 뒤로 밀기. 이 때 원래의 문자열 데이터가 사라지지 않게 함
    string_content[i + str.string_length] = string_content[i];
}
// 그리고 insert 되는 문자 다시 집어넣기
for (int i = 0; i < str.string_length; i++)
    string_content[i + loc] = str.string_content[i];

string_length = string_length + str.string_length;
return *this;
}

MyString& MyString::insert(int loc, const char* str) {
    MyString temp(str);
    return insert(loc, temp);
}

MyString& MyString::insert(int loc, char c) {
    MyString temp(c);
    return insert(loc, temp);
}

MyString& MyString::erase(int loc, int num) {
    // loc 의 앞부터 시작해서 num 문자를 지운다.
    if (num < 0 || loc < 0 || loc > string_length) return *this;

    // 지운다는 것은 단순히 뒤의 문자들을 앞으로 끌고 온다고
    // 생각하면 됩니다.

    for (int i = loc + num; i < string_length; i++) {
        string_content[i - num] = string_content[i];
    }

    string_length -= num;
    return *this;
}

int MyString::find(int find_from, const MyString& str) const {
    int i, j;
    if (str.string_length == 0) return -1;
    for (i = find_from; i <= string_length - str.string_length; i++) {
        for (j = 0; j < str.string_length; j++) {
            if (string_content[i + j] != str.string_content[j]) break;
        }

        if (j == str.string_length) return i;
    }
}
```

```

    return -1; // 찾지 못했음
}
int MyString::find(int find_from, const char* str) const {
    MyString temp(str);
    return find(find_from, temp);
}
int MyString::find(int find_from, char c) const {
    MyString temp(c);
    return find(find_from, temp);
}
int MyString::compare(const MyString& str) const {
    // (*this) - (str) 을 수행해서 그 1, 0, -1 로 그 결과를 리턴한다
    // 1 은 (*this) 가 사전식으로 더 뒤에 온다는 의미. 0 은 두 문자열
    // 이 같다는 의미, -1 은 (*this) 가 사전식으로 더 앞에 온다는 의미이다.

    for (int i = 0; i < std::min(string_length, str.string_length); i++) {
        if (string_content[i] > str.string_content[i])
            return 1;

        else if (string_content[i] < str.string_content[i])
            return -1;
    }

    // 여기 까지 했는데 끝나지 않았다면 앞 부분 까지 모두 똑같은 것이 된다.
    // 만일 문자열 길이가 같다면 두 문자열은 아예 같은 문자열이 된다.

    if (string_length == str.string_length) return 0;

    // 참고로 abc 와 abcd 의 크기 비교는 abcd 가 더 뒤에 오게 된다.
    else if (string_length > str.string_length)
        return 1;

    return -1;
}
int main() {
    MyString str1("abcdef");
    MyString str2("abcde");

    std::cout << "str1 and str2 compare : " << str1.compare(str2) << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

str1 and str2 compare : 1

와 같이 잘 나옴을 알 수 있습니다.

이것으로, 여태까지 배운 C++ 에 대한 내용을 종합해서 훌륭한 `MyString` 클래스를 만들었다고 볼 수 있습니다. 우리의 `MyString` 클래스는 다음과 같은 인터페이스를 제공합니다.

- 문자 c 혹은 C 형식 문자열 str에서 생성할 수 있는 생성자와 복사 생성자
- 문자열의 길이를 리턴하는 함수(length)
- 문자열 대입 함수(assign)
- 문자열 메모리 할당 함수(reserve) 및 현재 할당된 크기를 알아오는 함수(capacity)
- 특정 위치의 문자를 리턴하는 함수(at)
- 특정 위치에 특정 문자열을 삽입하는 함수(insert)
- 특정 위치의 특정 개수의 문자를 지우는 함수(erase)
- 특정 위치를 시작으로 특정 문자열을 검색하는 함수(find)
- 두 문자열을 사전식 비교하는 함수(compare)

이 정도면 괜찮은 문자열 클래스라고 볼 수 있지 않나요 ㅎㅎ. 이번 강좌를 통해서 현재 까지 배운 C++ 클래스에 좀더 친숙해 질 수 있는 좋은 경험이 되었으면 합니다. 자, 그럼 이것으로 이번 강좌를 마치도록 하겠습니다.

생각해보기

문제 1

사실 위 `erase` 함수에는 한 가지 버그 있습니다. 바로 사용자가 실수로 문자열의 실제 길이 보다 더 많이 지울 때 인데요, 이 문제는 한 번 고쳐보세요. 이 버그는 김민성 님이 댓글로 제보 해주셨습니다 :) (난이도 : 하)

문제 2

여러가지 검색 알고리즘(KMP, Boyer - Moore)들을 이용하는 `find` 함수를 만들어보세요. 어떤 알고리즘의 경우 미리 계산된 테이블이 필요할 텐데, 이러한 정보들 역시 `class` 변수로 처리하셔도 됩니다. (난이도 : 상)

클래스의 `explicit` 과 `mutable` 키워드

안녕하세요 여러분! 이번 강좌는 클래스에서 비교적 자주 쓰이지는 않지만 그래도 나름 중요한 두 개의 키워드인 `explicit` 과 `mutable`에 대해 다루어 보도록 하겠습니다.

`explicit`

지난 번에 만들었던 `MyString` 클래스를 기억 하시나요? 여기에 아래와 같이 미리 크기를 할당 받는 새로운 생성자를 추가하도록 합시다.

```
#include <iostream>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length;    // 문자열 길이

    int memory_capacity;

public:
    // capacity 만큼 미리 할당함.
    MyString(int capacity);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length() const;
};

MyString::MyString(int capacity) {
    string_content = new char[capacity];
    string_length = 0;
    memory_capacity = capacity;
}

MyString::MyString(const char* str) {
    string_length = 0;
    while (str[string_length++]) {}

    string_content = new char[string_length];
    memory_capacity = string_length;
```

```

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}
MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}
MyString::~MyString() { delete[] string_content; }
int MyString::length() const { return string_length; }

int main() { MyString s(3); }

```

성공적으로 컴파일 하면

실행 결과

Capacity : 3

와 같이 나옵니다.

우리가 추가해준

```
// capacity 만큼 미리 할당함.
MyString(int capacity);
```

위 생성자가 `capacity` 를 받아서 그 만큼의 공간을 미리 할당하게 됩니다. 그렇다면 아래와 같이 `MyString` 을 인자로 받는 함수를 생각해봅시다.

```
void DoSomethingWithString(MyString s) {
    // Do something...
}
```

그렇다면 일단 아래와 같은 코드는 컴파일 될까요?

```
DoSomethingWithString(MyString("abc"))
```

당연히 되겠지요. `MyString` 객체를 생성해서 이를 인자로 전달합니다. 그렇다면 `MyString` 을 명시적으로 생성하지 않을 경우는 어떨까요?

```
DoSomethingWithString("abc")
```

일단 `DoSomethingWithString` 함수를 살펴보면 인자로 `MyString` 을 받고 있습니다. 하지만 `"abc"` 는 `MyString` 타입이 아니지요. 그런데 C++ 컴파일러는 꽤나 똑똑해서 `"abc"` 를 어떻게 하면 `MyString` 으로 바꿀 수 있는지 생각해봅니다. 그리고 다행이도 `MyString` 의 생성자들 중에서는

```
// 문자열로 부터 생성
MyString(const char* str);
```

위와 같이 `const char*` 로 부터 생성하는 것이 있었습니다. 따라서, `DoSomethingWithString ("abc")` 는 알아서

```
DoSomethingWithString(MyString("abc"))
```

로 변환되어서 컴파일 됩니다. 위와 같은 변환을 암시적 변환(**implicit conversion**) 이라고 부릅니다. 하지만 암시적 변환이 언제나 사용자에게 편리한 것은 아닙니다. 때로는 예상치 못한 경우에 암시적 변환이 일어날 수도 있습니다.

예를 들어서 아래와 같은 문장은 어떨까요?

```
DoSomethingWithString(3)
```

이는 아마도 높은 확률로 위 함수를 사용자가 잘못 사용했을 가능성이 높습니다. 왜냐하면 문자열을 받는 함수에 문자열을 전달해야지 정수 데이터를 전달하려는 일은 없기 때문이죠. 하지만 컴파일러는 위 문장을 오류로 판단하지 않습니다. 왜냐하면;

```
// capacity 만큼 미리 할당함.
MyString(int capacity);
```

위와 같이 `int` 인자를 받는 `MyString` 생성자가 있기 때문에 위 함수는

```
DoSomethingWithString(MyString(3))
```

으로 변환되어서 컴파일 됩니다. 즉, 사용자가 의도하지 않은 암시적 변환이 일어나게 됩니다.

하지만 다행이도 C++ 에서는 원하지 않는 암시적 변환을 할 수 없도록 컴파일러에게 명시할 수 있습니다. 바로 `explicit` 키워드를 통해 말이지요.

```
#include <iostream>

class MyString {
```

```

char* string_content; // 문자열 데이터를 가리키는 포인터
int string_length; // 문자열 길이

int memory_capacity;

public:
// capacity 만큼 미리 할당함. (explicit 키워드에 주목)
explicit MyString(int capacity);

// 문자열로 부터 생성
MyString(const char* str);

// 복사 생성자
MyString(const MyString& str);

~MyString();

int length() const;
int capacity() const;
};

// .. (생략) ..

void DoSomethingWithString(MyString s) {
    // Do something...
}

int main() {
    DoSomethingWithString(3); // *****
}

```

컴파일 하였다면

컴파일 오류

```

test5.cc:56:3: error: no matching function for call to
  'DoSomethingWithString'
  DoSomethingWithString(3); // *****
  ^~~~~~
test5.cc:51:6: note: candidate function not viable: no known
  conversion from 'int' to 'MyString' for 1st argument
void DoSomethingWithString(MyString s) {
  ^
1 error generated.

```

위와 같이 DoSomethingWithString(3) 부분에서 컴파일 오류가 발생함을 알 수 있습니다. 그

이유는 `int capacity` 를 인자로 받는 생성자가

```
// capacity 만큼 미리 할당함. (explicit 키워드에 주목)
explicit MyString(int capacity);
```

위와 같이 `explicit` 으로 되어 있기 때문이지요. `explicit` 은 `implicit` 의 반대말로, 명시적이라는 뜻을 가지고 있습니다.

컴파일러에서 이 `MyString` 생성자를 `explicit` 으로 선언한다면 이 생성자를 이용한 암시적 변환을 수행하지 못하게 막을 수 있습니다. 실제 컴파일 오류 메세지를 보아도, `int` 에서 `MyString` 으로 변환할 수 없다고 나옵니다.

`explicit` 은 또한 해당 생성자가 복사 생성자의 형태로도 호출되는 것을 막게 됩니다. 예를 들어서;

```
MyString s = "abc"; // MyString s("abc")
MyString s = 5; // MyString s(5)
```

`MyString(int capacity);` 에 `explicit` 이 없을 경우, 위 코드는 잘 작동합니다. 왜냐하면 컴파일러가 알아서 적당한 생성자를 골라서 호출되기 때문이지요. 하지만 생각해보면

```
MyString s = 5; // MyString s(5)
```

는 마치 `s` 에 5 를 대입하고 있다는 의미를 전달하게 됩니다. 실제로는 `capacity` 를 5 로 해주는 것인대도 말이지요. 따라서, `explicit` 으로 `MyString(int capacity)` 를 설정하면

```
MyString s(5); // 허용
MyString s = 5; // 컴파일 오류!
```

위와 같이 명시적으로 생성자를 부를 때 예만 허용할 수 있게 됩니다.

mutable

다음으로 살펴볼 키워드로 `mutable` 이 있습니다. 혹시 이전에 배우신 `const` 멤버 함수 기억하시나요? `const` 함수 내부에서는 멤버 변수들의 값을 바꾸는 것이 불가능 합니다. 하지만, 만약에 멤버 변수를 `mutable` 로 선언하였다면 `const` 함수에서도 이를 값을 바꿀 수 있습니다.⁴⁾

예를 들어 아래 예제를 간단히 보실까요.

4) `mutable` 이란 단어 뜻이 '변이 가능한' 이라 보시면 됩니다

```
#include <iostream>

class A {
    int data_;

public:
    A(int data) : data_(data) {}
    void DoSomething(int x) const {
        data_ = x; // 불가능!
    }

    void PrintData() const { std::cout << "data: " << data_ << std::endl; }
};

int main() {
    A a(10);
    a.DoSomething(3);
    a.PrintData();
}
```

컴파일 하였다면

컴파일 오류

```
test6.cc:9:11: error: cannot assign to non-static data member
→   within const member function 'DoSomething'
    data_ = x;
~~~~~ ^
test6.cc:8:8: note: member function 'A::DoSomething' is declared
→   const here
void DoSomething(int x) const {
~~~~~^~~~~~
1 error generated.
```

위와 같이 `const` 함수 안에서 멤버 변수에 값을 대입한다는 오류를 볼 수 있습니다. 하지만 `data_` 를 `mutable`로 선언하면 어떨까요.

```
#include <iostream>

class A {
    mutable int data_;

public:
    A(int data) : data_(data) {}
```

```

void DoSomething(int x) const {
    data_ = x; // 가능!
}

void PrintData() const { std::cout << "data: " << data_ << std::endl; }

int main() {
    A a(10);
    a.DoSomething(3);
    a.PrintData();
}

```

성공적으로 컴파일 하였다면

실행 결과

data: 3

위 처럼 `data_` 의 값이 `const` 함수 안에서 바뀐 것을 알 수 있습니다.

그런데 생각해보면 `mutable` 을 쓸 바에는 차라리 그냥 `DoSomething()` 에서 `const` 를 빼버리는게 낫지 않을까요? 왜 `mutable` 키워드를 만들었을까요?

그래서 `mutable` 이 왜 필요한데?

먼저 멤버 함수를 왜 `const` 로 선언하는지부터 생각해봅시다. 클래스의 멤버 함수들은 이 객체는 이러이러한 일을 할 수 있습니다 라는 의미를 나타내고 있습니다.

그리고 멤버 함수를 `const` 로 선언하는 의미는 이 함수는 객체의 내부 상태에 영향을 주지 않습니다를 표현하는 방법입니다. 대표적인 예로 읽기 작업을 수행하는 함수들을 들 수 있습니다.

대부분의 경우 의미상 상수 작업을 하는 경우, 실제로도 상수 작업을 하게 됩니다. 하지만, 실제로 꼭 그렇지만은 않습니다. 예를 들어서 아래와 같은 서버 프로그램을 만든다고 해봅시다.

```

class Server {
    // .... (생략) ....

    // 이 함수는 데이터베이스에서 user_id 에 해당하는 유저 정보를 읽어서 반환한다.
    User GetUserInfo(const int user_id) const {
        // 1. 데이터베이스에 user_id 를 검색
        Data user_data = Database.find(user_id);

        // 2. 리턴된 정보로 User 객체 생성
        return User(user_data);
    }
}

```

```
    }
};
```

이 서버에는 `GetUserInfo`라는 함수가 있는데 입력 받은 `user_id`로 데이터베이스에서 해당 유저를 조회해서 그 유저의 정보를 리턴하는 함수입니다. 당연히도 데이터베이스를 업데이트 하지도 않고, 무언가 수정하는 작업도 당연히 없기 때문에 `const` 함수로 선언되어 있습니다.

그런데 대개 데이터베이스에 요청한 후 받아오는 작업은 꽤나 오래 걸립니다. 그래서 보통 서버들의 경우 메모리에 캐쉬(cache)를 만들어서 자주 요청되는 데이터를 굳이 데이터베이스까지 가서 찾지 않아도 메모리에서 빠르게 조회할 수 있도록 합니다.

물론 캐쉬는 데이터베이스만큼 크지 않기 때문에 일부 유저들 정보 밖에 포함하지 않습니다. 따라서 캐쉬에 해당 유저가 없다면 (이를 캐쉬 미스-cache miss 라고 합니다), 데이터베이스에 직접 요청 해야겠지요. 대신 데이터베이스에서 유저 정보를 받으면 캐쉬에 저장해놓아서 다음에 요청할 때는 빠르게 받을 수 있게 됩니다.⁵⁾

이를 구현한다면 아래와 같겠지요.

```
class Server {
    // .... (생략) ....

    Cache cache; // 캐쉬!

    // 이 함수는 데이터베이스에서 user_id 에 해당하는 유저 정보를 읽어서 반환한다.
    User GetUserInfo(const int user_id) const {
        // 1. 캐쉬에서 user_id 를 검색
        Data user_data = cache.find(user_id);

        // 2. 하지만 캐쉬에 데이터가 없다면 데이터베이스에 요청
        if (!user_data) {
            user_data = Database.find(user_id);

            // 그 후 캐쉬에 user_data 등록
            cache.update(user_id, user_data); // <-- 불가능
        }

        // 3. 리턴된 정보로 User 객체 생성
        return User(user_data);
    }
};
```

그런데 문제는 `GetUserInfo`가 `const` 함수라는 점입니다. 따라서

5) 보통 한 번 요청된 정보는 계속해서 요청될 확률이 높기 때문에 캐쉬에 넣게 됩니다. 물론 캐쉬 크기는 한정적이니까 이전에 오래된 캐쉬부터 지우게 됩니다.

```
cache.update(user_id, user_data); // <-- 불가능
```

위 처럼 캐쉬를 업데이트 하는 작업을 수행할 수 없습니다. 왜냐하면 캐쉬를 업데이트 한다는 것은 캐쉬 내부의 정보를 바꿔야 된다는 뜻이기 때문이죠. 따라서 저 update 함수는 `const` 함수가 아닐 것입니다.

그렇다고 해서 `GetUserInfo`에서 `const` 를 빼기도 좀 뭐한것이, 이 함수를 사용하는 사용자의 입장에선 당연히 `const` 로 정의되어야 하는 함수 이기 때문이지요. 따라서 이 경우, `Cache` 를 `mutable` 로 선언해버리면 됩니다.

```
mutable Cache cache; // 캐쉬!
```

위 처럼 말이지요. 이렇듯, `mutable` 키워드는 `const` 함수 안에서 해당 멤버 변수에 `const` 가 아닌 작업을 할 수 있게 만들어줍니다.

그렇다면 이것으로 클래스의 `explicit` 과 `mutable` 키워드들에 대해 알아보았습니다. 다음 시간에는 내가 만든 클래스에 연산자를 사용할 수 있게 해주는 연산자 오버로딩에 대해 배울 것입니다.

연산자 오버로딩(overloading)

안녕하세요 여러분! 지난 강좌에서 만들었던 `MyString` 을 손 좀 봐주었나요? 아마도 `MyString` 을 이용하여 여러가지 작업을 하면서 다음과 같은 생각을 하셨을 수도 있었을 것입니다.

- `if(str1.compare(str2) == 0)` 하지 말고 `if(str1 == str2)` 하면 어떨까?
- `str1.insert(str1.length(), str2)` 하지 말고 `str1 = str1 + str2;` 하면 어떨까?
- `str1[10] = 'c';` 와 같은 것도 할 수 있을까?

물론 C 언어에서는 이러한 것을 상상조차 할 수 없었습니다. `+`, `-`, `==`, `[]` 와 같은 기본 연산자들은 모두 C 언어에 기본적으로 정의되어 있는 데이터 타입(`int`, `float` 등)에만 사용 가능한 것들 이였기 때문이죠. 이들을 구조체 변수에 사용한다는 것은 불가능하였습니다.

하지만 놀랍게도 C++ 에서는 사용자 정의 연산자를 사용할 수 있습니다. 어떠한 연산자들이 가능하나면, `::` (범위 지정), `.` (멤버 지정), `.*` (멤버 포인터로 멤버 지정) 을 제외한 여러분이 상상하는 모든 연산자를 사용할 수 있다는 것입니다. 대표적으로

- `+`, `-`, `*` 와 같은 산술 연산자
- `+=`, `-=` 와 같은 축약형 연산자
- `>=`, `==` 와 같은 비교 연산자
- `&&`, `||` 와 같은 논리 연산자
- `->` 나 `*` 와 같은 멤버 선택 연산자 (여기서 `*` 는 역참조 연산자입니다. 포인터에서 `*p` 할 때처럼)
- `++`, `--` 증감 연산자
- `[]` (배열 연산자) 와 심지어 `()` 까지 (함수 호출 연산자)

까지 모두 여러분이 직접 만들 수 있습니다.

이 때 이러한 기본 연산자들을 직접 사용자가 정의하는 것을 연산자를 **오버로딩(overloading)**한다고 부릅니다. 이전에 같은 이름의 함수를 인자만 다르게 사용하는 것을 '함수를 오버로딩 했다'라고 불렀던 것 처럼, 기본 연산자를 여러분이 설계한 클래스에 맞게 직접 사용하는 것을 '연산자를 오버로딩 했다'라고 부릅니다.

MyString 의 == 연산자 오버로딩

일단 연산자 오버로딩을 사용하기 위해서는, 다음과 같이 오버로딩을 원하는 연산자 함수를 제작하면 됩니다.

(리턴 타입) operator(연산자) (연산자가 받는 인자)

(※ 참고적으로 위 방법 외에는 함수 이름으로 연산자를 절대 넣을 수 없습니다) 예를 들어서 우리가 == 를 오버로딩 하고 싶다면, == 연산자는 그 결과값이 언제나 bool 이고, 인자로는 == 로 비교하는 것 하나만 받게 됩니다. 따라서 다음과 같이 함수를 정의하면 됩니다.

```
bool operator==(MyString& str);
```

이제, 우리가 str1 == str2 라는 명령을 한다면 이는 str1.operator==(str2) 로 내부적으로 변환되서 처리됩니다. 그리고 그 결과값을 리턴하게 되겠지요. 사실 operator== 를 만드는 것 자체는 별로 어려운 일은 아닙니다. 왜냐하면 이미 MyString 에서 compare 라는 좋은 함수를 제공하고 있기 때문이지요. 간단하게 만들어 보면 다음과 같습니다.

```
bool MyString::operator==(MyString& str) {
    return !compare(str); // str 과 같으면 compare에서 0 을 리턴한다.
}
```

여기서 !compare(str) 을 리턴하는 이유는 compare 함수에서 str 과 *this 가 같으면 0 을 리턴하도록 하였는데, operator== 은 둘이 같으면 true 를 리턴해야 되기 때문입니다. 따라서 NOT 연산자인 ! 를 앞에 붙여서 리턴하면 올바르게 작동할 수 있습니다. 그럼, 실제로 우리의 새롭게 오버로딩한 == 연산자가 잘 작동하는지 살펴봅시다.

```
#include <string.h>
#include <iostream>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이

    int memory_capacity; // 현재 할당된 용량
```

```
public:  
    // 문자 하나로 생성  
    MyString(char c);  
  
    // 문자열로 부터 생성  
    MyString(const char* str);  
  
    // 복사 생성자  
    MyString(const MyString& str);  
  
    ~MyString();  
  
    int length() const;  
    int capacity() const;  
    void reserve(int size);  
  
    void print() const;  
    void println() const;  
  
    char at(int i) const;  
  
    int compare(MyString& str);  
    bool operator==(MyString& str);  
};  
  
MyString::MyString(char c) {  
    string_content = new char[1];  
    string_content[0] = c;  
    memory_capacity = 1;  
    string_length = 1;  
}  
MyString::MyString(const char* str) {  
    string_length = strlen(str);  
    memory_capacity = string_length;  
    string_content = new char[string_length];  
  
    for (int i = 0; i != string_length; i++) string_content[i] = str[i];  
}  
MyString::MyString(const MyString& str) {  
    string_length = str.string_length;  
    string_content = new char[string_length];  
  
    for (int i = 0; i != string_length; i++)  
        string_content[i] = str.string_content[i];  
}  
MyString::~MyString() { delete[] string_content; }  
int MyString::length() const { return string_length; }  
void MyString::print() const {  
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];  
}
```

```
void MyString::println() const {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];

    std::cout << std::endl;
}

int MyString::capacity() const { return memory_capacity; }

void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char* prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];

        delete[] prev_string_content;
    }

    // 만일 예약하려는 size 가 현재 capacity 보다 작다면
    // 아무것도 안해도 된다.
}

char MyString::at(int i) const {
    if (i >= string_length || i < 0)
        return 0;
    else
        return string_content[i];
}

int MyString::compare(MyString& str) {
    // (*this) - (str) 을 수행해서 그 1, 0, -1 로 그 결과를 리턴한다
    // 1 은 (*this) 가 사전식으로 더 뒤에 온다는 의미. 0 은 두 문자열
    // 이 같다는 의미, -1 은 (*this) 사 사전식으로 더 앞에 온다는 의미이다.

    for (int i = 0; i < std::min(string_length, str.string_length); i++) {
        if (string_content[i] > str.string_content[i])
            return 1;

        else if (string_content[i] < str.string_content[i])
            return -1;
    }

    // 여기 까지 했는데 끝나지 않았다면 앞 부분 까지 모두 똑같은 것이 된다.
    // 만일 문자열 길이가 같다면 두 문자열은 아예 같은 문자열이 된다.

    if (string_length == str.string_length) return 0;

    // 참고로 abc 와 abcd 의 크기 비교는 abcd 가 더 뒤에 오게 된다.
    else if (string_length > str.string_length)
        return 1;

    return -1;
}
```

```

}

bool MyString::operator==(MyString& str) {
    return !compare(str); // str 과 같으면 compare 에서 0 을 리턴한다.
}

int main() {
    MyString str1("a word");
    MyString str2("sentence");
    MyString str3("sentence");

    if (str1 == str2)
        std::cout << "str1 와 str2 같다." << std::endl;
    else
        std::cout << "str1 와 str2 는 다르다." << std::endl;

    if (str2 == str3)
        std::cout << "str2 와 str3 는 같다." << std::endl;
    else
        std::cout << "str2 와 str3 는 다르다" << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

str1 와 str2 는 다르다.
str2 와 str3 는 같다.

```

와 같이 잘 나옵니다. 위 코드에서도 쉽게 알 수 있지만 str1 과 str2 은 다르고, str2 와 str3 는 같기 때문에 위와 같이 제대로 처리되고 있음을 알 수 있습니다.

복소수 (Complex number) 클래스 만들기

MyString 클래스를 이용해서 설명을 계속 하려고 했지만, MyString 자체가 너무 비대한 바람에 좀더 간결하게 설명을 하기 위해 새로운 클래스를 만들어보도록 할 것입니다. 바로 복소수(Complex Number)를 다루는 클래스입니다.

주의 사항

참고로 C++ 표준 라이브러리에 `std::complex` 가 정의되어 있어서 굳이 여러분이 만들어서 쓸 필요는 없습니다. 복소수 클래스를 제작하는 이유는 순전히 교육용입니다.

복소수가 정확히 무엇인지 모르시는 분들을 위해서 간략하게 먼저 설명을 하고 가겠습니다.

$$i = \sqrt{-1}$$

일단 실수의 제곱근에 대해서는 무엇인지 다들 아실 것이라 생각합니다. 그런데 실수의 제곱은 언제나 양수이기 때문에 위와 같이 음수의 제곱근은 실수로 표현할 수 없게 됩니다. 따라서, 음수의 제곱근을 나타내기 위해서 특별한 수를 정의하였는데 이를 허수(imaginary number) 이라 부르며, 실제로 존재하지 않는 수학적으로만 존재하는 수라고 볼 수 있습니다. 그리고 특히 -1의 제곱근을 위의 수식처럼 i 로 표기합니다.

따라서, 이 때문에 예를 들어서 -4의 제곱근은!

$$2i = \sqrt{-4}$$

와 같이 생각할 수 있겠지요. 그리고 복소수는, 이 허수와 실수를 모두 포함하는 수 체계로, 허수와 실수의 합으로 표현할 수 있습니다. 다시 말해서, 임의의 복소수 z 는 다음과 같은 꼴입니다.

$$z = a + bi$$

물론 여기서 a, b 는 모두 실수입니다.

우리가 만들고자 하는 것은 이 복소수를 나타내는 클래스를 구성하겠다는 이야기입니다. 임의의 복소수 하나를 표현하기 위해서 두 개의 값(실수부, 허수부)이 필요하기 때문에 반드시 클래스로 구현을 해야 하겠죠. 따라서, 기본적으로 복소수 클래스 `Complex` 는 다음과 같이 간단하게 만들 수 있습니다.

```
class Complex {
    private:
        double real, img;

    public:
        Complex(double real, double img) : real(real), img(img) {}
};
```

복소수는 언제나 실수부와 허수부로 나뉘어지므로, `Complex` 클래스 역시 실수부의 값과 허수부의 값을 나타내는 `real` 과 `img` 변수가 있습니다. 여기서 문제는 이전에 만들었던 `MyString` 과는 다르게, 사칙 연산이 엄청나게 자주 쓰인다는 것입니다. 당연하게도 문자열의 덧셈 (+ 연산) 까지는 생각할 수 있었다 해도, 곱셈이나 나눗셈 연산 자체는 고려할 필요가 없는데, 복소수의 경우 당연히 클래스 인터페이스 차원에서 곱셈과 나눗셈을 지원해주어야만 합니다.

참고로, 복소수의 사칙 연산은 실수부와 허수부 따로 생각하여 진행됩니다. 간단히 말하면

$$z_1 + z_2 = (a_1 + ib_1) + (a_2 + ib_2) = (a_1 + a_2) + i(b_1 + b_2)$$

와 같은 관계가 성립한다는 이야기 이죠. 곱셈의 경우는 좀 더 복잡한데, 사실 분배법칙과 허수 둘을 곱하면 다시 -1 이 된다는 점만 생각하면 아래의 관계식도 어렵지 않게 생각할 수 있습니다.

$$z_1 z_2 = (a_1 + ib_1)(a_2 + ib_2) = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + a_2 b_1)$$

나눗셈의 경우, 이전에 무리수의 유리화를 하셨던 것을 생각하면 간단합니다.

$$\frac{z_1}{z_2} = \frac{a_1 + ib_1}{a_2 + ib_2} = \frac{(a_1 + ib_1)(a_2 - ib_2)}{a_2^2 + b_2^2} = \frac{a_1 a_2 + b_1 b_2 + i(a_2 b_1 - a_1 b_2)}{a_2^2 + b_2^2}$$

즉 분모를 실수화 할 수 있도록 분모의 결례를 문자와 분모에 곱함으로써 실수로 나누는 것으로 쉽게 바꿀 수 있습니다. (실수로 나누는 것은 실수부, 허수부의 실수값을 그냥 실수로 나누면 됩니다)

그래서 만일 다음과 같이 연산자의 오버로딩을 모른다고 가정하고 Complex 클래스를 구성하여 봅시다.

```
class Complex {
private:
    double real, img;

public:
    Complex(double real, double img) : real(real), img(img) {}

    Complex plus(const Complex& c);
    Complex minus(const Complex& c);
    Complex times(const Complex& c);
    Complex divide(const Complex& c);
};
```

이렇게 된다면 만일 int 형 변수였다면

```
a + b / c + d;
```

로 간단하게 쓸 수 있었던 명령을

```
a.plus(b.divide(c)).plus(d);
```

와 같이 복잡한 함수식을 이용해서 표현해야만 합니다. 이는, 가독성이 떨어질 뿐더러 위 식을 딱 보고 도대체 무슨 작업을 하려고 하는지도 쉽게 알 수 없습니다.

하지만 연산자 오버로딩을 이용해서 plus 를 operator+ 로, divide 를 operator/ 로, 등등 바꿔준다면 단순히 프로그래머가 a + b/c + d; 게 쓴다고 해도, 컴파일러가 a.operator+(b.operator/(c)).

로 알아서 변환시켜서 처리하기 때문에 속도나 다른 면의 어떠한 차이 없이 뛰어난 가독성과 편리함을 얻을 수 있게 됩니다.

이를 바탕으로 간단히 Complex 클래스를 만들어본다면

```
#include <iostream>

class Complex {
private:
    double real, img;

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    void println() { std::cout << "(" << real << ", " << img << ")" <<
        std::endl; }
};

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}
Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}
Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}
Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}

int main() {
    Complex a(1.0, 2.0);
    Complex b(3.0, -2.0);

    Complex c = a * b;

    c.println();
```

```
}
```

성공적으로 컴파일 하였다면

실행 결과

```
( 7 , 4 )
```

와 같이 잘 나옵을 알 수 있습니다. 여기서 가장 중요하게 봐야 할 부분은 바로, 사칙연산 연산자 함수들의 리턴 타입입니다.

```
Complex operator+(const Complex& c);
Complex operator-(const Complex& c);
Complex operator*(const Complex& c);
Complex operator/(const Complex& c);
```

위 4 개의 연산자 함수 모두 `Complex&` 가 아닌 `Complex` 를 리턴하고 있습니다. 간혹가다,

```
Complex& operator+(const Complex& c) {
    real += c.real;
    img += c.img;
    return *this;
}
```

로 잘못 생각하는 경우도 있습니다. 물론 이렇게 설계하였을 경우, `Complex` 를 리턴하는 연산자 함수는 값의 복사가 일어나기 때문에 속도 저하가 발생하지만 위처럼 레퍼런스를 리턴하게 되면 값의 복사 대신 레퍼런스만 복사하는 것이므로 큰 속도의 저하는 나타나지 않습니다. 하지만, 위와 같이 `operator+` 를 정의할 경우 다음과 같은 문장이 어떻게 처리되는지 생각해봅시다.

```
Complex a = b + c + b;
```

아마도 위 문장을 쓴 사람 입장에서는 결과적으로 `a = 2 * b + c;` 를 의도하였을 것입니다.

하지만, 실제로 처리되는 것을 보자면, `(b.plus(c)).plus(b)` 가 되는데, `b.plus(c)` 를 하면서 `b` 에는 `(b + c)` 가 들어가고, 거기에 다시 `plus(b)` 를 하게 된다면 값 자체만 보자면 `(b + c) + (b + c)` 가 되서 (왜냐하면 현재 `b` 에는 `b + c` 가 들어가 있으니까) `a = 2 * b + 2 * c` 가 되기 때문입니다. 이러한 문제를 막기 위해서는 반드시 사칙 연산의 경우 반드시 값을 리턴해야 만 합니다.

또한 함수 내부에서 읽기만 수행되고 값이 바뀌지 않는 인자들에 대해서는 `const` 키워드를 붙여주는 것이 바람직합니다. `operator+` 의 경우, `c` 의 값을 읽기만 하지 `c` 의 값을 어떤 변화도 주지 않으므로 `const Complex&` 타입으로 인자를 받았습니다.

주의 사항

인자의 값이 함수 내부에서 바뀌지 않는다고 확신할 때에는 `const` 키워드를 붙여주시기 바랍니다. 이는 나중에 발생할 수 있는 실수들을 줄여줍니다.

대입 연산자 함수

아마 `Complex` 클래스를 구현하면서 한 가지 빼뜨렸다고 생각하고 있는 것이 있을 것입니다. 바로, 대입 연산자 (=) 이지요. 아마도, 대입 연산자야 말로 가장 먼저 구현했어야 할 연산자 함수가 아니였을까 합니다.

```
Complex& operator=(const Complex& c);
```

기본적으로 대입 연산자 함수는, 기존의 사칙연산 연산자 함수와는 다르게, `Complex&` 타입을 리턴합니다. 사실 대입 연산자 자체의 의미를 생각해 볼 때 리턴값을 `void`로 해도 무방하지만, 프로그래머들은 종종 `if((i = x) < y)` 와 같은 문장을 사용하기 때문에 리턴값을 주는 것이 인터페이스 차원에서 더 낫다고 생각합니다.

이 때 `Complex` 타입을 리턴하지 않고 굳이 `Complex&` 타입을 리턴하냐면, 대입 연산 이후에 이 값을 가지고 다른 연산을 수행하지는 않기 때문입니다. 예를 들어서 `(i = 3) + 4` 와 같은 명령을 내리지는 않기 때문이지요. 그렇기에 값에 의한 복사가 발생하는 것 보다는 레퍼런스를 리턴하는 것이 더 올바른 판단이라고 봅니다.

이와 같은 사실을 바탕으로 `operator=` 함수를 완성시켜 보면 아래와 같습니다.

```
Complex& Complex::operator=(const Complex& c)
{
    real = c.real;
    img = c.img;
    return *this;
}
```

그럼 제대로 작동하는지 확인해보면

```
#include <iostream>

class Complex {
private:
    double real, img;
```

```
public:  
    Complex(double real, double img) : real(real), img(img) {}  
    Complex(const Complex& c) { real = c.real, img = c.img; }  
  
    Complex operator+(const Complex& c);  
    Complex operator-(const Complex& c);  
    Complex operator*(const Complex& c);  
    Complex operator/(const Complex& c);  
  
    Complex& operator=(const Complex& c);  
    void println() { std::cout << "(" << real << " , " << img << " ) " <<  
        std::endl; }  
};  
  
Complex Complex::operator+(const Complex& c) {  
    Complex temp(real + c.real, img + c.img);  
    return temp;  
}  
Complex Complex::operator-(const Complex& c) {  
    Complex temp(real - c.real, img - c.img);  
    return temp;  
}  
Complex Complex::operator*(const Complex& c) {  
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);  
    return temp;  
}  
Complex Complex::operator/(const Complex& c) {  
    Complex temp(  
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),  
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));  
    return temp;  
}  
Complex& Complex::operator=(const Complex& c) {  
    real = c.real;  
    img = c.img;  
    return *this;  
}  
  
int main() {  
    Complex a(1.0, 2.0);  
    Complex b(3.0, -2.0);  
    Complex c(0.0, 0.0);  
    c = a * b + a / b + a + b;  
    c.println();  
}
```

성공적으로 컴파일 하였다면

실행 결과

(10.9231 , 4.61538)

와 같이 잘 작동함을 알 수 있습니다.

한 가지 재미있는 사실은 굳이 `operator=` 를 만들지 않더라도, 위 소스를 컴파일 하면 잘 작동한다는 점입니다. 이는 컴파일러 차원에서 디폴트 대입 연산자(default assignment operator)를 지원하고 있기 때문인데, 이전에 [복사 생성자를 다룰 때 디폴트 복사 생성자](#)가 있었던 것과 일맥상통합니다.

디폴트 복사 생성자와 마찬가지로 디폴트 대입 연산자 역시 얇은 복사를 수행합니다. 따라서, 깊은 복사가 필요한 클래스의 경우 (예를 들어, 클래스 내부적으로 동적으로 할당되는 메모리를 관리하는 포인터가 있다던지) 대입 연산자 함수를 꼭 만들어주어야 할 필요가 있습니다.

여담이지만, 이제 여러분은 다음 두 문장의 차이를 완벽히 이해 하실 수 있을 것이라 믿습니다.

```
Some_Class a = b; // ①
```

와

```
Some_Class a; // ②
a = b;
```

말이지요. 이전에 이에 대해서 이야기 하였을 때 연산자 오버로딩을 배우지 못하였기 때문에 대충 두루뭉실하게 넘어갔지만 이제는 제대로 이해할 수 있습니다. ①의 경우, 아예 `a` 의 '복사 생성자' 가 호출되는 것이고, ②의 경우 `a` 의 그냥 기본 생성자가 호출 된 다음, 다음 문장에서 대입 연산자 함수가 실행되는 것입니다. 위 두 문장은 비록 비슷해 보이기는 해도 아예 다른 것이지요.

마찬가지 이유로 대입 사칙연산 함수들인, `+=`, `-=` 등을 구현할 수 있습니다. 일단 `=` 와 마찬가지로 아래와 같이 `Complex&` 를 리턴하고

```
Complex& operator+=(const Complex& c);
Complex& operator-=(const Complex& c);
Complex& operator*=(const Complex& c);
Complex& operator/=(const Complex& c);
```

그 내부 구현은 간단히 미리 만들어 놓은 `operator+`, `operator-` 등을 이용해서 처리하면 됩니다.

```
Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
```

```

}
Complex& Complex::operator=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}
Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}
Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}

```

와 같아 말이지요. 전체 소스를 살펴보자면;

```

#include <iostream>

class Complex {
private:
    double real, img;

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex& operator/=(const Complex& c);

    void println() {
        std::cout << "(" << real << ", " << img << ")" << std::endl;
    }
};

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}
Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}
Complex Complex::operator*(const Complex& c) {

```

```

Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}
Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}
Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}
Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}
Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}
Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}
int main() {
    Complex a(1.0, 2.0);
    Complex b(3.0, -2.0);
    a += b;
    a.println();
    b.println();
}

```

성공적으로 컴파일 하였다면

실행 결과

```
( 4 , 0 )
( 3 , -2 )
```

와 같이 잘 출력됨을 알 수 있습니다. a의 값만 바뀐 채 b에는 아무런 영향이 없지요.

참고로, 연산자 오버로딩을 사용하게 된다면 $a+= b$ 와 $a = a + b;$ 가 같다고 보장되지 않는다는 점을 명심해야 합니다. 컴파일러는 $\text{operator}+$ 와 $\text{operator}=$ 를 정의해놓았다고 해서 $a+=b$ 를 자동으로 $a = a + b;$ 로 바꾸어 주지 않습니다. 반드시 $\text{operator}+ =$ 를 따로 만들어야지 $+ =$ 를 사용할 수 있게 됩니다. 이와 같은 사실은 $++$ 을 $+= 1$ 로 바꾸어 주지 않는다면지, $--$ 를 $-= 1$

로 바꾸어 주지 않는다는 사실과 일맥상통합니다. 즉, 연산자 오버로딩을 하게 된다면 여러분이 생각하는 모든 연산자들에 대해 개별적인 정의가 필요합니다.

문자열로 Complex 수와 덧셈하기

이번에는 `operator+` 를 개량해서, 꼭 Complex 수를 더하는 것이 아니라, 문자열로도 덧셈을 할 수 있도록 `operator+` 함수를 만드려 보려고 합니다. 다시 말해서,

```
y = z + "3+i2";
```

이런 문장을 사용하였을 경우 성공적으로 처리할 수 있게 된다는 의미이지요. 참고로, 문자열로 복소수를 어떻게 표현해야 할지에 대해서는 모종의 약속이 필요한데, 우리 Complex 클래스의 경우 다음과 같은 꼴로 표현하도록 정합시다.

(부호) (실수부) (부호) i (허수부)

예를 들어서 "2+i3" 은 Complex 수 (2, 3) 을 나타낸 것이라 생각합니다. 또한, "2-i3" 은 (2, -3) 을 나타낸 것이 되겠지요.

만일 실수부나 허수부의 값이 0 이라면 굳이 안써주어도 되는데, 예를 들어서 그냥 "3" 은 (3, 0) 을 나타내며, "-5i" 는 (0, -5) 를 나타내게 됩니다. 참고로 우리의 실수부와 허수부는 `double` 변수이기 때문에 문자열로 입력 받을 때 단순히 정수 부분만 받는 것이 아니라 소수점 아래 부분도 처리해 주어야만 할 것입니다. 이를 바탕으로 `operator+` 함수를 만들어 보도록 합시다.

```
Complex Complex::operator+(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)

    int begin = 0, end = strlen(str);
    double str_img = 0.0, str_real = 0.0;

    // 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
    int pos_i = -1;
    for (int i = 0; i != end; i++) {
        if (str[i] == 'i') {
            pos_i = i;
            break;
        }
    }

    // 만일 'i' 가 없다면 이 수는 실수 뿐이다.
    if (pos_i == -1) {
```

```

str_real = get_number(str, begin, end - 1);

Complex temp(str_real, str_img);
return (*this) + temp;
}

// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
str_real = get_number(str, begin, pos_i - 1);
str_img = get_number(str, pos_i + 1, end - 1);

if (pos_i >= 1 && str[pos_i - 1] == '-') str_img *= -1.0;

Complex temp(str_real, str_img);
return (*this) + temp;
}

```

일단 문자열을 덧셈의 피연산자로 사용하게 되므로, `operator+` 의 인자는 `Complex &` 가 아니라 `const char *` 가 됩니다. 저의 경우, 이제 입력 받은 '문자열 복소수' 를 분석하기 위해서 가장 중요한 'i' 의 위치를 먼저 찾도록 하였습니다. 왜냐하면 이 'i' 를 기준으로 복소수의 실수부와 허수부가 나뉘어지기 때문이지요.

```

// 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
int pos_i = -1;
for (int i = 0; i != end; i++) {
    if (str[i] == 'i') {
        pos_i = i;
        break;
    }
}

```

따라서 위와 같이 `pos_i` 에 `str` 의 'i' 의 위치를 찾도록 하였습니다. 물론, 입력 받은 문자열에 반드시 'i' 가 있어야만 하는 것은 아닙니다. 왜냐하면 이전에도 말했듯이 복소수가 그냥 실수라면 굳이 허수 부분을 표현하지 않을 수 있기 때문입니다. 따라서, 아래와 같이 i 가 없을 경우 간단히 따로 처리할 수 있습니다.

```

// 만일 'i' 가 없다면 이 수는 실수 뿐이다.
if (pos_i == -1) {
    str_real = get_number(str, begin, end - 1);

    Complex temp(str_real, str_img);
    return (*this) + temp;
}

```

참고로 우리가 사용하는 `get_number` 함수는 특정 문자열에서 수 부분을 `double` 값으로 반환하는 함수입니다.

사실 C 언어 표준 라이브러리인 `stdlib.h`에서 `atof`라는 함수를 제공해서 우리의 `get_number` 함수와 정확히 똑같은 작업을 하는 함수를 사용할 수 있지만, 한 번 이 함수를 직접 만들어보는 것도 나쁘지 않을 것이라 생각해서 `Complex` 클래스 내의 멤버 함수로 포함시켰습니다.

다만, 이 `get_number`의 경우 `operator+` 함수의 내부적으로 사용되는 함수이지, 굳이 인터페이스로 제공할 필요는 없기 때문에 `private`으로 설정하였습니다.

```
// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
str_real = get_number(str, begin, pos_i - 1);
str_img = get_number(str, pos_i + 1, end - 1);
```

자 이제, 다시 `operator+` 함수를 돌아와서 살펴보자면 만일 `i`가 포함되어 있다면 `i`를 기준으로 왼쪽의 실수부와 오른쪽의 허수부로 나뉘게 됩니다. 이 때 `str_real`은 `get_number` 함수를 이용해서 정확히 실수 값을 얻을 수 있습니다. (왜냐하면 맨 뒤에 숫자 뒤에 딸려오는 문자들은 `get_number`에서 알아서 무시된다) 하지만 `str_img`의 경우 `i` 앞의 부호 부분이 잘리기 때문에 정확한 실수 값을 얻을 수 없기 때문에 따로

```
if (pos_i >= 1 && str[pos_i - 1] == '-') str_img *= -1.0;
```

로 해서 `str_img`의 정확한 부호를 처리하도록 하였습니다.

```
double Complex::get_number(const char *str, int from, int to) {
    bool minus = false;
    if (from > to) return 0;

    if (str[from] == '-') minus = true;
    if (str[from] == '-' || str[from] == '+') from++;

    double num = 0.0;
    double decimal = 1.0;

    bool integer_part = true;
    for (int i = from; i <= to; i++) {
        if (isdigit(str[i]) && integer_part) {
            num *= 10.0;
            num += (str[i] - '0');
        } else if (str[i] == '.')
            integer_part = false;
        else if (isdigit(str[i]) && !integer_part) {
            decimal /= 10.0;
            num += ((str[i] - '0') * decimal);
        } else
            break; // 그 이외의 이상한 문자들이 올 경우
    }
}
```

```

if (minus) num *= -1.0;

return num;
}

```

저의 경우 `get_number` 함수를 위와 같이 구현하였습니다. 만일 `from` 이 `to` 보다 크다면 당연히, 올바르지 않는 입력으로 0 을 반환하도록 하였습니다. (사실 이렇게 모든 예외적인 경우를 세세하게 처리하는 일도 매우 중요합니다)

그리고, 특별히 부호를 처리하기 위해서 `minus` 라는 `bool` 변수를 도입해서 마지막에 `minus` 가 `true` 일 경우에 부호를 음수로 바꾸도록 하였습니다.

```

if (str[from] == '-' || str[from] == '+') from++;

```

일단 부호 부분은 위와 같이 처리해서 부호 부분 바로 다음 부터 처리하도록 합니다.

```

for (int i = from; i <= to; i++) {
    if (isdigit(str[i]) && integer_part) {
        num *= 10.0;
        num += (str[i] - '0');
    } else if (str[i] == '.')
        integer_part = false;
    else if (isdigit(str[i]) && !integer_part) {
        decimal /= 10.0;
        num += ((str[i] - '0') * decimal);
    } else
        break; // 그 이외의 이상한 문자들이 올 경우
}

```

`double` 형 변수로 입력받은 문자열을 처리할 때 유의할 점은, `for` 문에서 맨 앞자리 수 부터 읽는다는 점입니다. 예를 들어서 123.456 이라면 1, 2, 3... 순으로 값을 입력 받게 되는데 이 때문에 소수점 앞 부분과 뒷 부분의 처리를 다르게 해야만 합니다. 소수점 앞 부분을 입력받을 때 (즉, `integer_part` 변수가 `true` 일 때) 에는 간단히

```

num *= 10.0;
num += (str[i] - '0');

```

를 해서 문자열 부분의 값을 읽어들일 수 있습니다. 즉 1 → 12 → 123 이 되겠지요. 참고로 `str[i] - '0'` 을 하는 기법은 상당히 자주 쓰이는데, ASCII 테이블 상에서 0 부터 9 까지 숫자들이 크기 순으로 연속적으로 배열되어 있기 때문에 단순히 '0' 을 빼버리면 그 숫자에 해당하는 실제 정수 값을 구할 수 있게 됩니다.

```

else if (isdigit(str[i]) && !integer_part) {
    decimal /= 10.0;
    num += ((str[i] - '0') * decimal);
}

```

그리고 이번에는 소수점 뒷 부분을 읽어들일 차례입니다. 소수점 뒷 부분의 경우 decimal 이란 새로운 변수를 도입하여서, 현재 읽어들이는 위치에 해당하는 값을 구할 수 있게 되는데요, 예를 들어 123.456에서 4의 경우 decimal은 0.1, 5는 0.01 등이 되겠지요. 이와 같은 방식으로 해서 우리는 원래의 문자열을 double 값으로 바꿀 수 있게 됩니다.

```

#include <iostream>
#include <cstring>

class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }

    Complex operator+(const Complex& c);
    Complex operator+(const char* str);

    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex& operator/=(const Complex& c);

    Complex& operator=(const Complex& c);

    void println() { std::cout << "(" << real << " , " << img << " ) " <<
        std::endl; }
};

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}

double Complex::get_number(const char* str, int from, int to) {
    bool minus = false;
    if (from > to) return 0;
}

```

```
if (str[from] == '-') minus = true;
if (str[from] == '-' || str[from] == '+') from++;

double num = 0.0;
double decimal = 1.0;

bool integer_part = true;
for (int i = from; i <= to; i++) {
    if (isdigit(str[i]) && integer_part) {
        num *= 10.0;
        num += (str[i] - '0');
    } else if (str[i] == '.')
        integer_part = false;
    else if (isdigit(str[i]) && !integer_part) {
        decimal /= 10.0;
        num += ((str[i] - '0') * decimal);
    } else
        break; // 그 이외의 이상한 문자들이 올 경우
}

if (minus) num *= -1.0;

return num;
}

Complex Complex::operator+(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)

    int begin = 0, end = strlen(str);
    double str_img = 0.0, str_real = 0.0;

    // 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
    int pos_i = -1;
    for (int i = 0; i != end; i++) {
        if (str[i] == 'i') {
            pos_i = i;
            break;
        }
    }

    // 만일 'i' 가 없다면 이 수는 실수 뿐이다.
    if (pos_i == -1) {
        str_real = get_number(str, begin, end - 1);

        Complex temp(str_real, str_img);
        return (*this) + temp;
    }

    // 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
```

```
str_real = get_number(str, begin, pos_i - 1);
str_img = get_number(str, pos_i + 1, end - 1);

if (pos_i >= 1 && str[pos_i - 1] == '-') str_img *= -1.0;

Complex temp(str_real, str_img);
return (*this) + temp;
}

Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}

Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}

Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}

Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}

Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}

Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}

Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}

Complex& Complex::operator=(const Complex& c) {
    real = c.real;
    img = c.img;
    return *this;
}

int main() {
    Complex a(0, 0);
    a = a + "-1.1 + i3.923";
    a.println();
}
```

성공적으로 컴파일 하였다면

실행 결과

(-1.1 , 3.923)

와 같이 잘 실행되는 것을 알 수 있습니다.

그런데, + 뿐만이 아니라, -, * 등의 모든 연산자들에 대해 이 기능을 지원하기 위해서 각각의 코드를 반복적으로 쓰는 것은 매우 귀찮은 일이 아닐 수 없습니다. 이와 같은 완전 불편한 작업을 막기 위해 아예 `const char *`로 오버로딩되는 `Complex` 생성자를 추가하는 것도 나쁘지 않다고 생각 됩니다.

그렇게 된다면 길고 복잡했던 `operator+` (`const char * str`) 부분을 다음과 같이 간단하게 줄일 수 있기 때문이지요.

```
Complex Complex::operator+(const char* str) {
    Complex temp(str);
    return (*this) + temp;
}
```

그렇다면, 간단히 `Complex(const char* str)` 을 만들어본다면 아래와 같습니다.

```
Complex::Complex(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)

    int begin = 0, end = strlen(str);
    img = 0.0;
    real = 0.0;

    // 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
    int pos_i = -1;
    for (int i = 0; i != end; i++) {
        if (str[i] == 'i') {
            pos_i = i;
            break;
        }
    }

    // 만일 'i' 가 없다면 이 수는 실수 뿐이다.
    if (pos_i == -1) {
        real = get_number(str, begin, end - 1);
        return;
    }
}
```

```
// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
real = get_number(str, begin, pos_i - 1);
img = get_number(str, pos_i + 1, end - 1);

if (pos_i >= 1 && str[pos_i - 1] == '-') img *= -1.0;
}
```

그렇게 된다면, 나머지 함수들도,

```
Complex Complex::operator-(const char* str) {
    Complex temp(str);
    return (*this) - temp;
}

Complex Complex::operator*(const char* str) {
    Complex temp(str);
    return (*this) * temp;
}

Complex Complex::operator/(const char* str) {
    Complex temp(str);
    return (*this) / temp;
}
```

로 간단하게 구현할 수 있게 됩니다.

```
#include <cstring>
#include <iostream>

class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }
    Complex(const char* str);

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex operator+(const char* str);
    Complex operator-(const char* str);
    Complex operator*(const char* str);
    Complex operator/(const char* str);
```

```
Complex& operator+=(const Complex& c);
Complex& operator-=(const Complex& c);
Complex& operator*=(const Complex& c);
Complex& operator/=(const Complex& c);

Complex& operator=(const Complex& c);

void println() {
    std::cout << "(" << real << " , " << img << " ) " << std::endl;
}
};

Complex::Complex(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)

    int begin = 0, end = strlen(str);
    img = 0.0;
    real = 0.0;

    // 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
    int pos_i = -1;
    for (int i = 0; i != end; i++) {
        if (str[i] == 'i') {
            pos_i = i;
            break;
        }
    }

    // 만일 'i' 가 없다면 이 수는 실수 뿐이다.
    if (pos_i == -1) {
        real = get_number(str, begin, end - 1);
        return;
    }

    // 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
    real = get_number(str, begin, pos_i - 1);
    img = get_number(str, pos_i + 1, end - 1);

    if (pos_i >= 1 && str[pos_i - 1] == '-') img *= -1.0;
}

double Complex::get_number(const char* str, int from, int to) {
    bool minus = false;
    if (from > to) return 0;

    if (str[from] == '-') minus = true;
    if (str[from] == '-' || str[from] == '+') from++;

    double num = 0.0;
    double decimal = 1.0;
```

```
bool integer_part = true;
for (int i = from; i <= to; i++) {
    if (isdigit(str[i]) && integer_part) {
        num *= 10.0;
        num += (str[i] - '0');
    } else if (str[i] == '.')
        integer_part = false;
    else if (isdigit(str[i]) && !integer_part) {
        decimal /= 10.0;
        num += ((str[i] - '0') * decimal);
    } else
        break; // 그 이외의 이상한 문자들이 올 경우
}

if (minus) num *= -1.0;

return num;
}

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}

Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}

Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}

Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}

Complex Complex::operator+(const char* str) {
    Complex temp(str);
    return (*this) + temp;
}

Complex Complex::operator-(const char* str) {
    Complex temp(str);
    return (*this) - temp;
}

Complex Complex::operator*(const char* str) {
    Complex temp(str);
    return (*this) * temp;
}

Complex Complex::operator/(const char* str) {
    Complex temp(str);
    return (*this) / temp;
```

```

}

Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}

Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}

Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}

Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}

Complex& Complex::operator=(const Complex& c) {
    real = c.real;
    img = c.img;
    return *this;
}

int main() {
    Complex a(0, 0);
    a = a + "-1.1 + i3.923";
    a.println();
    a = a - "1.2 -i1.823";
    a.println();
    a = a * "2.3+i22";
    a.println();
    a = a / "-12+i55";
    a.println();
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

( -1.1 , 3.923 )
( -2.3 , 5.746 )
( -131.702 , -37.3842 )
( -0.150113 , 2.42733 )

```

와 같이 제대로 계산됨을 알 수 있습니다.

한 가지 재미 있는 점은, `Complex(const char *str)` 생성자만 남겨놓고, `operator+(const char *str)` 계열들을 모두 지워보시고 컴파일 해보세요. 다시 말해서;

```
#include <cstring>
#include <iostream>

class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }
    Complex(const char* str);

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex& operator/=(const Complex& c);

    Complex& operator=(const Complex& c);

    void println() {
        std::cout << "(" << real << " , " << img << " ) " << std::endl;
    }
};

Complex::Complex(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)

    int begin = 0, end = strlen(str);
    img = 0.0;
    real = 0.0;

    // 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
    int pos_i = -1;
    for (int i = 0; i != end; i++) {
        if (str[i] == 'i') {
            pos_i = i;
            break;
        }
    }

    // 만일 'i' 가 없다면 이 수는 실수 뿐이다.
    if (pos_i == -1) {
```

```
    real = get_number(str, begin, end - 1);
    return;
}

// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
real = get_number(str, begin, pos_i - 1);
img = get_number(str, pos_i + 1, end - 1);

if (pos_i >= 1 && str[pos_i - 1] == '-') img *= -1.0;
}

double Complex::get_number(const char* str, int from, int to) {
    bool minus = false;
    if (from > to) return 0;

    if (str[from] == '-') minus = true;
    if (str[from] == '-' || str[from] == '+') from++;

    double num = 0.0;
    double decimal = 1.0;

    bool integer_part = true;
    for (int i = from; i <= to; i++) {
        if (isdigit(str[i]) && integer_part) {
            num *= 10.0;
            num += (str[i] - '0');
        } else if (str[i] == '.')
            integer_part = false;
        else if (isdigit(str[i]) && !integer_part) {
            decimal /= 10.0;
            num += ((str[i] - '0') * decimal);
        } else
            break; // 그 이외의 이상한 문자들이 올 경우
    }

    if (minus) num *= -1.0;

    return num;
}
Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}
Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}
Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}
Complex Complex::operator/(const Complex& c) {
```

```

Complex temp(
    (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
    (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
return temp;
}

Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}
Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}
Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}
Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}
Complex& Complex::operator=(const Complex& c) {
    real = c.real;
    img = c.img;
    return *this;
}

int main() {
    Complex a(0, 0);
    a = a + "-1.1 + i3.923";
    a.println();
    a = a - "1.2 -i1.823";
    a.println();
    a = a * "2.3+i22";
    a.println();
    a = a / "-12+i55";
    a.println();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

( -1.1 , 3.923 )
( -2.3 , 5.746 )
( -131.702 , -37.3842 )
( -0.150113 , 2.42733 )

```

놀랍게도 정확히 동일하게 나옵니다. 아니 이게 뭔가요. 고생 고생을 해서

```
Complex operator+(const char *str);
Complex operator-(const char *str);
Complex operator*(const char *str);
Complex operator/(const char *str);
```

들을 모두 만들어 주었더니, 결과적으로 생성자 하나만 만들면 충분하다는 것이였나요? 놀랍게도, 우리의 컴파일러는 매우 똑똑하기 때문에 이와 같은 일이 가능합니다. 우리가

```
a = a + "-1.1 + i3.923";
```

와 같은 문장을 사용하였을 때, 앞에서 이야기 해왔듯이 컴파일러가 위 문장을

```
a = a.operator+("-1.1 + i3.923");
```

로 바꿔줍니다. 하지만, 우리에게는 `operator+(const char *str)` 이 없고, `operator+(const Complex& c)` 밖에 없기 때문에 직접적으로 오버로딩 되지는 않습니다. 그렇지만, 컴파일러는 매우 똑똑하기 때문에 그 다음 순위로 오버로딩 될 수 있는 함수들이 있는지 없는지 확인해봅니다. 그런데 놀랍게도, 우리에게는 `const char *`에서 `Complex`를 생성할 수 있는 생성자

```
Complex(const char* str);
```

가 있기 때문에 컴파일러는 문자열 리터럴로 부터 `const Complex` 타입의 객체를 새롭게 생성할 수 있게 된다는 것입니다. 즉, 위 문장은 은 다음과 같이 변환됩니다.

```
a = a.operator+(Complex("-1.1 + i3.923"));
```

그럼 이제 `const Complex`에 인자로 전달할 수 있게 되어서 제대로 프로그램이 작동을 하게 되지요. 여기서 한 가지 짚고 넘어가야 할 점은, 만일 우리가 `operator+` 함수의 인자가 `const Complex& c` 가 아니라 그냥 `Complex& c`로 받도록 하였다면 위와 같은 변환은 이루어지지 않습니다. 왜냐하면 `-1.1 + i3.923` 자체가 문자열 리터럴 이므로, 이를 바탕으로 생성된 객체 역시 상수 여야 하기 때문입니다. 따라서 여러 모로 함수 인자의 값을 변형하지 않는다고 확신이 들면 무조건 `const` 인자로 받도록 하는 것이 좋습니다.

이러한 방식으로 여러분의 `Complex` 클래스의 문자열을 이용해서 복소수 덧셈을 수행할 수 있는 훌륭한 기능을 추가하였습니다. 하지만 문제는 다음과 같은 문장은 실행이 될까요?

```
a = "-1.1 + i3.923" + a;
```

사실 이 문장이나, 원래의

```
a = a + "-1.1 + i3.923";
```

문장이나 정확히 동일한 식입니다. 왜냐하면 + 연산자는 교환 법칙이 성립해야만 하기 때문이죠. 하지만 여러분도 이미 짐작하셨겠지만 전자의 경우에는 성공적으로 컴파일 되지 않습니다. 왜냐하면 `a + "-1.1+i3.923"`의 경우 이 문장이 `a.operator+("-1.1+i3.923")`으로 변환되어서 정확히 수행될 수 있지만 `"-1.1 + i3.923" + a`의 경우에는 이 같은 변환이 불가능 하기 때문입니다.

그렇다면 이러한 문제를 어떻게 해결할 수 있을까요?

이에 대해서는 다음 시간에 다루어 보도록 하겠습니다.

생각해보기

문제 1

그렇다면 `Complex` 클래스의 연산자 오버로딩을 수행하면서 쌓은 지식을 바탕으로 `MyString` 함수를 완전하게 만들어봅시다. 즉, 강좌 서두에서 지적한 사항들을 모두 구현하시면 됩니다. (난이도 : 下)

잡다한 연산자들의 오버로딩

안녕하세요 여러분. 잘 지내셨나요? 올해 안으로 C++ 강좌를 끝내기 위한 노력의 일환으로 강좌를 열심히 업로드 하려고 노력중인 Psi 입니다. 그래도 강좌를 읽으시는 여러분들은 꼼꼼히 읽어보시고, 궁금하신 점들은 꼭 댓글로 남겨 주시거나 메일로 질문해 주시기 바랍니다 :)

지난 강좌에서, 마지막에 다음과 같은 문제점을 지적하였습니다.

```
a = a + "-1.1 + i3.923"; // ①
```

는 잘 컴파일 되서 실행되지만

```
a = "-1.1 + i3.923" + a; // ②
```

는 컴파일 되지 않습니다. 왜냐하면, ①의 경우 `a.operator+("i3.923");` 으로 변환될 수 있지만 ②는 그렇지 못하기 때문이죠. 하지만, 원칙적으로 클래스를 사용하는 사용자의 입장에서 ①이 된다면 당연히 ②도 수행될 수 있어야 연산자 오버로딩을 하는 명분이 생깁니다. 다행 스럽게도, 사실 컴파일러는 이항 연산자 (피연산자를 두 개를 취하는 연산자 - 예를 들어서 +, -, *, /, ->, = 등이 이항 연산자이다) 를 다음과 같은 두 개의 방식으로 해석합니다.

어떤 임의의 연산자 @ 에 대해서, `a@b` 는¹⁾

```
*a.operator@(b);  
*operator@(a, b);
```

두 가지 방법으로 해석됩니다.²⁾

즉, 컴파일 시에 둘 중 하나의 가장 가까운 것을 택해서 처리됩니다. `a.operator@(b)` 에서의 `operator@` 는 `a` 의 클래스의 멤버 변수로써 사용되는 것이고, `operator@(a,b)` 에서의 `operator@` 는 그냥 일반적인 함수를 의미하게 됩니다. 따라서 이를 처리하기 위해 함수를 정의하여 봅시다.

```
Complex operator+(const Complex& a, const Complex& b) {  
    Complex temp(a);  
    return temp.operator+(b);  
}
```

1) C++ 에 @ 연산자는 없지만, 여기서 '임의의 연산자' 를 나타내기 위해 잠시 사용하겠습니다.

2) 참고로 이는 일부 연산자들에 대해서는 해당되지 않는데 대표적으로 [] 연산자 (첨자), -> 연산자 (멤버 접근), 대입 연산자 (=), () 함수 호출 연산자들의 경우 멤버 함수로만 존재할 수 있습니다. 즉, 따로 멤버 함수가 아닌 전역 함수로 뒹 수 없다는 의미입니다. 따라서 이들 함수를 오버로딩 할 때 주의하시기 바랍니다.

우리의 또 다른 `operator+` 는 두 개의 `const Complex&` 타입의 인자 `a, b` 를 받게 됩니다. 앞에서도 말했지만 컴파일러는 정확히 일치하지 않는 경우, 가장 가까운 '가능한' 오버로딩 되는 함수를 찾게 되는데, 마침 우리에게는 `Complex(const char *)` 타입의 생성자가 있으므로,

```
"-1.1 + i3.923" + a;
```

는

```
operator+(Complex("-1.1 + i3.923"), a);
```

가 되어서 잘 실행되게 됩니다. 실제로 컴파일 해보면

```
#include <cstring>
#include <iostream>

class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }
    Complex(const char* str);

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex& operator/=(const Complex& c);

    Complex& operator=(const Complex& c);

    void println() {
        std::cout << "(" << real << ", " << img << ")" << std::endl;
    }
};

Complex operator+(const Complex& a, const Complex& b) {
    Complex temp(a);
    return temp.operator+(b);
}

Complex::Complex(const char* str) {
```

```
// 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.  
// 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"  
// 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)  
  
int begin = 0, end = strlen(str);  
img = 0.0;  
real = 0.0;  
  
// 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.  
int pos_i = -1;  
for (int i = 0; i != end; i++) {  
    if (str[i] == 'i') {  
        pos_i = i;  
        break;  
    }  
}  
  
// 만일 'i' 가 없다면 이 수는 실수 뿐이다.  
if (pos_i == -1) {  
    real = get_number(str, begin, end - 1);  
    return;  
}  
  
// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.  
real = get_number(str, begin, pos_i - 1);  
img = get_number(str, pos_i + 1, end - 1);  
  
if (pos_i >= 1 && str[pos_i - 1] == '-') img *= -1.0;  
}  
double Complex::get_number(const char* str, int from, int to) {  
    bool minus = false;  
    if (from > to) return 0;  
  
if (str[from] == '-') minus = true;  
if (str[from] == '-' || str[from] == '+') from++;  
  
double num = 0.0;  
double decimal = 1.0;  
  
bool integer_part = true;  
for (int i = from; i <= to; i++) {  
    if (isdigit(str[i]) && integer_part) {  
        num *= 10.0;  
        num += (str[i] - '0');  
    } else if (str[i] == '.')  
        integer_part = false;  
    else if (isdigit(str[i]) && !integer_part) {  
        decimal /= 10.0;  
        num += ((str[i] - '0') * decimal);  
    } else  
        break; // 그 이외의 이상한 문자들이 올 경우
```

```
    }

    if (minus) num *= -1.0;

    return num;
}

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}

Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}

Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}

Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}

Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}

Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}

Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}

Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}

Complex& Complex::operator=(const Complex& c) {
    real = c.real;
    img = c.img;
    return *this;
}

int main() {
    Complex a(0, 0);
    a = "-1.1 + i3.923" + a;
    a.println();
```

```
}
```

성공적으로 컴파일 하였다면

실행 결과

```
( -1.1 , 3.923 )
```

와 같이 잘 실행됨을 알 수 있습니다. 그런데, 아마 `operator+` 를 자세히 살펴보신 분들은 아마 다음과 같은 문제점을 확인할 수 있었을 것입니다.

```
Complex operator+(const Complex& a, const Complex& b) {
    Complex temp(a);
    return temp.operator+(b);
}
```

왜 굳이 귀찮게 `temp` 라는 새로운 `Complex` 객체를 정의하여서 `temp` 와의 + 연산을 리턴하느냐 입니다. 그냥 `a + b` 할 것을 불필요하게 복사 생성을 한 번 더 하게 되어서 성능의 하락이 발생하게 됩니다. 하지만, 그냥 `a + b` 를 하게 된다면;

```
Complex operator+(const Complex& a, const Complex& b) { return a + b; }
```

위 코드를 컴파일 시에 다음과 같은 경고 메세지를 볼 수 있을 것입니다.

컴파일 오류

```
warning C4717: 'operator+' : recursive on all control paths,
→ function will cause runtime stack overflow
```

즉, `a + b` 에서 `a.operator+(b)` 가 호출되는 것이 아니라, `operator+(a,b)` 가 호출 되기 때문에 재귀적으로 무한히 많이 함수가 호출되어 오류가 발생한다는 것이지요. 실제로 실행해 보아도 프로그램이 강제로 종료되는 모습을 볼 수 있습니다. 따라서 이와 같은 문제를 방지하기 위해서 우리는 다음과 같이 강제로 멤버 함수 `operator+` 를 호출하도록 지정하였습니다.

```
Complex operator+(const Complex& a, const Complex& b) { return a.operator+(b); }
```

이 역시 컴파일 되지 않습니다. 아마 오류의 내용은 다음과 같을 것입니다.

컴파일 오류

```
error C2662: 'Complex::operator +' : cannot convert 'this' pointer
→   from 'const Complex' to 'Complex &'
```

이 말은 즉슨, `a` 가 `const Complex` 인데, 우리가 호출하고자 하는 멤버 함수 `operator+` 는 `const` 함수가 아니기 때문입니다. 상당히 골치아픈 문제가 아닐 수 없습니다. (참고로 `const` 함수가 무엇인지 기억이 나지 않으시는 분들은 이 강좌를 다시 읽어보시기 바랍니다)

`const` 객체는 언제나 값이 바뀔 수 없으며, 만일 `const` 객체의 멤버 함수 호출 시에는 그 함수가 객체의 값을 바꾸지 않는다고 보장할 수 있도록 `const` 함수여야만 합니다. 하지만 멤버 함수 `operator+` 는 `const` 성이 없으므로, `operator+` 를 호출하는 것은 불가능 해지지요.

이 문제를 해결할 수 있는 유일한 방법은 `Complex operator+(const Complex& a, const Complex& b)` 내부에서 다른 함수들을 호출하지 않고 직접 덧셈을 수행하면 됩니다. 다만 이 방법도 문제가 있지요. 멤버 함수가 아닌 외부 함수 `operator+` 에서 객체의 `private` 정보에 접근할 수 있어야만 하는데, 이 것이 불가능 하기 때문입니다. 하지만, 놀랍게도 C++ 에서는 이를 가능케 하는 키워드가 있습니다.

friend 는 모든 것을 공유한다.

아마 이 글을 읽는 독자 여러분들은 자신의 모든 것을 아낌없이 털어놓을 수 있는 절친한 친구 한 두 명쯤은 있을 것입니다. 그 친구와 나 사이에는 어떠한 정보도 열람할 수 있는 관계가 되지요. 그런데 재미있는 사실에는 비슷한 역할을 하는 키워드가 C++ 에도 있다는 점입니다. 그 이름도 역시 `friend` 입니다.

```
class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }
    Complex(const char* str);

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
```

```

Complex& operator==(const Complex& c);
Complex& operator!=(const Complex& c);
Complex& operator/(const Complex& c);

Complex& operator=(const Complex& c);

friend Complex operator+(const Complex& a, const Complex& b);

void println() {
    std::cout << "(" << real << " , " << img << " ) " << std::endl;
}
};

```

위와 같이 Complex 클래스 안에서

```
friend Complex operator+(const Complex& a, const Complex& b);
```

라 같이 쓰면 우리의 `Complex operator+(const Complex& a, const Complex& b);` 함수는 이제 Complex 의 friend 가 됩니다. 즉, Complex 클래스의 입장에서는 자신의 새로운 친구인 `operator+`에게 마음의 문을 열고 모든 정보에 접근할 수 있도록 허가하는 것입니다.

`private` 냐 `public` 이냐에 관계 없이 `Complex operator+(const Complex& a, const Complex& b);` 함수는 이제 어떤 Complex 객체라도 그 내부 정보에 접근할 수 있습니다.

따라서, 다음과 같은 코드를 사용하는 것도 가능하지요.

```

Complex operator+(const Complex& a, const Complex& b) {
    Complex temp(a.real + b.real, a.img + b.img);
    return temp;
}

```

이제 이 `operator+` 함수는 마치 Complex 클래스의 멤버 변수인양, 객체들의 정보에 접근할 수 있게 됩니다. `real` 변수는 `private` 이지만, `a.real` 을 해도 무방하지요. 이렇게 된다면, 이전의 `operator+`에서 불필요하게 `temp` 객체를 생성했던 것과는 달리 필요한 것만 사용하면 됩니다.

한 가지 재미 있는 사실은 `friend` 키워드는 함수에만 적용할 수 있는 것이 아니라, 다른 클래스 자체도 `friend` 로 지정할 수 있습니다. 예를 들어서,

```

class A {
private:
    int x;

    friend B;
};

```

```
class B {
private:
    int y;
};
```

와 같이 할 경우, A 는 B 를 `friend` 로 지정하게 된 것입니다. 한 가지 주의할 사실은, 우리가 흔히 생각하는 `friend` 관계와는 다르게, C++ 에서 `friend` 는 짹사랑과 비슷합니다. 즉, A 는 자기 생각에 B 가 `friend` 라고 생각하는 것이므로, B 에게 A 의 모든 것을 공개합니다.

즉 클래스 B 에서 A 의 `private` 변수인 `x` 에 접근할 수 있게 됩니다. 하지만 B 에는 A 가 `friend` 라고 지정하지 않았으므로, B 의 입장에서는 A 에게 어떠한 내용도 공개하지 않습니다 (`public` 변수들 빼고). 따라서 A 는 B 의 `private` 변수인 `int y` 에 접근할 수 없게 됩니다.

입출력 연산자 오버로딩 하기

아마도, 눈치를 채신 분들이 있겠지만 우리가

```
std::cout << a;
```

라고 하는 것은 사실 `std::cout.operator<<(a)` 를 하는 것과 동일한 명령이었습니다. 즉, 어떤 `std::cout` 이라는 객체에 멤버 함수 `operator<<` 가 정의되어 있어서 `a` 를 호출하게 되는 것이지요. 그런데, `std::cout` 이 `int` 나 `double` 변수, 심지어 문자열 까지 자유 자재로 `operator<<` 하나로 출력할 수 있었던 이유는 그 많은 수의 `operator<<` 함수들이 오버로딩 되어 있다는 뜻입니다.

실제로 우리가 `include` 하던 `iostream` 의 헤더파일의 내용을 살펴보면 (실제로는 `ostream`에 정의되어 있습니다. 다만 `iostream` 이 `ostream` 을 `include` 하고 있음) `ostream` 클래스에

```
ostream& operator<<(bool val);
ostream& operator<<(short val);
ostream& operator<<(unsigned short val);
ostream& operator<<(int val);
ostream& operator<<(unsigned int val);
ostream& operator<<(long val);
ostream& operator<<(unsigned long val);
ostream& operator<<(float val);
ostream& operator<<(double val);
ostream& operator<<(long double val);
ostream& operator<<(void* val);
```

와 같이 엄청난 수의 `operator<<` 가 정의되어 있는 것을 알 수 있습니다. 이를 덕분에 우리가 편하게 인자의 타입에 관계없이 손쉽게 출력을 사용할 수 있게 되는 것이지요.

그렇다면 한 번 우리의 `Complex` 클래스에서 `ostream` 클래스의 연산자 `operator<<`를 자유롭게 사용할 수 있으면 어떨까요. 예를 들어서

```
Complex c;
std::cout << c;
```

를 하게 되면 마치

```
Complex c;
c.println();
```

을 한 것과 같은 효과를 내도록 말이지요. 당연하게도, `ostream` 클래스에 `operator<<` 멤버 함수를 새롭게 추가하는 것은 불가능 합니다. 이는 표준 헤더파일의 내용을 수정하는 것과 같기 때문이죠. 대부분의 경우 표준 헤더파일은 읽기만 가능이고, 원칙적으로 표준 라이브러리의 내용을 수정하는 것은 좋지 않습니다 (정확히 말하면 하면 안됩니다!). 따라서, 우리는 `ostream` 클래스에 `Complex` 객체를 오버로딩하는 `operator<<` 연산자 함수를 추가할 수는 없지요.

그 대신에, 여태 까지 배운 내용에 따르면 아예 `operator<<` 전역 함수 하나를 정해서 `Complex`의 `friend`로 지정한 다음에 사용할 수 있습니다. 그 함수는 아마 다음과 같이 생겼겠지요.

```
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << "(" << c.real << ", " << c.img << ")";
    return os;
}
```

여기서 왜 `std::cout`이 아니고 `os`라고 의문을 가질 수도 있는데, `std::cout` 자체가 `iostream`에서 하나 만들어 놓은 `ostream` 객체입니다. 따라서 `ostream&` 타입으로 `std::cout` 객체를 받아서 이를 출력하면 됩니다. 마찬가지로, `Complex` 클래스 내부에서 `friend` 선언을 해주시면 됩니다. 참고로 `operator<<`에서 `ostream&` 타입을 리턴하는 이유는 다음과 같은 문장을 처리할 수 있기 위해서입니다.

```
std::cout << "a의 값은 : " << a << "이다. " << std::endl;
```

`<<` 연산자는 왼쪽 부터 오른쪽 순으로 실행되기 때문에 가장 먼저 `std::cout.operator<<("a의 값은?")`이 실행되고, 그 자리에 `std::cout`이 다시 리턴됩니다. 그 다음에는 `operator <<(std::cout, a);`가 되어서 쭉쭉 이어질 수 있도록 이와 같이 `ostream&`를 리턴하게 되는 것입니다. 참고로, `Complex` 클래스 내부에는

```
friend ostream& operator<<(ostream& os, const Complex& c);
```

위와 같이 `friend` 선언을 해주시면 됩니다. 비슷한 방법으로 `Complex` 객체 `c`에 대해 `cin >> c;` 와 같은 작업을 할 수 있습니다. 다만, 이번에는 `cin`은 `istream` 객체이고, `operator>>` 를 오버로딩 해야 된다는 점이 다를 뿐이지요.

```
#include <cstring>
#include <iostream>

class Complex {
private:
    double real, img;

    double get_number(const char* str, int from, int to);

public:
    Complex(double real, double img) : real(real), img(img) {}
    Complex(const Complex& c) { real = c.real, img = c.img; }
    Complex(const char* str);

    Complex operator+(const Complex& c);
    Complex operator-(const Complex& c);
    Complex operator*(const Complex& c);
    Complex operator/(const Complex& c);

    Complex& operator+=(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex& operator/=(const Complex& c);

    Complex& operator=(const Complex& c);

    friend Complex operator+(const Complex& a, const Complex& b);
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);

    void println() {
        std::cout << "(" << real << " , " << img << " ) " << std::endl;
    }
};

std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << "(" << c.real << " , " << c.img << " ) ";
    return os;
}

Complex operator+(const Complex& a, const Complex& b) {
    Complex temp(a.real + b.real, a.img + b.img);
    return temp;
}

Complex::Complex(const char* str) {
    // 입력 받은 문자열을 분석하여 real 부분과 img 부분을 찾아야 한다.
    // 문자열의 꼴은 다음과 같습니다 "[부호](실수부)(부호)i(허수부)"
    // 이 때 맨 앞의 부호는 생략 가능합니다. (생략시 + 라 가정)
```

```
int begin = 0, end = strlen(str);
img = 0.0;
real = 0.0;

// 먼저 가장 기준이 되는 'i' 의 위치를 찾는다.
int pos_i = -1;
for (int i = 0; i != end; i++) {
    if (str[i] == 'i') {
        pos_i = i;
        break;
    }
}

// 만일 'i' 가 없다면 이 수는 실수 뿐이다.
if (pos_i == -1) {
    real = get_number(str, begin, end - 1);
    return;
}

// 만일 'i' 가 있다면, 실수부와 허수부를 나누어서 처리하면 된다.
real = get_number(str, begin, pos_i - 1);
img = get_number(str, pos_i + 1, end - 1);

if (pos_i >= 1 && str[pos_i - 1] == '-') img *= -1.0;
}

double Complex::get_number(const char* str, int from, int to) {
    bool minus = false;
    if (from > to) return 0;

    if (str[from] == '-') minus = true;
    if (str[from] == '-' || str[from] == '+') from++;

    double num = 0.0;
    double decimal = 1.0;

    bool integer_part = true;
    for (int i = from; i <= to; i++) {
        if (isdigit(str[i]) && integer_part) {
            num *= 10.0;
            num += (str[i] - '0');
        } else if (str[i] == '.')
            integer_part = false;
        else if (isdigit(str[i]) && !integer_part) {
            decimal /= 10.0;
            num += ((str[i] - '0') * decimal);
        } else
            break; // 그 이외의 이상한 문자들이 올 경우
    }

    if (minus) num *= -1.0;
}
```

```

    return num;
}

Complex Complex::operator+(const Complex& c) {
    Complex temp(real + c.real, img + c.img);
    return temp;
}

Complex Complex::operator-(const Complex& c) {
    Complex temp(real - c.real, img - c.img);
    return temp;
}

Complex Complex::operator*(const Complex& c) {
    Complex temp(real * c.real - img * c.img, real * c.img + img * c.real);
    return temp;
}

Complex Complex::operator/(const Complex& c) {
    Complex temp(
        (real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
        (img * c.real - real * c.img) / (c.real * c.real + c.img * c.img));
    return temp;
}

Complex& Complex::operator+=(const Complex& c) {
    (*this) = (*this) + c;
    return *this;
}

Complex& Complex::operator-=(const Complex& c) {
    (*this) = (*this) - c;
    return *this;
}

Complex& Complex::operator*=(const Complex& c) {
    (*this) = (*this) * c;
    return *this;
}

Complex& Complex::operator/=(const Complex& c) {
    (*this) = (*this) / c;
    return *this;
}

Complex& Complex::operator=(const Complex& c) {
    real = c.real;
    img = c.img;
    return *this;
}

int main() {
    Complex a(0, 0);
    a = "-1.1 + i3.923" + a;
    std::cout << "a의 값은 : " << a << " 이다. " << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```
a 의 값은 : ( -1.1 , 3.923 ) 이다.
```

와 같이 잘 실행됨을 알 수 있습니다.

첨자 연산자 (operator[])

이번에는 배열에서 원소를 지정할 때 사용되는 첨자 연산자 [] 를 오버로딩 해보도록 합시다.³⁾ 우리가 전에 만들었던 MyString 클래스에서 개개의 문자에 접근하기 위해 [] 를 지원해주어야만 하는데요, [] 를 이용해서 str[10] 과 같이 10 번째 문자에 정확하게 접근할 수 있게 됩니다.

여기서 그렇다면 MyString 클래스의 operator[] 를 추가해보도록 합시다. operator[] 함수는 자명하게도 인자로 몇 번째 문자인지, int 형 변수를 인덱스로 받게 됩니다. 따라서 operator[] 는 다음과 같은 원형을 가집니다.

```
char& operator[](const int index);
```

index 로 [] 안에 들어가는 값을 받게 됩니다. 그리고 char& 를 인자로 리턴하는 이유는

```
str[10] = 'c';
```

와 같은 명령을 수행하기 때문에, 그 원소의 레퍼런스를 리턴하게 되는 것이지요. 실제로 operator[] 의 구현은 아래와 같이 매우 단순합니다.

```
char& MyString::operator[](const int index) { return string_content[index]; }
```

위와 같이 index 번째의 string_content 를 리턴해서, operator[] 를 사용하는 사용자가, 이의 레퍼런스를 가질 수 있게 되지요. 그렇다면, 전체 소스를 한 번 살펴보도록 합시다.

```
#include <iostream>
#include <cstring>

class MyString {
    char* string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이
```

3) 참고로 왜 첨자 연산자라고 부르냐면, 배열의 원소를 지정할 때 [] 안에 넣는 수를 첨자(subscript) 라고 부르기 때문입니다

```
int memory_capacity; // 현재 할당된 용량

public:
    // 문자 하나로 생성
    MyString(char c);

    // 문자열로 부터 생성
    MyString(const char* str);

    // 복사 생성자
    MyString(const MyString& str);

    ~MyString();

    int length();

    void print() const;
    void println() const;

    char& operator[](const int index);
};

MyString::MyString(char c) {
    string_content = new char[1];
    string_content[0] = c;
    memory_capacity = 1;
    string_length = 1;
}

MyString::MyString(const char* str) {
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString& str) {
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::~MyString() { delete[] string_content; }

int MyString::length() { return string_length; }

void MyString::print() const {
    for (int i = 0; i != string_length; i++) {
        std::cout << string_content[i];
    }
}

void MyString::println() const {
```

```

for (int i = 0; i != string_length; i++) {
    std::cout << string_content[i];
}
std::cout << std::endl;
}

char& MyString::operator[](const int index) { return string_content[index]; }

int main() {
    MyString str("abcdef");
    str[3] = 'c';

    str.println();
}

```

성공적으로 컴파일 하였다면

실행 결과

abcccef

와 같이 제대로 str[3] 의 'd' 를 'c' 로 잘 바꾸었음을 알 수 있습니다.

이 정도만 하면 MyString 클래스는 거의 웬만한 문자열 클래스 뺨치게 완전한 모습을 갖추었다고 볼 수 있습니다. 문자열 삽입, 삭제, 대입 뿐만이 아니라 개개의 문자의 조작 까지 원활하게 수행할 수 있는 훌륭한 문자열 클래스라고 할 수 있지요.

int Wrapper 클래스 - 타입 변환 연산자

Wrapper 라는 것은 원래 우리가 흔히 음식을 포장할 때 '랩(wrap)으로 싼다' 라고 하는 것 처럼, '포장지' 라는 의미의 단어입니다. 즉 Wrapper 클래스는 무언가를 포장하는 클래스라는 의미인데, C++ 에서 프로그래밍을 할 때 어떤 경우에 기본 자료형들을 객체로써 다루어야 할 때가 있습니다. 이럴 때, 기본 자료형들 (int, float 등등) 을 클래스로 포장해서 각각의 자료형을 객체로 사용하는 것을 Wrapper 클래스를 이용한다는 뜻입니다.

즉, int 자료형을 감싸는 int Wrapper 클래스 Int 는 다음과 같이 구성할 수 있습니다.

```

class Int
{
    int data;
    // some other data

public:
    Int(int data) : data(data) {}

```

```
Int(const Int& i) : data(i.data) {}  
};
```

위 Int 클래스에 int 형 자료형을 보관하는 data 라는 변수를 정의해 놓았는데, 이렇게 한다면 int 형 데이터를 저장하는 객체로 Int 클래스를 사용할 수 있을 것입니다. 우리는 이 Int 객체가 int 의 Wrapper 클래스의 객체인 만큼, int 와 정확히 똑같이 작동하도록 만들고 싶습니다. 다시 말해서 다음과 같은 명령을 내려도 아무 하자 없이 잘 실행될 수 있도록 말이지요.

```
Int x = 3;           // Wrapper 객체  
int a = x + 4;     // 그냥 평범한 int 형 변수 a
```

이를 잘 수행하기 위해서라면, 여태까지 연산자 오버로딩을 열심히 배워오신 여러분 생각이라면 그렇다면 int 변수에 사용되는 모든 연산자 함수들을 만들어주면 되겠군!

이라고 생각이 들 것입니다. 물론, 이렇게 해도 잘 작동하는 Wrapper 클래스를 만들 수 있을 것입니다. 하지만, 그 수 많은 연산자들을 일일히 오버로딩을 하는 것은 정말로 고통스러운 일이 아닐 수 없습니다.

왜 이러한 일이 고통스러운 것인가요? Complex 클래스를 만들 때만 해도, Complex 객체에서 + 나 - 연산자가 하는 일 자체가 int 변수끼리 하는 일과 완전히 달랐기 때문에 반드시 operator+ 나 operator- 등을 만들어주어야만 했을 것입니다. 하지만 이 int Wrapper 클래스 객체끼리 하는 일은 그냥 단순히 int 형 변수끼리 하는 일과 정확히 똑같기 때문에 굳이 이미 제공하는 기능을 다시 만들어야 한다는 점이지요.

그렇다면, 그냥 이 Wrapper 클래스의 객체를 마치 'int 형 변수' 라고 컴파일러가 생각할 수는 없는 것일까요. 물론 가능합니다. 왜냐하면 우리에게는 타입 변환 연산자가 있기 때문이지요. 만일 컴파일러가 이 클래스의 객체를 int 형 변수로 변환할 수 있다면, 비록 operator+ 등을 정의하지 않더라도 컴파일러가 가장 이 객체를 int 형 변수로 변환 한 다음에 + 를 수행할 수 있기 때문입니다.

타입 변환 연산자는 다음과 같이 정의합니다.

operator (변환 하고자 하는 타입) ()

예를 들어 우리의 int Wrapper 클래스의 경우 다음과 같은 타입 변환 연산자를 정의할 수 있지요.

operator int()

한 가지 주의할 점은, 생성자처럼 함수의 리턴 타입을 써주시면 안됩니다. 이는 C++에서 변환 연산자를 정의하기 위한 언어 상의 규칙이라고도 볼 수 있습니다. 그렇게 된다면, 우리의 int 변환 연산자는 다음과 같이 간단하게 구성할 수 있겠지요.

```
operator int() { return data; }
```

그냥 단순히 `data` 를 리턴해주면 됩니다. 그렇게 된다면 우리의 `Wrapper` 클래스의 객체를 '읽는' 데에는 아무런 문제가 없게 됩니다. 왜냐하면 컴파일러 입장에서 적절한 변환 연산자로 `operator int` 를 찾아서 결국 `int` 로 변환해서 처리하기 때문이지요. 다만 문제는 '대입' 시에 발생하는데, 다행이도 디폴트 대입 연산자가 이 역시 알아서 잘 처리할 것이기 때문에 걱정 안하셔도 됩니다.

```
#include <iostream>

class Int {
    int data;
    // some other data

    public:
    Int(int data) : data(data) {}
    Int(const Int& i) : data(i.data) {}

    operator int() { return data; }
};

int main() {
    Int x = 3;
    int a = x + 4;

    x = a * 2 + x + 4;
    std::cout << x << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

21

와 같이 `Int` 객체가 `int` 변수 처럼 정확히 동일하게 작동되고 있음을 알 수 있습니다.

전위/후위 증감 연산자

마지막으로 살펴볼 연산자로 우리가 흔히 `++`, `--` 로 사용하는 전위/후위 증감 연산자들입니다. 아마, 이 연산자를 오버로딩 하기 전에 한 가지 궁금증이 드셨을 텐데요, 과연 C++ 컴파일러는 전위/후위 증감을 구분 해서 오버로딩 시켜주나 입니다. 다시 말해;

```
a++;
++a;
```

위 두 ++ 연산자들을 구분해서 오버로딩을 시켜주나 이말이죠. 두 연산자 모두 `operator++` 이기 때문입니다.

C++ 언어에서는 다음과 같은 재미있는 방법으로 구분합니다. 일단 C++ 언어에서는 다음과 같은 재미있는 방법으로 구분합니다. 일단;

```
operator++();
operator--();
```

은 전위 증감 연산자 (`++x`, `--x`) 를 오버로딩 하게 됩니다. 그렇다면 후위 증감 연산자 (`x++`, `x--`) 는 어떨까요. 바로

```
operator++(int x);
operator--(int x);
```

로 구현하게 됩니다. 물론 인자 `x` 는 아무런 의미가 없습니다. 단순히 컴파일러 상에서 전위와 후위를 구별하기 위해 `int` 인자를 넣어주는 것이지요.

실제로 `++` 을 구현하면서 인자로 들어가는 값을 사용하는 경우는 없습니다. 따라서 그냥

```
operator++(int);
operator--(int);
```

와 같이 해도 무방합니다.

한 가지 중요한 점은, 전위 증감 연산의 경우 값이 바뀐 자기 자신을 리턴해야 하고, 후위 증감의 경우 값이 바뀌기 이전의 객체를 리턴해야 된다는 점입니다.

왜냐하면 전위와 후위 연산자가 어떻게 다른지 생각해봤을 때

```
int x = 1;
func(++x);
```

를 하게 되면 `func` 에는 2 가 인자로 전달되지만,

```
int x = 1;
func(x++);
```

의 경우 `func` 에 1 이 인자로 전달되고, 나중에 `x++` 이 실행되어서 `x` 가 2 가 되기 때문이지요. 따라서 이를 감안한다면 아래와 같은 꼴이 됩니다.

```
A& operator++() {
    // A ++ 을 수행한다.
    return *this;
}
```

전위 연산자는 간단히 `++` 에 해당하는 연산을 수행한 후에 자기 자신을 반환해야 합니다. 반면에 후위 연산자의 경우

```
A operator++(int) {
    A temp(A);
    // A++ 을 수행한다.
    return temp;
}
```

`++` 을 하기 전에 객체를 반환해야 하므로, `temp` 객체를 만들어서 이전 상태를 기록한 후에, `++` 을 수행한 뒤에 `temp` 객체를 반환하게 됩니다.

따라서 후위 증감 연산의 경우 추가적으로 복사 생성자를 호출하기 때문에 전위 증감 연산보다 더 느립니다!

그렇다면 아래와 같이 예제를 살펴봅시다.

클래스 자체에는 별거 없지만 전위와 후위가 호출될 때를 구별하기 위해 메세지를 출력하도록 하였습니다.

```
#include <iostream>

class Test {
    int x;

public:
    Test(int x) : x(x) {}
    Test(const Test& t) : x(t.x) {}

    Test& operator++() {
        x++;
        std::cout << "전위 증감 연산자" << std::endl;
        return *this;
    }

    // 전위 증감과 후위 증감에 차이를 두기 위해 후위 증감의 경우 인자로 int 를
    // 받지만 실제로는 아무것도 전달되지 않는다.
}
```

```

Test operator++(int) {
    Test temp(*this);
    x++;
    std::cout << "후위 증감 연산자" << std::endl;
    return temp;
}

int get_x() const {
    return x;
};

void func(const Test& t) {
    std::cout << "x : " << t.get_x() << std::endl;
}

int main() {
    Test t(3);

    func(++t); // 4
    func(t++); // 4 가 출력됨
    std::cout << "x : " << t.get_x() << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

전위 증감 연산자
x : 4
후위 증감 연산자
x : 4
x : 5

```

와 같이 제대로 골라서 실행되고 있음을 알 수 있습니다.

정리

연산자 오버로딩에 대해 다루면서 몇 가지 중요한 포인트 들만 따로 정리해보자면;

- 두 개의 동등한 객체 사이에서의 이항 연산자는 멤버 함수가 아닌 외부 함수로 오버로딩 하는 것이 좋습니다. (예를 들어 Complex 의 operator+(const Complex&, const Complex&) 와 같이 말입니다.)

- 두 개의 객체 사이의 이항 연산자 이지만 한 객체만 값이 바뀐다던지 등의 동등하지 않는 이항 연산자는 멤버 함수로 오버로딩 하는 것이 좋습니다. (예를 들어서 `operator+=` 는 이항 연산자 이지만 `operator+=(const Complex&)` 가 낫다)
- 단항 연산자는 멤버 함수로 오버로딩 하는 것이 좋습니다 (예를 들어 `operator++` 의 경우 멤버 함수로 오버로딩 합니다)
- 일부 연산자들은 반드시 멤버 함수로만 오버로딩 해야 합니다 (강좌 앞 부분 참고)

자, 이것으로 가장 많이 사용되는 연산자 함수들에 대해 알아보았습니다. 이제 슬슬 C++ 언어의 강력함이 느껴지시나요? 다음 강좌에서는 여태까지 배운 내용들을 총 망라하는 조그마한 프로젝트를 해볼려고 합니다. 그 프로젝트는 아래 '생각해보기'에 나와 있는데요, 다음 강좌를 보기 전 까지 아래 문제를 한 번 해결해 보시기 (해결은 못해도 최소한 노력은 하시기) 바랍니다.

생각해보기

문제 1

N 차원 배열을 제공하는 클래스를 만들어보세요. 이는 여러분이 여태까지 배운 내용을 시험할 것입니다. 참고로, 원소에 접근하기 위해서는 `a[1][2][3]` 과 같은 방법으로 접근하겠지요. 다만 N 차원 배열이기 때문에 (N 은 객체 생성시에 사용자가 입력합니다) 2 차원 배열은 `a[1][2]`, 3 차원 배열은 `a[1][2][3]` 과 같은 방식으로 접근할 것입니다. (난이도 : 最上)

문제 2

영어를 잘하시는 분들은 연산자 오버로딩에 관해 정리해 놓은 다음 글을 읽어보시기를 추천합니다. 참고로 이 글에서 다루지만 본 강좌에서는 다루지 않는 일부 내용들은 아직 배운 내용이 아니라 생략한 것이므로 너무 걱정하지 마시고 복습하는 느낌으로 천천히 읽어보시면 좋습니다. (난이도 : 中)

N 차원 배열 만들기 프로젝트

이번 강좌의 내용은 C++ 을 처음 배우는 분들에게는 이해하기 벅거울 수 있습니다. 만일 내용이 도저히 이해가 되지 않는다면, 이전 연산자 오버로딩 강좌 두 개를 꼼꼼히 다시 읽어보신 다음에, 이 강좌 앞부분 (C++ 스타일의 캐스팅) 만 읽고 넘어가셔도 좋습니다. 하지만 C++ 을 어느 정도 배웠고 복습하시는 차원에서 보시는 분들은 꼭 읽어보시기 바랍니다.

안녕하세요 여러분~ 지난번 강좌의 생각해보기는 잘 해보셨나요? 아마도 꽤나 어려웠을 것이라 생각합니다. 사실 N 차원 배열을 만드는 것 까지는 하셨을 지 모르겠지만, N 차원 배열을 [] 를 이용해서 원소에 접근하는 것을 구현하는 일은 상당한 수준의 아이디어가 필요하기 마련이지요.

이번 강좌에서는 이 N 차원 배열 만들기 프로젝트를 진행하면서, C++ 여러 라이브러리에서 주요하게 사용되는 몇 가지 아이디어들을 살펴보고 갈 것입니다.

본격적으로 프로젝트에 들어가기에 앞서 C++ 에 또 새롭게 추가된 한 가지 내용을 살펴보도록 하겠습니다.

C++ 스타일의 캐스팅

기존의 C 언어에서는, 캐스팅은 크게 2 가지 방법으로 발생하였습니다. 하나는 그냥 컴파일러에서 알아서 캐스팅 하는 암시적(**implicit**) 캐스팅과, 우리가 직접 이러이러하게 캐스팅 하라고 지정하는 명시적(**explicit**) 캐스팅이 있었지요.

암시적 캐스팅의 경우 `int` 와 `double` 변수와의 덧셈을 수행할 때, `int` 형 변수가 자동으로 `double` 변수로 캐스팅 되는 것과 같은 것을 말하고, 명시적 캐스팅의 경우 예를 들어 `void *` 타입의 주소를 특정 구조체 포인터 타입의 주소로 바꾼다던지 등의 캐스팅이 있습니다.

이 때, 명시적 캐스팅은 다음과 같이 수행되었지요.

```
ptr = (Something *)other_ptr;
int_variable = (int)float_variable;
```

와 같이 말이지요. 즉, 괄호 안에 원하는 타입을 넣고 변환을 수행한 것입니다. 하지만 이러한 방식을 사용하다 보니까 프로그래머들 사이에서 몇 가지 문제점들을 발견하였습니다.

먼저 위와 같은 타입 캐스팅의 경우 말도 안되는 캐스팅에 대해서 컴파일러가 오류를 발생시키지 않습니다. 따라서 프로그래머의 실수에 취약합니다.

뿐만 아니라 괄호 안에 타입을 넣는 방식으로 변환을 수행하는 탓에, 코드의 가독성이 떨어지게 됩니다. 즉,

```
function((int)variable);
```

와 같이 함수 호출에도 괄호를 사용하는데, 괄호가 너무 많아지게 된다면 읽는 사람이나 코드를 유지보수하는 사람 입장에서 여러모로 불편합니다. 뿐만 아니라 우리가 캐스팅을 하는데에는 여러가지 이유가 있기 마련인데, 위와 같은 C 형식 캐스팅에서는 읽는이가 그 캐스팅의 의미를 명확하게 알 수 없습니다.

하지만 C++ 에서는 다음과 같은 4 가지의 캐스팅을 제공하고 있습니다.

- `static_cast` : 우리가 흔히 생각하는, 언어적 차원에서 지원하는 일반적인 타입 변환
- `const_cast` : 객체의 상수성(const) 를 없애는 타입 변환. 쉽게 말해 `const int` 가 `int` 로 바뀐다.
- `dynamic_cast` : 파생 클래스 사이에서의 다운 캐스팅 (→ 정확한 의미는 나중에 다시 배울 것입니다)
- `reinterpret_cast` : 위험을 감수하고 하는 캐스팅으로 서로 관련이 없는 포인터들 사이의 캐스팅 등

이 때 이러한 캐스팅을 사용하는 방법은 다음과 같습니다.

```
(원하는 캐스팅 종류)<바꾸려는 타입>(무엇을 바꿀 것인가?)
```

예를 들어서, `static_cast` 로 `float` 타입의 `float_variable` 이라는 변수를 `int` 타입의 변수로 타입 변환하기 위해서는;

```
static_cast<int>(float_variable);
```

이렇게 해주시면 됩니다. 이는 C 언어에서

```
(int)(float_variable)
```

을 한 것과 동일한 문장입니다. 사실 현재까지 배운 내용 정도에서는, `static_cast` 만 사용하지 나머지 캐스팅들은 별로 신경 안쓰셔도 됩니다. 여러분이 C 언어에서 수행하였던 대부분의 아무런 문제없는 캐스팅들은 모두 `static_cast` 로 해주시면 됩니다. 강좌를 진행하면서 나머지 캐스팅들을 어떠한 상황에서 사용하는지 차근 차근 알아보도록 할 것입니다.

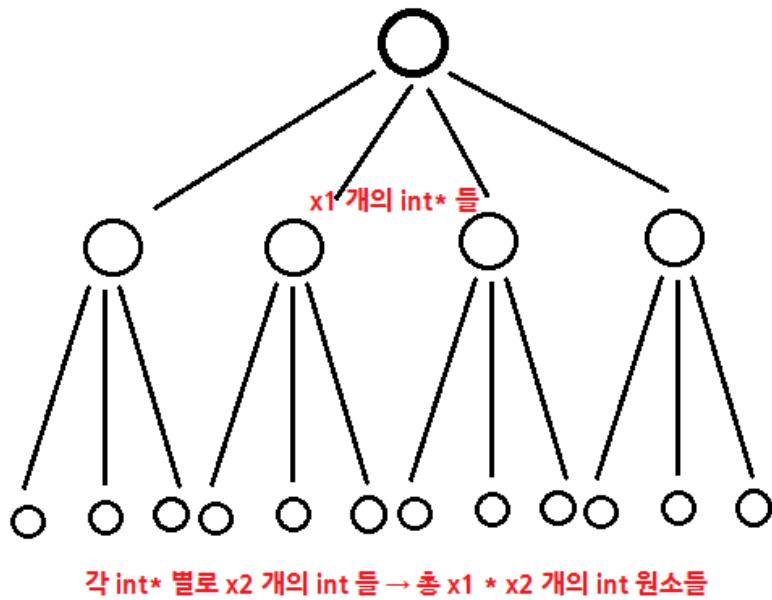
N 차원 배열 만들기

N 차원 배열을 구현하는 방법은 크게 두 가지 방법이 있다고 생각합니다. 사용자가 원하는 배열을 $\text{arr}[x_1][x_2] \dots [x_n]$ 이라고 해본다면, 첫 번째 방법은 말 그대로 $x_1 * x_2 * \dots * x_n$ 크기의 일 차원 배열을 할당한 뒤에 접근할 때 정확한 위치를 찾아주는 방법이지요. 이러한 방식으로 구현한다면 메모리도 정확히 필요한 만큼만 사용할 수 있기에 좋은 방법이라고 생각합니다. 이 방법으로 한번 여러분들이 직접 구현해 보시기 바랍니다.

제가 이 N 차원 배열을 구현하는 프로젝트에서 사용할 아이디어는, 이전에 2 차원 배열의 동적할당을 수행하면서 얻은 아이디어와 비슷합니다. 예전에 동적으로 2 차원 배열을 구현할 때 다음과 같이 구성하였습니다. (참고로 아래 코드에서 할당한 2 차원 배열의 크기는 $\text{arr}[x_1][x_2]$ 입니다)

```
int** arr;
arr = new int*[x1];
for (int i = 0; i < x1; i++) arr[i] = new int[x2];
```

즉 더블 포인터 arr 을 정의한 뒤에, arr 에 int^* 타입의 x_1 크기의 1 차원 배열을 먼저 할당한 다음에, 이 int^* 배열의 각 원소에 대해서 또 x_2 크기의 1 차원 배열을 모두 할당한 것이지요. 이와 같은 형태를 그림으로 표현하자면 다음과 같습니다!



위와 같이 문어발 형식으로 맨 처음에는 x_1 개의 int^* 배열을 생성한 뒤에, 각 int^* 에 대해서 x_2 개의 int 배열을 만들게 된다면, 전체적으로 볼 때 마치 $\text{int arr}[x_1][x_2]$ 를 한 것과 정확히 동일한 효과를 낼 수 있게 됩니다. 하지만 이와 같은 방식의 문제점으로는 원래 $\text{int arr}[x_1][x_2]$

를 하게 된다면 정확히 $x_1 * x_2$ 만큼의 메모리만 잡아먹게 되지만, 이 방법을 할 경우, 포인터 자체가 잡아먹는 크기 때문에 $x_1 * x_2 + x_1 + 1$ 만큼의 메모리를 잡아먹게 된다는 뜻입니다.

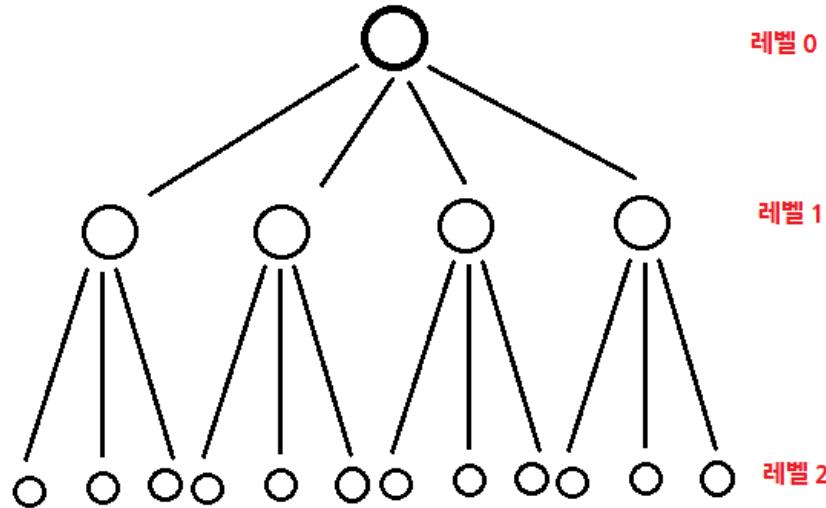
하지만 위 방식의 좋은 점은, 메모리가 허용하는 한 크기가 매우 매우 큰 배열도 생성할 수 있다는 점입니다. 첫 번째 방식의 경우 전체 배열의 원소 수가 `int` 크기를 넘어가게 된다면, 따로 큰 수의 정수를 다룰 수 있는 라이브러리를 사용해서 메모리를 동적으로 할당해주어야 할 것입니다. 하지만, 제가 이 N 차원 배열 클래스에서 사용할 방식의 경우, 전체 원소 수가 아니라, 한 차원의 수가 `int` 크기만 넘어가지 않으면 됩니다. 다시 말해, `(int 크기) * (int 크기) * ... * (int 크기)` 개의 원소를 사용할 수 있으며, 아마 이 정도 크기는 웬만한 사이즈의 프로젝트에서는 충분할 것입니다. 3 차원 배열의 경우 가능한 최대 원소 개수가 2 의 96승, 즉 79228162514264337593543950336 개나 됩니다.

그런데 여기서 한 가지 문제점이 무엇이냐면, 우리가 만들어야 할 배열은 정해진 상수 차원의 배열이 아니라, N 차원의 배열이라는 뜻입니다. 만일 3 차원 배열을 만들었다면 `int***` 을 이용하였을 것이고 4 차원 배열은 `int ****` 을 이용하였을 터인데 (물론 불편하기는 하지만, 쉽게 생각하자면 말입니다.) N 차원 배열의 경우 N 개의 `*` 들이 들어간 포인터를 정의할 수 없는 터입니다.

하지만 관점을 바꾸어서 조금만 생각해보면 이 문제는 손쉽게 해결할 수 있음을 알 수 있습니다. 위에서 포인터를 사용하는 것이 단순히 다음 레벨의 배열들을 가리키기 위함이라면, 굳이 N 포인터를 사용하지 않고도 만들 수 있기 때문입니다.

```
struct Address {  
    int level;  
    void* next;  
};
```

이 생각을 바탕으로 하나의 작은 구조체를 만들어 보자면 위와 같이 `Address` 라는 구조체를 생각해봅시다.!

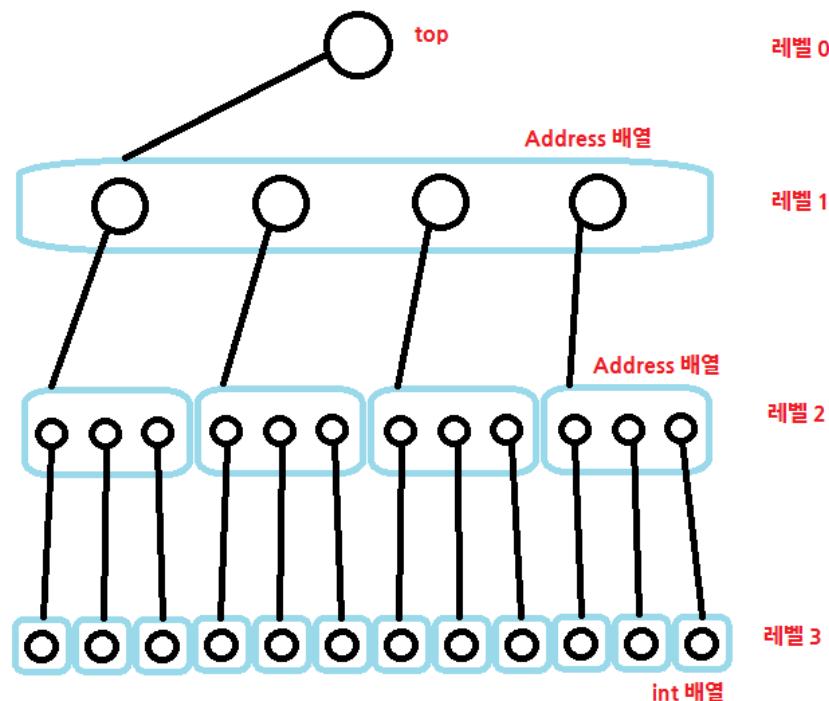


위와 같이 각각의 행들을 한 개의 레벨이라고 생각했을 때, 맨 처음에 우리가 정의하던 `int **` 변수는 0 레벨, 그 다음에 `int **` 가 가리키던 `int *` 배열들은 1 레벨, 그리고 실제 `int` 형 데이터가 들어가 있는 곳은 2 레벨이라고 생각할 수 있습니다. 그런데 Address 구조체를 도입한다면 굳이 `int **` 등으로 귀찮게 할 필요 없이 모두 `void *` 포인터 하나로 정리할 수 있습니다.

이 것이 어떻게 가능하다면, `top` 이라는 Address 객체를 도입을 합니다. 이 `top` 은 위 그림에서 맨 위의 레벨 0 에 해당하며 (그래서 이름도 역시 `top` 입니다) 따라서 `top` 의 `level` 값은 0 이 됩니다. 그렇다면 이 `top` 의 `next` 에는 무엇이 들어가게 될까요? 이미 예상하겠지만, 레벨이 1 인 Address 배열의 시작 주소가 들어가게 됩니다. 그럼, 이 `top` 이 가리키고 있는 Address 배열의 각각 원소들의 `level` 은 당연히 1 이 되겠고, 이들의 `next` 에는 무엇이 들어갈까요. 예상했던 대로, 이번에는 Address 배열이 아닌, `int` 배열의 시작 주소가 들어가겠지요. (왜냐하면 2 차원 배열이기 때문이죠!. 실질적으로 데이터는 여기에 보관이 됩니다)

여기서 좋은 점은 포인터라는 것이 타입의 상관없이 모두 `void *` 으로 값을 보관할 수 있으므로 필요할 때에만 적당한 포인터 타입으로 변환하면 됩니다. 정리해보자면, N 차원 배열이라고 할때 Address 들은 총 0 레벨 부터 N - 1 레벨 까지 생성되며, N - 1 레벨의 경우 `next` 에 실제로 보관할 데이터에 해당하는 배열(여기서는 `int`)의 시작 주소값이 들어가게 되고, 나머지 0 부터 N - 2 레벨 까지는 그 다음 레벨의 Address 배열의 시작 주소값이 들어가게 됩니다.

마지막으로 이해가 가지지 않는 분들을 위해 3차원 배열의 경우 어떻게 이 방법으로 구성할 수 있는지 예시 그림을 첨부하였습니다.!



위 그림은 3차원 배열일 때 어떻게 구성할 수 있는지 나타낸 그림입니다. 그림에도 잘 표현되어 있지만, 검은색 선은 `next` 가 가리키고 있는 것을 의미하고, 파란색 테두리는 하나의 배열을 의미하게 됩니다. 마찬가지로 0 레벨의 `next` 는 1 레벨의 Address 배열의 시작 주소값을 가리키고 있고, 1 레벨의 Address 들의 `next` 는 각각 2 레벨의 Address 배열의 시작 주소값을 가리키고 있습니다. 이 때 3 차원 배열이므로, (3 - 1) 레벨인 2 레벨의 Address 들의 `next` 들은 0, 1 레벨들과는 다르게 int 배열의 시작 주소값을 가리키게 됩니다.

이러한 레벨 방식을 도입해서 처리하는 이유는 각 레벨에서의 배열 크기가 모두 다를 수 있기 때문입니다. 예를 들어서, `arr[3][2][1]` 을 했을 경우, 1 레벨의 배열 크기가 3, 2 레벨의 배열 크기가 2, 그리고 마지막 int 배열 (레벨 3에 해당) 크기가 1 이 되면 됩니다. 위 그림의 경우 `arr[4][3][1]` 을 나타낸 것이라 볼 수 있겠지요.

이러한 아이디어를 바탕으로 일단 우리의 N 차원 Array 배열의 클래스를 대략적으로 설계해보도록 합시다.

```

class Array {
    const int dim; // 몇 차원 배열인지
    int* size; // size[0] * size[1] * ... * size[dim - 1] 짜리 배열이다.

public:
    Array(int dim, int* array_size) : dim(dim) {
        size = new int[dim];
        for (int i = 0; i < dim; i++) size[i] = array_size[i];
    }
}

```

```
| };
```

일단 우리의 `Array` 배열에 들어가야 할 중요한 정보로 '몇 차원' 배열인지에 대한 정보와, 각 차원에서의 크기 정보를 반드시 포함하고 있어야만 할 것입니다. 따라서, '몇 차원' 인지는 `dim`에 아예 상수값으로 저장하도록 하고 (반드시 상수로 정할 필요는 없습니다만, `dim`을 상수로 정한 것은 외부 사용자들에게 한 번 `Array`의 차원을 정하면 바꿀 수 없다는 것을 의미합니다. 물론 여러분들이 원하신다면 굳이 상수로 안하는 대신에 배열의 차원을 조절해주는 `resize` 같은 함수를 추가해주어야 겠지요), `size` 배열에 각 차원에 대한 정보를 가지게 하였습니다.

그런데 여기서 중요한 것이 빠진 것 같습니다. 바로 실질적으로 데이터를 보관하는 부분인데요, 앞에서도 설명하였듯이 우리의 거대한 N 차원 배열은 마치 거대한 나무처럼 가느다란 줄기로 부터 시작해서 엄청나게 큰 뿌리로 퍼지는 모습입니다. 하지만 이 거대한 배열을 가리키게 위해서 `Array`에서 필요한 것은 단 하나, 바로 맨 상단의 시작점일 뿐이지요. 이 시작점은 `Address *` 타입으로, 이를 `top`이라고 부르기로 하였습니다. 따라서 최종적으로 `Array`에 필요한 `data` 멤버들은 다음과 같습니다.

```
class Array {
    const int dim; // 몇 차원 배열인지
    int* size; // size[0] * size[1] * ... * size[dim - 1] 짜리 배열이다.

    Address* top;

public:
    Array(int dim, int* array_size) : dim(dim) {
        size = new int[dim];
        for (int i = 0; i < dim; i++) size[i] = array_size[i];
    }
};
```

아 물론, `Address`라는 새로운 구조체를 도입하였기 때문에 `Address`의 정의 자체도 넣어야만 합니다. 한 가지 재미있는 점은 클래스 안에도 클래스를 넣을 수 있다는 사실인데, 외부에서 우리 `Array` 배열이 내부적으로 어떻게 작동하는지 공개하고 싶지 않고, 또 내부 정보에 접근하는 것을 원치 않기 때문에 `Array` 안에 `Address` 구조체를 넣어 버리겠습니다. (참고로 C++에서 구조체는 모든 멤버 함수, 변수가 디폴트로 `public`인 클래스라고 생각하시면 됩니다)

```
class Array {
    const int dim; // 몇 차원 배열인지
    int* size; // size[0] * size[1] * ... * size[dim - 1] 짜리 배열이다.

    struct Address {
        int level;
        // 맨 마지막 레벨(dim - 1 레벨)은 데이터 배열을 가리키고, 그 위 상위
        // 레벨에서는 다음 Address 배열을 가리킨다.
        void* next;
    };
};
```

```

};

Address* top;

public:
Array(int dim, int* array_size) : dim(dim) {
    size = new int[dim];
    for (int i = 0; i < dim; i++) size[i] = array_size[i];
}
};

```

따라서 최종적으로 위와 같은 모습이 됩니다.

자 그러면 이제 본격적으로 `top` 을 시작으로 N 차원 배열을 생성해보도록 하겠습니다. 위의 그림과 같은 구조를 구현하기 위해서는 이전에 동적으로 2 차원 배열을 생성하였을 때처럼 `for` 문으로 간단히 수행할 수 있는 것이 아닙니다. 왜냐하면 일단 `for` 문으로 하기 위해서는 몇 중 `for` 문을 사용할지 컴파일 시에 정해져야 하는데, 이 경우 N 차원인 임의의 차원이므로 그럴 수 없기 때문입니다.

하지만 이와 같은 문제를 해결하는 아주 좋은 아이디어가 있는데 바로 재귀 함수를 이용하는 것입니다. 재귀 함수를 구성하기 위해서는 다음과 같은 두 가지 스텝만 머리속으로 생각하고 있으면 됩니다.

- 함수에서 처리하는 것, 즉 현재 단계에서 다음 단계로 넘어가는 과정은 무엇인가?
- 재귀 호출이 종료되는 조건은 무엇인가?

일단 우리는 두 번째 질문에 대한 해답을 이미 알고 있습니다. 재귀 함수 호출이 종료되기 위한 조건은 바로 현재 처리하고 있는 `Address` 배열의 레벨이 $(\text{dim} - 1)$ 이면 됩니다. 즉, `Address` 배열의 레벨이 $(\text{dim} - 1)$ 이면, 이 배열의 원소들 (즉 $(\text{dim} - 1)$ 레벨들의 `Address` 들) 의 `next`에는 `int` 배열의 데이터가 들어가게 재귀 호출이 끝나게 되지요.

그렇다면 첫 번째 질문에 대한 답, 즉 현재 단계에서 다음 단계로 넘어 가는 과정은 무엇일까요? 이 역시 사실 간단합니다. 현재 n 레벨의 `Address` 배열이라면, 이들의 `next`에 다음 레벨인 $n + 1$ 레벨의 `Address` 배열을 지정해주고, 또 이 각각의 원소에 대해 처리하도록 하면 되는 것입니다.

따라서 이 생각들을 정리하면 다음과 같은 코드를 짤 수 있습니다.

```

// address 를 초기화 하는 함수이다. 재귀 호출로 구성되어 있다.
void initialize_address(Address *current) {
    if (!current) return;
    if (current->level == dim - 1) { // 두 번째 질문 (종료 조건)
        current->next = new int[size[current->level]];
        return;
    }
    current->next = new Address[size[current->level]];
    for (int i = 0; i != size[current->level];

```

```
i++) { // 다음 단계로 넘어가는 과정
    static_cast<Address *>(current->next) + i)->level = current->level + 1;

    initialize_address(static_cast<Address *>(current->next) + i);
}
}
```

위 `initialize_address` 함수는 인자로 들어온 `current` 를 처리하고 그 다음 단계로 넘어가도록 합니다. 맨 처음의 `if(!current)` 부분은 단순히 `current` 가 `NULL` 일 때 예외적으로 처리하는 부분이니까 무시하도록 하고, 두 번째 `if` 문은 위에서 설명한 '종료 조건' 이 됩니다. 이 채귀 호출이 끝나기 위한 조건으로 현재 처리하고 있는 `Address` 의 레벨이 `dim - 1` 일 때, 이들의 `next` 에는 `Address` 배열이 아닌 `int` 배열의 시작 주소가 들어가게 됩니다.

반면에 종료 조건에 해당하지 않는 경우 `initialize_address` 함수에서 어떻게 처리하는지 볼까요. 일단;

```
current->next = new Address[size[current->level]];
```

를 통해서 `current` 의 `next` 에 크기가 `size [current->level]` 인 새로운 시작 주소값을 만들어주고 있습니다. 이 때 왜 배열의 크기가 `size [current->level]` 인지는 여러분도 잘 아실 것이라 생각합니다. 예를 들어서 `arr[3][4][5]` 의 경우 `current` 가 0 레벨이라면 `next` 에 만드는 배열의 크기는 3, 1 레벨이라면 4, 2 레벨이라면 5 가 되는 것과 같은 이치입니다.

이렇게 `Address` 배열을 만들게 된다면, 이 각각의 원소들에 대해서도 종료 조건에 도달하기 전까지 동일한 처리를 계속 반복해주어야만 하겠지요? 따라서 아래처럼 `for` 문으로

```
for (int i = 0; i != size[current->level]; i++) { // 다음 단계로 넘어가는 과정
    static_cast<Address *>(current->next) + i)->level = current->level + 1;
    initialize_address(static_cast<Address *>(current->next) + i);
}
```

새롭게 생성한 각각의 원소들에 대해 `initialize_address` 를 동일하게 수행하고 있습니다.

```
(static_cast<Address *>(current->next) + i)->level = current->level + 1;
```

위 처럼 그 `current` 의 `next` 가 가리키고 있는 원소들의 레벨 값을 다음 단계로 설정한 다음에

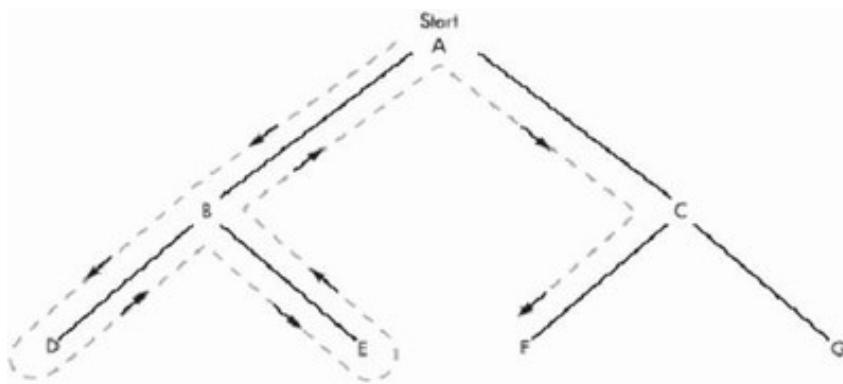
```
initialize_address(static_cast<Address *>(current->next) + i);
```

각각의 원소들에 대한 `initialize_address` 함수를 호출하게 됩니다. 참고로,

```
(static_cast<Address *>(current->next) + i)
```

라는 표현이 무슨 뜻인지는 C 언어를 충실히 배운 여러분이라면 무슨 의미인지 잘 알고 계실 것이라 생각합니다. (`current->next` 를 시작 주소로 하는 `Address` 배열의 `i` 번째 원소를 가리키는 포인터)

참고로 말하자면, 이러한 방식으로 함수를 재귀호출하게 된다면, 맨 위의 그림을 '깊이 우선 탐색' 하는 것과 동일합니다. (아래 그림 참조)



생성자를 만들었으므로, 소멸자도 비슷한 방식으로 만들어주면 됩니다. 다만, 소멸자의 경우 주의할 점이, 생성자는 '위에서 아래로' 메모리들을 점차 확장 시켜 나갔지만, 소멸자는 '아래에서 위로' 메모리를 점차 소멸시켜 나가야 된다는 점입니다. 물론, 이로 살짝 바꾸는 것은 별로 어려운 일이 아닙니다.

```
void delete_address(Address *current) {
    if (!current) return;
    for (int i = 0; current->level < dim - 1 && i < size[current->level]; i++) {
        delete_address(static_cast<Address *>(current->next) + i);
    }

    delete[] current->next;
}
```

위와 같이 `delete_address` 함수를 생각해 볼 수 있습니다. (여러분들이 한 번 직접 생각해보세요)

이들을 조합해서, 우리의 `Array` 클래스의 생성자를 수정해보도록 합시다.

```
Array(int dim, int* array_size) : dim(dim) {
    size = new int[dim];
    for (int i = 0; i < dim; i++) size[i] = array_size[i];
```

```

top = new Address;
top->level = 0;

initialize_address(top);
}

Array(const Array& arr) : dim(arr.dim) {
    size = new int[dim];
    for (int i = 0; i < dim; i++) size[i] = arr.size[i];

    top = new Address;
    top->level = 0;

    initialize_address(top);
}

~Array() {
    delete_address(top);
    delete[] size;
}

```

위 두개는 `Array` 의 기본 생성자와 복사 생성자, 그리고 소멸자를 나타낸 것입니다. 재귀 함수의 시작으로 `current`에 `top`을 전달하였고, 이 `top`을 시작으로 함수들이 쭈르륵 재귀 호출되면서 거대한 N 차원 메모리 구조가 생성되거나 소멸됩니다. 항상 유의할 점은 소멸자에서 동적으로 할당한 모든 것들을 정리해 주어야 한다는 점인데, 재귀 호출로 생성한 메모리 구조만을 소멸해야 되는 것이 아니라 `size` 역시 동적으로 할당한 것이므로 꼭 해제해 주어야만 합니다.

operator[] 문제

이제 생성을 하였는데, 문제는 어떻게 N 차원 배열의 각각의 원소에 접근하느냐입니다. 우리의 클래스는 다른 복잡한 방법을 사용하지 않고 마치 진짜 배열을 다루던 것처럼 `[]`를 이용해서 원소에 접근하는 기능을 제공하고 싶습니다. 하지만 문제는 C++에는 1 개의 `[]`를 취하는 연산자는 있어도 N 개의 `[]`들을 취하는 연산자는 없다는 점입니다.

그렇다면, 여러개의 `[]`들을 어떻게 처리하느냐면 예를 들어 우리가

```
arr[1][2][3][4]
```

를 하였을 때, 제일 먼저 `arr[1]`이 처리되며, 첫 번째 차원으로 1을 선택했다는 정보가 담긴 어떠한 객체 `T`를 리턴합니다. 그리고,

```
(T)[2][3][4]
```

가 수행이 되겠지요. 이 `T` 또한 `operator[]` 가 있어서, 두번째 차원으로 2를 선택했다는 정보가 담긴 객체 `T'`을 리턴합니다. 그렇다면 이제,

```
(T') [3] [4]
```

가 되겠고, 마찬가지로 계속 진행하게 된다면

```
T'''
```

이 남게 됩니다. 우리는 이 T''' 가 int 타입임을 바라고 있지요. 그런데 도대체 이를 어떻게 구현해야 할까요?

일단 Array 가 아닌 새로운 타입의 객체를 만들어야 한다는 것만은 분명합니다. 왜냐하면, 만일 operator[] 가 Array& 타입이라면, 1 차원 Array 배열에 대해서

```
arr[1] = 3;
```

과 같은 문장은 말이 안되기 때문입니다. 그렇다고 해서 operator[] 가 int& 타입을 리턴할 수도 없는 처지입니다. 왜냐하면, 만일 int& 타입을 리턴하였을 경우에 1 차원 배열인

```
arr[1] = 3;
```

과 같은 문장은 쉽게 처리할 수 있다고 하지만, 그 보다 고차원 배열에 대해서

```
arr[1][2] = 3;
```

은 어떻게 처리할 것인가요? arr[1] 의 리턴 타입이 int& 라면 int 에 대한 operator[] 는 정의되어 있지도 않고 정의 할 수도 없습니다. '그렇다면 상황에 따라서 1 차원이면 int 를, 그 보다 고차원 배열이면 다른 것을 리턴하면 되지 않느냐?' 라고 물을 수 있지만 '오버로딩'의 원칙 상 동일한 인자를 받는 함수에 대해서는 한 가지 리턴 타입만이 가능합니다.

하지만 조금만 기억을 더듬어 올라간다면, 필요할 때 int 처럼 작동하지만 int 가 아닌 클래스를 만들 수 있었습니다. 바로 int 의 Wrapper 클래스였지요. int 의 Wrapper 클래스는 타입 변환 연산자를 제공해서 int 와의 연산을 수행하거나, 대입등을 할 때 마치 int 처럼 작동하도록 만들 수 있습니다. 그렇다면 우리는 operator[] 가 int 의 Wrapper 클래스 객체를 리턴해서, 실제 int 값에 접근할 때에는 int 변수 처럼 행동하고, 위에서 T 나 T' 처럼 원소에 접근해 가는 중간 단계의 산물일 경우, 그 중간 단계의 정보를 포함하는 것으로 사용하면 됩니다.

이러한 생각을 바탕으로 int 의 Wrapper 클래스 Int 의 열개를 그려보자면 다음과 같습니다.

```
class Int {
    void* data;

    int level;
    Array* array;
};
```

먼저 `level` 정보는 반드시 포함하고 있어야만 합니다. 왜냐하면, 이 `Int` 가 맨 마지막 '실제 `int` 정보'를 포함하고 있는 객체인지, 아니면 원소를 참조해 나가는 중간 과정에서의 산물인지를 구별할 수 있어야 하기 때문입니다.

예를 들어서

```
arr[1][2];
```

를 생각해 볼 때 맨 처음 `arr[1]` 은 `level` 이 1 인 `Int` 가 리턴됩니다. 이 때, 이는 `int` 데이터가 아니라, `[1][2]` 를 참조해 나가기 위한 중간 과정이지요. (이 것을 `Int` 가 어떻게 구별하느냐면, `Int` 가 가지고 있는 `array` 의 `dim` 정보를 참조하면 되겠지요!)

이 때의 `Int` 에는 '현재 `arr[1]` 를 가리키고 있음'에 대한 정보가 `Int` 의 `data` 에 들어가 있습니다. 그 다음에 `Int` 의 `operator[]` 를 수행하게 된다면 (따라서 `Int` 클래스의 `operator[]` 역시 만들어야 합니다), 이번에는 `level` 이 2 인 `Int` 가 리턴이 됩니다. 사용자가 `level` 이 2 인 `Int` 에 대입 연산을 하게 된다면, `void * data` 를 `int` 원소를 가리키고 있는 주소로 해석해서 실제로 `int` 변수 처럼 대입이 수행이 되겠지요.

참고로 `array` 는 어떤 배열의 `Int` 인지 가리키는 역할을 합니다.

먼저 `Int` 의 생성자는 아래와 같이 구성할 수 있습니다.

```
Int(int index, int _level = 0, void *_data = NULL, Array *_array = NULL)
    : level(_level), data(_data), array(_array) {
    if (_level < 1 || index >= array->size[_level - 1]) {
        data = NULL;
        return;
    }
    if (level == array->dim) {
        // 이제 data에 우리의 int 자료형을 저장하도록 해야 한다.
        data = static_cast<void *>(
            static_cast<int *>(static_cast<Array::Address *>(data)->next) + index);
    } else {
        // 그렇지 않을 경우 data에 그냥 다음 addr을 넣어준다.
        data = static_cast<void *>(
            static_cast<Array::Address *>(static_cast<Array::Address *>(data)->next) +
            index);
    }
};
```

Int 생성자의 내용을 설명하기 전에, 위에 Int 생성자의 인자로 이상한 것들이 보이지요? 왜 인자에 값을 미리 대입하고 있는 것인가요?

```
int index, int _level = 0, void *_data = NULL, Array *_array = NULL
```

이들은 모두 디폴트 인자(default argument)라고 부르는 것이며, 함수에 값을 전달해주지 않는다면 인자에 기본으로 이 값들이 들어가게 됩니다. 예를 들어서 우리가 Int 의 생성자에

```
Int(3)
```

이라고 호출하였다면, index 에는 3 이 들어가겠지만, _level 에는 0, _data 과 _array 에는 NULL 이 들어가겠지요. 만약에 인자를 지정해 주었다면, 디폴트 값 대신에 지정한 인자가 들어가게 됩니다. 예를 들어서

```
Int(3, 1)
```

이렇게 한다면 index 에는 3, _level 에는 1, 그리고 나머지에는 디폴트 값인 NULL 이 들어갑니다. 또한 한 가지 당연한 사실이지만, 디폴트 인자들은 함수의 맨 마지막 인자부터 '연속적으로'만 사용할 수 있습니다. 왜냐하면 만일 우리가 디폴트 인자를

```
int index, int _level = 0, void *_data = NULL, Array *_array
```

이렇게 중간에 두었다면 사용자가

```
Int(3, 1, ptr)
```

을 했을 때, 이 ptr 의 의미가 _data 는 디폴트 인자인 NULL 을 사용하고 _array 에는 ptr 을 사용하라는 것인지, 아니면 _data 에 ptr 이 들어가게 _array 에 인자를 안주는 오류인지 컴파일러가 알 수 없기 때문입니다. 이렇게 디폴트 인자를 사용하는 경우는 대부분 프로그래머들의 편의를 위해서인데, 다만 함수의 원형을 정확히 알아야지, 일부 인자를 실수로 누락하게 된다면 디폴트 인자로 경고가 발생하지 않고, 그로 인해 함수가 이상하게 작동할 수 있으므로 주의가 필요합니다.

자 그럼 이제 Int 생성자의 내부를 살펴보도록 하겠습니다.

```
if (_level < 1 || index >= array->size[_level - 1]) {
    data = NULL;
    return;
}
```

일단 위와 같이 오류가 발생하였을 경우 처리하는 모습입니다. 클래스를 구현할 때는 항상 클래스를 사용하는 사용자가 어떠한 이상한 짓을 하더라도 대처할 수 있는 자세가 필요합니다. 위와 같이 꼼꼼하게 발생할 수 있는 예외 상황을 처리하도록 합시다.

```
if (level == array->dim) {
    // 이제 data에 우리의 int 자료형을 저장하도록 해야 한다.
    data = static_cast<void*>(
        static_cast<int*>(static_cast<Array::Address*>(data)->next) + index));
} else {
    // 그렇지 않을 경우 data에 그냥 다음 addr을 넣어준다.
    data = static_cast<void*>(
        static_cast<Array::Address*>(static_cast<Array::Address*>(data)->next) +
        index);
}
```

그 다음 부분을 살펴보자면, 상당히 중요한 부분입니다. 먼저 if 문에서 이 Int의 level과 array->dim이 같다는 것은 무엇을 의미할까요? 이 말은, 원소에 접근하는 단계의 중간 산물이 아니라, 실질적으로 접근이 완료 되었다는 것입니다. 따라서, Int의 data에는 (int*) 타입의 포인터 주소값이 (void*로 다시 캐스팅 되어서) 들어가겠지요. 즉, level == array->dim이 되는 상황은 예컨대 3 차원 배열에서

```
arr[1][2][3];
```

을 하였을 때 arr[1]은 level 1 째리 Int 객체 T를 리턴해서

```
T[2][3]
```

이 되고, T[2]는 level 2 째리 Int 객체 T'을 리턴해서

```
T'[3]
```

이 되고, 다시 T'[3]은 level 3 째리 Int 객체 T''을 리턴하게 되는데, 이 T''의 data가 가리키는 포인터가 이전들과는 다르게 int 변수의 주소값이라는 것이지요. 그렇다면 당연히도 else 부분에서는, data에 다음 Address 값이 들어갑니다.

```
data = static_cast<void*>(
    static_cast<Array::Address*>(static_cast<Array::Address*>(data)->next) +
    index);
```

위와 같이 `data` 를 만들게 된다면, 결과적으로 맨 마지막에서 사용자가 원하는 `int` 데이터를 정확히 찾아낼 수 있겠습니다.

이와 같은 사실을 바탕으로 하면 `Array` 의 `operator[]` 와 `Int` 의 `operator[]` 는 별로 어렵지 않게 만들 수 있습니다. 먼저 `Array` 의 `operator[]` 를 살펴보면

```
Int Array::operator[](const int index) {
    return Int(index, 1, static_cast<void *>(top), this);
}
```

위와 같이 `Int` 를 리턴하게 되며, `level` 로는 1, 그리고 `data` 인자로는 `top` 을 전달합니다. 따라서 `Int` 생성자에서, 생성되는 객체가 `top` 의 `next` 가 가리키고 있는 `index` 번째 원소를 `data` 로 가질 수 있게 되지요.

```
Int operator[](const int index) {
    if (!data) return 0;
    return Int(index, level + 1, data, array);
}
```

`Int` 의 `operator[]` 의 경우, `level` 에 다음 레벨을 전달함으로써 다음 단계의 `Int` 를 생성합니다. 그리고 위의 `!data` 를 검사하는 문장은 만일 `data` 가 NULL 이라면 (즉 예외 상황), 0 을 리턴하도록 하였습니다.

자 이제, `Int` 가 `Wrapper` 클래스로써 동작하기에 가장 필수적인 요소인 타입 변환 연산자를 살펴보면;

```
operator int() {
    if (data) return *static_cast<int *>(data);
    return 0;
}
```

매우 간단합니다. 타입 변환 연산자가 호출되는 상태에서의 `Int` 객체의 `data` 에는 `int` 원소의 주소값이 들어가 있기 때문에 `void*` 를 `int *` 타입으로 변환에서 그 값을 리턴하면 됩니다.

자 그럼, 실제 전체 코드를 살펴보도록 합시다.

```
// 대망의 Array 배열
#include <iostream>

namespace MyArray {
    class Array;
    class Int;

    class Array {
```

```
friend Int;

const int dim; // 몇 차원 배열인지
int* size; // size[0] * size[1] * ... * size[dim - 1] 짜리 배열이다.

struct Address {
    int level;
    // 맨 마지막 레벨(dim - 1 레벨)은 데이터 배열을 가리키고, 그 위 상위
    // 레벨에서는 다음 Address 배열을 가리킨다.
    void* next;
};

Address* top;

public:
    Array(int dim, int* array_size) : dim(dim) {
        size = new int[dim];
        for (int i = 0; i < dim; i++) size[i] = array_size[i];

        top = new Address;
        top->level = 0;

        initialize_address(top);
    }

    Array(const Array& arr) : dim(arr.dim) {
        size = new int[dim];
        for (int i = 0; i < dim; i++) size[i] = arr.size[i];

        top = new Address;
        top->level = 0;

        initialize_address(top);
    }

    // address 를 초기화 하는 함수이다. 재귀 호출로 구성되어 있다.
    void initialize_address(Address* current) {
        if (!current) return;
        if (current->level == dim - 1) {
            current->next = new int[size[current->level]];
            return;
        }
        current->next = new Address[size[current->level]];
        for (int i = 0; i != size[current->level]; i++) {
            static_cast<Address*>(current->next) + i)->level = current->level + 1;
            initialize_address(static_cast<Address*>(current->next) + i);
        }
    }

    void delete_address(Address* current) {
        if (!current) return;
        for (int i = 0; current->level < dim - 1 && i < size[current->level]; i++) {
            delete_address(static_cast<Address*>(current->next) + i);
        }
    }
}
```

```
    delete[] current->next;
}
Int operator[](const int index);
~Array() {
    delete_address(top);
    delete[] size;
}
};

class Int {
    void* data;

    int level;
    Array* array;

public:
    Int(int index, int _level = 0, void* _data = NULL, Array* _array = NULL)
        : level(_level), data(_data), array(_array) {
        if (_level < 1 || index >= array->size[_level - 1]) {
            data = NULL;
            return;
        }
        if (level == array->dim) {
            // 이제 data에 우리의 int 자료형을 저장하도록 해야 한다.
            data = static_cast<void*>(
                static_cast<int*>(static_cast<Array::Address*>(data)->next) + index));
        } else {
            // 그렇지 않을 경우 data에 그냥 다음 addr을 넣어준다.
            data = static_cast<void*>(static_cast<Array::Address*>(
                static_cast<Array::Address*>(data)->next) +
                index));
        }
    };
    Int(const Int& i) : data(i.data), level(i.level), array(i.array) {}

    operator int() {
        if (data) return *static_cast<int*>(data);
        return 0;
    }
    Int& operator=(const int& a) {
        if (data) *static_cast<int*>(data) = a;
        return *this;
    }

    Int operator[](const int index) {
        if (!data) return 0;
        return Int(index, level + 1, data, array);
    }
};

Int Array::operator[](const int index) {
```

```
    return Int(index, 1, static_cast<void*>(top), this);
}
} // namespace MyArray
int main() {
    int size[] = {2, 3, 4};
    MyArray::Array arr(3, size);

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                arr[i][j][k] = (i + 1) * (j + 1) * (k + 1);
            }
        }
    }

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                std::cout << i << " " << j << " " << k << " " << arr[i][j][k]
                    << std::endl;
            }
        }
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
0 0 0 1
0 0 1 2
0 0 2 3
0 0 3 4
0 1 0 2
0 1 1 4
0 1 2 6
0 1 3 8
0 2 0 3
0 2 1 6
0 2 2 9
0 2 3 12
1 0 0 2
1 0 1 4
1 0 2 6
1 0 3 8
```

```

1 1 0 4
1 1 1 8
1 1 2 12
1 1 3 16
1 2 0 6
1 2 1 12
1 2 2 18
1 2 3 24

```

와 같이 제대로 실행됨을 볼 수 있습니다.

한 가지 중요하게 살펴볼 점은, 전체 클래스를 MyArray 라는 이름 공간으로 감쌌다는 점입니다. 이를 통해 혹여라도 다른 라이브러리에서 Array 라는 클래스를 정의하더라도 문제 없을 것입니다.

또한 주목할 점은, 두 개의 클래스를 한 파일에서 사용하기 때문에 클래스의 정의 순서가 매우 중요하다는 점입니다. 소스 상단에

```

class Array;
class Int;

```

와 같이 클래스를 '선언' 하였습니다. 클래스를 선언하지 않는다면, 아래 Array 클래스에서

```
friend Int;
```

를 할 수 없게 됩니다. 왜냐하면 컴파일러 입장에서 Int 가 뭔지 알 틱이 없기 때문입니다. 따라서 friend 선언을 하기 전에, 이와 같이 class Int 를 먼저 맨 위에 선언해서 사용할 수 있도록 해야 합니다. 그럼에도 불구하고, 맨 밑에

```

Int Array::operator[](const int index) {
    return Int(index, 1, static_cast<void *>(top), this);
}

```

를 Array 클래스 안에 넣지 않고 따로 빼 놓은 이유는 Int 를 실제로 '이용' 하기 위해서는 클래스 선언 만으로 충분하지 않기 때문입니다. 클래스 선언을 통해서는 클래스의 내부 정보가 필요가 없는 것들, 예컨대 friend 선언이나 클래스의 포인터를 정의하는 등의 행동만 가능하지, 위 Array 처럼 구체적으로 Int 클래스의 내부 정보 (생성자) 를 사용하는 경우에는 반드시 Int 클래스의 정의가 선행 되어야만 합니다. 따라서 어쩔 수 없이 Array 클래스의 operator[] 만 따로 빼 놓았습니다.

자. 여러분들은 아주 훌륭한 N 차원 Array 클래스를 제작하게 된 것입니다. 정말 놀랍지 않나요? C 언어 배우던 시절에는 정말 상상 조차 할 수 없는 위력적인 기능이 아닐 수 없습니다. C++ 에서

연산자 오버로딩을 지원한 덕택에 이러한 것들을 만들 수 있게 된 것입니다. 하지만, 사실 약간 불편한 점은 하나 있습니다. 모든 원소에 접근하려면 N 중 `for` 문을 사용해주어야만 합니다.

사실 2차원이나 3차원 정도의 배열을 사용한다면 2~3 중 `for` 문을 사용하는 것은 기꺼히 승낙할 수 있습니다. 하지만 여러분들 4 중 `for` 문은 한 번이라도 돌려보셨나요? 아마도 이 정도 `for` 문을 중첩해서 돌린다면 보기도 안좋을 뿐더러 복잡하기만 할 것입니다. 따라서, 우리의 `Array` 클래스에 또다른 기능으로, 모든 원소들을 순차적으로 접근할 수 있는 반복자(iterator)라는 것을 추가해보도록 할 것입니다.

이를 위해 `Array`에 `Iterator`라는 클래스를 추가할 것입니다.

```
class Iterator {
    int* location;
    Array* arr;
}
```

이 `iterator`는 배열의 특정한 원소에 대한 포인터라고 생각하면 됩니다. C 언어에서 배열의 어떤 원소를 가리키고 있는 포인터 `ptr`에 `ptr ++` 을 했다면 다음 원소를 가리켰듯이, 반복자 `itr`이 `Array`의 어느 원소를 가리키고 있을 때 `itr ++` 을 하면 그 다음 원소를 가리키게 됩니다. 따라서 사용자는 N 중 `for` 문을 사용해서 전체 원소를 참조하는 방법 보다는 단순히 `itr` 을 이용해서 `Array`의 첫 원소부터 `itr ++` 을 통해 마지막 원소까지 가리킬 수 있게 할 것입니다.

이를 위해서, 우리의 `Iterator` 클래스에, 현재 `Iterator` 가 어떤 원소를 가리키고 있는지에 대한 정보를 멤버 변수로 가지게 하겠습니다. 이는 `int * location`에 배열로 보관되는데, 예를 들어 3 차원 배열에서 `Iterator` 가

```
arr[1][2][3]
```

을 가리키고 있다면 `location` 배열에는 {1, 2, 3} 이렇게 들어가게 되는 것이지요. 상당히 단순한 방법이지요? 그렇기 때문에 `operator++()` 함수 자체도 매우 간단하게 만들 수 있습니다.

```
Iterator& operator++() {
    if (location[0] >= arr->size[0]) return (*this);

    bool carry = false; // 받아 올림이 있는지
    int i = arr->dim - 1;
    do {
        // 어차피 다시 돌아온다는 것은 carry 가 true
        // 라는 의미 이므로 ++ 을 해야 한다.
        location[i]++;
        if (location[i] >= arr->size[i] && i >= 1) {
            // i 가 0 일 경우 0 으로 만들지 않는다 (이려면 begin 과 중복됨)
            location[i] -= arr->size[i];
            carry = true;
        }
    } while (carry);
}
```

```

        i--;
    } else
        carry = false;

} while (i >= 0 && carry);

return (*this);
}

```

어떻게 위와 같은 코드가 나왔냐면 예를 들어 우리가 [2][3][4]의 크기를 가지는 배열을 선언했다고 해봅시다. 그리고 어떤 *itr* 가 현재 원소 [1][1][3] 을 가리키고 있다고 해봅시다. 그럼 *itr* ++ 을 하게 된다면 [1][1][4] 가 되는 것이 아니라, 받아 올림이 되며 [1][2][0] 이 되겠지요. 이번에는 *itr* 가 [0][2][3] 인 상태에서 *itr* ++ 을 하면 어떨까요? 일단 [0][2][4] 가 되는 것이 아니라 1 받아 올림 되며 [0][3][0] 이 되는데, 3 역시 받아 올림 되서 [1][0][0] 이 되겠지요? 이와 같은 과정을 위에서 do - while 문으로 처리하였습니다. bool 변수 carry 는 '받아 올림이 있다' 라는 의미입니다.

참고로 [1][2][3] 은 이 배열의 맨 마지막 원소가 됩니다. 그런데 여기서 *itr* ++ 을 하면, 원칙상 [0][0][0] 이 되어야 하는데, 이렇게 된다면, 이 배열을 사용하는 사람 입장에서 상당히 골치 아파집니다. 왜냐하면 C++ 의 거의 대부분의 라이브러리에서 그러하지만 어떠한 배열의 시작(begin)은 맨 첫번째 원소를 의미하고, 마지막(end)은 맨 마지막 원소 바로 다음을 의미하기 때문입니다.

만일 우리가 배열의 처음 부터 마지막 까지 쭉 참조해 나가고 싶은데, 마지막 원소 다음에 다시 맨 처음 원소로 돌아온다면 for 문에 입장에서 이게 다시 돌아온 것인지, 새로 시작하는 것인지 구별을 할 수 없기 때문입니다. 하지만 마지막 원소 다음을 '마지막' 이라 한다면, for 문의 조건문으로 '마지막에 도달하면 끝내라' 이렇게 명령을 한다면 쉽게 해결할 수 있습니다.

그러한 이유에서, 우리의 iterator 는 [1][2][3] 다음에는 [0][0][0] 이 아닌 [2][0][0] 이라 하고, (물론 여기에 해당하는 원소는 당연히 없습니다.) 이를 '마지막' 이라 하기로 하였습니다. 물론 마지막의 값을 참조하려고 하면 오류가 발생하겠지요.

이렇기 때문에 do - while 문 안에서도 특별히

```

if (location[i] >= arr->size[i] && i >= 1) {
    // i 가 0 일 경우 0 으로 만들지 않는다 (이러면 begin 과 중복됨)
    location[i] -= arr->size[i];
    carry = true;
    i--;
} else
    carry = false;

```

에서 location[i] >= arr->size[i] 말고도 i >= 1 이라는 특별한 조건을 넣어서 처리하도록 하였습니다.

참고로, 전위 증가 연산자를 만들었으므로 후위 증가 연산자도 만드는 것을 잊지 마세요.

```
Iterator operator++(int) {
    Iterator itr(*this);
    ++(*this);
    return itr;
}
```

그렇다면 가장 중요한 * 연산자는 어떨까요. (*itr) 을 통해 실제 데이터에 접근해야 하므로, Int 를 리턴하게 됩니다. 따라서 그 모양은 다음과 같겠지요.

```
Int Array::Iterator::operator*() {
    Int start = arr->operator[](location[0]);
    for (int i = 1; i <= arr->dim - 1; i++) {
        start = start.operator[](location[i]);
    }
    return start;
}
```

그냥 우리가 arr[][][] 해주던 일을 그냥 for 문으로 하는 것에 불과하다는 점을 알 수 있습니다. 매우 간단하지요.

이제 Array 클래스에 현재 배열의 시작과 끝을 Iterator 객체로 리턴해주는 것만 만들어주면 됩니다. 각각을 begin 와 end 라고 해보면;

```
Iterator begin() {
    int* arr = new int[dim];
    for (int i = 0; i != dim; i++) arr[i] = 0;

    Iterator temp(this, arr);
    delete[] arr;

    return temp;
}
Iterator end() {
    int* arr = new int[dim];
    arr[0] = size[0];
    for (int i = 1; i < dim; i++) arr[i] = 0;

    Iterator temp(this, arr);
    delete[] arr;

    return temp;
}
```

매우 단순합니다. begin 은 그냥 {0, 0, ... 0} 인 Iterator 를 리턴해주면 되는 것이고, end 는 {size[0], 0, 0, .. 0} 을 인 Iterator 를 리턴해주면 되는 것이니까요.

그럼 전체 소스 코드는 아래와 같습니다.

```
#include <iostream>

namespace MyArray {
    class Array;
    class Int;

    class Array {
        friend Int;

        const int dim; // 몇 차원 배열인지
        int* size; // size[0] * size[1] * ... * size[dim - 1] 짜리 배열이다.

        struct Address {
            int level;
            // 맨 마지막 레벨(dim - 1 레벨)은 데이터 배열을 가리키고, 그 위 상위
            // 레벨에서는 다음 Address 배열을 가리킨다.
            void* next;
        };

        Address* top;

        public:
        class Iterator {
            int* location;
            Array* arr;

            friend Int;

            public:
            Iterator(Array* arr, int* loc = NULL) : arr(arr) {
                location = new int[arr->dim];
                for (int i = 0; i != arr->dim; i++)
                    location[i] = (loc != NULL ? loc[i] : 0);
            }
            Iterator(const Iterator& itr) : arr(itr.arr) {
                location = new int[arr->dim];
                for (int i = 0; i != arr->dim; i++) location[i] = itr.location[i];
            }
            ~Iterator() { delete[] location; }
            // 다음 원소를 가리키게 된다.
            Iterator& operator++() {
                if (location[0] >= arr->size[0]) return (*this);

                bool carry = false; // 받아 올림이 있는지
                int i = arr->dim - 1;
                do {
                    // 어차피 다시 돌아온다는 것은 carry 가 true
                    // 라는 의미 이므로 ++ 을 해야 한다.
                    location[i]++;

```

```
    if (location[i] >= arr->size[i] && i >= 1) {
        // i 가 0 일 경우 0 으로 만들지 않는다 (이러면 begin 과 중복됨)
        location[i] -= arr->size[i];
        carry = true;
        i--;
    } else
        carry = false;

} while (i >= 0 && carry);

return (*this);
}

Iterator& operator=(const Iterator& itr) {
    arr = itr.arr;
    location = new int[itr.arr->dim];
    for (int i = 0; i != arr->dim; i++) location[i] = itr.location[i];

    return (*this);
}

Iterator operator++(int) {
    Iterator itr(*this);
    ++(*this);
    return itr;
}

bool operator!=(const Iterator& itr) {
    if (itr.arr->dim != arr->dim) return true;

    for (int i = 0; i != arr->dim; i++) {
        if (itr.location[i] != location[i]) return true;
    }

    return false;
}

Int operator*();
};

friend Iterator;
Array(int dim, int* array_size) : dim(dim) {
    size = new int[dim];
    for (int i = 0; i < dim; i++) size[i] = array_size[i];

    top = new Address;
    top->level = 0;

    initialize_address(top);
}

Array(const Array& arr) : dim(arr.dim) {
    size = new int[dim];
    for (int i = 0; i < dim; i++) size[i] = arr.size[i];

    top = new Address;
```

```
top->level = 0;

    initialize_address(top);
}
// address 를 초기화 하는 함수이다. 재귀 호출로 구성되어 있다.
void initialize_address(Address* current) {
    if (!current) return;
    if (current->level == dim - 1) {
        current->next = new int[size[current->level]];
        return;
    }
    current->next = new Address[size[current->level]];
    for (int i = 0; i != size[current->level]; i++) {
        static_cast<Address*>(current->next) + i)->level = current->level + 1;
        initialize_address(static_cast<Address*>(current->next) + i);
    }
}
void delete_address(Address* current) {
    if (!current) return;
    for (int i = 0; current->level < dim - 1 && i < size[current->level]; i++) {
        delete_address(static_cast<Address*>(current->next) + i);
    }

    delete[] current->next;
}
Int operator[](const int index);
~Array() {
    delete_address(top);
    delete[] size;
}

Iterator begin() {
    int* arr = new int[dim];
    for (int i = 0; i != dim; i++) arr[i] = 0;

    Iterator temp(this, arr);
    delete[] arr;

    return temp;
}
Iterator end() {
    int* arr = new int[dim];
    arr[0] = size[0];
    for (int i = 1; i < dim; i++) arr[i] = 0;

    Iterator temp(this, arr);
    delete[] arr;

    return temp;
}
};
```

```
class Int {
    void* data;

    int level;
    Array* array;

public:
    Int(int index, int _level = 0, void* _data = NULL, Array* _array = NULL)
        : level(_level), data(_data), array(_array) {
        if (_level < 1 || index >= array->size[_level - 1]) {
            data = NULL;
            return;
        }
        if (level == array->dim) {
            // 이제 data에 우리의 int 자료형을 저장하도록 해야 한다.
            data = static_cast<void*>(
                static_cast<int*>(static_cast<Array::Address*>(data)->next) + index);
        } else {
            // 그렇지 않을 경우 data에 그냥 다음 addr을 넣어준다.
            data = static_cast<void*>(static_cast<Array::Address*>(
                static_cast<Array::Address*>(data)->next) +
                index);
        }
    };

    Int(const Int& i) : data(i.data), level(i.level), array(i.array) {}

    operator int() {
        if (data) return *static_cast<int*>(data);
        return 0;
    }
    Int& operator=(const int& a) {
        if (data) *static_cast<int*>(data) = a;
        return *this;
    }

    Int operator[](const int index) {
        if (!data) return 0;
        return Int(index, level + 1, data, array);
    }
};

Int Array::operator[](const int index) {
    return Int(index, 1, static_cast<void*>(top), this);
}

Int Array::Iterator::operator*() {
    Int start = arr->operator[](location[0]);
    for (int i = 1; i <= arr->dim - 1; i++) {
        start = start.operator[](location[i]);
    }
    return start;
}
```

```
} // namespace MyArray
int main() {
    int size[] = {2, 3, 4};
    MyArray::Array arr(3, size);

    MyArray::Array::Iterator itr = arr.begin();
    for (int i = 0; itr != arr.end(); itr++, i++) (*itr) = i;
    for (itr = arr.begin(); itr != arr.end(); itr++)
        std::cout << *itr << std::endl;

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                arr[i][j][k] = (i + 1) * (j + 1) * (k + 1) + arr[i][j][k];
            }
        }
    }

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                std::cout << i << " " << j << " " << k << " " << arr[i][j][k]
                    << std::endl;
            }
        }
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
0
1
2
3
4
5
6
7
8
9
10
11
12
```

```
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
0 0 0 1  
0 0 1 3  
0 0 2 5  
0 0 3 7  
0 1 0 6  
0 1 1 9  
0 1 2 12  
0 1 3 15  
0 2 0 11  
0 2 1 15  
0 2 2 19  
0 2 3 23  
1 0 0 14  
1 0 1 17  
1 0 2 20  
1 0 3 23  
1 1 0 20  
1 1 1 25  
1 1 2 30  
1 1 3 35  
1 2 0 26  
1 2 1 33  
1 2 2 40  
1 2 3 47
```

와 같이 잘 실행됨을 알 수 있습니다. 한 가지 눈여겨 볼 점은

```
MyArray::Array::Iterator itr = arr.begin();
```

```
for (int i = 0; itr != arr.end(); itr++, i++) (*itr) = i;
```

와 같이 수행하는 부분입니다. 이와 같이 반복자를 이용하는 것은 C++에서 매우 많이 사용되고 있는 방법으로, 나중에 표준 라이브러리들에 대해 살펴볼 때 다시 한번 등장하게 됩니다.

자 여러분 수고하셨습니다! 아마 이번 강좌는 제가 여태까지 진행한 강좌 중에 가장 어려웠고 저도 설명하기에 가장 어려운 내용이 아니였나 싶네요. 다음 강좌에서는 C++의 또 다른 쇼킹할 만한 기능들에 대해 살펴보도록 하겠습니다. 감사합니다.

생각해보기

문제 1

앞서 N 차원 배열을 구현하는 또 다른 방법 (그냥 $x_1 * \dots * x_n$ 개의 1 차원 배열을 만든 뒤에, [] 연산자를 위와 같이 특별한 방법을 이용하여 접근할 수 있게 하는 것) 으로 N 차원 배열을 구현해봅시다. (난이도 : 上)

클래스의 상속

안녕하세요 여러분!! 제가 그간 많이 바빠서 강좌를 진행하지 못하고 있었는데요, 요새 여유가 조금 생겨서 다시 C++ 강좌를 진행할 수 있게 되었습니다. 오랫동안 기다리셨던 분들에게는 정말로 죄송하다고 전하고 싶네요. 이전 강좌까지 클래스와 연산자 오버로딩에 대해 배우면서 C++ 의 새로운 맛을 보았다면, 이제부터는 본격적으로 C++ 의 진한 국물을 우려내는 듯한 강좌가 될 것 같습니다.

표준 string 클래스

아마 제 강좌를 훌륭하게 따라 오신 분들이라면 지난 강좌에서 `MyString` 클래스를 만드셨던 것이 기억이 나실 것입니다. C 언어 스타일의 문자열은 여러가지 문제점들이 많기 때문에 (예를 들어 문자열의 길이를 한 번에 알 수 없고 마지막 NULL 문자까지 하나 하나 읽어야 된다는 듯지..) 문자열을 처리할 수 있는 새로운 무언가가 계속 필요해야 했습니다.

사실 우리가 예전에 만든 `MyString` 클래스도 문자열 처리를 꽤나 훌륭하게 하지만, 실제로 속도가 매우 중요한 환경에서 그대로 쓰기에는 부족한 점이 많습니다. 하지만 많은 프로그래머들의 노력 끝에 `string`이라는 빠르고, 안전하고 사용하기 매우 간단한 문자열 클래스가 표준으로 채택이 됩니다.

(이 `string`에 들어가 있는 몇 가지 기술을 소개해보자면, 짧은 문자열에 대해서는 동적으로 메모리를 할당하지 않고 그냥 지역 변수로 보관을 하고, 문자열을 복사를 할 때 그 복사된 문자열의 내용이 바뀌지 않는 한 실제로 데이터를 복사하는 것이 아니라 원래 문자열을 가리키기만 한다 등등 속도를 향상시키기 위한 여러 노력이 접목되어 있습니다)¹⁾

```
#include <iostream>
#include <string>

int main() {
```

¹⁾ 물론 이는 `std::string` 구현 방식에 따라 다릅니다. 여러분이 어떤 라이브러리를 사용함에 따라 아닐 수도 있습니다.

```
// 표준이므로 std 안에 string 이 정의되어 있습니다.
std::string s = "abc";

std::cout << s << std::endl;

return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

abc

와 같이 abc 가 잘 출력됨을 알 수 있습니다.

일단 기본적으로 "abc" 는 컴파일러 상에서는 C 형식 문자열로 인식됩니다. 즉, 위 문장은 string 클래스의 인자를 const char * 로 받는 생성자를 호출한 것으로 볼 수 있겠지요.

```
#include <iostream>
#include <string>

int main() {
    std::string s = "abc";
    std::string t = "def";
    std::string s2 = s;

    std::cout << s << " 의 길이 : " << s.length() << std::endl;
    std::cout << s << " 뒤에 " << t << " 를 붙이면 : " << s + t << std::endl;

    if (s == s2) {
        std::cout << s << " 와 " << s2 << " 는 같다 " << std::endl;
    }
    if (s != t) {
        std::cout << s << " 와 " << t << " 는 다르다 " << std::endl;
    }
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

abc 의 길이 : 3
abc 뒤에 def 를 붙이면 : abcdef

```
abc 와 abc 는 같다  
abc 와 def 는 다르다
```

위에는 몇 가지 `string` 클래스의 기능들을 간단히 보여드린 것입니다. 예를 들어 문자열의 길이를 출력하는 `length` 함수라던지, 연산자 오버로딩을 사용해서 + 연산자가 실제로 문자열을 결합시키는 함수로 사용되고 있습니다.

특히 편리한 점으로 C 형식 문자열이였을 경우 문자열을 비교하기 위해서 `strcmp` 함수를 사용했어야 하고,

```
if (s == s2) std::cout << s << " 와 " << s2 << " 는 같다 " << std::endl;  
if (s != t) std::cout << s << " 와 " << t << " 는 다르다 " << std::endl;
```

와 같이 == 나 != 로 비교하는 것이 불가능 하였습니다. (왜냐하면 이는 문자열의 주소값을 비교하는 것이였으니까요! - 혹시 기억이 잘 나지 않는 분들은 [이 강좌를 다시 보고 오세요!](#)) 하지만 이 `string` 클래스는 == 와 != 연산자들을 모두 오버로딩해서 제대로 비교를 수행합니다. 뿐만 아니라 크기 비교 >=, <= 등도 제대로 수행이 되지요.

사실 `string`에서 제공하는 함수와 기능들인 제가 소개한 것 말고도 엄청나게 많아서 한 강좌에 다 채워놓지 못할 정도입니다. 위에서는 가장 많이 쓰는 기능만 소개해놓았고 모든 정보를 원하신다면 [여기](#)를 참조하시면 됩니다.

표준 문자열 `std::string`에는 `length` 함수 말고도, 문자열 사이에 문자열을 삽입하는 `insert` 함수나, 특정 위치를 지우는 `erase`나 문자열을 치환하는 `replace` 등등 수 많은 유용한 함수들이 많습니다.

사원 관리 프로그램

자, 이제 여러분은 한 회사의 사원 관리 프로그램을 만들어달라는 의뢰를 받게 됩니다.!



우리가 만들어야 할 프로그램의 목적은 회사의 사원들의 월급을 계산해서 한달에 총 얼마나 되는 돈을 월급으로 지출해야 하는지 알려주는 단순한 프로그램입니다. 그렇다면 일단 여러분은 각 사원들에 정보를 클래스로 만들어서 데이터를 보관하도록 하겠지요.

사원들의 필요한 데이터는 이름, 나이, 직책과 직책의 순위에 해당하는 숫자값 (예를 들어 평사원이면 1, 대리면 2 이런 식으로) 정도입니다. 이를 바탕으로 간단히 클래스를 구성해본다면 다음과 같이 짤 수 있을 것입니다.

```
class Employee {
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;           // 순위 (값이 클수록 높은 순위)

public:
    Employee(std::string name, int age, std::string position, int rank)
        : name(name), age(age), position(position), rank(rank) {}

    // 복사 생성자
    Employee(const Employee& employee) {
        name = employee.name;
        age = employee.age;
        position = employee.position;
        rank = employee.rank;
    }

    // 디폴트 생성자
    Employee() {}

    void print_info() {
        std::cout << name << " (" << position << " , " << age << ") ==> "
            << calculate_pay() << "만원" << std::endl;
    }
    int calculate_pay() { return 200 + rank * 50; }
};
```

일단 저는 3 개의 생성자들을 정의해놓았는데요, 여태까지 강의를 잘 따라오신 분들은 각 생성자가 어떤식으로 동작하는지, 단번에 눈치챌 수 있으셨겠죠?

여기서 눈여겨볼 점은 `calculate_pay` 함수인데, 기본급 200 에 직위에 따라 50 을 곱해서 더 받도록 하였습니다. (아 이건 물론 제가 임의로 정한것이고 일반적인 회사에서 이렇게 한다고 주장하는 것이 아닙니다 ㅎ)

자 이제 각각의 `Employee` 클래스를 만들었으니, 이 `Employee` 객체들을 관리할 수 있는 무언가가 있어야겠지요? 물론 단순히 배열을 사용해서 사원들을 관리할 수 있겠지만, 그렇게 된다면 굉장히 불편하겠지요? 그래서 저는 `EmployeeList` 클래스를 만들어서 간단하게 처리하도록 할 것입니다.

일단 우리는 다음과 같은 멤버 변수들을 이용해서 사원 데이터를 처리할 것입니다.

```
int alloc_employee;           // 할당한 총 직원 수
int current_employee;         // 현재 직원 수
Employee **employee_list;    // 직원 데이터
```

MyString을 만들었던 기억을 되살려보자면, 언제나 동적으로 데이터를 할당하는 것을 처리하기 위해서는 두 개의 변수가 필요 했는데, 하나는 현재 할당된 총 크기고, 다른 하나는 그 중에서 실제로 사용하고 있는 양이지요. 이렇게 해야지만 할당된 크기 보다 더 많은 양을 실수로 사용하는 것을 막을 수 있습니다. 따라서 우리도 alloc_employee 가 할당된 크기를 알려주는 배열이고, current_employee 는 현재 employee_list 에 등록된 사원 수라고 볼 수 있지요.

employee_list 가 Employee** 타입으로 되어 있는 이유는, 우리가 이를 Employee* 객체를 담는 배열로 사용할 것이기 때문입니다. 그렇다면 EmployeeList 클래스의 생성자는 아래와 같이 쉽게 구성할 수 있겠지요.

```
EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
    employee_list = new Employee*[alloc_employee];
    current_employee = 0;
}
```

그리고 사원을 추가하는 함수는 아래처럼 단순하게 구성할 수 있습니다.

```
void add_employee(Employee* employee) {
    // 사실 current_employee 보다 alloc_employee 가 더
    // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
    // 최대한 단순하게 생각해서 alloc_employee 는
    // 언제나 current_employee 보다 크다고 생각한다.
    // (즉 할당된 크기는 현재 총 직원수 보다 많음)
    employee_list[current_employee] = employee;
    current_employee++;
}
```

물론 위 주석에도 잘 설명되어 있듯이, alloc_employee 보다 current_employee 가 더 많아진다면 새로 재할당을 하고 데이터를 모두 복사해야겠지만 (여러분의 코드는 그렇게 쓰세요!) 제가 주목하고 싶은 부분은 여기가 아니라서 그냥 위 처럼 간단하게 생각합시다.

그리고 나머지 짜잘한 함수들을 완성해준다면 다음과 같이 EmployeeList 클래스를 구성할 수 있게 됩니다.

```
class EmployeeList {
    int alloc_employee;           // 할당한 총 직원 수
    int current_employee;         // 현재 직원 수
```

```
Employee** employee_list; // 직원 데이터

public:
EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
    employee_list = new Employee*[alloc_employee];
    current_employee = 0;
}
void add_employee(Employee* employee) {
    // 사실 current_employee 보다 alloc_employee 가 더
    // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
    // 최대한 단순하게 생각해서 alloc_employee 는
    // 언제나 current_employee 보다 크다고 생각한다.
    // (즉 할당된 크기는 현재 총 직원수 보다 많음)
    employee_list[current_employee] = employee;
    current_employee++;
}
int current_employee_num() { return current_employee; }

void print_employee_info() {
    int total_pay = 0;
    for (int i = 0; i < current_employee; i++) {
        employee_list[i]->print_info();
        total_pay += employee_list[i]->calculate_pay();
    }

    std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
}
~EmployeeList() {
    for (int i = 0; i < current_employee; i++) {
        delete employee_list[i];
    }
    delete[] employee_list;
}
};
```

그렇다면 실제 프로그램을 구성해볼까요. 현재 무한 상사에는 다음과 같은 사원들이 있습니다.



따라서 전체 코드는 다음과 같습니다.

```
#include <iostream>
#include <string>

class Employee {
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;             // 순위 (값이 클 수록 높은 순위)

public:
    Employee(std::string name, int age, std::string position, int rank)
        : name(name), age(age), position(position), rank(rank) {}

    // 복사 생성자
    Employee(const Employee& employee) {
        name = employee.name;
        age = employee.age;
        position = employee.position;
        rank = employee.rank;
    }

    // 디폴트 생성자
    Employee() {}

    void print_info() {
        std::cout << name << " (" << position << " , " << age << ") ==> "
              << calculate_pay() << "만원" << std::endl;
    }
    int calculate_pay() { return 200 + rank * 50; }
};
```

```
class EmployeeList {
    int alloc_employee;           // 할당한 총 직원 수
    int current_employee;         // 현재 직원 수
    Employee** employee_list;    // 직원 데이터

public:
    EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
        employee_list = new Employee*[alloc_employee];
        current_employee = 0;
    }
    void add_employee(Employee* employee) {
        // 사실 current_employee 보다 alloc_employee 가 더
        // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
        // 최대한 단순하게 생각해서 alloc_employee 는
        // 언제나 current_employee 보다 크다고 생각한다.
        // (즉 할당된 크기는 현재 총 직원수 보다 많음)
        employee_list[current_employee] = employee;
        current_employee++;
    }
    int current_employee_num() { return current_employee; }

    void print_employee_info() {
        int total_pay = 0;
        for (int i = 0; i < current_employee; i++) {
            employee_list[i]->print_info();
            total_pay += employee_list[i]->calculate_pay();
        }

        std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
    }
    ~EmployeeList() {
        for (int i = 0; i < current_employee; i++) {
            delete employee_list[i];
        }
        delete[] employee_list;
    }
};

int main() {
    EmployeeList emp_list(10);
    emp_list.add_employee(new Employee("노홍철", 34, "평사원", 1));
    emp_list.add_employee(new Employee("하하", 34, "평사원", 1));

    emp_list.add_employee(new Employee("유재석", 41, "부장", 7));
    emp_list.add_employee(new Employee("정준하", 43, "과장", 4));
    emp_list.add_employee(new Employee("박명수", 43, "차장", 5));
    emp_list.add_employee(new Employee("정형돈", 36, "대리", 2));
    emp_list.add_employee(new Employee("길", 36, "인턴", -2));
    emp_list.print_employee_info();
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
노홍철 (평사원 , 34) ==> 250만원
하하 (평사원 , 34) ==> 250만원
유재석 (부장 , 41) ==> 550만원
정준하 (과장 , 43) ==> 400만원
박명수 (차장 , 43) ==> 450만원
정형돈 (대리 , 36) ==> 300만원
길 (인턴 , 36) ==> 100만원
총 비용 : 2300만원
```

와 같이 잘 실행된다는 것을 볼 수 있습니다.

아 이렇게 사원 관리 프로그램을 잘 만들어서 제출해달라는 찰나, 무한 상사로 부터 연락을 한 통 받습니다. 차장 이상 급들은 관리데이터에 근속 년수를 포함시켜서 월급에 추가해달라고 말이지요. 그래서 저는 올며가며 겨자먹기로 Manager 클래스를 추가하였습니다. 사실 Employee 클래스랑 거의 똑같지만, 어쩔 수 없지요. 더 짜증나는 부분은 EmployeeList 클래스에서도 Employee 와 Manager 를 따로 처리해야 된다는 점입니다. 아무튼, 일단 Manager 클래스를 구성해봅시다.

```
class Manager {
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;           // 순위 (값이 클 수록 높은 순위)
    int year_of_service;

public:
    Manager(std::string name, int age, std::string position, int rank,
            int year_of_service)
        : year_of_service(year_of_service),
          name(name),
          age(age),
          position(position),
          rank(rank) {}

    // 복사 생성자
    Manager(const Manager& manager) {
        name = manager.name;
        age = manager.age;
        position = manager.position;
        rank = manager.rank;
        year_of_service = manager.year_of_service;
    }
}
```

```
// 디폴트 생성자
Manager() {}

int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
void print_info() {
    std::cout << name << " (" << position << " , " << age << ", "
        << year_of_service << "년차) ==> " << calculate_pay() << "만원"
        << std::endl;
}
};
```

기존의 Employee 클래스와 다 똑같고, int year_of_service 하나만 추가된 것을 볼 수 있습니다. 물론 월급을 계산하는 calculate_pay 함수나, 정보를 출력하는 print_info 함수가 약간 바뀌게 되었습니다. 이번에는 EmployeeList 클래스를 살펴보도록 합시다.

당연하게도 Employee 배열과 Manager 배열을 따로 만들어야만 합니다. 따라서 각 배열에 사용하고 있는 크기를 나타낼 변수도 따로 지정해야만 하겠지요. 따라서

```
int alloc_employee; // 할당한 총 직원 수

int current_employee; // 현재 직원 수
int current_manager; // 현재 매니저 수

Employee **employee_list; // 직원 데이터
Manager **manager_list; // 매니저 데이터
```

와 같이 바꿔주어야 합니다. (여기서 또한 간단하게 처리하기 위해서 각 배열에 할당한 크기는 모두 동일하다고 생각합니다). 그리고 무엇보다도 EmployeeList 클래스의 나머지 부분을 바꿔주면

```
class EmployeeList {
    int alloc_employee; // 할당한 총 직원 수

    int current_employee; // 현재 직원 수
    int current_manager; // 현재 매니저 수

    Employee** employee_list; // 직원 데이터
    Manager** manager_list; // 매니저 데이터

    public:
        EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
            employee_list = new Employee*[alloc_employee];
            manager_list = new Manager*[alloc_employee];

            current_employee = 0;
            current_manager = 0;
        }
}
```

```

void add_employee(Employee* employee) {
    // 사실 current_employee 보다 alloc_employee 가 더
    // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
    // 최대한 단순하게 생각해서 alloc_employee 는
    // 언제나 current_employee 보다 크다고 생각한다.
    // (즉 할당된 크기는 현재 총 직원수 보다 많음)
    employee_list[current_employee] = employee;
    current_employee++;
}
void add_manager(Manager* manager) {
    manager_list[current_manager] = manager;
    current_manager++;
}
int current_employee_num() { return current_employee + current_manager; }

void print_employee_info() {
    int total_pay = 0;
    for (int i = 0; i < current_employee; i++) {
        employee_list[i]->print_info();
        total_pay += employee_list[i]->calculate_pay();
    }
    for (int i = 0; i < current_manager; i++) {
        manager_list[i]->print_info();
        total_pay += manager_list[i]->calculate_pay();
    }
    std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
}
~EmployeeList() {
    for (int i = 0; i < current_employee; i++) {
        delete employee_list[i];
    }
    for (int i = 0; i < current_manager; i++) {
        delete manager_list[i];
    }
    delete[] employee_list;
    delete[] manager_list;
}
};

```

와 같이 구성할 수 있습니다. 두 개의 배열을 관리하기 때문에 똑같은 코드를 변수 이름만 바꿔서 한 번 더 써야 합니다. 상당히 귀찮기 다름 없지요. 이를 바탕으로 전체 코드를 구성해보면 다음과 같습니다.

```

#include <iostream>
#include <string>

class Employee {
    std::string name;
    int age;

```

```
std::string position; // 직책 (이름)
int rank;           // 순위 (값이 클 수록 높은 순위)

public:
Employee(std::string name, int age, std::string position, int rank)
: name(name), age(age), position(position), rank(rank) {}

// 복사 생성자
Employee(const Employee& employee) {
    name = employee.name;
    age = employee.age;
    position = employee.position;
    rank = employee.rank;
}

// 디폴트 생성자
Employee() {}

void print_info() {
    std::cout << name << " (" << position << " , " << age << ") ==> "
        << calculate_pay() << "만원" << std::endl;
}
int calculate_pay() { return 200 + rank * 50; }
};

class Manager {
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;           // 순위 (값이 클 수록 높은 순위)
    int year_of_service;

public:
Manager(std::string name, int age, std::string position, int rank, int
→ year_of_service)
: year_of_service(year_of_service),
    name(name),
    age(age),
    position(position),
    rank(rank) {}

// 복사 생성자
Manager(const Manager& manager) {
    name = manager.name;
    age = manager.age;
    position = manager.position;
    rank = manager.rank;
    year_of_service = manager.year_of_service;
}
```

```
// 디폴트 생성자
Manager() {}

int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
void print_info() {
    std::cout << name << " (" << position << " , " << age << ", " <<
    ↵ year_of_service
    << "년차) ==> " << calculate_pay() << "만원" << std::endl;
}
};

class EmployeeList {
    int alloc_employee; // 할당한 총 직원 수

    int current_employee; // 현재 직원 수
    int current_manager; // 현재 매니저 수

    Employee** employee_list; // 직원 데이터
    Manager** manager_list; // 매니저 데이터

public:
    EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
        employee_list = new Employee*[alloc_employee];
        manager_list = new Manager*[alloc_employee];

        current_employee = 0;
        current_manager = 0;
    }

    void add_employee(Employee* employee) {
        // 사실 current_employee 보다 alloc_employee 가 더
        // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
        // 최대한 단순하게 생각해서 alloc_employee 는
        // 언제나 current_employee 보다 크다고 생각한다.
        // (즉 할당된 크기는 현재 총 직원수 보다 많음)
        employee_list[current_employee] = employee;
        current_employee++;
    }

    void add_manager(Manager* manager) {
        manager_list[current_manager] = manager;
        current_manager++;
    }

    int current_employee_num() { return current_employee + current_manager; }

    void print_employee_info() {
        int total_pay = 0;
        for (int i = 0; i < current_employee; i++) {
            employee_list[i]->print_info();
            total_pay += employee_list[i]->calculate_pay();
        }

        for (int i = 0; i < current_manager; i++) {
            manager_list[i]->print_info();
        }
    }
}
```

```

        total_pay += manager_list[i]->calculate_pay();
    }
    std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
}
~EmployeeList() {
    for (int i = 0; i < current_employee; i++) {
        delete employee_list[i];
    }
    for (int i = 0; i < current_manager; i++) {
        delete manager_list[i];
    }
    delete[] employee_list;
    delete[] manager_list;
}
};

int main() {
    EmployeeList emp_list(10);
    emp_list.add_employee(new Employee("노홍철", 34, "평사원", 1));
    emp_list.add_employee(new Employee("하하", 34, "평사원", 1));

    emp_list.add_manager(new Manager("유재석", 41, "부장", 7, 12));
    emp_list.add_manager(new Manager("정준하", 43, "과장", 4, 15));
    emp_list.add_manager(new Manager("박명수", 43, "차장", 5, 13));
    emp_list.add_employee(new Employee("정형돈", 36, "대리", 2));
    emp_list.add_employee(new Employee("길", 36, "인턴", -2));
    emp_list.print_employee_info();
    return 0;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

노홍철 (평사원 , 34) ==> 250만원
하하 (평사원 , 34) ==> 250만원
정형돈 (대리 , 36) ==> 300만원
길 (인턴 , 36) ==> 100만원
유재석 (부장 , 41, 12년차) ==> 610만원
정준하 (과장 , 43, 15년차) ==> 475만원
박명수 (차장 , 43, 13년차) ==> 515만원
총 비용 : 2500만원

```

와 같이 잘 실행됩니다.

상속 (Inheritance)

여러분도 느꼈겠지만, Manager 클래스를 추가하면서 복사 붙여넣기 신공을 참 여러번 반복한다고 느꼈을 것입니다. 이게 어쩔 수 없는 이유가 Manager 의 코드 자체가 Employee 의 대부분을 포함하고 있기 때문이지요. C++ 에서는 이와 같은 경우, 다른 클래스의 내용을 그대로 포함할 수 있는 작업을 가능토록 해줍니다. 바로 상속 이라는 것을 통해 말이지요.

사실 상속이라는 단어 속에 무언가를 물려 받아서 사용한다는 의미가 있습니다. 즉, C++ 에서 상속을 통해 다른 클래스의 정보를 물려 받아서 사용할 수 있습니다.

일단은 바로 Employee 와 Manager 클래스에 적용하기 전에 간단한 클래스를 먼저 만들어서 어떻게 C++ 에서 상속이라는 기능이 사용되는지 알아보도록 하겠습니다.

```
class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};
```

위는 우리의 설명을 도와줄 기반 클래스입니다. 그리고, 아래는 기반 클래스(Base)를 물려받은 파생(Derived) 클래스의 모습입니다.

주의 사항

보통 부모 - 자식 클래스라고도 이야기 많이 합니다. 다만 자식 이란 단어 속에서 한 개의 부모만 가지는 의미가 담겨 있는데 (엄마가 2 명일 수는 없자나요), C++ 의 경우 여러 명의 부모를 가질 수 있기에, 부모, 자식 클래스라 하기 보단, 기반, 파생 클래스라 부르는 것이 낫다고 생각합니다.

```
class Derived : public Base {
    std::string s;

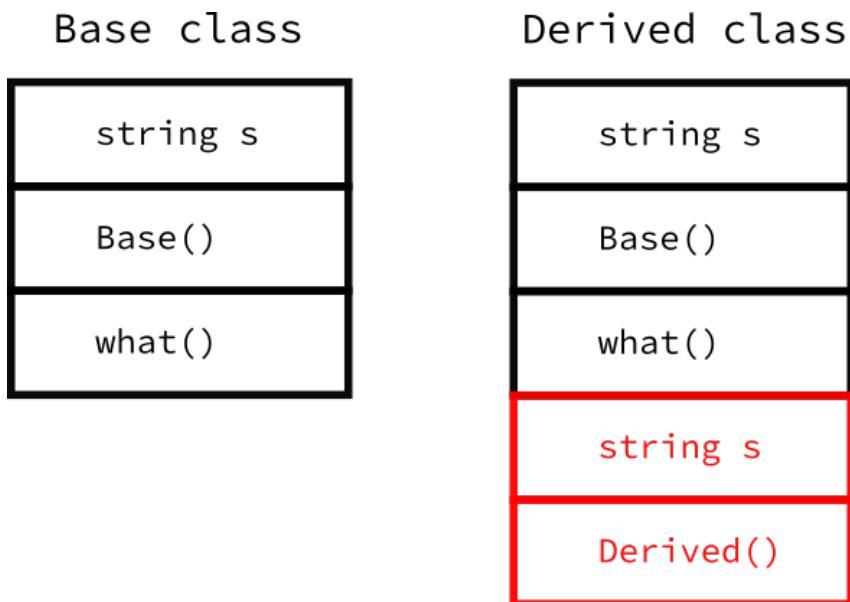
public:
    Derived() : Base(), s("파생") {
        std::cout << "파생 클래스" << std::endl;

        // Base 에서 what() 을 물려 받았으므로
        // Derived 에서 당연히 호출 가능하다
        what();
    }
};
```

가장 먼저 눈에 띠는 부분은 바로 맨 위 `class` 의 정의 부분으로

```
class Derived : public Base
```

와 같이 되어 있습니다. 이는 `Derived` 가 `Base` 를 `public` 형식으로 상속을 받겠다는 의미가 됩니다. `public` 형식으로 상속받는게 무엇인지에 대해서는 좀 있다가 이야기를 하도록 하고, 아무튼 위 처럼 상속을 받은 후에 `Derived` 는 다음과 같은 모습이 됩니다.



마치 `Derived` 클래스 안에 `Base` 클래스의 코드가 그대로 들어가 있는 것 처럼 말이지요. 따라서 아래 처럼 `Derived` 클래스에서 `Base` 클래스의 `what` 함수를 호출 할 수 있게 됩니다.

```
Derived() : Base(), s("파생") {
    std::cout << "파생 클래스" << std::endl;

    // Base 에서 what() 을 물려 받았으므로
    // Derived 에서 당연히 호출 가능하다
    what();
}
```

그리고 또 하나 눈여겨 봐야 할 점은 `Derived` 의 생성자 호출 부분입니다. `Derived` 의 생성자는 위 처럼 초기화자 리스트에서 기반의 생성자를 호출해서 기반의 생성을 먼저 처리 한 다음에, `Derived` 의 생성자가 실행되어야 합니다. 따라서 아래 처럼

```
Derived() : Base(), s("파생")
```

초기화 리스트에서 `Base` 를 통해 기반의 생성자를 먼저 호출하게 됩니다. 참고로 기반 클래스의 생성자를 명시적으로 호출하지 않을 경우 기반 클래스의 디폴트 생성자가 호출됩니다.

그렇다면 아래의 코드를 살펴보도록 합시다.

```
#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
    Derived() : Base(), s("파생") {
        std::cout << "파생 클래스" << std::endl;

        // Base 에서 what() 을 물려 받았으므로
        // Derived 에서 당연히 호출 가능하다
        what();
    }
};

int main() {
    std::cout << " === 기반 클래스 생성 ===" << std::endl;
    Base p;

    std::cout << " === 파생 클래스 생성 ===" << std::endl;
    Derived c;

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
==== 기반 클래스 생성 ====
기반 클래스
==== 파생 클래스 생성 ====
기반 클래스
파생 클래스
기반
```

와 같이 나옴을 알 수 있습니다.

일단 여러분은 기반 클래스 생성에서 왜 저런식으로 출력되는지는 당연히 알고 계실 것입니다. `Base` 의 생성자에서 '기반 클래스' 를 출력을 하게 되지요. 그렇다면 이번에는 `Derived` 객체를 만들 때 왜 저런식으로 출력되는지 살펴보도록 합시다.

```
std::cout << " === 파생 클래스 생성 ===" << std::endl;
Derived c;
```

일단 위와 같이 `Derived` 의 인자가 없는 생성자를 호출하게 됩니다.

```
Derived() : Base(), s("파생") {
    std::cout << "파생 클래스" << std::endl;

    // Base 에서 what() 을 물려 받았으므로
    // Derived 에서 당연히 호출 가능하다
    what();
}
```

이제 위에서 `Derived` 의 `s`에 파생을 넣게 되고, `Derived` 생성자의 내부를 실행하기 전에 `Base` 의 생성자를 먼저 호출하게 됩니다. 따라서, 파생 클래스 생성 바로 아래에 파생 클래스가 출력하기 이전에 `Base` 의 생성자가 호출되어서 기반 클래스가 먼저 출력하게 되는 것이지요.

그렇다면 이제 `what()` 함수를 호출하는 부분을 살펴봅시다. `Derived` 에서 정의되어 있지 않는 `what` 을 어떻게 호출할 수 있느냐면, 당연하게도, `Base` 의 모든 정보를 상속 받았기 때문에 `Derived`에서도 `what` 을 호출 할 수 있게 되는 것입니다.

그런데, `what` 함수를 호출했을 때, 파생이 아니라 기반라고 출력이 되었는데, `what` 함수를 보면 `s`의 값을 출력하도록 되어 있습니다. 이러한 일이 발생한 이유는, `what` 함수는 `Base`에 정의가 되어 있기 때문에 `Derived`의 `s`가 아니라 `Base`의 `s`가 출력되어 "기반"라고 나오게 되는 것입니다.

그렇다면 만일 `Derived`에도 `what` 함수를 정의해주면 어떨까요.

```
#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
```

```

Derived() : Base(), s("파생") {
    std::cout << "파생 클래스" << std::endl;

    // Base에서 what()을 물려 받았으므로
    // Derived에서 당연히 호출 가능하다
    what();
}

void what() { std::cout << s << std::endl; }
};

int main() {
    std::cout << " === 기반 클래스 생성 ===" << std::endl;
    Base p;

    std::cout << " === 파생 클래스 생성 ===" << std::endl;
    Derived c;

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

기반 클래스
==== 파생 클래스 생성 ====
기반 클래스
파생 클래스
파생

```

이번에는 `Derived` 와 `Base` 에 둘다 `what()` 함수가 정의되어 있습니다. 이 경우, `Derived`에서 아래처럼 `what` 을 호출하게 되면 무엇이 호출 될까요?

어떤 분들은 컴파일 상에서 문제가 발생하지 않을까 라고 생각할 수도 있는데, 사실 두 함수는 같은 이름이지만 (심지어 인자들도 같지만), 다른 클래스에 정의되어 있는 것이기 때문에 다른 함수로 취급됩니다. (물론, `Derived` 안에 `what`에 두 개 정의되어 있다면 문제가 되었겠지요)

위 경우에는 `Derived`에 `what` 함수가 정의되어 있기 때문에 `Derived`의 생성자에서 `what`을 호출 할 때 (굳이) 멀리 `Base`의 함수들 까지 뛰지지 않고, 바로 앞에 있는 `Derived`의 `what` 함수를 호출하게 됩니다.

이런 것을 가리켜 오버라이딩(overriding)이라고 합니다. 즉, `Derived`의 `what` 함수가 `Base`의 `what` 함수를 오버라이딩 한 것이지요.²⁾

2) 간혹 함수의 오버로딩(overloading)과 혼동하시는 분들이 많은데, 오버로딩은 같은 이름의 함수를 인자를 달리 하여 정의하는 것을 의미하는 것입니다. 상속에서의 오버라이딩과는 전혀 다른 이야기입니다

새로운 친구 protected

다음과 같은 코드를 생각해봅시다.

```
class Base {
    std::string parent_string;

public:
    Base() : parent_string("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << parent_string << std::endl; }
};

class Derived : public Base {
    std::string child_string;

public:
    Derived() : child_string("파생"), Base() {
        std::cout << "파생 클래스" << std::endl;

        // 그렇다면 현재 private 인 Base 의
        // parent_string 에 접근할 수 있을까?
        parent_string = "바꾸기";
    }

    void what() { std::cout << child_string << std::endl; }
};
```

만일 컴파일 하였다면 아래와 같은 컴파일 애러를 볼 수 있습니다.

컴파일 오류

```
error C2248: 'Base::parent_string' : cannot access private member
→ declared in class 'Base'
```

아니 이게 도대체 무슨 말인가요! 기껏 상속 받았더니, 접근할 수 없다니요. 하지만 사실 **private** 멤버 변수들은 그 어떠한 경우에서도 자기 클래스 말고는 접근할 수 없습니다.

그렇지만 종종 파생 클래스(상속 받는 클래스 - 위 경우 **Derived** 클래스)에서 원래 기반의 클래스 (즉 여기서 **Base**) 의 데이터에 직접 접근할 필요성이 있습니다.

예를 들어서 우리의 예시의 경우 **Employee** 클래스를 기반 클래스로 해서 **Manager** 클래스가 상속 받았을 때, **name**이나 **age**에 접근할 필요성이 있겠지요. 하지만 이들은 **private**으로 되어 있기 때문에 접근이 불가합니다.

다행이도 C++에서는 **protected**라는 **public**과 **private**에 중간 위치에 있는 접근 지시자를 지원합니다. 이 키워드는, 상속받는 클래스에서는 접근 가능하고 그 외의 기타 정보는 접근 불가능이라고 보시면 됩니다. 부모(기반 클래스)와 자식(파생 클래스)으로 쉽게 비유하자면

- **private** : (부모님들한테 안가르쳐 주는) 자신만의 비밀번호
- **protected** : 집 현관문 비밀번호 (가족들은 알지만 그 외의 사람들은 접근불가)
- **public** : 집 주소 (가족 뿐만이 아니라 다른 사람들도 알 수 있다)

이렇게 3 단계로 멤버의 접근 허용 범위를 지정할 수 있습니다. 그렇다면 실제로 **private**을 **protected**로 바꾼다면 잘 실행됨을 알 수 있습니다.

```
class Base {
protected:
    std::string parent_string;

public:
    Base() : parent_string("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << parent_string << std::endl; }
};

class Derived : public Base {
    std::string child_string;

public:
    Derived() : Base(), child_string("파생") {
        std::cout << "파생 클래스" << std::endl;

        // 그렇다면 현재 private 인 Base 의
        // parent_string 에 접근할 수 있을까?
        parent_string = "바꾸기";
    }

    void what() { std::cout << child_string << std::endl; }
};
```

위 코드는 아주 잘 컴파일 됩니다.

그렇다면 이제

```
class Derived : public Base
```

에서 이 **public** 키워드의 의미를 밝힐 때가 됐군요. 사실 저 키워드가 **public**이냐 **protected**이냐 **private**이냐에 따라 상속 받는 클래스에서 기반 클래스의 멤버들이 실제로 어떻게 작동하는지 영향을 줍니다. 이게 무슨 말이냐면

- 만일 위처럼 `public` 형태로 상속 하였다면 기반 클래스의 접근 지시자들에 영향 없이 그대로 작동합니다. 즉 파생 클래스 입장에서 `public` 은 그대로 `public` 이고, `protected` 는 그대로 `protected` 이고, `private` 은 그대로 `private` 입니다.
- 만일 `protected` 로 상속하였다면 파생 클래스 입장에서 `public` 은 `protected` 로 바뀌고 나머지는 그대로 유지됩니다.
- 만일 `private` 으로 상속하였다면 파생 클래스 입장에서 모든 접근 지시자들이 `private` 가 됩니다.

실제로 아래와 같은 예제를 살펴봅시다.

```
#include <iostream>
#include <string>

class Base {
public:
    std::string parent_string;

    Base() : parent_string("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << parent_string << std::endl; }
};

class Derived : private Base {
    std::string child_string;

public:
    Derived() : child_string("파생"), Base() {
        std::cout << "파생 클래스" << std::endl;
    }

    void what() { std::cout << child_string << std::endl; }
};

int main() {
    Base p;
    // Base 에서는 parent_string 이 public 이므로
    // 외부에서 당연히 접근 가능하다.
    std::cout << p.parent_string << std::endl;

    Derived c;
    // 반면에 Derived 에서는 parent_string 이
    // (private 상속을 받았기 때문에) private 이
    // 되어서 외부에서 접근이 불가능하다.
    std::cout << c.parent_string << std::endl;

    return 0;
}
```

컴파일 하였다면

컴파일 오류

```
test.cc: In function 'int main()':
test.cc:31:13: error: 'std::__cxx11::string Base::parent_string'
  ↳ is inaccessible within this context
    std::cout << c.parent_string << std::endl;
               ^~~~~~
test.cc:7:10: note: declared here
  string parent_string;
               ^~~~~~
```

위에 코드 주석에 잘 설명되어 있지만 `Base` 객체에서 `parent_string` 을 접근한다면 `public` 이므로 `main` 함수에서도 잘 접근할 수 있지만 `Derived`에서 `parent_string` 을 접근하려고 한다면, `private` 상속을 받았기 때문에 비록 `Base`에서 `public` 이더라도, `Derived`에서는 `private` 으로 처리됩니다. 따라서 접근할 수 없지요.

사원 관리 프로그램에 적용해보기

그렇다면 이제 우리가 새롭게 습득한 도구인 '상속' 을 `Manager` 와 `Employee` 클래스 사이에 적용해보도록 합시다. 아래는 기존 `Manager` 클래스를 그대로 가져온 것인데, 원래의 `Employee` 클래스와 중복되는 부분을 굵은 글씨로 나타내보았습니다.

```
class Manager {
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;           // 순위 (값이 클 수록 높은 순위)
    int year_of_service;

public:
    Manager(std::string name, int age, std::string position, int rank,
            int year_of_service)
        : name(name),
          age(age),
          position(position),
          rank(rank),
          year_of_service(year_of_service) {}

    // 복사 생성자
    Manager(const Manager& manager) {
        name = manager.name;
        age = manager.age;
```

```

        position = manager.position;
        rank = manager.rank;
        year_of_service = manager.year_of_service;
    }

// 디폴트 생성자
Manager() {}

int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
void print_info() {
    std::cout << name << " (" << position << " , " << age << ", "
        << year_of_service << "년차) ==> " << calculate_pay() << "만원"
        << std::endl;
}
};


```

이제 이를 바꿔보도록 합시다. 참고로, 한 가지 중요한 점은 Manager 의 calculate_pay 함수나 print_info 함수 등에서 Base 의 name, position 등을 참조하고 있기 때문에 Base 의 이 멤버 변수들을 private 속성으로 놔두면 안되고 protected 로 바꿔주어야만 합니다.

아무튼, Employee 를 상속받는 버전으로 바꾼 아래의 Manager 클래스입니다.

```

class Manager : public Employee {
    int year_of_service;

public:
    Manager(std::string name, int age, std::string position, int rank,
            int year_of_service)
        : year_of_service(year_of_service), Employee(name, age, position, rank) {}

// 복사 생성자
Manager(const Manager& manager)
    : Employee(manager.name, manager.age, manager.position, manager.rank) {
    year_of_service = manager.year_of_service;
}

// 디폴트 생성자
Manager() : Employee() {}

int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
void print_info() {
    std::cout << name << " (" << position << " , " << age << ", "
        << year_of_service << "년차) ==> " << calculate_pay() << "만원"
        << std::endl;
}
};


```

먼저 Employee 와 중복되었던 멤버 변수들이 Employee 를 상속함으로써 사라진 것을 볼 수 있습

니다. 그리고 Manager 의 생성자들이 기반 클래스의 생성자를 먼저 호출한다 라는 원칙에 맞게 아래처럼 바뀐 것을 볼 수 있습니다.

```
Manager(std::string name, int age, std::string position, int rank,
        int year_of_service)
: year_of_service(year_of_service), Employee(name, age, position, rank) {}

// 복사 생성자
Manager(const Manager& manager)
: Employee(manager.name, manager.age, manager.position, manager.rank) {
    year_of_service = manager.year_of_service;
}

// 디폴트 생성자
Manager() : Employee() {}
```

위에 굵은 글씨로 표시한 것이 모두 Manager 의 생성자에서 기반 클래스인 Employee 의 생성자를 먼저 호출하는 모습입니다. 상속을 통해서 귀찮게 복사 + 붙여 넣기를 하던 코드를 훨씬 간결하고 알아보기 쉽게 바꿀 수 있게 되었습니다. 전체 코드는 아래와 같습니다.

```
#include <iostream>
#include <string>

class Employee {
protected:
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;           // 순위 (값이 클수록 높은 순위)

public:
    Employee(std::string name, int age, std::string position, int rank)
        : name(name), age(age), position(position), rank(rank) {}

    // 복사 생성자
    Employee(const Employee& employee) {
        name = employee.name;
        age = employee.age;
        position = employee.position;
        rank = employee.rank;
    }

    // 디폴트 생성자
    Employee() {}

    void print_info() {
        std::cout << name << " (" << position << " , " << age << ") ==> "
```

```
        << calculate_pay() << "만원" << std::endl;
    }
    int calculate_pay() { return 200 + rank * 50; }
};

class Manager : public Employee {
    int year_of_service;

public:
    Manager(std::string name, int age, std::string position, int rank,
            int year_of_service)
        : Employee(name, age, position, rank), year_of_service(year_of_service) {}

    // 복사 생성자
    Manager(const Manager& manager)
        : Employee(manager.name, manager.age, manager.position, manager.rank) {
        year_of_service = manager.year_of_service;
    }

    // 디폴트 생성자
    Manager() : Employee() {}

    int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
    void print_info() {
        std::cout << name << " (" << position << " , " << age << ", "
                  << year_of_service << "년차) ==> " << calculate_pay() << "만원"
                  << std::endl;
    }
};

class EmployeeList {
    int alloc_employee; // 할당한 총 직원 수

    int current_employee; // 현재 직원 수
    int current_manager; // 현재 매니저 수

    Employee** employee_list; // 직원 데이터
    Manager** manager_list; // 매니저 데이터

public:
    EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
        employee_list = new Employee*[alloc_employee];
        manager_list = new Manager*[alloc_employee];

        current_employee = 0;
        current_manager = 0;
    }

    void add_employee(Employee* employee) {
        // 사실 current_employee 보다 alloc_employee 가 더
        // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
        // 최대한 단순하게 생각해서 alloc_employee 는
        // 언제나 current_employee 보다 크다고 생각한다.
    }
};
```

```
// (즉 할당된 크기는 현재 총 직원수 보다 많음)
employee_list[current_employee] = employee;
current_employee++;
}
void add_manager(Manager* manager) {
manager_list[current_manager] = manager;
current_manager++;
}
int current_employee_num() { return current_employee + current_manager; }

void print_employee_info() {
int total_pay = 0;
for (int i = 0; i < current_employee; i++) {
employee_list[i]->print_info();
total_pay += employee_list[i]->calculate_pay();
}
for (int i = 0; i < current_manager; i++) {
manager_list[i]->print_info();
total_pay += manager_list[i]->calculate_pay();
}
std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
}
~EmployeeList() {
for (int i = 0; i < current_employee; i++) {
delete employee_list[i];
}
for (int i = 0; i < current_manager; i++) {
delete manager_list[i];
}
delete[] employee_list;
delete[] manager_list;
}
};

int main() {
EmployeeList emp_list(10);
emp_list.add_employee(new Employee("노홍철", 34, "평사원", 1));
emp_list.add_employee(new Employee("하하", 34, "평사원", 1));
emp_list.add_manager(new Manager("유재석", 41, "부장", 7, 12));
emp_list.add_manager(new Manager("정준하", 43, "과장", 4, 15));
emp_list.add_manager(new Manager("박명수", 43, "차장", 5, 13));
emp_list.add_employee(new Employee("정형돈", 36, "대리", 2));
emp_list.add_employee(new Employee("길", 36, "인턴", -2));
emp_list.print_employee_info();
return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
노홍철 (평사원 , 34) ==> 250만원
하하 (평사원 , 34) ==> 250만원
정형돈 (대리 , 36) ==> 300만원
길 (인턴 , 36) ==> 100만원
유재석 (부장 , 41, 12년차) ==> 610만원
정준하 (과장 , 43, 15년차) ==> 475만원
박명수 (차장 , 43, 13년차) ==> 515만원
총 비용 : 2500만원
```

와 같이 잘 실행됩니다.

자 그럼 이번 강좌는 여기에서 마치도록 하겠습니다. 사실 여기 까지만 읽으면 굳이 상속을 왜 쓰는지 이해하기 어렵다고 생각할 수 있습니다. 상속의 진짜 유용함은 다음 강좌에서 다루도록 하겠습니다 :)

가상 함수와 다형성

안녕하세요 여러분! 지난번 강좌는 재미있으셨나요. C++ 이란 산을 넘기 위해 아직도 지나가야 할 관문이 앞에 수 없이 많지만, 그래도 이번 강좌를 통해서 그 관문 하나는 지나갈 수 있으리라 생각합니다. 이번 강좌에서는 객체지향프로그래밍의 핵심 개념 하나에 대해서 배울 것입니다. 기대하세요 :)

is - a 와 has - a

일단 이야기를 진행하기 전에, 어떠한 경우에서 상속을 사용하는지 생각해봅시다. C++에서 상속을 도입한 이유는 단순히 똑같은 코드를 또 쓰는 것을 막기 위한 *Ctrl + C, Ctrl + V* 방지용으로 위한 것이 아닙니다 (물론 그러한 이유도 약간 있겠지만). 실제 이유는 상속이라는 기능을 통해서 객체지향프로그래밍에서 추구하는 실제 객체의 추상화를 좀 더 효과적으로 할 수 있게 되었습니다.

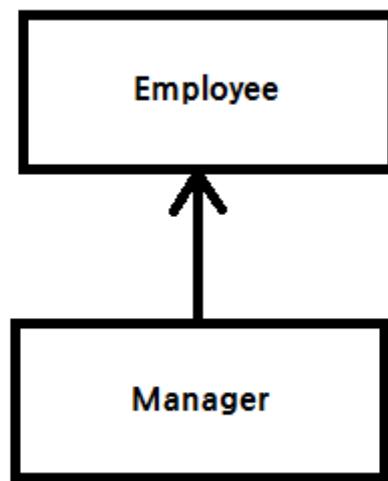
이게 무슨 말이냐면 상속이란 것이 없던 C 언어에서는 어떠한 구조체 사이의 관계를 표현할 수 있는 방법이 없었습니다. 하지만 C++에서 상속이란 것을 도입함으로써, 클래스 사이에 관계를 표현할 수 있게 되었는데, 예를 들어서 Manager 가 Employee 를 상속한다;

```
class Manager : public Employee
```

의 의미는,

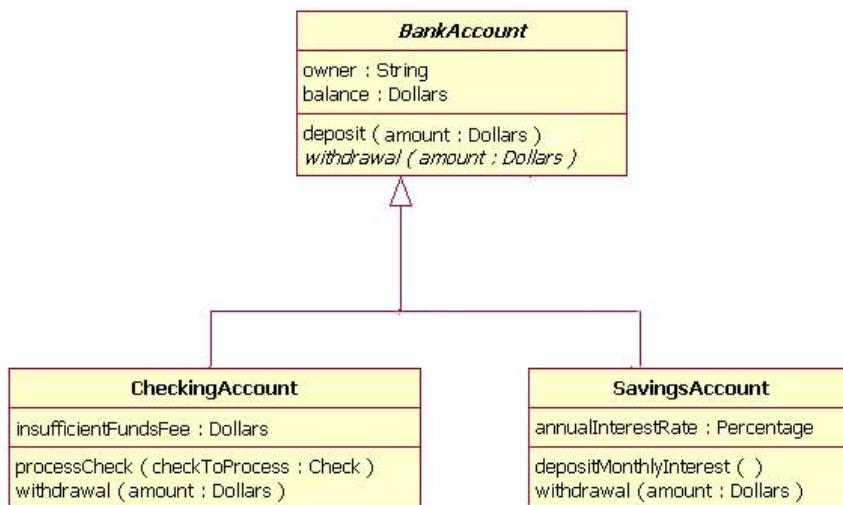
- Manager 클래스는 Employee 의 모든 기능을 포함한다
- Manager 클래스는 Employee 의 기능을 모두 수행할 수 있기 때문에 (Manager에게는 약간 기분 나쁘겠지만) Manager 를 Employee 라고 칭해도 무방하다
- 즉, 모든 Manager 는 Employee 이다
- Manager is a Employee !!

따라서, 모든 상속 관계는 *is a* 관계라고 볼 수 있습니다. 당연한 점은, 이를 뒤바꾸면 성립되지 않는다는 점입니다. 즉 Manager 는 Employee 이지만 Employee 는 Manager 가 아닙니다. 그렇기에, Manager 를 Employee 로 부를 수 있지만, Employee 는 Manager 로 (미안하게도) 부를 수 없습니다.



프로그램 설계 시에 클래스들 간의 상속 관계를 도표로 나타내는 경우가 종종 있는데, 많은 경우 파생 클래스가 기반 클래스를 화살표로 가리키게 그립니다.

실제 세상에서 *is a* 관계로 이루어진 것들은 수 없이 많습니다. 예를 들어, '사람'이라는 클래스가 있다면, '프로그래머는 사람이다 (A programmer is a human)' 이므로, 만일 우리가 프로그래머 클래스를 만든다면 사람이라는 클래스를 상속 받을 수 있도록 구성할 수 있습니다.



위는 또 다른 *is - a* 관계의 예로, BankAccount (은행 계좌)라는 클래스가 있고 Checking Account (자유롭게 입출금이 가능한 계좌지만 이자가 없다) 와 Savings Account (비교적 자유롭게 입출금이 불가능하지만, 매 달 이자가 붙음) 가 이를 상속 받고 있습니다. 즉, 같은 계좌지만 기능이 약간 씩 다른 두 계좌 클래스들이 좀 더 '일반적인' BankAccount 클래스를 상속 받았지요.

이를 통해서 상속의 또 하나의 중요한 특징을 알 수 있습니다. 바로 클래스가 파생되면 파생될 수록 좀 더 특수화 (*구체화; specialize*) 된다는 의미입니다. 즉, Employee 클래스가 일반적인 사원을

위한 클래스 였다면 Manager 클래스 들은 그 일반적인 사원들 중에서도 좀 더 특수한 부류의 사원들을 의미하게 됩니다.

또, BankAccount 도 일반적인 은행 계좌를 위한 클래스 였다면, 이를 상속 받는 CheckingAccount, SavingsAccount 들은 좀 더 구체적인 클래스가 되지요. 반대로, 기반 클래스로 거슬러 올라가면 올라갈 수록 좀 더 일반화 (generalize) 된다고 말합니다.

그렇다면 모든 클래스들의 관계를 is - a 로만 표현할 수 있을까요? 당연히 그렇지 않습니다. 어떤 클래스들 사이에서는 is - a 대신에 has - a 관계가 성립하기도 합니다. 예를 들어서, 간단히 자동차 클래스를 생각해봅시다. 자동차 클래스를 구성하기 위해서는 엔진 클래스, 브레이크 클래스, 오디오 클래스 등 수 많은 클래스들이 필요합니다. 그렇다고 이들 사이에 is a 관계를 도입 할 수 없습니다. (자동차 is a 엔진? 자동차 is a 브레이크?) 그 대신, 이들 사이는 has - a 관계로 쉽게 표현할 수 있습니다.

즉, 자동차는 엔진을 가진다 (자동차 has a 엔진), 자동차는 브레이크를 가진다 (자동차 has a 브레이크) 이와 같이 말이지요. 이런 has - a 관계는 우리가 흔히 해왔듯이 다음과 같이 클래스로 나타내면 됩니다.

```
class Car {
private:
    Engine e;
    Brake b; // 아마 break 아니냐고 생각하는 사람들이 있을 텐데 :)
    ...
};
```

또 다른 예로 바로 우리의 EmployeeList 를 들을 수도 있습니다. EmployeeList 는 Employee 들과 has - a 관계 이지요. 따라서, 실제로 EmployeeList 클래스를 보면

```
class EmployeeList {
    int alloc_employee; // 할당한 총 직원 수
    int current_employee; // 현재 직원 수
    Employee **employee_list; // 직원 데이터
```

와 같이 Employee 를 포함하고 있음을 알 수 있습니다.

(다시 보는) 오버라이딩

```
#include <iostream>
#include <string>

class Base {
    std::string s;
```

```

public:
Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
std::string s;

public:
Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

void what() { std::cout << s << std::endl; }
};

int main() {
std::cout << " === 기반 클래스 생성 ===" << std::endl;
Base p;

p.what();

std::cout << " === 파생 클래스 생성 ===" << std::endl;
Derived c;

c.what();

return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

==== 기반 클래스 생성 ====
기반 클래스
기반
==== 파생 클래스 생성 ====
기반 클래스
파생 클래스
파생

```

이미 저번 강좌에서도 이야기 했었지만, Base에서 what을 호출하면 당연히 Base의 what이 실행되어서 '기반'라고 나오고, Base를 상속받는 Derived 클래스에서 what을 호출하면, Derived의 what이 Base의 what을 오버라이드 해서 Derived의 what이 호출되게 됩니다.

이번에는 코드를 약간 변형해보도록 하겠습니다.

```

#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
    Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

int main() {
    Base p;
    Derived c;

    std::cout << "==== 포인터 버전 ===" << std::endl;
    Base* p_c = &c;
    p_c->what();

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

기반 클래스
기반 클래스
파생 클래스
==== 포인터 버전 ===
기반

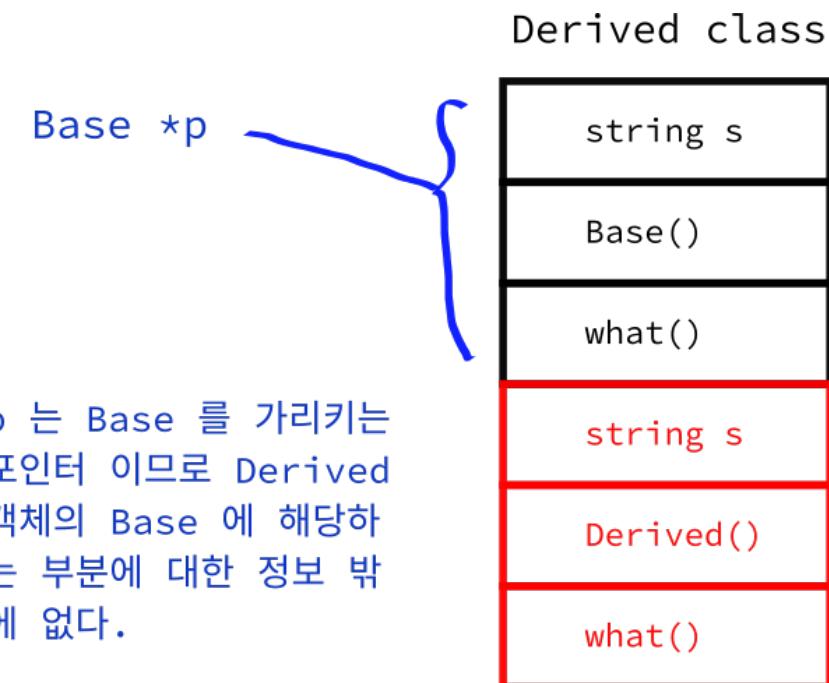
```

이번에는 `Derived` 의 객체 `c` 를 `Base` 객체를 가리키는 포인터에 넣었습니다.

```
Base* p_c = &c;
```

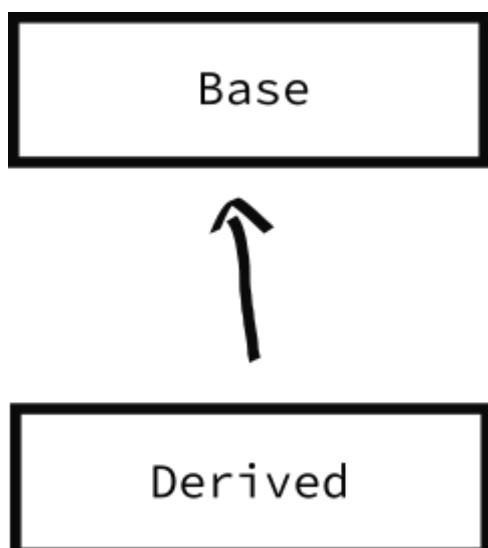
어떤 분들은 이와 같은 대입이 가능하냐고 물을 수 있습니다. Base 와 Derived 는 다른 클래스 이니까요. 하지만, 그 분들이 간과하고 있는 점은 Derived 가 Base 를 상속 받고 있다는 점입니다. 상속 받는다면 뭐죠? **Derived is a Base**

즉 (말이 조금 이상하지만) Derived 객체 c 도 어떻게 보면 Base 객체이기 때문에 Base 객체를 가리키는 포인터가 c 를 가리켜도 무방하다는 것입니다. 이를 그림으로 표현한다면 아래와 같습니다.



그 대신 p 는 엄연한 Base 객체를 가리키는 포인터입니다. 따라서, p 의 what 을 실행한다면 p 는 당연히 '아 Base 의 what 을 실행해 주어야 겠구나' 하고, Base 의 what 을 실행해서, Base 의 what 은 Base 의 s 를 출력하게 됩니다. 따라서 위 처럼 '기반' 가 출력됩니다.

이러한 형태의 캐스팅을 (즉 파생 클래스에서 기반 클래스로 캐스팅 하는 것) 을 업 캐스팅 이라고 부릅니다.



위 그림을 보면 왜 업 캐스팅이라 부르는지 이해가 확 되지요.

그렇다면 업 캐스팅의 반대인 다운 캐스팅도 있을까요?

```
#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
    Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

int main() {
    Base p;
    Derived c;

    std::cout << "==== 포인터 버전 ===" << std::endl;
    Derived* p_p = &p;
    p_p->what();

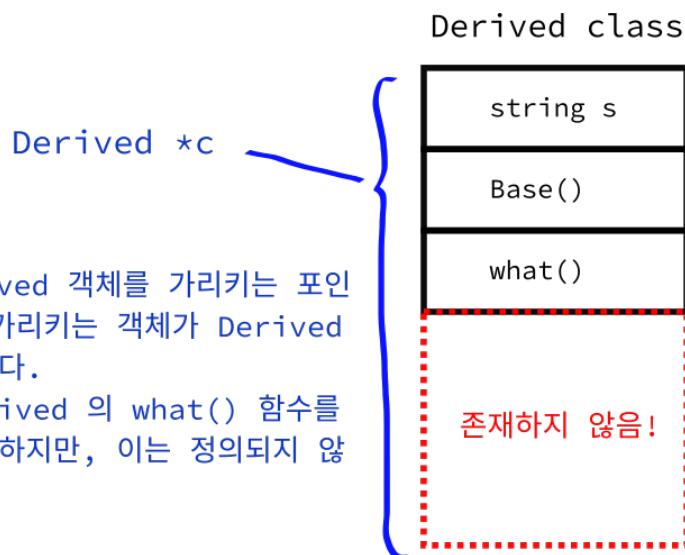
    return 0;
}
```

컴파일 한다면 다음과 같은 오류 메세지를 볼 수 있습니다.

컴파일 오류

```
error C2440: 'initializing' : cannot convert from 'Base *' to
                           '^Derived *'
```

사실 위와 같은 오류가 발생한 이유는 간단합니다.



만일 Derived* 포인터가 Base 객체를 가리킨다고 해봅시다. 그렇다면 p_p->what() 하게 된다면 Derived의 what 함수가 호출되어야만 하는데, 이는 불가능 합니다. (왜냐하면 p_p 가 가리키는 객체는 Base 객체 이므로 Derived에 대한 정보가 없습니다). 따라서, 이와 같은 문제를 막기 위해서 컴파일러 상에서 함부로 다운 캐스팅 하는 것을 금지하고 있습니다.

하지만 다음과 같은 상황은 어떨까요.

```
#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
    Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

int main() {
    Base p;
    Derived c;

    std::cout << "==== 포인터 버전 ===" << std::endl;
    Base* p_p = &c;
```

```

Derived* p_c = p_p;
p_c->what();

return 0;
}

```

컴파일 하였다면

컴파일 오류

```

error C2440: 'initializing' : cannot convert from 'Base *' to
        'Derived *'

```

`Derived* p_c`에 `Base *`를 대입하면 안된다는 똑같은 오류가 발생합니다. 하지만 우리는 `p_p`가 가리키는 것이 `Base` 객체가 아니라 `Derived` 객체라는 사실을 알고 있습니다. 그렇기 때문에 비록 `Base *` 포인터를 다운 캐스팅 함에도 불구하고 `p_p`가 실제로는 `Derived` 객체를 가리키기 때문에

```
Derived* p_c = p_p;
```

를 해도 전혀 문제가 없습니다. 이를 위해서는 아래처럼 강제적으로 타입 변환을 하면 됩니다.

```
Derived* p_c = static_cast<Derived*>(p_p);
```

비록 약간은 위험하지만 (만일 `p_p` 가 사실은 `Derived` 객체를 가리키지 않는다면?) 컴파일 오류를 발생시키지 않고 성공적으로 컴파일 할 수 있습니다. 그렇다면 만일 `p_p` 가 사실 `Base` 객체를 가리키는데 강제적으로 타입 변환을 해서 `what` 을 실행한다면 어떨까요?

```

#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    void what() { std::cout << s << std::endl; }
};

class Derived : public Base {
    std::string s;

public:

```

```

Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

void what() { std::cout << s << std::endl; }
};

int main() {
Base p;
Derived c;

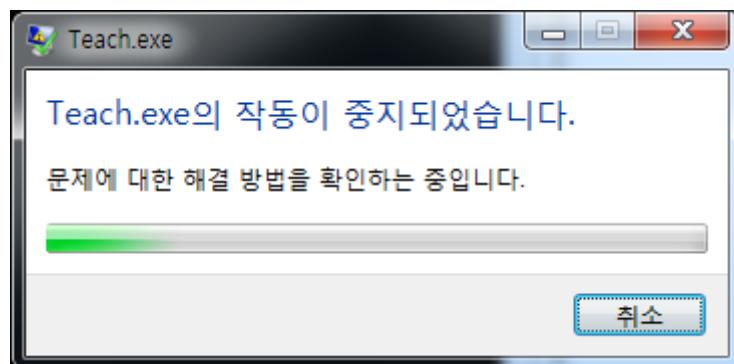
std::cout << "==== 포인터 버전 ===" << std::endl;
Base* p_p = &p;

Derived* p_c = static_cast<Derived*>(p_p);
p_c->what();

return 0;
}

```

성공적으로 컴파일 하였다면



와 같은 런타임 오류가 발생하게 됩니다.

이러한 강제적으로 다운 캐스팅을 하는 경우, 컴파일 타임에서 오류를 찾아내기 매우 힘들기 때문에 다운 캐스팅은 작동이 보장되지 않는 한 매우매우 권장하지 않는 바입니다.

dyanmic_cast

이러한 캐스팅에 따른 오류를 미연에 방지하기 위해서, C++에서는 상속 관계에 있는 두 포인터들 간에 캐스팅을 해주는 `dynamic_cast`라는 것을 지원합니다. 이를 사용하는 방법은 `static_cast` 와 거의 동일합니다.

```
Derived* p_c = dyanmic_cast<Derived*>(p_p);
```

하지만 위의 경우 컴파일 하게 된다면

컴파일 오류

```
test.cc: In function 'int main()':
test.cc:27:44: error: cannot dynamic_cast 'p_p' (of type 'class
    ↵ Base*') to type 'class Derived*' (source type is not
    ↵ polymorphic)
    Derived* p_c = dynamic_cast<Derived*>(p_p);
```

와 같이 캐스팅 할 수 없다는 컴파일 오류가 발생하게 됩니다.

EmployeeList 다시 보기

자, 그럼 이제 위에서 다룬 내용을 가지고 EmployeeList 를 어떻게 하면 좀 더 간단하게 만들 수 있을지 생각해봅시다.

```
class EmployeeList {
    int alloc_employee; // 할당한 총 직원 수

    int current_employee; // 현재 직원 수
    int current_manager; // 현재 매니저 수

    Employee **employee_list; // 직원 데이터
    Manager **manager_list; // 매니저 데이터
    // ...
```

위와 같은 구성에서 가장 문제가 되는 것이 각 클래스 별로 데이터를 따로 보관해야 된다는 것입니다. 즉 Employee 들은 Employee * 가 가리켜야 하고, Manager 들은 Manager * 가 가리켜야 합니다. 만일 무한 상사에서 클래스 하나를 더 추가해달라고 연락이 왔다면 때릴지도 모르겠지요.

하지만, 한 가지 위에서 배운 사실은, 업 캐스팅은 매우 자유롭게 수행될 수 있다는 점입니다. 즉, Employee * 가 Manager 객체를 가리켜도 별 문제가 없다는 것이지요. 그렇다면 manager_list 를 그냥 지워 버리고, employee_list 가 Employee, Manager 상관없이 가리키게 해도 될까요? 그러면 참 좋겠지만 다음과 같은 문제점이 있습니다.

```
void print_employee_info() {
    int total_pay = 0;
    for (int i = 0; i < current_employee; i++) {
        employee_list[i]->print_info();
        total_pay += employee_list[i]->calculate_pay();
    }
    ...
}
```

바로 여기서, `employee_list[i]->print_info()` 를 하게 되면 무조건 Employee 클래스의 `print_info` 함수가 호출된다는 것입니다. 왜냐하면 위에서도 이야기 하였듯이, `employee_list[i]` 는 Employee 객체를 가리키는 포인터이기 때문에 자신이 가리키는 객체가 Employee 객체라고 생각합니다.

하지만 우리는 Manager 객체와 Employee 객체 모두 `Employee*` 가 가리키도록 하였으므로, 만일 `employee_list[i]` 가 가리키는 것이 Manager 객체 일 때, Manager 의 `print_info` 함수가 아니라 Employee 의 `print_info` 함수가 호출되어서 다른 결과를 냅니다.

마찬가지로 `calculate_pay` 함수도 Manager 의 `calculate_pay` 가 호출 되어야 하는데 Employee 의 `calculate_pay` 가 호출되어어서 (월급이 더 적게 나오는 괴상) 틀린 결과가 나옵니다. 나쁜 회사였으면 환영할 일이였겠지만 착한 우리의 입장에선 이 문제를 꼭 해결해야 합니다.

실제로,

```
#include <iostream>
#include <string>

class Employee {
protected:
    std::string name;
    int age;

    std::string position; // 직책 (이름)
    int rank;             // 순위 (값이 클 수록 높은 순위)

public:
    Employee(std::string name, int age, std::string position, int rank)
        : name(name), age(age), position(position), rank(rank) {}

    // 복사 생성자
    Employee(const Employee& employee) {
        name = employee.name;
        age = employee.age;
        position = employee.position;
        rank = employee.rank;
    }

    // 디폴트 생성자
    Employee() {}

    void print_info() {
        std::cout << name << " (" << position << " , " << age << ") ==> "
            << calculate_pay() << "만원" << std::endl;
    }
    int calculate_pay() { return 200 + rank * 50; }
};

class Manager : public Employee {
```

```
int year_of_service;

public:
Manager(std::string name, int age, std::string position, int rank,
         int year_of_service)
    : year_of_service(year_of_service), Employee(name, age, position, rank) {}

// 복사 생성자
Manager(const Manager& manager)
    : Employee(manager.name, manager.age, manager.position, manager.rank) {
    year_of_service = manager.year_of_service;
}

// 디폴트 생성자
Manager() : Employee() {}

int calculate_pay() { return 200 + rank * 50 + 5 * year_of_service; }
void print_info() {
    std::cout << name << " (" << position << " , " << age << ", "
        << year_of_service << "년차) ==> " << calculate_pay() << "만원"
        << std::endl;
}
};

class EmployeeList {
    int alloc_employee;           // 할당한 총 직원 수
    int current_employee;         // 현재 직원 수
    Employee** employee_list;    // 직원 데이터

public:
EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
    employee_list = new Employee*[alloc_employee];

    current_employee = 0;
}
void add_employee(Employee* employee) {
    // 사실 current_employee 보다 alloc_employee 가 더
    // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
    // 최대한 단순하게 생각해서 alloc_employee 는
    // 언제나 current_employee 보다 크다고 생각한다.
    // (즉 할당된 크기는 현재 총 직원수 보다 많음)
    employee_list[current_employee] = employee;
    current_employee++;
}
int current_employee_num() { return current_employee; }

void print_employee_info() {
    int total_pay = 0;
    for (int i = 0; i < current_employee; i++) {
        employee_list[i]->print_info();
        total_pay += employee_list[i]->calculate_pay();
    }
}
```

```

        std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
    }
~EmployeeList() {
    for (int i = 0; i < current_employee; i++) {
        delete employee_list[i];
    }
    delete[] employee_list;
}
};

int main() {
EmployeeList emp_list(10);
emp_list.add_employee(new Employee("노홍철", 34, "평사원", 1));
emp_list.add_employee(new Employee("하하", 34, "평사원", 1));
emp_list.add_employee(new Manager("유재석", 41, "부장", 7, 12));
emp_list.add_employee(new Manager("정준하", 43, "과장", 4, 15));
emp_list.add_employee(new Manager("박명수", 43, "차장", 5, 13));
emp_list.add_employee(new Employee("정형돈", 36, "대리", 2));
emp_list.add_employee(new Employee("길", 36, "인턴", -2));
emp_list.print_employee_info();
return 0;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

노홍철 (평사원 , 34) ==> 250만원
하하 (평사원 , 34) ==> 250만원
유재석 (부장 , 41) ==> 550만원
정준하 (과장 , 43) ==> 400만원
박명수 (차장 , 43) ==> 450만원
정형돈 (대리 , 36) ==> 300만원
길 (인턴 , 36) ==> 100만원
총 비용 : 2300만원

```

와 같이 전부다 Employee 의 print_info 와 calculate_pay 함수가 호출되어서 원래 결과와 달라집니다.

그런데 놀랍게도 이러한 문제를 5초 만에 해결할 수 있는 방법이 있습니다.

virtual 키워드

EmployeeList 문제를 해결하기 전에 좀 더 간단한 예시로 살펴보겠습니다.

```
#include <iostream>

class Base {
public:
    Base() { std::cout << "기반 클래스" << std::endl; }

    virtual void what() { std::cout << "기반 클래스의 what()" << std::endl; }
};

class Derived : public Base {
public:
    Derived() : Base() { std::cout << "파생 클래스" << std::endl; }

    void what() { std::cout << "파생 클래스의 what()" << std::endl; }
};

int main() {
    Base p;
    Derived c;

    Base* p_c = &c;
    Base* p_p = &p;

    std::cout << " == 실제 객체는 Base == "
    p_p->what();

    std::cout << " == 실제 객체는 Derived == "
    p_c->what();

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
기반 클래스
기반 클래스
파생 클래스
== 실제 객체는 Base ==
기반 클래스의 what()
== 실제 객체는 Derived ==
파생 클래스의 what()
```

어라? 위 결과를 보셨다면 놀라움을 금치 못하셨을 것입니다.

```

Base* p_c = &c;
Base* p_p = &p;

std::cout << " == 실제 객체는 Base == " << std::endl;
p_p->what();

std::cout << " == 실제 객체는 Derived == " << std::endl;
p_c->what();

```

분명히 여기서 p_p 와 p_c 모두 Base 객체를 가리키는 포인터입니다. 따라서, p_p->what() 와 p_c->what() 을 하면 모두 Base 객체의 what() 함수가 실행되어서 둘 다 '기반' 라고 출력이 되어야만 했습니다.

그런데, 놀랍게도, 실제 p_p 와 p_c 가 무엇과 결합해 있는지 아는 것 처럼 (p_p 는 Base 객체를 가리키고, p_c 는 Derived 객체를 가리킴) 이에 따른 적절한 what 함수를 호출해준 것입니다. 이와 같은 일이 가능해진 이유는 바로;

```

class Base {
public:
    Base() { std::cout << "기반 클래스" << std::endl; }

    virtual void what() { std::cout << "기반 클래스의 what()" << std::endl; }
};

```

이 virtual 키워드 하나 때문입니다. 이 virtual 키워드는, 다음과 같은 역할을 합니다.

```
p_c->what();
```

위 코드를 실행시에 (런타임), 컴퓨터 입장에서;

"흠, p_c 는 Base 포인터니까 Base 의 what() 을 실행해야지" "어 근데 what 이 virtual 이네?" "잠깐. 이거 실제 Base 객체 맞어? 아니네 Derived 객체네" "그럼 Derived 의 what 을 실행해 야지"

반면에

```
p_p->what();
```

였을 경우에는

"흠, p_c 는 Base 포인터니까 Base 의 what() 을 실행해야지" "어 근데 what 이 virtual 이네?"
 "잠깐. 이거 실제 Base 객체 맞어? 어 맞네." "Base 의 what 을 실행하자"

이렇게 컴파일 시에 어떤 함수가 실행될지 정해지지 않고 런타임 시에 정해지는 일을 가리켜서 동적 바인딩(dynamic binding) 이라고 부릅니다. 즉,

```
p_c->what();
```

에서 Derived 의 what 을 실행할지, Base 의 what 을 실행하지 결정은 런타임에 이루어지게 됩니다. 물론 위 코드에선 컴파일 시에 무조건 p_c->what() 이 Derived 의 what 이 실행되도록 정해진 거 아니냐고 물을 수 있지만 다음과 같은 상황을 생각해보세요.

```
// i 는 사용자로부터 입력받는 변수
if (i == 1) {
    p_p = &c;
} else {
    p_p = &p;
}
p_p->what();
```

이렇게 된다면 p_p->what() 이 어떤 what 일지에는 런타임에 정해지겠지요? 물론 동적 바인딩의 반대말로 정적 바인딩(static binding) 이란 말도 있습니다. 이는 컴파일 타임에 어떤 함수가 호출될지 정해지는 것으로 여태까지 여러분이 알고 오셨던 함수에 해당합니다.

덧붙여서, **virtual** 키워드가 붙은 함수를 가상 함수(virtual function) 라고 부릅니다. 이렇게 파생 클래스의 함수가 기반 클래스의 함수를 오버라이드 하기 위해서는 두 함수의 꼴이 정확히 같아야 합니다.

override 키워드

```
#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    virtual void what() { std::cout << s << std::endl; }
};
```

```

class Derived : public Base {
    std::string s;

public:
    Derived() : s("파생"), Base() { std::cout << "파생 클래스" << std::endl; }

    void what() override { std::cout << s << std::endl; }
};

```

C++ 11 에서는 파생 클래스에서 기반 클래스의 가상 함수를 오버라이드 하는 경우, **override** 키워드를 통해서 명시적으로 나타낼 수 있습니다.

```
void what() override { std::cout << s << std::endl; }
```

위 경우 **Derived** 클래스의 **what** 함수는 **Base** 클래스의 **what** 함수를 오버라이드 하므로, **override** 키워드를 통해 이를 알려주고 있습니다.

override 키워드를 사용하게 되면, 실수로 오버라이드를 하지 않는 경우를 막을 수 있습니다. 예를 들어서;

```

#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    virtual void incorrect() { std::cout << "기반 클래스" << std::endl; }
};

class Derived : public Base {
    std::string s;

public:
    Derived() : Base(), s("파생") {}

    void incorrect() const { std::cout << "파생 클래스" << std::endl; }
};

int main() {
    Base p;
    Derived c;

    Base* p_c = &c;
    Base* p_p = &p;

    std::cout << " == 실제 객체는 Base == " << std::endl;
}

```

```

p_p->incorrect();

std::cout << " == 실제 객체는 Derived == " << std::endl;
p_c->incorrect();
return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

기반 클래스
기반 클래스
== 실제 객체는 Base ==
기반 클래스
== 실제 객체는 Derived ==
기반 클래스

```

와 같이 `incorrect` 함수가 제대로 오버라이드 되지 않았음을 알 수 있습니다. 그 이유는 `Base` 의 `incorrect` 함수와 `Derived` 의 `incorrect` 함수는 거의 똑같이 생기기는 했지만 사실 다르기 때문입니다. 왜냐하면 `Derived` 의 `incorrect` 함수는 상수 함수이고, `Base` 의 `incorrect`는 아니기 때문이지요.

따라서 컴파일러 입장에서 두 함수는 다른 함수로 간주되므로, `p_c->incorrect()` 를 하였을 때 `Derived` 의 `incorrect` 함수가 `Base` 의 `incorrect` 함수를 오버라이드 하는 것이 아니라, 그냥 `Base` 의 `incorrect` 함수를 호출하는 셈이 됩니다.

만약에 여러분의 의도가 `Derived` 의 `incorrect` 함수가 기반 클래스를 오버라이드 하는 것이였다면 큰 문제가 될 것입니다. 이 버그는 컴파일 타임에 잡을 수 없게 되니까요.

하지만, 실제로 `Derived` 의 `incorrect` 함수를 기반 클래스의 `incorrect` 함수를 오버라이드 하기 위해서 만들었다면, `override` 키워드를 써야겠지요.

```

#include <iostream>
#include <string>

class Base {
    std::string s;

public:
    Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

    virtual void incorrect() { std::cout << "기반 클래스" << std::endl; }
};

class Derived : public Base {

```

```

std::string s;

public:
Derived() : Base(), s("파생") {}

void incorrect() const override { std::cout << "파생 클래스" << std::endl; }

int main() {
Base p;
Derived c;

Base* p_c = &c;
Base* p_p = &p;

std::cout << " == 실제 객체는 Base == " << std::endl;
p_p->incorrect();

std::cout << " == 실제 객체는 Derived == " << std::endl;
p_c->incorrect();
return 0;
}

```

컴파일 하였다면

컴파일 오류

```

test.cc:19:8: error: 'void Derived::incorrect() const' marked
→ 'override', but does not override

```

위와 같이 Derived 의 incorrect 함수가 override 한다고 써있지만, 실제로는 아무것도 오버라이드 하지 않는다고 오류가 발생하게 됩니다. 만일 const 키워드를 지워준다면

```

#include <iostream>
#include <string>

class Base {
std::string s;

public:
Base() : s("기반") { std::cout << "기반 클래스" << std::endl; }

virtual void incorrect() { std::cout << "기반 클래스" << std::endl; }
};

class Derived : public Base {
std::string s;

public:
Derived() : Base(), s("파생") {}

```

```

void incorrect() override { std::cout << "파생 클래스" << std::endl; }
};

int main() {
    Base p;
    Derived c;

    Base* p_c = &c;
    Base* p_p = &p;

    std::cout << " == 실제 객체는 Base == " << std::endl;
    p_p->incorrect();

    std::cout << " == 실제 객체는 Derived == " << std::endl;
    p_c->incorrect();
    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

기반 클래스
기반 클래스
== 실제 객체는 Base ==
기반 클래스
== 실제 객체는 Derived ==
파생 클래스

```

제대로 실행됨을 알 수 있습니다.

그렇다면 Employee 문제를 어떻게 해결할까

이제 여러분은 그동안 골머리를 썩여왔던 EmployeeList 문제도 해결할 수 있게 되었습니다. 단순히 Employee 클래스의 calculate_pay 함수와 print_info 함수 앞에 virtual 만 붙여주면 깔끔하게 정리 되지요.

```

#include <iostream>
#include <string>

class Employee {
protected:
    std::string name;

```

```
int age;

std::string position; // 직책(이름)
int rank;           // 순위(값이 클수록 높은 순위)

public:
Employee(std::string name, int age, std::string position, int rank)
    : name(name), age(age), position(position), rank(rank) {}

// 복사 생성자
Employee(const Employee& employee) {
    name = employee.name;
    age = employee.age;
    position = employee.position;
    rank = employee.rank;
}

// 디폴트 생성자
Employee() {}

virtual void print_info() {
    std::cout << name << " (" << position << " , " << age << ") => "
        << calculate_pay() << "만원" << std::endl;
}
virtual int calculate_pay() { return 200 + rank * 50; }
};

class Manager : public Employee {
    int year_of_service;

public:
Manager(std::string name, int age, std::string position, int rank,
        int year_of_service)
    : year_of_service(year_of_service), Employee(name, age, position, rank) {}

    int calculate_pay() override { return 200 + rank * 50 + 5 * year_of_service; }
    void print_info() override {
        std::cout << name << " (" << position << " , " << age << ", "
            << year_of_service << "년차" => " << calculate_pay() << "만원"
            << std::endl;
    }
};

class EmployeeList {
    int alloc_employee;      // 할당한 총 직원 수
    int current_employee;    // 현재 직원 수
    Employee** employee_list; // 직원 데이터

public:
EmployeeList(int alloc_employee) : alloc_employee(alloc_employee) {
    employee_list = new Employee*[alloc_employee];
    current_employee = 0;
```

```

    }

void add_employee(Employee* employee) {
    // 사실 current_employee 보다 alloc_employee 가 더
    // 많아지는 경우 반드시 재할당을 해야 하지만, 여기서는
    // 최대한 단순하게 생각해서 alloc_employee 는
    // 언제나 current_employee 보다 크다고 생각한다.
    // (즉 할당된 크기는 현재 총 직원수 보다 많음)
    employee_list[current_employee] = employee;
    current_employee++;
}

int current_employee_num() { return current_employee; }

void print_employee_info() {
    int total_pay = 0;
    for (int i = 0; i < current_employee; i++) {
        employee_list[i]->print_info();
        total_pay += employee_list[i]->calculate_pay();
    }

    std::cout << "총 비용 : " << total_pay << "만원" << std::endl;
}

~EmployeeList() {
    for (int i = 0; i < current_employee; i++) {
        delete employee_list[i];
    }
    delete[] employee_list;
}
};

int main() {
    EmployeeList emp_list(10);
    emp_list.add_employee(new Employee("노홍철", 34, "평사원", 1));
    emp_list.add_employee(new Employee("하하", 34, "평사원", 1));

    emp_list.add_employee(new Manager("유재석", 41, "부장", 7, 12));
    emp_list.add_employee(new Manager("정준하", 43, "과장", 4, 15));
    emp_list.add_employee(new Manager("박명수", 43, "차장", 5, 13));
    emp_list.add_employee(new Employee("정형돈", 36, "대리", 2));
    emp_list.add_employee(new Employee("길", 36, "인턴", -2));
    emp_list.print_employee_info();
    return 0;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

노홍철 (평사원 , 34) ==> 250만원
하하 (평사원 , 34) ==> 250만원

```

```
유재석 (부장 , 41, 12년차) ==> 610만원  
정준하 (과장 , 43, 15년차) ==> 475만원  
박명수 (차장 , 43, 13년차) ==> 515만원  
정형돈 (대리 , 36) ==> 300만원  
길 (인턴 , 36) ==> 100만원  
총 비용 : 2500만원
```

와 같이 비록 `Employee*` 가 가리키고 있음에도 불구하고 `Manager` 면 `Manager` 의 함수를, `Employee` 면 `Employee` 의 함수를 잘 호출하고 있음을 알 수 있습니다. 물론 바뀐 것은 단 두 단어. `virtual` 키워들을 `Employee` 의 함수들 앞에 추가해놓은 것 뿐이지요.

```
employee_list[i]->print_info();  
total_pay += employee_list[i]->calculate_pay();
```

이 두 부분은 `employee_list[i]` 가 `Employee` 냐 `Manager` 에 따라서 다르게 동작하게 됩니다. 이렇게 같은 `print_info` 함수를 호출했음에도 불구하고 어떤 경우는 `Employee` 의 것이, 어떤 경우는 `Manager` 의 것이 호출되는 일; 즉 하나의 메소드를 호출했음에도 불구하고 여러가지 다른 작업들을 하는 것을 바로 **다형성(polymorphism)** 이라고 부릅니다.

참고로, 다형성을 뜻하는 영어 단어인 *polymorphism* 은, 여러개를 의미하는 그리스어 'poly' 와, 모습, 모양을 뜻하는 그리스어 'morphism' 에서 온 단어로 '여러가지 형태' 라는 의미입니다.

자 그러면 이번 강좌는 여기에서 마치도록 하겠습니다. 아마도 `virtual` 키워드를 처음 접했더라면 머가 어떻게 되는지 많이 헷갈릴 수 있는데 꼭 여러분 테스트 프로그램을 만들어보아서 확실히 이해하고 넘어가도록 합시다 :)

생각해보기

문제 1

그렇다면 프로그램 내부적으로 `virtual` 함수들은 어떻게 처리될까요? 즉, 이 포인터가 어떤 객체를 가리키는지 어떻게 알 수 있을까요? (난이도 : 上)

상속에 관련된 잡다한 내용들

안녕하세요 여러분. 지난 강좌에서는 놀라움의 연속이었던 `virtual` 키워드의 기능에 대해서 설명하였습니다. `virtual` 키워드를 통해서 동적 바인딩이라는 것을 이루어 낼 수 있었지요. 이번 강좌에서는 가상 함수와 상속에 관련하여 잡다한 내용들을 모두 짚고 넘어가도록 하겠습니다.

지난 시간에 배웠던 것을 간단히 정리해보자면 다음과 같습니다. `Parent` 클래스와 `Child` 클래스에 모두 `f`라는 가상함수가 정의되어 있고, `Child` 클래스가 `Parent`를 상속 받는다고 해봅시다. 그런 다음에 동일한 `Parent*` 타입의 포인터들도 각각 `Parent` 객체와 `Child` 객체를 가리킨다고 해봅시다.

```
Parent* p = new Parent();
Parent* c = new Child();
```

컴퓨터 입장에서 `p` 와 `c` 모두 `Parent`를 가리키는 포인터들이므로, 당연히

```
p->f();
c->f();
```

를 했을 때 모두 `Parent`의 `f()` 가 호출되어야 하겠지만, 실제로는 `f` 가 가상함수므로, '실제로 `p` 와 `c` 가 가리키는 객체의' `f`, 즉 `p->f()` 는 `Parent`의 `f` 를, `c->f()` 는 `Child`의 `f` 가 호출됩니다. 이와 같은 일이 가능한 이유는 `f` 를 가상함수로 만들었기 때문입니다.

`virtual` 소멸자

사실 클래스의 상속을 사용함으로써 중요하게 처리해야 되는 부분이 있습니다. 상속 시에, 소멸자를 가상함수로 만들어야 된다는 점입니다.

```
#include <iostream>

class Parent {
public:
    Parent() { std::cout << "Parent 생성자 호출" << std::endl; }
    ~Parent() { std::cout << "Parent 소멸자 호출" << std::endl; }
};

class Child : public Parent {
public:
    Child() : Parent() { std::cout << "Child 생성자 호출" << std::endl; }
    ~Child() { std::cout << "Child 소멸자 호출" << std::endl; }
};

int main() {
```

```

std::cout << "---- 평범한 Child 만들었을 때 ---" << std::endl;
{ Child c; }
std::cout << "---- Parent 포인터로 Child 가리켰을 때 ---" << std::endl;
{
    Parent *p = new Child();
    delete p;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

---- 평범한 Child 만들었을 때 ---
Parent 생성자 호출
Child 생성자 호출
Child 소멸자 호출
Parent 소멸자 호출
---- Parent 포인터로 Child 가리켰을 때 ---
Parent 생성자 호출
Child 생성자 호출
Parent 소멸자 호출

```

와 같이 나옵니다.

일단 평범하게 Child 객체를 만든 부분을 살펴봅시다.

```

std::cout << "---- 평범한 Child 만들었을 때 ---" << std::endl;
{ Child c; }

```

생성자와 소멸자의 호출 순서를 살펴보자면, Parent 생성자 → Child 생성자 → Child 소멸자 → Parent 소멸자 순으로 호출됨을 알 수 있습니다. 이와 같은 과정이 당연한 이유는 객체를 만들고 소멸시키는 일을 집을 짓고 철거하는 일로 비유할 수 있습니다.

집을 지을 때에는 큰 틀, 즉 기초공사를 하고 건물을 세운 다음에 (Parent 생성자 호출), 집 내부 공사, 인테리어, 가구 배치 등을 하게 됩니다 (Child 생성자 호출). 그리고 역으로 집을 철거할 때에는 안에 있는 내용물들을 모두 제거한 뒤에 (Child 소멸자 호출), 집 구조물을 철거하겠지요 (Parent 소멸자 호출).

그런데 문제는 그 아래 Parent 포인터가 Child 객체를 가리킬 때입니다.

```

std::cout << "---- Parent 포인터로 Child 가리켰을 때 ---" << std::endl;
{

```

```

Parent *p = new Child();
delete p;
}

```

`delete p` 를 하더라도, `p` 가 가리키는 것은 `Parent` 객체가 아닌 `Child` 객체 이기 때문에, 위에서 보통의 `Child` 객체가 소멸되는 것과 같은 순서로 생성자와 소멸자들이 호출되어야만 합니다. 그런데 실제로는, `Child` 소멸자가 호출되지 않습니다.

소멸자가 호출되지 않는다면 여러가지 문제가 생길 수 있습니다. 예를 들어서, `Child` 객체에서 메모리를 동적으로 할당하고 소멸자에서 해제하는데, 소멸자가 호출 안된다면 메모리 누수(**memory leak**)가 생기겠지요.

하지만 `virtual` 키워드를 배운 이상 여러분은 무엇을 해야 하는지 알고 계실 것입니다. 단순히 `Parent` 의 소멸자를 `virtual` 로 만들어버리면 됩니다. `Parent` 의 소멸자를 `virtual` 로 만들면, `p` 가 소멸자를 호출할 때, `Child` 의 소멸자를 성공적으로 호출할 수 있게 됩니다.

```

#include <iostream>

class Parent {
public:
    Parent() { std::cout << "Parent 생성자 호출" << std::endl; }
    virtual ~Parent() { std::cout << "Parent 소멸자 호출" << std::endl; }
};

class Child : public Parent {
public:
    Child() : Parent() { std::cout << "Child 생성자 호출" << std::endl; }
    ~Child() { std::cout << "Child 소멸자 호출" << std::endl; }
};

int main() {
    std::cout << "---- 평범한 Child 만들었을 때 ---" << std::endl;
    {
        // 이 {} 를 빠져나가면 c 가 소멸된다.
        Child c;
    }
    std::cout << "---- Parent 포인터로 Child 가리켰을 때 ---" << std::endl;
    {
        Parent *p = new Child();
        delete p;
    }
}

```

성공적으로 컴파일 하였다면

실행 결과

```
---- 평범한 Child 만들었을 때 ---
```

```

Parent 생성자 호출
Child 생성자 호출
Child 소멸자 호출
Parent 소멸자 호출
--- Parent 포인터로 Child 가리켰을 때 ---
Parent 생성자 호출
Child 생성자 호출
Child 소멸자 호출
Parent 소멸자 호출

```

와 같이 제대로 Child 소멸자가 호출됨을 알 수 있습니다.

여기서 한 가지 질문을 하자면, 그렇다면 왜 Parent 소멸자는 호출이 되었는가 인데, 이는 Child 소멸자를 호출하면서, Child 소멸자가 '알아서' Parent 의 소멸자도 호출해주기 때문입니다 (Child 는 자신이 Parent 를 상속받는다는 것을 알고 있습니다).

반면에 Parent 소멸자를 먼저 호출하게 되면, Parent 는 Child 가 있는지 없는지 모르므로, Child 소멸자를 호출해줄 수 없습니다 (Parent 는 자신이 누구에서 상속해주는지 알 수 없지요).

이와 같은 연유로, 상속될 여지가 있는 Base 클래스들은 (위 경우 Parent) 반드시 소멸자를 `virtual` 로 만들어주어야 나중에 문제가 발생할 여지가 없게 됩니다.

레퍼런스도 된다

여태 까지 기반 클래스에서 파생 클래스의 함수에 접근할 때 항상 기반 클래스의 포인터를 통해서 접근하였습니다. 하지만, 사실 기반 클래스의 레퍼런스여도 문제 없이 작동합니다. 아래 간단한 예제를 통해 살펴보겠습니다.

```

#include <iostream>

class A {
public:
    virtual void show() { std::cout << "Parent !" << std::endl; }
};

class B : public A {
public:
    void show() override { std::cout << "Child!" << std::endl; }
};

void test(A& a) { a.show(); }

int main() {
    A a;
    B b;
    test(a);
}

```

```
    test(b);  
  
    return 0;  
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Parent !  
Child!
```

와 같이 나옵니다.

```
void test(A& a) { a.show(); }
```

test 함수를 살펴보면 A 클래스의 레퍼런스를 받게 되어 있지만,

```
test(b);
```

를 통해서 B 클래스의 객체를 전달하였는데도 잘 작동하였습니다. 이는, B 클래스가 A 클래스를 상속 받고 있기 때문입니다. 즉, 함수에 타입이 기반 클래스여도 그 파생 클래스는 타입 변환되어 전달 할 수 있습니다.

따라서 test 함수에서 show() 를 호출하였을 때 인자로 b 를 전달하였다면, 비록 전달된 인자가 A의 객체라고 표현되어 있지만 show 함수가 virtual 으로 정의되어 있기 때문에 알아서 B 의 show 함수를 찾아내서 호출하게 됩니다. 물론 test 에 a 를 전달하였을 때에는 A 의 show 함수가 호출되겠지요.

가상 함수의 구현 원리

여태 까지 virtual 키워드의 능력을 본 바로는 이러한 의문이 들 수 도 있을 것입니다.

그냥 그럼 모든 함수들을 virtual 로 만들어버리면 안되나?

사실 이는 매우 좋은 질문입니다. 왜냐하면 모든 함수들을 virtual 로 만들어버린다고 해서 문제될 것이 전혀 없기 때문입니다. 간혹 가상 이라는 이름 때문에 혼동하시는 분이 계시는데, virtual 키워드를 붙여서 가상 함수로 만들었다 해도 실제로 존재하는 함수이고 정상적으로 호출도 할 수

있습니다. 또한 모든 함수들을 디폴트로 가상 함수로 만듬으로써, 언제나 동적 바인딩이 제대로 동작하게 만들 수 있습니다.

실제로 자바의 경우 모든 함수들이 디폴트로 `virtual` 함수로 선언됩니다.

그렇다면 왜 C++에서는 `virtual` 키워드를 이용해 사용자가 직접 `virtual`로 선언하도록 하였을까요? 그 이유는 가상 함수를 사용하게 되면 약간의 오버헤드 (**overhead**)가 존재하기 때문입니다.³⁾ 이를 이해하기 위해 가상 함수라는 것이 어떻게 구현되는지, 다시 말해 마술과 같은 동적 바인딩이 어떻게 구현되는지 살펴보도록 합시다.

예를 들어서 다음과 같은 간단한 두 개의 클래스를 생각해봅시다.

```
class Parent {
public:
    virtual void func1();
    virtual void func2();
};

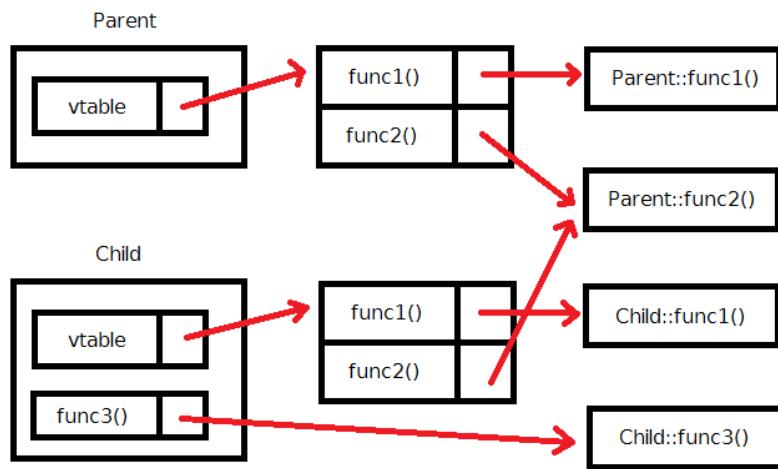
class Child : public Parent {
public:
    virtual void func1();
    void func3();
};
```

C++ 컴파일러는 가상 함수가 하나라도 존재하는 클래스에 대해서, 가상 함수 테이블(**virtual function table; vtable**)을 만들게 됩니다. 가상 함수 테이블은 전화 번호부라고 생각하시면 됩니다.

함수의 이름(전화번호부의 가게명)과 실제로 어떤 함수(그 가게의 전화번호)가 대응되는지 테이블로 저장하고 있는 것입니다

위 경우 `Parent` 와 `Child` 모두 가상 함수를 포함하고 있기 때문에 두 개 다 가상 함수 테이블을 생성하게 되지요. 그 결과;

3) 보통의 함수를 호출하는 것 보다 가상 함수를 호출하는 데 걸리는 시간이 조금 더 오래 걸립니다.



위와 같이 구성됩니다. 가상 함수와 가상 함수가 아닌 함수와의 차이점을 살펴보자면 **Child**의 **func3()** 같이 비 가상함수들은 그냥 단순히 특별한 단계를 걸치지 않고, **func3()** 을 호출하면 직접 실행됩니다.

하지만, 가상 함수를 호출하였을 때는 그 실행 과정이 다릅니다. 위에서도 보이다 싶이, 가상 함수 테이블을 한 단계 더 걸쳐서, 실제로 어떤 함수를 고를지 결정하게 됩니다. 예를 들어서;

```
Parent* p = Parent();
p->func1();
```

을 해봅시다. 그러면, 컴파일러는

1. p 가 **Parent** 를 가리키는 포인터 이니까, **func1()** 의 정의를 **Parent** 클래스에서 찾아봐야겠다.
2. **func1()** 이 가상함수네? 그렇다면 **func1()** 을 직접 실행하는게 아니라, 가상 함수 테이블에서 **func1()** 에 해당하는 함수를 실행해야겠다.

그리고 실제로 프로그램 실행시에, 가상 함수 테이블에서 **func1()** 에 해당하는 함수(**Parent::func1()**) 을 호출하게 됩니다.

에 해당하는 코드를 작성하게 됩니다. 그렇다면, 다음의 경우는 어떨까요?

```
Parent* c = Child();
c->func1();
```

위 처럼 똑같이 프로그램 실행시에 가상 함수 테이블에서 **func1()** 에 해당하는 함수를 호출하게 되는데, 이번에는 p 가 실제로는 **Child** 객체를 가리키고 있으므로, **Child** 객체의 가상 함수 테이

블을 참조하여, `Child::func1()` 을 호출하게 됩니다. 따라서 성공적으로 `Parent::func1()` 를 오버라이드 할 수 있습니다.

이와 같이 두 단계에 걸쳐서 함수를 호출함을 통해 소프트웨어적으로 동적 바인딩을 구현할 수 있게 됩니다. 이러한 이유로 가상 함수를 호출하는 경우, 일반적인 함수 보다 약간 더 시간이 오래 걸리게 됩니다.

물론 이 차이는 극히 미미하지만, 최적화가 매우 중요한 분야에서는 이를 감안할 필요가 있습니다. 아무튼 이러한 연유로 인해, 다른 언어들과는 다르게, C++ 에서는 멤버 함수가 디폴트로 가상함수가 되도록 설정하지는 않습니다.

순수 가상 함수(pure virtual function)와 추상 클래스 (abstract class)

```
#include <iostream>

class Animal {
public:
    Animal() {}
    virtual ~Animal() {}
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    Dog() : Animal() {}
    void speak() override { std::cout << "왈왈" << std::endl; }
};

class Cat : public Animal {
public:
    Cat() : Animal() {}
    void speak() override { std::cout << "야옹야옹" << std::endl; }
};

int main() {
    Animal* dog = new Dog();
    Animal* cat = new Cat();

    dog->speak();
    cat->speak();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
왈왈
야옹야옹
```

위 코드를 보면서 한 가지 특이한 점을 눈치 채셨을 것입니다.

```
class Animal {
public:
    Animal() {}
    virtual ~Animal() {}
    virtual void speak() = 0;
};
```

`Animal` 클래스의 `speak` 함수를 살펴봅시다. 다른 함수들과는 달리, 함수의 몸통이 정의되어 있지 않고 단순히 `= 0;` 으로 처리되어 있는 가상 함수입니다.

그렇다면 이 함수는 무엇을 하는 함수 일까요? 그 답은, "무엇을 하는지 정의되어 있지 않는 함수"입니다. 다시 말해 이 함수는 반드시 오버라이딩 되어야만 하는 함수 이지요.

이렇게, 가상 함수에 `= 0;` 을 붙여서, 반드시 오버라이딩 되도록 만든 함수를 완전한 가상 함수라 해서, 순수 가상 함수(pure virtual function)라고 부릅니다.

당연하게도, 순수 가상 함수는 본체가 없기 때문에, 이 함수를 호출하는 것은 불가능합니다. 그렇기 때문에, `Animal` 객체를 생성하는것 또한 불가능입니다. 왜냐하면,

```
Animal a;
a.speak();
```

하면 안되기 때문이지요. 물론, `speak()` 함수를 호출하는 것을 컴파일러 상에서 금지하면 되지 않느냐고 물을 수 있는데, C++ 개발자들은 이러한 방법 대신에 아예 `Animal` 의 객체 생성을 금지시키는 것으로 택하였습니다. (쉽게 말해 `Animal` 의 인스턴스를 생성할 수 없지요)

만일 `Animal` 의 객체를 생성하려고 한다면 다음과 같은 컴파일 오류를 만날 수 있습니다.

컴파일 오류

```
error C2259: 'Animal' : cannot instantiate abstract class
1>           due to following members:
1>           'void Animal::speak(void)' : is abstract
```

따라서 `Animal` 처럼, 순수 가상 함수를 최소 한 개 이상 포함하고 있는 클래스는 객체를 생성할 수 없으며, 인스턴스화 시키기 위해서는 이 클래스를 상속 받는 클래스를 만들어서 모든 순수 가상 함수를 오버라이딩 해주어야만 합니다.

이렇게 순수 가상 함수를 최소 한개 포함하고 있는- 반드시 상속 되어야 하는 클래스를 가리켜 **추상 클래스 (abstract class)**라고 부릅니다. (참고로, `private` 안에 순수 가상 함수를 정의하여도 문제 될 것이 없습니다. `private`에 정의되어 있다고 해서 오버라이드 안된다는 뜻이 아니기 때문이죠. 다만 자식 클래스에서 호출을 못할 뿐입니다.)

따라서;

```
class Dog : public Animal {
public:
    Dog() : Animal() {}
    void speak() override { std::cout << "왈왈" << std::endl; }
};
```

위처럼 `speak()`를 오버라이딩 함으로써 (정확히 말하면 `Animal`의 모든 순수 가상 함수를 오버라이딩 함으로써) `Dog` 클래스의 객체를 생성할 수 있게 됩니다. `Cat` 클래스도 마찬가지 이지요.

그렇다면 추상 클래스를 도대체 왜 사용하는 것일까요?

추상 클래스 자체로는 인스턴스화 시킬 수도 없고 (추상 클래스의 객체를 만들 수 없다) 사용하기 위해서는 반드시 다른 누구가 상속 해줘야만 하기 때문이지요. 하지만, 추상 클래스를 '설계도'라고 생각하면 좋습니다.

즉, 이 클래스를 상속받아서 사용하는 사람에게 "이 기능은 일반적인 상황에서 만들기 힘드니 너가 직접 특수화 되는 클래스에 맞추어서 만들어서 써라."라고 말해주는 것이지요. 예를 들어서 위에서 예를 든 `Animal` 클래스의 경우

```
class Animal {
public:
    Animal() {}
    virtual ~Animal() {}
    virtual void speak() = 0;
};
```

동물들이 소리를 내는 것은 맞으므로 `Animal` 클래스에 `speak` 함수가 필요합니다. 하지만 어떤 소리를 내는지는 동물마다 다르기 때문에 `speak` 함수를 가상 함수로 만들기는 불가능 합니다. 따라서 `speak` 함수를 순수 가상 함수로 만들게 되면 모든 `Animal` 들은 `speak()` 한다라는 의미 전달과 함께, 사용자가 `Animal` 클래스를 상속 받아서 (위 경우 `Dog` 와 `Cat`) `speak()` 를 상황에 맞게 구현하면 됩니다.

추상 클래스의 또 한가지 특징은 비록 객체는 생성할 수 없지만, 추상 클래스를 가리키는 포인터는 문제 없이 만들 수 있다는 점입니다. 위 예에서도 살펴보았듯이, 아무런 문제 없이 `Animal*` 의 변수를 생성하였습니다.

```
Animal* dog = new Dog();
Animal* cat = new Cat();

dog->speak();
cat->speak();
```

그리고 `dog` 와 `cat` 의 `speak` 함수를 호출하였는데, 앞에서도 배웠듯이, 비록 `dog` 와 `cat` 이 `Animal*` 타입 이지만, `Animal` 의 `speak` 함수가 오버라이드 되어서, `Dog` 와 `Cat` 클래스의 `speak` 함수로 대체되어서 실행이 됩니다.

다중 상속(multiple inheritance)

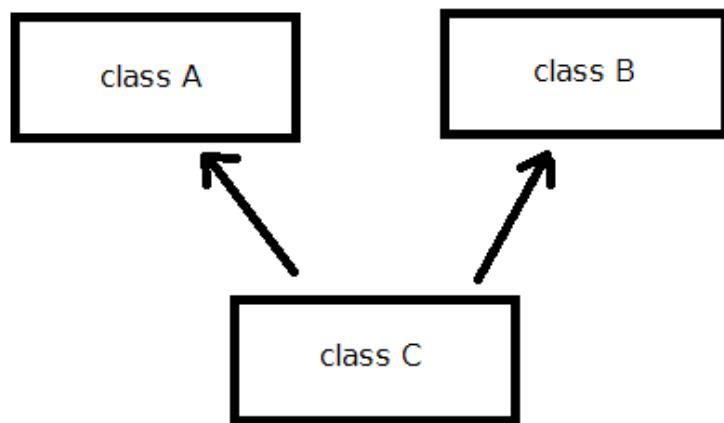
마지막으로 C++ 에서의 상속의 또 다른 특징인 다중 상속에 대해 알아보도록 합시다. C++ 에서는 한 클래스가 다른 여러 개의 클래스들을 상속 받는 것을 허용합니다. 이를 가리켜서 다중 상속 (multiple inheritance) 라고 부릅니다.

```
class A {
public:
    int a;
};

class B {
public:
    int b;
};

class C : public A, public B {
public:
    int c;
};
```

위 경우, 클래스 C 가 A 와 B 로 부터 동시에 같이 상속 받고 있습니다!



이를 그림으로 표현하자면 위 같은 모양이 되겠지요. 사실 다중 상속은 보통의 상속하고 똑같이 생각하시면 됩니다. 단순히 그냥 A 와 B 의 내용이 모두 C 에 들어간다고 생각하시면 됩니다. 따라서;

```

C c;
c.a = 3;
c.b = 2;
c.c = 4;
  
```

와 같은 것이 가능하게 되는 것이지요. 다중 상속에서 한 가지 재미있는 점은 생성자들의 호출 순서입니다. 여러분은 과연 위 예에서 A 의 생성자가 먼저 호출될지, B 의 생성자가 먼저 호출될지 궁금할 것입니다. 한 번 확인을 해보도록 하겠습니다.

```

#include <iostream>

class A {
public:
    int a;

    A() { std::cout << "A 생성자 호출" << std::endl; }
};

class B {
public:
    int b;

    B() { std::cout << "B 생성자 호출" << std::endl; }
};
  
```

```
class C : public A, public B {  
public:  
    int c;  
  
    C() : A(), B() { std::cout << "C 생성자 호출" << std::endl; }  
};  
int main() { C c; }
```

성공적으로 컴파일 하였다면

실행 결과

```
A 생성자 호출  
B 생성자 호출  
C 생성자 호출
```

위 처럼 A → B → C 순으로 호출됨을 알 수 있습니다. 그렇다면 이번에는,

```
class C : public A, public B
```

에서

```
class C : public B, public A
```

로 바꾸고 컴파일을 해보세요. 재미있게도;

실행 결과

```
B 생성자 호출  
A 생성자 호출  
C 생성자 호출
```

로 이번에는 B 의 생성자가 A 보다 먼저 호출됨을 알 수 있습니다. 몇 번 더 실험을 해보면 이 순서는 다른 것들에 의해 좌우되지 않고 오직 상속하는 순서에만 좌우 됨을 알 수 있습니다.

다중 상속 시 주의할 점

다중 상속은 C++ 에서 많이 사용되는 기법 중 하나입니다. 하지만, 다중 상속을 올바르게 사용하기 위해서는 몇 가지 주의해야 할 점들이 있습니다.

```

class A {
public:
    int a;
};

class B {
public:
    int a;
};

class C : public B, public A {
public:
    int c;
};

```

위처럼 만일 두 개의 클래스에서 이름이 같은 멤버 변수나 함수가 있다고 해봅시다. 예를 들어 위 예에서는 클래스 A 와 B 에 모두 a 라는 이름의 멤버 변수가 들어가 있습니다.

```

int main() {
    C c;
    c.a = 3;
}

```

그렇다면 만일 클래스 C 의 객체를 생성해서, 위처럼 중복되는 멤버 변수에 접근한다면;

컴파일 오류

```

error C2385: ambiguous access of 'a'
1>           could be the 'a' in base 'B'
1>           or could be the 'a' in base 'A'

```

위처럼 B 의 a 인지, A 의 a 인지 구분할 수 없다는 오류를 발생하게 됩니다. 마찬가지로, 클래스 A 와 B 에 같은 이름의 함수가 있다면 똑같이 어떤 함수를 호출해야 될지 구분할 수 없겠지요.

다중 상속 사용 시 또 한 가지 주의해야 할 점으로 다이아몬드 상속(diamond inheritance) 혹은 공포의 다이아몬드 상속(dreadful diamond of derivation) 이라고 부르는 형태의 다중 상속에 있습니다. 예를 들어 다음과 같은 형태의 상속 관계를 생각해봅시다.

```

class Human {
    // ...
};

class HandsomeHuman : public Human {
    // ...
};

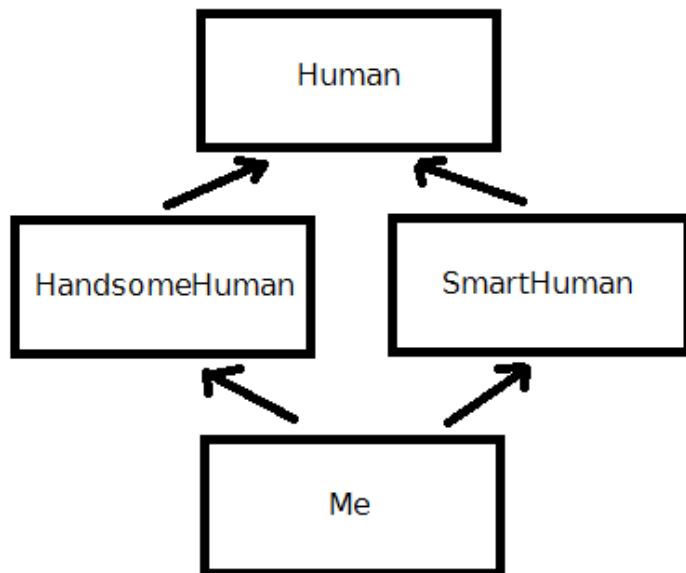
```

```
class SmartHuman : public Human {
    // ...
};

class Me : public HandsomeHuman, public SmartHuman {
    // ...
};
```

일단 베이스 클래스로 Human이라는 클래스가 있고, HandsomeHuman과 SmartHuman 클래스는 Human 클래스를 모두 상속 받습니다.

그리고 두 가지 특성을 모두 보유한 나(Me)라는 클래스는, HandsomeHuman과 SmartHuman 클래스를 둘 다 상속 받습니다. 이를 그림으로 표현하자면 아래와 같은 다이아몬드 모양이 나오게 됩니다.



상속이 되는 두 개의 클래스가 공통의 베이스 클래스를 포함하고 있는 형태를 가리켜서 다이아몬드 상속이라고 부릅니다. 이러한 형태의 상속에 문제점은 보기에도 명백합니다.

만일 Human에 name이라는 멤버 변수가 있다고 해봅시다. 그러면 HandsomeHuman과 SmartHuman은 모두 Human을 상속 받고 있으므로, 여기에도 name이라는 변수가 들어가게 됩니다.

그런데 Me가 이 두 개의 클래스를 상속 받으니 Me에서는 name이라는 변수가 겹치게 되는 것이지요. 결과적으로 볼 때 Handsome과 SmartHuman을 아무리 안겹치게 만든다고 해도, Human의 모든 내용이 중복되는 문제가 발생하게 됩니다.

다행이도 이를 해결할 수 있는 방법이 있습니다.

```
class Human {
public:
```

```
// ...
};

class HandsomeHuman : public virtual Human {
// ...
};

class SmartHuman : public virtual Human {
// ...
};

class Me : public HandsomeHuman, public SmartHuman {
// ...
};
```

이러한 형태로 Human 을 virtual 로 상속 받는다면, Me 에서 다중 상속 시에도, 컴파일러가 언제나 Human 을 한 번만 포함하도록 지정할 수 있게 됩니다. 참고로, 가상 상속 시에, Me 의 생성자에서 HandsomeHuman 과 SmartHuman 의 생성자를 호출함은 당연하고, Human 의 생성자 또한 호출해주어야만 합니다.

다중 상속은 언제 사용해야 할까?

C++ 공식 웹사이트의 FAQ 에 따르면 아래와 같은 가이드라인을 제시하고 있습니다.⁴⁾

예를 들어서 여러분이 차량(Vehicle) 에 관련한 클래스를 생성한다고 해봅시다. 차량의 종류로는 땅에서 다니는 차, 물에서 다니는 차, 하늘에서 다니는 차, 우주에서 다니는 차들이 있다고 해봅시다. (차 라고 하기 보다는 운송 수단이 좀 더 적절한 표현이겠네요..)

또한, 이 차량들은 각기 다른 동력원들을 사용하는데, 휘발유를 사용할 수 도 있고, 풍력으로 갈 수 도 있고 원자력으로 갈 수도 있고, 폐달을 밟아서 갈 수 도 있습니다.

이러한 차량들을 클래스로 나타내기 위해서, 다중 상속을 활용할 수 있지만 그 전에, 아래와 같은 질문들에 대한 대답을 생각해봅시다.

- LandVehicle 을 가리키는 Vehicle& 레퍼런스를 필요로 할까? 다시 말해, Vehicle 레퍼런스가 실제로는 LandVehicle 을 참조하고 있다면, Vehicle 의 멤버 함수를 호출하였을 때, LandVehicle 의 멤버 함수가 오버라이드 되어서 호출되기를 바라나요?
- GasPoweredVehicle 의 경우도 마찬가지 입니다. 만일 Vehicle 레퍼런스가 실제로는 GasPoweredVehicle 을 참조하고 있을 때, Vehicle 레퍼런스의 멤버함수를 호출한다면, GasPoweredVehicle 의 멤버 함수가 오버라이드 되어서 호출되기를 원하나요?

만일 두 개의 질문에 대한 대답이 모두 예 라면 다중 상속을 사용하는 것이 좋을 것입니다. 하지만 그 전에, 몇 가지 고려할 점이 더 있습니다. 만약에 이 차량이 작동하는 환경이 N 개가 있고 (땅, 물,

4) 물론 이 또한 절대적인 것은 아닙니다.

하늘, 우주 등등), 동력원의 종류가 M 개가 있다고 해봅시다.

이를 위해서, 크게 3 가지 방법으로 이러한 클래스를 디자인 할 수 있습니다. 바로 브리지 패턴 (bridge pattern), 중첩된 일반화 방식 (nested generalization), 다중 상속입니다. 각각의 방식에는 모두 장단점이 있습니다.

- 브리지 패턴의 경우 차량을 나타내는 한 가지 카테고리를 아예 멤버 포인터로 만들어버립니다. 예를 들어서 **Vehicle** 클래스의 파생 클래스로 **LandVehicle**, **SpaceVehicle** 클래스들이 있고, **Vehicle** 클래스의 멤버 변수로 어떤 엔진을 사용하는지 가리키는 **Engine*** 멤버 변수가 있습니다. 이 **Engine** 은 **GasPowered**, **NuclearPowered** 와 같은 **Engine** 의 파생 클래스들의 객체들을 가리키게 됩니다. 그리고 런타임 시에 사용자가 **Engine** 을 적절히 설정해주면 됩니다. 이 경우 동력원이나 환경을 하나 추가하더라도 클래스를 1 개만 더 만들면 됩니다. 즉, 총 $N + M$ 개의 클래스만 생성하면 된다는 뜻입니다.

하지만 오버라이딩 가지수가 $N + M$ 개 뿐이므로 최대 $N + M$ 개 알고리즘 밖에 사용할 수 없습니다. 만일 여러분이 $N \times M$ 개의 모든 상황에 대한 섬세한 제어가 필요하다면 브리지 패턴을 사용하지 않는 것이 좋습니다. 또한, 컴파일 타임 타입 체크를 적절히 활용할 수 없다는 문제가 있습니다. 예를 들어서 **Engine** 이 폐달이고 작동 환경이 우주라면, 애초에 해당 객체를 생성할 수 없어야 하지만 이를 컴파일 타임에서 강제할 방법이 없고 런타임에서나 확인할 수 있게 됩니다. 뿐만 아니라, 우주에서 작동하는 모든 차량을 가리킬 수 있는 기반 클래스를 만들 수 있지만 (**SpaceVehicle** 클래스), 작동 환경에 관계 없이 휘발유를 사용하는 모든 차량을 가리킬 수 있는 기반 클래스를 만들 수는 없습니다.

- 중첩된 일반화 방식을 사용하게 된다면, 한 가지 계층을 먼저 골라서 파생 클래스들을 생성합니다. 예를 들어서 **Vehicle** 클래스의 파생 클래스들로 **LandVehicle**, **WaterVehicle**, 등등이 있겠지요. 그 후에, 각각의 클래스들의 대해 다른 계층에 해당하는 파생 클래스들을 더 생성합니다. 예컨대 **LandVehicle** 의 경우 동력원으로 휘발유를 사용한다면 **GasPoweredLandVehicle**, 원자력을 사용한다면 **NuclearPoweredLandVehicle** 클래스를 생성할 수 있겠지요.

따라서 최대 $N \times M$ 가지의 파생 클래스들을 생성할 수 있게 됩니다. 따라서 브릿지 패턴에 비해서 좀 더 섬세한 제어를 할 수 있게 됩니다. 왜냐하면 오버라이딩 가지수가 $N + M$ 이 아닌 $N \times M$ 이 되기 때문이지요. 하지만 동력원을 하나 더 추가하게 된다면 최대 N 개의 파생 클래스를 더 만들어야 합니다. 뿐만 아니라 앞서 브릿지 패턴에서 나왔던 문제 - 휘발유를 사용하는 모든 차량을 가리킬 수 있는 기반 클래스를 만들 수 없다가 여전히 있습니다. 따라서 만약에 휘발유를 사용하는 차량들에서 공통적으로 사용되는 코드가 있다면 매 번 새로 작성해줘야만 합니다.

- 다중 상속을 이용하게 된다면, 브리지 패턴처럼 각 카테고리에 해당하는 파생 클래스들을 만들게 되지만, 그 대신 **Engine*** 멤버 변수를 없애고 동력원과 환경에 해당하는 클래스를 상속받는 파생 클래스들을 최대 $N \times M$ 개 만들게 됩니다. 예를 들어서 휘발유를 사용하며 지상에서 다니는 차량을 나타내는 **GasPoweredLandVehicle** 클래스의 경우 **GasPoweredEngine**

과 LandVehicle 두 개의 클래스를 상속받겠지요.

따라서 이 방식을 통해서 브리지 패턴에서 불가능 하였던 섬세한 제어를 수행할 수 있을 뿐더러, 말도 안되는 조합을 (예컨대 PedalPoweredSpaceVehicle) 컴파일 타입에서 확인할 수 있습니다 (애초에 정의 자체를 안하면 되니까요!). 또한 이전에 두 방식에서 발생하였던 휘발유를 사용하는 모든 차량을 가리킬 수 없다 문제를 해결할 수 있습니다. 왜냐하면 이제 GasPoweredEngine 을 통해서 휘발유를 사용하는 모든 차량을 가리킬 수 있기 때문이지요.

가장 중요한 점은, 위 3 가지 방식 중에서 절대적으로 우월한 방식은 없다는 것입니다. 상황에 맞게 최선의 방식을 골라서 사용해야 합니다.

다중 상속은 만능 툴이 아닙니다. 실제로 다중 상속을 이용해서 해결해야 될 것 같은 문제도 알고보면 단일 상속을 통해 해결할 수 있는 경우들이 있습니다. 하지만 적절한 상황에 다중 상속을 이용한다면 위력적인 도구가 될 수 있을 것입니다.

아무래도 이번 강좌는 상속에 대한 중요한 요소들을 간단하게 짚고 넘어가는 것이라 실질적인 프로그램은 만들지 않았습니다. 하지만, 가상 함수와 상속이 어떻게 돌아가는지 완벽히 이해하는 것이 좋습니다. 저의 경우, C++ 처음 배울 때, 이 부분에서 많이 혓갈려서 고생을 한 기억이 있습니다. 여러분들도 가상 함수를 포함하는 간단한 프로그램을 작성해서 어떻게 함수들이 호출되는지 살펴보시기 바랍니다.

C++ 표준 입출력 라이브러리

안녕하세요! 여러분. 정말 오래간만에 강좌를 올리는 것 같습니다. 그동안 제가 여러가지 하는 일이 매우 많았는데, 물론 아직도 강좌 쓸 시간은 거의 없지만 없는 시간을 짜내며 좋은 강좌를 쓰기 위해서 노력 중입니다. 아무튼, 제가 자주 댓글도 못 달아 드리고 업데이트도 엄청 느리게 하지만 (결코 죽은 것이 아닙니다!!), 언제나 제 블로그를 방문해주셔서 강의를 보시는 분들에게 감사의 말을 전하고 싶습니다.

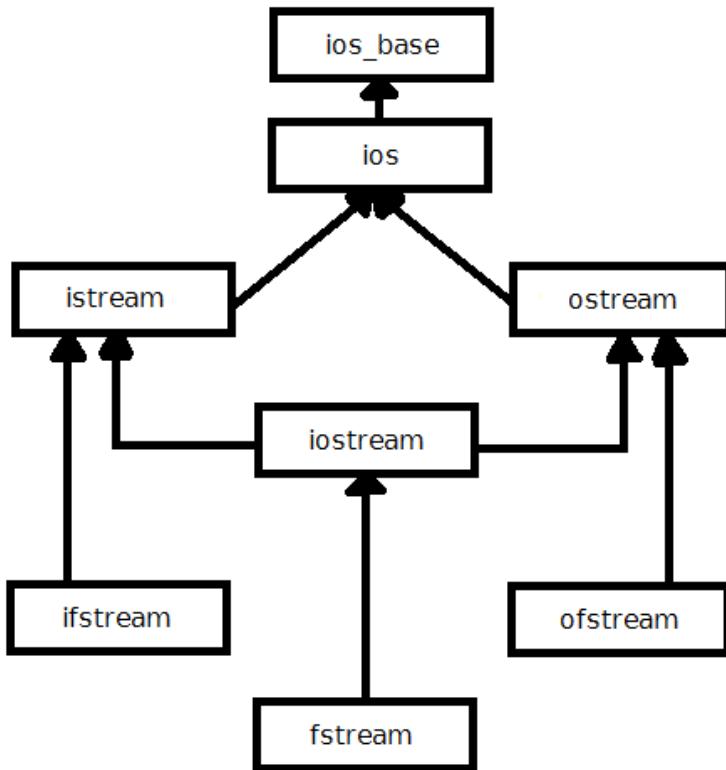
쉽진 않을 것 같지만 적어도 2016년 안에는 완결하는게 목표입니다.

이번 강좌에서는 여태 까지 우리가 크게 관심을 가지지 않았던 C++ 의 입출력 라이브러리에 대해서 알아보도록 하겠습니다. 맨날 `cout` 과 `cin` 을 쓰면서도, 정작 `cout` 과 `cin` 이 무엇인지는 한 번도 관심을 가지지 않았지요.

C++ 을 공부한 사람이라면 입출력 라이브러리를 한 번 쯤은 사용해 보았겠지만, 사실 정확히 어떻게 돌아가는지 이해하는 사람들은 드뭅니다. 그래서 이번 강좌를 시작으로 아마 3 개 강의에 걸쳐서 C++ 입출력 라이브러리에 대해서 자세히 알아보는 시간을 갖도록 할 것입니다.

C++ 입출력 라이브러리

C++ 의 입출력 라이브러리는 다음과 같은 클래스 들로 구성되어 있습니다 C++ 의 입출력 라이브러리는 다음과 같은 클래스 들로 구성되어 있습니다.



C++ 의 모든 입출력 클래스는 `ios_base` 를 기반 클래스로 하게 됩니다. `ios_base` 클래스는 많은 일은 하지 않고, 스트림의 입출력 형식 관련 데이터를 처리 합니다. 예를 들어서 실수 형을 출력할 때 정밀도를 어떤 식으로 할 것인지에 대해, 혹은 정수형을 출력 시에 10진수로 할지 16진수로 할지 등을 이 클래스에서 처리 합니다 C++ 의 모든 입출력 클래스는 `ios_base` 를 기반 클래스로 하게 됩니다. `ios_base` 클래스는 많은 일은 하지 않고, 스트림의 입출력 형식 관련 데이터를 처리 합니다. 예를 들어서 실수 형을 출력할 때 정밀도를 어떤 식으로 할 것인지에 대해, 혹은 정수형을 출력 시에 10진수로 할지 16진수로 할지 등을 이 클래스에서 처리 합니다.

그 다음으로 `ios` 클래스가 있습니다. 이 클래스에서는 실제로 스트림 버퍼를 초기화 합니다. 스트림 버퍼란, 데이터를 내보내거나 받아들이기 전에 임시로 저장하는 곳이라 볼 수 있습니다. 쉽게 설명하자면, 예를 들어서 우리가 하드디스크에서 파일을 하나 읽는다고 해봅시다. 만일 사용자가, 1 바이트 씩 읽는다고 했을 때, 실제로 프로그램은 1 byte 씩 읽는 것이 아닙니다.

실제로는 한 뭉터기 (예를 들어서 512 바이트) 를 한꺼번에 읽어서 스트림 버퍼에 잠시 저장해 놓은 뒤에 사용자가 요청할 때마다 1 바이트 씩 꺼내는 것이지요. 만일 버퍼를 다 읽는다면 다시 하드에서 512 바이트를 읽게 되는 것입니다. 이렇게 수행하는 이유는, 하드디스크에서 읽어오는 작업이 매우 느리기 때문에, 한 번 읽을 때 1 바이트 읽으면 엄청난 딜레이가 발생하게 됩니다. 이는 쓰는 작업에서도 마찬가지입니다. 쓸 때도 우리가 1 문자를 출력하게 되면, 하드에 바로 쓰는 것이 아니라 일단 버퍼에 보관한 후, 어느 정도 모인 뒤에 출력하게 됩니다.

`ios` 클래스에선 그 외에도, 현재 입출력 작업의 상태를 처리 합니다. 예를 들어, 파일을 읽다가 끝에 도달했는지 안했는지 확인하려면, `eof` 함수를 호출하면 됩니다. 또, 현재 입출력 작업을 잘 수행할

수 있는지 확인하려면 `good` 함수를 호출하면 됩니다.

istream 클래스

여태까지 `ios_base` 와 `ios` 클래스들이 입출력 작업을 위해 바탕을 깔아주는 클래스 였다면, `istream`은 실제로 입력을 수행하는 클래스입니다.

대표적으로 우리가 항상 사용하던 `operator>>` 가 이 `istream` 클래스에 정의되어 있는 연산자입니다. 또, `cin`은 `istream` 클래스의 객체 중 하나입니다. 그렇기 때문에 우리는

```
std::cin >> a;
```

와 같은 작업을 할 수 있었던 것이지요. 우리가, 어떤 타입에 대해서도 `cin`을 사용할 수 있었던 이유는 (`a`가 `char`이거나 `int`이거나 상관없이) 바로 `operator>>`가 그런 모든 기본 타입들에 대해서는 정의가 되어있기 때문입니다.

```
istream& operator>>(bool& val);

istream& operator>>(short& val);

istream& operator>>(unsigned short& val);

istream& operator>>(int& val);

istream& operator>>(unsigned int& val);

istream& operator>>(long& val);

istream& operator>>(unsigned long& val);

istream& operator>>(long long& val);

istream& operator>>(unsigned long long& val);

istream& operator>>(float& val);

istream& operator>>(double& val);

istream& operator>>(long double& val);

istream& operator>>(void*& val);
```

그렇다고 해서, 우리가 언제나 위 타입들 빼고는 `operator>>`로 받을 수 없는 것이 아닙니다. 실제로 `istream` 클래스의 멤버 함수로는 없지만;

```
std::string s;  
std::cin >> s;
```

`std::string` 클래스의 객체 `s` 도 `cin` 으로 입력 받을 수 있습니다. 이와 같은 일이 가능한 이유는 [이전에 연산자 오버로딩 강좌에서 배웠듯이](#), 멤버 함수를 두는 것 말고도, 외부 함수로 연산자 오버로딩을 할 수 있기 때문입니다.

이 경우에는

```
istream& operator>>(istream& in, std::string& s)  
{  
    // 구현한다  
}
```

와 같이 하면 되겠습니다.

`operator>>` 의 또 다른 특징으로는, 모든 공백문자 (띄어쓰기나 엔터, 탭 등)을 입력시에 무시해 버린다는 점입니다. 그렇기 때문에, 만일 `cin` 을 통해서 문장을 입력 받는다면, 첫 단어만 입력 받고 나머지를 읽을 수 없습니다. 예제로 간단히 살펴보자면

```
#include <iostream>  
#include <string>  
  
int main() {  
    std::string s;  
    while (true) {  
        std::cin >> s;  
        std::cout << "word : " << s << std::endl;  
    }  
}
```

성공적으로 컴파일 하였다면

실행 결과

```
this is a long sentence  
word : this  
word : is  
word : a  
word : long  
word : sentence
```

```
ABCD EFGH IJKL
word : ABCD
word : EFGH
word : IJKL
```

와 같이 문장을 입력하더라도, 공백문자에 따라서 각각을 분리해서 입력받게 되는 것입니다. 위와 같이 비록 `operator>>` 가 매우 편리해보이지만, 반드시 주의해야 할 점이 있는 점이 있습니다.

```
// 주의할 점
#include <iostream>
using namespace std;
int main() {
    int t;
    while (true) {
        std::cin >> t;
        std::cout << "입력 :: " << t << std::endl;
        if (t == 0) break;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
3
입력 :: 3
4
입력 :: 4
5
입력 :: 5
6
입력 :: 6
7
입력 :: 7
```

그냥 평범하게 숫자를 잘 입력 받는 프로그램입니다. 만일 사용자가 숫자만 꼬박 꼬박 잘 입력하면 정말 좋겠지만 문제는 그렇지 않는다는 것입니다. 프로그래머는 언제나 사용자의 기괴한 행동들에 대해서 모두 대응할 수 있어야만 합니다. 만일 사용자가, 숫자가 아니라 문자를 입력했다고 합시다. 그렇다면;

The screenshot shows a command-line interface window titled 'cmd.exe' with the path 'C:\Windows\system32\cmd.exe'. Inside the window, there is a single line of text that repeats the character sequence ': -858993460' multiple times. This visual representation corresponds to the infinite loop of the code shown in the previous section.

위와 같이 기괴한 결과를 보여줄 수 알 수 있습니다. (참고로 저는 단순히 'c' 하나만 쳤을 뿐입니다) 왜 이런 무한 루프에 빠지는 것일까요. 그 이유는 `operator>>` 가 어떻게 이를 처리하는지 이해하면 알 수 있습니다.

앞서 `ios` 클래스에서 스트림의 상태를 관리한다고 하였습니다. 이 때, 스트림의 상태를 관리하는 플래그 (flag - 그냥 비트 1 개라 생각하면 됩니다) 는 4 개가 정의되어 있습니다. 이 4 개의 플래그들이 스트림이 현재 어떠한 상태인지에 대해서 정보를 보관한다는 뜻입니다.

이 4 개의 플래그는 각각 `goodbit`, `badbit`, `eofbit`, `failbit` 이렇게 4 개 종류가 있습니다. 각각의 비트들이 켜져있는지, 꺼져있는지 (즉 1인지 0인지에 따라) 우리는 스트림의 상태를 알 수 있게 됩니다.

각각의 비트들에 대해 간단히 설명해보자면

- `goodbit` : 스트림에 입출력 작업이 가능할 때
- `badbit` : 스트림에 복구 불가능한 오류 발생시
- `failbit` : 스트림에 복구 가능한 오류 발생시
- `eofbit` : 입력 작업시에 EOF 도달시

위와 같은 상황 일 때 각각의 비트들이 켜지는 것입니다. 만일 위와 같이 문자를 입력할 경우 `operator>>` 가 어떤 비트를 켜게 될까요? 일단 `eofbit` 는 확실히 아닙니다. 끝에 도달한 것이 아니니까요.

그렇다면 `badbit` 일까요? `badbit` 는 스트림 상에서 복구할 수 없는 문제시 켜지는데 위 경우는 그렇게 심각한 것은 아닙니다. 그냥 현재 스트림 버퍼에 들어가 있는 '`c\n`' 이 문자열을 제거해버리면 되기 때문이지요.

위와 같이 타입에 맞지 않는 값을 넣어서 오류가 발생하는 경우에는 `failbit` 가 켜지게 됩니다. 그리고, 입력값을 받지 않고 리턴해버립니다.

문제는 이렇게 그냥 리턴해버리면서 버퍼에 남아 있는 '`c\n`' 이 문자열에는 손대지 않는다는 것입니다. 그렇기 때문에 다음에 또 읽고, 또 읽고, ... 결국 위와 같은 문제를 일으키게 됩니다.

```
// 해결 방안
#include <iostream>
#include <string>

int main() {
    int t;
    while (std::cin >> t) {
        std::cout << "입력 :: " << t << std::endl;
        if (t == 0) break;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
4
입력 :: 4
3
입력 :: 3
s
```

위와 같이 무한 루프에 빠지지 않고 제대로 처리됨을 알 수 있습니다. 어떻게 가능한 것일까요? 일단, `while` 문의 조건에 들어가 있는 저 식의 의미 부터 이해를 해봅시다.

```
while (std::cin >> t) {
```

위 식을 보기에 앞서, `ios` 에 정의되어 있는 함수들을 살펴보자면 다음과 같은 함수가 있음을 알 수 있습니다.

```
operator void*() const;
```

이 함수는 `ios` 객체를 `void*`로 변환해줍니다. 이 때, `NULL` 포인터가 아닌 값을 리턴하는 조건이, `failbit` 와 `badbit` 가 모두 `off` 일 때입니다. 즉, 스트림에 정상적으로 입출력 작업을 수행 할 수 있을 때 말입니다.

그럼 다시 `while` 문을 살펴보자면, 만일 우리가 '`s`' 를 입력한다면 `operator>>` 는 `cin` 객체의 `failbit` 를 켜게 됩니다. 그리고, `std::cin >> t` 후에 `cin` 이 리턴되는데 (`operator>>` 는 호출한 자신을 리턴!), 리턴값이 `while` 문의 조건식으로 들어가기 때문에 컴파일러는 적절한 타입 변환을 찾게 되고, 결국 `ios` 객체 \rightarrow `void*` \rightarrow `bool` 로 가는 2단 변환을 통해서 `while` 문을 잘 빠져나오게 됩니다. (※ `NULL` 포인터는 `bool` 상 `false` 입니다)

위와 같이 문제를 해결할 수 있었지만, 입력을 계속 진행 할 수는 없습니다. 왜냐하면 현재 `cin` 에 `fail` 비트가 켜진 상태이므로, 플래그를 초기화해버리지 않는 한 `cin` 을 이용하여 입력 받을 수 없게 됩니다.

```
#include <iostream>
#include <string>

int main() {
    int t;
    std::cin >> t; // 고의적으로 문자를 입력하면 failbit 가 켜진다
    std::cout << "fail 비트가 켜진 상태이므로, 입력받지 않는다" << std::endl;
    std::string s;
    std::cin >> s;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
s
fail 비트가 켜진 상태이므로, 입력받지 않는다
```

그렇다면 이 문제를 어떻게 해결 할 수 있을까요?

```
#include <iostream>
#include <string>

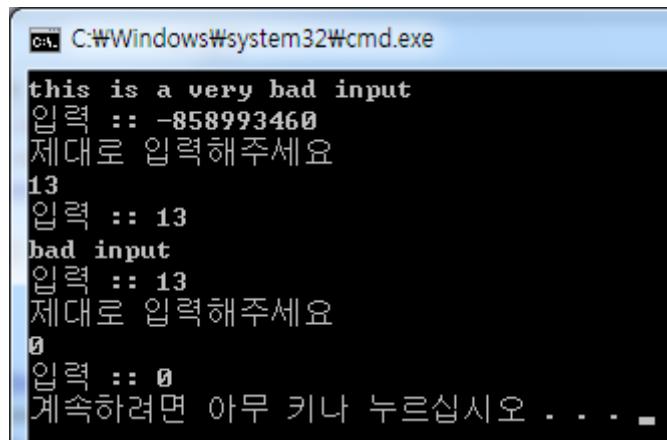
int main() {
    int t;
    while (true) {
        std::cin >> t;
        std::cout << "입력 :: " << t << std::endl;
        if (std::cin.fail()) {
            std::cout << "제대로 입력해주세요" << std::endl;
            std::cin.clear(); // 플래그들을 초기화 하고
```

```

        std::cin.ignore(100, '\n'); // 개행문자가 나올 때 까지 무시한다
    }
    if (t == 0) break;
}
}

```

성공적으로 컴파일 하였다면



위와 같이 잘 처리됩니다. 위 과정이 어떻게 가능한지 자세히 살펴보도록 합시다.

```
if (std::cin.fail()) {
```

먼저 `fail` 함수는 `ios`에 정의되어 있으며, `failbit`가 `true`거나 `badbit`가 `true`면 `true`를 리턴합니다. 만일 숫자가 아닌 것을 입력한다면 `failbit`가 `true` 이므로, `std::cin.fail()`은 `true`가 되어 조건문을 실행하게 됩니다.

```

std::cin.clear(); // 플래그들을 초기화 하고
std::cin.ignore(100, '\n'); // 버퍼를 비워버린다

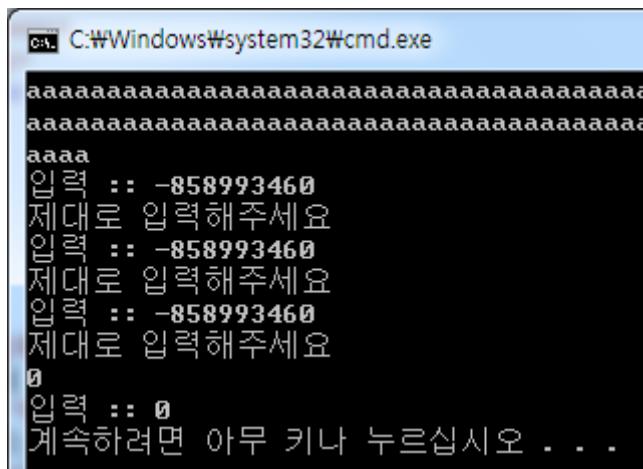
```

그리고 `clear()` 역시 `ios`에 정의되어 있으며, 인자를 주지 않을 경우 플래그를 `goodbit`으로 초기화 시켜 버립니다. 따라서 `fail` 상태를 지울 수 있게 되지요. 그 다음에 `ignore` 함수는 `istream`에 정의되어 있는데, 최대 첫번째 인자 만큼 (100), 두 번째 인자가 나올 때 까지 ('\n'), 버퍼에서 무시합니다 (두 번째 인자를 포함).

따라서, 만일 제가 *this is a very bad input* 을 입력하였다면 버퍼에는

this is a very bad input\n

이렇게 들어가 있고, `ignore` 함수를 통해 (최대 100 자 까지) 개행문자가 나올 때 까지 무시할 수 있게 됩니다.



만일 버퍼에 100자 이상을 집어 넣는다면 위와 같이 `ignore` 함수가 총 3번 호출됨을 알 수 있습니다.
(버퍼에 남아 있는 문자들이 다 지워질때 까지)

형식 플래그(format flag) 와 조작자 (Manipulator)

앞서 `ios_base` 클래스에서, 스트림의 입출력 형식을 바꿀 수 있다고 하였습니다. 예를 들어서, 여태까지 수를 입력하면 10진수로 처리되었지만, 이번에는 16진수로 처리할 수 있는 법입니다. 이를 어떻게 가능하게 하는지 아래의 예제로 보여드리겠습니다.

```
#include <iostream>
#include <string>

int main() {
    int t;
    while (true) {
        std::cin.setf(std::ios_base::hex, std::ios_base::basefield);
        std::cin >> t;
        std::cout << "입력 :: " << t << std::endl;
        if (std::cin.fail()) {
            std::cout << "제대로 입력해주세요" << std::endl;
            std::cin.clear(); // 플래그들을 초기화 하고
                // std::cin.ignore(100, '\n'); // 개행문자가 나올 때까지
                // 무시한다
        }
        if (t == 0) break;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
ff
입력 :: 255
0xFF
입력 :: 255
123
입력 :: 291
ABCDE
입력 :: 703710
```

위와 같이 16진수 입력을 잘 받는다는 것을 볼 수 있습니다. (출력 형식은 바꾸지 않았으므로, 10진수로 출력됩니다) 이처럼 입력 받는 형식을 16진수로 바꿔준 함수는 보시다 싶이, 아래와 같은 스트림의 설정을 바꾸는 `setf` 함수 덕분입니다.

```
std::cin.setf(ios_base::hex, ios_base::basefield);
```

`setf` 함수의 버전은 2 개가 있는데, 하나는 인자를 1 개만 받는 것이고, 다른 하나는 위처럼 인자를 2개 취하는 것입니다. 인자 1 개를 받는 `setf` 는 그냥 인자로 준 형식 플래그를 적용하는 것이지만, 2 개 취하는 것은, 두 번째 인자 (위에서 `basefield`) 의 내용을 초기화하고, 첫 번째 인자 (`hex`) 를 적용하는 것입니다.

위 경우, 수를 처리하는 방식은 1 가지 진수만 한 번에 처리할 수 있으므로, 몇 진법으로 수를 처리할지 보관하는 `basefield` 의 값을 초기화하고, 16진법 (`hex`) 플래그를 적용시킨 것입니다.

물론, 여러분이 16 진법을 처리하는 함수를 그냥 만들어도 됩니다. 수 대신에 문자열로 입력받아서 처리해도 되지요. 그렇지만, 사용자가 `0x` 를 앞에 붙일 수도 있고 안 붙일 수도 있고, `a123` 이라 쓸 수도 있고 `A123` 이라 쓸 수도 있고 등 여러가지 경우가 있기 때문에 차라리 마음 편하게 IO 라이브러리에서 지원하는 방식을 사용하는 것이 좋은 것 같습니다.

그런데 흥미롭게도, 비슷하지만 또 다른 방식으로 16진수를 받을 수 있습니다.

```
// 조작자의 사용
#include <iostream>
#include <string>

int main() {
    int t;
    while (true) {
        std::cin >> std::hex >> t;
        std::cout << "입력 :: " << t << std::endl;
        if (std::cin.fail()) {
            std::cout << "제대로 입력해주세요" << std::endl;
        }
    }
}
```

```

        std::cin.clear();           // 플래그들을 초기화 하고
        std::cin.ignore(100, '\n'); // 개행문자가 나올 때까지 무시한다
    }
    if (t == 0) break;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

ff
입력 :: 255
0xFF
입력 :: 255
123
입력 :: 291
ABCDE
입력 :: 703710

```

위 경우 역시 16진수를 잘 입력받는다는 사실을 알 수 있습니다. 이게 어떻게 된 일 일까요?

```
std::cin >> hex >> t;
```

바로 위에서 `hex` 가 `cin`에서 수를 받는 방식을 바꿔버렸기 때문입니다. 이 때문에 `hex` 와 같이, 스트림을 조작하여 입력 혹은 출력 방식을 바꿔주는 함수를 조작자라고 부릅니다 (그렇습니다! `hex` 는 함수입니다). 참고로, 위에서 사용하였던 형식 플래그 `hex` 와 이 `hex` 는 이름만 같지 아예 다른 것입니다. (그렇기에, 위에서는 `ios_base::hex` 로 사용하였죠)

위의 형식 플래그 `hex` 는 `ios_base`에 선언되어 있는 단순한 상수 '값'입니다. 반면에 조작자 `hex` 의 경우 `ios`에 정의되어 있는 '함수'입니다. 이 조작자 `hex`의 정의를 살펴보자면, 아래와 같이 `ios_base` 객체를 레퍼런스로 받고, 다시 그 객체를 리턴하도록 정의 되어 있습니다.

```
std::ios_base& hex(std::ios_base& str);
```

그렇다면, `operator>>` 중에서 위 함수를 인자로 가지는 경우도 있을까요? 물론 있습니다.

```
istream& operator>>(ios_base& (*pf)(ios_base&));
```

이렇게, `operator>>`에서 조작자를 받는다면 많은 일을 하는 것이 아니라 단순히 `pf(*this)`를 호출하게 됩니다. 호출된 `hex` 함수가 하는 일 또한 별로 없습니다. 단순히,

```
str.setf(std::ios_base::hex, std::ios_base::basefield)
```

를 수행해주는 것이지요.

이렇게, `setf`를 사용하지 않더라도, 간단하게 조작자를 사용하면 훨씬 쉽게 입력 형식을 바꿀 수 있게 됩니다. 조작자들의 종류는 위에서 설명한 `hex` 말고도, 꽤 많은데, `true`나 `false`를 1과 0으로 처리하는 대신 문자열 그대로 입력 받는 `boolalpha`도 있고, 출력 형식으로 왼쪽 혹은 오른쪽으로 정렬 시키는 `left`와 `right` 조작자 등 여러가지가 있습니다.

그 외에도 우리가 여태까지 아무 생각없이 사용하였던 `std::endl`도 있습니다. `endl`은 `hex`와는 달리 출력을 관장하는 `ostream`에 정의되어 있는 조작자로, 한 줄 개행문자를 출력하는 것 말고도, 버퍼를 모두 내보내는(`flush`) 역할도 수행합니다.

앞서 말했듯이, 문자 1개를 내보낸다고 해서 화면에 바로 출력되는 것이 아니라, 버퍼에 모은 다음에 버퍼에 어느 정도 쌓이면 비로소 출력하게 됩니다. 이렇게 한다고 해서 대부분의 경우 문제되지는 않습니다. 하지만 예를 들어 정해진 시간에 딱딱 맞춰서 화면에 출력해야 한다면 어떨까요? 이 경우 버퍼에 저장할 필요없이 화면에 바로 내보내야 할 것입니다.

이럴 경우를 위해서, 버퍼에 데이터가 얼마나 쌓여있든지 간에 바로 출력을 해주는 `flush` 함수가 있습니다. 따라서, `std::endl` 조작자는, 스트림에 '\n'을 출력하는 것과 더불어 `flush`를 수행해준다는 사실을 알 수 있습니다.

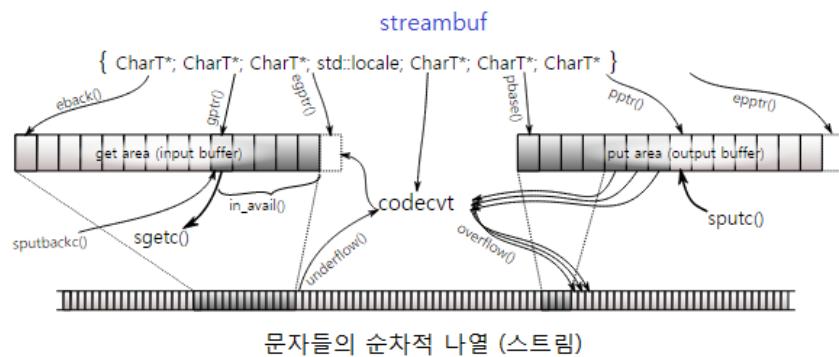
스트림 버퍼에 대해

모든 입출력 객체들은 이에 대응되는 스트림 객체를 가지고 있게 됩니다. 따라서 C++의 입출력 라이브러리에는 이에 대응되는 스트림 버퍼 클래스도 있는데, 이름이 `streambuf` 클래스입니다. 사실, 스트림이라 하면 그냥 쉽게 말해서 문자들의 순차적인 나열이라 보시면 됩니다. 그냥 문자들이 순차적으로 쭈르륵 들어오는 것이 (마치 냇가에서 물이 졸졸 흐르듯이 `stream` 단어의 사전적 의미는 시냇물입니다) 스트림이라 생각하시면 됩니다.

예를 들어서, 우리가 화면에 입력하는 문자도 스트림을 통해서 프로그램에 전달되는 것이고, 하드디스크에서 파일을 읽는 것도, 다른 컴퓨터와 TCP/IP 통신하는 것도 (결국 문자들을 쭈루륵 주고받는 것이니까), 모두 스트림을 통해 이루어진다는 것입니다.

심지어 C++에서는 `std::stringstream`을 통해서 평범한 문자열을 마치 스트림인 것처럼 이용할 수 있게 해줍니다.

`streambuf` 클래스는 스트림에 대한 가장 기본적인 책임을 담당하고 있습니다.!



위 사진은 `streambuf` 클래스에서 스트림을 어떤 식으로 추상화하고 있는지 보여주는 그림입니다. `streambuf` 는 그림과 같이 맨 아래에 나타나있는 스트림에서 입력을 받던지, 출력을 하던지, 혹은 입력과 출력을 동시에 (파일 입출력에서 "rw" 옵션을 생각해보세요) 수행할 수 도 있습니다.

`streambuf` 클래스는 스트림의 상태를 나타내기 위해서 세 개의 포인터를 정의하고 있습니다. 먼저 버퍼의 시작 부분을 가리키는 시작 포인터와, 다음으로 읽을 문자를 가리키고 있는 포인터 (우리가 흔히 말하는 스트림 위치 지정자), 그리고 버퍼의 끝 부분을 가리키고 있는 끝 포인터가 있습니다. `streambuf` 클래스는 입력 버퍼와 출력 버퍼를 구분해서 각각 `get area` 와 `put area` 라 부르는데, 이에 따라 각각을 가리키는 포인터도 `g` 와 `p` 를 붙여서 표현하게 됩니다.

아래 예제를 통해 `streambuf` 를 어떻게 하면 간단히 조작할 수 있는지 보여드리도록 하겠습니다.

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::cin >> s;

    // 위치 지정자를 한 칸 옮기고, 그 다음 문자를 훔쳐본다 (이 때는 움직이지 않음)
    char peek = std::cin.rdbuf()->snextc();
    if (std::cin.fail()) std::cout << "Failed";
    std::cout << "두 번째 단어 맨 앞글자 : " << peek << std::endl;
    std::cin >> s;
    std::cout << "다시 읽으면 : " << s << std::endl;
}
```

성공적으로 컴파일 하였다면

```
hello world 두 번째 단어 맨 앞글자 : w 다시 읽으면 : world
```

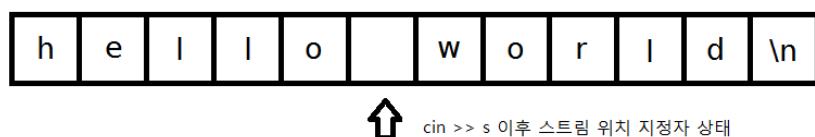
위와 같이 나옴을 알 수 있습니다.

```
char peek = std::cin.rdbuf()->snextc();
```

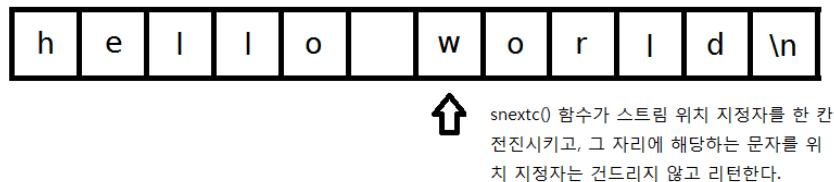
입력 객체 `cin`의 `rdbuf`를 호출하게 되면, `cin` 객체가 입력을 수행하고 있던 `streambuf` 객체를 가리키는 포인터를 리턴하게 됩니다. 이 때, `cin` 객체가 `istream` 객체 이므로, 오직 입력만을 수행하고 있기에, 이 `streambuf` 객체에는 오직 `get area` 만 있음을 알 수 있습니다.

`snextc` 함수는, 스트림 위치 지정자를 한 칸 전진시킨 후, 그 자리에 해당하는 문자를 엿봅니다 (읽는 것이 아닙니다). 엿보는 것과 읽는 것의 차이점은, 보통 `streambuf` 객체에서 읽게 되면, 스트림 위치 지정자를 한 칸 전진시켜서 다음 읽기 때 다음 문자를 읽을 수 있도록 준비해줍니다. 하지만 엿본다는 것은, 해당 문자를 읽고도 스트림 위치 지정자를 움직이지 않는다는 것이지요. 따라서 다음 읽기 때 엿본 문자를 읽을 수 있게 됩니다.

그렇다면 왜 `peek`의 결과가 `w`가 나올까요? 아래 그림을 통해 이해하시면 쉽습니다.



일단, 우리가 `hello world`를 친 다음, `std::cin >> s`를 한 이후의 `streambuf`의 상태입니다. 문자열의 경우 공백문자가 나오기 전 까지 읽어들이기 때문에 위와 같은 상태가 됩니다. 이제, `snextc()` 함수를 호출 했을 때 상태를 보자면;!



`snextc` 함수가 스트림 위치 지정자를 한 칸 전진시키므로, 공백 문자를 뛰어넘고, `w`를 가리키게 됩니다. 그리고, 이에 해당하는 문자인 `w`를 리턴하게 됩니다. 이 때 `snextc` 함수는 스트림 위치 지정자를 건드리지 않기 때문에,

```
std::cin >> s;
std::cout << "다시 읽으면 : " << s << std::endl;
```

에서 `world` 전체가 나오게 되지요.

`streambuf`에는 `snextc` 함수 말고도 수 많은 함수들이 정의되어 있습니다. 물론 이 함수들을 직접 사용할 일은 거의 없겠지만, C++ 입출력 라이브러리는 스트림 버퍼도 추상화해서 클래스로 만들었다는 것 정도로만 기억하시면 좋을 것 같습니다. 또한, C++에서 `streambuf`를 도입한 중요한 이유 한 가지는, 1 바이트 짜리 문자 뿐만이 아니라 `wchar_t`, 즉 다중 바이트 문자들 (우리가 흔히 말하는 UTF-8 같은 것이지요)에 대한 처리도 용이하게 하기 위해서입니다.

예를 들어서, 다중 바이트 문자의 경우, 사용자가 문자 한 개만 요구했음에도 스트림에서는 1 바이트만 읽을 수 있고, 2 바이트, 심지어 4 바이트 까지 필요한 경우가 있습니다. C++ 에서는 이러한 것들에 대한 처리를 스트림 버퍼 객체 자체에서 수행하도록 해서, 사용자가 입출력 처리를 이용하는데 훨씬 용이하게 하였습니다.

이상으로 C++ 에서의 입출력 라이브러리에 대해 간단히 알아보았습니다. 다음 강좌에서는 이제 이 라이브러리를 가지고 파일에서 어떠한 방식으로 입출력을 수행할 수 있는지 알아보도록 하겠습니다.

C++ 파일 입출력

안녕하세요 여러분! 지난 강좌에서 C++에서 표준 스트림과의 입출력에 대해 간단히 다루어보았습니다. 이번에는 이를 이용해서 파일 스트림과의 입출력을 다루어 보도록 하겠습니다. 사실, 파일 입출력은 표준 스트림에서 입출력 하는 것과 크게 다른 점은 없습니다. 다만, 스트림이 화면 혹은 키보드에서 파일로 바뀌었을 뿐이지요.

fstream

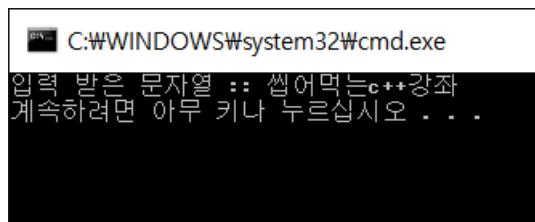
파일 스트림은 기본적인 `istream`이나 `ostream` 클래스 보다 더 지원하는 기능이 더 많기 때문에 이를 상속 받아서 작성되었으며, 각각을 상속 받은 것이 `ifstream`과 `ofstream`입니다. 이들 클래스를 모두 포함하는 라이브러리로 `fstream`을 사용하면 됩니다.

```
// 파일에서의 입출력
#include <fstream>
#include <iostream>
#include <string>

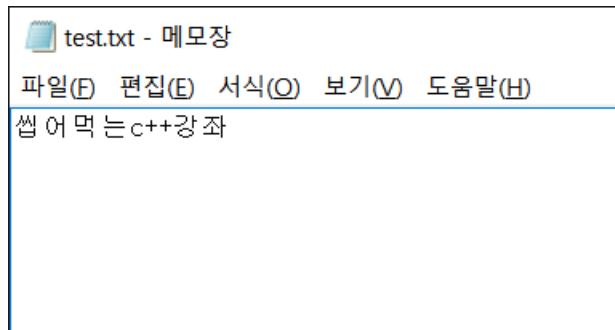
int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt");
    std::string s;

    if (in.is_open()) {
        in >> s;
        std::cout << "입력 받은 문자열 :: " << s << std::endl;
    } else {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
    }
    return 0;
}
```

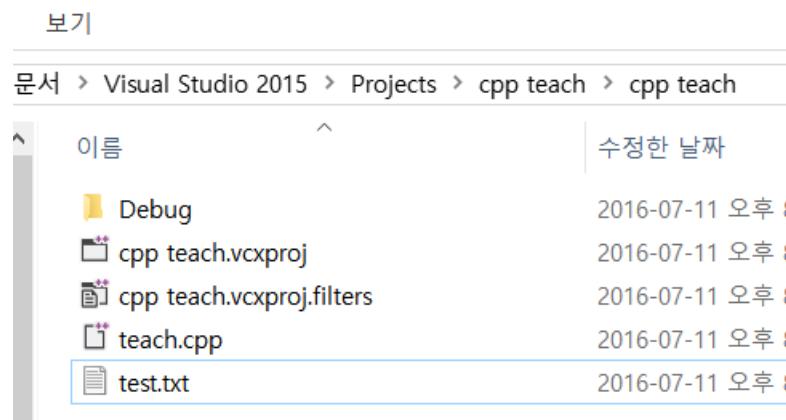
성공적으로 컴파일 하였다면



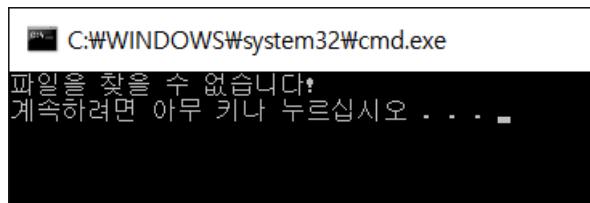
와 같이 나옵니다. 참고로 `test.txt` 파일에는 다음과 같이 써 있었습니다.



참고로 `test.txt`를 읽어드리는 경로는, 비주얼 스튜디오 상에서 실행하였을 때 소스파일이 들어 있는 위치와 동일합니다.



만일 `test.txt`를 지워버려서 파일을 찾을 수 없다면 `is_open`이 `false`를 반환해서



와 같이 파일을 찾을 수 없다고 표시 됩니다.

아마 소스가 매우 간단하므로 그냥 봐도 이해 하실 수 있으리라 생각합니다. 기존의 콘솔에서 사용자 입력을 받는 것과 별반 다를 게 없어 보입니다. 단 한가지 빼고요.

```
// 파일 읽기 준비
std::ifstream in("test.txt");
```

기존에 `cout`이나 `cin`을 사용했을 때에는 이미 표준 입출력에 연동이 되어 있는 상황이었지만, 파일 입출력에 경우 어느 파일에 입출력을 해야 할지 지정해야 하는데, 이를 `ifstream` 객체에 생성자에 연동하고자 하는 파일의 경로 ("C:\\a\\b\\c.txt" 와 같이)를 전달하면 됩니다. 위 경우

편의상 경로를 저렇게 썼지만 (이 경우 실행 파일과 같은 경로에 있는 파일을 찾게 됩니다. 다만 비주얼 스튜디오 상에서 실행할 경우에는 소스 파일과 같은 경로에 있는 것을 찾습니다)

위와 같이 생성자에 파일 경로를 지정하면, 해당하는 파일을 찾고 열게 됩니다. 만일 파일이 존재하지 않는다면 파일을 열 수 없습니다. 따라서 파일이 열렸는지의 유무는 다음과 같이 확인할 수 있습니다.

```
if (in.is_open()) {
```

`is_open` 은 기존의 `istream` 에는 없고 `ifstream` 에서 상속 받으면서 추가된 함수입니다. 파일이 열렸는지의 유무를 리턴합니다. 만일 해당 경로에 있는 파일이 존재하지 않는다면 `false` 를 리턴하겠지요.

```
in >> s;
```

마지막으로 마치 `cin` 을 사용 하는 것 처럼 `in` 객체를 이용해서 파일로 부터 읽을 수 있습니다. (`cin` 에서 `>>` 로 읽을 때 공백 문자가 나올 까지 읽었던 것처럼 여기도 동일합니다)

여기서 한 가지 흥미로운 점이 있습니다. C 언어에서 기억을 되돌려 보자면 파일 입출력을 한 후에 꼭 `fclose` 를 해주어야 했었습니다. 그런데 여기서하는 신기하게도 그러한 작업을 하지 않습니다. 왜 그렇냐면, 이미 `ifstream` 객체의 소멸자에서 자동적으로 `close` 를 해주기 때문입니다.

다만 `close` 를 직접 해야 되는 경우도 있습니다.

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt");
    std::string s;

    if (in.is_open()) {
        in >> s;
        std::cout << "입력 받은 문자열 :: " << s << std::endl;
    } else {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
    }

    in.close();
    in.open("other.txt");

    if (in.is_open()) {
        in >> s;
        std::cout << "입력 받은 문자열 :: " << s << std::endl;
    } else {
```

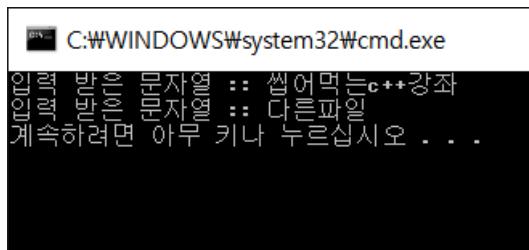
```

    std::cout << "파일을 찾을 수 없습니다!" << std::endl;
}

return 0;
}

```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

```

in.close();
in.open("other.txt");

```

위처럼 새로운 파일에서 같은 객체가 입력을 받기 위해서는 기존 파일 스트림과의 연결을 종료하고, 새로운 파일과 연결을 시켜주면 됩니다. 기존 파일과의 스트림 종료는 `close` 함수가, 새로운 파일과의 연결은 `open` 함수가 수행하고 있습니다. `open` 함수가 있기에 굳이 `ifstream` 객체 생성자에서 파일 경로를 바로 지정해줄 필요는 없고, 나중에 `open`으로 원하는 파일을 열어도 상관 없습니다.

```

// 이진수로 읽기
#include <fstream>
#include <iostream>
#include <string>

int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt", std::ios::binary);
    std::string s;

    int x;
    if (in.is_open()) {
        in.read((char*)&x, 4);
        std::cout << std::hex << x << std::endl;
    } else {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
    }
}

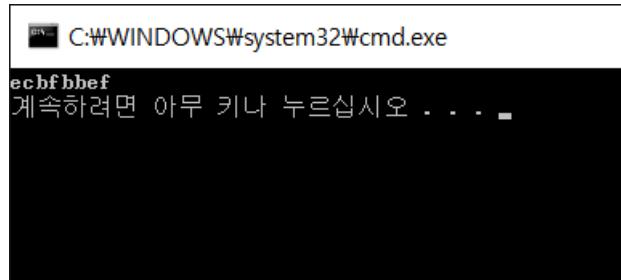
```

```

    return 0;
}

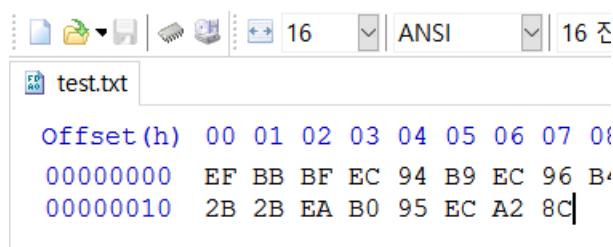
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

실제로 Hex 에디터로 `test.txt` 의 내용을 살펴보아도



와 같이 첫 부분이 일치하는 것으로 나타납니다.

"어 다른데요?"

라고 생각하시는 분들은 엔디안을 간과한 것인데, 우리가 쓰는 CPU의 경우 리틀 엔디안이라 해서, 높은 주소값에 높은 자리수가 온다고 생각하면 됩니다, 따라서 각각의 바이트가 EF / BB / BF / EC 가 거꾸로 EC / BF / BB / EF 이렇게 int 변수에 기록이 된 것입니다. (이에 대한 내용은 C 강좌에서도 다루었습니다)

```
std::ifstream in("test.txt", std::ios::binary);
```

일단 위와 같이 `ifstream` 객체를 생성할 때 생성자에 옵션으로 `binary` 형태로 받겠다고 명시할 수 있습니다. 이 말은 문제별 데이터를 받는게 아니라 그냥 이진 그대로의 값을 받아내겠다는 의미입니다. 만일 아무것도 명시 하지 않는다면 위에서 보았던 것처럼 문자열 형태로 데이터를 받습니다.

이 `binary` 는 단순한 숫자로 `ios`에 정의되어 있습니다. `binary` 말고도 설정할 수 있는 여러가지 옵션들이 있는데. 이들을 OR 해서 여러가지 옵션을 조합할 수 있습니다. ([여기](#)에서의 비트연산 활용 부분을 생각하시면 됩니다)

```
in.read((char*)(&x), 4);
```

`read` 함수는 말 그대로, 4 바이트의 내용을 읽으라는 의미로, 첫 번째 인자에 해당하는 베퍼를 전달해주어야 합니다. 우리의 경우 `int` 변수를 마치 4 바이트 짜리 `char` 배열이라 생각하게 해서 이를 전달하였습니다. 두 번째 인자로 반드시 몇 바이트를 읽을 지 전달해야 합니다.

```
char x[10];
in.read(x, 10);
```

실제로 예시 코드처럼 `int` 공간에 저장하는 경우는 없고, 위처럼 그냥 `char` 배열에 크기를 지정해서 읽어들이면 됩니다.

```
std::cout << std::hex << x << std::endl;
```

참고로 `cout`에서 사용한 `hex` 역시 지난 강좌에서 `cin`에서 사용한 `hex`과 비슷한 부류로 16진수로 정수 데이터를 표시해줍니다.

파일 전체 읽기

파일 전체를 한 번에 읽기

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt");
    std::string s;

    if (in.is_open()) {
        // 위치 지정자를 파일 끝으로 옮긴다.
        in.seekg(0, std::ios::end);

        // 그리고 그 위치를 읽는다. (파일의 크기)
        int size = in.tellg();

        // 그 크기의 문자열을 할당한다.
        s.resize(size);

        // 위치 지정자를 다시 파일 맨 앞으로 옮긴다.
        in.seekg(0, std::ios::beg);

        // 파일 전체 내용을 읽어서 문자열에 저장한다.
        in.read(&s[0], size);
        std::cout << s << std::endl;
    }
}
```

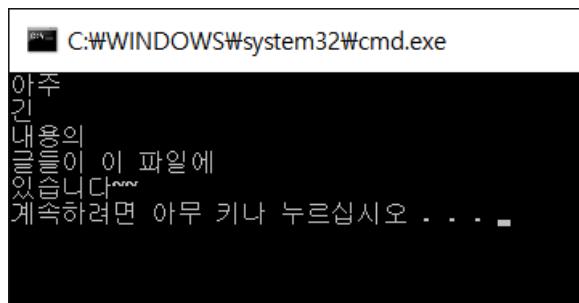
```

} else {
    std::cout << "파일을 찾을 수 없습니다!" << std::endl;
}

return 0;
}

```

성공적으로 컴파일 하였다면



와 같이 파일 전체의 내용이 잘 나오고 있습니다.

```

// 위치 지정자를 파일 끝으로 옮긴다.
in.seekg(0, std::ios::end);

```

C 언어에서 `fseek` 과 같은 함수로, 파일 위치 지정자를 사용자의 입맛에 맞게 이리저리 움직일 수 있습니다. 두 번째 인자는, 파일 내 위치를 의미하고, 첫 번째 인자는 그 위치로부터 얼마나 만큼 떨어져 있느냐를 의미합니다. 우리의 경우 위치 지정자를 파일의 끝에서 0 만큼 떨어진 것, 즉 파일의 끝으로 이동시켰습니다.

```

// 그리고 그 위치를 읽는다. (파일의 크기)
int size = in.tellg();

```

`tellg` 함수는 위치 지정자의 위치 (시작 지점으로부터의) 를 반환합니다. 현재 우리가 위치 지정자를 파일 끝으로 이동 시켜 놓았기 때문에 `tellg` 함수는 파일의 크기 (바이트 단위) 로 반환하겠지요. 그리고 문자열에 그 만큼의 크기를 할당해줍니다.

```

// 위치 지정자를 다시 파일 맨 앞으로 옮긴다.
in.seekg(0, std::ios::beg);

```

이제 파일을 읽어야 할 텐데, 파일 위치 지정자를 끝으로 옮겨 놓았기 때문에 읽기 위해서는 다시 처음으로 옮겨주어야 합니다. 옮기지 않을 경우 위치 지정자가 파일 끝에 있으므로 아무것도 읽지 못할 것입니다.

```
// 파일 전체 내용을 읽어서 문자열에 저장한다.
in.read(&s[0], size);
```

마지막으로 파일 전체에 내용을 문자열에 저장하면 됩니다.

파일 전체를 한 줄씩 읽기

```
// getline 으로 읽어들이기
#include <fstream>
#include <iostream>
#include <string>

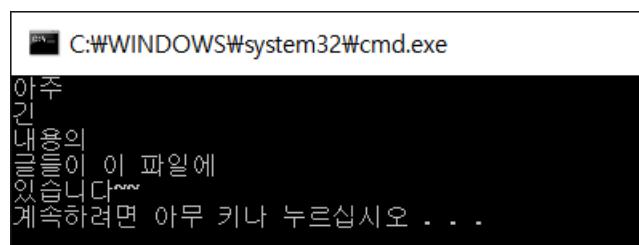
int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt");
    char buf[100];

    if (!in.is_open()) {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
        return 0;
    }

    while (in) {
        in.getline(buf, 100);
        std::cout << buf << std::endl;
    }

    return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

위 `ifstream` 객체의 멤버 함수로 존재하는 `getline` 함수는 파일에서 개행문자 (`\n`) 이 나올 때 가지 최대 지정한 크기 - 1 만큼 읽게됩니다. 왜 하나 적게 읽냐면, `buf` 의 맨 마지막 문자로 널 종료 문자를 넣어줘야 하기 때문이지요.

위 경우 buf에 최대 99 글자 까지 입력 받습니다. 물론 개행 문자 말고도 여러분이 지정한 문자가 나올 때 까지 읽는 것으로 바꿀 수도 있습니다. 이 경우 원하는 문자를 인자로 전달해주면 해당 문자가 나올 때 까지 입력 받습니다. 예를 들어서

```
in.getline(buf, 100, '\n');
```

이런식으로 하면 마침표가 나올 때 까지 입력받게 됩니다.

```
while (in) {
```

`ifstream`에는 자기 자신을 `bool`로 캐스팅 할 수 있는 캐스팅 연산자(`operator bool()`)가 오버로딩 되어 있습니다. 따라서 위와 같이 `while` 문 조건에 `in`을 전달한다면 `bool`로 캐스팅 하는 연산자 함수가 호출됩니다. 이 때 `in`이 `true`이기 위해서는 다음 입력 작업이 성공적이어야만 하고 현재 스트림에 오류 플래그가 켜져 있지 않아야만 합니다.

하지만 `getline` 함수는 개행 문자 (혹은 지정한 문자) 가 나오기 전에 지정한 버퍼의 크기가 다 차게 된다면 `failbit`를 키게 되므로 버퍼의 크기를 너무 작게 만든다면 정상적으로 데이터를 받을 수 없습니다. 따라서 `getline`을 사용하기 전에 이와 같은 조건을 꼭 확인해야 합니다.

이와 같은 한계점을 극복하기 위해서 `std::string`에서 `getline` 함수를 제공하고 있습니다.

```
// std::string 에 경의된 getline 사용
#include <fstream>
#include <iostream>
#include <string>

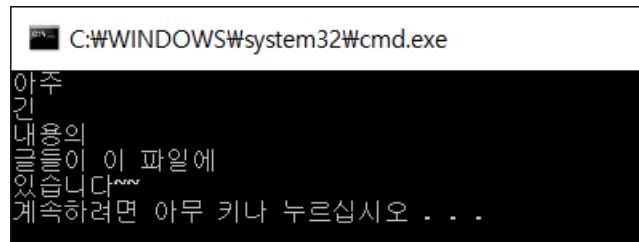
int main() {
    // 파일 읽기 준비
    std::ifstream in("test.txt");

    if (!in.is_open()) {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
        return 0;
    }

    std::string s;
    while (in) {
        getline(in, s);
        std::cout << s << std::endl;
    }

    return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

이 `getline` 함수는 `ifstream`에 정의되어 있는 것이 아니라, `std::string`에 정의되어 있는 함수로, 첫 번째 인자로 `istream` 객체를 받고, 두 번째 인자로 입력 받은 문자열을 저장할 `string` 객체를 받게 됩니다.

기존에 `ifstream`의 `getline`을 활용할 때 보다 훨씬 편리한 것이, 굳이 버퍼의 크기를 지정하지 않아도 알아서 개행문자 혹은 파일에 끝이 나올 때 까지 입력받게 됩니다.

주의 사항

한 가지 주의할 사항으로 `while` 문 조건으로 절대 `in.eof()`를 사용하면 안됩니다. 이러한 코드를 사용했다면 99 퍼센트의 확률로 잘못된 코드입니다. 왜냐하면 `eof` 함수는 파일 위치 지시자가 파일에 끝에 도달한 이후에 `true`를 리턴하기 때문입니다.

예를 들어서 `while` 문 안에서 파일을 쭈르륵 읽다가 파일 끝(EOF) 바로 직전까지 읽었다고 해봅시다. 그렇다면 아직 EOF를 읽지 않았으므로 `in.eof()`는 참인 상태일 것입니다. 그 상태에서 예컨대 `in >> data`를 하게 된다면 `data`에는 아무것도 들어가지 않게 됩니다. 즉 초기화가 되지 않은 상태로 남아있는 것입니다!

다시 말해 `in.eof()`는 `while` 문 안에서 파일 읽기가 안전하다 라는 것을 보장하지 않습니다. 정확한 사용법은 그냥 `while(in)`처럼 스트림 객체 자체를 전달하는 것입니다. 앞에서도 말했듯이 `istream` 객체는 다음 읽기가 안전할 때만 `true`로 캐스팅됩니다.

파일에 쓰기

```

#include <iostream>
#include <fstream>
#include <string>

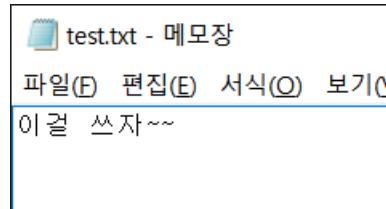
int main() {
    // 파일 쓰기 준비
    std::ofstream out("test.txt");

    std::string s;
    if (out.is_open()) {
        out << "이걸 쓰자~~";
    }
}

```

```
    return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 test.txt에 내용이 잘 써진 것을 알 수 있습니다.

만일 test.txt가 존재하지 않을 경우, test.txt를 생성한 뒤에, 생성이 성공하였다면 출력하게 됩니다. ofstream은 열려는 파일이 존재하지 않으면 해당 파일을 생성하고 열게 됩니다. 만일, 해당 파일이 이미 존재한다면, 특별한 설정을 하지 않는다면 해당 파일 내용이 다 지워지고 새로운 내용으로 덮어 써어지게 됩니다.

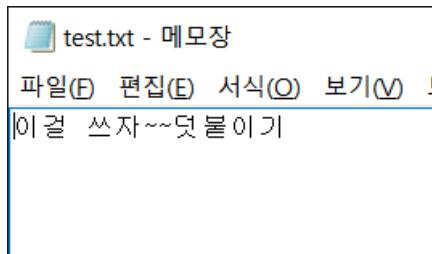
```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    // 파일 쓰기준비
    std::ofstream out("test.txt", std::ios::app);

    std::string s;
    if (out.is_open()) {
        out << "덧붙이기";
    }

    return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

`out` 객체를 생성할 때 옵션으로 `app` 을 주게 되면, 파일에 스트림을 연결할 때 기존 파일의 내용을 지우고 새로 쓰는 것이 아니라 위 사진처럼 그 뒤에 새로운 내용을 붙여 쓰게 됩니다.

앞서 나왔던 `ios::binary` 와 `ios::app` 말고도 4개가 더 있습니다. 이들을 나열해보자면

- `ios::ate` : 자동으로 파일 끝에서부터 읽기와 쓰기를 실시합니다. (즉 파일을 열 때 위치 지정자가 파일 끝을 가리키고 있게 됩니다)
- `ios::trunc` : 파일 스트림을 열면 기존에 있던 내용들이 모두 지워집니다. 기본적으로 `ofstream` 객체를 생성할 때 이와 같은 설정으로 만들어집니다.
- `ios::in, std::ios::out` : 파일에 입력을 할지 출력을 할지 지정하며, `ifstream` 과 `ofstream` 객체를 생성할 때 각각은 이미 설정되어 있습니다.

참고로 `ios::ate` 와 `ios::app` 은 비슷해 보이지만 차이가 있다면 `ios::app` 의 경우 원본 파일의 내용을 무조건 적으로 보장하지만, `ate` 는 위치 지정자를 그 이전으로 옮길 수 있습니다. 즉 `app` 의 경우 파일 위치 지정자가 기존 파일의 끝이 시작점이라 생각하여 움직이며 `ate` 의 경우 기존 파일을 포함해서 움직입니다 (사실 `ate` 의 경우 사용할 일이 거의 없을 것입니다).

그렇다고 해서 `ate` 를 이용해서 기존에 있는 파일 데이터 한 가운데에 무언가를 끼워 넣을 수 있는 것은 아닙니다.

```
// ate 와 app
#include <fstream>
#include <iostream>
#include <string>

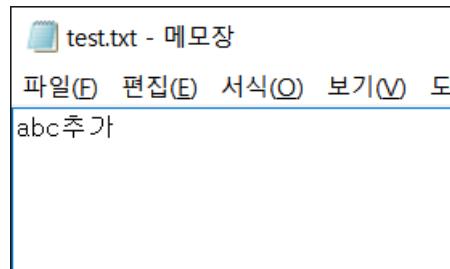
int main() {
    // 두 파일에는 모두 abc 라고 써 있었습니다.
    std::ofstream out("test.txt", std::ios::app);
    std::ofstream out2("test2.txt", std::ios::ate);

    out.seekp(3, std::ios::beg);
    out2.seekp(3, std::ios::beg);

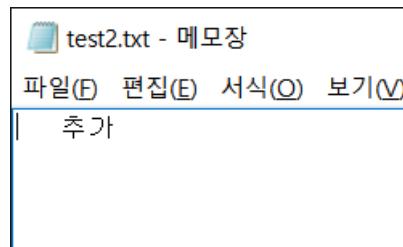
    out << "추가";
    out2 << "추가";

    return 0;
}
```

성공적으로 컴파일 하였다면



`app` 을 사용한 경우 위와 같이 `abc` 바로 뒤에 '추가' 문자열에 붙어 있는 것으로 나타납니다. 비록 파일 위치 지정자를 앞에서 3 칸 떨어진 곳으로 이동하였음에도, `app` 모드로 읽었을 때 현재 파일은 빈 파일이라 생각되어 위치 지정자라 움직일 공간이 없기에, 실제로 위치 지정자는 움직이지 않고 출력되었습니다.



반면에 `ate` 를 사용한 경우 앞에서 3 칸 떨어진 곳에 '추가' 라고 문자열이 출력된 반면 기존의 `abc`라는 데이터는 지워졌습니다. 즉 `ate` 모드로 파일을 열게 되면 비록 스트림 위치 지정자는 움직여서 3칸 뒤에 출력되기는 하였지만 기존에 써져 있던 내용은 모두 지워집니다. (`ate` 는 기존 파일의 내용을 보존하지 않습니다)

std::ofstream 연산자 오버로딩 하기

```
#include <fstream>
#include <iostream>
#include <string>

class Human {
    std::string name;
    int age;

public:
    Human(const std::string& name, int age) : name(name), age(age) {}
    std::string get_info() {
        return "Name :: " + name + " / Age :: " + std::to_string(age);
    }

    friend std::ofstream& operator<<(std::ofstream& o, Human& h);
};

}
```

```

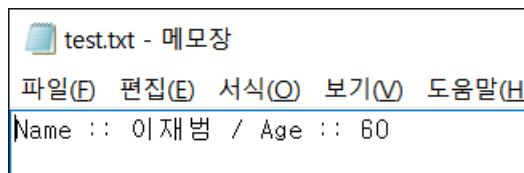
std::ofstream& operator<<(std::ofstream& o, Human& h) {
    o << h.get_info();
    return o;
}
int main() {
    // 파일 쓰기 준비
    std::ofstream out("test.txt");

    Human h("이재범", 60);
    out << h << std::endl;

    return 0;
}

```

성공적으로 컴파일 하였다면



와 같이 나옴을 알 수 있습니다. 이전 강좌에서 입출력 연산자 오버로딩을 한 번 해보았는데, `ofstream` 이라고 해서 달라지는 것은 없습니다. 단순히 `ofstream` 객체의 레퍼런스를 받고, 다시 이를 리턴하는 `operator<<` 함수를 정의해주면 됩니다.

문자열 스트림 (`std::stringstream`)

```

#include <iostream>
#include <sstream>

int main() {
    std::istringstream ss("123");
    int x;
    ss >> x;

    std::cout << "입력 받은 데이터 :: " << x << std::endl;

    return 0;
}

```

성공적으로 컴파일 하였다면

실행 결과

```
입력 받은 데이터 :: 123
```

와 같이 나옵니다.

`sstream`에는 `std::istringstream`이 정의되어 있는데 이는 마치 문자열을 하나의 스트림이라 생각하게 해주는 가상화 장치라고 보시면 됩니다.

```
std::istringstream ss("123");
```

예를 들어서 우리는 위를 통해서 문자열 "123"이 기록되어 있는 입력 스트림을 생성하였습니다. 마치 파일에 123이라 기록해놓고 거기서 입력 받는 것과 동일하다고 생각하면 됩니다.

```
int x;
ss >> x;
```

그래서 마치 파일에서 숫자를 읽어내는 것처럼 `std::istringstream`을 통해서 123을 읽어낼 수 있습니다.

이를 활용하면 `atoi`와 같은 함수를 사용할 필요 없이 간편하게 문자열에서 숫자로 변환하는 함수를 만들 수 있습니다.

```
#include <iostream>
#include <sstream>
#include <string>

double to_number(std::string s) {
    std::istringstream ss(s);
    double x;

    ss >> x;
    return x;
}

int main() {
    std::cout << "변환:: 1 + 2 = " << to_number("1") + to_number("2") << std::endl;

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
변환:: 1 + 2 = 3
```

위와 같이 간편하게 문자열을 숫자로 변환할 수 있습니다.

```
#include <iostream>
#include <sstream>
#include <string>

std::string to_str(int x) {
    std::ostringstream ss;
    ss << x;

    return ss.str();
}

int main() {
    std::cout << "문자열로 변환:: 1 + 2 = " << to_str(1 + 2) << std::endl;

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
문자열로 변환:: 1 + 2 = 3
```

와 같이 나옵니다.

이번에는 거꾸로 데이터를 출력할 수 있는 `std::ostringstream`이 있습니다. 위와 비슷한 방법으로 이번에는 거꾸로 숫자에서 문자열로 바꾸는 함수를 제작할 수 있습니다.

```
std::ostringstream ss;
ss << x;
```

위와 같이 `int` 변수 `x`의 값을 문자열 스트림에 출력하였습니다. 이 과정에서 자동으로 숫자에서 문자열로의 변환이 있겠지요.

```
return ss.str();
```

이제 `str` 함수로 현재 문자열 스트림에 쓰여 있는 값을 불러오기만 하면 끝납니다.

이상으로 이번 강좌를 마치도록 하겠습니다. 다음 강좌에서는 여태 까지 배운 내용들을 총 종합하여 큰 프로젝트 하나를 만들도록 하겠습니다. 다음 강좌에서 제작하기 전에, 아래 생각해보기를 통해서 먼저 여러분 스스로 구현해 보는 것도 좋을 것 같습니다.

생각 해보기

여러분은 콘솔 용 엑셀을 만들 것입니다. 물론 진짜 엑셀처럼 엄청 거대한 프로그램을 만들 겠다는 것은 아니고, 기본적인 것들만 구현한 엑셀 프로그램이 될 것입니다.

문제 1 (난이도 :中)

일단 엑셀의 셀들의 정보 (일단 단순한 `std::string` 이라고 생각합시다)에 대한 `Cell` 클래스가 있고 이 `Cell` 객체들을 모아두는 `Table` 클래스가 있습니다. `Table` 클래스에는 2차원 배열로 `Cell` 객체들에 대한 정보 (참고로 `Cell` 객체가 생성될 때마다 동적으로 `Cell` 객체를 생성합니다.)가 보관되어 있습니다. 또한 `Table` 클래스에 전체 데이터를 출력하는 `print_table` 함수가 가상으로 정의되어 있습니다.

여러분은 `Table` 클래스를 상속 받는 `TextTable`, `CSVTable`, `HTMLTable` 클래스를 만들어서 `print_table` 함수를 오버라이드 할 함수들을 제작할 것입니다. 예를 들어 `TextTable` 클래스의 `print_table` 함수는 텍스트 형식으로, `CSVTable`은 CSV 파일 형식으로 등등 만들어야 겠지요? 제가 아래 대충 프로그램의 골격을 잡아 놓았으니 여러분들은 이를 채우기만 하면 됩니다.

```

class Table;
class Cell {
    Table* table; // 어느 테이블에 속해있는지
    std::string data;
    int x, y; // 테이블에서의 위치
public:
    Cell(const std::string& data) : data(data){};
};

class Table {
    Cell*** data_base; // 왜 3중 포인터인지 잘 생각해보세요!
public:
    Table();
    virtual std::string print_table() = 0;
    void reg_cell(Cell* c, int row, int col); // Cell 을 등록한다
    std::string get_cell_std::string(int row,
                                    int col); // 해당 위치의 Cell 데이터를 얻는다.
    ~Table();
};

ostream& operator<<(ostream& o, Table& t) {
    o << t.print_table();
}

```

```
    return o;
}
class TextTable : public Table {};
class CSVTable : public Table {};
class HTMLTable : public Table {};
```

문제 2 (난이도 :最上 - 위의 문제와 이어집니다)

하지만 실제 엑셀의 경우 셀이 문자열 데이터만 들어가는 것이 아니라, 숫자나 날짜 심지어 수식 까지도 들어갈 수 있습니다. 따라서 우리는 `Cell` 을 상속 받는 4 개의 `StringCell`. `NumberCell`, `DateCell`, `ExprCell` 클래스들을 만들어야 합니다.

또한 `Cell` 클래스에 `to_numeric` (데이터를 숫자로 바꾼다)과 `std::stringify` (데이터를 문자열로 바꾼다) 함수들을 가상으로 정의하고, 4개의 클래스들이 이를 상속 받아서 구현할 수 있게 해야 합니다. (참고로 문자열을 숫자로 변환하면 그냥 0 이 되게 하면 됩니다)

또한 `ExprCell` 의 경우 간단한 수식에 대한 정보를 가지는 객체로, `Cell` 들 간의 연산을 사용할 수 있습니다. 예를 들어서 $A1+B2+C6-6$ 와 같은 데이터가 들어 있는 `ExprCell` 에 `to_numeric` 함수를 호출하면 $A1$, $B2$, $C6$ 의 값을 더하고 (각 셀에 `to_numeric` 을 해서), 6 을 빼준 결과값이 나와야 합니다.

참고로 프로그래밍의 편의를 위해서 `ExprCell` 의 경우, 셀을 지칭하는 것은 딱 두 글자 ($A1$, $Z9$ 처럼) 로 하고, 숫자는 오직 한 자리 수 정수, 그리고 가능한 연산자는 `+`, `-`, `*`, `/` 로 하겠습니다.

아마도 여태 까지 강좌에서 한 프로그래밍 중에 가장 도전적인 것이 아닌가 싶습니다. 참고로 위를 구현하기 위해서 여러가지 자료형이 필요할 텐데 (있으면 편리합니다!), 대표적으로 벡터와 스택이 있습니다. 벡터는 가변 길의 배열로, 배열 처럼 사용하면서 사용자가 임의의 위치에 자료를 넣다 뺏다 할 수 있는 구조고, 스택은 `pop` 과 `push` 밖에 없는 자료형으로, `push` 을 하면 새로운 데이터가 맨 위에 삽입되고, `pop` 을 하면 맨 위에 있는 것부터 제거되어 나오게 됩니다. (즉 먼저 넣은것이 나중에 나오는 자료구조입니다) 저는 이를 자료 구조를 새롭게 만들어서 사용하였습니다. 여러분들도 아마 필요하실 것입니다.

아마 여러분이 이 강좌에서 여태 까지 구현했던 프로그램 중에 가장 복잡할 것입니다. 많은 시간이 필요한 만큼 재미가 있을 테니 한 번 다음 강좌를 보기 전 까지 도전해보시기 바랍니다!

엑셀 만들기 프로젝트

안녕하세요 여러분! 이번 강좌에서는 여태 까지 배운 내용들을 바탕으로 하나의 작은 C++ 프로젝트를 진행해볼 예정입니다. 지난번 강좌의 생각해보기에서 공지하기는 했지만, 바로 콘솔로 Excel 을 만드는 것입니다. 물론 마이크로소프트의 그것 처럼 거대하게 만들 수는 없지만, 그래도 기본적인 것들은 지원 할 수 있는 형태로 만들어 볼 예정입니다.

본격적으로 Excel 을 구현하기에 앞서서, 몇 가지 자료 구조를 만들 것입니다. 자료구조라고 함은, 컴퓨터에서 데이터를 저장하는 방식이라 할 수 있는데, 그 구현에 따라서 장단점이 각각 있습니다. (장점만을 가진 자료 구조는 없습니다. 항상 무언가를 위해서 다른 무언가를 포기해야 되는 법이지요)

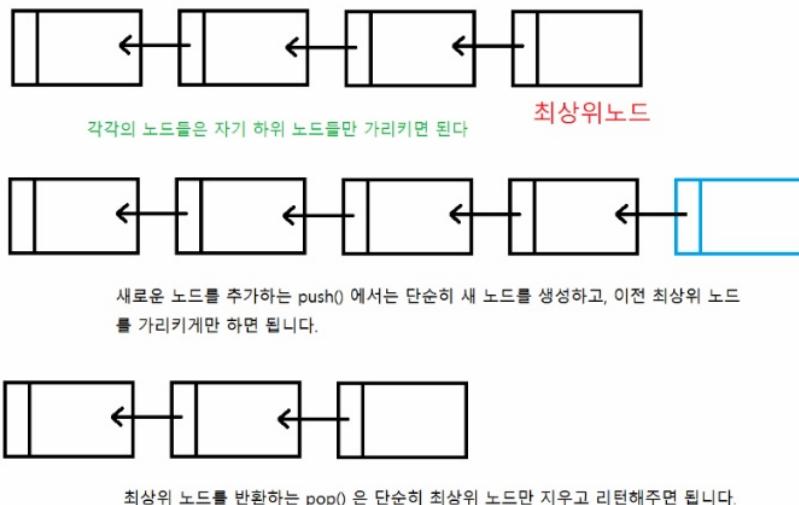
이 Excel 프로그램에서 사용할 자료구조는 크게 `Vector` 와 `Stack` 입니다. 참고로 이들은 수식을 분석하기 위해, 즉 `ExprCell` 객체의 `to_numeric` 함수 내에서 사용될 예정입니다.

각 자료구조들은 다음과 같은 특징을 가지고 있습니다.

- 벡터 (Vector) : 수학의 그 벡터와는 살짝 다른 느낌인데, 그냥 배열의 크기를 맘대로 조절할 수 있는 가변길이 배열이라 보시면 됩니다. 즉, 배열 처럼 [] 연산자로 임의의 위치에 있는 원소에 마음대로 접근할 수 있고 또 임의의 위치에 원소를 추가하거나 뺄 수 있습니다. 벡터를 만드는 방법은 이전에 문자열 클래스를 만들 때와 거의 비슷합니다. 문자열 역시 `char` 데이터를 담는 가변 길이 배열과 마찬가지 이기 때문이지요.
- 스택 (Stack) : 벡터와는 다르게 임의의 위치에 있는 원소에 접근할 수 없고 항상 최 상단에 있는 데이터에만 접근할 수 있습니다. 그리고 새로운 데이터를 추가하면 최상단에 오게 됩니다. 쉽게 말해서 설거지 용으로 쌓아 놓는 접시들이라 보면 됩니다. 새로운 설거지 거리가 오면 쌓여 있는 접시 맨 위에 오게 되고(push). 설거지를 위해서 접시를 뺄 때 맨 위의 접시 부터 빼겠지요(pop).

물론 스택은 그냥 벡터를 활용해서 만들 수 있습니다. 하지만 이는 마치 소 잡는 칼을 닦 잡는 데 쓰는 것이라고나 할까요. 보통 스택의 경우 링크드 리스트(Linked List - [이 강좌의 Node 부분](#)을 살펴보세요) 를 이용해서 구현을 합니다. 스택은 임의의 위치에 데이터에 접근할 필요가 없습니다.

단순히 최상위에 뭐가 있을지 궁금하고 또 거기에 새로운 것을 추가하던지 빼기만 하면 되겠지요. 아래 스택을 간단히 어떻게 구현할지 그림으로 보여드리겠습니다.



즉 스택의 경우 복잡하게 생각할 필요 없이 위와 같이 구성하면 됩니다.

벡터 클래스 (Vector)

먼저 문자열을 보관하기 위한 벡터 부터 만들겠습니다. 우리의 벡터 클래스는 다음과 같이 구성되어 있습니다.

```
class Vector {
    string* data;
    int capacity;
    int length;

public:
    // 생성자
    Vector(int n = 1);

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(string s);

    // 임의의 위치의 원소에 접근한다.
    string operator[](int i);

    // x 번째 위치한 원소를 제거한다.
    void remove(int x);

    // 현재 벡터의 크기를 구한다.
    int size();
```

```
~Vector();
};
```

먼저 클래스 소스를 살펴보도록 합시다.

```
string* data;
int capacity;
int length;
```

`Vector` 클래스는 위와 같이 데이터를 보관하기 위한 `data` (문자열 배열로 만들 것입니다), 현재 할당되어 있는 크기를 알려주는 `capacity`, 그리고 현재 실제로 사용하는 양인 `length` 와 같은 변수로 구성되어 있습니다.

```
// 생성자
Vector(int n = 1);
```

한 가지 특이한 점은 생성자에서 인자가 저렇게 `int n = 1` 과 같이 지정되어 있다는 것입니다. 이는 무엇이냐면, 만일 사용자가 인자를 지정하지 않으면, 알아서 `n = 1` 이 되게 지정한다는 것입니다. 다시 말해서

```
Vector a() Vector a(1)
```

은 동일한 작업입니다. 물론 사용자가 인자를 지정하면 해당 인자가 들어가겠지요. 이렇게 해당 인자의 기본 값을 지정해 놓은 것을 디폴트 인자 (**Default argument**) 라고 합니다. 이렇게 하면 사용자가 인자를 지정하지 않아도 디폴트 값이 들어가기 때문에 문제 없이 사용할 수 있습니다.

```
Vector::Vector(int n) : data(new string[n]), capacity(n), length(0) {}
```

`Vector`의 생성자를 살펴보면 위와 같습니다. 한 가지 흥미로운 점은 여기서는 디폴트 인자가 명시되지 않은 점입니다. 이는 C++ 규칙이기도 한데, 클래스 내부 함수 선언에서 디폴드 인자를 명시하였다며 함수 본체에서 명시하면 안되고, 반대로 함수 본체에서 명시하였다며 클래스 내부 함수 선언에 명시하면 안됩니다. 즉, 둘 중 한 곳에서만 표시해야 합니다.

```
void Vector::push_back(string s) {
    if (capacity <= length) {
        string* temp = new string[capacity * 2];
        for (int i = 0; i < length; i++) {
            temp[i] = data[i];
        }
        delete[] data;
        data = temp;
        capacity *= 2;
    }
    data[length] = s;
    length++;
}
```

```

        data = temp;
        capacity *= 2;
    }

    data[length] = s;
    length++;
}

string Vector::operator[](int i) { return data[i]; }

void Vector::remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

int Vector::size() { return length; }

Vector::~Vector() {
    if (data) {
        delete[] data;
    }
}

```

간단히 위처럼 `Vector` 클래스의 함수들을 만들어 보았습니다. 이 `Vector` 클래스는 일반적으로 다른 사람들이 사용할 것이 아니라 제가 이 프로젝트에서 간단히 사용하기 위해 만들어놓은 것이므로 몇 가지 문제점들이나 구현하지 않는 함수들(보통 `Vector` 클래스에는 중간에 원소를 추가하는 `insert` 나 검색하는 `find` 함수들도 세트로 다닙니다)이 있습니다. 물론 이렇게 한 이유는 이 정도로도 Excel 프로젝트에는 충분하기 때문에 문제 없습니다.

```

void Vector::push_back(string s) {
    if (capacity <= length) {
        string* temp = new string[capacity * 2];
        for (int i = 0; i < length; i++) {
            temp[i] = data[i];
        }
        delete[] data;
        data = temp;
        capacity *= 2;
    }

    data[length] = s;
    length++;
}

```

우리 `Vector` 클래스의 `push_back` 함수는 배열 맨 끝에 원소를 집어넣는 클래스입니다. 위

방법은 기존에 문자열 클래스에서 썼던 방법으로, 만일 배열이 다 차게 되면 1 칸을 더 늘리는 것이 아니라 현재 크기의 두 배 만큼을 새로 할당하고 새로 할당한 공간에 복사하는 것입니다. 이렇게 된다면 가장 효율적으로 공간과 시간을 활용할 수 있습니다.

스택 클래스

이번에는 스택 클래스입니다. 스택의 경우 위에서 말한 것처럼 링크드 리스트를 사용하기 때문에 데이터를 보관하기 위해서 배열을 사용하는 것이 아니라 하나의 노드를 만들어서 노드들을 체인처럼 엮을 것입니다. 이를 위해 아래와 같이 Stack 클래스 안에 Node라는 구조체를 정의하였습니다.

```
struct Node {
    Node* prev;
    string s;

    Node(Node* prev, string s) : prev(prev), s(s) {}
};
```

Node 객체에는 자기 보다 하위 Node를 가리키는 포인터(prev)와, 자신이 보관하는 데이터에 관한 값(s)을 보관하는 두 개의 변수로 구성되어 있습니다. 그냥 맨 위에 그려놓은 스택 구현 모습을 그대로 표현하였다고 생각하면 됩니다. 아래는 전체 Stack 클래스의 모습입니다.

```
class Stack {
    struct Node {
        Node* prev;
        string s;

        Node(Node* prev, string s) : prev(prev), s(s) {}
    };

    Node* current;
    Node start;

    public:
    Stack();

    // 최상단에 새로운 원소를 추가한다.
    void push(string s);

    // 최상단의 원소를 제거하고 반환한다.
    string pop();

    // 최상단의 원소를 반환한다. (제거 안함)
    string peek();

    // 스택이 비어있는지의 유무를 반환한다.
};
```

```
bool is_empty();
~Stack();
};
```

Node 들의 리스트를 정확하게 관리하기 위해서, `current` 와 `start` 를 만들어서 `current` 는 현재 최상위 노드를 가리키게 하고, `start` 는 맨 밑바닥을 이루는 노드, 즉 최하위 노드를 가리키게 하였습니다. `start` 노드를 둔 이유는, 마지막 노드에 도달하였을 때 그 여부를 알아야 하기 때문이지요.

```
Stack::Stack() : start(NULL, "") { current = &start; }
void Stack::push(string s) {
    Node* n = new Node(current, s);
    current = n;
}
string Stack::pop() {
    if (current == &start) return "";

    string s = current->s;
    Node* prev = current;
    current = current->prev;

    // Delete popped node
    delete prev;
    return s;
}
string Stack::peek() { return current->s; }
bool Stack::is_empty() {
    if (current == &start) return true;
    return false;
}
Stack::~Stack() {
    while (current != &start) {
        Node* prev = current;
        current = current->prev;
        delete prev;
    }
}
```

위와 같이 간단하게 `Stack` 을 구성하였습니다. 주의해야 할 점은, 소멸자에서 최상위 원소 부터 줄줄이 바닥에 도달할 때 까지 메모리에서 해제시켜야 완전히 `Stack` 객체를 소멸시킬 수 있습니다.

스택의 경우 위와 같이 문자열을 받는 것 말고도, 숫자 데이터를 보관하는 스택인 `NumStack` 클래스 또한 `string` 만 `int` 로 바꿔서 동일하게 만들었습니다.

최종적으로 아래는 우리가 만든 벡터와 스택 클래스의 헤더 파일인 `utils.h` 의 전체 내용입니다.

```
#ifndef UTILS_H
#define UTILS_H

#include <string>
using std::string

namespace MyExcel {
    class Vector {
        string* data;
        int capacity;
        int length;

    public:
        // 생성자
        Vector(int n = 1);

        // 맨 뒤에 새로운 원소를 추가한다.
        void push_back(string s);

        // 임의의 위치의 원소에 접근한다.
        string operator[](int i);

        // x 번째 위치한 원소를 제거한다.
        void remove(int x);

        // 현재 벡터의 크기를 구한다.
        int size();

        ~Vector();
    };

    class Stack {
        struct Node {
            Node* prev;
            string s;

            Node(Node* prev, string s) : prev(prev), s(s) {}

        };

        Node* current;
        Node start;

    public:
        Stack();

        // 최상단에 새로운 원소를 추가한다.
        void push(string s);

        // 최상단의 원소를 제거하고 반환한다.
        string pop();
    };
}
```

```

// 최상단의 원소를 반환한다. (제거 안함)
string peek();

// 스택이 비어있는지의 유무를 반환한다.
bool is_empty();

~Stack();
};

class NumStack {
    struct Node {
        Node* prev;
        double s;

        Node(Node* prev, double s) : prev(prev), s(s) {}
    };

    Node* current;
    Node start;

public:
    NumStack();
    void push(double s);
    double pop();
    double peek();
    bool is_empty();

    ~NumStack();
};

}

#endif

```

마찬가지로 아래는 해당 헤더파일 내용을 구현한 utility.cpp 입니다.

```

#include "utils.h"

namespace MyExcel {
Vector::Vector(int n) : data(new string[n]), capacity(n), length(0) {}
void Vector::push_back(string s) {
    if (capacity <= length) {
        string* temp = new string[capacity * 2];
        for (int i = 0; i < length; i++) {
            temp[i] = data[i];
        }
        delete[] data;
        data = temp;
        capacity *= 2;
    }
    data[length] = s;
}

```

```
    length++;
}

string Vector::operator[](int i) { return data[i]; }

void Vector::remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

int Vector::size() { return length; }

Vector::~Vector() {
    if (data) {
        delete[] data;
    }
}

Stack::Stack() : start(NULL, "") { current = &start; }

void Stack::push(string s) {
    Node* n = new Node(current, s);
    current = n;
}

string Stack::pop() {
    if (current == &start) return "";

    string s = current->s;
    Node* prev = current;
    current = current->prev;

    // Delete popped node
    delete prev;
    return s;
}

string Stack::peek() { return current->s; }

bool Stack::is_empty() {
    if (current == &start) return true;
    return false;
}

Stack::~Stack() {
    while (current != &start) {
        Node* prev = current;
        current = current->prev;
        delete prev;
    }
}

NumStack::NumStack() : start(NULL, 0) { current = &start; }

void NumStack::push(double s) {
    Node* n = new Node(current, s);
    current = n;
}

double NumStack::pop() {
    if (current == &start) return 0;
```

```

double s = current->s;
Node* prev = current;
current = current->prev;

// Delete popped node
delete prev;
return s;
}

double NumStack::peek() { return current->s; }

bool NumStack::is_empty() {
    if (current == &start) return true;
    return false;
}

NumStack::~NumStack() {
    while (current != &start) {
        Node* prev = current;
        current = current->prev;
        delete prev;
    }
}
}
}

```

참고로 우리의 Excel 관련한 모든 코드는 MyExcel이라는 이름 공간에 담겨 있을 것입니다.

본격적인 Cell 과 Table 클래스

```

class Cell {
protected:
    int x, y;
    Table* table;

    string data;

public:
    virtual string stringify();
    virtual int to_numeric();

    Cell(string data, int x, int y, Table* table);
};

```

Cell 클래스는 큰 테이블에서 한 칸을 의미하는 객체로, 해당 내용을 보관하는 문자열 data 와 어느 테이블에 위치해 있는지에 관련한 정보를 가지고 있는 table 과 그 위치 x, y 로 구성되어 있습니다.

또한, 가상 함수로 해당 셀 값을 문자열로 변환하는 `stringify` 함수와, 정수 데이터로 변환하는 `to_numeric` 함수도 선언되어 있습니다. 물론 문자열에 `to_numeric`을 수행하게 되면 당연히 0을 리턴하겠지만, 나중에 `Cell` 클래스를 `NumberCell`과 같은 클래스들이 상속 받기 위한 큰 그림이라고 보시면 됩니다.

따라서 `Cell` 멤버 함수들의 정의는 아래와 같이 간단하게 나타낼 수 있습니다.

```
Cell::Cell(string data, int x, int y, Table* table)
    : data(data), x(x), y(y), table(table) {}

string Cell::stringify() { return data; }

int Cell::to_numeric() { return 0; }
```

자 그럼 `Table` 클래스의 정의를 살펴보도록 하겠습니다.

```
class Table {
protected:
    // 행 및 열의 최대 크기
    int max_row_size, max_col_size;

    // 데이터를 보관하는 테이블
    // Cell* 을 보관하는 2차원 배열이라 생각하면 편하다
    Cell*** data_table;

public:
    Table(int max_row_size, int max_col_size);

    ~Table();

    // 새로운 셀을 row 행 col 열에 등록한다.
    void reg_cell(Cell* c, int row, int col);

    // 해당 셀의 정수값을 반환한다.
    // s : 셀 이름 (Ex. A3, B6 과 같이)
    int to_numeric(const string& s);

    // 행 및 열 번호로 셀을 호출한다.
    int to_numeric(int row, int col);

    // 해당 셀의 문자열을 반환한다.
    string stringify(const string& s);
    string stringify(int row, int col);

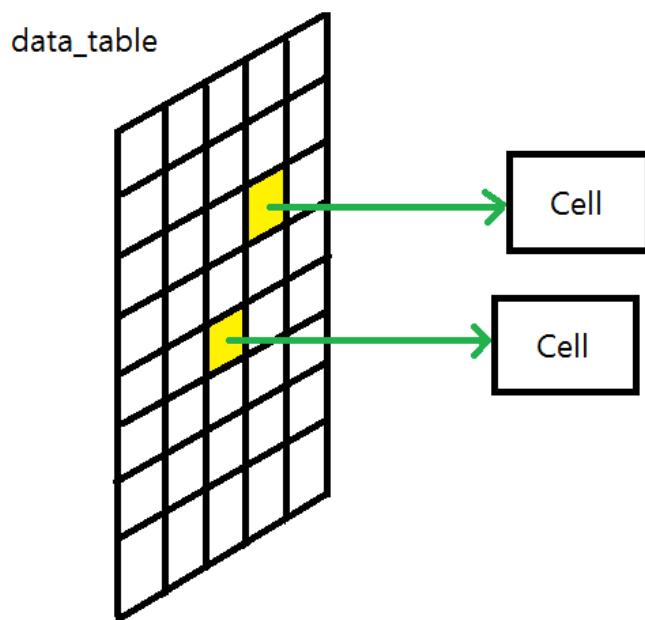
    virtual string print_table() = 0;
};
```

일단 `Table` 클래스는 `Cell` 객체들을 2 차원 배열로 보관하게 됩니다. 이 때, 객체 자체를 보관하는

것이 아니라, 객체는 필요할 때마다 동적으로 생성하고, 그 객체에 대한 포인터를 2차원 배열로 보관하고 있게 됩니다.

```
Table::Table(int max_row_size, int max_col_size)
    : max_row_size(max_row_size), max_col_size(max_col_size) {
    data_table = new Cell**[max_row_size];
    for (int i = 0; i < max_row_size; i++) {
        data_table[i] = new Cell*[max_col_size];
        for (int j = 0; j < max_col_size; j++) {
            data_table[i][j] = NULL;
        }
    }
}
```

따라서 Table 클래스의 생성자는 위와 같이 정의될 수 있습니다.



위 그림을 보면 쉽게 이해할 수 있듯이, 동적 할당으로 `Cell*` 배열을 생성한 후에, `Cell` 객체가 필요 할 때마다 생성해서 배열의 원소들이 이를 가리킬 수 있게 하였습니다.

```
Table::~Table() {
    for (int i = 0; i < max_row_size; i++) {
        for (int j = 0; j < max_col_size; j++) {
            if (data_table[i][j]) delete data_table[i][j];
        }
    }
    for (int i = 0; i < max_row_size; i++) {
        delete[] data_table[i];
    }
}
```

```
    delete[] data_table;
}
```

Table 소멸자도 이와 비슷합니다. 일단, 동적으로 생성된 Cell 객체를 모두 지워야 하고 그 다음에 Cell 배열 (1차원) 을 지워야 하고 마지막으로 2차원 테이블 자체를 메모리에서 지워야 합니다. 3 단계에 걸쳐서 Cell 의 흔적으로 메모리에서 날려버릴 수 있습니다.

```
void Table::reg_cell(Cell* c, int row, int col) {
    if (!(row < max_row_size && col < max_col_size)) return;

    if (data_table[row][col]) {
        delete data_table[row][col];
    }
    data_table[row][col] = c;
}
```

위는 Table 의 셀을 등록하는 함수입니다. 등록하고자 하는 위치를 인자로 받는데, 만일 해당 위치에 이미 다른 셀 객체가 등록되어 있다면 해당 객체를 delete 한 후에 등록시켜주면 됩니다.

```
int Table::to_numeric(const string& s) {
    // Cell 이름으로 받는다.
    int row = s[0] - 'A';
    int col = atoi(s.c_str() + 1) - 1;

    if (row < max_row_size && col < max_col_size) {
        if (data_table[row][col]) {
            return data_table[row][col]->to_numeric();
        }
    }
    return 0;
}

int Table::to_numeric(int row, int col) {
    if (row < max_row_size && col < max_col_size && data_table[row][col]) {
        return data_table[row][col]->to_numeric();
    }
    return 0;
}

string Table::stringify(const string& s) {
    // Cell 이름으로 받는다.
    int col = s[0] - 'A';
    int row = atoi(s.c_str() + 1) - 1;

    if (row < max_row_size && col < max_col_size) {
        if (data_table[row][col]) {
            return data_table[row][col]->stringify();
        }
    }
}
```

```

    return 0;
}
string Table::stringify(int row, int col) {
    if (row < max_row_size && col < max_col_size && data_table[row][col]) {
        return data_table[row][col]->stringify();
    }
    return "";
}
std::ostream& operator<<(std::ostream& o, Table& table) {
    o << table.print_table();
    return o;
}

```

마지막으로 해당하는 셀의 값을 반환하는 함수들로, 두 가지 형태로 구성되어 있는데 하나는 셀 이름(A1, B2 이렇게)을 받아서 해당하는 위치의 값을 리턴하는 함수와 행과 열 값을 받아서 해당 위치에 셀이 있으면 그 값을 리턴하는 함수들로 구성되어 있습니다.

또한 맨 마지막에 `ostream` 클래스의 `<<` 연산자를 오버로딩하는 함수를 하나 만들어서 파일이나 표준 스트림(`cout`) 입출력에 쉽게 사용할 수 있도록 하였습니다.

하지만 이 `Table` 클래스의 객체는 생성할 수 없습니다. 왜냐하면 아래와 같은 순수 가상 함수가 포함되어 있기 때문이지요.

```
virtual string print_table() = 0;
```

우리는 이 `Table` 클래스를 상속 받는 다른 클래스를 만들어서 이 함수를 구현해주어야만 합니다.

```

class TxtTable : public Table {
    string repeat_char(int n, char c);

    // 숫자로 된 열 번호를 A, B, .... Z, AA, AB, ... 이런 순으로 매겨준다.
    string col_num_to_str(int n);

public:
    TxtTable(int row, int col);

    // 텍스트로 표를 깨끗하게 출력해준다.
    string print_table();
};

```

위는 `Table` 클래스를 상속 받는 `TxtTable` 클래스입니다. 이 클래스는 `Table` 의 내용을 텍스트의 형태로 예쁘게 정리해서 출력해주는 역할을 하고 있습니다.

```
TxtTable::TxtTable(int row, int col) : Table(row, col) {}
```

```
// 텍스트로 표를 깨끗하게 출력해준다.
string TxtTable::print_table() {
    string total_table;

    int* col_max_wide = new int[max_col_size];
    for (int i = 0; i < max_col_size; i++) {
        unsigned int max_wide = 2;
        for (int j = 0; j < max_row_size; j++) {
            if (data_table[j][i] &&
                data_table[j][i]->stringify().length() > max_wide) {
                max_wide = data_table[j][i]->stringify().length();
            }
        }
        col_max_wide[i] = max_wide;
    }
    // 맨 상단에 열 정보 표시
    total_table += "      ";
    int total_wide = 4;
    for (int i = 0; i < max_col_size; i++) {
        if (col_max_wide[i]) {
            int max_len = max(2, col_max_wide[i]);
            total_table += " | " + col_num_to_str(i);
            total_table += repeat_char(max_len - col_num_to_str(i).length(), ' ');
            total_wide += (max_len + 3);
        }
    }

    total_table += "\n";
    // 일단 기본적으로 최대 9999 번째 행 까지 지원한다고 생각한다.
    for (int i = 0; i < max_row_size; i++) {
        total_table += repeat_char(total_wide, '-');
        total_table += "\n" + to_string(i + 1);
        total_table += repeat_char(4 - to_string(i + 1).length(), ' ');

        for (int j = 0; j < max_col_size; j++) {
            if (col_max_wide[j]) {
                int max_len = max(2, col_max_wide[j]);

                string s = "";
                if (data_table[i][j]) {
                    s = data_table[i][j]->stringify();
                }
                total_table += " | " + s;
                total_table += repeat_char(max_len - s.length(), ' ');
            }
        }
        total_table += "\n";
    }

    return total_table;
}
```

```

}

string TxtTable::repeat_char(int n, char c) {
    string s = "";
    for (int i = 0; i < n; i++) s.push_back(c);

    return s;
}

// 숫자로 된 열 번호를 A, B, .... Z, AA, AB, ... 이런 순으로 매겨준다.
string TxtTable::col_num_to_str(int n) {
    string s = "";
    if (n < 26) {
        s.push_back('A' + n);
    } else {
        char first = 'A' + n / 26 - 1;
        char second = 'A' + n % 26;

        s.push_back(first);
        s.push_back(second);
    }

    return s;
}

```

위는 그 구현입니다. `repeat_char` 과 `col_num_to_str` 함수는 단순히 `print_table`에서 사용할 부가적인 함수들입니다. `print_table` 함수는 각 열의 최대 문자열 길이를 계산한 뒤에, 이를 바탕으로 각 열의 폭을 결정해서 표를 출력해줍니다.

참고로 이 구현 방식에서 한 가지 중요한 것이 빠졌는데, 셀의 문자열 데이터에서 개행 문자가 있는 경우(즉 특정 셀이 여러 줄이 될 때)를 고려하지 않았습니다. 즉, 모든 셀은 최대 1 줄로만 그려지게 됩니다. 따라서 실제로는 각 행의 최대 높이 역시 열과 마찬가지로 계산해서 그려야 합니다. (이는 여러분의 몫으로 남기겠습니다)

```

// 생략
int main() {
    MyExcel::TxtTable table(5, 5);
    std::ofstream out("test.txt");

    table.reg_cell(new Cell("Hello~", 0, 0, &table), 0, 0);
    table.reg_cell(new Cell("C++", 0, 1, &table), 0, 1);

    table.reg_cell(new Cell("Programming", 1, 1, &table), 1, 1);
    std::cout << std::endl << table;
    out << table;
}

```

성공적으로 컴파일 하였다면

	A	B	C	D	E
1	Hello~	C++			
2		Programming			
3					
4					
5					

계속하려면 아무 키나 누르십시오 . . .

와 같이 잘 나오게 됩니다.

또한 `test.txt` 파일에도 역시

	A	B	C	D	E
1	Hello~	C++			
2		Programming			
3					
4					
5					

위와 같이 표가 잘 출력됩니다.

마찬가지로 저는 CSV 파일 형태와 HTML 형태로 데이터를 표현해주는 두 개의 클래스들을 더 만들었습니다.

```
class HtmlTable : public Table {
public:
    HtmlTable(int row, int col);

    string print_table();
};

class CSVTable : public Table {
public:
    CSVTable(int row, int col);
```

```
    string print_table();
};
```

딱히 특별한 것은 없고, HTML 파일 형식이나 CSV 파일 형식을 잘 알고 있다면 만드는데 큰 문제가 없을 것입니다. ([HTML 표](#), [CSV 파일 형식](#))

```
// 생략
int main() {
    MyExcel::CSVTable table(5, 5);
    std::ofstream out("test.csv");

    table.reg_cell(new Cell("Hello~", 0, 0, &table), 0, 0);
    table.reg_cell(new Cell("C++", 0, 1, &table), 0, 1);

    table.reg_cell(new Cell("Programming", 1, 1, &table), 1, 1);
    out << table;

    MyExcel::HtmlTable table2(5, 5);
    std::ofstream out2("test.html");

    table2.reg_cell(new Cell("Hello~", 0, 0, &table), 0, 0);
    table2.reg_cell(new Cell("C++", 0, 1, &table), 0, 1);
    table2.reg_cell(new Cell("Programming", 1, 1, &table), 1, 1);
    out2 << table2;
}
```

그리고 그 구현 내용은 다음과 같습니다.

```
HtmlTable::HtmlTable(int row, int col) : Table(row, col) {}

string HtmlTable::print_table() {
    string s = "<table border='1' cellpadding='10'>";
    for (int i = 0; i < max_row_size; i++) {
        s += "<tr>";
        for (int j = 0; j < max_col_size; j++) {
            s += "<td>";
            if (data_table[i][j]) s += data_table[i][j]->stringify();
            s += "</td>";
        }
        s += "</tr>";
    }
    s += "</table>";
    return s;
}

CSVTable::CSVTable(int row, int col) : Table(row, col) {}

string CSVTable::print_table() {
```

```

string s = "";
for (int i = 0; i < max_row_size; i++) {
    for (int j = 0; j < max_col_size; j++) {
        if (j >= 1) s += ",";
        // CSV 파일 규칙에 따라 문자열에 큰따옴표가 포함되어 있다면 "" 로
        // 치환하다.
        string temp;
        if (data_table[i][j]) temp = data_table[i][j]->stringify();

        for (int k = 0; k < temp.length(); k++) {
            if (temp[k] == '\"') {
                // k 의 위치에 " 를 한 개 더 집어넣는다.
                temp.insert(k, 1, '\"');

                // 이미 추가된 " 를 다시 확인하는 일이 없게 하기 위해
                // k 를 한 칸 더 이동시킨다.
                k++;
            }
        }
        temp = '\"' + temp + '\"';
        s += temp;
    }
    s += '\n';
}
return s;
}

```

성공적으로 컴파일 하였다면

	A	B	C	D
1	Hello~	C++		
2		Programming		
3				

CSV 파일의 경우 위와 같이 (실제) 엑셀에서 잘 열리고

Hello~	C++			
	Programming			

HTML 파일로 변환한 경우 위와 같이 브라우저 상에서 잘 표현됨을 알 수 있습니다.

이상으로 간단히 엑셀 만들기 프로젝트 1 부를 마치도록 하겠습니다. 다음 강좌에서는 Cell 을
상속 받는 클래스들을 만들어서 마치 실제 엑셀 처럼 작동하는 엑셀을 만들어 보도록 하겠습니다.

엑셀 만들기 프로젝트 2부

안녕하세요 여러분. 지난 강좌에 이어서 Excel 만들기 프로젝트를 계속 진행해 보도록 하겠습니다. 지난 강좌에서는 셀에 문자열 데이터만 넣을 수 있지만, 테이블을 여러가지 형태 (텍스트, HTML, CSV)로 출력할 수 있는 엑셀을 제작하였습니다.



하지만 시간에 만들었던 엑셀은 아직 엑셀이라 하기에는 기능이 조금 부족하였습니다. 실제 엑셀을 살펴보자면 셀에 문자열만 넣을 수 있는 것이 아니라 숫자 데이터도 넣을 수 있고 날짜도 넣을 수 있고, 심지어는 수식도 넣어서 연산 까지 할 수 있는 만능 셀입니다.

Cell 클래스 확장

앞서 말했듯이, Cell 클래스에는 `string` 데이터만 저장할 수 있기 때문에 이를 상속 받는 클래스들을 만들어서 셀에 다양한 데이터들을 보관할 수 있게 할 것입니다.

```
class Cell {
protected:
    int x, y;
    Table* table;

public:
    virtual string stringify() = 0;
    virtual int to_numeric() = 0;

    Cell(int x, int y, Table* table);
};
```

일단 기존의 `Cell` 클래스에서 문자열 데이터를 보관했던 것과는 달리 아예 그 항목을 빼버리고, 이를 상속 받는 클래스에서 데이터를 보관하도록 하였습니다. 또한, `stringify` 함수와 `to_numeric`

을 순수 가상 함수로 정의해서 이를 상속 받는 클래스에서 이 함수들을 반드시 구현 토록 하였습니다.

```

class StringCell : public Cell {
    string data;

public:
    string stringify();
    int to_numeric();

    StringCell(string data, int x, int y, Table* t);
};

class NumberCell : public Cell {
    int data;

public:
    string stringify();
    int to_numeric();

    NumberCell(int data, int x, int y, Table* t);
};

class DateCell : public Cell {
    time_t data;

public:
    string stringify();
    int to_numeric();

    DateCell(string s, int x, int y, Table* t);
};

```

일단 위 셋은 각각 문자열, 정수, 시간 정보를 보관하는 클래스들입니다. 사실 이들을 구현하는 것은 그렇게 어렵지 않습니다. 단순히 데이터를 문자열이나 정수 형으로 바꾸기만 해주면 되기 때문이지요. 참고로 DateCell의 경우에는 편의를 위해서 yyyy-mm-dd 형식으로만 입력을 받는 것으로 정하였습니다. 그 결과 다음과 같습니다.

```

Cell::Cell(int x, int y, Table* table) : x(x), y(y), table(table) {}

StringCell::StringCell(string data, int x, int y, Table* t)
    : data(data), Cell(x, y, t) {}

string StringCell::stringify() { return data; }

int StringCell::to_numeric() { return 0; }

/*
NumberCell

```

```

/*
NumberCell::NumberCell(int data, int x, int y, Table* t)
    : data(data), Cell(x, y, t) {}

string NumberCell::stringify() { return to_string(data); }
int NumberCell::to_numeric() { return data; }

*/
DateCell

/*
string DateCell::stringify() {
    char buf[50];
    tm temp;
    localtime_s(&temp, &data);

    strftime(buf, 50, "%F", &temp);

    return string(buf);
}
int DateCell::to_numeric() { return static_cast<int>(data); }

DateCell::DateCell(string s, int x, int y, Table* t) : Cell(x, y, t) {
    // 입력받는 Date 형식은 항상 yyyy-mm-dd 꼴이라 가정한다.
    int year = atoi(s.c_str());
    int month = atoi(s.c_str() + 5);
    int day = atoi(s.c_str() + 8);

    tm timeinfo;

    timeinfo.tm_year = year - 1900;
    timeinfo.tm_mon = month - 1;
    timeinfo.tm_mday = day;
    timeinfo.tm_hour = 0;
    timeinfo.tm_min = 0;
    timeinfo.tm_sec = 0;

    data = mktime(&timeinfo);
}

```

참고로 DateCell의 경우 구현이 조금 복잡한데 자세히 살펴보도록 하겠습니다.

```

// 입력받는 Date 형식은 항상 yyyy-mm-dd 꼴이라 가정한다.
int year = atoi(s.c_str());
int month = atoi(s.c_str() + 5);
int day = atoi(s.c_str() + 8);

```

일단 위 처럼 입력 받은 문자열을 연도, 월, 일로 구분하게 됩니다.

```
tm timeinfo;

timeinfo.tm_year = year - 1900;
timeinfo.tm_mon = month - 1;
timeinfo.tm_mday = day;
timeinfo.tm_hour = 0;
timeinfo.tm_min = 0;
timeinfo.tm_sec = 0;

data = mktime(&timeinfo);
```

이를 바탕으로 `timeinfo` 객체를 초기화 합니다. `tm` 클래스는 일월년 시분초 단위로 데이터를 보관하는 클래스입니다. 하지만 우리의 `DateCell` 은 `time_t` 형태로 데이터를 보관하고 있는데 그 변환을 위해 `mkttime` 에 `timeinfo` 를 전달하면 변환할 수 있습니다. 참고로 `time_t` 타입은 1970년 부터 현재 시간 까지 몇 초가 흘렀는지 보관하는 정수형 변수라고 생각하시면 됩니다.

```
class ExprCell : public Cell {
    string data;
    string* parsed_expr;

    Vector exp_vec;

    // 연산자 우선 순위를 반환합니다.
    int precedence(char c);

    // 수식을 분석합니다.
    void parse_expression();

public:
    ExprCell(string data, int x, int y, Table* t);

    string stringify();
    int to_numeric();
};
```

그렇다면 이제 대망의 `ExprCell` 을 살펴볼 차례입니다. 일단 `to_numeric()` 에 다 넣을 수 없어서 두 개의 함수를 새로 만들었습니다. `precedence` 함수는 입력받은 연산자의 우선순위를 반환하고, `parse_expression` 함수는 수식을 분석해서 계산하기 편하게 해주는 함수입니다. 계산하기 편하게 한다는게 무슨 말이냐고요? 아래를 봐주시기 바랍니다.

수식 계산하기 - 중위 표기법과 후위 표기법

우리는 흔히 수식을 나타내기 위해 다음과 같이 써 왔습니다.

3 + 4 * 5 + 4 * (7 - 2)

이렇게 표기하는 방식을 중위 표기법이라고 합니다. 사실 우리는 위 방식에 익숙해서 어떠한 순서로 계산하는지 쉽게 알 수 있지만 컴퓨터에 경우 이를 계산하는데 조금 어려울 수 있습니다. 일단 고려해야 할 점들이 먼저 괄호를 우선으로 계산하고, 그 다음에 * 와 / , 그리고 + 와 - 의 우선 순위로 나누어서 계산해야 합니다.

쉽게 말해 위 수식의 경우 비록 맨 앞에 3 + 4 이 있지만 사실은 4 * 5 를 먼저 계산해야 됩니다. 즉, 컴퓨터가 이 수식을 계산하기 위해서는 계산하는 위치를 우선 순위에 맞게 이리 저리 옮겨다녀야 합니다.

위 처럼 피연산자와 피연산자 사이에 연산자를 넣는 형태로 수식을 표현하는 방법을 중위 표기법 (infix notation) 이라고 부릅니다. 쉽게 말해 연산자가 '중간'에 들어가서 중위 표기법이지요.

반면에 아래의 수식을 살펴보도록 합시다.

3 4 5 * + 4 7 2 - * +

우리가 흔히 생각하는 수식의 모습이랑 사뭇 다릅니다. 사실 위 수식은 앞서 말한 수식과 정확히 동일한 수식인데, 그 표현 방식이 다를 뿐입니다. 이러한 형태로 수식을 표현하는 방식을 후위 표기법(postfix notation)이라고 합니다. 자세히 보자면 이전 수식과 다른 점을 두 가지 찾을 수 있는데, 하나는 이전과는 달리 연산자들이 피연산자 뒤쪽에 위치해 있다는 점과, 또 하나는 괄호가 사라졌다는 점입니다.

괄호가 사라졌다는 것이 무슨 의미가 있을까요?

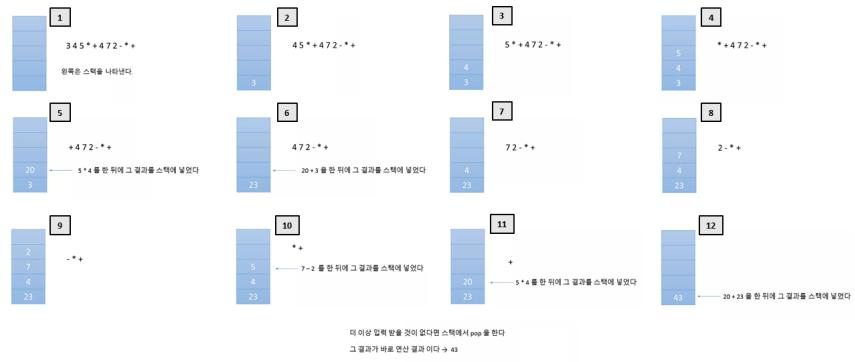
기존의 중위 표현법이 컴퓨터가 해석하기에 불편했던 점이 바로 연산자의 우선 순위나 괄호에 따라 이리 저리 계산하는 부분을 찾아다녀야 했던 점이었습니다. 하지만 후위 표기법에서는 놀랍게도 이리 저리 계산할 위치를 찾으려 돌아다닐 필요 없이, 읽어들이는 순서대로 계산을 쭉 할 수 있습니다. 하지만 후위 표기법에서는 놀랍게도 이리 저리 계산할 위치를 찾으려 돌아다닐 필요 없이, 읽어들이는 순서대로 계산을 쭉 할 수 있습니다.

물론 사람이 보기에는 조금 불편하지만 컴퓨터의 입장에서는, 즉 프로그래머의 입장에서는 코딩하기에 매우 편리한 표기 방법입니다.

그렇다면 이 후위 표기법으로 표현된 식을 컴퓨터가 어떻게 해석하는지 살펴보겠습니다. 컴퓨터는 아래와 같은 과정으로 위 후위 표기법으로 변환된 식을 계산합니다.

1. 피연산자를 만나면 스택에 push 합니다.
2. 연산자를 만나면 스택에서 두 개를 pop 한 뒤에 그 둘에 해당 연산을 한 후, 그 결과를 다시 스택에 push 합니다.

이와 같은 방식으로 위 수식을 계산해보도록 하겠습니다.



실제로 $3 + 4 * 5 + 4 * (7 - 2)$ 을 계산 했을 때와 그 결과가 같음을 알 수 있습니다.

이를 바탕으로 후위 표기법으로 된 수식을 계산하는 `is_numeric` 함수를 살펴보도록 하겠습니다.

```
int ExprCell::to_numeric() {
    double result = 0;
    NumStack stack;

    for (int i = 0; i < exp_vec.size(); i++) {
        string s = exp_vec[i];

        // 셀 일 경우
        if (isalpha(s[0])) {
            stack.push(table->to_numeric(s));
        }
        // 숫자 일 경우 (한 자리라 가정)
        else if (isdigit(s[0])) {
            stack.push(atoi(s.c_str()));
        } else {
            double y = stack.pop();
            double x = stack.pop();
            switch (s[0]) {
                case '+':
                    stack.push(x + y);
                    break;
                case '-':
                    stack.push(x - y);
                    break;
                case '*':
                    stack.push(x * y);
                    break;
                case '/':
                    stack.push(x / y);
                    break;
            }
        }
    }
    return stack.pop();
}
```

일단 우리는 `parse_expression` 함수를 통해서 입력 받은 중위 표기법으로 되어 있는 수식이, 후위 표기법으로 변환되어 있고, 그 결과가 `exp_vec`에 저장되어 있다고 생각해봅시다. `exp_vec`은 벡터 클래스 객체로, 각각의 원소가 후위 표기법으로 변환된 수식의 각각의 토큰이 됩니다. 즉, 앞선 예제의 경우 `exp_vec`은 `3, 4, 5, *, +, 4, 7, 2, -, *, +`으로 이루어진 배열이라 보시면 됩니다.

```
string s = exp_vec[i];
```

따라서 위와 같이 `for` 문을 통해 각각의 토큰(`exp_vec`의 각 원소들)에 접근할 수 있습니다.

```
// 셀 일 경우
if (isalpha(s[0])) {
    stack.push(table->to_numeric(s));
}
// 숫자 일 경우 (한 자리라 가정)
else if (isdigit(s[0])) {
    stack.push(atoi(s.c_str()));
}
```

그리고 각각의 토큰에 대해서, 셀 이름 (A3, B2 이렇게)이나 숫자일 경우 스택에 `push`하게 됩니다.

```
else {
    double y = stack.pop();
    double x = stack.pop();
    switch (s[0]) {
        case '+':
            stack.push(x + y);
            break;
        case '-':
            stack.push(x - y);
            break;
        case '*':
            stack.push(x * y);
            break;
        case '/':
            stack.push(x / y);
            break;
    }
}
```

아니면 연산자를 만날 경우 스택에서 두 번 `pop`을 해서 해당하는 피연산자들에 해당 연산자를 적용해서 다시 스택에 `push`하게 됩니다.

```
return stack.pop();
```

그리고 모든 계산이 끝나면 스택에 최종 결과값을 pop 하며 이를 리턴하게 됩니다.

자 그럼 이제 우리가 해야할 일은 중위 표기법으로 표기된 수식을 후위 표기법으로 변환하는 작업만 수행하면 됩니다.

중위 표기법을 후위 표기법으로 변환하기 (parse_expression 함수)

중위 표기법을 후위 표기법으로 변환하는 것은 다음의 방식을 따릅니다.

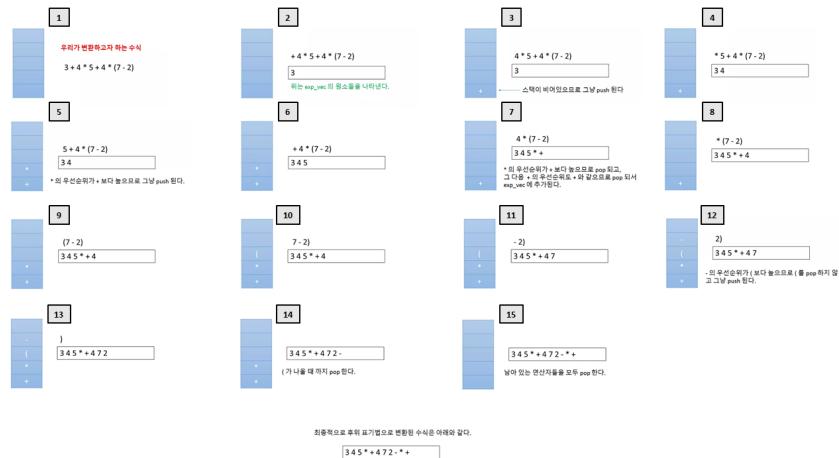
1. 피연산자 (셀 이름이나 숫자) 일 경우 그냥 exp_vec 에 넣습니다.
2. 여는 괄호((, [, { 와 같은 것들) 을 만날 경우 스택에 push 합니다.
3. 닫는 괄호(),], } 와 같은 것들) 을 만날 경우 여는 괄호가 pop 될 때 까지 pop 되는 연산자들을 exp_vec 에 넣습니다.
4. 연산자일 경우 자기 보다 우선순위가 낮은 연산자가 스택 최상단에 올 때 까지 (혹은 스택이 빌 때 까지) 스택을 pop 하고 (낮은 것은 pop 하지 않습니다), pop 된 연산자들을 exp_vec 에 넣습니다. 그리고 마지막에 자신을 스택에 push 합니다.

그리고 연산자들의 우선 순위는 아래의 함수에 의해 정의됩니다.

```
int ExprCell::precedence(char c) {
    switch (c) {
        case '(':
        case '[':
        case '{':
            return 0;
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
    }
    return 0;
}
```

괄호들이 가장 낮고, 그 다음이 + 와 - , 그리고 최상위 우선순위가 * 와 / 입니다.

그렇다면 $3 + 4 * 5 + 4 * (7 - 2)$ 가 어떻게 변환되는지 그림으로 살펴보도록 하겠습니다.



이제 위 방식으로 그대로 코드로만 옮기면 됩니다. 여기서는 코딩의 편의를 위해서 사용자가 입력하는 숫자는 1 자리 정수이고, 셀 이름 역시 2자로 제한하였습니다. 이를 확장하는 것은 크게 어렵지 않으니 여러분들이 직접 해보시기 바랍니다.

```
void ExprCell::parse_expression() {
    Stack stack;

    // 수식 전체를 ()로 둘러 사서 exp_vec에 남아있는 연산자들이 push 되게
    // 해줍니다.
    data.insert(0, "(");
    data.push_back(')');

    for (int i = 0; i < data.length(); i++) {
        if (isalpha(data[i])) {
            exp_vec.push_back(data.substr(i, 2));
            i++;
        } else if (isdigit(data[i])) {
            exp_vec.push_back(data.substr(i, 1));
        } else if (data[i] == '(' || data[i] == '[' ||
                   data[i] == '{') { // Parenthesis
            stack.push(data.substr(i, 1));
        } else if (data[i] == ')' || data[i] == ']' || data[i] == '}') {
            string t = stack.pop();
            while (t != "(" && t != "[" && t != "{") {
                exp_vec.push_back(t);
                t = stack.pop();
            }
        } else if (data[i] == '+' || data[i] == '-' || data[i] == '*' ||
                   data[i] == '/') {
            while (!stack.is_empty() &&
                   precedence(stack.peek()[0]) >= precedence(data[i])) {
                exp_vec.push_back(stack.pop());
            }
            stack.push(data.substr(i, 1));
        }
    }
}
```

```
    }
}
```

위 코드를 보면 변환 알고리즘을 그대로 옮겨놓았다고 생각하면 됩니다.

```
if (isalpha(data[i])) { // 셀 이름의 경우 첫 번째 글자가 알파벳이다.
    exp_vec.push_back(data.substr(i, 2));
    i++;
} else if (isdigit(data[i])) { // 첫번째 글자가 숫자라면 정수 데이터
    exp_vec.push_back(data.substr(i, 1));
}
```

일단 피연산자를 만날 경우 `exp_vec`에 무조건 집어넣으면 됩니다.

```
else if (data[i] == '(' || data[i] == '[' || data[i] == '{') { // Parenthesis
    stack.push(data.substr(i, 1));
}
else if (data[i] == ']' || data[i] == ']' || data[i] == '}') {
    string t = stack.pop();
    while (t != "(" && t != "[" && t != "{") {
        exp_vec.push_back(t);
        t = stack.pop();
    }
}
```

반면에 괄호의 경우 여는 괄호를 만나면 스택에 `push`하고, 닫는 괄호를 만나면 위처럼 여는 괄호가 스택에서 나올 때 까지 `pop`하고, 그 `pop` 한 연산자들을 벡터에 넣으면 됩니다. 주의할 점은 `pop`한 연산자가 괄호일 경우 넣지 않는다는 점입니다.

```
else if (data[i] == '+' || data[i] == '-' || data[i] == '*' || data[i] == '/') {
    while (!stack.is_empty() &&
           precedence(stack.peek()[0]) >= precedence(data[i])) {
        exp_vec.push_back(stack.pop());
    }
    stack.push(data.substr(i, 1));
}
```

마지막으로 연산자일 경우를 살펴봅시다. `peek`의 경우 스택의 최상단 원소를 `pop`하지는 않고 무엇인지만 살펴보는 것입니다. 만일 최상단 원소의 우선순위가 현재 연산자의 우선순위 보다 높다면 이를 스택에서 `pop`하고 이를 `exp_vec`에 넣어야겠지요. 위 `while` 문은 그 과정을 나타내고 있습니다.

그리고 맨 마지막에 스택에 현재 연산자를 넣습니다.

위 과정을 모두 마치면 후기 표기법으로 변환을 마칠 수 있을 것이라 생각되지만 사실 한 가지 빼먹은 사실이 있습니다. 마지막에 스택에 남아있는 연산자들을 모두 `pop` 해야 되기 때문이죠. 이를 `for` 문이 끝난 후에 `while` 문을 하나 더 넣어서 연산자를 `pop` 하는 과정을 넣을 수도 있지만 아래처럼 좀 더 간단하게 처리할 수도 있습니다.

```
// 수식 전체를 () 로 둘러 사서 exp_vec 에 남아있는 연산자들이 push 되게
// 해줍니다.
data.insert(0, "(");
data.push_back(')');
```

바로 수식 전체를 ()로 한 번 감싸는 것입니다. 그렇게 된다면 맨 마지막 괄호를 처리하면서 스택에 남아 있던 모든 연산자들이 `pop` 되겠지요.

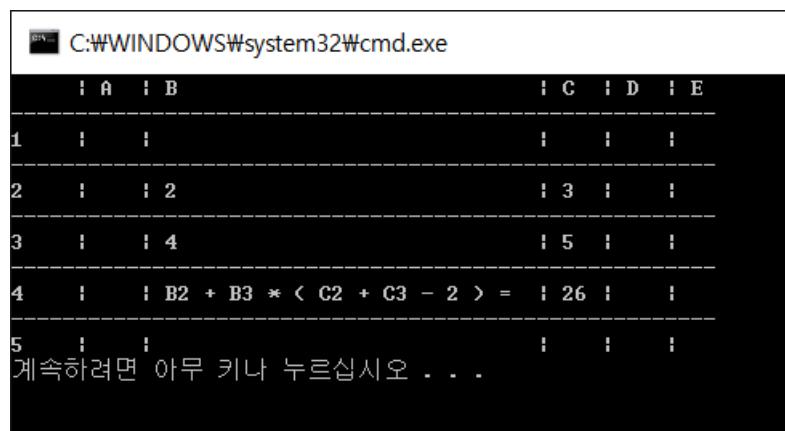
그렇다면 실제로 잘 작동하는지 살펴보도록 합시다.

```
// 생략
int main() {
    MyExcel::TxtTable table(5, 5);
    table.reg_cell(new NumberCell(2, 1, 1, &table), 1, 1);
    table.reg_cell(new NumberCell(3, 1, 2, &table), 1, 2);

    table.reg_cell(new NumberCell(4, 2, 1, &table), 2, 1);
    table.reg_cell(new NumberCell(5, 2, 2, &table), 2, 2);
    table.reg_cell(new ExprCell("B2+B3*(C2+C3-2)", 3, 3, &table), 3, 2);
    table.reg_cell(new StringCell("B2 + B3 * ( C2 + C3 - 2 ) = ", 3, 2, &table),
                  3, 1);

    std::cout << table;
}
```

성공적으로 컴파일 하였다면



와 같이 잘 작동하고 있음을 알 수 있습니다.

엑셀 프로그램

그렇다면 이제 실제로 사용자의 입력을 받아서 비록 마우스는 쓸 수 없더라도 키보드로 명령을 처리하는 엑셀 프로그램을 만들어보도록 하겠습니다.

```
class Excel {
    Table* current_table;

public:
    Excel(int max_row, int max_col, int choice);

    int parse_user_input(string s);
    void command_line();
};
```

위 클래스는 사용자의 입력을 받아서 실제 테이블을 생성하고 이를 관리해주는 클래스입니다. 또한 `parse_user_input` 함수의 경우 사용자의 입력을 인자로 받아서, 이를 처리하는 역할을 수행합니다.

```
Excel::Excel(int max_row, int max_col, int choice = 0) {
    switch (choice) {
        case 0:
            current_table = new TxtTable(max_row, max_col);
            break;
        case 1:
            current_table = new CSVTable(max_row, max_col);
            break;
        default:
            current_table = new HtmlTable(max_row, max_col);
    }
}
```

위는 Excel 객체의 생성자로 어떠한 형태의 테이블을 형성할 지 결정합니다.

```
int Excel::parse_user_input(string s) {
    int next = 0;
    string command = "";
    for (int i = 0; i < s.length(); i++) {
        if (s[i] == ' ') {
            command = s.substr(0, i);
            next = i + 1;
            break;
        } else if (i == s.length() - 1) {
            command = s.substr(0, i + 1);
            next = i + 1;
        }
    }
}
```

```
        break;
    }
}

string to = "";
for (int i = next; i < s.length(); i++) {
    if (s[i] == ' ' || i == s.length() - 1) {
        to = s.substr(next, i - next);
        next = i + 1;
        break;
    } else if (i == s.length() - 1) {
        to = s.substr(0, i + 1);
        next = i + 1;
        break;
    }
}

// Cell 이름으로 받는다.
int col = to[0] - 'A';
int row = atoi(to.c_str()) + 1;

string rest = s.substr(next);

if (command == "sets") {
    current_table->reg_cell(new StringCell(rest, row, col, current_table), row,
                           col);
} else if (command == "setn") {
    current_table->reg_cell(
        new NumberCell(atoi(rest.c_str()), row, col, current_table), row, col);
} else if (command == "setd") {
    current_table->reg_cell(new DateCell(rest, row, col, current_table), row,
                           col);
} else if (command == "sete") {
    current_table->reg_cell(new ExprCell(rest, row, col, current_table), row,
                           col);
} else if (command == "out") {
    ofstream out(to);
    out << *current_table;
    std::cout << to << "에 내용이 저장되었습니다" << std::endl;
} else if (command == "exit") {
    return 0;
}

return 1;
}
```

그리고 `parse_user_input` 함수는 사용자의 입력을 받아서 적절한 명령을 처리하게 됩니다.
예를 들어서

setn A1 10

이렇게 치면, A1 셀을 NumberCell로 생성하며, 10의 값으로 초기화 시켜줍니다.

혹은

sets B2 hello world!

의 경우 B2 셀을 StringCell로 생성하며 "hello world!"로 초기화 시켜줍니다.

날짜와 수식의 경우도 마찬가지이며, 각각 setd와 sete의 명령어를 사용하고 있습니다. 그 외에도, out을 통해서 원하는 파일에 출력할 수도 있고, exit를 하면 프로그램을 종료할 수 있습니다.

```
void Excel::command_line() {
    string s;
    std::getline(cin, s);

    while (parse_user_input(s)) {
        std::cout << *current_table << std::endl << ">> ";
        getline(cin, s);
    }
}
```

따라서 main 함수에서는

```
int main() {
    std::cout
        << "테이블 (타입) (최대 행 크기) (최대 열 크기) 를 순서대로 입력해주세요"
        << std::endl;
    std::cout << "* 참고 *" << std::endl;
    std::cout << "1 : 텍스트 테이블, 2 : CSV 테이블, 3 : HTML 테이블"
        << std::endl;

    int type, max_row, max_col;
    std::cin >> type >> max_row >> max_col;
    MyExcel::Excel m(max_row, max_col, type - 1);
    m.command_line();
}
```

마지막으로 command_line 함수는 사용자의 입력을 계속 기다리면서 내용이 업데이트 될 때마다 화면에 표를 출력해주게 됩니다. 실제로 사용하는 예시는 아래와 같습니다.

```

C:\WINDOWS\system32\cmd.exe
-----
5   |       |       |       |       |
>> sets A5 평균
| A       | B       | C       | D       | E
-----
1   | 이름    | 점수   |       |       |
-----  

2   | 흥길동  | 80     |       |       |
-----  

3   | 고길동  | 90     |       |       |
-----  

4   | 박길동  | 70     |       |       |
-----  

5   | 평균    |       |       |       |
-----  

>> sete B5 <B2+B3+B4>/3
| A       | B       | C       | D       | E
-----
1   | 이름    | 점수   |       |       |
-----  

2   | 흥길동  | 80     |       |       |
-----  

3   | 고길동  | 90     |       |       |
-----  

4   | 박길동  | 70     |       |       |
-----  

5   | 평균    | 80     |       |       |
-----
```

이상으로 위와 같이 나만의 미니 엑셀을 완성하였습니다!

사실 앞서 쪽 이야기 해 왔지만 제가 구현한 미니 엑셀은 코딩의 간소화를 위해서 몇 가지 제약들이 있습니다. 이러한 부분은 여러분들이 자유롭게 코딩하면서 더 확장 해 나가셨으면 좋겠습니다.

생각해보기

문제 1

ExprCell 의 죽에서 셀의 이름은 A3 과 같이 단 두 글자만 가능하다는 제약 조건이 있었습니다. 이를 임의의 크기의 이름도 가능하게 확장해보세요. (난이도 : 下)

문제 2

마찬가지로 가능한 숫자도 임의의 길이가 상관없게 확장해보세요. (난이도 : 下)

문제 3

사실 위와 같이 수식을 계산하는 경우 한 가지 문제가 있습니다. 바로 셀들이 서로를 참조할 수 있다는 것입니다. 예를 들어서 A1 = B1 이고 B1 = A1 으로 설정하였다면 B1 의 값을 알기 위해

A1의 값을 알아야 하고, 그럼 A1의 값을 알기 위해 B1의 값을 알아야 하고... 와 같은 순환 참조 문제가 발생합니다.

따라서 사용자가 타의든 자의든 순환 참조가 있는 식을 입력하였을 때 이를 감지하고 입력을 방지하는 루틴을 제공해야 합니다. (실제 Excel에서도 순환 참조되는 식을 입력하면 오류가 발생합니다) (난이도 : 上)

문제 4

실제 Excel의 경우 수식에서 여러가지 함수들을 지원합니다. 여기서도 수식에서 간단한 함수들을 지원하게 해보세요.. (난이도 : 上)

C++ 템플릿

안녕하세요 여러분! 지난번 강좌 생각해보기는 잘 해결 하셨나요? 뭐 해결 하지 못하고 넘어왔더라도 괜찮습니다. 이번 강좌에서는 여태까지 배워 왔던 것과 전혀 다른 새로운 세계를 탐험해 볼 것입니다.

지난번 강좌에서 벡터와 스택 자료형을 만드셨던 것을 기억 하시나요? 벡터(vector)는 쉽게 생각하면 가변 길이 배열로 사용자가 배열 처럼 사용하는데 크기를 마음대로 줄이고 늘릴 수 있는 편리한 자료형입니다. 스택(stack)의 경우 나중에 들어간 것이 먼저 나온다(*LIFO - last in first out*) 형태의 자료형으로 pop 과 push 를 통해서 여러가지 작업들을 처리할 수 있습니다.

하지만 한 가지 문제는 우리가 담으려고 하는 데이터 타입이 바뀔 때마다 다른 벡터를 만들어주어야 한다는 점이 있습니다. 예를 들어서 아래의 Vector 클래스를 살펴봅시다.

```
class Vector {
    std::string* data;
    int capacity;
    int length;

public:
    // 생성자
    Vector(int n = 1) : data(new std::string[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(std::string s) {
        if (capacity <= length) {
            std::string* temp = new std::string[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }

            delete[] data;
            data = temp;
            capacity *= 2;
        }
    }
}
```

```

    data[length] = s;
    length++;
}

// 임의의 위치의 원소에 접근한다.
std::string operator[](int i) { return data[i]; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

```

위 Vector 클래스의 경우 string 데이터 밖에 저장할 수 없습니다. 만일 사용자가 int 를 원한다면 어떨까요? 혹은 char 데이터를 담고 싶다면 어떻게 할까요? 물론, 각각의 경우에서 코드만 살짝 살짝 바꿔주면 될 것입니다. 예를 들어서 char 을 담는 Vector 의 경우

```

class Vector {
    char* data;
    int capacity;
    int length;

public:
    // 생성자
    Vector(int n = 1) : data(new char[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(char s) {
        if (capacity <= length) {
            char* temp = new char[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }
            delete[] data;
            data = temp;
            capacity *= 2;
        }
    }
};

```

```

    data[length] = s;
    length++;
}

// 임의의 위치의 원소에 접근한다.
char operator[](int i) { return data[i]; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

```

위와 같이 `string` 부분만 살짝 `char`로 바꿔주면 됩니다. 하지만 이는 큰 낭비가 아닐 수 없습니다. 위 `char`을 보관하는 `vector`와 `string`을 보관하는 `vector`의 코드는 단순히 `string` 부분이 `char`로 바뀐 차이 밖에 없습니다. 나머지 부분은 정확히 일치합니다.

쉽게 말해서 우리가 `T`라는 타입의 객체들을 보관하는 `vector`를 만들고 싶을 경우

```

class Vector {
    T* data;
    int capacity;
    int length;

public:
    // 생성자
    Vector(int n = 1) : data(new T[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(T s) {
        if (capacity <= length) {
            T* temp = new T[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }
            delete[] data;
            data = temp;
        }
    }
};

```

```

        capacity *= 2;
    }

    data[length] = s;
    length++;
}

// 임의의 위치의 원소에 접근한다.
T operator[](int i) { return data[i]; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

```

만약에 우리가 `int` 를 담는 `vector` 가 필요할 경우 `T` 를 `int` 로 바꿔면 되고, `string` 을 담는 것이 필요하다면 `T` 가 `string` 으로 바꿔면 됩니다. 쉽게 말해 컴파일러가 `T` 부분에 우리가 원하는 타입으로 채워서 코드를 생성해주면 엄청 편할 것입니다.

마치 어떠한 틀에 타입을 집어넣으면 원하는 코드가 딱딱 튀어 나오는 틀 같이 말이죠.

C++ 템플릿(template)

놀랍게도, 우리가 원하는 작업을 C++ 에서는 `template` 이라는 이름으로 지원하고 있습니다. 영어 단어 `template` 의 뜻을 찾아보자면, 형판 이라고 나옵니다. 쉽게 말해 어떠한 물건을 찍어내는 틀이라고 생각하면 됩니다.

C++ 에서도 정확히 같은 의미로 사용되고 있습니다. 사용자 (프로그래머)가 원하는 타입을 넣어 주면 딱딱 알아서 코드를 찍어내는 틀이라고 생각하면 됩니다 C++ 에서도 정확히 같은 의미로 사용되고 있습니다. 사용자 (프로그래머)가 원하는 타입을 넣어주면 딱딱 알아서 코드를 찍어내는 틀이라고 생각하면 됩니다.

```
// 템플릿 첫 활용
```

```
#include <iostream>
#include <string>

template <typename T>
class Vector {
    T* data;
    int capacity;
    int length;

public:
    // 생성자
    Vector(int n = 1) : data(new T[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(T s) {
        if (capacity <= length) {
            T* temp = new T[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }
            delete[] data;
            data = temp;
            capacity *= 2;
        }

        data[length] = s;
        length++;
    }

    // 임의의 위치의 원소에 접근한다.
    T operator[](int i) { return data[i]; }

    // x 번째 위치한 원소를 제거한다.
    void remove(int x) {
        for (int i = x + 1; i < length; i++) {
            data[i - 1] = data[i];
        }
        length--;
    }

    // 현재 벡터의 크기를 구한다.
    int size() { return length; }

    ~Vector() {
        if (data) {
            delete[] data;
        }
    }
};

int main() {
```

```
// int 를 보관하는 벡터를 만든다.
Vector<int> int_vec;
int_vec.push_back(3);
int_vec.push_back(2);

std::cout << "----- int vector -----" << std::endl;
std::cout << "첫번째 원소 : " << int_vec[0] << std::endl;
std::cout << "두번째 원소 : " << int_vec[1] << std::endl;

Vector<std::string> str_vec;
str_vec.push_back("hello");
str_vec.push_back("world");
std::cout << "----- std::string vector -----" << std::endl;
std::cout << "첫번째 원소 : " << str_vec[0] << std::endl;
std::cout << "두번째 원소 : " << str_vec[1] << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
----- int vector -----
첫번째 원소 : 3
두번째 원소 : 2
----- std::string vector -----
첫번째 원소 : hello
두번째 원소 : world
```

와 같이 나옵니다.

일단 클래스 상단에 템플릿이 정의된 부분을 살펴봅시다.

```
template <typename T>
class Vector {
    T* data;
    int capacity;
    // ...
```

맨 위에

`template <typename T>`

는 아래에 정의되는 클래스에 대해 템플릿을 정의하고, 템플릿 인자로 `T` 를 받게 되며, `T` 는 반드시 어떠한 타입의 이름임을 명시하고 있습니다. 위 경우 템플릿 문장 아래 오는 것이 `class Vector` 이므로 이 `Vector` 클래스에 대한 템플릿을 명시하는데, 만약에 밑에 오는 것이 함수일 경우 함수에 대한 템플릿이 됩니다. (밑에 함수 템플릿에 대해서도 설명하겠습니다).

참고로, 간혹

template <class T>

라고 쓰는 경우도 있는데, 이는 정확히 `typename T` 와 동일한 것 입니다. `class T` 라고 해서 T 자리에 꼭 클래스가 와야 하는 것이 아닙니다. 똑같이 `int`, `char` 등이 올 수 있습니다.

주의 사항

`template <typename T>` 와 `template <class T>` 는 정확히 같은 의미를 같지만 되도록이면 `typename` 키워드를 사용하기를 권장합니다.

왜 똑같은 템플릿에 두 개의 키워드를 정의하였냐였는지는 C++의 역사와 관련이 있습니다. C++을 처음 만들었던 Bjarne Stroustrup은 처음에 `template`의 인자로 `class` 키워드를 사용하였는데, 굳이 새로운 키워드를 만들고 싶지 않아서 였기 때문입니다.

하지만, 시간이 흘러서 C++ 위원회는 이로 인한 혼동을 막기 위해 (왜냐하면 `class T` 라 하면 T 자리에 꼭 클래스만 와야 하는 것처럼 느껴지니까) `typename`이라는 이름을 사용하기로 하였습니다. 물론, 이전 코드와의 호환을 위해 `class`는 그대로 남겨 놓았지요. (자세한 내막은 [여기](#) 참조)

이렇게 정의한 템플릿의 인자에 값을 전달하기 위해서는

```
Vector<int> int_vec;
```

와 같이, <> 안에 전달하려는 것을 명시해주면 됩니다. 우리의 `Vector` 템플릿은 템플릿 인자로 타입을 받게 되는데, 위 경우 T에 `int` 가 전달되게 됩니다.

여태까지는 인자로 특정한 '값' 혹은 '객체'를 전달해왔지만 '타입' 그 자체를 전달한 적은 없었습니다. 하지만 템플릿을 통해 타입을 전달할 수 있게 됩니다.

```
Vector<int> // 혹은  
Vector<std::string>
```

위와 같이 `Vector`의 템플릿의 인자에 타입을 전달하게 되면, 컴파일러는 이것을 보고 실제 코드를 생성하게 됩니다. 예를 들어서, `Vector <int>`의 경우

```
class Vector {  
    int* data;  
    int capacity;  
    int length;  
  
public:  
    // 어떤 타입을 보관하는지  
    typedef T value_type;
```

```
// 생성자
Vector(int n = 1) : data(new int[n]), capacity(n), length(0) {}

// 맨 뒤에 새로운 원소를 추가한다.
void push_back(int s) {
    if (capacity <= length) {
        int* temp = new int[capacity * 2];
        for (int i = 0; i < length; i++) {
            temp[i] = data[i];
        }
        delete[] data;
        data = temp;
        capacity *= 2;
    }

    data[length] = s;
    length++;
}

// 임의의 위치의 원소에 접근한다.
int operator[](int i) { return data[i]; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
```

T 가 정확히 int 로 치환된 위와 같은 코드를 써내겠지요. Vector<std::string> 의 경우 마찬가지로 T 가 string 으로 치환된 코드를 생성하게 됩니다.



위 사진처럼, `template`에 인자로 어떠한 타입을 전달하느냐에 따라 `int`를 저장하는 `Vector`를 만들거나, `string`을 저장하는 `Vector`를 만들어낼 수 있는 것입니다. 마치 틀에 쇠물을 넣으면 쇠로 된 물건이 나오고, 구리물을 넣으면 구리로 된 물건이 나오는 것처럼 말입니다.

```
Vector<int> int_vec;
```

따라서 위 코드는 `Vector`의 `T`가 `int`로 치환된 클래스의 객체 `int_vec`을 생성하게 되는 것이지요. 위와 같이 클래스 템플릿에 인자를 전달해서 실제 코드를 생성하는 것을 클래스 템플릿 인스턴스화 (class template instantiation)라고 합니다.

템플릿이 인스턴스화 되지 않고 놓그러니 있다면 컴파일 시에 아무런 코드로 변환되지 않습니다. 템플릿은 반드시 인스턴스화 되어야지만 비로소 컴파일러가 실제 코드를 생성하게 되지요. 마치 틀 자체로는 아무런 의미가 없지만, 그 틀에 채워넣어 나오는 물건에 관심이 있는 것처럼 말이지요.

위와 같이 템플릿에 사용자가 원하는 타입을 템플릿 인자로 전달하면, 컴파일러는 그 인자를 바탕으로 코드를 생성하여 이를 컴파일하게 됩니다. 하지만, 간혹 일부 타입에 대해서는 다른 방식으로 처리해야 할 경우가 있습니다.

예를 들어서 `bool` 데이터를 보관하는 벡터를 생각해봅시다.

```
Vector<bool> int_vec;
```

사실 쉽게 생각해보면 그냥 위처럼 템플릿 인자에 `bool`을 전달하여 `bool`을 저장하는 벡터로 사용할 수도 있을 것입니다. 하지만 문제가 하나 있는데, C++에서 기본으로 처리하는 단위가 1 byte라는 점입니다.

다시 말해 사실 `bool` 데이터 형은 1개 비트 만으로도 충분히 저장할 수 있지만, 8비트를 사용해서 1개 `bool` 값을 저장해야 된다는 뜻이지요. 이는 엄청난 메모리 낭비가 아닐 수 없습니다. 따라서 우리는 `Vector<bool>`에 대해서는 특별히 따로 처리해줘야만 합니다.

템플릿 특수화 (template specialization)

이와 같이 일부 경우에 대해서 따로 처리하는 것을 템플릿 특수화 라고 합니다. 템플릿 특수화는 다음과 같이 수행할 수 있습니다. 예를 들어서

```
template <typename A, typename B, typename C>
class test {};
```

위와 같은 클래스 템플릿이 정의되어 있을 때, "아 나는 A 가 int 고 C 가 double 일 때 따로 처리하고 싶어!" 면,

```
template <typename B>
class test<int, B, double> {};
```

와 같이 특수화 하고 싶은 부분에 원하는 타입을 전달하고 위에는 일반적인 템플릿을 쓰면 되겠지요. 만약에 B 조차도 특수화 하고 싶다면,

```
template <>
class test<int, int, double> {};
```

와 같이 써주면 됩니다. 한 가지 중요한 점은, 전달하는 템플릿 인자가 없더라도 특수화 하고 싶다면 `template<>` 라도 남겨줘야 된다는 점입니다. 그렇다면 우리의 `bool` 벡터의 경우;

```
template <>
class Vector<bool> {
    ... // 원하는 코드
}
```

와 같이 따로 처리해주면 되겠지요.

`Vector<bool>` 을 구현하기 위해서는 저는 평범한 `int` 배열을 이용할 것입니다. 1 개의 `int` 는 4 바이트 이므로, 32 개의 `bool` 데이터들을 한데 묶어서 저장할 수 있겠지요. 이를 통해서 원래 방식대로라면 `bool` 이 1 바이트로 저장되지만, 이렇게 하면 `bool` 을 1 비트로 정확히 표현할 수 있게 됩니다.



int 의 경우 32 비트 이므로, 1 개의 int 는 32 개의 bool 데이터를 보관할 수 있다.

따라서 N 번째 bool 데이터는 N / 32 번째 int 안에 저장되어 있게 됩니다.

이렇게 한데 둑어서 저장하면 메모리 관련 측면에서는 효율이 매우 높아지지만, 이를 구현하는데는 조금 더 복잡해집니다. 왜냐하면 `int` 데이터에서 정확히 한 비트의 정보만 뽑아서 보여주어야 하기 때문이지요. 예를 들어서 N 번째 `bool` 데이터는 $N / 32$ 번째 `int`에 들어가 있고, 그 안에서 정확히 $N \% 32$ 번째 비트가 됩니다.

이와 같은 내용으로 구현을 하면 다음과 같습니다.

```
#include <iostream>
#include <string>

template <typename T>
class Vector {
    T* data;
    int capacity;
    int length;

public:
    // 어떤 타입을 보관하는지

    typedef T value_type;

    // 생성자
    Vector(int n = 1) : data(new T[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(T s) {
        if (capacity <= length) {
            T* temp = new T[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }
            delete[] data;
            data = temp;
            capacity *= 2;
        }

        data[length] = s;
        length++;
    }

    // 임의의 위치의 원소에 접근한다.
    T operator[](int i) { return data[i]; }

    // x 번째 위치한 원소를 제거한다.
    void remove(int x) {
        for (int i = x + 1; i < length; i++) {
            data[i - 1] = data[i];
        }
        length--;
    }
}
```

```
// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

template <>
class Vector<bool> {
    unsigned int* data;
    int capacity;
    int length;

public:
    typedef bool value_type;

// 생성자
Vector(int n = 1)
    : data(new unsigned int[n / 32 + 1]), capacity(n / 32 + 1), length(0) {
    for (int i = 0; i < capacity; i++) {
        data[i] = 0;
    }
}

// 맨 뒤에 새로운 원소를 추가한다.
void push_back(bool s) {
    if (capacity * 32 <= length) {
        unsigned int* temp = new unsigned int[capacity * 2];
        for (int i = 0; i < capacity; i++) {
            temp[i] = data[i];
        }
        for (int i = capacity; i < 2 * capacity; i++) {
            temp[i] = 0;
        }
    }

    delete[] data;
    data = temp;
    capacity *= 2;
}

if (s) {
    data[length / 32] |= (1 << (length % 32));
}

length++;
}
```

```
// 임의의 위치의 원소에 접근한다.
bool operator[](int i) { return (data[i / 32] & (1 << (i % 32))) != 0; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        int prev = i - 1;
        int curr = i;

        // 만일 curr 위치에 있는 비트가 1 이라면
        // prev 위치에 있는 비트를 1 로 만든다.
        if (data[curr / 32] & (1 << (curr % 32))) {
            data[prev / 32] |= (1 << (prev % 32));
        }
        // 아니면 prev 위치에 있는 비트를 0 으로 지운다.
        else {
            unsigned int all_ones_except_prev = 0xFFFFFFFF;
            all_ones_except_prev ^= (1 << (prev % 32));
            data[prev / 32] &= all_ones_except_prev;
        }
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

int main() {
    // int 를 보관하는 벡터를 만든다.
    Vector<int> int_vec;
    int_vec.push_back(3);
    int_vec.push_back(2);

    std::cout << "----- int vector -----" << std::endl;
    std::cout << "첫번째 원소 : " << int_vec[0] << std::endl;
    std::cout << "두번째 원소 : " << int_vec[1] << std::endl;

    Vector<std::string> str_vec;
    str_vec.push_back("hello");
    str_vec.push_back("world");
    std::cout << "----- std::string vector -----" << std::endl;
    std::cout << "첫번째 원소 : " << str_vec[0] << std::endl;
    std::cout << "두번째 원소 : " << str_vec[1] << std::endl;

    Vector<bool> bool_vec;
```

```
bool_vec.push_back(true);
bool_vec.push_back(true);
bool_vec.push_back(false);
bool_vec.push_back(false);
bool_vec.push_back(false);
bool_vec.push_back(true);
bool_vec.push_back(false);

std::cout << "----- int vector -----" << std::endl;
for (int i = 0; i < int_vec.size(); i++) {
    std::cout << int_vec[i];
}
std::cout << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
----- int vector -----
첫번째 원소 : 3
두번째 원소 : 2
----- std::string vector -----
첫번째 원소 : hello
두번째 원소 : world
----- bool vector -----
11000101010101010
```

와 같이 나옵니다.

```
unsigned int* data;
int capacity;
int length;
```

먼저 `Vector<bool>` 의 멤버 변수들 부터 살펴봅시다. 일단, `int` 배열로 `unsigned int` 를 사용하기로 하였습니다. 굳이 `unsigned int` 를 사용한 이유는 그냥 `int` 를 사용했을 때 `shift` 연산 시에 귀찮은 문제가 발생할 수도 있기 때문입니다.

`capacity` 는 생성된 `data` 배열의 크기를 의미하고 `length` 는 실제 `bool` 데이터의 길이를 의미합니다. 즉 1 `capacity` 에 32 개의 `bool` 이 들어갈 수 있게 되는 것이지요. (`int` 가 32 비트라 생각하면)

```
// 맨 뒤에 새로운 원소를 추가한다.
void push_back(bool s) {
    if (capacity * 32 <= length) {
        unsigned int* temp = new unsigned int[capacity * 2];
        for (int i = 0; i < capacity; i++) {
            temp[i] = data[i];
        }

        for (int i = capacity; i < 2 * capacity; i++) {
            temp[i] = 0;
        }

        delete[] data;
        data = temp;
        capacity *= 2;
    }

    if (s) {
        data[length / 32] |= (1 << (length % 32));
    }
    length++;
}
```

다음으로 `push_back` 함수를 살펴보도록 합시다. 일단 윗 부분은 동일합니다. 만일 현재 길이가 `capacity` 를 초과하게 되면 현재 크기의 2 배로 새로 배열을 만들어서 복사하게 됩니다. 그리고 기본적으로 배열 전체를 0 으로 초기화 하기 때문에 기본적으로 `false` 로 설정되어 있다고 보시면 됩니다.

```
if (s) {
    data[length / 32] |= (1 << (length % 32));
}
```

따라서 위 처럼 만일 `true` 를 추가하였을 때에만 해당 비트를 `true` 로 바꿔주면 됩니다. 어떤 비트에 1 을 OR 연산하게 되면 그 비트는 무조건 1 이 됩니다. 그리고 0 을 OR 연산하게 되면 그 비트의 값은 그대로 보존이 되지요. 따라서 OR 연산은 특정 비트에만 선택적으로 1로 바꾸는데 매우 좋은 연산입니다. 아래 그림을 보면 이해가 잘 되실 것입니다. 주변 나머지 비트들의 값은 보존하면서 특정 비트만 1 로 바꿔줍니다.

```

0010010101
OR 000010000000
-----
001011010101

```

위 경우에도 $1 \ll (\text{length} \% 32)$ 을 통해서 정확히 $\text{length} \% 32$ 번째 비트만 1로 바꾼 뒤, OR 연산을 해서 해당 int의 자리를 true로 만들어 줄 수 있습니다.

```
// 임의의 위치의 원소에 접근한다.
bool operator[](int i) { return (data[i / 32] & (1 << (i % 32))) != 0; }
```

그렇다면 임의의 위치에 접근하는 것은 어떨까요.

이번에는 거꾸로 AND 연산을 통해 해당 값을 가져올 수 있습니다. 어떤 비트에 0을 AND 하게 되면 그 비트는 무조건 0이 되고, 1을 AND 연산 하게 되면 해당 비트가 1이여야지만 1이 됩니다. 즉, 1을 AND 하게 되면 해당 비트의 값이 무엇인지 알아낼 수 있는 것입니다. 아래 그림을 보면 더 이해하기 쉽습니다.

```

0010010101
AND 000001000000
-----
000001000000

```

```
data[i / 32] & (1 << (i % 32))
```

따라서 위 작업을 수행하게 되면 해당 위치에 있는 비트가 1일 때 예만 저 값이 0이 아니게 되고 0이면 저 값 전체가 0이 됩니다.

```
// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        int prev = i - 1;
```

```

int curr = i;

// 만일 curr 위치에 있는 비트가 1 이라면
// prev 위치에 있는 비트를 1 로 만든다.
if (data[curr / 32] & (1 << (curr % 32))) {
    data[prev / 32] |= (1 << (prev % 32));
}
// 아니면 prev 위치에 있는 비트를 0 으로 지운다.
else {
    unsigned int all_ones_except_prev = 0xFFFFFFFF;
    all_ones_except_prev ^= (1 << (prev % 32));
    data[prev / 32] &= all_ones_except_prev;
}
length--;
}
}

```

마지막으로 `remove` 함수를 살펴봅시다.

사실 이전 버전에서는 그냥 오른쪽 원소를 왼쪽으로 쭈르륵 덮어 씌어나가는 방식으로 간단히 만들 수 있었지만 이 `bool` 버전의 경우 그 '왼쪽' 과 '오른쪽' 원소를 읽어내는거 자체가 꽤나 복잡하므로 간단하게는 구현하기 힘듭니다.

```

// 만일 curr 위치에 있는 비트가 1 이라면
// prev 위치에 있는 비트를 1 로 만든다.
if (data[curr / 32] & (1 << (curr % 32))) {
    data[prev / 32] |= (1 << (prev % 32));
}

```

그래도 일단 특정 비트를 1 로 만드는 것은 간단합니다. 이전에도 하였듯이, OR 을 수행해주면 됩니다.

```

// 아니면 prev 위치에 있는 비트를 0 으로 지운다.
else {
    unsigned int all_ones_except_prev = 0xFFFFFFFF;
    all_ones_except_prev ^= (1 << (prev % 32));
    data[prev / 32] &= all_ones_except_prev;
}

```

그렇다면 오른쪽 비트가 0 일 때 왼쪽 비트를 0 으로 만드는 작업은 어떻게 할까요? 앞서 AND 연산의 경우 1 을 AND 하게 되면 원래 비트가 보존되고, 0 을 AND 하면 0 이 된다고 하였습니다. 따라서, 우리가 해야할 일은 원하는 자리의 비트만 0 이고 나머지는 1 인 것을 AND 해주면 됩니다. 그 과정을 수행하는 부분이 아래와 같습니다.

```
unsigned int all_ones_except_prev = 0xFFFFFFFF;
all_ones_except_prev ^= (1 << (prev % 32));
```

`all_ones_except_prev` 는 0 으로 지우려고 하는 위치만 0 이고 나머지 부분은 1 인 값입니다. 참고로 기억하는지는 모르겠지만, ^는 XOR 연산자로, 두 비트가 같으면 0, 다르면 1 이되는 연산입니다 (덧셈을 생각하시면 됩니다). 따라서 `0xFFFFFFFF` 는 모든 비트가 1 인 `int` 인데, XOR 연산을 통해서 해당 비트만 0 으로 바꿔줍니다.

그 후에 AND 연산을 하게 되면 해당 비트만 0 으로 지워버릴 수 있게 되지요.

실제로, C++ 표준 라이브러리에서 `vector` 를 지원을 하는데, 그 구현을 살펴보면 위처럼 `bool` 일 때만 따로 특수화 시켜서 처리하고 있습니다. C++ 표준 라이브러리의 `vector` 는 나중 강좌에서 다루도록 하겠습니다.

템플릿 특수화를 통해 템플릿 인자의 값이 특정 값일 때 따로 처리하는 방식에 대해 알아보았습니다.

함수 템플릿 (Function template)

이번에는 클래스 템플릿이 아닌 함수 템플릿에 대해 알아보도록 하겠습니다.

```
#include <iostream>
#include <string>

template <typename T>
T max(T& a, T& b) {
    return a > b ? a : b;
}

int main() {
    int a = 1, b = 2;
    std::cout << "Max (" << a << "," << b << ") ? : " << max(a, b) << std::endl;

    std::string s = "hello", t = "world";
    std::cout << "Max (" << s << "," << t << ") ? : " << max(s, t) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Max (1,2) ? : 2
Max (hello,world) ? : world
```

와 같이 나옵니다.

```
template <typename T>
T max(T& a, T& b) {
    return a > b ? a : b;
}
```

위 부분에서 템플릿 함수를 정의하고 있습니다. 클래스 템플릿과 마찬가지로, 위 함수도 인스턴스화 되기 전 까지는 컴파일 시에 아무런 코드로 변환되지 않습니다.

```
std::cout << "Max (" << a << "," << b << ") ? : " << max(a, b) << std::endl;
```

실제로 위 템플릿 함수가 인스턴스화 되는 부분은 바로 위 코드에서 `max(a, b)` 가 호출되는 부분입니다. 신기하게도, 클래스를 인스턴스화 했을 때 와는 다르게 `<>` 하는 부분이 없습니다. 원래 대로라면

```
max<int>(a, b)
```

이렇게 했었겠지요. 하지만 C++ 컴파일러는 생각보다 똑똑해서, `a` 와 `b` 의 타입을 보고 알아서 `max (a, b)` 를 `max<int>(a, b)` 로 인스턴스화 해줍니다.

```
std::cout << "Max (" << s << "," << t << ") ? : " << max(s, t) << std::endl;
```

`string`의 경우도 마찬가지입니다. C++ 컴파일러가 알아서 `max<string>(s, t)` 로 생각해서 인스턴스화 해줍니다.

그렇다면 다음 함수를 살펴봅시다.

```
template <typename Cont>
void bubble_sort(Cont& cont) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (cont[i] > cont[j]) {
                cont.swap(i, j);
            }
        }
    }
}
```

위 함수는 임의의 컨테이너를 받아서 이를 정렬해 주는 함수입니다 (사실 위 함수는 소위 말하는 거품 정렬 (버블소트) 방식을 사용하는데 매우 느립니다. 여러분들이 더 빠른 정렬 알고리즘(퀵소트,

등등)으로 구현해보세요!). 컨테이너란 쉽게 생각해서 데이터를 보관하는 것이라 생각하면 되는데, 앞서 소개하였던 `Vector` 도 하나의 예가 될 수 있습니다.

물론, 저 함수가 작동을 하려면 컨테이너에 `size` 함수와, `swap` 함수, 그리고 `[]` 연산자가 정의되어 있어야 겠지요.

```
#include <iostream>

template <typename T>
class Vector {
    T* data;
    int capacity;
    int length;

public:
    // 어떤 타입을 보관하는지
    typedef T value_type;

    // 생성자
    Vector(int n = 1) : data(new T[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(int s) {
        if (capacity <= length) {
            int* temp = new T[capacity * 2];
            for (int i = 0; i < length; i++) {
                temp[i] = data[i];
            }
            delete[] data;
            data = temp;
            capacity *= 2;
        }

        data[length] = s;
        length++;
    }

    // 임의의 위치의 원소에 접근한다.
    T operator[](int i) { return data[i]; }

    // x 번째 위치한 원소를 제거한다.
    void remove(int x) {
        for (int i = x + 1; i < length; i++) {
            data[i - 1] = data[i];
        }
        length--;
    }

    // 현재 벡터의 크기를 구한다.
    int size() { return length; }
```

```
// i 번째 원소와 j 번째 원소 위치를 바꾼다.
void swap(int i, int j) {
    T temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

template <typename Cont>
void bubble_sort(Cont& cont) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (cont[i] > cont[j]) {
                cont.swap(i, j);
            }
        }
    }
}

int main() {
    Vector<int> int_vec;
    int_vec.push_back(3);
    int_vec.push_back(1);
    int_vec.push_back(2);
    int_vec.push_back(8);
    int_vec.push_back(5);
    int_vec.push_back(3);

    std::cout << "정렬 이전 ---- " << std::endl;
    for (int i = 0; i < int_vec.size(); i++) {
        std::cout << int_vec[i] << " ";
    }

    std::cout << std::endl << "정렬 이후 ---- " << std::endl;
    bubble_sort(int_vec);
    for (int i = 0; i < int_vec.size(); i++) {
        std::cout << int_vec[i] << " ";
    }
    std::cout << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
정렬 이전 ----
```

```
3 1 2 8 5 3
```

```
정렬 이후 ----
```

```
1 2 3 3 5 8
```

와 같이 나옵니다.

참고로 기존에 만들었던 `Vector`에는 `swap` 함수가 없어서 새로 추가하였습니다.

```
template <typename Cont>
void bubble_sort(Cont& cont) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (cont[i] > cont[j]) {
                cont.swap(i, j);
            }
        }
    }
}
```

위 부분을 보면 정렬 함수를 템플릿으로 구현한 것을 볼 수 있습니다.

```
bubble_sort(int_vec);
```

와 같이 위 함수를 호출한다면 컴파일러는 인자로 전달된 객체의 타입을 보고 알아서 인스턴스화 한 뒤에 컴파일하게 되지요. 위 경우에 `int_vec` 이 `Vector<int>` 타입 이므로, `Cont`에 `Vector<int>` 가 전달 될 것입니다.

다만 한 가지 중요한 사실은,

```
for (int i = 0; i < cont.size(); i++) {
```

에서나,

```
if (cont[i] > cont[j]) {
    cont.swap(i, j);
```

에서 `size()`, `operator[]`, `swap()` 등이 사용되었다는 것입니다. 만약에 `Cont`로 전달된 타입에 저러한 것들이 정의가 되어 있지 않다면 어떨까요? 예를 들어서

```
struct dummy {};
```

dummy a; bubble_sort(a);
를 하게 된다면

컴파일 오류

```
error C2039: 'size': is not a member of 'dummy'  
...
```

컴파일 시에, 위와 같은 저 클래스에서 멤버 함수나 변수들을 찾을 수 없다는 오류들을 봄아내게 됩니다. 당연한 이야기지만 이와 같이 템플릿으로 발생되는 오류는 프로그램이 실행되었을 때가 아니라 컴파일 할 때 발생한다는 사실입니다. 왜냐하면 컴파일 시에 모든 템플릿을 실제 코드로 변환하여 실행하기 때문이지요.

재미있게도, 이와 같이 컴파일 시에 모든 템플릿들이 인스턴스화 되다는 사실을 가지고 또 여러가지 흥미로운 코드를 짤 수 있는데 이러한 방식을 템플릿 메타프로그래밍 (template metaprogramming) 이라고 합니다. 자세한 내용은 나중 강좌들에서 다루도록 하겠습니다.

그런데 위 bubble_sort 함수에서는 한 가지 부족한 점이 있습니다. 만약에, 정렬 순서를 역순으로 하고 싶다면 어떨까요? 위는 오름 차순으로 정렬하였지만, 간혹 내림 차순으로 정렬을 하고 싶을 수도 있습니다. 아니면 아예 다른 기준으로 정렬을 할 수도 있겠지요. 이를 위해서라면 크게 세 가지 방법이 있습니다.

1. bubble_sort2 를 만들어서 부등호 방향을 반대로 바꿔준다.
2. operator> 를 오버로딩해서 원하는 방식으로 만들어준다.
3. cont[i] 와 cont[j] 의 비교를 > 로 하지 말고 특정 함수에 넣어서 전달한다.

첫번째 방법은 C 를 배우는 단계에서나 적합한 방법입니다. 여러분은 C++ 를 배우고 있으니 더 나은 방법을 생각해야겠지요? 두번째 방법은 여러분들이 만든 객체를 사용할 때 적용할 수 있는 방법입니다. 예를 들어서,

```
struct customClass { // ..  
bool operator<(const customClass& c) { // Do something } };  
Vector<customClass> a; bubble_sort(a);
```

와 같이 여러분이 직접 정의한 클래스에 대해 operator< 를 오버로딩 할 수 있다면 원하는 방식으로 정렬을 수행할 수 있겠지요. 하지만 위처럼 기본적으로 operator< 를 오버로딩 할 수 없는 상황이라면 어떨까요? 예를 들어서 int 나 string 은 이미 내부적으로 operator< 가 정의되어 있기 때문에 이를 따로 오버로딩을 할 수 없습니다.

그렇다면 3 번째 방법은 어떨까요?

함수 객체(Function Object - Functor) 의 도입

그럼 다음과 같은 함수를 생각해봅시다.

```
template <typename Cont, typename Comp>

void bubble_sort(Cont& cont, Comp& comp) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (!comp(cont[i], cont[j])) {
                cont.swap(i, j);
            }
        }
    }
}
```

위 함수는 기존의 `bubble_sort` 와는 달리 아예 `Comp` 라는 클래스를 템플릿 인자로 받고, 함수 자체도 `Comp` 객체를 따로 받습니다. 그렇다면 이 `comp` 객체가 무슨 일을 하냐면;

```
if (!comp(cont[i], cont[j])) {
```

이 `if` 문에서 마치 함수를 호출하는 것 처럼 사용되는데, `cont[i]` 와 `cont[j]` 를 받아서 내부적으로 크기 비교를 수행한 뒤에 그 결과를 리턴하고 있습니다.

한 가지 중요한 사실은 `comp` 는 함수가 아니라 객체이고, `Comp` 클래스에서 `()` 연산자를 오버로딩한 버전입니다. 자세한 내용은 아래 전체 코드를 보면서 설명하겠습니다.

```
#include <iostream>

template <typename T>
class Vector {
    T* data;
    int capacity;
    int length;

public:
    // 어떤 타입을 보관하는지
    typedef T value_type;

    // 생성자
    Vector(int n = 1) : data(new T[n]), capacity(n), length(0) {}

    // 맨 뒤에 새로운 원소를 추가한다.
    void push_back(int s) {
        if (capacity <= length) {
            int* temp = new T[capacity * 2];
```

```
    for (int i = 0; i < length; i++) {
        temp[i] = data[i];
    }
    delete[] data;
    data = temp;
    capacity *= 2;
}

data[length] = s;
length++;
}

// 임의의 위치의 원소에 접근한다.
T operator[](int i) { return data[i]; }

// x 번째 위치한 원소를 제거한다.
void remove(int x) {
    for (int i = x + 1; i < length; i++) {
        data[i - 1] = data[i];
    }
    length--;
}

// 현재 벡터의 크기를 구한다.
int size() { return length; }

// i 번째 원소와 j 번째 원소 위치를 바꾼다.
void swap(int i, int j) {
    T temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}

~Vector() {
    if (data) {
        delete[] data;
    }
}
};

template <typename Cont>
void bubble_sort(Cont& cont) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (cont[i] > cont[j]) {
                cont.swap(i, j);
            }
        }
    }
}
```

```
template <typename Cont, typename Comp>
void bubble_sort(Cont& cont, Comp& comp) {
    for (int i = 0; i < cont.size(); i++) {
        for (int j = i + 1; j < cont.size(); j++) {
            if (!comp(cont[i], cont[j])) {
                cont.swap(i, j);
            }
        }
    }
}

struct Comp1 {
    bool operator()(int a, int b) { return a > b; }
};

struct Comp2 {
    bool operator()(int a, int b) { return a < b; }
};

int main() {
    Vector<int> int_vec;
    int_vec.push_back(3);
    int_vec.push_back(1);
    int_vec.push_back(2);
    int_vec.push_back(8);
    int_vec.push_back(5);
    int_vec.push_back(3);

    std::cout << "정렬 이전 ---- " << std::endl;
    for (int i = 0; i < int_vec.size(); i++) {
        std::cout << int_vec[i] << " ";
    }

    Comp1 comp1;
    bubble_sort(int_vec, comp1);

    std::cout << std::endl << std::endl << "내림차순 정렬 이후 ---- " << std::endl;
    for (int i = 0; i < int_vec.size(); i++) {
        std::cout << int_vec[i] << " ";
    }
    std::cout << std::endl;

    Comp2 comp2;
    bubble_sort(int_vec, comp2);

    std::cout << std::endl << "오름차순 정렬 이후 ---- " << std::endl;
    for (int i = 0; i < int_vec.size(); i++) {
        std::cout << int_vec[i] << " ";
    }
    std::cout << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

정렬 이전 ----

3 1 2 8 5 3

내림차순 정렬 이후 ----

8 5 3 3 2 1

오름차순 정렬 이후 ----

1 2 3 3 5 8

와 같이 나옵니다.

```
struct Comp1 {
    bool operator()(int a, int b) { return a > b; }
};

struct Comp2 {
    bool operator()(int a, int b) { return a < b; }
};
```

일단 위 두 클래스를 살펴보도록 합시다. Comp1 과 Comp2 모두 아무 것도 하지 않고 단순히 operator() 만 정의하고 있습니다. 그리고 이 Comp1 과 Comp2 객체들은 bubble_sort 함수 안에서

```
if (!comp(cont[i], cont[j])) {
```

마치 함수인양 사용되지요. 이렇게, 함수는 아니지만 함수 인 척을 하는 객체를 함수 객체 (Function Object), 혹은 줄여서 **Functor** 라고 부릅니다. 이 Functor 덕분에, bubble_sort 함수 내에서 두 객체간의 비교를 사용자가 원하는 대로 할 수 있게 되지요.

따라서 사용자들은 입맛에 맞게, 보통의 < 연산자로 비교를 수행하는

```
template <typename Cont>
void bubble_sort(Cont& cont)
```

위 bubble_sort 함수를 사용하거나;

```
template <typename Cont, typename Comp>
void bubble_sort(Cont& cont, Comp& comp)
```

특수한 경우에 따로 비교하는 것을 직접 수행하고 싶은 경우에 **Comp** 객체를 받아서 비교를 수행하는 새로운 **bubble_sort** 함수를 사용할 수 있습니다.

물론

```
bubble_sort(int_vec, comp1);
```

를 했다면 두 번째 버전으로 템플릿 인스턴스화 되서 함수가 호출되고,

```
bubble_sort(int_vec);
```

그냥 위처럼 한다면 첫 번째 버전으로 템플릿 인스턴스화 되서 함수가 호출되겠지요.

실제로, C++ 표준 라이브러리의 **sort** 함수를 살펴보아도 비교 클래스를 받지 않는

```
template <class RandomIt> void sort(RandomIt first, RandomIt last);
```

와

```
template <class RandomIt, class Compare> void sort(RandomIt first, RandomIt last, Compare comp);
```

비교 클래스를 받는 위 버전으로 구성되어 있습니다. (저 함수의 인자들에 대해서는 나중 강좌에서 자세히 다룰 테니 지금은 넘어가셔도 됩니다!)

이와 같이 비교 클래스를 받아서 원하는 비교 작업을 수행할 수 있게 됩니다. 만약에 C 였다면 위 **sort** 함수를 어떻게 만들었을 지 생각해봅시다. 일단 원하는 클래스를 받는다는 생각 자체가 불가능하기 때문에 (**template** 이 없으니까) **functor** 는 꿈도 못 꾸었겠지요. 대신에, 비교 작업을 수행하는 함수의 포인터를 받아서 이를 처리하였을 것입니다.

그렇다면 뭐가 더 나은 방법일까요? **Functor**? 아니면 구닥다리 함수 포인터?

이미 예상하셨겠지만, **Functor** 를 사용하는것이 여러 모로 훨씬 편리한 점이 많습니다. 일단, 클래스 자체에 여러가지 내부 **state** 를 저장해서 비교 자체가 복잡한 경우에도 손쉽게 사용자가 원하는 방식으로 만들어낼 수 있습니다. 뿐만 아니라, 함수포인터로 함수를 받아서 처리한다면 컴파일러가 최적화를 할 수 없지만, **Functor** 를 넘기게 된다면 컴파일러가 **operator()** 자체를 인라인화 시켜서 매우 빠르게 작업을 수행할 수 있습니다.¹⁾

1) 실제로 C 의 **qsort** 와 C++ 의 표준 **sort** 함수를 비교한다면 C++ 버전이 훨씬 빠릅니다. 왜냐하면 C 의 **qsort** 는 비교를 수행하기 위해 매번 함수를 호출시켜야 하지만, C++ 버전의 경우 그 함수를 인라인화 시켜버리면 되기 때문이지요. (함수 호출 필요 없음)

타입이 아닌 템플릿 인자 (non-type template arguments)

한 가지 재미있는 점은 템플릿 인자로 타입만 받을 수 있는 것이 아닙니다. 예를 들어 아래와 같은 코드를 보실까요.

```
#include <iostream>

template <typename T, int num>
T add_num(T t) {
    return t + num;
}

int main() {
    int x = 3;
    std::cout << "x : " << add_num<int, 5>(x) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

x : 8

와 같이 나옵니다. 먼저 `add_num` 함수의 정의 부분부터 살펴봅시다.

```
template <typename T, int num>
T add_num(T t) {
    return t + num;
}
```

`template` 의 인자로 `T` 를 받고, 추가적으로 마치 함수의 인자처럼 `int num` 을 또 받고 있습니다. 해당 템플릿 인자들은 `add_num` 함수를 호출할 때 `<>` 를 통해 전달하는 인자들이 들어가게 됩니다. 우리의 예제의 경우

`add_num<int, 5>(x)`

위와 같이 `T` 에 `int` 를, `num` 에 `5` 를 전달하였으므로 생성되는 `add_num` 함수는 아래와 같습니다.

```
int add_num(int t) { return t + 5; }
```

참고로 만약에 `add_num` 에 템플릿 인자 `<>` 를 지정하지 않았더라면 아래와 같은 컴파일 타임 오류가 발생하게 됩니다.

컴파일 오류

```
test2.cc: In function ‘int main()’:
test2.cc:10:35: error: no matching function for call to
  → ‘add_num(int& )’
    std::cout << "x : " << add_num(x) << std::endl;
                                         ^
test2.cc:4:3: note: candidate: template<class T, int num> T
  → add_num(T)
    T add_num(T t) {
      ^~~~~~
test2.cc:4:3: note:   template argument deduction/substitution
  → failed:
test2.cc:10:35: note:   couldn't deduce template parameter ‘num’
    std::cout << "x : " << add_num(x) << std::endl;
```

왜냐하면 상식적으로 컴파일러가 `num` 에 뭐가 들어가는지 알길이 없이 때문이죠. 따라서 위처럼 `num` 의 값을 결정할 수 없다고 불만을 제시하는 오류가 발생하게 됩니다.

한 가지 중요한 점은 템플릿 인자로 전달할 수 있는 타입들이 아래와 같이 제한적입니다. (자세한 내용은 [여기 참조](#))²⁾

- 정수 타입들 (`bool, char, int, long` 등등). 당연히 `float` 과 `double` 은 제외
- 포인터 타입
- `enum` 타입
- `std::nullptr_t` (널 포인터)

타입이 아닌 템플릿 인자를 가장 많이 활용하는 예시는 컴파일 타입에 값들이 정해져야 하는 것들이 되겠습니다. 대표적인 예시로 배열을 들 수 있겠지요. C 에서의 배열의 가장 큰 문제점은 함수에 배열을 전달할 때 배열의 크기에 대한 정보를 잃어버린다는 점입니다.

하지만 템플릿 인자로 배열의 크기를 명시한다면 (어차피 배열의 크기는 컴파일 타임에 정해지는 것이니까), 이 문제를 완벽하게 해결 할 수 있습니다. 이와 같은 기능을 가진 배열을 C++ 11 부터 제공되는 `std::array` 를 통해 사용할 수 있습니다.

2) C++ 20 부터 이 제한이 좀 더 완화되었습니다.

```
#include <array>
#include <iostream>

int main() {
    // 마치 C 에서의 배열처럼 {} 를 통해 배열을 정의할 수 있다.
    // {} 문법은 16 - 1 강에서 자세히 다루므로 여기서는 그냥 이렇게
    // 쓰면 되는구나 하고 이해하고 넘어가면 됩니다.
    std::array<int, 5> arr = {1, 2, 3, 4, 5};
    // int arr[5] = {1, 2, 3, 4, 5}; 와 동일

    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

성공적으로 컴파일 하였으면

실행 결과

1 2 3 4 5

와 같이 나옵니다. 사용법은 C 에서의 배열과 동일합니다.

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

위처럼 배열의 원소들의 타입과 (int) 크기 (5) 를 템플릿 인자로 명시한 뒤에, 초기화만 해주면 됩니다. 그리고 마치 C 에서 배열을 정의할 때처럼 {} 를 이용해서 생성하면 됩니다. 참고로 {} 는 유니폼 초기화(uniform initialization) 이라 불리는 C++ 11 에서 추가된 개념인데, 지금은 그냥 std::array 의 생성자를 호출하는 또 하나의 방법이라 생각하시면 되고 나중에 16 - 1 강에서 좀 더 자세히 다룹니다.

한 가지 재미있는 점은 이 arr 은 런타임에서 동적으로 크기가 할당되는 것이 아니라는 점입니다. 마치 배열처럼 컴파일 시에 int 5 개를 가지는 메모리를 가지고 스택에 할당됩니다.

또한 중요한 점으로 이 배열을 함수에 전달하기 위해서는 그냥 std::array 를 받는 함수를 만들면 안됩니다. std::array<int, 5> 자체가 하나의 타입이기 때문에;

```
#include <array>
#include <iostream>

void print_array(const std::array<int, 5>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
```

```

    }
    std::cout << std::endl;
}

int main() {
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    print_array(arr);
}

```

성공적으로 컴파일 하였다면

실행 결과

1 2 3 4 5

와 같이 나옵니다.

```
void print_array(const std::array<int, 5>& arr) {
```

문제는 각 array 크기 별로 함수를 만들어줘야 합니다. 하지만 우리는 알고 있지요. 여기서도 역시 템플릿을 쓸 수 있다는 것을 말이죠. 따라서 그냥

```

#include <array>
#include <iostream>

template <typename T>
void print_array(const T& arr) {
    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::array<int, 5> arr = {1, 2, 3, 4, 5};
    std::array<int, 7> arr2 = {1, 2, 3, 4, 5, 6, 7};
    std::array<int, 3> arr3 = {1, 2, 3};

    print_array(arr);
    print_array(arr2);
    print_array(arr3);
}

```

성공적으로 컴파일 하였다면

실행 결과

```
1 2 3 4 5
1 2 3 4 5 6 7
1 2 3
```

와 같이 잘 나옵니다.

디폴트 템플릿 인자

마지막으로 살펴볼 점은 함수에 디폴트 인자를 지정할 수 있는 것처럼 템플릿도 디폴트 인자를 지정 할 수 있습니다. 예를 들어서 이전에 만들었던 `add_num` 함수를 다시 살펴봅시다.

```
#include <iostream>

template <typename T, int num = 5>
T add_num(T t) {
    return t + num;
}

int main() {
    int x = 3;
    std::cout << "x : " << add_num(x) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
x : 8
```

와 같이 나옵니다. 템플릿 디폴트 인자는 함수 디폴트 인자랑 똑같이 인자 뒤에 `= (디폴트 값)` 을 넣어주면 됩니다.

```
template <typename T, int num = 5>
```

위 경우 `num`에 디폴트로 5가 전달됩니다. 따라서

```
std::cout << "x : " << add_num(x) << std::endl;
```

위 문장은 마치 `add_num<int, 5>` 를 한 것과 동일합니다. (참고로 `int` 는 컴파일러가 자동으로 추론해주었습니다.)

타입 역시 디폴트로 지정이 가능합니다. 예를 들어서 아래와 같은 `min` 함수를 생각해봅시다.

```
template <typename T, typename Comp>
T Min(T a, T b) {
    Comp comp;
    if (comp(a, b)) {
        return a;
    }
    return b;
}
```

`Min` 함수는 임의의 두 원소를 받아서 작은 원소를 리턴합니다. 이 때 이 원소들을 어떻게 비교할지는 `Comp` 라는 객체가 이를 수행합니다.

물론 우리는 `int` 와 같이 간단한 애들은 그냥 `<` 를 사용해서 대소 비교를 하면 되지만 일반적인 객체들에 대해서도 모두 동작하게 하려면 따로 두 원소를 비교하는 `Comp` 라는 클래스가 필요합니다.

예를 들어서 `int` 간의 대소 비교를 위해서는

```
template <typename T>
struct Compare {
    bool operator()(const T& a, const T& b) const { return a < b; }
};

int a = 3, b = 4;
std::cout << "min : " << Min<int, Compare<int>>(a, b);
```

와 같이 `Compare` 타입을 굳이 써서 전달해줘야 하지요. 하지만 디폴트 생성자를 이용하면 아래처럼 매우 간단하게 사용할 수 있게 됩니다.

```
#include <iostream>
#include <string>

template <typename T>
struct Compare {
    bool operator()(const T& a, const T& b) const { return a < b; }
};

template <typename T, typename Comp = Compare<T>>
T Min(T a, T b) {
    Comp comp;
    if (comp(a, b)) {
        return a;
    }
}
```

```

return b;
}

int main() {
    int a = 3, b = 5;
    std::cout << "Min " << a << " , " << b << " :: " << Min(a, b) << std::endl;

    std::string s1 = "abc", s2 = "def";
    std::cout << "Min " << s1 << " , " << s2 << " :: " << Min(s1, s2)
        << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Min 3 , 5 :: 3
Min abc , def :: abc

```

와 같이 잘 나옵니다. 이들 모두 `Comp`로 디폴트 타입인 `Compare<T>`가 전달되어서 `<`를 통해 비교를 수행하였습니다.

이번 강좌는 내용이 상당히 길었는데요, 이상으로 C++ 템플릿 첫 번째 강좌를 마치도록 하겠습니다. 그 만큼 템플릿의 위력이 강력하다는 뜻이지요. 다음 강좌에서는 템플릿이 C++에게 하사한 새로운 패러다임의 세계로 떠나볼 것입니다.

생각 해보기

문제 1

`template` 을 사용해서 이전에 만들어 놓았던 Excel 프로젝트 코드를 깔끔하게 만들어보세요. 아마 10 배는 깔끔해질 것입니다 :) (난이도 : 下)

문제 2

위 `Vector`로 2차원, 3차원 배열 등을 똑같이 만들어낼 수 있을까요? (난이도 : 下)

문제 3

위에서 컴파일러가 마법처럼 템플릿 인자에 타입을 정해준다고 하지만 사실은 어떤 타입으로 추측할지 결정하는 일련의 규칙들이 있습니다. [자세한 내용은 여기를 참고해주세요](#)

가변 길이 템플릿

안녕하세요 여러분. 지난번 강좌에서 다룬 템플릿은 어땠셨나요? 템플릿을 잘 사용한다면 써야 되는 코드의 양을 비약적으로 줄일 수 있습니다. 이번 강좌는 그 연장선으로써, 템플릿을 사용해서 임의의 개수의 인자를 받는 방법에 대해서 이야기 해보도록 할 것입니다.

가변 길이 템플릿

파이썬을 써보신 분들은 아시겠지만 파이썬의 경우 아래와 같이 `print` 함수를 이용하면 인자로 전달된 것들을 모두 출력할 수 있습니다.

```
print(1, 3.1, "abc")
print("adfasf", var)
```

그렇다면 C++에서도 이와 같은 기능을 구현할 수 있을까요? 재미있게도 C++ 템플릿을 이용하면 임의의 개수의 인자를 받는 함수를 구현할 수 있습니다. 바로 아래 예제를 보시지요.

```
#include <iostream>

template <typename T>
void print(T arg) {
    std::cout << arg << std::endl;
}

template <typename T, typename... Types>
void print(T arg, Types... args) {
    std::cout << arg << ", ";
    print(args...);
}

int main() {
    print(1, 3.1, "abc");
    print(1, 2, 3, 4, 5, 6, 7);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1, 3.1, abc  
1, 2, 3, 4, 5, 6, 7
```

와 같이 잘 나옵니다. 그렇다면 위 코드가 어떻게 작동하는지 살펴보겠습니다.

```
template <typename T, typename... Types>
```

먼저 위와 같이 `typename` 뒤에 ... 으로 오는 것을 **템플릿 파라미터 팩(parameter pack)** 이라고 부릅니다. 템플릿 파라미터 팩의 경우 0 개 이상의 템플릿 인자들을 나타냅니다.

```
void print(T arg, Types... args) {
```

마찬가지로 함수에 인자로 ... 로 오는 것을 **함수 파라미터 팩** 이라고 부르며, 0 개 이상의 함수 인자를 나타냅니다. 템플릿 파라미터 팩과 함수 파라미터 팩의 차이점은 템플릿의 경우 타입 앞에 ... 이 오고, 함수의 경우 타입 뒤에 ... 가 온다는 점입니다.

파라미터 팩은 추론된 인자를 제외한 나머지 인자들을 나타내게 됩니다. 예를 들어서

```
print(1, 3.1, "abc");
```

위와 같은 `print` 함수 호출을 살펴보도록 합시다. C++ 컴파일러는 이 두 개의 `print` 함수 정의를 살펴보면서 어느 것을 택해야 할지 정해야 합니다. 첫 번째 `print` 의 경우 인자로 단 1 개만 받기 때문에 후보에서 제외되고 두 번째 `print` 가 택해집니다.

```
template <typename T, typename... Types>  
void print(T arg, Types... args) {  
    std::cout << arg << ", "  
    print(args...);  
}
```

`print` 의 첫 번째 인자는 1 이므로 `T` 는 `int` 로 추론되고, `arg` 에는 1 이 오게 됩니다. 그리고 `args` 에는 나머지 3.1 과 "abc" 가 오게 됩니다.

```
print(args...);
```

따라서 위 `args...` 에는 `print` 에 전달되었던 나머지 인자들이 쭈르륵 오게 되겠지요. 따라서 위 코드는 마치

```
void print(int arg, double arg2, const char* arg3) {
    std::cout << arg << ", ";
    print(arg2, arg3);
}
```

을 한 것과 마찬가지로 됩니다. 자 그럼 이제 재귀적으로 다시 인자 2 개를 받는 `print` 를 호출하겠습니다. 역시나 첫 번째 후보는 탈락하고, 두 번째 후보인 파라미터 팩을 받는 함수가 채택되어서 `T` 에는 `double` 이고 나머지 `Types...` 부분에는 `const char*` 이 들어가겠지요.

따라서 이를 통해 생성된 `print` 함수는

```
void print(double arg, const char* arg2) {
    std::cout << arg << ", ";
    print(arg2);
}
```

와 같이 생겼을 것입니다.

```
print(arg2);
```

자 그럼 이제 어떤 `print` 가 오버로드 될까요? 앞서 말했듯이 파라미터 팩은 0 개 이상의 인자들을 나타낸다고 하였습니다. 따라서

```
template <typename T, typename... Types>
void print(T arg, Types... args);
```

위 함수도 가능하고 (이 경우 `args...` 에 아무것도 전달되지 않습니다. 즉 `print()` 가 호출됩니다.)

```
template <typename T>
void print(T arg);
```

위도 가능합니다. 결론적으로 말하자면, 첫 번째 `print` 가 호출됩니다. 이는 C++ 규칙 상, 파라미터 팩이 없는 함수의 우선순위가 높기 때문입니다. 아무튼 덕분에 마지막에 `endl` 이 출력될 수 있었습니다.

순서를 바꾼다면?

한 가지 재밌는 점은 두 `print` 함수의 위치를 바꿔서 쓴다면 컴파일 오류가 발생한다는 점입니다.

```
#include <iostream>

template <typename T, typename... Types>
void print(T arg, Types... args) {
    std::cout << arg << ", ";
    print(args...);
}

template <typename T>
void print(T arg) {
    std::cout << arg << std::endl;
}

int main() {
    print(1, 3.1, "abc");
    print(1, 2, 3, 4, 5, 6, 7);
}
```

컴파일 하였다면

컴파일 오류

```
test3.cc: In instantiation of ‘void print(T, Types ...) [with T =
→ const char*; Types = {}]’:
test3.cc:7:8:   recursively required from ‘void print(T, Types
→ ... ) [with T = double; Types = {const char*}]’
test3.cc:7:8:   required from ‘void print(T, Types ...) [with T =
→ int; Types = {double, const char*}]’
test3.cc:16:22:   required from here
test3.cc:7:8: error: no matching function for call to ‘print()’
    print(args...);
    ~~~~~^~~~~~
test3.cc:5:6: note: candidate: template<class T, class ... Types>
→ void print(T, Types ...)
void print(T arg, Types... args) {
    ^~~~
test3.cc:5:6: note:   template argument deduction/substitution
→ failed:
test3.cc:7:8: note:   candidate expects at least 1 argument, 0
→ provided
    print(args...);
```

```
~~~~~^~~~~~
```

위와 같은 오류가 발생하게 됩니다. 그 이유는 C++ 컴파일러는 함수를 컴파일 시에, 자신의 앞에 정의되어 있는 함수들 밖에 보지 못하기 때문입니다. 따라서 `void print(T arg, Types... args)` 이 함수를 컴파일 할 때, `void print(T arg)` 이 함수가 존재함을 모르는 셈이지요.

그렇게 된다면, 마지막에 `print("abc")` 의 오버로딩을 찾을 때, 파라미터 팩이 있는 함수를 택하게 되는데, 그 경우 그 함수 안에서 `print()` 가 호출이 됩니다. 하지만 우리는 `print()` 를 정의하지 않았기에 컴파일러가 이 함수를 찾을 수 없다고 오류를 뿐만 하게 되는 것입니다.

따라서 항상 템플릿 함수를 작성할 때 그 순서에 유의해서 써야 합니다.

임의의 개수의 문자열을 합치는 함수

가변 길이 템플릿을 활용한 또 다른 예시로 임의의 길이의 문자열을 합쳐주는 함수를 들 수 있습니다. 예를 들어서 `std::string` 에서 문자열을 합치기 위해서는

```
concat = s1 + s2 + s3;
```

과 같이 해야 했는데, 잘 알다 시피 위는 사실

```
concat = s1.operator+(s2).operator+(s3);
```

와 같습니다. 문제는 `s2` 를 더할 때 메모리 할당이 발생하고, `s3` 을 더할 때 메모리 할당이 또 한번 발생할 수 있다는 뜻입니다. 합쳐진 문자열의 크기는 미리 알 수 있으니 차라리 한 번에 필요한 만큼 메모리를 할당해버리는 것이 훨씬 낫습니다.³⁾

```
std::string concat;
concat.reserve(s1.size() + s2.size() + s3.size()); // 여기서 할당 1 번 수행
concat.append(s1);
concat.append(s2);
concat.append(s3);
```

를 하게 된다면 깔끔하게 메모리 할당 1 번으로 끝낼 수 있습니다. 그렇다면 위와 같은 작업을 도와주는 함수를 만든다면 어떨까요? 아래처럼 말이지요.

```
std::string concat = StrCat(s1, "abc", s2, s3);
```

3) 메모리 할당/해제는 매우 느린 작업 중 하나입니다.

을 한다면 깔끔하게 concat에 `s1 + "abc" + s2 + s3` 한 문자열이 들어가게 됩니다. 물론 불필요한 메모리 할당이 없이 말이지요. 하지만 문제는 `StrCat` 함수가 임의의 개수의 인자를 받아야 된다는 것이지요. 여기서 바로 가볍길이 템플릿을 사용하면 됩니다.

첫 번째 시도

```
#include <iostream>
#include <string>

template <typename String>
std::string StrCat(const String& s) {
    return std::string(s);
}

template <typename String, typename... Strings>
std::string StrCat(const String& s, Strings... strs) {
    return std::string(s) + StrCat(strs...);
}

int main() {
    // std::string 과 const char* 을 혼합해서 사용 가능하다.
    std::cout << StrCat(std::string("this"), " ", "is", " ", std::string("a"),
                        " ", std::string("sentence"));
}
```

성공적으로 컴파일 하였다면

실행 결과

this is a sentence

와 같이 나옵니다. 위에서 파라미터 팩이 어떻게 작동하는지 이해하신 분들은 위 코드를 이해하기 쉬우실 것입니다. 우리의 `StrCat`은 재귀적으로 정의되어 있는데;

```
return std::string(s) + StrCat(strs...);
```

위에서 나머지 인자들을 합친 문자열과 현재 문자열(`s`)를 더해주게 됩니다. 그리고 당연히도 재귀 호출의 베이스 케이스인

```
template <typename String>
std::string StrCat(const String& s) {
    return std::string(s);
}
```

를 호출해야 하겠지요. `s` 는 `std::string` 으로 매번 감싸는 이유는 `s` 가 꼭 `std::string` 일 필요는 없기 때문이죠 (예컨대 `const char*` 일 수도 있음)

하지만 위 구현은 문제가 있습니다. 위에서도 이야기 했듯이 결과적으로 `std::string` 의 `operator+` 를 매번 호출하는 셈이기 때문이지요. 따라서 `StrCat` 에 전달된 인자가 5 개라면 메모리 할당이 최대 5 번씩이나 일어날 수 있게되는 셈입니다.

효율적으로 `StrCat` 을 구현하기 위해서는 합쳐진 문자열의 길이를 먼저 계산한 뒤에, 메모리를 할당하고, 그 다음에 문자열을 붙이는 것이 좋을 것입니다.

두 번째 시도

그렇다면 먼저 합쳐진 문자열의 길이를 먼저 구하는 함수를 만들어야 할 것입니다. 물론 이 역시 가변 길이 템플릿을 사용하면 매우 간단합니다.

```
size_t GetStringSize(const char* s) { return strlen(s); }

size_t GetStringSize(const std::string& s) { return s.size(); }

template <typename String, typename... Strings>
size_t GetStringSize(const String& s, Strings... strs) {
    return GetStringSize(s) + GetStringSize(strs...);
}
```

`GetStringSize` 함수는 그냥 임의의 개수의 문자열을 받아서 각각의 길이를 더한 것들을 리턴하게 됩니다. 참고로 `const char*` 과 `std::string` 모두 잘 작동하게 하기 위해서 인자 1 개만 받는 `GetStringSize` 의 오버로드를 각각의 경우에 대해 준비하였습니다.

그렇다면 수정된 `StrCat` 의 모습은 아래와 같을 것입니다.

```
template <typename String, typename... Strings>
std::string StrCat(const String& s, Strings... strs) {
    // 먼저 합쳐질 문자열의 총 길이를 구한다.
    size_t total_size = GetStringSize(s, strs...);

    // reserve 를 통해 미리 공간을 할당해 놓는다.
    std::string concat_str;
    concat_str.reserve(total_size);

    concat_str = s;

    // concat_str 에 문자열들을 붙인다.
    AppendToString(&concat_str, strs...);

    return concat_str;
}
```

먼저 `GetSize()` 를 통해서 합쳐진 문자열의 총 길이를 계산한 뒤에, 합쳐진 문자열을 보관할 `concat_str` 이라는 변수를 만들었습니다. 그리고 `reserve` 함수를 통해서 필요한 만큼 미리 공간을 할당해 놓죠.

그 다음에는 이제 `concat_str` 뒤에 나머지 문자열들을 가져다 붙여야 합니다. 이 과정을 수행하는 함수를 `AppendToString` 이라고 해봅시다. 그렇다면 `AppendToString` 은 아래와 같이 구성할 수 있을 것입니다.

```
void AppendToString(std::string* concat_str) { return; }

template <typename String, typename... Strings>
void AppendToString(std::string* concat_str, const String& s, Strings... strs) {
    concat_str->append(s);
    AppendToString(concat_str, strs...);
}
```

`AppendToString` 의 첫 번째 인자로는 합쳐진 문자열을 보관할 문자열을 계속 전달하고, 그 뒤로 합칠 문자열들을 인자로 전달하게 됩니다. 그리고 재귀 호출의 맨 마지막 단계로 `strs...` 가 아무 인자도 없을 때 까지 진행하므로, 재귀 호출을 끝내기 위해선 `AppendToString(std::string* concat_str)` 을 만들어줘야겠지요.

전체 완성된 코드를 보면 아래와 같습니다.

```
#include <cstring>
#include <iostream>
#include <string>

size_t GetSize(const char* s) { return strlen(s); }

size_t GetSize(const std::string& s) { return s.size(); }

template <typename String, typename... Strings>
size_t GetSize(const String& s, Strings... strs) {
    return GetSize(s) + GetSize(strs...);
}

void AppendToString(std::string* concat_str) { return; }

template <typename String, typename... Strings>
void AppendToString(std::string* concat_str, const String& s, Strings... strs) {
    concat_str->append(s);
    AppendToString(concat_str, strs...);
}

template <typename String, typename... Strings>
std::string StrCat(const String& s, Strings... strs) {
    // 먼저 합쳐질 문자열의 총 길이를 구한다.
    size_t total_size = GetSize(s, strs...);
```

```

// reserve 를 통해 미리 공간을 할당해 놓는다.
std::string concat_str;
concat_str.reserve(total_size);

concat_str = s;
AppendToString(&concat_str, strs...);

return concat_str;
}

int main() {
// std::string 과 const char* 을 혼합해서 사용 가능하다.
std::cout << StrCat(std::string("this"), " ", "is", " ", std::string("a"),
                     " ", std::string("sentence"));
}

```

성공적으로 컴파일 하였다면

실행 결과

this is a sentence

와 같이 잘 나옵니다.

sizeof...

`sizeof` 연산자는 인자의 크기를 리턴하지만 파라미터 팩에 `sizeof...` 을 사용할 경우 전체 인자의 개수를 리턴하게 됩니다. 예를 들어서 원소들의 평균을 구하는 함수를 생각해봅시다.

```

#include <iostream>

// 재귀 호출 종료를 위한 베이스 케이스
int sum_all() { return 0; }

template <typename... Ints>
int sum_all(int num, Ints... nums) {
    return num + sum_all(nums...);
}

template <typename... Ints>
double average(Ints... nums) {
    return static_cast<double>(sum_all(nums...)) / sizeof...(nums);
}

```

```
int main() {
    // (1 + 4 + 2 + 3 + 10) / 5
    std::cout << average(1, 4, 2, 3, 10) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

4

와 같이 잘 구합니다. 코드를 살펴보자면;

```
int sum_all() { return 0; }

template <typename... Ints>
int sum_all(int num, Ints... nums) {
    return num + sum_all(nums...);
}
```

sum_all 함수는 전달된 인자들의 합을 리턴하는 함수입니다. 파라미터 팩을 이해하셨더라면 위 코드를 이해하는데 큰 문제가 없을 것입니다.

```
template <typename... Ints>
double average(Ints... nums) {
    return static_cast<double>(sum_all(nums...)) / sizeof...(nums);
}
```

한편, average 함수의 경우 전달된 전체 인자 개수로 합을 나눠줘야만 합니다. 여기서 sizeof... 연산자가 활용됩니다. sizeof...에 파라미터 팩 (nums)를 전달하면 nums에 해당하는 실제 인자의 개수를 리턴해줍니다. 우리의 경우 인자를 5개 전달하였으므로 5가 되었겠지요.

Fold Expression

C++ 11에서 도입된 가변 길이 템플릿은 매우 편리하지만 한 가지 단점이 있어야 합니다. 재귀 함수 형태로 구성해야 하기 때문에, 반드시 재귀 호출 종료를 위한 함수를 따로 만들어야 한다는 것이지요.

예를 들어서 위에서 만들었던 sum_all 함수를 다시 살펴보자면;

```
// 재귀 호출 종료를 위한 베이스 케이스
int sum_all() { return 0; }
```

위와 같이 재귀 함수 호출을 종료하기 위해 베이스 케이스를 꼭 만들어줘야 한다는 점입니다. 이는 코드의 복잡도를 쓸데없이 늘리게 됩니다.

하지만 C++ 17 에 새로 도입된 Fold 형식을 사용한다면 이를 훨씬 간단하게 표현할 수 있습니다.

```
#include <iostream>

template <typename... Ints>
int sum_all(Ints... nums) {
    return (... + nums);
}

int main() {
    // 1 + 4 + 2 + 3 + 10
    std::cout << sum_all(1, 4, 2, 3, 10) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

20

과 같이 나옵니다.

```
return (... + nums);
```

위 문장이 바로 C++ 17 에 추가된 Fold 형식으로, 위는 아래와 같이 컴파일러에서 해석됩니다.

```
return (((1 + 4) + 2) + 3) + 10;
```

위와 같은 형태를 단항 좌측 Fold (Unary left fold)라고 부릅니다. C++ 17 에서 지원하는 Fold 방식의 종류로 아래 표와 같이 총 4 가지가 있습니다. 참고로 I 는 초기값을 의미하며 파라미터 팩이 아닙니다.

이름	Fold 방식	실제 전개 형태
(E op ...)	단항 우측 Fold	($E_1 \text{ op } (\dots \text{ op } (E_{N-1} \text{ op } E_N))$)
(... op E)	단항 좌측 Fold	(($E_1 \text{ op } E_2$) op ...) op E_N)
(E op ... op I)	이항 우측 Fold	($E_1 \text{ op } (\dots \text{ op } (E_{N-1} \text{ op } (E_N \text{ op } I)))$)
(I op ... op E)	이항 좌측 Fold	((($I \text{ op } E_1$) op E_2) op ...) op E_N)

여기서 `op` 자리에는 대부분의 이항 연산자들이 포함될 수 있습니다. 예를 들어서 `+`, `-`, `<`, `<<`, `->`, `,` 등등이 있습니다. 전체 목록은 [여기](#)를 참조하시면 됩니다.

한 가지 중요한 점은 Fold 식을 쓸 때 꼭 `()`로 감싸줘야 한다는 점입니다. 위 경우

```
return (... + nums);
```

대신에

```
return ... + nums;
```

로 컴파일 하게 된다면

컴파일 오류

```
test2.cc:6:10: error: expected primary-expression before '...'
    ↗ token
    return ... + nums;
          ^~~

test2.cc:6:10: error: expected ';' before '...' token
test2.cc:6:10: error: expected primary-expression before '...'
    ↗ token
```

위와 같은 오류가 발생하게 됩니다. (위 표에 `()` 가 Fold 식에 포함되어 있는 것입니다!)

이항 Fold 의 경우 아래와 같은 예시를 들 수 있습니다.

```
#include <iostream>

template <typename Int, typename... Ints>
Int diff_from(Int start, Ints... nums) {
    return (start - ... - nums);
}

int main() {
    // 100 - 1 - 4 - 2 - 3 - 10
    std::cout << diff_from(100, 1, 4, 2, 3, 10) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

80

와 같이 나옵니다.

```
return (start - ... - nums);
```

위 식은 위 표에 따르면 이항 좌측 Fold입니다. 왜냐하면 `start` 가 초기값이고 `nums` 가 파라미터 팩 부분이기 때문이지요. 따라서 위 식은 실제로는 아래와 같이 컴파일 됩니다.

```
return (((((100 - 1) - 4) - 2) - 3) - 10);
```

따라서 위처럼 80이라는 결과를 얻을 수 있겠지요.

한 가지 더 재미있는 점은, 연산자를 사용하면 각각의 인자들에 대해 원하는 식을 실행할 수 있습니다.

```
#include <iostream>

class A {
    public:
        void do_something(int x) const {
            std::cout << "Do something with " << x << std::endl;
        }
};

template <typename T, typename... Ints>
void do_many_things(const T& t, Ints... nums) {
    // 각각의 인자들에 대해 do_something 함수들을 호출한다.
    (t.do_something(nums), ...);
}

int main() {
    A a;
    do_many_things(a, 1, 3, 2, 4);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Do something with 1
Do something with 3
```

```
Do something with 2  
Do something with 4
```

와 같이 나옵니다.

```
(t.do_something(nums), ...);
```

위는 사실상 모든 인자들에 대해서 각각 `t.do_something(arg)` 를 실행한 것과 같습니다. 즉 실제 컴파일 되는 코드는

```
t.do_something(1);  
t.do_something(3);  
t.do_something(2);  
t.do_something(4);
```

가 되겠지요.

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. 가변 길이 템플릿을 잘 활용한다면 작성해야 하는 코드의 양을 줄일 수 있습니다.

다음 강좌에서는 템플릿 메타프로그래밍이라는, 템플릿을 통해 생성된 코드로 프로그래밍을 하는 새로운 패러다임에 대해서 다룰 것입니다.

뭘 배웠지?

- 파라미터 팩(...)을 사용해서 임의의 개수의 인자를 받는 템플릿을 작성할 수 있습니다.
- C++ 17 에 새로 추가된 Fold 형식에 대해 배웠습니다.

템플릿 메타프로그래밍 1부

안녕하세요 여러분! 지난 강좌들에서 다룬 템플릿을 통해서 프로그래밍이 좀 더 편해진 것 같나요? 이렇게 템플릿을 통해서 타입이 마치 인자 인것 처럼 사용하는 것을 바로 일반화 프로그래밍 (generic programming) 혹은 그냥 제너릭 프로그래밍이라고 부릅니다.

이전에 이야기 하였듯이 템플릿 인자로는 타입 뿐만이 아니라 특정한 조건을 만족하는 값들도 올 수 있습니다. 9 - 1 강의에서 보았던 `std::array` 가 어떻게 구현되었는지 만들어본다면 아래와 비슷할 것입니다.

```
/* 나만의 std::array 구현 */
#include <iostream>

template <typename T, unsigned int N>
class Array {
    T data[N];

public:
    // 배열을 받는 래퍼런스 arr
    Array(T (&arr)[N]) {
        for (int i = 0; i < N; i++) {
            data[i] = arr[i];
        }
    }

    T* get_array() { return data; }

    unsigned int size() { return N; }

    void print_all() {
        for (int i = 0; i < N; i++) {
            std::cout << data[i] << ", ";
        }
        std::cout << std::endl;
    };
};

int main() {
    int arr[3] = {1, 2, 3};

    // 배열 wrapper 클래스
    Array<int, 3> arr_w(arr);

    arr_w.print_all();
}
```

성공적으로 컴파일 하였다면

실행 결과

1, 2, 3,

와 같이 나옵니다.

```
// 배열 wrapper 클래스
Array<int, 3> arr_w(arr);
```

위와 같이 템플릿 인스턴스화를 하게 되면, 템플릿에 T 자리에는 int 가, N 자리에는 3 이 들어가겠지요. 그렇다면 컴파일러는

```
T data[N];
```

를

```
int data[3];
```

으로 대체해서 코드를 생성하게 되고, 마찬가지로

```
// 배열을 받는 레퍼런스 arr
Array(T (&arr)[N]) {
    for (int i = 0; i < N; i++) {
        data[i] = arr[i];
    }
}
```

생성자 역시

```
// 배열을 받는 레퍼런스 arr
Array(int (&arr)[3]) {
    for (int i = 0; i < 3; i++) {
        data[i] = arr[i];
    }
}
```

로 아예 코드가 생성되어 실행됩니다. 참고로 이처럼 배열을 감싸는 wrapper 클래스를 만들어서 마치 배열처럼 사용한다면 (물론 그러기 위해서는 [] 연산자도 오버로드 해야겠죠?) 배열을 사용함으로써 발생하는 문제들을 많이 해결할 수 있게 됩니다.

예를 들어서, 일반 배열은 배열 범위가 넘어가도 알 수 없지만, 위 Array 클래스는 index 범위가 넘어가는 곳을 가리키면 뭘가 메세지를 띄우든 오류를 발생 시키든 해서 사용자에게 알려 줄 수 있습니다.

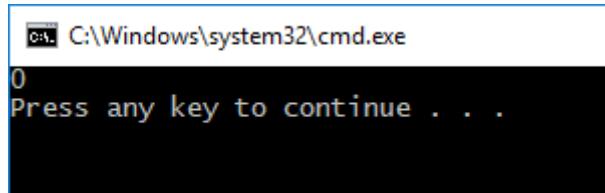
그런데, 과연 아래 두 개 클래스는 같은 클래스 일까요? 다른 클래스 일까요?

```
Array<int, 5> Array<int, 3>
```

간단히 아래 코드로 확인해 볼 수 있습니다.

```
std::cout << (typeid(Array<int, 3>) == typeid(Array<int, 5>)) << std::endl;
```

참고로 typeid를 사용하려면 <typeinfo> 헤더파일을 추가해주시면 됩니다. 그 결과는 당연하게도



와 같이 다르다고 나옵니다. 왜 다르냐면 당연히, 다른 템플릿 인자로 인스턴스화 되었기 때문이지요. 컴파일러는 Array<int, 5> 와 Array<int, 3> 를 위해 각기 다른 코드를 생성하며 다른 클래스의 객체들을 만들게 됩니다.

그렇다면 아래와 같이 정의된 Int 클래스를 생각해봅시다.

```
template <int N>
struct Int {
    static const int num = N;
};
```

이 클래스는 템플릿 인자로 int 값을 받습니다. 참고로, 왜 static const에 값을 저장하냐면, 첫 번째로 C++ 클래스 멤버 중에서 클래스 자체에서 저런 식으로 초기화를 할 수 있는 멤버의 타입은 static const 밖에 없고, 두 번째로 static const 약 말로 이 클래스는 이 것이다라는 의미를 가장 잘 나타내기 때문입니다.

왜냐하면 static 타입 멤버의 특성 상, 이 클래스가 생성한 객체들 사이에서 공유되는 값이기 때문에 '이 타입이면 이 값을 나타낸다'라고 볼 수 있습니다. 또한 const 이므로, 그 나타내는 값이 변하지 않게 됩니다.

따라서 아래처럼 마치 객체를 생성하듯 타입들을 생성할 수 있습니다.

```
typedef Int<1> one;
typedef Int<2> two;
```

그렇다면 저 one 타입과 two 타입은 1 과 2 의 값을 나타내는 타입이 됩니다. (one 과 two 는 객체가 아닙니다!)

그럼 이제 one 과 two 를 가지고 무엇을 할 수 있을까요? 재미있게도 마치 int 변수를 다루는 것처럼 연산자를 만들 수 있습니다. 아래 예제를 살펴볼까요.

```
#include <iostream>
#include <typeinfo>

template <int N>
struct Int {
    static const int num = N;
};

template <typename T, typename U>
struct add {
    typedef Int<T::num + U::num> result;
};

int main() {
    typedef Int<1> one;
    typedef Int<2> two;

    typedef add<one, two>::result three;

    std::cout << "Addtion result : " << three::num << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

Addtion result : 3

와 같이 실제 계산 결과가 잘 나오게 됩니다.

덧셈을 수행하는 템플릿 클래스를 살펴봅시다.

```
template <typename T, typename U>
struct add {
    typedef Int<T::num + U::num> result;
};
```

위 `add` 클래스의 템플릿은 인자로 두 개의 타입을 받아서 그 타입의 `num` 멤버를 더해서 새로운 타입인 `result`를 만들어 내게 됩니다.

```
typedef add<one, two>::result three;
```

위 부분은 실제 덧셈을 수행하는 부분입니다. `add` 클래스를 함수라고 생각한다면 그 계산 결과를 내부 `result` 타입으로 반환한다고 보면 됩니다. 아무튼 `one`과 `two`를 더한 것을 나타내는 타입이 `result`로 정의되고, 이를 `three`라고 부르겠습니다.

실제로, 그 결과를 보면

```
std::cout << "Addtion result : " << three::num << std::endl;
```

를 통해서 3 이 잘 출력됨을 알 수 있습니다.

한 가지 흥미로운 점은 저 3이라는 값이 프로그램이 실행되면서 계산되는 것이 아니라는 점입니다. 컴파일 시에, 컴파일러가 `three::num` 을 3 으로 치환 해버립니다.

```

typedef Int<1> one;
typedef Int<2> two;

typedef add<one, two>::result three;

cout << "Addtion result : " << three::num << endl;
}                                     static const int Int<3>::num = 3

```

실제로 마우스를 올려보면 저 값이 3 이란 사실을 알 수 있습니다.

다시 말해, 저 덧셈이 수행 되는 시기는 컴파일 타임이고, 런타임 시에는 단순히 그 결과를 보여주게 되는 것입니다.

템플릿 메타 프로그래밍 (Template Meta Programming - TMP)

여태까지 타입은 어떠한 객체에 무엇을 저장하느냐를 지정하는데 사용해 왔지, 타입 자체가 어떠한 값을 가지지는 않았습니다. 하지만, 바로 위 예제를 통해서 알 수 있듯이, 템플릿을 사용하면 객체를 생성하지 않더라도, 타입에 어떠한 값을 부여할 수 있고, 또 그 타입들을 가지고 연산을 할 수 있다는 점입니다.

또한 타입은 반드시 컴파일 타임에 확정되어야 하므로, 컴파일 타임에 모든 연산이 끝나게 됩니다. 이렇게 타입을 가지고 컴파일 타임에 생성되는 코드로 프로그래밍을 하는 것을 **메타 프로그래밍**(meta

programming) 이라고 합니다. C++ 의 경우 템플릿을 가지고 이러한 작업을 하기 때문에 템플릿 메타 프로그래밍, 줄여서 TMP 라고 부릅니다.

```
/* 컴파일 타임 팩토리얼 계산 */
#include <iostream>
template <int N>
struct Factorial {
    static const int result = N * Factorial<N - 1>::result;
};

template <>
struct Factorial<1> {
    static const int result = 1;
};

int main() {
    std::cout << "6! = 1*2*3*4*5*6 = " << Factorial<6>::result << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
6! = 1*2*3*4*5*6 = 720
```

와 같이 잘 나옵니다.

팩토리얼(factorial) 은 단순히 1 부터 n 까지 곱한 것이라 생각하면 됩니다. 예를 들어 3 팩토리얼 ($3!$ 이라 씁니다) 은 $1 * 2 * 3$ 이라 생각하면 됩니다. 이 팩토리얼을 어떻게 하면 이전 예제와 같은 템플릿을 사용한 구조로 나타낼 수 있을까요? 사실 아래와 같이 매우 단순합니다.

```
template <int N>
struct Factorial {
    static const int result = N * Factorial<N - 1>::result;
};
```

만약에 저 Factorial 을 일반적인 함수로 구성하려고 했다면 아마 아래와 같은 재귀 함수 형태를 사용했겠지요.

```
int factorial(int n) {
    if (n == 1) return 1;

    return n * factorial(n - 1);
}
```

따라서 우리는 위처럼 재귀 함수 호출이 끝나게 하기 위해선, n 이 1 일 때 따로 처리를 해주어야 합니다. 템플릿 역시 마찬가지로 $n = 1$ 일 때 따로 처리할 수 있는데 바로 아래처럼 템플릿 특수화를 이용해주면 됩니다.

```
template <>
struct Factorial<1> {
    static const int result = 1;
};
```

컴파일러는 `Factorial<1>` 타입의 경우만 따로 `result = 1`로 만들어주게 되어서 재귀적 구조가 끝날 수 있게 해줍니다.

위 예제에서 볼 수 있듯이, 저기서 실질적으로 값을 가지는 객체는 아무 것도 없습니다. 즉, '720'이라는 값을 가지고 있는 변수는 메모리 상에서 없다는 뜻입니다 (물론 `std::cout`에서 출력 할 때 빼고). 저 화면에 나타나는 720이라는 값은, 단순히 컴파일러가 만들어낸 `Factorial<6>`이라는 타입을 나타내고 있을 뿐입니다.

사실 여러분 한테 `factorial` 을 계산하라는 함수를 만들라고 이야기 했다면 십중팔구 그냥 단순히 `for` 문으로 구현을 하였을 것입니다. 하지만 안타깝게도 템플릿으로는 `for` 문을 쓸 수 없기 때문에 위와 같은 재귀적 구조를 사용하였습니다. 한 가지 다행인 소식은 `for` 문으로 구현할 수 있는 모든 코드는 똑같이 템플릿 메타 프로그래밍을 이용해서 구현할 수 있습니다.

더군다나 위에서 보셨듯이 `if` 문 역시 템플릿 특수화를 통해 TMP 로 구현할 수 있습니다.

TMP 를 왜 쓰는가?

한 가지 재미있는 사실은 어떠한 C++ 코드도 템플릿 메타 프로그래밍 코드로 변환할 수 있다는 점입니다(물론 엄청나게 코드가 길어지겠지만요). 게다가 템플릿 메타 프로그래밍으로 작성된 코드는 모두 컴파일 타임에 모든 연산이 끝나기 때문에 프로그램 실행 속도를 향상 시킬 수 있다는 장점이 있습니다 (당연히도 컴파일 시간은 엄청 늘어나게 됩니다).

하지만 그렇다고 해서 템플릿 메타 프로그래밍으로 프로그램 전체를 구현하는 일은 없습니다. 일단 템플릿 메타 프로그래밍은 매우 복잡합니다. 물론 위 `Factorial` 예제는 꽤 간단하였지만 아래 좀 더 복잡한 예제를 다루면서 왜 템플릿 메타 프로그래밍이 힘든 것인지 이야기 하겠습니다.

그 뿐만이 아니라, 템플릿 메타 프로그래밍으로 작성된 코드는 버그를 찾는 것이 매우 힘듭니다. 일단 기본적으로 '컴파일' 타임에 연산하는 것이기 때문에 디버깅이 불가능 하고, C++ 컴파일러에 특성 상 템플릿 오류 시에 엄청난 길이의 오류를 내뿜게 됩니다.

따라서 TMP 를 이용하는 경우는 꽤나 제한적이지만, 많은 C++ 라이브러리들이 TMP 를 이용해서 구현되었고 (Boost 라이브러리), TMP 를 통해서 컴파일 타임에 여러 오류들을 잡아낼 수도 있고 (Ex. 단위나 통화 일치 여부등등) 속도가 매우 중요한 프로그램의 경우 TMP 를 통해서 런타임 속도도 향상 시킬 수 있습니다.

아래에서 좀 더 복잡한 예제를 가지고 그렇다면 TMP 를 어떻게 사용할 지에 대해서 자세히 알아보도록 하겠습니다.

컴퓨터 상에서 두 수의 최대공약수를 구하기 위해선 보통 유클리드 호제법을 이용합니다. 이는 매우 간단한데, 이 알고리즘을 일반적인 함수로 나타내자면 아래와 같습니다.

```
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }

    return gcd(b, a % b);
}
```

따라서 이를 그대로 TMP 로 바꿔보면 아래와 같습니다. (여러분도 직접 해보세요!)

```
#include <iostream>

template <int X, int Y>
struct GCD {
    static const int value = GCD<Y, X % Y>::value;
};

template <int X>
struct GCD<X, 0> {
    static const int value = X;
};

int main() {
    std::cout << "gcd (36, 24) :: " << GCD<36, 24>::value << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
gcd (36, 24) :: 12
```

와 같이 잘 계산됩니다.

이 최대 공약수 계산 클래스를 만든 이유는, 바로 `Ratio` 클래스를 만들기 위함입니다. `Ratio` 클래스는 유리수($\frac{p}{q}$ 꼴로 쓸 수 있는 수)를 오차 없이 표현해 주는 클래스입니다.

물론 TMP 를 사용하지 않고 간단하게 클래스를 사용해서도 만들 수 있습니다. 하지만 일단 연습삼아서 한 번 TMP 를 사용해서 만들어 보겠습니다.

```
template <int N, int D = 1>
struct Ratio {
    typedef Ratio<N, D> type;
    static const int num = N;
    static const int den = D;
};
```

먼저 `Ratio` 클래스는 위 처럼 정의할 수 있겠습니다. 위 처럼 분자와 분모를 템플릿 인자로 받고, 타입을 나타내게 됩니다. 참고로 편의상

```
typedef Ratio<N, D> type;
```

`typedef`로 '자기 자신을 가리키는 타입'을 넣어 주었습니다. 이는 마치 클래스에서의 `this`와 비슷한 역할입니다.

그렇다면 이 `Ratio`로 덧셈을 수행하는 템플릿을 만들어보겠습니다. 상당히 직관적입니다.

```
template <class R1, class R2>
struct _Ratio_add {
    typedef Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den> type;
};
```

두 분수의 더한 결과를 `Ratio`에 분자 분모로 전달하면 알아서 기약분수로 만들어줍니다.

```
typedef Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den> type;
```

그 후에, 그 덧셈 결과를 `type`로 나타내게 됩니다. 따라서 덧셈을 수행하기 위해서는

```
typedef _Ratio_add<rat, rat2>::type result;
```

이런 식으로 사용하면 되겠지요. 하지만 한 발 더 나아가서, 귀찮게 `::type`를 치고 싶지 않다고 해봅시다. 다시 말해 `Ratio_add`를 하면 그 자체로 두 `Ratio`가 더해진 타입이 되는 것이지요. 이는 아래와 같이 구현할 수 있습니다.

```
template <class R1, class R2>
struct Ratio_add : _Ratio_add<R1, R2>::type {};
```

바로 `_Ratio_add<R1, R2>::type`를 상속 받는 `Ratio_add` 클래스를 만들어 버리는 것입니다! 상당히 재미있는 아이디어입니다. 따라서 `Ratio_add`는 마치 `Ratio` 타입 처럼 사용할 수 있게 됩니다. 전체 코드를 살펴 보자면 아래와 같습니다.

```
#include <iostream>
#include <typeinfo>

template <int X, int Y>
struct GCD {
    static const int value = GCD<Y, X % Y>::value;
};

template <int X>
struct GCD<X, 0> {
    static const int value = X;
};

template <int N, int D = 1>
struct Ratio {
    typedef Ratio<N, D> type;
    static const int num = N; // 분자
    static const int den = D; // 분모
};

template <class R1, class R2>
struct _Ratio_add {
    typedef Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den> type;
};

template <class R1, class R2>
struct Ratio_add : _Ratio_add<R1, R2>::type {};

int main() {
    typedef Ratio<2, 3> rat;
    typedef Ratio<3, 2> rat2;
    typedef Ratio_add<rat, rat2> rat3;

    std::cout << rat3::num << " / " << rat3::den << std::endl;

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

13 / 6

와 같이 잘 계산되어서 나옵니다.

참고로 C++11 부터 `typedef` 대신에 좀 더 직관적인 `using`이라는 키워드를 사용할 수 있습니다.

```
typedef Ratio_add<rat, rat2> rat3;
using rat3 = Ratio_add<rat, rat2>;
```

위 두 문장 모두 동일한 의미를 가집니다. 다만 **using** 을 사용하였을 경우 **typedef** 보다 좀 더 이해하기가 쉽습니다. 특히, 함수 포인터의 경우 만일 **void** 를 리턴하고 **int**, **int** 를 인자로 받는 함수의 포인터의 타입을 **func** 라고 정의하기 위해서는 **typedef** 로

```
typedef void (*func)(int, int);
```

위와 같이 사용해야 했지만 (놀랍게도 **func** 이 새로 정의된 타입 이름이 됩니다) **using** 키워드를 사용하면

```
using func = void (*)(int, int);
```

아래와 같이 매우 직관적으로 나타낼 수 있습니다. 따라서 위의 코드를 수정하자면;

```
int main() {
    using rat = Ratio<2, 3>;
    using rat2 = Ratio<3, 2>;

    using rat3 = Ratio_add<rat, rat2>;
    std::cout << rat3::num << " / " << rat3::den << std::endl;

    return 0;
}
```

로 간단하고 좀 더 직관적으로 나타낼 수 있습니다. 다시 한 번 말하지만, 마치 **Ratio** 클래스의 객체를 생성한 것 같지만, 실제로 생성된 객체는 한 개도 없고, 단순히 타입들을 컴파일러가 만들어낸 것 뿐입니다.

마찬가지 방법으로 모든 사칙연산들을 구현하자면 아래와 같습니다.

```
#include <iostream>

template <int X, int Y>
struct GCD {
    static const int value = GCD<Y, X % Y>::value;
};

template <int X>
struct GCD<X, 0> {
    static const int value = X;
```

```
};

template <int N, int D = 1>
struct Ratio {
    private:
        const static int _gcd = GCD<N, D>::value;

    public:
        typedef Ratio<N / _gcd, D / _gcd> type;
        static const int num = N / _gcd;
        static const int den = D / _gcd;
};

template <class R1, class R2>
struct _Ratio_add {
    using type = Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_add : _Ratio_add<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_subtract {
    using type = Ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_subtract : _Ratio_subtract<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_multiply {
    using type = Ratio<R1::num * R2::num, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_multiply : _Ratio_multiply<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_divide {
    using type = Ratio<R1::num * R2::den, R1::den * R2::num>;
};

template <class R1, class R2>
struct Ratio_divide : _Ratio_divide<R1, R2>::type {};

int main() {
    using r1 = Ratio<2, 3>;
    using r2 = Ratio<3, 2>;

    using r3 = Ratio_add<r1, r2>;
    std::cout << "2/3 + 3/2 = " << r3::num << " / " << r3::den << std::endl;
```

```
using r4 = Ratio_multiply<r1, r3>;
std::cout << "13 / 6 * 2 /3 = " << r4::num << " / " << r4::den << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
2/3 + 3/2 = 13 / 6
13 / 6 * 2 /3 = 13 / 9
```

와 같이 나옵니다.

자 여기까지 따라 오셨다면 한 가지 궁금증이 들 것입니다.

"음 그래. `Ratio` 로 이용해서 재미있는 것을 할 수 있는거 같애. 컴파일 타임에 유리수 사칙 연산을 계산할 수 있다는 것도 좋아. 근데 도대체 저게 왜 필요하지?"

바로 다음 강좌에서 알아 보도록 하겠습니다!

생각 해보기

문제 1

N 번째 피보나치 수를 나타내는 TMP 를 만들어보세요. 참고로 피보나치 수는, N 번째 항이 $N - 1$ 번째 항과 $N - 2$ 번째 항의 합으로 정의되는 수입니다. 참고로 1, 1, 2, 3, 5, ... 로 진행됩니다.(난이도 : 하)

```
int main() {
    std::cout << "5 번째 피보나치 수 :: " << fib<5>::result << std::endl; // 5
}
```

문제 2

TMP 를 사용해서 어떤 수가 소수인지 아닌지를 판별하는 프로그램을 만들어보세요. (난이도 : 상)참고로 이 문제는 다음 강좌에서 다룰 예정입니다!

```
int main() {
    std::cout << boolalpha;
    std::cout << "Is prime ? :: " << is_prime<2>::result << std::endl; // true
    std::cout << "Is prime ? :: " << is_prime<10>::result << std::endl; // false
}
```

```
    std::cout << "Is prime ? :: " << is_prime<11>::result << std::endl; // true  
    std::cout << "Is prime ? :: " << is_prime<61>::result << std::endl; // true  
}
```

템플릿 메타프로그래밍 2부

안녕하세요 여러분! 지난 강좌에서 왜 TMP를 활용하여 힘들게 힘들게 Ratio 클래스를 만들었는데, 아직 아마 이걸 왜 굳이 TMP로 만들었는지는 설명하지 않았었습니다.

그에 앞서, 이번 강좌에서 왜 Ratio 클래스를 만들었는지 설명하기 전에 지난 강좌의 생각해보기 문제에 대해서 짚고 넘어가보자 합니다.

지난 강의 생각해보기 문제

지난번 생각해보기 문제는 아래와 같습니다.

TMP를 사용해서 어떤 수가 소수인지 아닌지를 판별하는 프로그램을 만들어보세요. (난이도 : 상)

```
int main()
{
    std::cout << std::boolalpha;
    std::cout << "Is prime ? :: " << is_prime<2>::result << std::endl; // true
    std::cout << "Is prime ? :: " << is_prime<10>::result << std::endl; // false
    std::cout << "Is prime ? :: " << is_prime<11>::result << std::endl; // true
    std::cout << "Is prime ? :: " << is_prime<61>::result << std::endl; // true
}
```

사실 처음에 딱 보았을 때 도대체 어떻게 TMP로 구현할 것인지 감이 안잡혔을 것입니다. 하지만 만약에 소수인지 아닌지 판별하라는 '함수'를 작성하게 하였다면 잘 작성하였겠지요. 아마 여러분은 아래와 같은 코드를 쓰셨을 것입니다.

```
bool is_prime(int N) {
    if (N == 2) return true;
    if (N == 3) return true;

    for (int i = 2; i <= N / 2; i++) {
        if (N % i == 0) return false;
    }

    return true;
}
```

왜 2와 3 일 때 따로 처리하냐면 $N / 2$ 까지 나누는 걸로 비교할 때 2, 3 일 경우 제대로 처리가 안되기 때문입니다. 이제 여러분이 해야 할 일은 간단히 저 코드를 TMP 형식으로 옮기는 것입니다.

```
template <>
struct is_prime<2> {
```

```

    static const bool result = true;
};

template <>
struct is_prime<3> {
    static const bool result = true;
};

template <int N>
struct is_prime {
    static const bool result = !check_div<N, 2>::result;
};

template <int N, int d>
struct check_div {
    static const bool result = (N % d == 0) || check_div<N, d + 1>::result;
};

template <int N>
struct check_div<N, N / 2> {
    static const bool result = (N % (N / 2) == 0);
};

```

무언가 잘 짜여진 코드 같습니다. 하지만 실제로 컴파일 해보면 다음과 같은 오류가 발생합니다.

컴파일 오류

```

check_div<N,N/>: non-type parameter of a partial specialization
→ must be a simple identifier

```

바로

```

template <int N>
struct check_div<N, N / 2> {
    static const bool result = (N % (N / 2) == 0);
};

```

이 부분에서 발생하는 문제 이지요. 위 오류가 발생한 문제는 템플릿 부분 특수화 시에 반드시 다른 연산자가 붙지 않고 단순한 식별자만 입력해주어야만 합니다. 따라서 C++ 컴파일러에 한계 상

컴파일 오류

```

struct check_div<N, N / 2>

```

와 같은 문법은 불가능 합니다. 그렇다면 이를 어떻게 해결할 수 있을까요? 생각을 잘 해보면, N 을 int 인자로 나타내는 대신에, 아예 N 을 나타내는 '타입' 으로 구현하면 어떨까요? 그렇다면 N / 2 역시, 직접 계산하는것이 아니라 N / 2 를 나타내는 타입으로 대체할 수 있고 따라서 템플릿 부분 특수화 문제를 해결할 수 있습니다.

따라서 아래와 같이 int 값을 표현하는 타입을 만들 수 있습니다.

```
template <int N>
struct INT {
    static const int num = N;
};

template <typename a, typename b>
struct add {
    typedef INT<a::num + b::num> result;
};

template <typename a, typename b>
struct divide {
    typedef INT<a::num / b::num> result;
};

using one = INT<1>;
using two = INT<2>;
using three = INT<3>;
```

예를 들어 one 타입은 1을, two 타입은 2 를 나타내게 됩니다. 그렇다면 이를 바탕으로 TMP 코드를 수정해보도록 하겠습니다.

```
using one = INT<1>;
using two = INT<2>;
using three = INT<3>;

template <typename N, typename d>
struct check_div {
    // result 중에서 한 개라도 true 면 전체가 true
    static const bool result =
        (N::num % d::num == 0) || check_div<N, add<d, one>::result>::result;
};

template <typename N>
struct is_prime {
    static const bool result = !check_div<N, two>::result;
};

template <>
struct is_prime<two> {
    static const bool result = true;
```

```

};

template <>
struct is_prime<three> {
    static const bool result = true;
};

template <typename N>
struct check_div<N, divide<N, two>::result> {
    static const bool result = (N::num % (N::num / 2) == 0);
};

```

그런데 컴파일 한다면 다음과 같은 오류를 보게 됩니다.

컴파일 오류

```
'check_div': 'divide<N,two>::result' is not a valid template type
→ argument for parameter 'd'
```

왜 저런 오류가 발생하였을까요? 일단 오류가 발생하는 다음 두 부분의 코드를 살펴보겠습니다.

```
(N::num % d::num == 0) || check_div<N, add<d, one>::result>::result;
```

와

```
struct check_div<N, divide<N, two>::result> {
```

입니다. 먼저 컴파일러 입장에서 저 `::result` 를 어떻게 해석할지에 대해 생각해봅시다. 물론 우리는 `add<d, one>::result` 가 언제나 INT<> 타입이라는 사실을 알고 있습니다. 왜냐하면 `typename` 인자로 들어오는 `N` 과 `d` 가 항상 INT 타입이기 때문에 저 `result` 를 항상 '타입'이네라고 생각할 것입니다.

그런데, 컴파일러에 구조상 어떠한 식별자(변수 이름이든 함수 이름이든 코드 상의 이름들 - 위 코드의 경우 `add`, `check_div`, `result`, `one` 등등 ...) 를 보았을 때 이 식별자가 '값'인지 '타입'인지 결정을 해야 합니다. 왜냐하면 예를 들어서

```

template <typename T>
int func() {
    T::t* p;
}

class A {
    const static int t;

```

```
};

class B {
    using t = int;
};
```

위와 같은 템플릿 함수에서 저 문장을 해석할 때 만약에 클래스 A에 대해서, func 함수를 특수화 한다면, t가 어떠한 int 값이 되어서

```
T::t* p;
```

위 문장은 단순히 클래스 A의 t와 p를 곱하는 식으로 해석이 됩니다.

반면에 func 함수가 클래스 B에 대해서 특수화 된다면,

```
T::t* p;
```

이 문장은 int 형 포인터 p를 선언하는 꼴이 되겠지요. 따라서 컴파일러가 이 두 상황을 명확히 구분하기 위해 저 T::t가 타입인지 아니면 값인지 명확하게 알려줘야만 합니다.

우리가 쓴 코드도 마찬가지로 컴파일러가 result가 항상 '타입'인지 아니면 '값'인지 알 수 없습니다. 예컨대 만약에

```
template <>
struct divide<int a, int b> {
    const static int result = a + b;
};
```

이런 템플릿이 정의가 되어있다면, 만약에 N과 two가 그냥 int 값이였다면 저 result는 static const int 타입의 '값'이 됩니다. 이렇게 템플릿 인자에 따라서 어떠한 타입이 달라질 수 있는 것을 의존 타입(dependent type)이라고 부릅니다. 위 경우 저 result는 N에 의존하기 때문에 의존 타입이 되겠지요.

따라서 컴파일러가 저 문장을 성공적으로 해석하기 위해서는 우리가 반드시 "야 저 result는 무조건 타입이야"라고 알려주어야만 합니다. 이를 위해서는 간단히 아래 코드처럼

```
struct check_div<N, typename divide<N, two>::result> {
```

typename 키워드를 붙여주면 됩니다. 마찬가지로

```
(N::num % d::num == 0) || check_div<N, add<d, one>::result>::result;
```

에서 `typename` 키워드를 붙인다면

```
(N::num % d::num == 0) || check_div<N, typename add<d, one>::result>::result;
```

이 되겠지요. 참고로 의존 '값'의 경우 `typename` 을 안 붙여줘도 됩니다. 컴파일러는 어떤 식별자를 보았을 때 기본으로 '값'이라고 생각합니다. 따라서 `check_div` 앞에 아무것도 안올 수 있는 것입니다 (`check_div`의 `result`는 `static const bool` 이기 때문에!)

따라서 이를 고치면 다음과 같습니다.

```
template <typename N, typename d>
struct check_div {
    // result 중에서 한 개라도 true 면 전체가 true
    static const bool result = (N::num % d::num == 0) ||
                                check_div<N, typename add<d, one>::result>::result;
};

// 생략

template <typename N>
struct check_div<N, typename divide<N, two>::result> {
    static const bool result = (N::num % (N::num / 2) == 0);
};
```

마지막으로, 위 `is_prime` 을 사용하기 위해서는

```
is_prime<INT<11>>::result
```

이런 식으로 사용해야 합니다. 하지만 생각해보기에서 요구한 것은 `is_prime<11>::result` 를 사용하는 것이기 때문에 이를 위해서 `is_prime` 을 다음과 같이 정의하고, 기존의 `is_prime` 을 `_is_prime` 으로 바꾸도록 하겠습니다.

```
template <int N>
struct is_prime {
    static const bool result = _is_prime<INT<N>>::result;
};
```

그렇다면 전체 코드를 살펴보겠습니다.

```
#include <iostream>

template <int N>
struct INT {
```

```
    static const int num = N;
};

template <typename a, typename b>
struct add {
    typedef INT<a::num + b::num> result;
};

template <typename a, typename b>
struct divide {
    typedef INT<a::num / b::num> result;
};

using one = INT<1>;
using two = INT<2>;
using three = INT<3>;

template <typename N, typename d>
struct check_div {
    // result 중에서 한 개라도 true 면 전체가 true
    static const bool result = (N::num % d::num == 0) ||
                               check_div<N, typename add<d, one>::result>::result;
};

template <typename N>
struct _is_prime {
    static const bool result = !check_div<N, two>::result;
};

template <>
struct _is_prime<two> {
    static const bool result = true;
};

template <>
struct _is_prime<three> {
    static const bool result = true;
};

template <typename N>
struct check_div<N, typename divide<N, two>::result> {
    static const bool result = (N::num % (N::num / 2) == 0);
};

template <int N>
struct is_prime {
    static const bool result = _is_prime<INT<N>>::result;
};

int main() {
    std::cout << std::boolalpha;
```

```
std::cout << "Is 2 prime ? :: " << is_prime<2>::result << std::endl;
std::cout << "Is 10 prime ? :: " << is_prime<10>::result << std::endl;
std::cout << "Is 11 prime ? :: " << is_prime<11>::result << std::endl;
std::cout << "Is 61 prime ? :: " << is_prime<61>::result << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Is 2 prime ? :: true
Is 10 prime ? :: false
Is 11 prime ? :: true
Is 61 prime ? :: true
```

와 같이 제대로 판별함을 알 수 있습니다.

참고로 컴파일러에 따라서 재귀적으로 몇 번까지 사용 가능한지 깊이가 정해져있기 때문에, 꽤 큰 수를 넣는다면 컴파일 오류가 발생할 수도 있습니다.

자, 그렇다면 본격적으로 이번 강의를 시작해보도록 하겠습니다. 지난 강좌에서 `Ratio` 클래스를 만들면서 왜 굳이 이것을 TMP로 만들었을까, 그냥 일반적인 클래스로 만들었다면 훨씬 직관적이고 편하지 않나라고 많이 생각하셨을 것입니다. 바로 지금부터 TMP의 진짜 파워에 대해서 알아보도록 하겠습니다.

단위(Unit) 라이브러리

C++ 코드를 작성하는 이유는 여러가지가 있겠지만, 그 중 하나로 바로 여러 수치 계산을 사용하는데에도 많이 사용합니다. 예를 들어서 인공위성의 궤도를 계산한다던지, 입자의 운동을 계산한다던지 말이지요. 이러한 물리적 수치 계산 시에 꼭 필요한 것이 바로 '단위'입니다.

단위라 하면 쉽게 말해 킬로그램 (kg), 미터 (m), 초 (s) 등을 생각하시면 됩니다. 이러한 것들을 계산하는 프로그램들의 경우, `double`이나 `float` 변수에 들어가는 값에는 '단위'가 붙어서 들어가겠지요.

예를 들어서 핸드폰의 가속도 센서에서 부터 데이터를 받는 프로그램은 아마도 m/s^2 단위로 데이터를 받겠지요. 혹은 시계로 부터 데이터를 받는 프로그램은 s 단위로 데이터를 받을 것입니다.

아무튼 이렇게 단위가 붙은 데이터를 처리할 때 중요한 점은 바로 데이터를 연산할 때 항상 단위를 확인해야 된다는 점입니다. 예를 들어서, 다음과 같은 코드가 있다고 생각해봅시다.

```
float v1, v2; // v1, v2 는 속도
std::cout << v1 + v2;
```

당연히 v1 과 v2 는 속도 값을 나타내므로 같은 단위이기 때문에 더할 수 있습니다. (여기서 더할 수 있다는 말은 물리적으로 더한 값이 말이 된다는 의미입니다). 반면에;

```
float v; // 속도; m/s
float a; // 가속도; m/s^2
std::cout << v + a; // ???
```

만약에 v 가 속도를 나타내는 값이고, a 가 가속도를 나타내는 값이라면, v + a 는 불가능한 연산입니다. 만약에 프로그래머가 저러한 코드를 썼다면 분명히 실수일 것입니다. 물론 C++ 컴파일러 입장에서는 그냥 두 개의 float 변수를 더한 것이기 때문에 문제 없이 컴파일 됩니다. 하지만 프로그램을 돌리게 된다면 골치아픈 문제가 발생하겠지요.

실제로, NASA 의 경우 단위를 잘못 처리해서 1조원 짜리 화성 탐사선을 날려먹은 경우가 있습니다. 이 경우 1조원 자리 버그 이겠네요.

여러분 이라면 이러한 실수를 어떻게 막을 것인가요? 일단 가장 먼저 드는 생각으로 단위 데이터를 일반적인 변수에 보관하지 말고 클래스를 만들어서 클래스 객체에서 보관하는 것입니다. 그리고 operator+ 등으로 연산자들을 오버로딩 한 뒤에, 연산 시에 객체 끼리 단위를 체크해서 단위가 맞지 않으면 적절히 처리하면 됩니다.

물론 이 방법은 꽤나 괜찮아 보이지만 한 가지 문제가 있습니다. 만일 틀린 단위를 연산하는 코드가 매우 드물게 일어난다면 어떨까요? 즉 런타임에서 그 문제를 발견하지 못한 채 넘어갈 수 있다는 점입니다.

가장 이상적인 상황은 단위가 맞지 않는 연산을 수행하는 코드가 있다면 아예 컴파일 시에 오류를 발생시켜버리는 것입니다. 그렇다면 적어도 틀린 단위를 연산하는 일은 막을 수 있게 되고, 프로그램을 실행 시키면서 기다리는 수고를 줄일 수 있게 되지요.

이를 위해서 다음과 같은 클래스를 생각해봅시다.

```
template <typename U, typename V, typename W>
struct Dim {
    using M = U; // kg
    using L = V; // m
    using T = W; // s

    using type = Dim<M, L, T>;
};
```

이 `Dim`이라는 클래스는 어떠한 데이터의 단위를 나타내기 위해서 사용됩니다. 어떠한 물리량의 단위를 나타내기 위해서는 무게(kg), 길이(m), 시간(s) 이 3 개로 나타낼 수 있습니다. (실제로는 8개가 필요하지만 단순화를 위해 3개만 사용하도록 하겠습니다).

예를 들어서 속도의 경우 m/s 이므로, 저 `Dim` 클래스로 표현하자면 `Dim<0, 1, -1>` 로 나타낼 수 있습니다. 왜냐하면 $m/s = kg^0 m^1 s^{-1}$ 이기 때문이지요.

마찬가지로 힘의 경우 단위가 $kg\ m\ /s^2$ 이므로 `Dim` 클래스로 표현하자면 `Dim<1, 1, -2>` 가 됩니다.

물론 저 `Dim` 의 경우 템플릿 인자로 타입을 받기 때문에 단순히 `Dim<0, 1, -1>` 이렇게 사용할 수 있는 것이 아닙니다. 대신에 앞서 만들었던 `Ratio` 클래스를 이용해서 저 숫자들을 '타입'으로 표현해주어야 합니다. 따라서, 실제로는

```
Dim<1, 1, -2>
```

가 아니라

```
Dim<Ratio<1, 1>, Ratio<1, 1>, Ratio<-2, 1>>
```

이런 식으로 정의를 해야겠지요. 그렇다면 `Dim` 끼리 더하고 빼는 템플릿 클래스도 아래와 같이 만들 수 있게 됩니다.

```
template <typename U, typename V>
struct add_dim_ {
    typedef Dim<typename Ratio_add<typename U::M, typename V::M>::type,
                typename Ratio_add<typename U::L, typename V::L>::type,
                typename Ratio_add<typename U::T, typename V::T>::type>
        type;
};

template <typename U, typename V>
struct subtract_dim_ {
    typedef Dim<typename Ratio_subtract<typename U::M, typename V::M>::type,
                typename Ratio_subtract<typename U::L, typename V::L>::type,
                typename Ratio_subtract<typename U::T, typename V::T>::type>
        type;
};
```

왜 `typename` 이 저렇게도 많이 붙어있는지는 아마 잘 이해하실 거라 생각합니다. 왜냐하면 예를 들어 `M`의 경우 `U`에 의존한 타입이고, `type`의 경우도 마찬가지로 `U`와 `V`에 의존하는 타입이기 때문이지요.

자 이제, 실제 데이터를 담는 클래스를 만들어보도록 하겠습니다.

```
template <typename T, typename D>
struct quantity {
    T q;
    using dim_type = D;
};
```

일단 위처럼 `q`라는 멤버 변수에 데이터를 담고, (데이터의 타입은 `T`가 되겠지요), `dim_type`에 차원 정보를 담게 됩니다. 차원 정보는 데이터와는 다르게 'Dim 타입' 그 자체로 표현됩니다.

자 이제, 실제로 `quantity` 객체를 가지고 연산을 수행하기 위해서는 우리가 연산자들을 오버로드 해줘야만 합니다. 일단 간단히 + 와 - 연산자를 어떻게 오버로드 할지 생각해봅시다. 앞서 말했듯이, 두 개의 데이터를 더하거나 빼기 위해서는 반드시 단위가 일치해야 합니다. 이 말은, `dim_type`이 같은 타입이어야만 하다는 것이지요.

따라서 `operator+` 와 `operator-` 는 다음과 같이 간단히 정의할 수 있습니다.

```
quantity operator+(quantity<T, D> quant) { return quantity<T, D>(q + quant.q); }
quantity operator-(quantity<T, D> quant) { return quantity<T, D>(q - quant.q); }
```

위 `operator+` 는 인자로 받는 `quantity`의 데이터 타입과 Dim 타입이 일치해야지만 인스턴스화 됩니다. 만약에, 데이터 타입이나 Dim 타입이 일치하지 않았더라면 저 `operator+` 는 인스턴스화 될 수 없고 따라서 컴파일러는 저 `operator+` 를 찾을 수 없다는 오류를 발생시키게 됩니다!

그렇다면 실제로 테스트를 해볼까요.

```
#include <iostream>

template <int X, int Y>
struct GCD {
    static const int value = GCD<Y, X % Y>::value;
};

template <int X>
struct GCD<X, 0> {
    static const int value = X;
};

template <int N, int D = 1>
struct Ratio {
    private:
        const static int _gcd = GCD<N, D>::value;

    public:
        typedef Ratio<N / _gcd, D / _gcd> type;
        static const int num = N / _gcd;
        static const int den = D / _gcd;
```

```
};

template <class R1, class R2>
struct _Ratio_add {
    using type = Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_add : _Ratio_add<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_subtract {
    using type = Ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_subtract : _Ratio_subtract<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_multiply {
    using type = Ratio<R1::num * R2::num, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_multiply : _Ratio_multiply<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_divide {
    using type = Ratio<R1::num * R2::den, R1::den * R2::num>;
};

template <class R1, class R2>
struct Ratio_divide : _Ratio_divide<R1, R2>::type {};

template <typename U, typename V, typename W>
struct Dim {
    using M = U;
    using L = V;
    using T = W;

    using type = Dim<M, L, T>;
};

template <typename U, typename V>
struct add_dim_ {
    typedef Dim<typename Ratio_add<typename U::M, typename V::M>::type,
                typename Ratio_add<typename U::L, typename V::L>::type,
                typename Ratio_add<typename U::T, typename V::T>::type>
        type;
};

template <typename U, typename V>
```

```

struct subtract_dim_ {
    typedef Dim<typename Ratio_subtract<typename U::M, typename V::M>::type,
              typename Ratio_subtract<typename U::L, typename V::L>::type,
              typename Ratio_subtract<typename U::T, typename V::T>::type>
    type;
};

template <typename T, typename D>
struct quantity {
    T q;
    using dim_type = D;

    quantity operator+(quantity<T, D> quant) {
        return quantity<T, D>(q + quant.q);
    }

    quantity operator-(quantity<T, D> quant) {
        return quantity<T, D>(q - quant.q);
    }

    quantity(T q) : q(q) {}
};

int main() {
    using one = Ratio<1, 1>;
    using zero = Ratio<0, 1>;

    quantity<double, Dim<one, zero, zero>> kg(1);
    quantity<double, Dim<zero, one, zero>> meter(1);
    quantity<double, Dim<zero, zero, one>> second(1);

    // Good
    kg + kg;

    // Bad
    kg + meter;
}

```

컴파일 하였다면 다음과 같은 오류가 납니다.

컴파일 오류

```

no operator "+" matches these operands
binary '+': no operator found which takes a right-hand operand of
  ↳ type 'quantity<double,Dim<zero,one,zero>>' (or there is no
  ↳ acceptable conversion)

```

즉 위 + 에 해당하는 연산자 함수를 찾을 수 없다는 것이지요. 예상했던 대로,

```
// Bad
kg + meter;
```

위 부분에서 오류가 발생하는데, `kg` 와 `meter` 의 단위가 다르기 때문에 발생하게 됩니다. 반면에

// Good kg + kg;

는 잘 컴파일되지요.

그렇다면 이제 `*` 와 `/` 연산자만 만들어주면 되겠습니다. 하지만 `*` 와 `/`의 경우 `+` 와 `-` 보다 좀 더 까다롭습니다. 왜냐하면 `*` 와 `/`의 경우 굳이 `Dim` 이 일치하지 않아도 되거든요! 다만 이 연산을 수행하였을 때 새로운 차원의 데이터가 나올 뿐입니다.

예를 들어서 가속도를 나타내기 위해서는

```
meter / (second * second)
```

이렇게 해주면 됩니다. 다만 새로운 차원의 데이터 (`Dim<zero, one, minus_two>`) 가 탄생할 뿐이지요. 따라서, `operator*` 와 `operator/`의 경우 두 개의 다른 차원의 값을 받아도 처리할 수 있어야 합니다. 따라서 `operator*` 와 `/` 를 정의해보자면 아래와 같습니다.

```
template <typename D2>
quantity<T, typename add_dim_<D, D2>::type> operator*(quantity<T, D2> quant) {
    return quantity<T, typename add_dim_<D, D2>::type>(q * quant.q);
}

template <typename D2>
quantity<T, typename subtract_dim_<D, D2>::type> operator/(
    quantity<T, D2> quant) {
    return quantity<T, typename subtract_dim_<D, D2>::type>(q / quant.q);
}
```

새로 만들어지는 탑입의 차원은 당연히 `add_dim_<D, D2>::type` 이 되겠고 (`operator*`의 경우), 그 값은 그냥 실제 값을 곱해주면 됩니다. 이와 더불어서

```
3 * kg
```

과 같은 곱도 처리해야 하기 때문에, 아래와 같은 함수들도 정의해줘야 합니다.

```
quantity<T, D> operator*(T scalar) { return quantity<T, D>(q * scalar); }
quantity<T, D> operator/(T scalar) { return quantity<T, D>(q / scalar); }
```

이는 위 처럼 일반적인 차원이 없는 값 과의 곱도 지원해줍니다. 그렇다면 예를 들어서 아래와 같이 정의된 F 의 타입은 어떻게 될까요?

```
// F 의 타입은?  
F = kg * meter / (second * second);
```

일단 F 의 차원은 계산해보면 $(1, 1, -2)$ 이렇게 나올 것 입니다. 따라서, F 의 `dim` 타입은 `<Ratio<1, 1>, Ratio<1, 1>, Ratio<-2, 1>>` 가 되겠지요. 다시 말해, F 를 다음과 같이 나타낼 수 있습니다.

```
quantity<double, Dim<one, one, Ratio<-2, 1>>> F =  
    kg * meter / (second * second);
```

그런데, 매번 변수를 정의할 때마다 저렇게 길고 긴 타입을 써주는 것은 매우 귀찮은 일입니다. 저 `kg * meter / (second * second)` 를 계산해서 나오는 객체의 타입이 저렇게 된다는 사실은 저도 알고 컴파일러도 알고 있습니다. 컴파일러가 쉽게 알아낼 수 있는 타입을 굳이 우리가 써주어야 할까요? 똑똑한 컴파일러가 타입을 알아서 생각하도록 하면 안될까요?

물론 가능합니다.

타입을 알아서 추측해라! - auto 키워드

C++ 코드를 많이 짜면서 느꼈겠지만, 객체를 생성할 때, 많은 경우 굳이 타입을 쓰지 않아도 알아서 추측할 수 있는 경우들이 많이 있습니다.

예를 들어서,

```
(??) a = 3;
```

와 같이 썼다면 저 `(??)` 는 아마 `int` 를 의도한 것이겠지요. 아니면

```
some_class a;  
(??) b = a;
```

의 경우 저 `(??)` 에는 아마 `some_class` 가 들어가겠지요? 즉 객체가 복사 생성 될 때, 그 복사 생성하는 대상의 타입을 확실히 알 수 있다면 굳이 그 객체의 타입을 명시하지 않아도 컴파일러가 알아낼 수 있습니다.

물론 때때로 컴파일러가 타입을 제대로 유추할 수 없는 경우도 있습니다. 예를 들어서, 우리의 위 예제 코드에서

```
quantity<double, Dim<one, zero, zero>> kg(1);
```

의 경우 만약에 저 타입 부분을 가리고

```
(??) kg(1);
```

와 같이 살펴본다면 어떨까요? 컴파일러에 입장에서는 단순히 생각해봤을 때 그냥 1로 초기화하는 변수 이므로 (?)에는 int가 들어가겠지요. 따라서 이 경우에는 우리가 원하는 타입으로 생성할 수 없습니다. 반면에,

```
(??) F = kg * meter / (second * second);
```

F의 경우 우리가 굳이 타입을 적지 않아도 컴파일러가 오른쪽의 연산을 통해서 F의 타입을 정확하게 알아낼 수 있습니다.

이와 같이 컴파일러가 타입을 정확히 알아낼 수 있는 경우 굳이 그 길고 긴 타입을 적지 않고 간단히 auto로 표현할 수 있습니다. 그리고 그 auto에 해당하는 타입은 컴파일 시에 컴파일러에 의해 추론됩니다. 아래 간단한 예제를 살펴볼까요.

```
#include <iostream>
#include <typeinfo>

int sum(int a, int b) { return a + b; }

class SomeClass {
    int data;

public:
    SomeClass(int d) : data(d) {}
    SomeClass(const SomeClass& s) : data(s.data) {}
};

int main() {
    auto c = sum(1, 2); // 함수 리턴 타입으로부터 int라고 추측 가능
    auto num = 1.0 + 2.0; // double로 추측 가능!

    SomeClass some(10);
    auto some2 = some;

    auto some3(10); // SomeClass 객체를 만들까요?

    std::cout << "c의 타입은? :: " << typeid(c).name() << std::endl;
    std::cout << "num의 타입은? :: " << typeid(num).name() << std::endl;
    std::cout << "some2의 타입은? :: " << typeid(some2).name() << std::endl;
```

```
    std::cout << "some3 의 타입은? :: " << typeid(some3).name() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
c 의 타입은? :: i
num 의 타입은? :: d
some2 의 타입은? :: 9SomeClass
some3 의 타입은? :: i
```

와 같이 나옵니다.

```
std::cout << "c 의 타입은? :: " << typeid(c).name() << std::endl;
std::cout << "num 의 타입은? :: " << typeid(num).name() << std::endl;
std::cout << "some2 의 타입은? :: " << typeid(some2).name() << std::endl;
```

일단 위 3줄은 우리의 예상대로 `auto` 키워드가 잘 타입을 추론해줍니다. `c`의 경우 함수의 리턴 타입으로 부터 `int` 타입이라는 것을 알 수 있고, `num`의 경우 `1.0 + 2.0`의 결과가 `double` 이므로 `num` 역시 `double` 타입 변수로 초기화 됩니다. 마지막으로 `some2`의 경우 `SomeClass` 타입인 `some`으로 부터 복사 생성 되므로 `SomeClass` 타입이 되지요.

마지막으로 `some3` 를 살펴봅시다.

```
auto some3(10); // SomeClass 객체를 만들까요?
```

이전에 `some` 을 만들 때 `SomeClass some(10)` 으로 만들었기 때문에 저 `some3` 도 혹시 `SomeClass` 타입으로 추론하지 않을까 생각할 수 있습니다. 하지만 컴파일러는 최대한 단순하게 가능한 방법으로 추론하기 때문에 (실제로 `auto` 타입을 추론하는 방법은 템플릿에 들어갈 타입을 추론하는 것과 동일합니다), 그냥 `int` 변수로 만들어 버립니다.

하지만 아래의 `F` 의 경우 정확히 타입을 추론할 수 있기 때문에 그냥

```
// F 의 타입은 굳이 알필요 없다!
auto F = kg * meter / (second * second);
```

위와 같이 `auto` 키워드를 이용하면 됩니다.

참고로 편의를 위해 `quantity` 를 `ostream` 으로 출력해주는 함수인

```
template <typename T, typename D>
std::ostream& operator<<(std::ostream& out, const quantity<T, D>& q) {
    out << q.q << "kg^" << D::M::num / D::M::den << "m^" << D::L::num / D::L::den
    << "s^" << D::T::num / D::T::den;

    return out;
}
```

를 제작하였습니다. 따라서 전체 코드를 살펴보면 다음과 같습니다.

```
#include <iostream>
#include <typeinfo>

template <int X, int Y>
struct GCD {
    static const int value = GCD<Y, X % Y>::value;
};

template <int X>
struct GCD<X, 0> {
    static const int value = X;
};

template <int N, int D = 1>
struct Ratio {
private:
    const static int _gcd = GCD<N, D>::value;

public:
    typedef Ratio<N / _gcd, D / _gcd> type;
    static const int num = N / _gcd;
    static const int den = D / _gcd;
};

template <class R1, class R2>
struct _Ratio_add {
    using type = Ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_add : _Ratio_add<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_subtract {
    using type = Ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_subtract : _Ratio_subtract<R1, R2>::type {};
```

```
template <class R1, class R2>
struct _Ratio_multiply {
    using type = Ratio<R1::num * R2::num, R1::den * R2::den>;
};

template <class R1, class R2>
struct Ratio_multiply : _Ratio_multiply<R1, R2>::type {};

template <class R1, class R2>
struct _Ratio_divide {
    using type = Ratio<R1::num * R2::den, R1::den * R2::num>;
};

template <class R1, class R2>
struct Ratio_divide : _Ratio_divide<R1, R2>::type {};

template <typename U, typename V, typename W>
struct Dim {
    using M = U;
    using L = V;
    using T = W;

    using type = Dim<M, L, T>;
};

template <typename U, typename V>
struct add_dim_ {
    typedef Dim<typename Ratio_add<typename U::M, typename V::M>::type,
                typename Ratio_add<typename U::L, typename V::L>::type,
                typename Ratio_add<typename U::T, typename V::T>::type>
        type;
};

template <typename U, typename V>
struct subtract_dim_ {
    typedef Dim<typename Ratio_subtract<typename U::M, typename V::M>::type,
                typename Ratio_subtract<typename U::L, typename V::L>::type,
                typename Ratio_subtract<typename U::T, typename V::T>::type>
        type;
};

template <typename T, typename D>
struct quantity {
    T q;
    using dim_type = D;

    quantity operator+(quantity<T, D> quant) {
        return quantity<T, D>(q + quant.q);
    }

    quantity operator-(quantity<T, D> quant) {
```

```

        return quantity<T, D>(q - quant.q);
    }

template <typename D2>
quantity<T, typename add_dim_<D, D2>::type> operator*(quantity<T, D2> quant) {
    return quantity<T, typename add_dim_<D, D2>::type>(q * quant.q);
}

template <typename D2>
quantity<T, typename subtract_dim_<D, D2>::type> operator/(
    quantity<T, D2> quant) {
    return quantity<T, typename subtract_dim_<D, D2>::type>(q / quant.q);
}

// Scalar multiplication and division
quantity<T, D> operator*(T scalar) { return quantity<T, D>(q * scalar); }

quantity<T, D> operator/(T scalar) { return quantity<T, D>(q / scalar); }

quantity(T q) : q(q) {}

template <typename T, typename D>
std::ostream& operator<<(std::ostream& out, const quantity<T, D>& q) {
    out << q.q << "kg^" << D::M::num / D::M::den << "m^" << D::L::num / D::L::den
    << "s^" << D::T::num / D::T::den;

    return out;
}

int main() {
    using one = Ratio<1, 1>;
    using zero = Ratio<0, 1>;

    quantity<double, Dim<one, zero, zero>> kg(2);
    quantity<double, Dim<zero, one, zero>> meter(3);
    quantity<double, Dim<zero, zero, one>> second(1);

    // F 의 타입은 굳이 알필요 없다!
    auto F = kg * meter / (second * second);
    std::cout << "2 kg 물체를 3m/s^2 의 가속도로 밀기 위한 힘의 크기는? " << F
        << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

2 kg 물체를 3m/s² 의 가속도로 밀기 위한 힘의 크기는? 6kg¹m¹s⁻²

와 같이 잘 나옵니다.

`auto` 키워드는 템플릿의 사용으로 복잡해진 타입 이름들을 간단하게 나타낼 수 있는 획기적인 방법입니다. 물론 짧은 이름의 타입일 경우 그냥 써주는 것이 좋지만 (왜냐면 그 코드를 읽는 사람에 입장에서 한눈에 타입을 알 수 있으면 좋기 때문에), 위 경우 처럼 복잡한 타입 이름의 경우, 그 타입을 쉽게 추측할 수 있다면 `auto` 키워드를 활용하는 것도 좋습니다.

이것으로 템플릿 메타프로그래밍에 대한 강좌를 마치도록 하겠습니다. 사실 실제 현업에서 템플릿 메타 프로그래밍을 활용하는 경우는 그다지 많지 않습니다. 왜냐하면 일단 TMP의 특성상 복잡하고, 머리를 매우 많이 써야되고, 무엇보다도 버그가 발생하였을 때 찾는 것이 매우 힘듭니다.

하지만 우리의 `Unit` 클래스 처럼 TMP를 적절하게 활용하면 런타임에서 찾아야 하는 오류를 컴파일 타임에서 미리 다 잡아낼 수도 있고, 런타임 시에 수행해야 하는 연산들도 일부 컴파일 타임으로 옮길 수 있습니다.

만약에 TMP를 직접 작성할 일이 있다면 이미 TMP를 그나마 편하게 수행하기 위해 만들어진 `boost::MPL` 라이브러리가 있습니다. 이 라이브러리를 활용하신다면 비교적 쉽게 TMP 코드를 짤 수 있을 것입니다!

다음 강좌에서는 C++의 또 다른 막강한 무기인 표준 라이브러리 (STL)에 대해 알아보도록 하겠습니다!

생각 해보기

문제 1

컴파일러가 `auto` 키워드에 들어갈 타입을 추측하는 방법은 템플릿에서 들어갈 타입을 추측하는 방법과 같습니다. [여기를 클릭해서 읽어보세요!](#)

C++ 표준 라이브러리 (컨테이너와 알고리즘)

C++ 표준 컨테이너

안녕하세요 여러분! 지난번 템플릿 메타프로그래밍 강좌는 어떠셨나요? TMP를 활용해서 프로그래밍을 하는 것은 엄청 머리아픈 일이지만 적당히 잘 쓰면 꽤 괜찮은 도구입니다.

하지만 이번 강좌는 조금 다릅니다. 이번 강좌에서 배우게 될 C++의 표준 템플릿 라이브러리(STL)은 사용하는 것도 엄청 간단한데, 여러분이 하는 프로그래밍 능률을 100% 향상 시킬 수 있는 엄청난 도구입니다. 사실 이 STL의 도입으로 C++이 한발 더 도약한 것도 과언이 아니라 볼 수 있습니다.

C++ 표준 템플릿 라이브러리 (Standard Template Library - STL)

사실 C++ 표준 라이브러리를 보면 꽤나 많은 종류의 라이브러리들이 있습니다. 예를 들어서, 대표적으로 입출력 라이브러리 (iostream 등등), 시간 관련 라이브러리 (chrono), 정규표현식 라이브러리 (regex) 등등 들이 있지요. 하지만 보통 C++ 템플릿 라이브러리(STL)를 일컫는다면 다음과 같은 세 개의 라이브러리들을 의미합니다.

- 임의 타입의 객체를 보관할 수 있는 컨테이너 (container)
- 컨테이너에 보관된 원소에 접근할 수 있는 반복자 (iterator)
- 반복자들을 가지고 일련의 작업을 수행하는 알고리즘 (algorithm)

각 라이브러리의 역할을 쉽게 생각하면 다음과 같이 볼 수 있습니다. 여러분이 우편 배달부가 되어서 편지들을 여러개의 편지함에 넣는다고 생각해봅시다. 편지를 보관하는 각각의 편지함들은 '컨테이너'라고 생각하시면 됩니다. 그리고, 편지를 보고 원하는 편지함을 찾는 일은 '반복자'들이 수행하지요. 마지막으로, 만일 편지들을 편지함에 날짜 순서로 정렬하여 넣는 일은 '알고리즘'이 수행할 것입니다.

한 가지 주목할 만한 점은

- 임의 타입의 객체를 보관할 수 있는 컨테이너 (container)

에서 나타나 있듯이 우리가 다루려는 객체가 어떤 특성을 갖는지 무관하게 라이브러리를 자유롭게 사용할 수 있다는 것입니다 (바로 템플릿 덕분이죠!). 우리가 만일 사용하려는 자료형이 `int` 나 `string`과 같은 평범한 애들이 아니라, 우리가 만든 임의의 클래스의 객체들이여도 자유롭게 위 라이브러리의 기능들을 모두 활용할 수 있습니다. 만일 C 였다면 불가능했을 일입니다.

또한 반복자의 도입으로 알고리즘 라이브러리에 필요한 최소한의 코드만을 작성할 수 있게 되었습니다. 다시 말하면, 기존의 경우 M 개 종류의 컨테이너가 있고 N 종류의 알고리즘이 있다면 이 모든 것을 지원하려면 MN 개의 알고리즘 코드가 있어야만 했습니다.

하지만 반복자를 이용해서 컨테이너를 추상화 시켜서 접근할 수 있기 때문에 N 개의 알고리즘 코드만으로 M 종류의 컨테이너들을 모두 지원할 수 있게됩니다. (후에 알고리즘 라이브러리에 대해서 설명할 때 더 와닿을 것입니다)

C++ STL 컨테이너 - 벡터 (`std::vector`)

C++ STL에서 컨테이너는 크게 두 가지 종류가 있습니다. 먼저 배열 처럼 객체들을 순차적으로 보관하는 시퀀스 컨테이너 (**sequence container**) 와 키를 바탕으로 대응되는 값을 찾아주는 연관 컨테이너 (**associative container**) 가 있습니다.

먼저 시퀀스 컨테이너의 경우 `vector`, `list`, `deque` 이렇게 3 개가 정의되어 있습니다. 먼저 벡터(`vector`)의 경우, 쉽게 생각하면 가변길이 배열이라 보시면 됩니다 (템플릿 강의에서 `Vector`를 제작하신 것을 기억 하시나요?) 벡터에는 원소들이 메모리 상에서 실제로 순차적으로 저장되어 있고, 따라서 임의의 위치에 있는 원소를 접근하는 것을 매우 빠르게 수행할 수 있습니다.

정확히 얼마나 빠르다고?

사실 '매우 빠르다'라는 말은 주관적일 수 밖에 없습니다. 따라서 어떠한 작업의 수행 속도를 나타내기 위해선 수학적으로 나타내야 합니다.

컴퓨터 공학에선 어떠한 작업의 처리 속도를 복잡도(**complexity**)라고 부르고, 그 복잡도를 **Big O** 표기법이라는 것으로 나타냅니다. 이 표기법은, N 개의 데이터가 주어져 있을 때 그 작업을

수행하기 위해 몇 번의 작업을 필요로 하는지 N 에 대한 식으로 표현하는 방식입니다. (즉 복잡도가 클 수록 작업이 수행되는데 걸리는 시간이 늘어나겠지요)

예를 들어 가장 기초적인 버블 정렬을 생각해봅시다. 버블 정렬의 코드는 간단히 보자면 아래와 같습니다.

```
for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
        if (arr[i] > arr[j]) {
            swap(arr, i, j)
        }
    }
}
```

따라서 N 개의 원소가 있는 `arr`이라는 배열을 정렬하기 위해서는 일단 적어도

$$\frac{N(N - 1)}{2}$$

번의 반복이 필요하지요 ($(N - 1 + N - 2 + \dots + 1)$) 따라서 Big O 표현법으로 이 정렬이 얼마나 빠르게 수행될 수 있는지 나타내면

$$O\left(\frac{N(N - 1)}{2}\right)$$

라고 볼 수 있습니다. 보통 Big O 표현법으로 나타낼 때, 최고차항만을 나타냅니다 (그리고 통상적으로 최고차항의 계수도 생략합니다). 왜냐하면 N 이 엄청 커지게 되면 최고 차항 말고는 그닥 의미가 없게 되버리기 때문이지요 (최고 차항에 비해 크기가 너무 작기 때문에). 따라서 최종적으로, 버블 정렬 알고리즘의 복잡도는

$$O(N^2)$$

라고 볼 수 있습니다. 일반적으로 어떠한 알고리즘이 $O(N^2)$ 꼴이면 그닥 좋은 편은 아닙니다. 왜냐하면 N 이 10000만 되더라도, 10^8 번의 작업을 처리해야 하기 때문이죠. 다행이도 정렬 알고리즘의 경우 쿼크소트(Quicksort)라는 알고리즘을 활용하면 아래와 같은 복잡도로 연산을 처리할 수 있습니다.

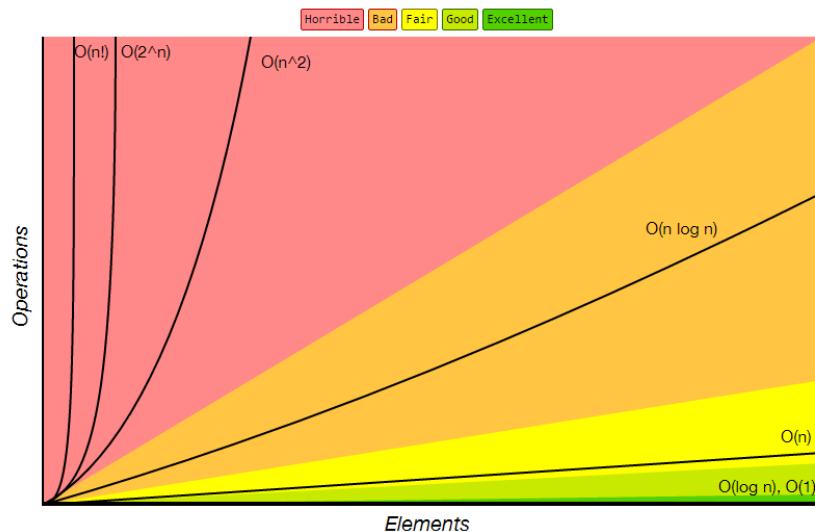
$$O(N \log N)$$

물론 쿼크소트 알고리즘을 사용했을 때 항상 버블 정렬 방식 보다 빠르게 정렬할 수 있다는 의미는 아닙니다. 왜냐하면 저 항 앞에 어떠한 계수가 붙어있는지 알 수 없기 때문이지요. 만약에 버블 정렬이 $O(N^2)$ 이고 쿼크소트가 $O(100000N \log N)$ 이었다면 N 이 1000일 때 버블 정렬이 더 빠르게

수행됩니다. (물론 이렇게 극단적이지 않습니다. 쿼소트가 거의 대부분 더 빠르게 됩니다!)

하지만, N 이 정말 커진다면 언젠가는 쿼소트가 베를 정렬보다 더 빨리 수행되는 때가 발생합니다.

아래 그림을 보면 각각의 O 에 대해 복잡도가 어떻게 증가하는지 볼 수 있습니다.



가장 이상적인 복잡도는 $O(1)$ 이지만 이는 거의 불가능하고 (이는 마치 전체 데이터를 채 보지 않은 채 작업을 끝낼 수 있다는 의미입니다), 보통 $O(\log n)$ 이 알고리즘이 낼 수 있는 가장 빠른 속도를 의미합니다. 그 다음으로 좋은 것이 당연히 $O(n)$ 이고, $O(n \log n)$ 순입니다.

그렇다면 다시 벡터 자료형으로 돌아오겠습니다. `vector`의 경우, 임의의 위치에 있는 원소에 접근을 $O(1)$ 로 수행할 수 있습니다. 게다가 맨 뒤에 새로운 원소를 추가하거나 제거하는 것 역시 $O(1)$ 에 수행합니다. `vector`의 임의의 원소에 접근하는 것은 배열처럼 []를 이용하거나, `at` 함수를 이용하면 됩니다. 또한 맨 뒤에 원소를 추가하거나 제거하기 위해서는 `push_back` 혹은 `pop_back` 함수를 사용하면 됩니다. 아래 예를 보겠습니다.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;
    vec.push_back(10); // 맨 뒤에 10 추가
    vec.push_back(20); // 맨 뒤에 20 추가
    vec.push_back(30); // 맨 뒤에 30 추가
    vec.push_back(40); // 맨 뒤에 40 추가

    for (std::vector<int>::size_type i = 0; i < vec.size(); i++) {
        std::cout << "vec 의 " << i + 1 << " 번째 원소 :: " << vec[i] << std::endl;
    }
}
```

성공적으로 컴파일 하였으면

실행 결과

```
vec 의 1 번째 원소 :: 10
vec 의 2 번째 원소 :: 20
vec 의 3 번째 원소 :: 30
vec 의 4 번째 원소 :: 40
```

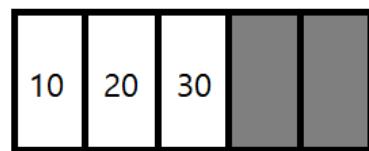
와 같이 우리가 넣은 순서대로 잘 나옴을 알 수 있습니다.

참고로 벡터의 크기를 리턴하는 함수인 `size`의 경우, 그리던하는 값의 타입은 `size_type` 멤버 타입으로 정의되어 있습니다.

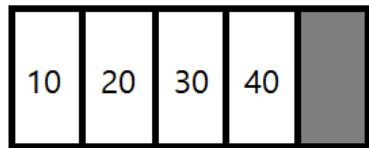
참고로 맨 뒤에 원소를 추가하는 작업은 엄밀히 말하자면 **amortized $O(1)$** 이라고 합니다. (`amortized`의 뜻은 분할상환이란 뜻인데, 아마 아래 설명을 읽으시면 왜 그런 이름을 붙였는지 이해하실 수 있을 것입니다)

왜냐면 보통은 `vector`의 경우 현재 가지고 있는 원소의 개수 보다 더 많은 공간을 할당해 놓고 있습니다. 예를 들어 현재 `vector`에 있는 원소의 개수가 10개라면 이미 20개를 저장할 수 있는 공간을 미리 할당해놓게 됩니다. 따라서 만약에 뒤에 새로운 원소를 추가하게 된다면 새롭게 메모리를 할당할 필요가 없이, 그냥 이미 할당된 공간에 그 원소를 쓰기만 하면 됩니다. 따라서 대부분의 경우 $O(1)$ 으로 `vector` 맨 뒤에 새로운 원소를 추가하거나 지울 수 있습니다.

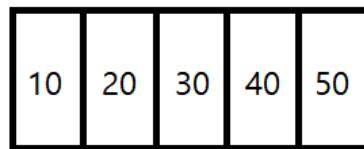
문제가 되는 상황은 할당된 공간을 다 채웠을 때입니다. 이 때는 어쩔 수 없이, 새로운 큰 공간을 다시 할당하고, 기존의 원소들을 복사하는 수 밖에 없습니다. 따라서 이 경우 n 개의 원소를 모두 복사해야 하기 때문에 $O(n)$ 으로 수행됩니다. 하지만 이 $O(n)$ 으로 수행되는 경우가 매우 드물기 때문에, 전체적으로 평균을 내보았을 때 $O(1)$ 으로 수행될 것을 알 수 있습니다. 이렇게 amortized $O(1)$ 이라고 부르게 됩니다. 아래 그림에서 자세히 설명하고 있습니다.



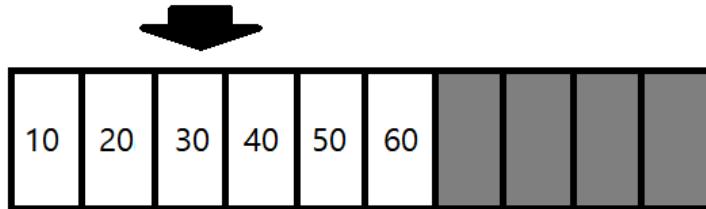
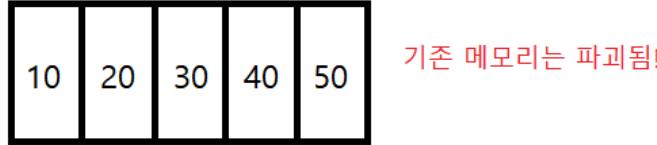
현재 벡터 길이보다 좀 더 많은 공간이 미리 할당되어 있다!



`vec.push_back(40);`
을 수행하면 그냥 미리 할당된 공간에 복사하면 된다. O(1)



`vec.push_back(50);` 까지는 문제없다!



`vec.push_back(60);`
을 수행하면 기존의 할당된 공간이 다 차기 때문에 새롭게
더 큰 메모리를 할당하고 복사를 수행해야한다 O(n)

물론 `vector` 라고 만능은 아닙니다. 맨 뒤에 원소를 추가하거나 제거하는 것은 빠르지만, 임의의 위치에 원소를 추가하거나 제거하는 것은 $O(n)$ 으로 느립니다. 왜냐하면 어떤 자리에 새로운 원소를 추가하거나 뺄 경우 그 뒤에 오는 원소들을 한 칸씩 이동시켜 주어야만 하기 때문이지요. 따라서 이는 n 번의 복사가 필요로 합니다.

따라서 만일 맨 뒤가 아닌 위치에 데이터를 추가하거나 제거하는 작업이 많은 일일 경우 `vector` 를 사용하면 안되겠지요. 결과적으로 `vector` 의 복잡도를 정리해보자면 아래와 같습니다.

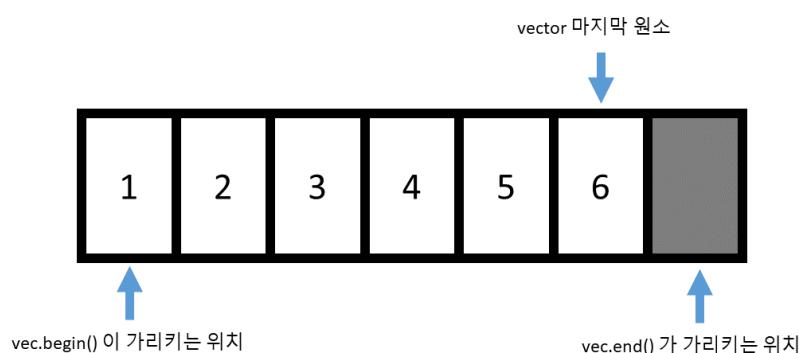
- 임의의 위치 원소 접근 ([], at) : $O(1)$
- 맨 뒤에 원소 추가 및 제거 (push_back/pop_back) : amortized $O(1)$; (평균적으로 $O(1)$ 이지만 최악의 경우 $O(n)$)
- 임의의 위치 원소 추가 및 제거 (insert, erase) : $O(n)$

위처럼 어떠한 작업을 하냐에 따라서 속도차가 매우 크기 때문에, C++ 표준 라이브러리를 잘 사용하기 위해서는 내가 이 컨테이너를 어떠한 작업을 위해 사용하는지 정확히 인지하고, 적절한 컨테이너를 골라야 합니다. 후에 설명할 다른 자료 구조를 사용하면 `vector` 가 빠른 작업이 느릴 수도 있고, `vector` 가 느린 작업을 빠르게 할 수도 있습니다.

반복자 (iterator)

앞서 반복자는 컨테이너에 원소에 접근할 수 있는 포인터와 같은 객체라고 하였습니다. 물론 벡터의 경우 [] 를 이용해서 정수형 변수로 마치 배열처럼 임의의 위치에 접근할 수 있지만, 반복자를 사용해서도 마찬가지 작업을 수행할 수 있습니다. 특히 후에 배울 알고리즘 라이브러리의 경우 대부분이 반복자를 인자로 받아서 알고리즘을 수행합니다.

반복자는 컨테이너에 `iterator` 멤버 타입으로 정의되어 있습니다. `vector` 의 경우 반복자를 얻기 위해서는 `begin()` 함수와 `end()` 함수를 사용할 수 있는데 이는 다음과 같은 위치를 리턴합니다.



`begin()` 함수는 예상했던 대로, `vector` 의 첫번째 원소를 가리키는 반복자를 리턴합니다. 그런데, 흥미롭게도 `end()` 의 경우 `vector` 의 마지막 원소 한 칸 뒤를 가리키는 반복자를 리턴하게 됩니다. 왜 `end` 의 경우 `vector` 의 마지막 원소를 가리것이 아니라, 마지막 원소의 뒤를 가리키는 반복자를 리턴할까요?

이에 여러가지 이유가 있겠지만, 가장 중요한 점이 이를 통해 빈 벡터를 표현할 수 있다는 점입니다. 만일 `begin() == end()` 라면 원소가 없는 벡터를 의미하겠지요. 만약에 `vec.end()` 가 마지막 원소를 가리킨다면 비어있는 벡터를 표현할 수 없게 됩니다.

```
// 반복자 사용 예시
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    // 전체 벡터를 출력하기
    for (std::vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr) {
        std::cout << *itr << std::endl;
    }

    // int arr[4] = {10, 20, 30, 40}
    // *(arr + 2) == arr[2] == 30;
    // *(itr + 2) == vec[2] == 30;

    std::vector<int>::iterator itr = vec.begin() + 2;
    std::cout << "3 번째 원소 :: " << *itr << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
10
20
30
40
3 번째 원소 :: 30
```

와 같이 잘 수행됨을 알 수 있습니다.

```
// 전체 벡터를 출력하기
for (std::vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr) {
    std::cout << *itr << std::endl;
}
```

`vector` 의 반복자의 타입은 위 처럼 `std::vector<>::iterator` 멤버 타입으로 정의되어 있고, `vec.begin()` 이나 `vec.end()` 함수가 이를 리턴합니다. `end()` 가 `vector` 의 마지막 원소

바로 뒤를 가리키기 때문에 `for` 문에서 `vector` 전체 원소를 보고 싶다면 `vec.end()` 가 아닐 때 까지 반복하면 됩니다.

앞서 반복자를 마치 포인터처럼 사용한다고 하였는데, 실제로 현재 반복자가 가리키는 원소의 값을 보고 싶다면;

```
std::cout << *itr << std::endl;
```

포인터로 `*` 를 해서 가리키는 주소값의 값을 보았던 것처럼, `*` 연산자를 이용해서 `itr` 이 가리키는 원소를 볼 수 있습니다. 물론 `itr` 은 실제 포인터가 아니고 `*` 연산자를 오버로딩해서 마치 포인터처럼 동작하게 만든 것입니다. `*` 연산자는 `itr` 이 가리키는 원소의 레퍼런스를 리턴합니다.

```
std::vector<int>::iterator itr = vec.begin() + 2;
std::cout << "3 번째 원소 :: " << *itr << std::endl;
```

또한 반복자 역시 `+` 연산자를 통해서 그 만큼 떨어져 있는 원소를 가리키게 할 수도 있습니다. (그냥 배열을 가리키는 포인터와 정확히 똑같이 동작한다고 생각하시면 됩니다!)

반복자를 이용하면 아래와 같이 `insert` 와 `erase` 함수도 사용할 수 있습니다.

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
         ++itr) {
        std::cout << *itr << std::endl;
    }
}
int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "처음 벡터 상태" << std::endl;
    print_vector(vec);
    std::cout << "-----" << std::endl;

    // vec[2] 앞에 15 추가
    vec.insert(vec.begin() + 2, 15);
    print_vector(vec);
}
```

```

std::cout << "-----" << std::endl;
// vec[3] 제거
vec.erase(vec.begin() + 3);
print_vector(vec);
}

```

성공적으로 컴파일 하였다면

실행 결과

처음 벡터 상태

```

10
20
30
40
-----
```

```

10
20
15
30
40
-----
```

```

10
20
15
40
```

와 같이 잘 나옵니다.

참고로 템플릿 버전의 경우,

```

for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
      ++itr) {
```

와 같이 앞에 **typename** 을 추가해줘야만 합니다. 그 이유는, **iterator** 가 **std::vector<T>** 의 의존 타입이기 때문입니다. **의존 타입이 무엇인지 기억 안나시는 분은 이 강좌를 참조하시기 바랍니다**

```

// vec[2] 앞에 15 추가
vec.insert(vec.begin() + 2, 15);
```

앞서 `insert` 함수를 소개하였는데, 위 처럼 인자로 반복자를 받고, 그 반복자 앞에 원소를 추가해 줍니다. 위 경우 `vec.begin() + 2` 앞에 15를 추가하므로 10, 20, 30, 40에서 10, 20, 15, 30, 40이 됩니다.

```
vec.erase(vec.begin() + 3);
print_vector(vec);
```

또 아까전에 언급하였던 `erase`도 인자로 반복자를 받고, 그 반복자가 가리키는 원소를 제거합니다. 위 경우 4 번째 원소인 30이 지워지겠지요. 물론 `insert`과 `erase` 함수 모두 $O(n)$ 으로 느린편입니다.

참고로 `vector`에서 반복자로 `erase`나 `insert` 함수를 사용할 때 주의해야 할 점이 있습니다.

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    std::cout << "[ ";
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
         ++itr) {
        std::cout << *itr << " ";
    }
    std::cout << "]";
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);
    vec.push_back(20);

    std::cout << "처음 벡터 상태" << std::endl;
    print_vector(vec);

    std::vector<int>::iterator itr = vec.begin();
    std::vector<int>::iterator end_itr = vec.end();

    for (; itr != end_itr; ++itr) {
        if (*itr == 20) {
            vec.erase(itr);
        }
    }

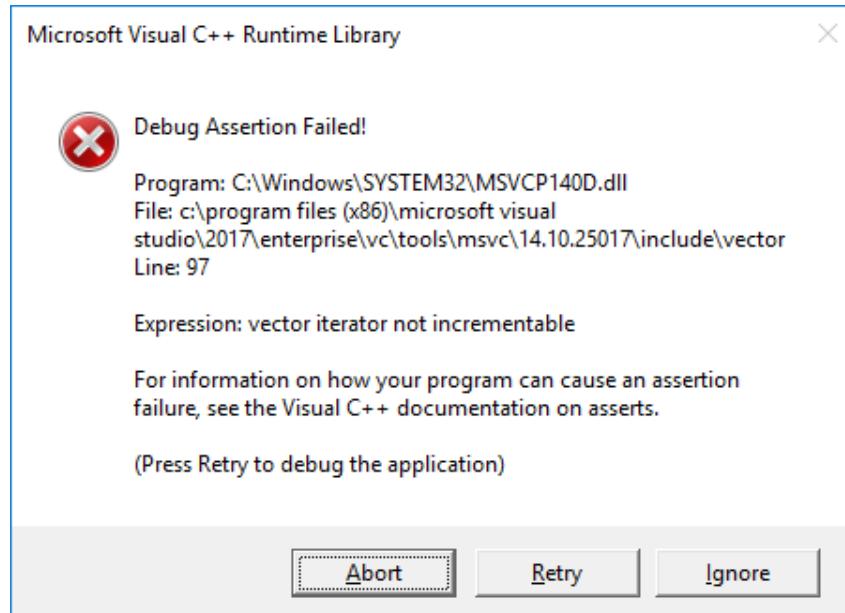
    std::cout << "값이 20인 원소를 지운다!" << std::endl;
}
```

```

    print_vector(vec);
}

```

컴파일 후 실행하였다면 아래와 같은 오류가 발생합니다.



왜 이런 오류가 발생하는 것일까요?

```

for ( ; itr != end_itr; itr++) {
    if (*itr == 20) {
        vec.erase(itr);
    }
}

```

문제는 바로 위 코드에서 발생합니다. 컨테이너에 원소를 추가하거나 제거하게 되면 기존에 사용하였던 모든 반복자들을 사용할 수 없게됩니다. 다시 말해 위 경우 `vec.erase(itr)` 을 수행하게 되면 더이상 `itr` 은 유효한 반복자가 아니게 되는 것이지요. 또한 `end_itr` 역시 무효화 됩니다.

따라서 `itr != end_itr` 이 영원히 성립되며 무한 루프에 빠지게되어 위와 같은 오류가 발생합니다.

그렇다면

```

std::vector<int>::iterator itr = vec.begin();
for ( ; itr != vec.end(); itr++) {
    if (*itr == 20) {
        vec.erase(itr);
    }
}

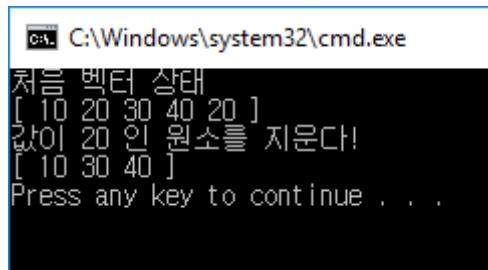
```

와 같이 코드를 고치면 오류가 없어질까요? 실행해보시면 알겠지만 여전히 위와 같은 오류가 발생합니다. 왜냐하면 `itr` 이 유효한 반복자가 아니기 때문에 `vec.end()` 로 올바른 `end` 반복자 값을 매번 가지고 와도 `for` 문이 끝나지 않게 되는 것입니다. 결과적으로 코드를 제대로 고치려면 다음과 같이 해야 합니다.

```
std::vector<int>::iterator itr = vec.begin();

for ( ; itr != vec.end(); ++itr) {
    if (*itr == 20) {
        vec.erase(itr);
        itr = vec.begin();
    }
}
```

성공적으로 컴파일 하였다면



와 같이 제대로 값이 20 인 원소만 지워졌음을 알 수 있습니다.

사실 생각해보면 위 바뀐 코드는 꽤나 비효율적임을 알 수 있습니다. 왜냐하면 20 인 원소를 지우고, 다시 처음으로 돌아가서 원소들을 찾고 있기 때문이지요. 그냥 20 인 원소 바로 다음 위치 부터 찾아나가면 될 텐데 말입니다.

```
for (std::vector<int>::size_type i = 0; i != vec.size(); i++) {
    if (vec[i] == 20) {
        vec.erase(vec.begin() + i);
        i--;
    }
}
```

그렇다면 아예 위 처럼 굳이 반복자를 쓰지 않고 `erase` 함수에만 반복자를 바로 만들어서 전달하면 됩니다.

```
vec.erase(vec.begin() + i);
```

를 하게 되면 `vec[i]` 를 가리키는 반복자를 `erase` 에 전달할 수 있습니다. 하지만 사실 위 방법은 그리 권장하는 방법은 아닙니다. 기껏 원소에 접근하는 방식은 반복자를 사용하는 것으로 통일하

였는데, 위 방법은 이를 모두 깨버리고 그냥 기존의 배열처럼 정수형 변수 *i*로 원소에 접근하는 것이기 때문입니다.

하지만 후에 C++ 알고리즘 라이브러리에 대해 배우면서 이 문제를 깔끔하게 해결 하는 방법에 대해 다루도록 할 것입니다. 일단 임시로는 위 방법처럼 처리하도록 하세요 :)

`vector`에서 지원하는 반복자로 `const_iterator`가 있습니다. 이는 마치 `const` 포인터를 생각하시면 됩니다. 즉, `const_iterator`의 경우 가리키고 있는 원소의 값을 바꿀 수 없습니다. 예를 들어서

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
         ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "초기 vec 상태" << std::endl;
    print_vector(vec);

    // itr 은 vec[2] 를 가리킨다.
    std::vector<int>::iterator itr = vec.begin() + 2;

    // vec[2] 의 값을 50으로 바꾼다.
    *itr = 50;

    std::cout << "-----" << std::endl;
    print_vector(vec);

    std::vector<int>::const_iterator citr = vec.cbegin() + 2;

    // 상수 반복자가 가리키는 값은 바꿀수 없다. 불가능!
    *citr = 30;
}
```

컴파일 하였다면

컴파일 오류

```
'citr': you cannot assign to a variable that is const
```

와 같이, `const` 반복자가 가리키고 있는 값을 바꿀 수 없다고 오류가 발생합니다. 주의할 점은, `const` 반복자의 경우

```
std::vector<int>::const_iterator citr = vec.cbegin() + 2;
```

와 같이 `cbegin()` 과 `cend()` 함수를 이용하여 얻을 수 있습니다. 많은 경우 반복자의 값을 바꾸지 않고 참조만 하는 경우가 많으므로, `const iterator` 를 적절히 이용하는 것이 좋습니다.

`vector` 에서 지원하는 반복자 중 마지막 종류로 역반복자 (reverse iterator) 가 있습니다. 이는 반복자와 똑같지만 벡터 뒤에서 부터 앞으로 거꾸로 간다는 특징이 있습니다. 아래 예제를 살펴볼까요.

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
         ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "초기 vec 상태" << std::endl;
    print_vector(vec);

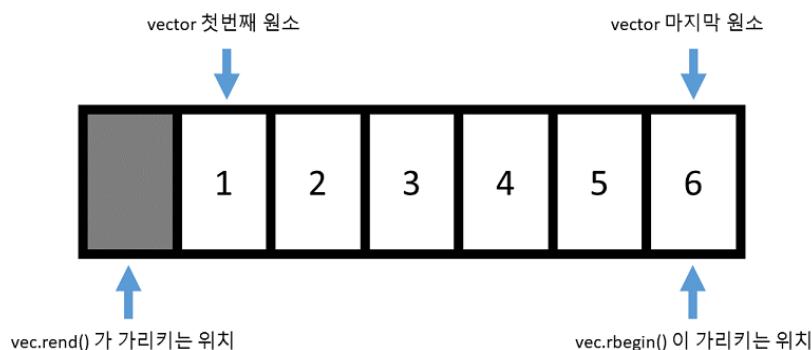
    std::cout << "역으로 vec 출력하기!" << std::endl;
    // itr 은 vec[2] 를 가리킨다.
    std::vector<int>::reverse_iterator r_iter = vec.rbegin();
    for (; r_iter != vec.rend(); r_iter++) {
        std::cout << *r_iter << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
초기 vec 상태
10
20
30
40
역으로 vec 출력하기!
40
30
20
10
```

와 같이 역으로 벡터의 원소들을 출력할 수 있습니다.



이전에 반복자의 `end()` 가 맨 마지막 원소의 바로 뒤를 가리켰던 것처럼, 역반복자의 `rend()` 역시 맨 앞 원소의 바로 앞을 가리키게 됩니다. 또한 반복자의 경우 값이 증가하면 뒤쪽 원소로 가는 것처럼, 역반복자의 경우 값이 증가하면 앞쪽 원소로 가게 됩니다.

또 반복자가 상수 반복자가 있는 것처럼 역반복자 역시 상수 역반복자가 있습니다. 그 타입은 `const_reverse_iterator` 타입이고, `crbegin()`, `crend()`로 얻을 수 있습니다.

역반복자를 사용하는 것은 매우 중요합니다. 아래와 같은 코드를 살펴볼까요.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
```

```
// 끝에서부터 출력하기
for (std::vector<int>::size_type i = vec.size() - 1; i >= 0; i--) {
    std::cout << vec[i] << std::endl;
}

return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
3
2
1
// ... (생략) ...
0
0
0
1
0
593
0
0
[1] 22180 segmentation fault (core dumped) ./test
```

와 같이 오류가 발생하게 됩니다. 맨 뒤의 원소 부터 제대로 출력하는 코드 같은데 왜 이런 문제가 발생하였을까요? 그 이유는 `vector` 의 `index` 를 담당하는 타입이 부호 없는 정수 이기 때문입니다. 따라서 `i` 가 0 일 때 `i --` 를 하게 된다면 -1 이 되는 것이 아니라, 해당 타입에서 가장 큰 정수가 되버리게 됩니다.

따라서 `for` 문이 영원히 종료할 수 없게 되죠.

이 문제를 해결하기 위해서는 부호 있는 정수로 선언해야 하는데, 이 경우 `vector` 의 `index` 타입과 일치하지 않아서 타입 캐스팅을 해야 한다는 문제가 발생하게 됩니다.

따라서 가장 현명한 선택으로는 역으로 원소를 참조하고 싶다면, 역반복자를 사용하는 것입니다.

범위 기반 for 문 (range based for loop)

위와 같이 컨테이너의 원소를 `for` 문으로 접근하는 패턴은 매우 많이 등장하는데, C++ 11 에서부터는 이와 같은 패턴을 매우 간단하게 나타낼 수 있는 방식을 제공하고 있습니다. 바로 범위 기반(range-based) `for` 문이라 불리는 것입니다.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);

    // range-based for 문
    for (int elem : vec) {
        std::cout << "원소 : " << elem << std::endl;
    }

    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
원소 : 1
원소 : 2
원소 : 3
```

와 같이 나옵니다. 범위 기반 `for` 문의 경우 아래와 같은 형태로 써주시면 됩니다.

```
for /* 원소를 받는 변수 정의 */ : /* 컨테이너 */ {  
}
```

위 경우

```
for (int elem : vec) {  
    std::cout << "원소 : " << elem << std::endl;  
}
```

의 형태로 썼을 경우, `elem`에 `vec`의 원소들이 매 루프마다 복사되어서 들어가게 됩니다. 마치

```
elem = vec[i];
```

를 한 것과 말이지요. 만약에 복사 하기 보다는 레퍼런스를 받고 싶다면 어떨까요? 매우 간단합니다. 단순히 레퍼런스 타입으로 바꿔버리면 되죠. 예를 들어서 기존의 `print_vec` 함수를 범위 기반 `for` 문을 사용해서 어떻게 바꿀 수 있는지 살펴봅시다.

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
         ++itr) {
        std::cout << *itr << std::endl;
    }
}

template <typename T>
void print_vector_range_based(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (const auto& elem : vec) {
        std::cout << elem << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "print_vector" << std::endl;
    print_vector(vec);
    std::cout << "print_vector_range_based" << std::endl;
    print_vector_range_based(vec);

    return 0;
}
```

실행 결과

```
print_vector
1
```

```

2
3
4
print_vector_range_based
1
2
3
4

```

와 같이 동일하게 나타남을 알 수 있습니다.

```

for (const auto& elem : vec) {
    std::cout << elem << std::endl;
}

```

위와 같이 `const auto&`로 `elem`을 선언하였으므로, `elem`은 `vec`의 원소들을 상수 레퍼런스로 접근하게 됩니다.

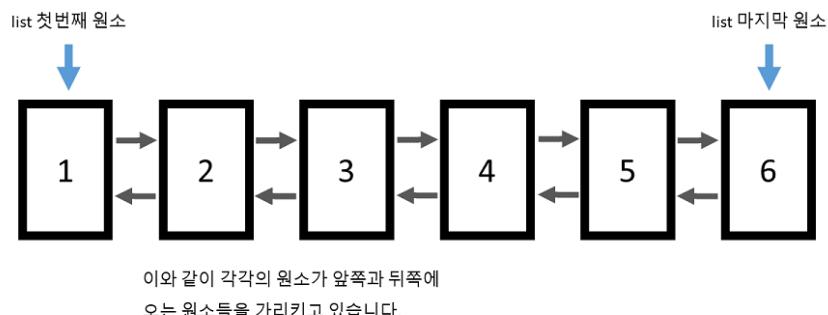
이와 같이 범위 기반 `for` 문을 활용한다면 코드를 직관적으로 나타낼 수 있어서 매우 편리합니다.

참고로 앞서 설명한 함수들 말고도 `vector`에는 수 많은 함수들이 있고, 또 오버로드 되는 여러가지 버전들이 있습니다.

예를 들어 `insert` 함수만 해도 5 개의 오버로드 되는 버전들이 있습니다 (물론 하는 역할은 똑같지만 편의를 위해 여러가지 방식으로 사용할 수 있게 만들어 놓은 것입니다). 이 모든 것들을 강좌에서 소개하는 것은 시간 낭비이고, [C++ 레퍼런스를 보면 잘 정리되어 있으니 이를 참조하시기 바랍니다.](#)

리스트 (list)

리스트(list)의 경우 양방향 연결 구조를 가진 자료형이라 볼 수 있습니다!



따라서 `vector` 와는 달리 임의의 위치에 있는 원소에 접근을 바로 할 수 없습니다. `list` 컨테이너 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 합니다.

그래서 리스트에는 아예 [] 나 `at` 함수가 아예 정의되어 있지 않습니다.

물론 리스트의 장점이 없는 것은 아닙니다. `vector` 의 경우 맨 뒤를 제외하고는 임의의 위치에 원소를 추가하거나 제거하는 작업이 $O(n)$ 이였지만 리스트의 경우 $O(1)$ 으로 매우 빠르게 수행될 수 있습니다. 왜냐하면 원하는 위치 앞과 뒤에 있는 링크값만 바꿔주면 되기 때문입니다.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst;

    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        std::cout << *itr << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
10
20
30
40
```

와 같이 잘 나옵니다.

한 가지 재미있는 점은 리스트의 반복자의 경우 다음과 같은 연산밖에 수행할 수 없습니다.

```
itr++ // itr ++
itr-- // --itr 도 됩니다.
```

다시 말해

```
itr + 5 // 불가능!
```

와 같이 임의의 위치에 있는 원소를 가리킬 수 없다는 것입니다. 반복자는 오직 한 칸씩 밖에 움직일 수 없습니다.

이와 같은 이유는 `list`의 구조를 생각해보면 알 수 있습니다. 앞서 말했듯이 리스트는 왼쪽 혹은 오른쪽을 가리키고 있는 원소들의 모임으로 이루어져 있기 때문에, 한 번에 한 칸씩 밖에 이동할 수 없습니다. 즉, 메모리 상에서 원소들이 연속적으로 존재하지 않을 수 있다는 뜻입니다. 반면에 벡터의 경우 메모리 상에서 연속적으로 존재하기 때문에 쉽게 임의의 위치에 있는 원소를 참조할 수 있습니다.

이렇게 리스트에서 정의되는 반복자의 타입을 보면 `BidirectionalIterator` 타입임을 알 수 있습니다. 이름에서도 알 수 있듯이 양방향으로 이동할 수 되어, 한 칸씩 밖에 이동할 수 없습니다. 반면에 벡터에서 정의되는 반복자의 타입은 `RandomAccessIterator` 타입입니다.

즉, 임의의 위치에 접근할 수 있는 반복자입니다 (참고로 `RandomAccessIterator`는 `BidirectionalIterator`를 상속받고 있습니다)

```
#include <iostream>
#include <list>

template <typename T>
void print_list(std::list<T>& lst) {
    std::cout << "[ ";
    // 전체 리스트를 출력하기 (이 역시 범위 기반 for 문을 쓸 수 있습니다)
    for (const auto& elem : lst) {
        std::cout << elem << " ";
    }
    std::cout << "]" << std::endl;
}

int main() {
    std::list<int> lst;

    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);

    std::cout << "처음 리스트의 상태" << std::endl;
    print_list(lst);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        // 만일 현재 원소가 20 이라면
        // 그 앞에 50 을 집어넣는다.
        if (*itr == 20) {
            lst.insert(itr, 50);
        }
    }

    std::cout << "값이 20 인 원소 앞에 50 을 추가" << std::endl;
    print_list(lst);
}
```

```

for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    // 값이 30 인 원소를 삭제한다.
    if (*itr == 30) {
        lst.erase(itr);
        break;
    }
}

std::cout << "값이 30 인 원소를 제거한다" << std::endl;
print_list(lst);
}

```

성공적으로 컴파일 하면

실행 결과

처음 리스트의 상태
[10 20 30 40]
값이 20 인 원소 앞에 50 을 추가
[10 50 20 30 40]
값이 30 인 원소를 제거한다
[10 50 20 40]

와 같이 잘 나옵니다.

```

for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    // 만일 현재 원소가 20 이라면
    // 그 앞에 50 을 집어넣는다.
    if (*itr == 20) {
        lst.insert(itr, 50);
    }
}

```

앞서 설명하였지만 리스트의 반복자는 BidirectionalIterator 이기 때문에 ++ 과 -- 연산만 사용 가능합니다. 따라서 위처럼 for 문으로 하나 하나 원소를 확인해보는 것은 가능하지요. vector 와는 다르게 insert 작업은 O(1) 으로 매우 빠르게 실행됩니다.

```

for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    // 값이 30 인 원소를 삭제한다.
    if (*itr == 30) {
        lst.erase(itr);
        break;
    }
}

```

```

    }
}

```

마찬가지로 `erase` 함수를 이용하여 원하는 위치에 있는 원소를 지울 수 도 있습니다. 리스트의 경우는 벡터와는 다르게, 원소를 지워도 반복자가 무효화 되지 않습니다. 왜냐하면, 각 원소들의 주소값들은 바뀌지 않기 때문이죠!

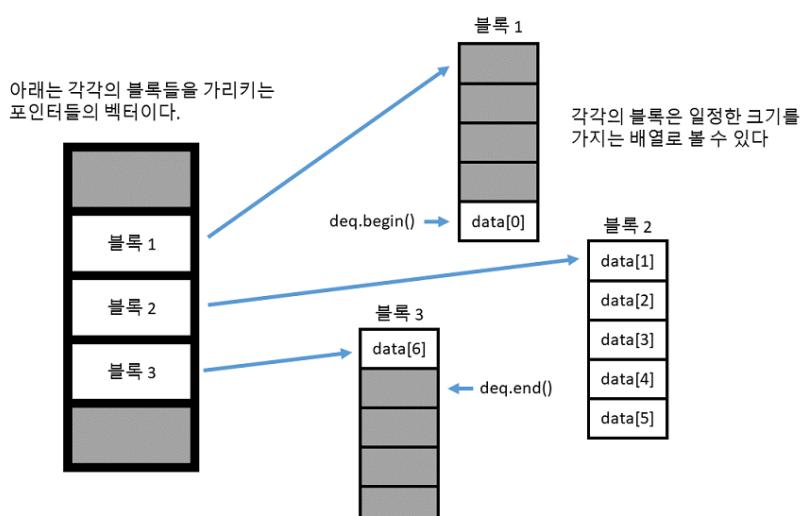
덱 (deque - double ended queue)

마지막으로 살펴볼 컨테이너는 덱(deque) 이라고 불리는 자료형 입니다. 덱은 벡터와 비슷하게 $O(1)$ 으로 임의의 위치의 원소에 접근할 수 있으며 맨 뒤에 원소를 추가/제거 하는 작업도 $O(1)$ 으로 수행할 수 있습니다. 뿐만아니라 벡터와는 다르게 맨 앞에 원소를 추가/제거 하는 작업 까지도 $O(1)$ 으로 수행 가능합니다.

임의의 위치에 있는 원소를 제거/추가 하는 작업은 벡터와 마찬가지로 $O(n)$ 으로 수행 가능합니다. 뿐만 아니라 그 속도도 벡터 보다 더 빠릅니다 (이 부분은 아래 덱이 어떻게 구현되어 있는지 설명하면서 살펴보겠습니다.)

그렇다면 덱이 벡터에 비해 모든 면에서 비교 우위에 있는 걸까요? 안타깝게도 벡터와는 다르게 덱의 경우 원소들이 실제로 메모리 상에서 연속적으로 존재하지는 않습니다. 이 때문에 원소들이 어디에 저장되어 있는지에 대한 정보를 보관하기 위해 추가적인 메모리가 더 필요로 합니다. (실제 예로, 64 비트 `libc++` 라이브러리의 경우 1 개의 원소를 보관하는 덱은 그 원소 크기에 비해 8 배나 더 많은 메모리를 필요로 합니다).

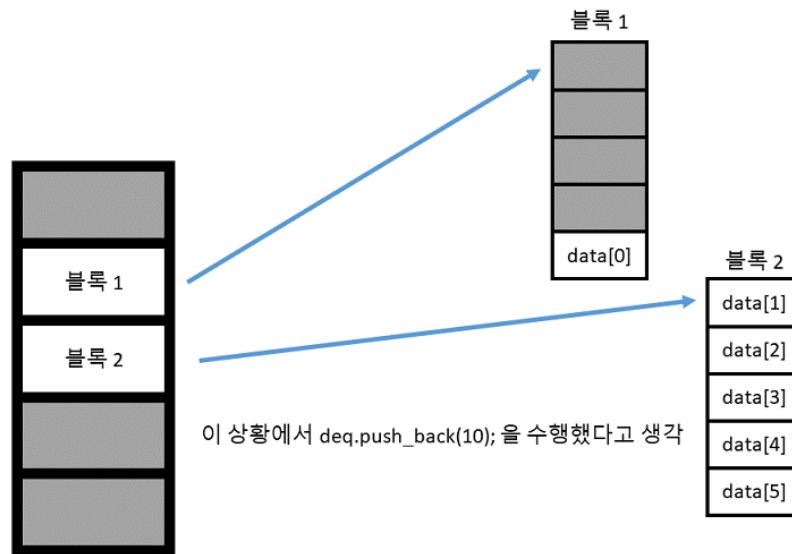
즉 덱은 실행 속도를 위해 메모리를 (많이) 희생하는 컨테이너라 보면 됩니다!.



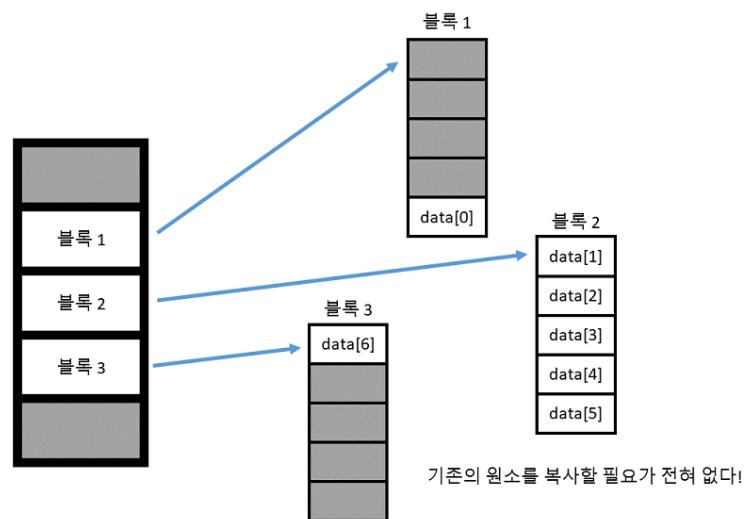
위 그림은 덱이 어떠한 구조를 가지는지 보여줍니다. 일단, 벡터와는 다르게 원소들이 메모리에 연속되어 존재하는 것이 아니라 일정 크기로 잘려서 각각의 블록 속에 존재합니다. 따라서 이 블록들이

메모리 상에 어느 곳에 위치하여 있는지 저장하기 위해서 각각의 블록들의 주소를 저장하는 벡터가 필요로 합니다.

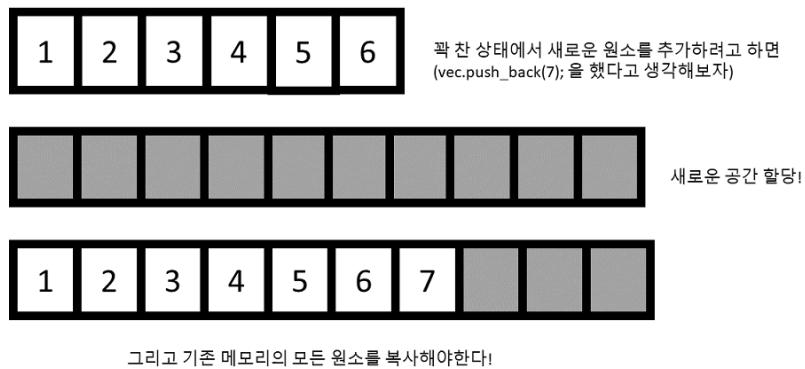
참고로 이 벡터는 기존의 벡터와는 조금 다르게, 새로 할당 시에 앞쪽 및 뒤쪽 모두에 공간을 남겨놓게 됩니다. (벡터의 경우 뒤쪽에만 공간이 남았지요) 따라서 이를 통해 맨 앞과 맨 뒤에 $O(1)$ 의 속도로 `insert` 및 `erase` 를 수행할 수 있는 것입니다. 그렇다면 왜 텍이 벡터 보다 원소를 삽입하는 작업이 더 빠른 것일까요?



위와 같은 상황에서 `deq.push_back(10)` 을 수행하였다고 생각해봅시다.



그렇다면 단순히 새로운 블록을 만들어서 뒤에 추가되는 원소를 넣어주면 됩니다. 즉 기존의 원소들을 복사할 필요가 전혀 없다는 의미입니다. 반면에 벡터의 경우



위 그림에서도 잘 알 수 있듯이, 만약에 기존에 할당한 메모리가 꽉 차면 모든 원소들을 새로운 공간에 복사해야 합니다. 따라서 평균적으로 텍이 벡터보다 더 빠르게 작동합니다. (물론 텍의 경우 블록 주소를 보관하는 벡터가 꽉 차게 되면 새로운 공간에 모두 복사해야 합니다.)

하지만 블록 주소의 개수는 전체 원소 개수 보다 적고 (위 경우 $N / 5$ 가 되겠네요. 왜냐하면 각 블록에 원소가 5개씩 있으므로), 대체로 벡터에 저장되는 객체들의 크기가 주소값의 크기보다 크기 때문에 복사 속도가 훨씬 빠릅니다.)

```
#include <deque>
#include <iostream>

template <typename T>
void print_deque(std::deque<T>& dq) {
    // 전체 덱을 출력하기
    std::cout << "[ ";
    for (const auto& elem : dq) {
        std::cout << elem << " ";
    }
    std::cout << " ] " << std::endl;
}

int main() {
    std::deque<int> dq;
    dq.push_back(10);
    dq.push_back(20);
    dq.push_front(30);
    dq.push_front(40);

    std::cout << "초기 dq 상태" << std::endl;
    print_deque(dq);

    std::cout << "맨 앞의 원소 제거" << std::endl;
    dq.pop_front();
    print_deque(dq);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
초기 dq 상태
[ 40 30 10 20 ]
맨 앞의 원소 제거
[ 30 10 20 ]
```

와 같이 잘 수행됩니다.

```
dq.push_back(10);
dq.push_back(20);
dq.push_front(30);
dq.push_front(40);
```

위와 같이 `push_back` 과 `push_front` 를 이용해서 맨 앞과 뒤에 원소들을 추가하였고,

```
dq.pop_front();
```

`pop_front` 함수를 이용해서 맨 앞의 원소를 제거할 수 있습니다.

앞서 말했듯이 텍 역시 벡터 처럼 임의의 위치에 원소에 접근할 수 있으므로 `[]` 와 `at` 함수를 제공하고 있고, 반복자 역시 `RandomAccessIterator` 타입이고 벡터랑 정확히 동일한 방식으로 작동합니다.

그래서 어떤 컨테이너를 사용해야돼?

어떠한 컨테이너를 사용할지는 전적으로 이 컨테이너를 가지고 어떠한 작업들을 많이 하느냐에 달려 있습니다.

- 일반적인 상황에서는 그냥 벡터를 사용한다 (거의 만능이다!)
- 만약에 맨 끝이 아닌 중간에 원소들을 추가하거나 제거하는 일을 많이 하고, 원소들을 순차적으로만 접근 한다면 리스트를 사용한다.
- 만약에 맨 처음과 끝 모두에 원소들을 추가하는 작업을 많이하면 텍을 사용한다.

참고적으로 $O(1)$ 으로 작동한다는 것은 언제나 이론적인 결과일 뿐이며 실제로 프로그램을 짜게 된다면, $O(\log n)$ 이나 $O(n)$ 보다도 느릴 수 있습니다. (n 의 크기에 따라서) 따라서 속도가 중요한 환경이라면 적절한 벤치마크를 통해서 성능을 가늠해 보는것도 좋습니다.

자 이번 강좌는 이것으로 마치도록 하겠습니다. 다음 강좌에서는 다른 종류의 컨테이너인 연관 컨테이너에 대해서 배웁니다.

생각 해보기

문제 1

`deque` 를 구현해보세요. (난이도 : 중)

문제 2

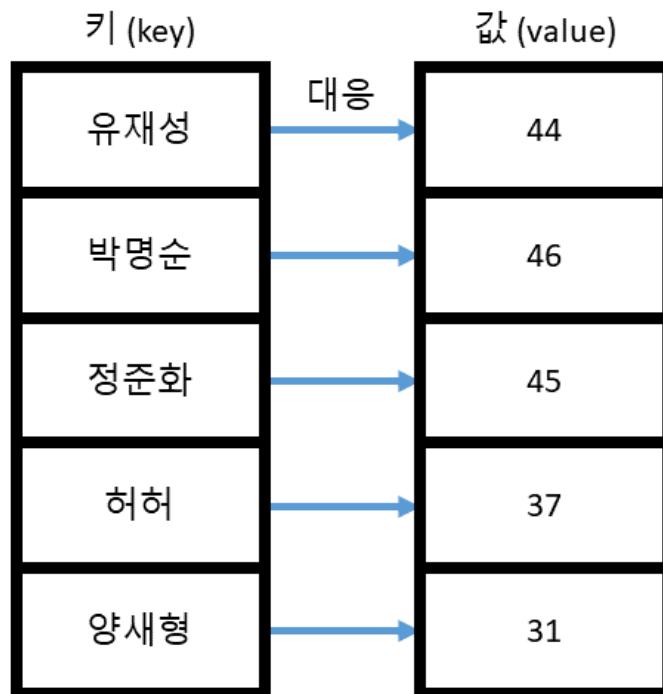
여기에서 시퀀스 컨테이너들의 모든 함수들을 찾아볼 수 있습니다. 한 번 읽어보세요!

C++ 의 표준 연관 컨테이너들

안녕하세요 여러분! 지난 강좌에서 시퀀스 컨테이너들 (`vector`, `list`, `deque`) 에 대해서 다루어 보았습니다. 시퀀스 컨테이너들은 말 그대로 '원소' 자체를 보관하는 컨테이너들입니다.

이번 강좌에서는 다른 종류의 컨테이너인 연관 컨테이너(**associative container**)에 대해서 다루어 볼 것입니다. 연관 컨테이너는 시퀀스 컨테이너와는 다르게 키(key) - 값(value) 구조를 가집니다. 다시 말해 특정한 키를 넣으면 이에 대응되는 값을 돌려준다는 것이지요. 물론 템플릿 라이브러리 이기 때문이 키와 값 모두 임의의 타입의 객체가 될 수 있습니다.

예를 들어서 어떤 웹사이트에서 회원 관리를 있다고 생각해봅시다. 사용자의 로그인을 처리하기 위해서는 아이디를 키로 가지고, 비밀번호를 값으로 가지는 데이터 구조가 필요할 것입니다. 왜냐하면 사용자가 로그인을 할 때 올바르게 입력하였는지 확인하기 위해선, 입력한 아이디에 대응되어 있던 비밀번호를 가지고, 실제 사용자가 입력한 비밀번호와 비교를 해야 되기 때문이지요.



이름(key)을 바탕으로 나이(value)를 얻을 수 있다.

위처럼 연관 컨테이너는 키를 바탕으로 이에 대응되는 값을 얻을 수 있는 구조입니다.

우리는 위와 같이 주어진 자료에서 보통 두 가지 종류의 질문을 할 수 있습니다.

- 박명순이 데이터에 존재하나요? (특정 키가 연관 컨테이너에 존재하는지 유무) → True

- 만약 존재한다면 이에 대응되는 값이 무엇인가요? (특정 키에 대응되는 값이 무엇인지 질의)
→ 46

C++ 에서는 위 두 가지 작업을 처리할 수 있 C++ 에서는 위 두 가지 작업을 처리할 수 있는 연관 컨테이너라는 것을 제공합니다. 전자의 경우 셋(set) 과 멀티셋(multiset)이고, 후자의 경우 맵(map) 과 멀티맵(multimap)입니다. 물론 맵과 멀티맵을 셋처럼 사용할 수 있습니다. 왜냐하면 해당하는 키가 맵에 존재하지 않으면 당연히 대응되는 값을 가져올 수 없기 때문이지요.

하지만 맵의 경우 셋 보다 사용하는 메모리가 크기 때문에 키의 존재 유무 만 궁금하다면 셋을 사용하는 것이 좋습니다. 그렇다면 셋 부터 어떻게 사용하는지 살펴보겠습니다.

셋(set)

```
#include <iostream>
#include <set>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (typename std::set<T>::iterator itr = s.begin(); itr != s.end(); ++itr) {
        std::cout << *itr << " ";
    }
    std::cout << " ] " << std::endl;
}

int main() {
    std::set<int> s;
    s.insert(10);
    s.insert(50);
    s.insert(20);
    s.insert(40);
    s.insert(30);

    std::cout << "순서대로 정렬되어 나온다" << std::endl;
    print_set(s);

    std::cout << "20 이 s 의 원소인가요? :: ";
    auto itr = s.find(20);
    if (itr != s.end()) {
        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }

    std::cout << "25 가 s 의 원소인가요? :: ";
    itr = s.find(25);
    if (itr != s.end()) {
```

```

        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }
}

```

성공적으로 컴파일 하였다면

실행 결과

```

순서대로 정렬되서 나온다
[ 10 20 30 40 50  ]
20 이 s 의 원소인가요? :: Yes
25 가 s 의 원소인가요? :: No

```

와 같이 나옵니다.

```

s.insert(10);
s.insert(50);
s.insert(20);
s.insert(40);
s.insert(30);

```

셋에 원소를 추가하기 위해서는 시퀀스 컨테이너처럼 `insert` 함수를 사용하면 됩니다. 한 가지 다른점은, 시퀀스 컨테이너처럼 '어디에' 추가할지에 대한 정보가 없다는 점입니다. 시퀀스 컨테이너가 상자 하나에 원소를 한 개씩 담고, 각 상자에 번호를 매긴 것이라면, 셋은 그냥 큰 상자 안에 모든 원소들을 쑤셔 넣은 것이라 보면 됩니다. 그 상자 안에 원소가 어디에 있는지는 중요한게 아니고, 그 상자 안에 원소가 '있냐/없냐' 만이 중요한 정보입니다.

셋에 원소를 추가하거나 지우는 작업은 $O(\log N)$ 에 처리됩니다. 시퀀스 컨테이너의 경우 임의의 원소를 지우는 작업이 $O(N)$ 으로 수행되었다는 점을 생각하면 훨씬 빠르다고 볼 수 있습니다.

```

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (typename std::set<int>::iterator itr = s.begin(); itr != s.end();
          ++itr) {
        std::cout << *itr << " ";
    }
    std::cout << " ] " << std::endl;
}

```

셋 역시 셋에 저장되어 있는 원소들에 접근하기 위해 반복자를 제공하며, 이 반복자는 `BidirectionalIterator`입니다. 즉, 시퀀스 컨테이너의 리스트처럼 임의의 위치에 있는 원소에 접근하는 것은 불가능하고 순차적으로 하나씩 접근하는 것 밖에 불가능 합니다.

한 가지 흥미로운 점은 우리 셋에 원소를 넣었을 때 $10 \rightarrow 50 \rightarrow 20 \rightarrow 40 \rightarrow 30$ 으로 넣었지만 실제로 반복자로 원소들을 모두 출력했을 때 나온 순서는 $10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50$ 순으로 나왔다는 점입니다. 다시 말해 셋의 경우 내부에 원소를 추가할 때 정렬된 상태를 유지하며 추가합니다.

앞서 셋을 큰 상자라 생각하고 그 안에 원소들을 쑤셔 넣은 것이라 했는데, 실제로 마구 쑤셔 넣지는 않고 순서를 지키면서 쑤셔 넣습니다. 이 때문에 시퀀스 컨테이너와는 다르게 원소를 추가하는 작업이 $O(\log N)$ 으로 진행됩니다.

또한 셋의 진가는 앞서 말했듯이 원소가 있느냐 없냐를 확인할 때 드러납니다.

```
std::cout << "20 이 s 의 원소인가요? :: ";
auto itr = s.find(20);
if (itr != s.end()) {
    std::cout << "Yes" << std::endl;
} else {
    std::cout << "No" << std::endl;
}
```

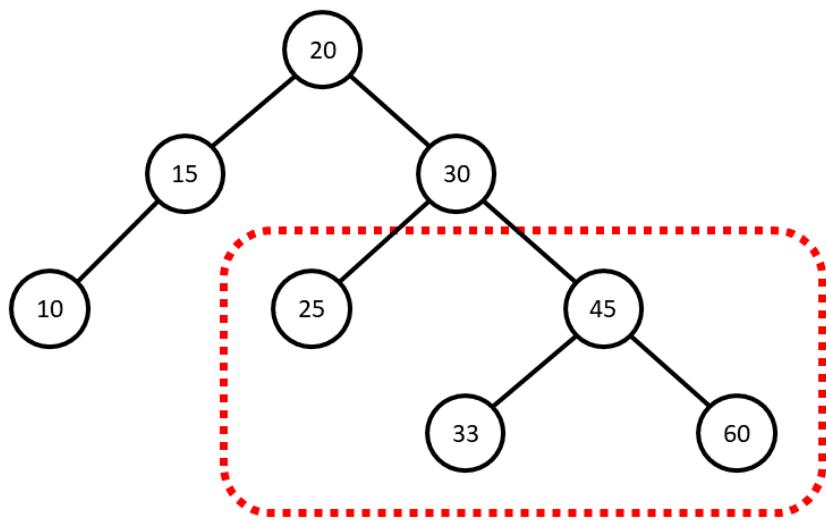
셋에는 `find` 함수가 제공되며, 이 `find` 함수를 통해 이 셋에 원소가 존재하는지 아닌지 확인할 수 있습니다. 만일 해당하는 원소가 존재한다면 이를 가리키는 반복자를 리턴하고 (`std::set<T>::iterator` 타입입니다) 만일 존재하지 않는다면 `s.end()` 를 리턴하게 되지요.

만일 벡터였다면 원소가 존재하는지 아닌지 확인하기 위해 벡터의 처음부터 끝 까지 하나씩 비교해가면서 찾았어야 했겠죠. 만일 원소가 없었더라면 벡터에 있는 모든 원소를 확인하였을 것입니다 (즉 벡터에서 `find` 는 $O(N)$ 이라 볼 수 있습니다).

하지만 셋의 경우 놀랍게도 $O(\log N)$ 으로 원소가 존재하는지 확인할 수 있습니다. 이것이 가능한 이유는 셋 내부적으로 원소들이 정렬된 상태를 유지하기 때문에 비교적 빠르게 원소의 존재 여무를 확인할 수 있습니다.

따서 20 을 찾았을 때 `Yes` 가 나오고 셋에 없는 원소인 25 를 찾는다면 `No` 가 출력됩니다.

셋이 이러한 방식으로 작업을 수행할 수 있는 이유는 바로 내부적으로 트리 구조로 구성되어 있기 때문입니다.



위 그림은 흔히 볼 수 있는 트리 구조를 나타냅니다. 각각의 원소들은 트리의 각 노드들에 저장되어 있고, 다음과 같은 규칙을 지키고 있습니다.

- 왼쪽에 오는 모든 노드들은 나보다 작다
- 오른쪽에 있는 모든 노드들은 나보다 크다

예를 들어 오른쪽의 30 을 살펴볼까요 (위 그림에서 점선으로 표시한 부분). 30 왼쪽에 오는 노드는 25 로 30 보다 작고, 오른쪽에 오는 노드들은 33, 45, 60 으로 모두 30 보다 큽니다. 어떤 노드들을 살펴보아도 이러한 규칙을 지키고 있음을 알 수 있습니다.

그렇다면 위 구조에서 25 를 찾으려면 어떻게 할까요?

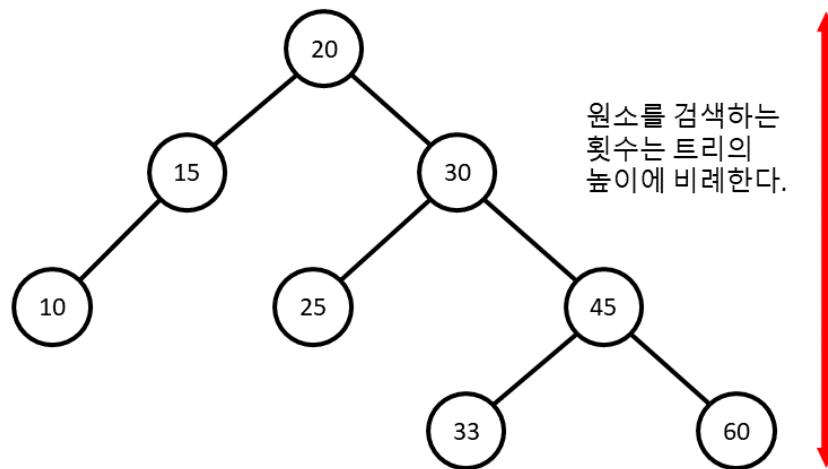
1. 일단 최상위 노드 (루트 노드라 합니다) 와 비교 : $25 > 20 \rightarrow$ 오른쪽 노드로 간다
2. 30 과 비교 : $25 < 30 \rightarrow$ 왼쪽 노드로 간 30 과 비교 : $25 < 30 \rightarrow$ 왼쪽 노드로 간다
3. 25 와 비교 : $25 == 25 \rightarrow$ 당첨 25 와 비교 : $25 == 25 \rightarrow$ 당첨!

전체 원소 개수는 8개 이지만, 단 3번의 비교로 원소를 정확히 찾을 수 있습니다.

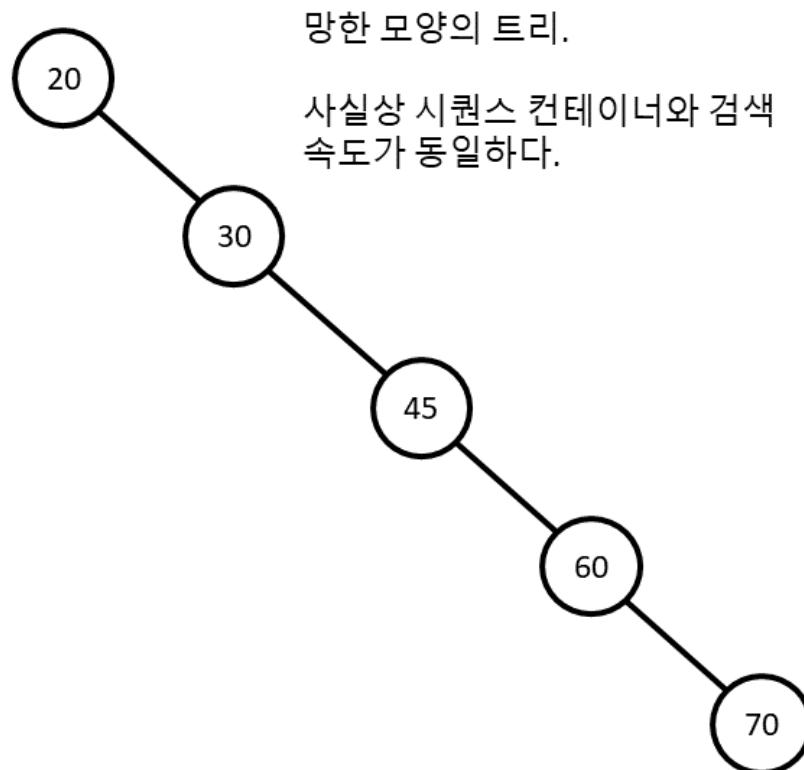
그렇다면 12 를 찾으려면 어떻게 할까요? 참고로 12 는 위 셋에 들어있지 않은 원소입니다.

1. 루트 노드와 비교 : $12 < 20 \rightarrow$ 왼쪽 노드로 간다
2. 15 와 비교 : $12 < 15 \rightarrow$ 왼쪽 노드로 간 15 와 비교 : $12 < 15 \rightarrow$ 왼쪽 노드로 간다
3. 10 과 비교 : $12 > 10 \rightarrow$ 오른쪽 노드로 가야하지만 오른쪽에 아무것도 없다. 따라서 이 원소는 존재하지 않는다 10 과 비교 : $12 > 10 \rightarrow$ 오른쪽 노드로 가야하지만 오른쪽에 아무것도 없다. 따라서 이 원소는 존재하지 않는다.

만일 벡터 였다면 원소들을 처음부터 끝까지 확인해봐야 했지만 셋의 경우 단 3번의 비교만으로 12가 셋에 존재하는지 아닌지 여부를 판단할 수 있었습니다.



아마 깨달으신 분들도 있겠지만, 원소를 검색하는데 필요한 횟수는 트리의 높이와 정확히 일치합니다. 즉, 15는 단 2번의 비교로 찾아낼 수 있고, 맨 밑에 있는 60이나 33의 경우 총 4번의 비교가 필요하겠지요. 따라서, 트리의 경우 최대한 모든 노드들을 꽉 채우는 것이 중요합니다. 예를 들어서



어쩌다 보니 트리가 위처럼 되버렸다면 사실상 시퀀스 컨테이너와 검색 속도가 동일할 것입니다. 위와 같이 한쪽으로 아예 치우쳐버린 트리를 균형잡히지 않은 트리 (unbalanced tree)라고 부릅니다.

실제 셋의 구현을 보면 위와 같은 상황이 발생하지 않도록 앞서 말한 두 개의 단순한 규칙 보다 더 많은 규칙들을 도입해서 트리를 항상 균형 잡히도록 유지하고 있습니다.

따라서 셋의 구현 상 $O(\log N)$ 으로 원소를 검색할 수 있다는 것이 보장됩니다. (궁금하신 분들만! **대부분의 셋 구현에서 사용하고 있는 트리 구조는 여기서 볼 수 있습니다**)

또한 셋의 중요한 특징으로 바로 셋 안에는 중복된 원소들이 없다는 점이 있습니다.

```
#include <iostream>
#include <set>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (const auto& elem : s) {
        std::cout << elem << " ";
    }
    std::cout << " ] " << std::endl;
}

int main() {
    std::set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(30);
    s.insert(20);
    s.insert(10);

    print_set(s);
}
```

성공적으로 컴파일 하였다면

실행 결과

[10 20 30]

와 같이 나옵니다. 분명히

```
s.insert(10);
s.insert(20);
s.insert(30);
s.insert(20);
s.insert(10);
```

위와 같이 10 과 20 을 두 번씩 넣었지만 실제로는 한 번씩 밖에 나오지 않습니다. 이는 셋 자체적으로 이미 같은 원소가 있다면 이를 `insert` 하지 않기 때문입니다. 따라서 마지막 두 `insert` 작업은 무시되었을 것입니다.

참고로 시퀀스 컨테이너들과 마찬가지로 `set` 역시 범위 기반 for 문을 지원합니다. 원소들의 접근 순서는 반복자를 이용해서 접근하였을 때와 동일합니다.

만약에 중복된 원소를 허락하고 싶다면 멀티셋(multiset) 을 사용하면 되는데, 이는 후술 하겠습니다.

여러분이 만든 클래스 객체를 셋에 넣고 싶을 때

위와 같이 기본 타입들 말고, 여러분이 만든 클래스의 객체를 셋의 원소로 사용할 때 한 가지 주의해야 할 점이 있습니다. 아래는 할 일 (Todo) 목록을 저장하기 위해 셋을 사용하는 예시입니다. Todo 클래스는 2 개를 멤버 변수로 가지는데 하나는 할 일의 중요도이고, 하나는 해야할 일의 설명입니다.

```
#include <iostream>
#include <set>
#include <string>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
    std::cout << " ] " << std::endl;
}

class Todo {
    int priority; // 중요도. 높을 수록 급한것!
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}
};

int main() {
    std::set<Todo> todos;

    todos.insert(Todo(1, "농구 하기"));
    todos.insert(Todo(2, "수학 숙제 하기"));
    todos.insert(Todo(1, "프로그래밍 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영화 보기"));
}
```

그런데 컴파일 하였다면 아래와 같은 오류가 발생합니다.

컴파일 오류

```
binary '<': no operator found which takes a left-hand operand of
→ type 'const Todo' (or there is no acceptable conversion)
```

왜 발생하였을까요? 생각을 해봅시다. 앞서 셋은 원소들을 저장할 때 내부적으로 정렬된 상태를 유지한다고 하였습니다. 즉 정렬을 하기 위해서는 반드시 원소 간의 비교를 수행해야 겠지요. 하지만, 우리의 Todo 클래스에는 `operator<` 가 정의되어 있지 않습니다. 따라서 컴파일러는 < 연산자를 찾을 수 없기에 위와 같은 오류를 뿐어내는 것입니다.

그렇다면 직접 Todo 클래스에 `operator<` 를 만들어주는 수 밖에 없습니다.

```
#include <iostream>
#include <set>
#include <string>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
}

class Todo {
    int priority;
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}

    bool operator<(const Todo& t) const {
        if (priority == t.priority) {
            return job_desc < t.job_desc;
        }
        return priority > t.priority;
    }

    friend std::ostream& operator<<(std::ostream& o, const Todo& td);
};

std::ostream& operator<<(std::ostream& o, const Todo& td) {
    o << "[ 중요도: " << td.priority << "] " << td.job_desc;
    return o;
}

int main() {
```

```

std::set<Todo> todos;

todos.insert(Todo(1, "농구 하기"));
todos.insert(Todo(2, "수학 숙제 하기"));
todos.insert(Todo(1, "프로그래밍 프로젝트"));
todos.insert(Todo(3, "친구 만나기"));
todos.insert(Todo(2, "영화 보기"));

print_set(todos);

std::cout << "-----" << std::endl;
std::cout << "숙제를 끝냈다면!" << std::endl;
todos.erase(todos.find(Todo(2, "수학 숙제 하기")));
print_set(todos);
}

```

컴파일 하였다면

실행 결과

```
[ 중요도: 3] 친구 만나기
[ 중요도: 2] 수학 숙제 하기
[ 중요도: 2] 영화 보기
[ 중요도: 1] 농구 하기
[ 중요도: 1] 프로그래밍 프로젝트
```

숙제를 끝냈다면!

```
[ 중요도: 3] 친구 만나기
[ 중요도: 2] 영화 보기
[ 중요도: 1] 농구 하기
[ 중요도: 1] 프로그래밍 프로젝트
```

와 같이 잘 실행됩니다.

먼저 < 연산자를 어떻게 구현하였는지 살펴보겠습니다.

```

bool operator<(const Todo& t) const {
    if (priority == t.priority) {
        return job_desc < t.job_desc;
    }
    return priority > t.priority;
}

```

셋이서 < 를 사용하기 위해서는 반드시 위와 같은 형태로 함수를 작성해야 합니다. 즉 `const Todo` 를 레퍼런스로 받는 `const` 함수로 말이지요. 이를 지켜야 하는 이유는 셋 내부적으로 정렬 시에 상수 반복자를 사용하기 때문입니다. (상수 반복자는 상수 함수만을 호출할 수 있습니다.) 이를 지켜야 하는 이유는 셋 내부적으로 정렬 시에 상수 반복자를 사용하기 때문입니다. (상수 반복자는 상수 함수만을 호출할 수 있습니다.)

우리의 `Todo` < 연산자는 중요도가 다르면,

```
return priority > t.priority;
```

로 해서 중요도 값이 높은 일이 위로 가게 하였습니다. 만약 중요도가 같다면

```
return job_desc < t.job_desc;
```

로 비교해서 `job_desc` 가 사전상에서 먼저 오는것이 먼저 나오게 됩니다.

한 가지 유의해야 할 점은 셋 내부에서 두 개의 원소가 같냐 다르냐를 판별하기 위해서 == 를 이용하지 않는다는 점입니다. 두 원소 A 와 B 가 셋 내부에서 같다는 의미는 `A.operator<(B)` 와 `B.operator<(A)` 가 모두 `false` 라는 뜻입니다. (예를 들어서 a 와 b 가 값이 같다고 하면 a < b 가 `false` 이고 b < a 도 `false` 이므로 a == b 이라 생각함)

만약에 우리가 중요도가 같을 때 따로 처리하지 않고 그냥

```
bool operator<(const Todo& t) const { return priority > t.priority; }
```

게 했다면 어떻게 되었을까요? 그 결과는 아래와 같습니다.

```
C:\Windows\system32\cmd.exe
[ 중요도: 3] 친구 만나기
[ 중요도: 2] 주학 축제 하기
[ 중요도: 1] 농구 하기
-----
[ 중요도: 3] 친구 만나기
[ 중요도: 1] 농구 하기
Press any key to continue . . . ■
```

위와 같이 중요도가 같은 애들은 추가 되지 않습니다. 왜냐하면 앞서 말했듯이 셋에는 중복된 원소를 허락하지 않습니다. 그런데, 셋의 입장에서

```
Todo(1, "농구 하기")
```

와

```
Todo(1, "프로그래밍 프로젝트")
```

를 보았을 때

`Todo(1, "농구 하기") < Todo(1, "프로그래밍 프로젝트")` `Todo(1, "프로그래밍 프로젝트") > Todo(1, "농구 하기")`

가 둘다 `false` 이므로, 두 개의 원소는 같은 것이라 생각하기 때문입니다! 따라서 나중에 추가된 '프로그래밍 프로젝트' 는 셋에 추가되지 않습니다. 같은 이유로 영화 보기도 추가되지 않습니다.

따라서 `operator<` 를 설계할 때 반드시 다른 객체는 `operator<` 상에서도 구분될 수 있도록 만들어야 합니다. 다시 말해 A 랑 B 가 다른 객체라면, A < B 혹은 B < A 중 하나는 반드시 `True`여야 합니다.

엄밀히 말하자면 `operator<` 는 다음과 같은 조건들을 만족해야 합니다. (A 랑 B 가 다른 객체라면)

- A < A 는 거짓
- A < B != B < A
- A < B이고 B < C 이면 A < C
- A == B 이면 A < B 와 B < A 둘 다 거짓
- A == B 이고 B == C 이면 A == C

위와 같은 조건을 만족하는 < 연산자는 *strict weak ordering* 을 만족한다고 합니다. 지켜야 할 조건들이 꽤나 많이 보이는데 사실, 상식적으로 `operator<` 를 설계하였다면 위 조건들은 모두 만족할 수 있습니다.

만약에, 위 중 하나라도 조건이 맞지 않는다면 `set` 이 제대로 동작하지 않고 (컴파일 타임에는 오류가 발생하지 않습니다), 런타임 상에서 오류가 발생할 텐데 정말 디버깅 하기 힘들 것입니다 :(

마지막으로 보여드릴 것은, 클래스 자체에 `operator<` 를 두지 않더라도 셋을 사용하는 방법입니다. 예를 들어서 우리가 외부 라이브러리를 사용하는데, 만약에 그 라이브러리의 한 클래스의 객체를 셋에 저장하고 싶다고 해봅시다. 우리가 사용하는 외부 클래스에 `operator<` 가 정의되어 있지 않다는 점입니다. 이럴 경우, 셋을 사용하기 위해서는 따로 객체를 비교할 수 있는 방법을 알려주어야 합니다.

아래 예제를 보실까요.

```
#include <iostream>
#include <set>
#include <string>

template <typename T, typename C>
```

```

void print_set(std::set<T, C>& s) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
}

class Todo {
    int priority;
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}

    friend struct TodoCmp;

    friend std::ostream& operator<<(std::ostream& o, const Todo& td);
};

struct TodoCmp {
    bool operator()(const Todo& t1, const Todo& t2) const {
        if (t1.priority == t2.priority) {
            return t1.job_desc < t2.job_desc;
        }
        return t1.priority > t2.priority;
    }
};

std::ostream& operator<<(std::ostream& o, const Todo& td) {
    o << "[ 중요도: " << td.priority << "] " << td.job_desc;
    return o;
}

int main() {
    std::set<Todo, TodoCmp> todos;

    todos.insert(Todo(1, "농구 하기"));
    todos.insert(Todo(2, "수학 숙제 하기"));
    todos.insert(Todo(1, "프로그래밍 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영화 보기"));

    print_set(todos);

    std::cout << "-----" << std::endl;
    std::cout << "숙제를 끝냈다면!" << std::endl;
    todos.erase(todos.find(Todo(2, "수학 숙제 하기")));
    print_set(todos);
}

```

성공적으로 컴파일 하였다면

실행 결과

```
[ 중요도: 3] 친구 만나기
[ 중요도: 2] 수학 숙제 하기
[ 중요도: 2] 영화 보기
[ 중요도: 1] 농구 하기
[ 중요도: 1] 프로그래밍 프로젝트
```

숙제를 끝냈다면!

```
[ 중요도: 3] 친구 만나기
[ 중요도: 2] 영화 보기
[ 중요도: 1] 농구 하기
[ 중요도: 1] 프로그래밍 프로젝트
```

와 같이 나옵니다. 달라진 점은 일단 Todo 클래스에서 `operator<` 가 삭제되었습니다. 하지만 셋을 사용하기 위해 반드시 Todo 객체간의 비교를 수행해야 하기 때문에 다음과 같은 클래스를 만들었습니다.

```
struct TodoCmp {
    bool operator()(const Todo& t1, const Todo& t2) const {
        if (t1.priority == t2.priority) {
            return t1.job_desc < t2.job_desc;
        }
        return t1.priority > t2.priority;
    }
};
```

앞서 템플릿 첫 강좌에서 함수 객체를 배운 것이 기억 나시나요? 위 클래스는 정확히 함수 객체를 나타내고 있습니다. 이 `TodoCmp` 타입을

```
std::set<Todo, TodoCmp> todos;
```

위처럼 `set`에 두번째 인자로 넘겨주게 되면 셋은 이를 받아서 `TodoCmp` 클래스에 정의된 함수 객체를 바탕으로 모든 비교를 수행하게 됩니다. 실제로 `set` 클래스의 정의를 살펴보면;

```
template <class Key, class Compare = std::less<Key>,
          class Allocator = std::allocator<Key> // ← 후에 설명하겠습니다
        >
class set;
```

와 같이 생겼는데, 두 번째 인자로 `Compare` 를 받는다는 것을 알 수 있습니다. (템플릿 디폴트 인자로 `std::less<Key>` 가 들어있는데 이는 `Key` 클래스의 `operator<` 를 사용한다는 의미와 같습니다. `Compare` 타입을 전달하지 않으면 그냥 `Key` 클래스의 `operator<` 로 비교를 수행합니다.)

결과적으로 셋은 원소의 삽입과 삭제를 $O(\log N)$ 원소의 탐색도 $O(\log N)$ 에 수행하는 자료 구조입니다.

맵 (map)

맵은 셋과 거의 똑같은 자료 구조입니다. 다만 셋의 경우 키만 보관했지만, 맵의 경우 키에 대응되는 값(value) 까지도 같이 보관하게 됩니다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(std::map<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (auto itr = m.begin(); itr != m.end(); ++itr) {
        std::cout << itr->first << " " << itr->second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    // 참고로 2017년 7월 4일 현재 투수 방어율 순위입니다.

    // 맵의 insert 함수는 pair 객체를 인자로 받습니다.
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.23));
    pitcher_list.insert(std::pair<std::string, double>("해커", 2.93));

    pitcher_list.insert(std::pair<std::string, double>("피어밴드", 2.95));

    // 타입을 지정하지 않아도 간단히 std::make_pair 함수로
    // std::pair 객체를 만들 수 도 있습니다.
    pitcher_list.insert(std::make_pair("차우찬", 3.04));
    pitcher_list.insert(std::make_pair("장원준", 3.05));
    pitcher_list.insert(std::make_pair("헥터", 3.09));

    // 혹은 insert 를 안쓰더라도 [] 로 바로
    // 원소를 추가할 수 있습니다.
    pitcher_list["니퍼트"] = 3.56;
    pitcher_list["박종훈"] = 3.76;
    pitcher_list["켈리"] = 3.90;
```

```

print_map(pitcher_list);

std::cout << "박세웅 방어율은? :: " << pitcher_list["박세웅"] << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

니퍼트 3.56
박세웅 2.23
박종훈 3.76
장원준 3.05
차우찬 3.04
켈리 3.9
피어밴드 2.95
해커 2.93
헥터 3.09
박세웅 방어율은? :: 2.23

```

와 같이 나옵니다.

```
std::map<std::string, double> pitcher_list;
```

맵의 경우 템플릿 인자로 2 개를 가지는데, 첫번째는 키의 타입이고, 두 번째는 값의 타입입니다. 우리는 투수 이름을 키로 가지고 대응되는 값을 그 투수의 방어율로 하는 맵을 만들 예정입니다.

```

pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.23));
pitcher_list.insert(std::pair<std::string, double>("해커 ", 2.93));
pitcher_list.insert(std::pair<std::string, double>("피어밴드 ", 2.95));

```

맵에 원소를 넣기 위해서는 반드시 `std::pair` 객체를 전달해야 합니다. `std::pair` 객체는 별다른게 아니고,

```

template <class T1, class T2>
struct std::pair {
    T1 first;
    T2 second;
};

```

로 생긴 단순히 2 개의 객체를 멤버로 가지는 객체입니다. 문제는 `std::pair` 객체를 사용할 때마다 위처럼 템플릿 인자를 초기화 해야 하는데 꽤나 귀찮습니다. 그래서 STL에서는 `std::make_pair` 함수를 제공해줍니다,

```
pitcher_list.insert(std::make_pair("차우찬", 3.04));
pitcher_list.insert(std::make_pair("장원준", 3.05));
pitcher_list.insert(std::make_pair("헥터", 3.09));
```

이 함수는 인자로 들어오는 객체를 보고 타입을 추측해서 알아서 `std::pair` 객체를 만들어서 리턴해줍니다. 따라서 굳이 귀찮게 타입을 명시해줄 필요가 없습니다.

한 가지 재미있는 점은

```
// 혹은 insert 를 안쓰더라도 [] 로 바로
// 원소를 추가할 수 있습니다.
pitcher_list["니퍼트"] = 3.56;
pitcher_list["박종훈"] = 3.76;
pitcher_list["켈리"] = 3.90;
```

맵의 경우 `operator[]` 를 이용해서 새로운 원소를 추가할 수 도 있습니다 (만일 해당하는 키가 맵에 없다면). 만일 키가 이미 존재하고 있다면 값이 대체될 것입니다.

```
template <typename K, typename V>
void print_map(std::map<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (auto itr = m.begin(); itr != m.end(); ++itr) {
        std::cout << itr->first << " " << itr->second << std::endl;
    }
}
```

맵의 경우도 셋과 마찬가지로 반복자를 이용해서 순차적으로 맵에 저장되어 있는 원소들을 탐색할 수 있습니다. 참고로 셋의 경우 `*itr` 가 저장된 원소를 바로 가리켰는데, 맵의 경우 반복자가 맵에 저장되어 있는 `std::pair` 객체를 가리키게 됩니다. 따라서 `itr->first` 를 하면 해당 원소의 키를, `itr->second` 를 하면 해당 원소의 값을 알 수 있습니다.

참고로 해당 `for` 문을 범위 기반 `for` 문으로 바꿔본다면 아래와 같습니다.

```
template <typename K, typename V>
void print_map(std::map<K, V>& m) {
    // kv 에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}
```

반복자를 이용한 버전과 매우 유사하게, 맵의 키와 대응되는 원소를 `first` 와 `second` 를 이용해서 참조할 수 있습니다. 이 역시 반복자를 사용한 형태보다 더 간단하므로 권장됩니다.

```
std::cout << "박세웅 방어율은? :: " << pitcher_list["박세웅"] << std::endl;
```

만약에 맵에 저장된 값을 찾고 싶다면 간단히 [] 연산자를 이용하면 됩니다. [] 연산자는 인자로 키를 받아서 이를 맵에서 찾아서 대응되는 값을 돌려줍니다.

하지만, [] 연산자를 사용할 때 주의해야 할 점이 있습니다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv 에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    pitcher_list["오승환"] = 3.58;
    std::cout << "류현진 방어율은? :: " << pitcher_list["류현진"] << std::endl;

    std::cout << "-----" << std::endl;
    print_map(pitcher_list);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
류현진 방어율은? :: 0
```

```
-----
```

```
류현진 0
```

```
오승환 3.58
```

와 같이 나옵니다.

```
pitcher_list["오승환"] = 3.58;
```

일단 위와 같이 `pitcher_list`에 오승환의 방어율만 추가하였기 때문에 류현진의 방어율을 검색하면 아무것도 나오지 않는게 정상입니다. 그런데,

```
std::cout << "류현진 방어율은? :: " << pitcher_list["류현진"] << std::endl;
```

위 처럼 류현진의 방어율을 맵에서 검색하였을 때, 0이라는 값이 나왔습니다. 없는 값을 참조하였으니 오류가 발생해야 정상인데 오히려 값을 돌려주었네요. 이는 [] 연산자가, 맵에 없는 키를 참조하게 되면, 자동으로 값의 디폴트 생성자를 호출해서 원소를 추가해버리기 때문입니다.

`double`의 디폴트 생성자의 경우 그냥 변수를 0으로 초기화 해버립니다. 따라서 되도록이면 `find` 함수로 원소가 키가 존재하는지 먼저 확인 후에, 값을 참조하는 것이 좋습니다. 아래는 `find` 함수를 이용해서 안전하게 키에 대응되는 값을 찾는 방법입니다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

template <typename K, typename V>
void search_and_print(std::map<K, V>& m, K key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        std::cout << key << " --> " << itr->second << std::endl;
    } else {
        std::cout << key << "은(는) 목록에 없습니다" << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    pitcher_list["오승환"] = 3.58;

    print_map(pitcher_list);
    std::cout << "-----" << std::endl;

    search_and_print(pitcher_list, std::string("오승환"));
    search_and_print(pitcher_list, std::string("류현진"));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
오승환 3.58
-----
오승환 --> 3.58
류현진은(는) 목록에 없습니다
```

와 같이 나옵니다.

```
template <typename K, typename V>
void search_and_print(const std::map<K, V>& m, K key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        std::cout << key << " --> " << itr->second << std::endl;
    } else {
        std::cout << key << "은(는) 목록에 없습니다" << std::endl;
    }
}
```

위처럼 `find` 함수는 맵에서 해당하는 키를 찾아서 이를 가리키는 반복자를 리턴합니다. 만약에, 키가 존재하지 않는다면 `end()`를 리턴합니다.

마지막으로 짚고 넘어갈 점은 맵 역시 셋처럼 중복된 원소를 허락하지 않는다는 점입니다. 이미, 같은 키가 원소로 들어 있다면 나중에 오는 `insert`는 무시됩니다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    // 맵의 insert 함수는 std::pair 객체를 인자로 받습니다.
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.23));
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.93));
```

```

print_map(pitcher_list);

// 2.23 이 나올까 2.93 이 나올까?
std::cout << "박세웅 방어율은? :: " << pitcher_list["박세웅"] << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

박세웅 2.23
박세웅 방어율은? :: 2.23

```

와 같이 먼저 `insert` 된 원소가 나오게 됩니다. 즉, 이미 같은 키를 가지는 원소가 있다면 그 `insert` 작업은 무시됩니다. 만약에, 원소에 대응되는 값을 바꾸고 싶다면 `insert` 를 하지 말고, [] 연산자로 대응되는 값을 바꿔주면 됩니다.

멀티셋(multiset)과 멀티맵(multimap)

앞서 셋과 맵 모두 중복된 원소를 허락하지 않습니다. 만일, 이미 원소가 존재하고 있는데 `insert` 를 하였으면 무시가 되었지요. 하지만 멀티셋과 멀티맵은 중복된 원소를 허락합니다.

```

#include <iostream>
#include <set>
#include <string>

template <typename K>
void print_set(const std::multiset<K>& s) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : s) {
        std::cout << elem << std::endl;
    }
}

int main() {
    std::multiset<std::string> s;

    s.insert("a");
    s.insert("b");
    s.insert("a");
    s.insert("c");
    s.insert("d");
    s.insert("c");
}

```

```

    print_set(s);
}

```

성공적으로 컴파일 하였다면

실행 결과

```

a
a
b
c
c
d

```

와 같이 나옵니다. 만약에 기존의 `set` 이었다면 그냥 `a, b, c, d` 이렇게 나왔어야 하지만, 멀티셋의 경우 중복된 원소를 허락하기 때문에 `insert` 한 모든 원소들이 쭈르륵 나오게 됩니다.

```

#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::multimap<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::multimap<int, std::string> m;
    m.insert(std::make_pair(1, "hello"));
    m.insert(std::make_pair(1, "hi"));
    m.insert(std::make_pair(1, "ahihi"));
    m.insert(std::make_pair(2, "bye"));
    m.insert(std::make_pair(2, "baba"));

    print_map(m);

    // 뭐가 나올까요?
    std::cout << "-----" << std::endl;
    std::cout << m.find(1)->second << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```
1 hello
1 hi
1 ahihi
2 bye
2 baba
-----
hello
```

와 같이 나옵니다.

일단 맵 과는 다르게, 한 개의 키에 여러개의 값이 대응될 수 있다는 것은 알 수 있습니다. 하지만 이 때문에 [] 연산자를 멀티맵의 경우 사용할 수 없습니다. 왜냐하면 예를 들어서

m[1]

을 했을 때 "hello" 를 리턴해야 할지, 아니면 "hi" 를 리턴해야 할지 알 수 없기 때문이지요. 따라서 멀티맵의 경우 아예 [] 연산자를 제공하지 않습니다. 그렇다면

```
std::cout << m.find(1)->second << std::endl;
```

위 처럼 `find` 함수를 사용했을 때 무엇을 리턴할까요? 일단 해당하는 키가 없으면 `m.end()` 를 리턴합니다. 그렇다면 위 경우 1 이라는 키에 3 개의 문자열이 대응되어 있는데 어떤거를 리턴해야 할까요? 제일 먼저 `insert` 한것? 아니면 문자열 중에서 사전 순으로 가장 먼저 오는 것?

사실 C++ 표준을 읽어보면 무엇을 리턴하라고 정해놓지 않았습니다. 즉, 해당되는 값들 중 아무거나 리턴해도 상관 없다는 뜻입니다. 위 경우 `hello` 가 나왔지만, 다른 라이브러리를 쓰는 경우 `hi` 가 나올 수도 있고, `ahihhi` 가 나올 수도 있습니다.

그렇다면 1 에 대응되는 값들이 뭐가 있는지 어떻게 알까요? 이를 위해 멀티맵은 다음과 같은 함수를 제공하고 있습니다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::multimap<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
```

```

    }
}

int main() {
    std::multimap<int, std::string> m;
    m.insert(std::make_pair(1, "hello"));
    m.insert(std::make_pair(1, "hi"));
    m.insert(std::make_pair(1, "ahihi"));
    m.insert(std::make_pair(2, "bye"));
    m.insert(std::make_pair(2, "baba"));

    print_map(m);

    std::cout << "-----" << std::endl;

    // 1 을 키로 가지는 반복자들의 시작과 끝을
    // std::pair 로 만들어서 리턴한다.
    auto range = m.equal_range(1);
    for (auto itr = range.first; itr != range.second; ++itr) {
        std::cout << itr->first << " : " << itr->second << " "
    }
}

```

성공적으로 컴파일 하였다면

실행 결과

```

1 hello
1 hi
1 ahihi
2 bye
2 baba
-----
1 : hello
1 : hi
1 : ahihi

```

와 같이 나옵니다.

```
auto range = m.equal_range(1);
```

equal_range 함수의 경우 인자로 멀티맵의 키를 받은 뒤에, 이 키에 대응되는 원소들의 반복자들 중에서 시작과 끝 바로 다음을 가리키는 반복자를 std::pair 객체로 만들어서 리턴합니다. 즉, begin() 과 end() 를 std::pair 로 만들어서 세트로 리턴한다고 볼 수 있겠지요. 다만, first

로 시작점을, `second` 로 끝점 바로 뒤를 알 수 있습니다. 왜 끝점 바로 뒤를 가리키는 반복자를 리턴하는지는 굳이 설명 안해도 알겠죠?

```
for (auto itr = range.first; itr != range.second; ++itr) {
    std::cout << itr->first << " : " << itr->second << " " << std::endl;
}
```

따라서 위처럼 1에 대응되는 모든 원소들을 볼 수 있게 됩니다.

정렬되지 않은 셋과 맵 (`unordered_set`, `unordered_map`)

`unordered_set` 과 `unordered_map` (한글로 하면 너무 길어서 그냥 영문으로 표기하겠습니다)은 C++ 11에 추가된 비교적 최근 나온 컨테이너들입니다 (위에 것들은 모두 C++ 98에 추가되었었죠).

이 두 개의 컨테이너는 이름에서도 알 수 있듯이 원소들이 정렬되어 있지 않습니다.

이 말이 무슨 말이냐면, 셋이나 맵의 경우 원소들이 순서대로 정렬되어서 내부에 저장되지만, `unordered_set`과 `unordered_map`의 경우 원소들이 순서대로 정렬되서 들어가지 않는다는 뜻입니다. 따라서 반복자로 원소들을 하나씩 출력해보면 거의 랜덤한 순서로 나오는 것을 볼 수 있습니다.

```
#include <iostream>
#include <string>
#include <unordered_set>

template <typename K>
void print_unordered_set(const std::unordered_set<K>& m) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : m) {
        std::cout << elem << std::endl;
    }
}

int main() {
    std::unordered_set<std::string> s;

    s.insert("hi");
    s.insert("my");
    s.insert("name");
    s.insert("is");
    s.insert("psi");
    s.insert("welcome");
```

```
s.insert("to");
s.insert("c++");

print_unordered_set(s);
}
```

성공적으로 컴파일 하였다면

실행 결과

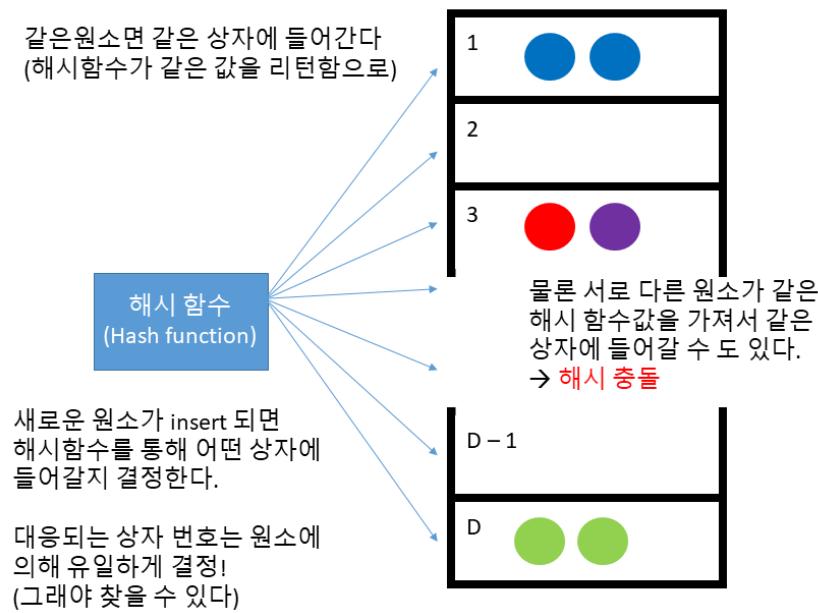
```
c++
to
my
name
hi
is
psi
welcome
```

와 같이 나옵니다.

실제로 `unordered_set` 의 모든 원소들을 반복자로 출력해보면 딱히 순서대로 나오는 것 같지는 않습니다. 원소를 넣은 순서도 아니고, `string` 문자열 순서도 아니고 그냥 랜덤한 순서입니다.

그런데 이 `unordered_set` 에 한 가지 놀라운 점이 있습니다. 바로 `insert`, `erase`, `find` 모두가 $O(1)$ 으로 수행된다는 점입니다! 셋이나 맵의 경우 $O(log n)$ 이었지만, `unordered_set` 과 `unordered_map` 의 경우 상수 시간에 원소를 삽입하고, 검색할 수 있습니다.

이 놀라운 일이 어떻게 가능한건지 `unordered_set` 과 `unordered_map` 이 어떻게 구현되었는지 살펴보면 알 수 있습니다.



해시 함수(Hash function)

`unordered_set` 과 `unordered_map` 은 원소를 삽입하거나 검색 하기 위해 먼저 해시 함수라는 것을 사용합니다 (사실 그래서 원래 `hashset` 이나 `hashmap` 이란 이름을 붙이려고 했지만 이미 이러한 이름을 너무 많이 사용하고 있어서 충돌을 피하기 위해 저런 이름을 골랐다고 합니다).

해시 함수란 임의의 크기의 데이터를 고정된 크기의 데이터로 대응시켜주는 함수라고 볼 수 있습니다. 이 때 보통 고정된 크기의 데이터라고 하면 일정 범위의 정수값을 의미합니다.

`unordered_set` 과 `unordered_map` 의 경우, 해시함수는 1 부터 D (= 상자의 수)까지의 값을 반환하고 그 해시값 (해시 함수로 계산한 값)을 원소를 저장할 상자의 번호로 삼게 됩니다. 해시 함수는 구조상 최대한 1 부터 D 까지 고른 값을 반환하도록 설계되었습니다. 따라서 모든 상자를 고루 고루 사용할 수 있게 되지요.

해시 함수의 가장 중요한 성질은, 만약에 같은 원소를 해시 함수에 전달한다면 같은 해시값을 리턴한다는 점입니다. 이 덕분에 원소의 탐색을 빠르게 수행할 수 있습니다.

예를 들어 사용자가 파란공이 `unordered_set` 에 들어있는지 아닌지 확인한다고 해봅시다. 파란 공을 해시 함수에 대입하면 1 을 리턴합니다. 따라서 1 번 상자를 살펴보면 이미 파란공이 있는 것을 알 수 있지요. 따라서 파란공이 `unordered_set` 에 이미 존재하고 있음을 알 수 있습니다.

그런데 재미있는 점은 해시 함수가 해시값 계산을 상수 시간에 처리한다는 점입니다. 따라서 `unordered_set` 과 `unordered_map` 모두 탐색을 상수 시간에 처리할 수 있습니다.

물론 뺨간색 공과 보라색 공처럼 다른 원소임에도 불구하고 같은 해시값을 갖는 경우가 있을 것입니다. 이를 해시 충돌(**hash collision**) 이라고 하는데, 이 경우 같은 상자에 다른 원소들이 들어있게 됩니다.

따라서 만약에 보라색 공이 이 셋에 포함되어 있는지 확인하고 싶다면 먼저 보라색 공의 해시값을

계산 한 뒤에, 해당하는 상자에 있는 모든 원소들을 탐색해보아야 할 것입니다.

해시 함수는 최대한 1부터 N 까지 고른 값을 반환하도록 설계되었습니다. 뿐만 아니라 상자의 수도 충분히 많아야 상수 시간 탐색을 보장할 수 있습니다. 하지만 그럼에도 운이 매우 매우 나쁘다면 다른 색들의 공이 모두 1 번 상자에 들어갈 수 도 있습니다. 이 경우 탐색이 $O(1)$ 은 커녕 $O(N)$ (여기서 n 은 상자의 개수가 아니라 원소의 개수) 으로 실행될 것입니다.

따라서 `unordered_set` 과 `unordered_map` 의 경우 평균적으로 $O(1)$ 시간으로 원소의 삽입/탐색을 수행할 수 있지만 최악의 경우 $O(N)$ 으로 수행될 수 있습니다. (그냥 `set` 과 `map` 의 경우 평균도 $O(\log N)$ 최악의 경우에도 $O(\log N)$ 으로 실행됩니다)

이 때문에 보통의 경우에는 그냥 안전하게 맵이나 셋을 사용하고, 만약에 최적화가 매우 필요한 작업일 경우에만 해시 함수를 잘 설계해서 `unordered_set` 과 `unordered_map` 을 사용하는 것이 좋습니다.¹⁾

또한 처음부터 많은 개수의 상자를 사용할 수 없기 때문에 (메모리를 낭비할 순 없으므로..) 상자의 개수는 삽입되는 원소가 많아짐에 따라 점진적으로 늘어나게 됩니다. 문제는 상자의 개수가 늘어나면 해시 함수를 바꿔야 하기 때문에 (더 많은 값들을 해시값으로 반환할 수 있도록) 모든 원소들을 처음부터 끝 까지 다시 `insert` 해야 합니다. 이를 `rehash` 라 하며 $O(N)$ 만큼의 시간이 걸립니다.

```
#include <iostream>
#include <string>
#include <unordered_set>

template <typename K>
void print_unordered_set(const std::unordered_set<K>& m) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : m) {
        std::cout << elem << std::endl;
    }
}

template <typename K>
void is_exist(std::unordered_set<K>& s, K key) {
    auto itr = s.find(key);
    if (itr != s.end()) {
        std::cout << key << " 가 존재!" << std::endl;
    } else {
        std::cout << key << " 가 없다" << std::endl;
    }
}

int main() {
    std::unordered_set<std::string> s;
    s.insert("hi");
}
```

1) 기본 타입들(int, double 등등) 과 `std::string` 의 경우 라이브러리 자체적으로 해시 함수가 내장되어 있으므로, 그냥 사용하셔도 됩니다

```

s.insert("my");
s.insert("name");
s.insert("is");
s.insert("psi");
s.insert("welcome");
s.insert("to");
s.insert("c++");

print_unordered_set(s);
std::cout << "-----" << std::endl;
is_exist(s, std::string("c++"));
is_exist(s, std::string("c"));

std::cout << "-----" << std::endl;
std::cout << "'hi' 를 삭제" << std::endl;
s.erase(s.find("hi"));
is_exist(s, std::string("hi"));
}

```

성공적으로 컴파일 하였다면

실행 결과

```

c++
to
my
name
hi
is
psi
welcome
-----
c++ 가 존재!
c 가 없다
-----
'hi' 를 삭제
hi 가 없다

```

와 같이 나옵니다.

```

template <typename K>
void is_exist(std::unordered_set<K>& s, K key) {
    auto itr = s.find(key);
    if (itr != s.end()) {

```

```

        std::cout << key << " 가 존재!" << std::endl;
    } else {
        std::cout << key << " 가 없다" << std::endl;
    }
}

```

일단 위에서 볼 수 있듯이, `unordered_set` 과 `unordered_map` 모두 `find` 함수를 지원하며, 사용법은 그냥 셋과 정확히 동일합니다. `find` 함수의 경우 만일 해당하는 원소가 존재한다면 이를 가리키는 반복자를, 없다면 `end` 를 리턴합니다.

```

s.erase(s.find("hi"));
is_exist(s, std::string("hi"));

```

또한 원소를 제거하고 싶다면 간단히 `find` 함수로 원소를 가리키는 반복자를 찾은 뒤에, 이를 전달하면 됩니다.

내가 만든 클래스를 `unordered_set`/`unordered_map`의 원소로 넣고 싶을 때

그렇다면 여러분이 만든 클래스를 직접 `unordered_set` 혹은 `unordered_map` 에 넣으려면 어떻게 해야 할까요? 안타깝게도 셋이나 맵에 넣는것 보다 훨씬 어렵습니다. 왜냐하면 먼저 내 클래스의 객체를 위한 '해시 함수'를 직접 만들어줘야 하기 때문입니다. (그렇기 때문에 셋과 맵을 사용하는 것을 권장하는 것입니다!)

물론 셋이나 맵 과는 다르게 순서대로 정렬하지 않기 때문에 `operator<` 는 필요하지 않습니다. 하지만 `operator==` 는 필요한데, 왜냐하면 해시 충돌 발생 시에 상자안에 있는 원소들과 비교를 해야하기 때문이지요.

한 가지 다행인 점은 C++ 에서 기본적인 타입들에 대한 해시 함수들을 제공하고 있습니다. 우리는 이들을 잘만 이용하기만 하면 됩니다.

```

class Todo {
    int priority; // 중요도. 높을 수록 급한것!
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}
};

```

그렇다면 위 Todo 클래스의 해시 함수를 만들어보겠습니다. 기본적으로 `unordered_set` 과 `unordered_map`은 해시 함수 계산을 위해 hash 함수 객체를 사용합니다. hash 함수 객체는 아래와 같이 생겼습니다.

예를 들어 `string` 함수의 해시값을 계산하고 싶다면

```
hash<string> hash_fn;
size_t hash_val = hash_fn(str); // str 의 해시값 계산
```

위와 같이 수행하게 되는 것인지요. `string` 을 템플릿 인자로 받는 `hash_fn` 객체를 만든 뒤에, ([Functor](#) 입니다) 마치 함수를 사용하는 것처럼 사용하면 됩니다.

따라서 Todo 함수의 해시 함수를 계산하는 함수 객체를 만들기 위해 다음과 같이 `hash` 클래스의 Todo 특수화 버전을 만들어줘야 합니다.

```
// hash 클래스의 Todo 템플릿 특수화 버전!
template <>
struct hash<Todo> {
    size_t operator()(const Todo& t) const {
        // 해시 계산
    }
};
```

해시 함수는 객체의 `operator()` 를 오버로드하면 되고 `std::size_t` 타입을 리턴하면 됩니다. 보통 `size_t` 타입은 `int` 랑 동일한데, 이 말은 해시값으로 0 부터 4294967295 까지 가능하다는 뜻입니다. 물론 그렇다고 해서 이 만큼의 상자를 사용하는 것은 아니고, 현재 컨테이너가 사용하고 있는 상자 개수로 나눈 나머지를 상자 번호로 사용하겠지요.

```
#include <functional>
#include <iostream>
#include <string>
#include <unordered_set>

template <typename K>
void print_unordered_set(const std::unordered_set<K>& m) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : m) {
        std::cout << elem << std::endl;
    }
}

template <typename K>
void is_exist(std::unordered_set<K>& s, K key) {
    auto itr = s.find(key);
    if (itr != s.end()) {
        std::cout << key << " 가 존재!" << std::endl;
```

```

} else {
    std::cout << key << " 가 없다" << std::endl;
}
}

class Todo {
    int priority; // 중요도. 높을 수록 급한것!
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}

    bool operator==(const Todo& t) const {
        if (priority == t.priority && job_desc == t.job_desc) return true;
        return false;
    }

    friend std::ostream& operator<<(std::ostream& o, const Todo& t);
    friend struct std::hash<Todo>;
};

// Todo 해시 함수를 위한 함수객체(Functor)
// 를 만들어줍니다!
namespace std {
template <>
struct hash<Todo> {
    size_t operator()(const Todo& t) const {
        hash<string> hash_func;

        return t.priority ^ (hash_func(t.job_desc));
    }
};

} // namespace std
std::ostream& operator<<(std::ostream& o, const Todo& t) {
    o << "[중요도 : " << t.priority << " ] " << t.job_desc;
    return o;
}

int main() {
    std::unordered_set<Todo> todos;

    todos.insert(Todo(1, "농구 하기"));
    todos.insert(Todo(2, "수학 숙제 하기"));
    todos.insert(Todo(1, "프로그래밍 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영화 보기"));
    print_unordered_set(todos);
    std::cout << "-----" << std::endl;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```
[중요도 : 2 ] 영화 보기
[중요도 : 1 ] 프로그래밍 프로젝트
[중요도 : 3 ] 친구 만나기
[중요도 : 1 ] 농구 하기
[중요도 : 2 ] 수학 숙제 하기
-----
```

와 같이 나옵니다.

먼저 Todo 를 위해 정의한 해시 함수를 살펴보도록 하겠습니다.

```
// Todo 해시 함수를 위한 함수객체(Functor)
// 를 만들어줍니다!
namespace std {
template <>
struct hash<Todo> {
    size_t operator()(const Todo& t) const {
        hash<string> hash_func;

        return t.priority ^ (hash_func(t.job_desc));
    }
};
} // namespace std
```

다행이도 C++ STL 에서는 기본적인 타입들 (`int`, `std::string` 등등)에 대한 해시 함수를 제공하기 때문에 우리의 Todo 클래스의 해시 함수는 이들을 잘 사용해서 짬뽕만 시키면 됩니다. 일단 `priority` 는 `int` 값 이므로 그냥 해시값 자체로 쓰기로 하고, `string` 의 해시값은 `hash_func` 객체로 이용해서 계산하면 됩니다

결과적으로 두 해시값을 짬뽕 시키기 위해서 XOR 연산을 이용하였습니다.

참고로 왜 `hash` 클래스가 `namespace std` 안에 정의되어 있느냐면 (이미 위에서 `using namespace std` 를 했음에도 불구하고), 특정 `namespace` 안에 새로운 클래스/함수를 추가하기 위해서는 위치별 명시적으로 `namespace` (이름) 를 써줘야만 합니다. ([여기를 참고](#))

그리고 마지막으로 아래와 같이 간단히 `==` 연산자를 추가해주면 됩니다.

```
bool operator==(const Todo& t) const {
    if (priority == t.priority && job_desc == t.job_desc) return true;
    return false;
}
```

그럼 위처럼 Todo 객체를 마음껏 `unordered_set`에서 사용할 수 있게 됩니다!

주의 사항

해시 함수를 직접 제작하는 일은 꽤나 어렵습니다. 가장 큰 이유로, `unordered_set`이나 `unordered_map`이 제대로 된 성능을 발휘하기 위해서는 해시 함수가 입력 받은 키를 잘 훑어야 합니다.

만일 해시 함수의 결과가 특정 범위의 값에게만 집중되어 있다면, 해시 함수를 이용한 컨테이너의 성능이 그냥 `map`이나 `set` 보다 훨씬 못하니만이 되게 됩니다. 특히 악의적인 사용자가 해당 허점을 이용해서 프로그램의 성능을 저하시킬 수도 있겠지요.

따라서 가장 권장하는 방식은 [여기](#)에 나온 기본 타입들의 대한 해시 함수들을 사용할 것이고, 해시 함수의 성능을 정확히 검증할 수 없다면, `map`이나 `set`을 사용하는 것이 나을 것입니다.

그렇다면 뭘 써야되?

아래와 같이 간단히 생각하시면 됩니다.

- 데이터의 존재 유무 만 궁금할 경우 → `set`
- 중복 데이터를 허락할 경우 → `multiset` (`insert`, `erase`, `find` 모두 $O(\log N)$. 최악의 경우에도 $O(\log N)$)
- 데이터에 대응되는 데이터를 저장하고 싶은 경우 → `map`
- 중복 키를 허락할 경우 → `multimap` (`insert`, `erase`, `find` 모두 $O(\log N)$. 최악의 경우에도 $O(\log N)$)
- 속도가 매우매우 중요해서 최적화를 해야하는 경우 → `unordered_set`, `unordered_map` (`insert`, `erase`, `find` 모두 $O(1)$. 최악의 경우엔 $O(N)$). 그러므로 해시함수와 상자 개수를 잘 설정해야 한다!)

그렇다면 이번 강좌는 여기에서 마치도록 하겠습니다. 다음 강좌에서는 STL 알고리즘을 이용한 여러가지 작업들에 대해 알아보도록 하겠습니다!

C++ 표준 알고리즘 라이브러리

그냥 C++ 의 <algorithm> 라이브러리의 함수들 목록을 보고 싶다면 [여기](#)를 클릭해주세요

안녕하세요 여러분! 이번 강좌에서는 C++ 표준 라이브러리의 알고리즘(algorithm) 라이브러리에 대해서 알아보도록 하겠습니다. 알고리즘 라이브러리는 앞선 강좌에서 이야기 했었던 대로, 컨테이너에 반복자들을 가지고 이런 저런 작업을 쉽게 수행할 수 있도록 도와주는 라이브러리입니다.

여기서 말하는 이런 저런 작업이란, 정렬이나 검색과 같이 단순한 작업들 말고도, '이런 조건이 만족하면 컨테이너에서 지워줘' 나 '이런 조건이 만족하면 1 을 더해' 와 같은 복잡한 명령의 작업들도 알고리즘 라이브러리를 통해 수행할 수 있습니다.

우리는 알고리즘에 정의되어 있는 여러가지 함수들로 작업을 수행하게 됩니다. 이 때 이 함수들은 크게 아래와 같은 두 개의 형태를 가지고 있습니다.

```
template <typename Iter> void do_something(Iter begin, Iter end);
```

거나

```
template <typename Iter, typename Pred> void do_something(Iter begin, Iter end, Pred pred)
```

와 같은 꼴을 따르고 있습니다. 전자의 경우, 알고리즘을 수행할 반복자의 시작점과 끝점 바로 뒤를 받고, 후자의 경우 반복자는 동일하게 받되, '특정한 조건' 을 추가 인자로 받게 됩니다. 이러한 '특정한 조건'을 서술자(**Predicate**) 이라고 부르며 저기 **Pred** 에는 보통 **bool** 을 리턴하는 함수 객체(**Functor**) 를 전달하게 됩니다. (이번 강좌에서 함수 객체를 매우 편리하게 만들어주는 람다 함수에 대해 다룰 것입니다!)

정렬 (sort, stable_sort, partial_sort)

첫번째로 알고리즘 라이브러리에서 지원하는 정렬(sort) 에 대해서 알아보도록 하겠습니다. 사실 정렬이라 하면 한 가지 밖에 없을 것 같은데 정렬 알고리즘에서는 무려 3 가지 종류의 함수를 지원하고 있습니다. 이를 살펴보자면 각각 다음과 같습니다.

- **sort** : 일반적인 정렬 함수라 생각하시면 됩니다.
- **stable_sort** : 정렬을 하되 원소들 간의 순서를 보존합니다. 이 말이 무슨 말이냐면, 만약에 벡터에 [a, b] 순으로 있었는데, a 와 b 가 크기가 같다면 정렬을 [a,b] 혹은 [b,a] 로 할 수 있습니다. **sort** 의 경우 그 순서가 랜덤으로 정해집니다. 하지만 **stable_sort** 의 경우 그 순서를 반드시 보존합니다. 즉 컨테이너 상에서 [a,b] 순으로 있었다면 정렬 시에도 (크기가 같다면) [a,b] 순으로 나오게 됩니다. 이 때문에 **sort** 보다 좀 더 느립니다.

- `partial_sort` : 배열의 일부분만 정렬합니다 (아래 자세히 설명하겠습니다)

그렇다면 각각의 함수들을 사용해보도록 하겠습니다!

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(6);
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(2);

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());
    std::sort(vec.begin(), vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
정렬 전 ----
5 3 1 6 4 7 2
정렬 후 ----
1 2 3 4 5 6 7
```

위와 같이 잘 정렬되어서 나옵을 알 수 있습니다.

```
sort(vec.begin(), vec.end());
```

`sort` 함수는 위와 같이 정렬할 원소의 시작 위치와, 마지막 위치 바로 뒤를 반복자로 받습니다.

참고로 `sort`에 들어가는 반복자의 경우 반드시 임의접근 반복자(RandomAccessIterator) 타입을 만족해야 하므로, 우리가 봐왔던 컨테이너들 중에선 벡터와 데크만 가능하고 나머지 컨테이너는 `sort` 함수를 적용할 수 없습니다. (예를 들어 리스트의 경우 반복자 타입이 양방향 반복자(BidirectionalIterator) 이므로 안됩니다)

```
list<int> l;
sort(l.begin(), l.end());
```

만약에 위 처럼 리스트를 정렬하려고 했다간;

컴파일 오류

```
Error C2784 'unknown-type std::operator -(const
→   std::move_iterator<_RanIt> &, const std::move_iterator<_RanIt2>
→   &)': could not deduce template argument for 'const
→   std::move_iterator<_RanIt> &' from
→   'std::_List_unchecked_iterator<std::_List_val<std::_List_simple_types<int>>
```

위와 같은 무시무시한 컴파일 오류를 맛보게 될 것입니다!

`sort` 함수는 기본적으로 오름차순으로 정렬을 해줍니다. 그렇다면 만약에 내림 차순으로 정렬하고 싶다면 어떻게 할까요? 만약에 여러분이 직접 만든 타입이였다면 단순히 `operator<`를 반대로 바꿔준다면 오름차순에서 내림차순이 되었겠지만, 이 경우 `int`이기 때문에 이는 불가능 합니다.

하지만 앞서 대부분의 알고리즘은 3 번째 인자로 특정한 조건을 전달한다고 하였는데, 여기에 우리가 비교를 어떻게 수행할 것인지에 대해 알려주면 됩니다.

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

struct int_compare {
    bool operator()(const int& a, const int& b) const { return a > b; }
};

int main() {
```

```

std::vector<int> vec;
vec.push_back(5);
vec.push_back(3);
vec.push_back(1);
vec.push_back(6);
vec.push_back(4);
vec.push_back(7);
vec.push_back(2);

std::cout << "정렬 전 ----" << std::endl;
print(vec.begin(), vec.end());
std::sort(vec.begin(), vec.end(), int_compare());

std::cout << "정렬 후 ----" << std::endl;
print(vec.begin(), vec.end());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

정렬 전 ----
5 3 1 6 4 7 2
정렬 후 ----
7 6 5 4 3 2 1

```

와 같이 내림 차순으로 정렬되어 나옵니다.

```

struct int_compare {
    bool operator()(const int& a, const int& b) const { return a > b; }
};

```

일단 위와 같이 함수 객체를 위한 구조체를 정의해주시고, 그 안에 **operator()** 함수를 만들어주면 함수 객체 준비는 땡입니다.

```
std::sort(vec.begin(), vec.end(), int_compare());
```

그리고 위와 같이 생성된 함수 객체를 전달하면 됩니다. 그런데 말입니다. 사실 **int**나 **string**과 같은 기본 타입들은 모두 <혹은> 연산자들이 기본으로 내장되어 있습니다. 그렇다면 굳이 그렇게 귀찮게 함수 객체를 만들 필요는 없을 것 같습니다. 템플릿도 배운 마당에 그냥

```

template <typename T>
struct greater_comp {

```

```
bool operator()(const T& a, const T& b) const { return a > b; }
```

요런게 있어서 굳이 귀찮게 `int` 따로 `string` 따로 만들 필요가 없을 것 같습니다. 다행이도 `functional` 해더에 다음과 같은 템플릿 클래스가 존재합니다.

```
std::sort(vec.begin(), vec.end(), greater<int>());
```

저 `greater`에 우리가 사용하고자 하는 타입을 넣게 되면 위와 같은 함수 객체를 자동으로 만들어 줍니다. 물론 그 해당하는 타입의 `>` 연산자가 존재해야겠지요. `int`의 경우 기본 타입이기 때문에 당연히 존재합니다.

다음으로 살펴볼 함수는 `partial_sort` 함수입니다.

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}
int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(6);
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(2);

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());
    std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
정렬 전 ----
5 3 1 6 4 7 2
정렬 후 ----
1 2 3 6 5 7 4
```

와 같이 나옵니다. 앞서 `partial_sort` 함수는 일부만 정렬하는 함수라고 하였습니다. `partial_sort` 는 인자를 아래와 같이 3 개를 기본으로 받습니다.

```
std::partial_sort(start, middle, end)
```

이 때 정렬을 `[start, end]` 전체 원소들 중에서 `[start, middle]` 까지 원소들이 전체 원소들 중에서 제일 작은애들 순으로 정렬 시킵니다. 예를 들어서 위 경우

```
std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());
```

위와 같이 `vec.begin()` 부터 `vec.end()` 까지 (즉 벡터 전체에서) 원소들 중에서, `vec.begin()` 부터 `vec.begin() + 3` 까지에 전체에서 가장 작은 애들만 순서대로 저장하고 나머지 위치는 상관 없다! 이런 식입니다. 따라서 위와 같이

```
5 3 1 6 4 7 2
```

에서 가장 작은 3개 원소인 1, 2, 3 만이 정렬되어서

```
1 2 3 6 5 7 4
```

앞에 나타나게 되고 나머지 원소들은 그냥 랜덤하게 남아있게 됩니다. 전체 원소의 개수가 N 개이고, 정렬하려는 부분의 크기가 M 이라면 `partial_sort` 의 복잡도는 $O(N \log M)$ 가 됩니다.

만약에 우리가 전체 배열을 정렬할 필요가 없을 경우, 예를 들어서 100 명의 학생 중에서 상위 10 명의 학생의 성적순을 보고 싶다, 이런 식이면 굳이 `sort` 로 전체를 정렬 할 필요 없이 `partial_sort` 로 10 개만 정렬 하는 것이 더 빠르게 됩니다.

마지막으로 `stable_sort` 에 대해 살펴보도록 하겠습니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>
```

```
template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

struct User {
    std::string name;
    int age;

    User(std::string name, int age) : name(name), age(age) {}

    bool operator<(const User& u) const { return age < u.age; }
};

std::ostream& operator<<(std::ostream& o, const User& u) {
    o << u.name << " , " << u.age;
    return o;
}

int main() {
    std::vector<User> vec;
    for (int i = 0; i < 100; i++) {
        std::string name = "";
        name.push_back('a' + i / 26);
        name.push_back('a' + i % 26);
        vec.push_back(User(name, static_cast<int>(rand() % 10)));
    }

    std::vector<User> vec2 = vec;

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::sort(vec.begin(), vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "stable_sort 의 경우 ---" << std::endl;
    std::stable_sort(vec2.begin(), vec2.end());
    print(vec2.begin(), vec2.end());
}
```

성공적으로 컴파일 하였다면

Press any key to continue . . .

와 같이 나옵니다.

앞서 `stable_sort` 는 원소가 삽입되어 있는 순서를 보존하는 정렬 방식이라고 하였습니다. `stable_sort` 가 확실히 어떻게 `sort` 와 다른지 보여주기 위해서 다음과 같은 클래스를 만들었습니다.

```
struct User {
    std::string name;
    int age;

    User(std::string name, int age) : name(name), age(age) {}

    bool operator<(const User& u) const { return age < u.age; }
};
```

이 `User` 클래스는 `name` 과 `age` 를 멤버로 갖는데, 크기 비교는 이름과 관계없이 모두 `age` 로 하게 됩니다. 즉 `age` 가 같다면 크기가 같다고 볼 수 있습니다.

```
for (int i = 0; i < 100; i++) {
    std::string name = "";
    name.push_back('a' + i / 26);
    name.push_back('a' + i % 26);
    vec.push_back(User(name, static_cast<int>(rand() % 10)));
}
```

처음에 벡터에 원소들을 쭈르륵 삽입하는 부분인데, 이름은 `aa`, `ab`, `ac`, ... 순으로 하되 `age` 의 경우 0 부터 10 사이의 랜덤한 값을 부여하였습니다. 즉 `name` 의 경우 `string` 순서대로 되어 있고, `age` 의 경우 랜덤한 순서로 되어 있습니다.

앞서 말했듯이 `stable_sort` 는 삽입되어 있던 원소들 간의 순서를 보존한다고 하였습니다. 따라서 같은 `age` 라면 반드시 삽입된 순서, 즉 `name` 순으로 나올 것입니다. (왜냐하면 애초에 `name` 순으로 넣었기 때문!)

그 결과를 살펴보면 확연히 다름을 알 수 있습니다. 먼저 `sort`의 경우

```
dh, ck, cx, ad, cw, cu, co
```

순으로 나와 있고 (`age` 가 0 일 때) `stable_sort`의 경우 `age` 가 0 일 때

```
ad, ck, co, cu, cw, cx, dh
```

순으로 나오게 됩니다. 다시 말해 `sort` 함수의 경우 정렬 과정에서 원소들 간의 상대적 위치를 랜덤하게 바꿔버리지만 `stable_sort`의 경우 그 순서를 처음에 넣었던 상태 그대로 유지함을 알 수 있습니다.

당연히도 이러한 제약 조건 때문에 `stable_sort`는 그냥 `sort` 보다 좀 더 오래걸립니다. C++ 표준에 따르면 `sort` 함수는 최악의 경우에서도 $O(n \log n)$ 이 보장되지만 `stable_sort`의 경우 최악의 경우에서 $O(n (\log n)^2)$ 으로 작동하게 됩니다. 조금 더 느린 편이지요.

원소 제거 (`remove`, `remove_if`)

다음으로 살펴볼 함수는 원소를 제거하는 함수입니다. 사실 이미 대부분의 컨테이너에서는 원소를 제거하는 함수를 지원하고 있습니다. 예를 들어서,

```
std::vector<int> vec;
// ....
vec.erase(vec.begin() + 3);
```

을 하게 되면, `vec[3]` 에 해당하는 원소를 제거하게 됩니다.

그런데 사실 이 함수 하나로는 많은 작업들을 처리하기에 부족합니다. 예를 들어서 벡터에서 값이 3인 원소를 제거하려면 어떻게 해야 할까요? 이전 강좌에서 다루었지만 아마 아래와 같이 할 수 있을 것입니다.

```
std::vector<int>::iterator itr = vec.begin();

for ( ; itr != vec.end(); ++itr) {
    if (*itr == 3) {
        vec.erase(itr);
        itr = vec.begin();
    }
}
```

이렇게 했던 이유는 바로 원소가 제거될 때마다 기존에 제거하였던 반복자들이 초기화 되기 때문입니다. 따라서 해당 위치를 가리키는 반복자를 다시 가져와야 되지요. 물론 굳이 반복자를 쓰지

않고 그냥 일반 변수를 이용해서 배열을 다루듯이 처리할 수도 있겠지만 '원소 접근은 반복자로 수행한다'에 따른 약속에는 충실한 방법이 아닙니다.

그렇다면 어떻게 이를 해결할 수 있을까요?

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 값이 3 인 원소 제거 ---" << std::endl;
    vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
    print(vec.begin(), vec.end());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 값이 3 인 원소 제거 ---
[5] [1] [2] [4]
```

와 같이 나옵니다.

위 코드가 어떻게 작동하는지 설명하기에 앞서 `erase` 함수를 살펴보도록 합시다. 벡터의 `erase` 함수는 2 가지 형태가 있는데, 하나는 우리가 잘 알고 있는

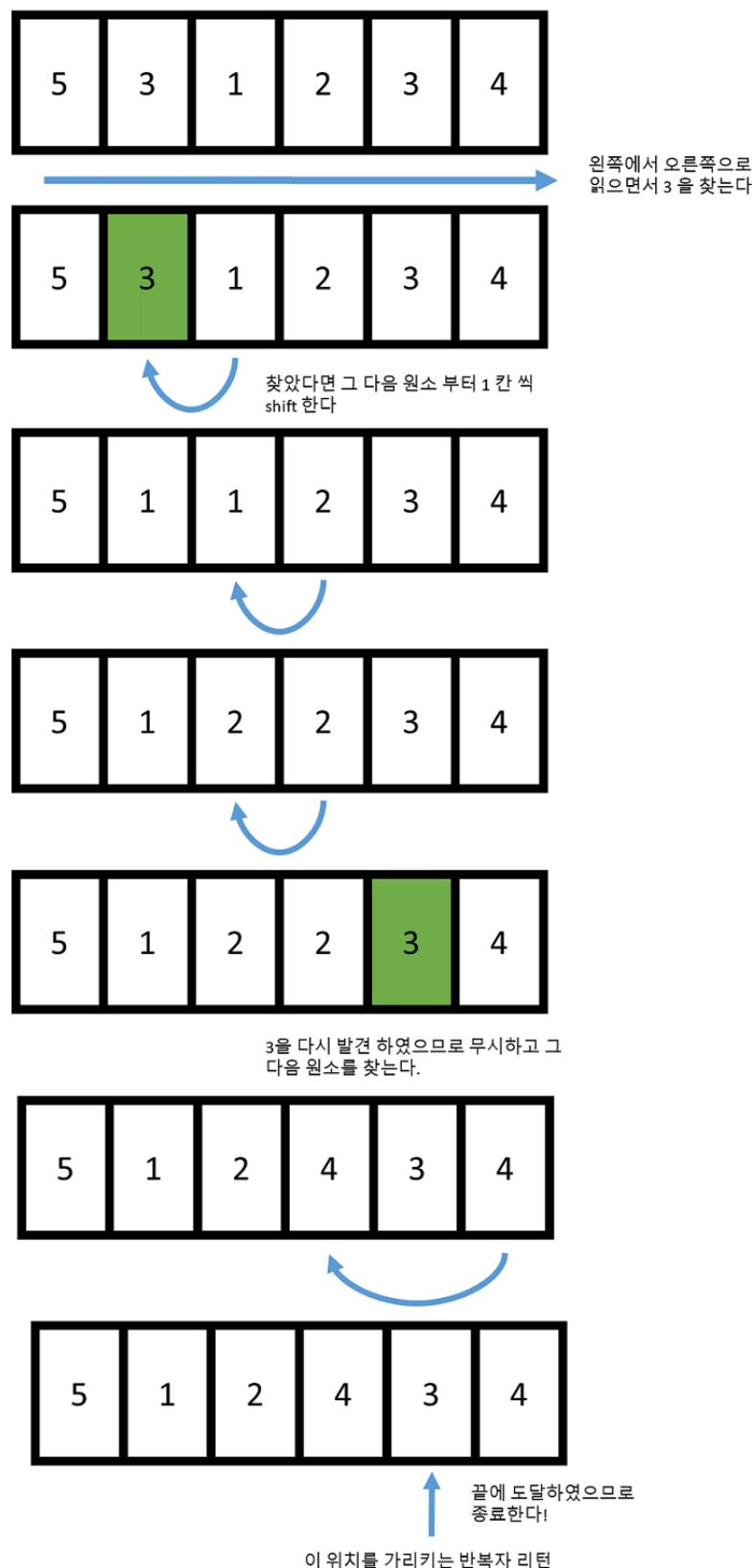
```
Iterator erase(Iterator pos);
```

와 같은 형태가 있고, 다른 하나는

```
Iterator erase(Iterator first, Iterator last);
```

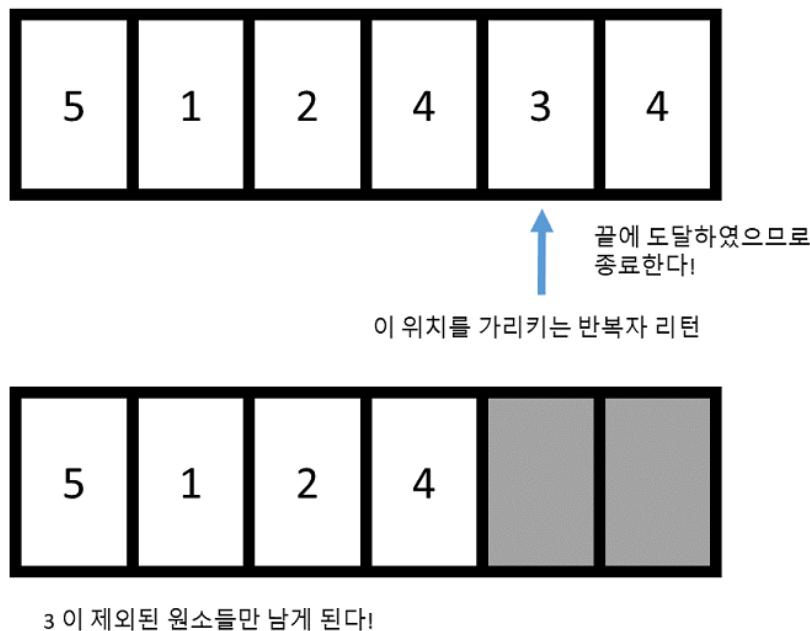
와 같은 형태가 있습니다. 전자의 경우 그냥 `pos` 가 가리키는 원소를 벡터에서 지우지만 후자의 경우 `first` 부터 `last` 사이에 있는 모든 원소들을 지우는 형태입니다. 물론 이 두 함수 모두 우리의 목표인 '값이 3 인 원소 제거'를 수행하는데 부족함이 있습니다. 물론 후자의 함수를 사용하면 좋겠지만, 값이 3 인 원소들이 벡터에서 연속적으로 존재하지 않기 때문이지요.

하지만 어떤 편리한 함수가 있어서 값이 3 인 원소들을 벡터에서 연속적으로 존재할 수 있게 해주면 어떨까요?



위와 같이, 만일 값이 3인 원소를 만나면 그 뒤에 있는 원소들로 주르륵 쉬프트 해주게 됩니다.

따라서, 자연스럽게 알고리즘이 끝나게 되면은 해당하는 위치에서 전 까지 3 이 제외된 원소들로
주르륵 채워지게 되겠지요.



다시말해, 반복이 끝나는 위치 부터 벡터 맨 뒤 까지 제거해버리면 3 이 짙 제거된 벡터만 남게
되지요. `remove` 함수는 원소의 이동만을 수행하지 실제로 원소를 삭제하는 연산을 수행하지는
않습니다. 따라서 벡터에서 실제로 원소를 지우기 위해서는 반드시 `erase` 함수를 호출하여 실제로
원소를 지워줘야만 합니다.

```
vec.erase(remove(vec.begin(), vec.end(), 3), vec.end());
```

따라서 위처럼 `remove` 함수를 이용해서 값이 3 인 원소들을 뒤로 보내버리고, 그 원소들을 벡터에
서 삭제해버리게 됩니다.

참고로 말하자면 `remove` 함수의 경우 반복자의 타입이 `ForwardIterator` 입니다. 즉, 벡터
뿐만이 아니라, 리스트, 혹은 셋이나 맵에서도 모두 사용할 수 있습니다!

그렇다면 이번에는 값이 딱 얼마로 정해진 것이 아니라 특정한 조건을 만족하는 원소들을 제거하려면
어떻게 해야 할까요? 당연히도 이 원소가 그 조건을 만족하는지 아닌지를 판단할 함수를 전달해야
됩니다. 이를 위해선 `remove_if` 함수를 사용해야 합니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>
```

```

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

struct is_odd {
    bool operator()(const int& i) { return i % 2 == 1; }
};

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수 인 원소 제거 ---" << std::endl;
    vec.erase(std::remove_if(vec.begin(), vec.end(), is_odd()), vec.end());
    print(vec.begin(), vec.end());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 홀수 인 원소 제거 ---
[2] [4]

```

와 같이 나옵니다.

```
vec.erase(remove_if(vec.begin(), vec.end(), is_odd()), vec.end());
```

`remove_if` 함수는 세번째 인자로 조건을 설명할 함수 객체를 전달받습니다.

```

struct is_odd {
    bool operator()(const int& i) { return i % 2 == 1; }
};

```

위와 같이 `is_odd` 구조체에 `operator()`를 만들어서 함수 객체를 전달하시면 됩니다. 당연히도, 함수 객체로 실제 함수를 전달할 수도 있습니다. 이 경우

```
template <typename Iter, typename Pred>
remove_if(Iter first, Iter last, Pred pred)
```

에서 `Pred` 가 함수 포인터 타입이 되겠지요.

```
bool odd(const int& i) { return i % 2 == 1; }
int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수 인 원소 제거 ---" << std::endl;
    vec.erase(std::remove_if(vec.begin(), vec.end(), odd), vec.end());
    print(vec.begin(), vec.end());
}
```

위와 같이 실제 함수를 전달한다면 앞서 만들었던 함수 객체와 정확히 동일하게 동작합니다.

remove_if에 조건 추가하기

우리의 `remove_if` 함수는 함수 객체가 인자를 딱 1 개 만 받는다고 가정합니다. 따라서 호출되는 `operator()` 을 통해선 원소에 대한 정보 말고는 추가적인 정보를 전달하기는 어렵습니다.

하지만 예를 들어서 홀수인 원소들을 삭제하되 처음 2개만 삭제한다고 해봅시다. 함수 객체의 경우 사실 클래스의 객체이기 때문에 멤버 변수를 생각할 수 있습니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
```

```

    std::cout << "[" << *begin << "] ";
    begin++;
}
std::cout << std::endl;
}

struct is_odd {
    int num_delete;

    is_odd() : num_delete(0) {}

    bool operator()(const int& i) {
        if (num_delete >= 2) return false;

        if (i % 2 == 1) {
            num_delete++;
            return true;
        }

        return false;
    }
};

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수인 원소 앞의 2개 제거 ---" << std::endl;
    vec.erase(std::remove_if(vec.begin(), vec.end(), is_odd()), vec.end());
    print(vec.begin(), vec.end());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 홀수인 원소 앞의 2개 제거 ---
[2] [3] [4]

```

와 같이 나옵니다.

```

struct is_odd {
    int num_delete;

    is_odd() : num_delete(0) {}

    bool operator()(const int& i) {
        if (num_delete >= 2) return false;

        if (i % 2 == 1) {
            num_delete++;
            return true;
        }

        return false;
    }
};

```

예상과는 사뭇 다른 결과가 나왔습니다. 홀수 원소 2 개가 아니라 3 개가 삭제됐네요. 분명히 우리는 2개 이상 되면 `false` 를 리턴하라고 명시했는데도 말이지요.

사실 C++ 표준에 따르면 `remove_if` 에 전달되는 함수 객체의 경우 이전의 호출에 의해 내부 상태가 달라지면 안됩니다. 다시 말해, 위처럼 함수 객체 안에 인스턴스 변수 (`num_delete`) 를 넣는 것은 원칙상 안된다는 것이지요.

그 이유는 `remove_if` 를 실제로 구현 했을 때, 해당 함수 객체가 여러번 복사 될 수 있기 때문입니다. 물론, 이는 어떻게 구현하느냐에 따라서 달라집니다. 예를 들어 아래 버전을 살펴볼까요.

```

template <class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                         UnaryPredicate pred) {
    ForwardIterator result = first;
    while (first != last) {
        if (!pred(*first)) { // <-- 함수 객체 pred 를 실행하는 부분
            *result = std::move(*first);
            ++result;
        }
        ++first;
    }
    return result;
}

```

위 버전에 경우 인자로 전달된 함수 객체 `pred` 는 복사되지 않고 계속 호출됩니다. 따라서 사실 우리의 원래 코드가 위 `remove_if` 를 바탕으로 실행됐더라면 2 개만 정확히 지워질 수 있습니다. 하지만 문제는 C++ 표준은 `remove_if` 함수를 어떤 방식으로 구현하라고 정해 놓지 않습니다. 어떤 라이브러리들의 경우 아래와 같은 방식으로 구현되었습니다 (사실 대부분의 라이브러리들이

아래와 비슷합니다.)

```
template <class ForwardIt, class UnaryPredicate>
ForwardIt remove_if(ForwardIt first, ForwardIt last, UnaryPredicate pred) {
    first = std::find_if(first, last, pred); // <- pred 한 번 복사됨
    if (first != last)
        // 아래 for 문은 first + 1 부터 시작한다고 봐도 된다 (++i != last)
        for (ForwardIt i = first; ++i != last;)
            if (!pred(*i)) // <-- pred 호출 하는 부분
                *first++ = std::move(*i);
    return first;
}
```

참고로 `find_if` 함수의 경우 인자로 전달된 조건 `pred` 가 참인 첫 번째 원소를 리턴합니다. 그런데 문제는 `find_if` 가 함수 객체 `pred` 의 레퍼런스를 받는 것이 아니라, 복사 생성된 버전을 받는다는 점입니다. 따라서, `find_if` 호출 후에 아래 `for` 문에서 이미 한 개 원소를 지웠다는 정보가 소멸되게 됩니다. 후에 호출되는 `pred` 들은 이미 `num_delete` 가 1 인지 모른 채 0 부터 다시 시작하게 되죠.

다시 한 번 말하자면, 함수 객체에는 절대로 특정 상태를 저장해서 이에 따라 결과가 달라지는 루틴을 짜면 안됩니다. 위처럼 이해하기 힘든 오류가 발생할 수도 있습니다.

그렇다면 위 문제를 어떻게 하면 해결할 수 있을까요? 한 가지 방법은 `num_delete` 를 객체 내부 변수가 아니라 외부 변수로 빼는 방법입니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

struct is_odd {
    int* num_delete;

    is_odd(int* num_delete) : num_delete(num_delete) {}

    bool operator()(const int& i) {
        if (*num_delete >= 2) return false;

        if (i % 2 == 1) {
```

```

        (*num_delete)++;
    return true;
}

return false;
}
};

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수인 원소 앞의 2개 제거 ---" << std::endl;
    int num_delete = 0;
    vec.erase(std::remove_if(vec.begin(), vec.end(), is_odd(&num_delete)),
              vec.end());
    print(vec.begin(), vec.end());
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 홀수인 원소 앞의 2개 제거 ---
[1] [2] [3] [4]

```

와 같이 제대로 나옵니다. 위 경우, `num_delete`에 관한 정보를 아예 함수 객체 밖으로 빼서 보관해버렸습니다. 물론 함수 객체에 내부 상태인 `num_delete`의 주소값은 변하지 않으므로 문제될 것이 없습니다.

그런데 한 가지 안 좋은 점은 이렇게 STL을 사용할 때마다 외부에 클래스나 함수를 하나 씩 만들어줘야 된다는 점입니다. 물론 프로젝트의 크기가 작다면 크게 문제가 되지는 않겠지만 프로젝트의 크기가 커진다면, 만약 다른 사람이 코드를 읽을 때 '이 클래스는 뭐하는 거지?' 혹은 '이 함수는 뭐하는 거지?'와 같은 궁금증이 생길 수도 있고 심지어 잘못 사용할 수도 있습니다.

따라서 가장 이상적인 방법은 STL 알고리즘을 사용할 때 그 안에 직접 써놓는 것입니다. 마치

```
vec.erase(std::remove_if(vec.begin(), vec.end(),
                        bool is_odd(int i) { return i % 2 == 1; }),
          vec.end());
```

뭐 이런 식으로 말이지요. 문제는 위 문법이 C++에서 허용되지 않는 점입니다. 하지만 놀랍게도 C++ 11부터 위 문제를 해결할 방법이 나타났습니다.

람다 함수(lambda function)

람다 함수는 C++에서는 C++ 11에서 처음으로 도입되었습니다. 람다 함수를 통해 쉽게 이름이 없는 함수 객체를 만들수 있게 되었습니다. 그렇습니다. 익명의 함수 객체 말입니다.

람다 함수를 사용한 예제 부터 먼저 살펴보겠습니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수인 원소 제거 ---" << std::endl;
    vec.erase(std::remove_if(vec.begin(), vec.end(),
                           [] (int i) -> bool { return i % 2 == 1; })),
    vec.end());
    print(vec.begin(), vec.end());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 홀수인 원소 제거 ---
[2] [4]
```

와 같이 나옵니다.

람다 함수를 정의한 부분부터 살펴보도록 합시다.

```
[](int i) -> bool { return i % 2 == 1; }
```

람다 함수는 위와 같은 꼴로 정의됩니다. 일반적인 꼴을 살펴보자면

[capture list] (받는 인자) -> 리턴 타입 { 함수 본체 }

와 같은 형태입니다. `capture_list` 가 뭔지는 아래에서 설명하도록 하고, 위 함수 꼴을 살펴보자면 인자로 `int i` 를 받고, `bool` 을 리턴하는 람다 함수를 정의한 것입니다. 리턴 타입을 생략한다면 컴파일러가 알아서 함수 본체에서 `return` 문을 보고 리턴 타입을 추측해줍니다. (만약에 `return` 경로가 여러군데여서 추측할 수 없다면 컴파일 오류가 발생하지요)

리턴 타입을 생략할 경우

[capture list] (받는 인자) {함수 본체}

이런 식으로 더 간단히 쓸 수 있습니다. 위 예제의 경우

```
[](int i) { return i % 2 == 1; }
```

로 쓴다면 알아서 "아 `bool` 타입을 리턴하는 함수구나" 라고 컴파일러가 만들어줍니다.

앞서 람다 함수가 이름이 없는 함수라 했는데 실제로 위를 보면 함수에 이름이 붙어 있지 않습니다! 즉 임시적으로 함수를 생성한 것이지요. 만약에 이 함수를 사용하고 싶다면

```
[](int i) { return i % 2 == 1; }(3); // true
```

와 같이 그냥 바로 호출할 수 도 있고

```
auto func = [](int i) { return i % 2 == 1; };
func(4); // false;
```

람다 함수로 `func`이라는 함수 객체를 생성한 후에 호출할 수도 있지요.

하지만 람다 함수도 말 그대로 함수 이기 때문에 자기 자신만의 스코프를 가집니다. 따라서 일반적인 상황이라면 함수 외부에서 정의된 변수들을 사용할 수 없겠지요. 예를 들어서 최대 2 개 원소만 지우고 싶은 경우

```
std::cout << "벡터에서 홀수인 원소 최대 2 개 제거 ---" << std::endl;
int num_erased = 0;
vec.erase(std::remove_if(vec.begin(), vec.end(),
    [](int i) {
        if (num_erased >= 2)
            return false;
        else if (i % 2 == 1) {
            num_erased++;
            return true;
        }
        return false;
}), vec.end());
print(vec.begin(), vec.end());
```

위와 같이 람다 함수 외부에 몇 개를 지웠는지 변수를 정의한 뒤에 사용해야만 하는데 (함수 안에 정의하면 함수 호출될 때마다 새로 생성되니까요!) 문제는 그 변수에 접근할 수 없다는 점입니다. 하지만 놀랍게도 람다 함수의 경우 그 변수에 접근할 수 있습니다. 바로 캡쳐 목록(capture list)을 사용하는 것입니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}
int main() {
    std::vector<int> vec;
    vec.push_back(5);
```

```

vec.push_back(3);
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);

std::cout << "처음 vec 상태 -----" << std::endl;
print(vec.begin(), vec.end());

std::cout << "벡터에서 홀수인 원소 ---" << std::endl;
int num_erased = 0;
vec.erase(std::remove_if(vec.begin(), vec.end(),
    [&num_erased](int i) {
        if (num_erased >= 2)
            return false;
        else if (i % 2 == 1) {
            num_erased++;
            return true;
        }
        return false;
}),
vec.end());
print(vec.begin(), vec.end());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터에서 홀수인 원소 ---
[1] [2] [3] [4]

```

와 같이 잘 됨을 알 수 있습니다.

```

[&num_erased](int i) {
    if (num_erased >= 2)
        return false;
    else if (i % 2 == 1) {
        num_erased++;
        return true;
    }
    return false;
}

```

위와 같이 캡쳐 목록에는 어떤 변수를 캡쳐할지 써주면 됩니다. 위 경우 `num_erased`를 캡쳐하였습니다. 즉 람다 함수 내에서 `num_erased`를 마치 같은 스코프 안에 있는 것처럼 사용할 수 있게 됩니다.

이 때 `num_erased` 앞에 `&` 가 붙어있는데 이는 실제 `num_erased`의 레퍼런스를 캡쳐한다는 의미입니다. 즉 함수 내부에서 `num_erased`의 값을 바꿀 수 있게 되지요. 만약에 아래처럼

```
[num_erased](int i){
    if (num_erased >= 2)
        return false;
    else if (i % 2 == 1) {
        num_erased++;
        return true;
    }
    return false;
})
```

`&` 를 앞에 붙이지 않는다면 `num_erased`의 복사본을 얻게 되는데, 그 복사본의 형태는 `const`입니다. 따라서 위처럼 함수 내부에서 `num_erased`의 값을 바꿀 수 없게 되지요. 그렇다면 클래스의 멤버 함수 안에서 람다를 사용할 때 멤버 변수들을 참조하려면 어떻게 해야 할까요?

```
class SomeClass {
    std::vector<int> vec;

    int num_erased;

public:
    SomeClass() {
        vec.push_back(5);
        vec.push_back(3);
        vec.push_back(1);
        vec.push_back(2);
        vec.push_back(3);
        vec.push_back(4);

        num_erased = 1;

        vec.erase(std::remove_if(vec.begin(), vec.end(),
                               [&num_erased](int i) {
                                   if (num_erased >= 2)
                                       return false;
                                   else if (i % 2 == 1) {
                                       num_erased++;
                                       return true;
                                   }
                                   return false;
                               }),
                  vec.end());
    }
}
```

```
    }
};
```

예를 들어 위와 같은 예제를 생각해봅시다. 쉽게 생각해보면 그냥 똑같이 `num_erased` 를 & 로 캡쳐해서 람다 함수 안에서 사용할 수 있을 것 같지만 실제로는 컴파일 되지 않습니다. 왜냐하면 `num_erased` 가 일반 변수가 아니라 객체에 종속되어 있는 멤버 변수 이기 때문이지요. 즉 람다 함수는 `num_erased` 를 캡쳐해! 라고 하면 이 `num_erased` 가 이 객체의 멤버 변수가 아니라 그냥 일반 변수라고 생각하게 됩니다.

이를 해결하기 위해선 직접 멤버 변수를 전달하기 보다는 `this` 를 전달해주면 됩니다.

```
num_erased = 0;

vec.erase(std::remove_if(vec.begin(), vec.end(),
    [this](int i) {
        if (this->num_erased >= 2)
            return false;
        else if (i % 2 == 1) {
            this->num_erased++;
            return true;
        }
        return false;
}),
vec.end());
```

위와 같이 `this` 를 복사본으로 전달해서 (참고로 `this` 는 레퍼런스로 전달할 수 없습니다) 함수 안에서 `this` 를 이용해서 멤버 변수들을 참조해서 사용하면 됩니다.

위에 설명한 경우 말고도 캡쳐 리스트의 사용 방법은 꽤나 많은데 아래 간단히 정리해보도록 하겠습니다.

- [] : 아무것도 캡쳐 안함
- [&a, b] : a 는 레퍼런스로 캡쳐하고 b 는 (변경 불가능한) 복사본으로 캡쳐
- [&] : 외부의 모든 변수들을 레퍼런스로 캡쳐
- [=] : 외부의 모든 변수들을 복사본으로 캡쳐

와 같이 되겠습니다.

원소 수정하기 (transform)

다음으로 살펴볼 함수는 원소들을 수정하는 함수들입니다. 많은 경우 컨테이너 전체 혹은 일부를 순회하면서 값들을 수정하는 작업을 많이 할 것입니다. 예를 들어서 벡터의 모든 원소에 1 씩 더한다와 같은 작업들을 말이지요. 이러한 작업을 도와주는 함수는 바로 `transform` 함수입니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 -----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터 전체에 1 을 더한다" << std::endl;
    std::transform(vec.begin(), vec.end(), vec.begin(),
                  [] (int i) { return i + 1; });
    print(vec.begin(), vec.end());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
처음 vec 상태 -----
[5] [3] [1] [2] [3] [4]
벡터 전체에 1 을 더한다
```

[6] [4] [2] [3] [4] [5]

와 같이 나옵니다.

`transform` 함수는 다음과 같은 꼴로 생겼습니다.

`transform (시작 반복자, 끝 반복자, 결과를 저장할 컨테이너의 시작 반복자, Pred)`

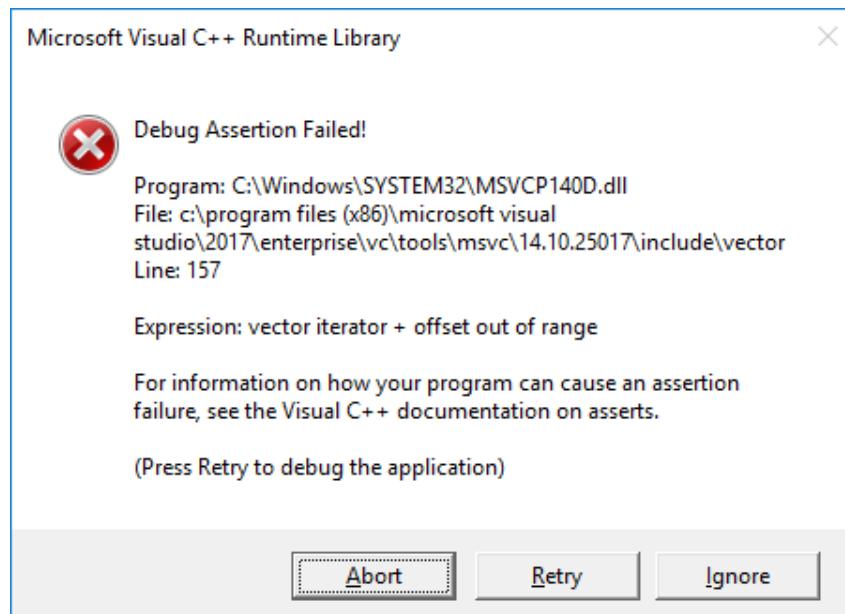
우리가 사용한 예의 경우

```
transform(vec.begin(), vec.end(), vec.begin(), [](int i) { return i + 1; });
```

로 하였으므로 `vec` 의 시작(begin) 부터 끝(end) 까지 각 원소에 `[] (int i) {return i + 1}` 함수를 적용시킨 결과를 `vec.begin()` 부터 저장하게 됩니다. 즉 결과적으로 각 원소에 1 을 더한 결과로 덮어 씌우게 되는 것이지요. 상당히 간단합니다. 한 가지 주의할 점은 값을 저장하는 컨테이너의 크기가 원래의 컨테이너보다 최소한 같거나 커야 된다는 점입니다. 예를 들어서 단순하게

```
std::transform(vec.begin(), vec.end(), vec.begin() + 1,
              [](int i) { return i + 1; });
```

이렇게 썼다고 해봅시다. `transform` 함수는 `vec` 의 처음부터 끝까지 주르륵 순회하지만 저장하는 쪽의 반복자는 `vec` 의 두 번째 원소부터 저장하기 때문에 결과적으로 마지막에 한 칸이 모잘라서



위와 같은 오류를 발생하게 됩니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    // vec2 에는 6 개의 0 으로 초기화 한다.
    std::vector<int> vec2(6, 0);

    std::cout << "처음 vec 과 vec2 상태 -----" << std::endl;
    print(vec.begin(), vec.end());
    print(vec2.begin(), vec2.end());

    std::cout << "vec 전체에 1 을 더한 것을 vec2 에 저장 -- " << std::endl;
    std::transform(vec.begin(), vec.end(), vec2.begin(),
                  [] (int i) { return i + 1; });
    print(vec.begin(), vec.end());
    print(vec2.begin(), vec2.end());
}
```

성공적으로 컴파일 하였으면

실행 결과

```
처음 vec 과 vec2 상태 -----
[5] [3] [1] [2] [3] [4]
[0] [0] [0] [0] [0] [0]
vec 전체에 1 을 더한 것을 vec2 에 저장 --
```

```
[5] [3] [1] [2] [3] [4]
[6] [4] [2] [3] [4] [5]
```

와 같이 나옵니다.

```
std::transform(vec.begin(), vec.end(), vec2.begin(),
    [](int i) { return i + 1; });
```

위와 같이 `vec` 의 처음부터 끝 까지 읽으면서 1씩 더한 결과를 `vec2`에 저장하게 됩니다. 간단하지요! 물론 저 `transform` 함수 하나 덕분에 `for` 문을 쓸 필요도 없어질 뿐더러, 내가 이 코드에서 무슨 일을 하는지 더 간단 명료하게 나타낼 수도 있습니다.

원소를 탐색하는 함수(`find`, `find_if`, `any_of`, `all_of` 등 등)

마지막으로 살펴볼 함수들은 원소들을 탐색하는 계열의 함수들입니다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    auto result = std::find(vec.begin(), vec.end(), 3);
    std::cout << "3 은 " << std::distance(vec.begin(), result) + 1 << " 번째 원소"
```

```

        << std::endl;
}
```

성공적으로 컴파일 하였으면

실행 결과

3 은 2 번째 원소

와 같이 나옵니다.

`find` 함수는 단순히

```

template <class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value)
```

와 같이 생겼는데, `first` 부터 `last` 까지 주르륵 순회하면서 `value` 와 같은 원소가 있는지 확인하고 있으면 이를 가리키는 반복자를 리턴합니다. 위 경우

```
auto result = std::find(vec.begin(), vec.end(), 3);
```

`vec`에서 값이 3과 같은 원소를 찾아서 리턴하게 되지요. 반복자에 따라서 `forward_iterator`면 앞에서부터 찾고, `reverse_iterator`이면 뒤에서부터 거꾸로 찾게 됩니다. 물론 컨테이너에 중복되는 값이 있더라도 가장 먼저 찾은 것을 리턴합니다. 만약에 위 `vec`에서 모든 3을 찾고 싶다면 아래와 같이 하면 됩니다.

```

#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
}
```

```

vec.push_back(3);
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);

auto current = vec.begin();
while (true) {
    current = std::find(current, vec.end(), 3);
    if (current == vec.end()) break;
    std::cout << "3 은 " << std::distance(vec.begin(), current) + 1
        << " 번째 원소" << std::endl;
    current++;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

3 은 2 번째 원소
3 은 5 번째 원소

```

위와 같이 나옵니다.

```
current = find(current, vec.end(), 3);
```

위처럼 마지막으로 찾은 위치 바로 다음부터 계속 순차적으로 탐색해 나간다면 컨테이너에서 값이 3인 원소들을 모두 찾을 수 있게 됩니다.

다만 `find` 계열의 함수들을 사용할 때 한 가지 주의해야 할 점은, 만약에 컨테이너에서 기본적으로 `find` 함수를 지원한다면 이를 사용하는 것이 훨씬 빠릅니다. 왜냐하면 알고리즘 라이브러리에서의 `find` 함수는 그 컨테이너가 어떠한 구조를 가지고 있는지에 대한 정보가 하나도 없기 때문입니다.

예를 들어 `set`의 경우, `set`에서 사용하는 `find` 함수의 경우 $O(\log n)$ 으로 수행될 수 있는데 그 이유는 셋 내부에서 원소들이 정렬되어 있기 때문입니다. 또 `unordered_set`의 경우 `find` 함수가 $O(1)$ 로 수행될 수 있는데 그 이유는 `unordered_set` 내부에서 자체적으로 해시 테이블을 이용해서 원소들을 빠르게 탐색해 나갈 수 있기 때문입니다.

하지만 그냥 알고리즘 라이브러리의 `find` 함수의 경우 이러한 추가 정보가 있는 것을 하나도 모른채 우직하게 처음부터 하나씩 확인해 나가므로 평범한 $O(n)$ 으로 처리됩니다. 따라서 알고리즘 라이브러리의 `find` 함수를 사용할 경우 벡터와 같이 기본적으로 `find` 함수를 지원하지 않는 컨테이너에 사용하시기 바랍니다!

```

#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    auto current = vec.begin();
    while (true) {
        current =
            std::find_if(current, vec.end(), [] (int i) { return i % 3 == 2; });
        if (current == vec.end()) break;
        std::cout << "3 으로 나눈 나머지가 2 인 원소는 : " << *current << " 이다 "
            << std::endl;
        current++;
    }
}

```

성공적으로 컴파일 하였다면

실행 결과

```

3 으로 나눈 나머지가 2 인 원소는 : 5 이다
3 으로 나눈 나머지가 2 인 원소는 : 2 이다

```

와 같이 나옵니다.

```
current = std::find_if(current, vec.end(), [] (int i) { return i % 3 == 2; });
```

`find` 함수가 단순한 값을 받았다면 `find_if` 함수의 경우 함수 객체를 인자로 받아서 그 결과가 참인 원소들을 찾게 됩니다. 위 경우 3 으로 나눈 나머지가 2 인 원소들을 컨테이너에서 탐색하였습니다. 람다 함수로 사용하니 엄청 간결하지요?

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << "] ";
        begin++;
    }
    std::cout << std::endl;
}

struct User {
    std::string name;
    int level;

    User(std::string name, int level) : name(name), level(level) {}
    bool operator==(const User& user) const {
        if (name == user.name && level == user.level) return true;
        return false;
    }
};

class Party {
    std::vector<User> users;

public:
    bool add_user(std::string name, int level) {
        User new_user(name, level);
        if (std::find(users.begin(), users.end(), new_user) != users.end()) {
            return false;
        }
        users.push_back(new_user);
        return true;
    }

    // 파티원 모두가 15 레벨 이상이여야지 던전 입장 가능
    bool can_join_dungeon() {
        return std::all_of(users.begin(), users.end(),
                           [] (User& user) { return user.level >= 15; });
    }

    // 파티원 중 한명이라도 19렙 이상이면 특별 아이템 사용 가능
```

```

bool can_use_special_item() {
    return std::any_of(users.begin(), users.end(),
                       [](User& user) { return user.level >= 19; });
}
int main() {
    Party party;
    party.add_user("철수", 15);
    party.add_user("영희", 18);
    party.add_user("민수", 12);
    party.add_user("수빈", 19);

    std::cout << std::boolalpha;
    std::cout << "던전 입장 가능 ? " << party.can_join_dungeon() << std::endl;
    std::cout << "특별 아이템 사용 가능 ? " << party.can_use_special_item()
        << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

던전 입장 가능 ? false
특별 아이템 사용 가능 ? true

```

와 같이 나옵니다.

마지막으로 살펴볼 함수들은 `any_of` 와 `all_of` 입니다. `any_of` 는 인자로 받은 범위안의 모든 원소들 중에서 조건을 하나라도 충족하는 것이 있다면 `true` 를 리턴하고 `all_of` 의 경우 모든 원소들이 전부 조건을 충족해야 `true` 를 리턴합니다. 즉 `any_of` 는 OR 연산과 비슷하고 `all_of` 는 AND 연산과 비슷하다고 볼 수 있지요.

```

bool add_user(std::string name, int level) {
    User new_user(name, level);
    if (std::find(users.begin(), users.end(), new_user) != users.end()) {
        return false;
    }
    users.push_back(new_user);
    return true;
}

```

먼저 간단히 유저들의 정보를 담고 있는 `User` 구조체를 정의하였고, 그 `User` 들이 파티를 이룰 때 만들어지는 `Party` 클래스를 정의하였습니다. 그리고 위 `add_user` 함수를 사용하면 파티원을 추가할 수 있지요. 물론 중복되는 파티원이 없도록 벡터에 원소를 추가하기 전에 확인합니다.

```
// 파티원 모두가 15 레벨 이상이여야지 던전 입장 가능
bool can_join_dungeon() {
    return std::all_of(users.begin(), users.end(),
                       [](User& user) { return user.level >= 15; });
}
```

따라서 이 파티가 어떤 던전에 참가하고 싶은 경우 모든 파티원의 레벨이 15 이상이어야 하므로 위와 같이 `all_of` 함수를 사용해서 모든 원소들이 조건에 만족하는지 확인할 수 있습니다. 위 경우 민수가 12 레벨이여서 `false` 가 리턴되겠지요.

```
// 파티원 중 한명이라도 19렙 이상이면 특별 아이템 사용 가능
bool can_use_special_item() {
    return std::any_of(users.begin(), users.end(),
                       [](User& user) { return user.level >= 19; });
}
```

비슷하게도 한 명만 조건을 만족해도 되는 경우 위와 같이 `any_of` 함수를 사용하면 간단히 처리할 수 있습니다.

자 그러면 이번 강좌는 여기서 마치도록 하겠습니다. 사실 알고리즘 라이브러리를 살펴보면 이것 보다도 훨씬 많은 수의 여러가지 유용한 함수들이 정의되어 있습니다. 하지만 이 모든 함수들을 강좌에서 다루기에는 조금 무리가 있고, 이 정도 함수들만 알아놓아도 매우 편리하게 사용하실 수 있을 것이라 생각합니다!

C++ 문자열의 모든 것 (string과 string_view)

안녕하세요 여러분! 지난 강좌들에서 STL의 컨테이너들과 알고리즘 라이브러리에 대해 다루었습니다. 이번 강좌에서는 C++의 표준 문자열 라이브러리인 `<string>`에 대해 조금 더 자세히 알아보도록 할 것입니다.

basic_string

이전에 6-1 강에서 `string` 클래스에 대해 간단히 소개드린적이 있습니다. 그 때 간단히 `std::string`의 사용법을 짚고 갔었는데요, 이번 강좌에서는 좀 더 자세히 파헤쳐보겠습니다.

`std::string`은 사실 `basic_string`이라는 클래스 템플릿의 인스턴스화 버전입니다. `basic_string`의 정의를 살펴보면 아래와 같습니다.

```
template <class CharT, class Traits = std::char_traits<CharT>,
          class Allocator = std::allocator<CharT> >
class basic_string;
```

`basic_string`은 `CharT` 타입의 객체들을 메모리에 연속적으로 저장하고, 여러가지 문자열 연산들을 지원해주는 클래스입니다. 만약에 `CharT` 자리에 `char`이 오게 된다면, 우리가 생각하는 `std::string`이 되는 것이죠. 사실 우리가 아는 `string` 말고도 총 5 가지 종류의 인스턴스화된 문자열들이 있는데;

타입	정의	비고
<code>std::string</code>	<code>std::basic_string<char></code>	
<code>std::wstring</code>	<code>std::basic_string<wchar_t></code>	<code>wchar_t</code> 의 크기는 시스템마다 다름. 윈도우에서는 2 바이트이고, 유닉스 시스템에서는 4 바이트
<code>std::u8string</code>	<code>std::basic_string<char8_t></code>	C++ 20에 새로 추가되었음; <code>char8_t</code> 는 1 바이트; UTF-8 문자열을 보관할 수 있음
<code>std::u16string</code>	<code>std::basic_string<char16_t></code>	<code>char16_t</code> 는 2 바이트; UTF-16 문자열을 보관할 수 있음
<code>std::u32string</code>	<code>std::basic_string<char32_t></code>	<code>char32_t</code> 는 4 바이트; UTF-32 문자열을 보관할 수 있음

와 같이 있습니다. UTF-8이나 UTF-16이 뭔지에 관해서는 아래에 좀더 설명할 것이니 여기서는 대충 저런들이 있구나 하고 넘어가면 됩니다.²⁾

그렇다면 나머지 템플릿 인자들을 살펴봅시다. Traits는 주어진 CharT 문자들에 대해 기본적인 문자열 연산들을 정의해놓은 클래스를 의미합니다. 여기서 기본적인 문자열 연산들이란, 주어진 문자열의 대소 비교를 어떻게 할 것인지, 주어진 문자열의 길이를 어떻게 쟈울 것인지 등을 말합니다.

다시 말해 `basic_string`에 정의된 문자열 연산들은 사실 전부다 Traits의 기본적인 문자열 연산들을 가지고 정의되어 있습니다. 덕분에 문자열들을 어떻게 보관하는지에 대한 로직과 문자열들을 어떻게 연산하는지에 대한 로직을 분리시킬 수 있었지요. 전자는 `basic_string`에서 해결하고, 후자는 Traits에서 담당하게 됩니다.

이렇게 로직을 분리한 이유는 `basic_string`의 사용자에게 좀더 자유를 부여하기 위해서입니다. 예를 들어서 `string`처럼 `char`이 기본 타입인 문자열에서, 문자열 비교시 대소 문자 구분을 하지 않는 버전을 만들고 싶다고 해봅시다.

그렇다면 그냥 처음부터 Traits에서 문자열들을 비교하는 부분만 살짝 바꿔주면 됩니다. 만일 Traits가 없었다면 `basic_string`에서 문자열을 비교하는 부분을 일일히 찾아서 고쳐야겠지요. 여기에서 예시 코드를 볼 수 있습니다.

Traits에는 `<string>`에 정의된 `std::char_traits` 클래스의 인스턴스화 버전을 전달합니다. 예를 들어서 `string`의 경우 `char_traits<char>`을 사용하게 됩니다.³⁾

숫자들의 순위가 알파벳 보다 낮은 문자열

Traits가 어떻게 활용되는지 좀더 자세히 살펴보기 위해, 직접 Traits 클래스를 오버로딩해서 문자열 비교시의 숫자들의 순위가 제일로 낮은 문자열을 만들어보겠습니다. 이게 무슨 말이냐면 원래 아스키 테이블에서 숫자들의 값이 알파벳 보다 작아서 더 앞에 오게 됩니다. 즉, `1a`가 `a1`보다 앞에 온다는 것이지요.

하지만 이를 바꿔서 숫자들이 다른 문자들 보다 우선순위가 낮은 문자열을 한 번 만들어보겠습니다.

```
#include <cctype>
#include <iostream>
#include <string>

// char_traits 의 모든 함수들은 static 함수입니다.
struct my_char_traits : public std::char_traits<char> {
    static int get_real_rank(char c) {
        // 숫자면 순위를 엄청 떨어트린다.
        if (isdigit(c)) {
            return c + 256;
        }
    }
}
```

2) 보시다시피, `wchar_t`의 크기가 시스템마다 다루기 때문에 확실한 2바이트 타입과 4바이트 타입을 만들기 위해 `char16_t`와 `char32_t`가 탄생했습니다.

3) 할당자(Allocator)에 대해서는 추후에 다른 강좌로 설명하도록 하겠습니다.

```

    }
    return c;
}

static bool lt(char c1, char c2) {
    return get_real_rank(c1) < get_real_rank(c2);
}

static int compare(const char* s1, const char* s2, size_t n) {
    while (n-- != 0) {
        if (get_real_rank(*s1) < get_real_rank(*s2)) {
            return -1;
        }
        if (get_real_rank(*s1) > get_real_rank(*s2)) {
            return 1;
        }
        ++s1;
        ++s2;
    }
    return 0;
};

int main() {
    std::basic_string<char, my_char_traits> my_s1 = "1a";
    std::basic_string<char, my_char_traits> my_s2 = "a1";

    std::cout << "숫자의 우선순위가 더 낮은 문자열 : " << std::boolalpha
        << (my_s1 < my_s2) << std::endl;

    std::string s1 = "1a";
    std::string s2 = "a1";

    std::cout << "일반 문자열 : " << std::boolalpha << (s1 < s2) << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

숫자의 우선순위가 더 낮은 문자열 : false
일반 문자열 : true

```

와 같이 잘 나옵니다.

```
struct my_char_traits : public std::char_traits<char> {
```

`basic_string` 의 Traits 에는 `char_traits` 에서 제공하는 모든 멤버 함수들이 구현된 클래스가 전달되어야 합니다. (꼭 `char_traits` 를 상속 받을 필요는 없습니다) 이를 가장 간편히 만들기 위해서는 그냥 `char_traits` 를 상속 받은 후, 필요한 부분만 새로 구현하면 됩니다.

`char_traits` 에 정의되는 함수들은 모두 `static` 함수들입니다. 그 이유는 `char_traits` 의 존재 이유를 생각해보면 당연한데, Traits 는 문자와 문자열들 간에 간단한 연산을 제공해주는 클래스이므로 굳이 데이터를 저장할 필요가 없기 때문입니다. (이를 보통 **Stateless** 하다고 합니다.)

일반적인 `char` 들을 다루는 `char_traits<char>` 에서 우리가 바꿔줘야 할 부분은 대소 비교하는 부분 뿐입니다. 따라서 아래와 같이 문자들 간의 크기를 비교하는 `lt` 함수와 길이 `n` 의 문자열의 크기를 비교하는 `compare` 함수를 새로 정의해줘야 했습니다.

```
static bool lt(char c1, char c2) {
    return get_real_rank(c1) < get_real_rank(c2);
}

static int compare(const char* s1, const char* s2, size_t n) {
    while (n-- != 0) {
        if (get_real_rank(*s1) < get_real_rank(*s2)) {
            return -1;
        }
        if (get_real_rank(*s1) > get_real_rank(*s2)) {
            return 1;
        }
        ++s1;
        ++s2;
    }
    return 0;
}
```

코드를 읽어보면 그다지 어렵지 않습니다. `get_real_rank` 함수는 문자를 받아서 숫자면 256 을 더해서 순위를 매우 떨어뜨립니다. 따라서 숫자들이 모든 문자들 뒤에 오게 되겠지요.

```
std::cout << "숫자의 우선순위가 더 낮은 문자열 : " << std::boolalpha
      << (my_s1 < my_s2) << std::endl;
```

따라서 실제로 `my_s1` 이 `my_s2` 보다 뒤에 온다고 나타나게 됩니다. (`my_s1 > my_s2`) 반면에 보통의 `string` 의 경우에는 `s1` 이 `s2` 앞에 나오겠지요.

이와 같이 간단히 Traits 만 바꿔주는 것으로 좀더 커스터마이징 된 `basic_string` 을 사용하실 수 있습니다.

짧은 문자열 최적화 (SSO)

이전에도 이야기 했지만, 메모리를 할당하는 작업은 시간을 꽤나 잡아먹습니다.

`basic_string` 이 저장하는 문자열의 길이는 천차 만별입니다. 때론 한 두 문자 정도의 짧은 문자열을 저장할 때도 있고, 수십만 바이트의 거대한 문자열을 저장할 때도 있습니다. 문제는 거대한 문자열은 매우 드물게 저장되는데 반해 길이가 짧은 문자열들은 굉장히 많이 생성되고 소멸 된다는 점입니다. 만일 매번 모든 문자열을 동적으로 메모리를 할당 받는다고 해봅시다. 길이가 짧은 문자열을 여러번 할당한다면 매번 메모리 할당이 이루어져야 하므로, 굉장히 비효율적일 것입니다.

따라서 `basic_string` 의 제작자들은 짧은 길이 문자열의 경우 따로 문자 데이터를 위한 메모리를 할당하는 대신에 그냥 객체 자체에 저장해버립니다. 이를 짧은 문자열 최적화(SSO - short string optimization) 이라고 부릅니다.

```
#include <iostream>
#include <string>

// 이와 같이 new 를 전역 함수로 정의하면 모든 new 연산자를 오버로딩 해버린다.
// (어떤 클래스의 멤버 함수로 정의하면 해당 클래스의 new 만 오버로딩됨)
void* operator new(std::size_t count) {
    std::cout << count << " bytes 할당 " << std::endl;
    return malloc(count);
}

int main() {
    std::cout << "s1 생성 --- " << std::endl;
    std::string s1 = "this is a pretty long sentence!!!";
    std::cout << "s1 크기 : " << sizeof(s1) << std::endl;

    std::cout << "s2 생성 --- " << std::endl;
    std::string s2 = "short sentence";
    std::cout << "s2 크기 : " << sizeof(s2) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
s1 생성 ---
34 bytes 할당
s1 크기 : 32
s2 생성 ---
s2 크기 : 32
```

위와 같이 나옵니다.

```
// 이와 같이 new 를 전역 함수로 정의하면 모든 new 연산자를 오버로딩 해버린다.
// (어떤 클래스의 멤버 함수로 정의하면 해당 클래스의 new 만 오버로딩됨)
void* operator new(std::size_t count) {
    std::cout << count << " bytes 할당 " << std::endl;
    return malloc(count);
}
```

먼저 메모리가 할당되는지 안되는지 확인하기 위해서 위와 같이 새로 new 연산자를 정의해겠습니다. 참고로 new 의 경우 위와 같이 클래스 외부의 함수로 정의하게 된다면 모든 new 연산자들이 위 함수를 사용하게 됩니다. 반면에 클래스 내에 멤버 함수로 new 를 정의하게 된다면, 해당 객체를 new 로 생성할 때 해당 new 함수가 호출됩니다.

아무튼 위와 같이 operator new 를 정의한 덕분에 basic_string 내부를 바꾸지 않고도 문자열 생성 시에 메모리 할당이 일어나는지 아닌지 관찰할 수 있습니다.

그리고 그 결과는 위와 같습니다. 길이가 긴 문자열 s1 을 생성할 때에는 메모리 할당이 발생하였고, 길이가 짧은 문자열인 s2 의 경우에는 메모리 할당이 발생하지 않았습니다.

그 대신 문자열 객체의 크기를 확인하였을 때 32 바이트로 꽉나 큅니다. 만일 정말 단순하게 문자열 라이브러리를 구현하였다면 문자열 길이를 저장할 변수 하나, 할당한 메모리 공간 크기 저장을 위한 변수 하나, 메모리 포인터 하나로 해소 총 12 바이트로 만들 수도 있을 것입니다. 하지만 라이브러리 제작자들은 메모리 사용량을 조금 희생한 대신 성능 향상을 꾀했습니다.

물론 라이브러리 마다 어느 길이 문자열 부터 바로 메모리 할당을 할 지는 다릅니다. 하지만 대부분의 주류 C++ 라이브러리 (gcc 의 libstdc++ 과 clang 의 libc++) 들은 어떤 방식이든 SSO 를 사용하고 있습니다.

여담으로, C++ 11 이전에 basic_string 의 구현에서는 **Copy On Write** 라는 기법도 사용되었습니다. 이는 문자열을 복사할 때, 바로 복사하는 것이 아니라, 복사된 문자열이 바뀔 때 비로소 복사를 수행하는 방식이었죠. 하지만 이는 C++ 11 에서 개정된 표준에 따라 불가능해졌습니다.

문자열 리터럴 정의하기

C 에서 문자열 리터럴을 정의하기 위해선 아래와 같이 하였습니다.

```
const char* s = "hello";
// 혹은
char s[] = "hello";
```

그렇다면 위 두 s 모두 "hello" 라는 문자열을 보관하게 됩니다.

C++ 의 경우는 어떨까요? 만약에

```
auto str = "hello"
```

를 하면 `str` 는 `string` 으로 정의될까요? 아닙니다. C++ 에서는 C 와 마찬가지로 `str` 의 타입은 `const char *` 로 정의됩니다. 이는 C 를 배우지 않고 C++ 부터 배우신 분들에게는 혼란스러울 여지가 있습니다. 만일 문자열을 꼭 만들어야겠다 한다면

```
string str = "hello"
```

위 처럼 타입을 꼭 명시해줘야 겠죠. 하지만 C++ 14 에 이 문제를 깜찍하게 해결할 수 있는 방법이 나왔습니다.

리터럴 연산자

재미있게도, C++ 14 에서 리터럴 연산자(literal operator) 라는 것이 새로 추가되었습니다.

```
auto str = "hello"s;
```

위와 같이 "" 뒤에 `s` 를 붙여주면 `auto` 가 `string` 으로 추론됩니다. 참고로 이 리터럴 연산자는

```
std::string operator"" s(const char *str, std::size_t len);
```

위 처럼 정의되어 있는데, `"hello"s` 는 컴파일 과정에서 `operator""s("hello", 5);` 로 바뀌게 됩니다. 참고로 해당 리터럴 연산자를 사용하기 위해서는 `std::string_literals` 네임 스페이스를 사용해야 합니다. 아래 코드를 보시죠

```
#include <iostream>
#include <string>
using namespace std::literals;

int main() {
    auto s1 = "hello"s;
    std::cout << "s1 길이 : " << s1.size() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
s1 길이 : 5
```

와 같이 제대로 `string` 으로 `auto` 가 추론된 것을 확인할 수 있습니다.

리터럴 연산자는 위처럼 문자열 리터럴만 가능한 것이 아니라 정수나 부동 소수점 리터럴들 역시 사용 가능합니다. 자세한 예시는 [여기](#)를 살펴보세요.

그 외의 여러가지 리터럴 정의 방법

사실 C++ 에는 "" 말고도 문자열 리터럴을 정의하는 몇 가지 방법이 더 있습니다.

```
std::string str = "hello";      // char[]
std::wstring wstr = L"hello";   // wchar_t[]
```

일단 그냥 "hello" 를 했다면 여러분이 생각한 대로 `char` 배열을 생성하게 됩니다. 하지만 `wchar_t` 문자열을 만들고 싶다면 앞에 그냥 L 을 붙여주면 됩니다. 그러면 컴파일러가 알아서 `wchar_t` 배열을 만들어줍니다. 그 외에도 몇 가지가 더 있습니다. 자세한 내용은 [여기](#) 를 참조해주세요.

C++ 11 에 추가된 유용한 기능으로 **Raw string literal** 이라는 것이 생겼습니다. 아래 코드를 보시지요.

```
#include <iostream>
#include <string>

int main() {
    std::string str = R"(asdfasdf
이 안에는
어떤 것들이 와도
// 이런것도 되고
#define hasldfjalskdfj
\n\n<--- Escape 안해도 됨
)";

    std::cout << str;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
asdfasdf
이 안에는
어떤 것들이 와도
// 이런것도 되고
#define hasldfjalskdfj
\n\n <--- Escape 안해도 됨
```

와 같이 나옵니다. R"()" 안에 오는 문자들은 모두 문자 그대로 `char` 배열 안에 들어가게 됩니다. 예를 들어서 이전에 "" 안에 \ 를 입력하기 위해서는 \\ 와 같이 써야 하고, " 를 입력하려면 \" 와 같이 해야 했지만, 위 경우 \ 을 넣으려면 그냥 \ 를 쓰고 " 을 넣으려면 그냥 " 를 쓰면 됩니다. 출력 결과를 보면 개행 문자 역시 그대로 잘 들어갔음을 알 수 있습니다.

다만 한 가지 문제는 닫는 괄호)" 를 문자열 안에 넣을 수 없다는 점입니다. 하지만 이는 구분 문자를 추가함으로써 해결할 수 있습니다.

```
#include <iostream>
#include <string>

int main() {
    std::string str = R"foo(
)"; <-- 무시됨
)foo";

    std::cout << str;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
); <-- 무시됨
```

와 같이 잘 나옵니다. Raw string 문법을 정확히 살펴보자면

```
R"/* delimiter */( /* 문자열 */ )/* delimiter */"
```

꼴로 쓰면 됩니다. `delimiter` 자리는 아무것도 없어도 되고, 위처럼 여러분이 원하는 문자열이 와도 되는데, 앞의 `delimiter` 와 뒤의 `delimiter` 는 같아야 합니다. 문법이 복잡하다고 느끼신다면 그냥 "delimiter(가 하나의 괄호라고 생각하시면 됩니다.

C++에서 한글 다루기

아무래도 이 글을 읽으시는 분들은 한국사람일 테니, 한글로 된 문자열을 다룰 일이 매우 많을 것이라 생각합니다. 하지만 한글을 다루는 일은 생각보다 복잡합니다.

처음에 컴퓨터가 만들어졌을 때, 대부분 영미권 국가에서 사용하였기 때문에 문자를 표현하는데 1 바이트 (= 255 개) 로도 충분하였습니다. 하지만 점차 전세계적으로 사용이 확대되면서 세계 각국의 문자를 나타내는 데에는 한계를 느끼게 되었죠.

이에 전세계 모든 문자들을 컴퓨터로 표현할 수 있도록 설계된 표준이 바로 **유니코드(Unicode)**입니다. 유니코드는 모든 문자들에 고유의 값을 부여하게 됩니다.⁴⁾ 예를 들어서 한글의 가는 0xAC00 의 값을 부여 받았고, 그 다음에 오는 문자가 각으로 0xAC01 이 됩니다. 참고로 0 부터 0x7F 까지는 기존에 사용되던 아스키 테이블과 호환을 위해 동일합니다. 즉, 영어 알파벳 A 의 경우 그대로 0x41 입니다.

현재 유니코드에 등록되어 있는 문자들의 개수는 대략 14 만 개 정도 되므로, 문자 하나를 한 개의 자료형에 보관하기 위해서는 최소 `int` 를 사용해야 합니다. (1 바이트나 2 바이트로는 불가)⁵⁾ 하지만 모든 문자들을 4 바이트 씩 지정해서 표현하는 것은 매우 비효율적입니다. 왜냐하면, 예를 들어 전체 텍스타가 모두 영어라면, 어차피 영문자는 값의 범위가 0 부터 127 사이 이므로 1 바이트 문자만 사용해도 전부 표현할 수 있기 때문이죠.

그래서 등장한 것이 바로 **인코딩(Encoding)** 방식입니다. 인코딩 방식에 따라 컴퓨터에서 문자를 표현하기 위해 동일하게 4 바이트를 사용하는 대신에, 어떤 문자는 1 바이트, 어떤 건 2 바이트 등등의 길이로 저장하게 됩니다. 유니코드에서는 아래와 같이 3 가지 형식의 인코딩 방식을 지원하고 있습니다.

- UTF-8 : 문자를 최소 1 부터 최대 4 바이트로 표현한다. (즉 문자마다 길이가 다르다!)
- UTF-16 : 문자를 2 혹은 4 바이트로 표현한다.
- UTF-32 : 문자를 4 바이트로 표현한다.

UTF-32의 경우 모든 문자들을 4 바이트로 할당하기 때문에 다루기가 매우 간단합니다. 예를 들어서 아래 코드를 살펴봅시다.

```
#include <iostream>
#include <string>

int main() {
    //           1234567890 123 4 567
    std::u32string u32_str = U"이건 UTF-32 문자열입니다";
```

4) 요즘에는 이모지도 쓸 수 있습니다

5) 물론 3 바이트로 되지 않느냐라고 물을 수 있는데 컴퓨터에서는 3 바이트 자료형이 없습니다.

```
    std::cout << u32_str.size() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

17

과 같이 나옵니다.

```
std::u32string u32_str = U"이건 UTF-32 문자열입니다";
```

u32string 은 C++ 에서 UTF-32 로 인코딩 된 문자열을 보관하는 타입이고, "" 앞에 붙은 U 는 해당 문자열 리터럴이 UTF-32 로 인코딩 하라는 의미입니다. 앞서 말했듯이 UTF-32 는 모든 문자들을 동일하게 4 바이트로 나타내기 때문에 문자열의 원소 개수와 실제 문자열의 크기가 일치 합니다.

실제로도 u32_str.size() 를 했을 때 출력한 결과와 문자열의 실제 길이가 일치함을 알 수 있습니다.

UTF-8 인코딩

UTF-32 방식의 인코딩은 다루기에 직관적이기는 하지만 자주 사용되는 인코딩 방식은 아닙니다. 왜냐하면 모든 문자에 4 바이트 씩 할당하는 것이 매우 비효율적이기 때문이죠. 그렇다면 현재 웹 상에서 많이 사용되는 **UTF-8** 인코딩 방식은 어떤지 살펴봅시다.

```
#include <iostream>
#include <string>

int main() {
    //          12 345678901 2 3456
    std::string str = u8"이건 UTF-8 문자열입니다";
    std::cout << str.size() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

32

와 같이 나옵니다. 먼저 UTF-8 형식의 문자열을 만들기 위해서는

```
std::string str = u8"이건 UTF-8 문자열입니다";
```

와 같이 "" 앞에 u8 을 써주면 됩니다. 그리고 대부분의 시스템의 경우 굳이 u8 을 안붙여도 파일의 형식이 UTF-8 일 것이므로 알아서 UTF-8 문자열이 될 것입니다.

문제는 위 프로그램 결과 입니다. 무언가 이상하죠?

분명히 문자열의 길이는 16 인데, 실제 출력된 것은 32 가 나왔습니다. 이는 UTF-8 인코딩 방식이 문자들에 최소 1 바이트 부터 최대 4 바이트 까지 지정하기 때문입니다. 일단 최소 단위가 1 바이트 이므로, UTF-8 인코딩 방식의 문자열은 char 원소들로 보관하는데, 어떤 문자는 char 1 개 만으로 충분하고, 어떤 원소는 char 원소 2 개, 3 개, 아니면 4 개 까지 필요로 하게 됩니다.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxx			
2	11	U+0080	U+07FF	110xxxx	10xxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxx	10xxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxx	10xxxxx	10xxxxx

위키피디아 UTF-8 설명 페이지에서 가져왔습니다.

위 표는 유니코드 별로 어떻게 UTF-8 로 인코딩 되는지 설명하고 있습니다. 예를 들어서 0 부터 0x7F 까지의 문자들은 1 바이트, 그 다음 0x80 부터 0x7FF 까지 문자들은 2 바이트, 0x800 부터 0xFFFF 까지는 3 바이트, 그리고 나머지가 4 바이트로 지정됩니다. 한글의 경우 0xAC00 부터 0xD7AF 까지 걸쳐 있으므로 전부 3 바이트로 표현됩니다.

이건 UTF-8 문자열 입니다 에는 한글이 8 개 있고, 영어 알파벳, 공백 문자, - 가 8 개 있습니다. 한글은 3 바이트, 나머지 애들은 1 바이트로 표현되므로 총 $3 \times 8 + 1 \times 8 = 32$, 총 32 개의 char 이 필요 합니다.

std::string 은 문자열이 어떤 인코딩으로 이루어져 있는지 관심이 없습니다. 그냥 단순하게 char 의 나열로 이루어져 있다고 생각합니다. 따라서 str.size() 를 했을 때, 문자열의 실제 길이가 아니라 그냥 char 이 몇 개가 있는지 알려줍니다. 따라서 위와 같이 32 가 출력되었습니다.

문제는 string 단에서 각각의 문자를 구분하지 못하기 때문에 불편함이 이만 저만이 아니라는 점입니다. 예를 들어서 두 번째 문자('건')를 추출하고 싶다면

```
// "건" 이 나와야 할 것 같지만 실제로는 이상한 것이 나온다.
std::cout << str[1];
```

와 같이 하면 될 것 같지만 실제로는 이상한 결과가 나옵니다. 아마 출력된 것은 이 의 UTF-8 인코딩의 두 번째 바이트이고, 해당 값은 UTF-8 인코딩에서 불가능한 값입니다.

그렇다고 해서 C++ 에서 UTF-8 문자열을 분석할 수 없다는 것은 아닙니다. 아래 처럼 하나씩 차례대로 읽어나가면 됩니다.

```
#include <iostream>
#include <string>

int main() {
    // 1 234567890 1 2 34 5 6
    std::string str = u8"이건 UTF-8 문자열입니다";
    size_t i = 0;
    size_t len = 0;

    while (i < str.size()) {
        int char_size = 0;

        if ((str[i] & 0b11111000) == 0b11110000) {
            char_size = 4;
        } else if ((str[i] & 0b11110000) == 0b11100000) {
            char_size = 3;
        } else if ((str[i] & 0b11100000) == 0b11000000) {
            char_size = 2;
        } else if ((str[i] & 0b10000000) == 0b00000000) {
            char_size = 1;
        } else {
            std::cout << "이상한 문자 발견!" << std::endl;
            char_size = 1;
        }

        std::cout << str.substr(i, char_size) << std::endl;

        i += char_size;
        len++;
    }
    std::cout << "문자열의 실제 길이 : " << len << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

이

건

U

```
T  
F  
-  
8  
  
문  
자  
열  
  
입  
니  
다  
문자열의 실제 길이 : 16
```

와 같이 잘 나옵니다.

코드를 살펴보면 매우 간단합니다. 예를 들어서 첫 번째 if 문을 살펴봅시다.

```
if ((str[i] & 0b11111000) == 0b11110000)
```

앞서 위에 있는 UTF-8 인코딩 방식을 살펴보면, 4 바이트로 인코딩되는 문자들은 첫 번째 바이트가 11110xxx 꼴입니다. 11111000 과 AND 연산을 했을 때 11110000 이 나오는 비트 형태는 11110xxx 형태 밖에 없으므로 성공적으로 분류를 하고 있다고 알 수 있습니다.

나머지 조건문들도 마찬가지입니다.

```
std::cout << str.substr(i, char_size) << std::endl;
```

그리고 위처럼 문자의 시작 위치에서 `char_size` 만큼을 읽어서 인코딩 된 문자를 정확하게 출력할 수 있습니다.

물론 UTF-8 형식의 문자열을 저장했다고 해서 `basic_string` 의 정의된 연산들을 사용할 수 없는 것은 아닙니다. `size()` 를 제외한 다른 모든 연산들은 문자열의 인코딩 방식과 무관합니다. 예를 들어서 문자열에서 원하는 글자를 검색하는 것은 인코딩과 무관하게 수행할 수 있습니다.

하지만 그래도 UTF-8 문자 그대로 한글 문자열을 다루는 것은 불편합니다. 특히 영문자와 섞여 있을 경우 알파벳은 1 바이트지만 한글은 3 바이트로 해석되기 때문에 반복자를 통해서 문자들을 순차적으로 뽑아내기 힘들지요. 하지만 UTF-16 인코딩 방식을 사용하면 이야기가 달라집니다.

UTF-16 인코딩

UTF-16 인코딩은 최소 단위가 2 바이트입니다. 따라서 UTF-16 으로 인코딩 된 문자열을 저장하는 클래스인 `u16string` 도 원소의 타입이 2 바이트 (`char16_t`) 입니다.

UTF-16 은 유니코드에서 0 부터 D7FF 번 까지, 그리고 E000 부터 FFFF 까지의 문자들을 2 바이트로 인코딩 합니다. 그리고 FFFF 보다 큰 문자들은 4 바이트로 인코딩 되지요. 참고로 D800 번 부터 DFFF 사이의 문자들은 어디에 인코딩 되냐고 물을 수 있는데, 이들은 유니코드 상 존재하지 않는 문자들입니다.⁶⁾

덕분에 UTF-16 인코딩 방식으로는 대부분의 문자들이 2 바이트로 인코딩 됩니다. 알파벳, 한글, 한자 전부다 말이지요. 물론 이모지나 이집트 상형문자와 같이 유니코드 상 높은 번호로 매핑되어 있는 애들은 4 바이트로 인코딩 됩니다.

```
#include <iostream>
#include <string>

int main() {
    // 1234567890 123 4 567
    std::u16string u16_str = u"이건 UTF-16 문자열입니다";
    std::cout << u16_str.size() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

17

와 같이 나옵니다.

만일 여러분이 일반적인 문자들만 수록되어 있는 텍스트를 다루신다면 `u16string` 을 사용하는 것 만큼 좋은 것이 없습니다. 거의 대부분의 문자들이 2 바이트로 인코딩 될 것이므로, 모든 문자들이 원소 1 개 만큼씩을 사용합니다. 따라서 위처럼 문자열의 길이와 `u16_str.size()` 가 일치하겠지요.

따라서 아래와 같이 한글의 초성만 분리해내는 코드를 작성할 수 도 있습니다.

```
#include <iostream>
#include <string>

int main() {
    std::u16string u16_str = u"안녕하세요 모두에 코드에 오신 것을 환영합니다";
```

6) 사실 UTF-16 인코딩을 위해서 일부러 해당 구간에는 대응되는 문자들지 않았습니다.

```

std::string jaum[] = {"ㄱ", "ㅋ", "ㄴ", "ㄷ", "ㄸ", "ㄹ", "ㅁ",
                      "ㅂ", "ㅃ", "ㅅ", "ㅆ", "ㅇ", "ㅈ", "ㅉ",
                      "ㅊ", "ㅋ", "ㅌ", "ㅍ", "ㅎ"};
```

```

for (char16_t c : u16_str) {
    // 유니코드 상에서 한글의 범위
    if (!(0xAC00 <= c && c <= 0xD7A3)) {
        continue;
    }
    // 한글은 AC00 부터 시작해서 한 초성당 총 0x24C 개 씩 있다.
    int offset = c - 0xAC00;
    int jaum_offset = offset / 0x24C;
    std::cout << jaum[jaum_offset];
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```
ㅇㄴㅎㅅㅇㅁㄷㅇㅋㄷㅇㅇㅅㄱㅇㅎㅇㅎㄴㄷ
```

위와 같이 잘 분리되었음을 알 수 있습니다. 한글은 유니코드 상에서 한 초성 당 588 개 씩 있습니다. 예를 들어 처음에 ㄱ을 시작으로, 각, 각, 간 순으로 진행되지요. [여기](#)에서 한글이 어떻게 유니코드 상에서 등록되어 있는지 볼 수 있습니다. 머리를 좀만 쓴다면, 초성-중성-종성 분리 까지 쉽게 가능합니다.

하지만 UTF-16 역시 때론 4 바이트로 문자를 인코딩 해야 하기 때문에 i 번째 문자를 str[i] 와 같이 접근할 수 있는 것은 아닙니다. 예를 들어서;

```

#include <iostream>
#include <string>

int main() {
    std::u16string u16_str = u"뷁";
    std::cout << u16_str.size() << std::endl;
}

```

성공적으로 컴파일 하였을 경우

실행 결과

4

위와 같이 실제 문자열의 길이와 사용된 원소의 개수가 차이가 나게 됩니다.

안타깝게도 C++에서는 요즘에 나온 Go 언어처럼 인코딩 된 문자열을 언어 단에서 간단히 처리할 수 있는 방법은 없습니다. 가장 편한 방법은 그냥 어떤 문자열이든 그냥 UTF-32 인코딩으로 바꿔버리면 되겠지만, 이는 메모리 사용량을 매우 증가시킵니다.

다행으로 UTF8-CPP라는 C++에서 여러 방식으로 인코딩 된 문자열을 쉽게 다룰 수 있게 도와주는 라이브러리가 있습니다 (표준 라이브러리는 아닙니다). [여기](#)에서 사용법을 볼 수 있으며 매우 간단합니다!

string_view

자 그럼 이번에는 좀 더 다른 주제에 대해 이야기 해보도록 하겠습니다. 만일 어떤 함수에다 문자열을 전달할 때, 문자열 읽기만 필요로 한다면 보통 `const std::string&` 으로 받던지 아니면 `const char*` 형태로 받게 됩니다.

하지만 각각의 방식은 문제점이 있습니다. 먼저 `const string&` 형태로 받을 경우를 살펴봅시다.

```
#include <iostream>
#include <string>

void* operator new(std::size_t count) {
    std::cout << count << " bytes 할당 " << std::endl;
    return malloc(count);
}

// 문자열에 "very"라는 단어가 있으면 true를 리턴함
bool contains_very(const std::string& str) {
    return str.find("very") != std::string::npos;
}

int main() {
    // 암묵적으로 std::string 객체가 불필요하게 생성된다.
    std::cout << std::boolalpha << contains_very("c++ string is very easy to use")
        << std::endl;

    std::cout << contains_very("c++ string is not easy to use") << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
31 bytes 할당
true
30 bytes 할당
```

```
false
```

와 같이 나옵니다.

`contains_very` 함수는 인자로 받은 문자열에 *very*라는 단어가 있으면 `true`를 리턴하는 함수입니다. 따라서 인자를 읽기만 하므로, `const string&`의 형태로 받으면 됩니다.

문제는 `contains_very` 함수에 문자열 리터럴을 전달한다면 (이는 `const char*`), 인자는 `string`만 받을 수 있기 때문에 암묵적으로 `string` 객체가 생성된다는 점입니다. 따라서 위 출력 결과처럼 불필요한 메모리 할당이 발생한 것을 볼 수 있습니다.

그렇다면 `contains_very` 함수를 `const char*` 형태의 인자로 받도록 바꾸면 안될까요? 그렇다면 두 가지 문제가 발생합니다.

- 먼저 `string`을 함수에 직접 전달할 수 없고 `c_str` 함수를 통해 `string`에서 `const char*` 주소값을 뽑아내야 합니다.
- `const char*`로 변환하는 과정에서 문자열의 길이에 대한 정보를 읽어버리게 됩니다. 만일 함수 내부에서 문자열 길이 정보가 필요하다면 매번 다시 계산해야 합니다.

이러한 연유로, `contains_very` 함수를 합리적으로 만들기 위해서는 `const string&`을 인자로 받는 오버로딩 하나, 그리고 `const char*`을 인자로 받는 오버로딩 하나를 각각 준비해야 한다는 문제점이 있었습니다.

하지만 위와 같은 문제는 C++ 17에서 `string_view`가 도입됨으로써 해결되었습니다.

소유하지 않고 읽기만 한다!

```
#include <iostream>
#include <string>

void* operator new(std::size_t count) {
    std::cout << count << " bytes 할당 " << std::endl;
    return malloc(count);
}

// 문자열에 "very"라는 단어가 있으면 true를 리턴함
bool contains_very(std::string_view str) {
    return str.find("very") != std::string_view::npos;
}

int main() {
    // string_view 생성 시에는 메모리 할당이 필요 없다.
    std::cout << std::boolalpha << contains_very("c++ string is very easy to use")
        << std::endl;
```

```
std::cout << contains_very("c++ string is not easy to use") << std::endl;
std::string str = "some long long long long long string";
std::cout << "-----" << std::endl;
std::cout << contains_very(str) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
true
false
37 bytes 할당
-----
false
```

와 같이 나옵니다.

`string_view` 는 이름 그대로 문자열을 읽기 만 하는 클래스입니다. 이 때문에 `string_view` 는 문자열을 소유하고 있지 않습니다. 즉, 어딘가 존재하는 문자열을 참조해서 읽기만 하는 것이지요. 따라서 `string_view` 가 현재 보고 있는 문자열이 소멸된다면 정의되지 않은 작업(Undefined behavior)이 발생하게 됩니다.

주의 사항

중요해서 한 번 더 강조하지만 `string_view` 는 문자열을 소유하고 있지 않기 때문에 현재 읽고 있는 문자열이 소멸되지 않은 상태인지 주의해야 합니다.

하지만 문자열을 소유하지 않고 읽기 만 한다는 특성 때문에 `string_view` 객체 생성시에 메모리 할당이 불필요 합니다. 그냥 읽고 있는 문자열의 시작 주소값만 복사하면 되기 때문이죠. 따라서 위처럼 `string`이나 `const char*` 을 전달하더라도 메모리 할당이 발생하지 않습니다.

뿐만 아니라 `const char*` 을 인자로 받았을 때에 비해 `string` 의 경우 문자열 길이가 그대로 전달되므로 불필요한 문자열 길이 계산을 할 필요가 없습니다. 또한 `const char*` 에서 `string_view` 를 생성하면서 문자열 길이를 한 번만 계산하면 되므로 효율적입니다.

`string_view` 에서 제공하는 연산들은 당연히도 원본 문자열을 수정하지 않는 연산들입니다. 대표적으로 `find` 와 부분 문자열을 얻는 `substr` 을 들 수 있습니다. 특히 `string` 의 경우 `substr` 이 실제로 부분 문자열을 새로 생성해야 하기 때문에 $O(n)$ 으로 수행되지만, `string_view` 의 경우 `substr` 로 또다른 `view` 를 생성하므로 $O(1)$ 로 매우 빠르게 수행됩니다.

```
#include <iostream>
#include <string>
```

```

void* operator new(std::size_t count) {
    std::cout << count << " bytes 할당" << std::endl;
    return malloc(count);
}

int main() {
    std::cout << "string ----" << std::endl;
    std::string s = "sometimes string is very slow";
    std::cout << "-----" << std::endl;
    std::cout << s.substr(0, 20) << std::endl << std::endl;

    std::cout << "string_view ----" << std::endl;
    std::string_view sv = s;
    std::cout << "-----" << std::endl;
    std::cout << sv.substr(0, 20) << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

string ----
30 bytes 할당
-----
21 bytes 할당
sometimes string is

string_view ----
-----
sometimes string is

```

와 같이 나옵니다.

보시다시피, `string` 의 `substr` 은 문자열을 새로 생성하였기에 메모리 할당이 발생하였지만 `string_view` 의 경우 `substr` 시에 메모리 할당이 발생하지 않았습니다.

물론 위 `string_view` 들은 모두 `s` 에서 만들어진 것이므로 `s` 가 소멸되면 사용할 수 없게 됩니다. 예를 들어 아래 예제를 보실까요.

```

#include <iostream>
#include <string>

std::string_view return_sv() {
    std::string s = "this is a string";

```

```
    std::string_view sv = s;

    return sv;
}

int main() {
    std::string_view sv = return_sv(); // <- sv 가 가리키는 s 는 이미 소멸됨!

    // Undefined behavior!!!!
    std::cout << sv << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

a string

와 같이 이상하게 나올 것입니다.

```
std::string_view sv = return_sv(); // <- sv 가 가리키는 s 는 이미 소멸됨!
```

위 sv 는 return_sv 안에서 만들어진 s 의 string_view 이지만 함수가 리턴하면서 지역 객체였던 s 가 소멸하였기 때문에 sv 는 소멸된 문자열을 가리키는 꼴이 되었습니다.

따라서 sv 를 사용하면 위와 같이 이상한 결과가 나옵니다 (물론 프로그램을 crash 시킬 수도 있겠지요). 반드시 string_view 가 살아 있는 문자열의 view 인지를 확인하고 사용해야 합니다.

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. 이번 강좌를 통해서 C++ 에서 제공되는 string 라이브러리와 string_view 라이브러리의 강력함을 알아보셨으면 합니다.

뭘 배웠지?

- `std::string` 은 `basic_string` 의 `char` 을 인자로 갖는 템플릿 인스턴스화 버전입니다. 그 외에도 `u8string`, `u16string`, `u32string` 이 있고 각각은 UTF-8, UTF-16, UTF-32 으로 인코딩 된 문자열을 보관할 수 있습니다.
- `std::char_traits` 를 사용해서 사용자가 원하는 기능을 가진 문자열을 생성할 수 있습니다.
- 유니코드는 전세계의 모든 문자들을 컴퓨터에서 표현하고자 각각의 문자에 대해 고유의 코드를 부여한 것입니다. 코드 그대로 그냥 저장하려면 4 바이트가 필요한데, 이는 매우 비효율적이므로 여러가지 인코딩 을 통해서 크기를 줄일 수 있습니다. 하지만 이 때문에 문자별로 인코딩 되는 길이가 다르다는 문제점이 있습니다.
- `string_view` 를 통해서 불필요한 복사를 막고 `const char*` 과 `const string&` 사이에서 깔끔하게 처리할 수 있습니다.

C++에서의 예외 처리

안녕하세요 여러분! 오래 간만에 인사 드립니다. 이번 강좌에서는 C++에서 예외 처리를 어떠한 방식으로 하는지에 대해 알아보도록 하겠습니다.

예외란?

우리가 이상적인 세상에서 살고 있다면, 그 어떤 예외적인 상황도 없을 것입니다. 프로그램 혹은 라이브러리 사용자들은 언제나 올바른 값을 입력값으로 줄 것이고, 컴퓨터 역시 무한한 자원을 사용할 수 있어서 어떠한 상황에서도 데이터들을 정상적으로 처리할 수 있을 것입니다.

하지만, 안타깝게도 이 세상은 그리 녹록하지 않습니다. 사람들은 실수를 하기 마련이고, 컴퓨터 역시 언제나 프로그램에 필요한 자원을 제공할 수 있는 것이 아닙니다. 예를 들어서 아래와 같은 `vector`의 사용 예시를 살펴봅시다.

```
std::vector<int> v(3); // 크기가 3인 벡터 만들
std::cout << v.at(4); // ??
```

위 경우, 크기가 3인 `vector`를 만들었지만 4 번째 원소를 요청하고 있습니다. 위와 같은 코드는 문법상 아무 문제가 없는 코드이지만, 막상 실행하게 되면 오류가 발생하게 됩니다.

다른 예로 아래와 같이 큰 메모리를 할당하는 경우를 생각해봅시다.

```
std::vector<int> v(10000000000);
// ?
```

여러분이 사용하는 대부분의 시스템의 경우 위와 같이 큰 메모리를 할당할 수 없습니다. 따라서, 위 코드 역시 문법상 틀린 것이 없는 코드이지만, 실제로 실행해보면 오류가 발생하게 됩니다.

이렇게 정상적인 상황에서 벗어난 모든 예외적인 상황들을 예외(exception)이라고 부릅니다.

기존의 예외 처리 방식

C 언어에서는 언어 차원에서 제공하는 예외 처리 방식이라는 것이 딱히 따로 존재하지 않았습니다. 따라서 아래와 같이, 어떤 작업을 실행한 뒤에 그 결과값을 확인하는 방식으로 처리하였습니다. 예를 들어서 아래 `malloc` 으로 메모리를 동적으로 할당하는 경우를 생각해봅시다.

```
char *c = (char *)malloc(1000000000);
if (c == NULL) {
    printf("메모리 할당 오류!");
    return;
}
```

`malloc` 의 경우 메모리 할당 실패시에 `NULL` 을 리턴하므로, 위와 같이 `c` 가 `NULL` 인지 확인함으로써 예외적인 상황을 처리할 수 있었습니다.

하지만 이러한 방식으로 예외를 처리한다면, 함수가 깊어지면 깊어질 수록 꽤나 귀찮게 됩니다. 예를 들어서 아래와 같은 예시를 살펴보세요.

```
bool func1(int *addr) {
    if (func2(addr)) {
        // Do something
    }
    return false;
}
bool func2(int *addr) {
    if (func3(addr)) {
        // Do something
    }
    return false;
}
bool func3(int *addr) {
    addr = (int *)malloc(1000000000);
    if (addr == NULL) return false;
    return true;
}
int main() {
    int *addr;
    if (func1(addr)) {
        // 잘 처리됨
    } else {
        // 오류 발생
    }
}
```

위 코드의 경우 `func3` 에서 '예외가 발생할 수 있는 작업' 을 수행하는데, 만약에 예외가 발생하게 된다면 `false` 를 리턴하게 되고, 잘 처리 되었다면 `true` 를 리턴합니다.

여기까지는 좋은데, 문제는 이 `func3` 가 `func2` 에서 호출되고, 다시 `func2` 는 `func1` 에서 호출되고, `func1` 은 `main` 에서 호출된다는 점입니다. 만약에 `main` 의 입장에서 `func3` 에서 문제가 발생했을 때 이를 캐치하기 위해서는, 각각의 함수들에서 처리 결과를 모두 리턴해야 할 것입니다.

위 코드는 예외가 `func3` 에서만 발생해서 간단하였지만, 만약에 `func2` 도 어떤 다른 작업을 해서 예외를 발생시킬 수 있다면 어떻게 해야 할까요? 상당히 골치 아픈 일입니다.

하지만 다행이도 C++ 에서는 위와 같은 불편한 예외 처리 방식을 획기적으로 해결시켰습니다.

예외 발생시키기 - `throw`

C 언어에서는 예외가 발생했을 때, 다른 값을 리턴하는 것으로 예외를 처리하였지만, C++ 에서는 예외가 발생하였다는 사실을 명시적으로 나타낼 수 있습니다. 바로 `throw` 문을 사용하면 됩니다.

예를 들어서 아래와 같이 매우 간단한 `vector` 클래스를 생각해봅시다.

```
template <typename T>
class Vector {
public:
    Vector(size_t size) : size_(size) {
        data_ = new T[size_];
        for (int i = 0; i < size_; i++) {
            data_[i] = 3;
        }
    }
    const T& at(size_t index) const {
        if (index >= size_) {
            throw out_of_range("vector 의 index 가 범위를 초과하였습니다.");
        }
        return data_[index];
    }
    ~Vector() { delete[] data_; }

private:
    T* data_;
    size_t size_;
};
```

만들어진 `vector` 의 요청한 위치에 있는 원소를 리턴하는 함수인 `at` 함수를 생각해봅시다¹⁾ 인자로 전달된 `index` 가 범위 이내라면, 간단하게 `data[index]` 를 리턴하면 되겠지만, 범위 밖이라면 어떻게 해야 할까요?

1) `at` 함수는 `operator[]` 와 같이 `index` 로 전달된 위치에 있는 원소를 리턴합니다. 하지만 차이점으로, `at` 의 경우 `const` 객체를 리턴해서, 이를 변경할 수 없습니다.

문제는 `at` 함수가 `const T&` 를 리턴하기 때문에, 따로 '오류 메세지' 를 리턴할 수 없다는 점입니다. 하지만 C++ 에서는 다음과 같이 예외가 발생하였음을 명시적으로 알릴 수 있습니다.

```
// 생략 ...
const T& at(size_t index) const {
    if (index >= size) {
        // 예외를 발생시킨다!
        throw std::out_of_range("vector 의 index 가 범위를 초과하였습니다.");
    }
    return data[index];
}
// 생략 ...
}
```

먼저, 예외를 발생시키는 부분을 자세히 살펴보겠습니다.

```
throw std::out_of_range("vector 의 index 가 범위를 초과하였습니다.");
```

C++ 에는 예외를 던지고 싶다면, `throw` 로 예외로 전달하고 싶은 객체를 써주면 됩니다. 예외로 아무 객체나 던져도 상관 없지만, C++ 표준 라이브러리에는 이미 여러가지 종류의 예외들이 정의되어 있어서 이를 활용하는 것도 좋습니다. 예를 들어서, 위 경우 `out_of_range` 객체를 `throw` 합니다. C++ 표준에는 `out_of_range` 외에도 `overflow_error`, `length_error`, `runtime_error` 등등 여러가지가 정의되어 있고 표준 라이브러리에서 활용되고 있습니다.

이렇게 예외를 `throw` 하게 되면, `throw` 한 위치에서 즉시 함수가 종료되고, 예외 처리하는 부분까지 점프하게 됩니다. 따라서 `throw` 밑에 있는 모든 문장은 실행되지 않습니다. 한 가지 중요한 점은 이렇게 함수에서 예외 처리하는 부분에 도달하기 까지 함수를 빠져나가면서, `stack` 에 생성되었던 객체들을 빠짐없이 소멸시켜 준다는 점입니다. 따라서 예외가 발생하여도 사용하고 있는 자원들을 제대로 소멸시킬 수 있습니다! (소멸자만 제대로 작성하였다면)

예외 처리 하기 - try 와 catch

그렇다면 이렇게 발생한 예외를 어떻게 처리할까요?

```
#include <iostream>
#include <stdexcept>

template <typename T>
class Vector {
    public:
        Vector(size_t size) : size_(size) {
```

```

data_ = new T[size_];
for (int i = 0; i < size_; i++) {
    data_[i] = 3;
}
const T& at(size_t index) const {
    if (index >= size_) {
        throw std::out_of_range("vector 의 index 가 범위를 초과하였습니다.");
    }
    return data_[index];
}
~Vector() { delete[] data_; }

private:
T* data_;
size_t size_;
};

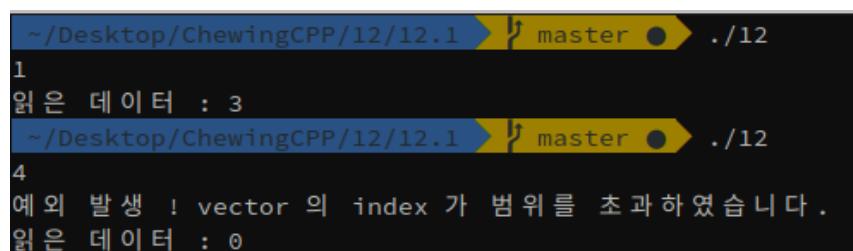
int main() {
Vector<int> vec(3);

int index, data = 0;
std::cin >> index;

try {
    data = vec.at(index);
} catch (std::out_of_range& e) {
    std::cout << "예외 발생 ! " << e.what() << std::endl;
}
// 예외가 발생하지 않았다면 3을 이 출력되고, 예외가 발생하였다면 원래 data 에
// 들어가 있던 0 이 출력된다.
std::cout << "읽은 데이터 : " << data << std::endl;
}
}

```

성공적으로 컴파일 하였다면



```

~/Desktop/ChewingCPP/12/12.1 ➤ master ➤ ./12
1
읽은 데이터 : 3
~/Desktop/ChewingCPP/12/12.1 ➤ master ➤ ./12
4
예외 발생 ! vector 의 index 가 범위를 초과하였습니다.
읽은 데이터 : 0

```

와 같이 나옵니다.

위에서도 볼 수 있듯이, 범위에 벗어난 값 (위 경우 3 이상) 을 입력하게 되었다면, 범위를 초과하였다는 메세지를 볼 수 있습니다. 그렇다면, 예외가 어떤 식으로 처리되었는지 살펴봅시다.

```

try {
}

```

```

    data = vec.at(index);
}

```

먼저 `try` 부분입니다. `try` 안에서 무언가 예외가 발생할만한 코드가 실행 됩니다. 만약에 예외가 발생하지 않았다면 마지 `try .. catch` 부분이 없는 것과 동일하게 실행 됩니다. `data`에는 `vec`의 `index` 번째 값이 들어가고 밑에 있는 `catch` 문은 무시 됩니다.

반면에 예외가 발생할 경우 이야기가 달라집니다. 예외가 발생하게되면, 그 즉시 `stack`에 생성된 모든 객체들의 소멸자들이 호출되고, 가장 가까운 `catch` 문으로 점프합니다. 따라서, 위 경우

```

if (index >= size_) {
    throw std::out_of_range("vector 의 index 가 범위를 초과하였습니다.");
}

```

의 `throw` 다음으로 실행되는 문장이 바로

```

catch (std::out_of_range& e) {
    std::cout << "예외 발생 ! " << e.what() << std::endl;
}

```

이 `catch` 부분이 됩니다. 여기서 `catch` 문은 `throw` 된 예외를 받는 부분인데, 어떤 예외를 받느냐면, `catch` 문 안에 정의된 예외의 꼴에 맞는 객체를 받게 됩니다. 우리의 `Vector`의 경우 `out_of_range`를 `throw` 하였는데, 위 `catch` 문이 `out_of_range`를 받으므로, 잘 받을 수 있습니다.

`out_of_range` 클래스는 아주 간단한데, 그냥 내부에 발생엔 예외에 관한 내용을 저장하는 문자열 필드가 달랑 하나 있고 이 역시 `what()` 함수로 그 값을 들여다 볼 수 있습니다. 위 경우 우리가 전달한 문장인 'vector의 index가 범위를 초과하였습니다' 가 나오게 됩니다.

스택 풀기 (stack unwinding)

앞서 `throw` 를 하게 된다면, 가장 가까운 `catch` 로 점프한다고 하였습니다. 이 말의 뜻이 무엇인지 아래 예제로 살펴봅시다.

```

#include <iostream>
#include <stdexcept>

class Resource {
public:
    Resource(int id) : id_(id) {}
    ~Resource() { std::cout << "리소스 해제 : " << id_ << std::endl; }
}

```

```

private:
    int id_;
};

int func3() {
    Resource r(3);
    throw std::runtime_error("Exception from 3!\n");
}

int func2() {
    Resource r(2);
    func3();
    std::cout << "실행 안됨!" << std::endl;
    return 0;
}

int func1() {
    Resource r(1);
    func2();
    std::cout << "실행 안됨!" << std::endl;
    return 0;
}

int main() {
    try {
        func1();
    } catch (std::exception& e) {
        std::cout << "Exception : " << e.what();
    }
}

```

성공적으로 실행하였으면

실행 결과

```

리소스 해제 : 3
리소스 해제 : 2
리소스 해제 : 1
Exception : Exception from 3!

```

와 같이 나옵니다.

먼저 살펴보아야 할 부분으로,

```

int func3() {
    Resource r(3);
    throw std::runtime_error("Exception from 3!\n");
}

```

에서 보시다시피, `func3` 함수에서 예외를 발생시키고 있습니다. 그런데, 이 `func3`은 `func2`가 호출하고, `func2`는 `func1`이 호출하고, 마지막으로 `func1`은 `main`에서 호출됩니다.

앞에서 말했듯이 예외가 발생하게 되면 가장 가까운 `catch`에서 예외를 받는다고 하였습니다. 그런데, `func1`, `2` 모두 예외를 받는 `catch` 구문이 없습니다. 따라서, 가장 가까운 `catch` 부분은, `main` 함수에 있는 `catch` 구문이 되고, 실제로도 예외가 `main` 함수에까지 잘 전달되어서 출력되었습니다.

또 한 가지 중요한 점은, 예외가 전파되면서 각 함수들에 정의되어 있던 객체들이 잘 소멸되었다는 점입니다.

만약에 예외가 발생하지 않았을 경우 어떻게 나오게 되냐면

```
#include <iostream>
#include <stdexcept>

class Resource {
public:
    Resource(int id) : id_(id) {}
    ~Resource() { std::cout << "리소스 해제 : " << id_ << std::endl; }

private:
    int id_;
};

int func3() {
    Resource r(3);
    return 0;
}

int func2() {
    Resource r(2);
    func3();
    std::cout << "실행!" << std::endl;
    return 0;
}

int func1() {
    Resource r(1);
    func2();
    std::cout << "실행!" << std::endl;
    return 0;
}

int main() {
    try {
        func1();
    } catch (std::exception& e) {
        std::cout << "Exception : " << e.what();
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
리소스 해제 : 3
실행!
리소스 해제 : 2
실행!
리소스 해제 : 1
```

와 같이 나옵니다.

위와 비교해보면, 정상적인 상황에서는 객체의 소멸자들은 함수가 종료될 때 호출되므로 "실행!" 이 먼저 출력되고, 그 뒤에 리소스 해제 되었다는 문장이 출력됩니다.

반면에 예외가 전파되는 과정에서는 바로 `catch` 부분으로 점프하면서, 각 함수들에 있던 객체들만 해제하기 때문에 리소스 해제 되었다는 것은 정상적으로 출력되지만, 그 "실행 안됨!" 부분은 실행되지 않습니다.

이와 같이 `catch`로 점프하면서 스택 상에서 정의된 객체들을 소멸시키는 과정을 스택 풀기(stack unwinding)이라고 부릅니다.

주의 사항

예외를 생성자에서 던질 때 주의해야 할 점이 하나 있습니다. 바로 생성자에서 예외가 발생 시에 소멸자가 호출되지 않는다라는 점입니다. 따라서, 만일 예외를 던지기 이전에 획득한 자원이 있다면 `catch`에서 잘 해제시켜 줘야만 합니다.

여러 종류의 예외 받기

앞서 `catch`는 여러 종류의 예외들을 받을 수 있다고 하였습니다. 이를 위해선, 한 개의 `try` 안에 받고자 하는 모든 종류의 예외를 `catch` 문으로 주렁 주렁 달면 됩니다. 아래 예제를 보실까요.

```
#include <iostream>
#include <string>

int func(int c) {
    if (c == 1) {
        throw 10;
    } else if (c == 2) {
        throw std::string("hi!");
    } else if (c == 3) {
        throw 'a';
    }
}
```

```

} else if (c == 4) {
    throw "hello!";
}
return 0;
}

int main() {
    int c;
    std::cin >> c;

    try {
        func(c);
    } catch (char x) {
        std::cout << "Char : " << x << std::endl;
    } catch (int x) {
        std::cout << "Int : " << x << std::endl;
    } catch (std::string& s) {
        std::cout << "String : " << s << std::endl;
    } catch (const char* s) {
        std::cout << "String Literal : " << s << std::endl;
    }
}
}

```

성공적으로 컴파일 하였다면

```

~/Desktop/ChewingCPP/12/12.1(master) $ ./12
1
Int : 10
~/Desktop/ChewingCPP/12/12.1(master) $ ./12
2
String : hi!
~/Desktop/ChewingCPP/12/12.1(master) $ ./12
3
Char : a
~/Desktop/ChewingCPP/12/12.1(master) $ ./12
4
String Literal : hello!

```

와 같이 나옵니다.

마치 switch 문처럼 catch 역시 여러 종류의 throw 된 객체를 모두 받을 수 있습니다. 위 경우,

```

catch (char x) {
    std::cout << "Char : " << x << std::endl;
}
catch (int x) {
    std::cout << "Int : " << x << std::endl;
}
catch (std::string& s) {

```

```
    std::cout << "String : " << s << std::endl;
}
catch (const char* s) {
    std::cout << "String Literal : " << s << std::endl;
}
```

첫번째 catch 문에서는 char 형 값을, 두 번째에서는 int 형 값을, 세 번째에서는 string 객체를, 마지막에서는 const char* 형 값을 받게 됩니다. 실제로도 각기 다른 값을 throw 하였을 때, 작동하는 catch 가 달라지는 것을 확인할 수 있습니다.

또한 한 가지 흥미로운 점은, 기반 클래스와 파생 클래스의 경우 처리하는 방식입니다.

```
#include <exception>
#include <iostream>

class Parent : public std::exception {
public:
    virtual const char* what() const noexcept override { return "Parent!\n"; }
};

class Child : public Parent {
public:
    const char* what() const noexcept override { return "Child!\n"; }
};

int func(int c) {
    if (c == 1) {
        throw Parent();
    } else if (c == 2) {
        throw Child();
    }
    return 0;
}

int main() {
    int c;
    std::cin >> c;

    try {
        func(c);
    } catch (Parent& p) {
        std::cout << "Parent Catch!" << std::endl;
        std::cout << p.what();
    } catch (Child& c) {
        std::cout << "Child Catch!" << std::endl;
        std::cout << c.what();
    }
}
```

성공적으로 컴파일 하였다면

```

1
Parent Catch!
Parent!
~/Desktop/CleaningCPP/12/12.1 > master ● ./12
2
Parent Catch!
Child!

```

와 같이 나옵니다.

이번에는 경우에 따라서 Parent 나 Child 클래스 객체를 리턴합니다. Parent 클래스 객체를 `throw` 하였을 때에는 예상했던대로 Parent 를 받는 `catch` 문이 실행되어서 "Parent Catch!" 가 출력되었습니다.

반면에 Child 객체를 `throw` 하였을 때에는 예상과는 다르게, Child 를 받는 `catch` 문이 아닌, Parent 를 받는 `catch` 문이 실행되어서 이 역시 "Parent Catch!" 가 출력되었습니다.

이와 같은 일이 발생한 이유는, `catch` 문의 경우 먼저 대입될 수 있는 객체를 받는데;

```
Parent& p = Child();
```

는 가능하기 때문에 Parent catch 가 먼저 받아버리는 것입니다. 따라서, 위와 같은 문제를 방지하기 위해서는 언제나 Parent catch 를 Child catch 보다 뒤에 써주는 것이 좋습니다. 왜냐하면 이를 통해서 Child 객체가 Parent catch 에 들어가는 것을 막을 수 있고,

```
Child &c = Parent(); // 오류
```

위는 성립되지 않기 때문에 Child catch 에 Parent 객체가 들어가지도 않습니다. 실제로 예를 보면;

```

#include <exception>
#include <iostream>

class Parent : public std::exception {
public:

    // what 은 std::exception 에 정의된 함수로, 이 예외가 무엇인지 설명하는 문자열을
    // 리턴하는 함수입니다.
    virtual const char* what() const noexcept override { return "Parent!\n"; }

};

class Child : public Parent {
public:

```

```

const char* what() const noexcept override { return "Child!\n"; }

};

int func(int c) {
    if (c == 1) {
        throw Parent();
    } else if (c == 2) {
        throw Child();
    }
    return 0;
}

int main() {
    int c;
    std::cin >> c;

    try {
        func(c);
    } catch (Child& c) {
        std::cout << "Child Catch!" << std::endl;
        std::cout << c.what();
    } catch (Parent& p) {
        std::cout << "Parent Catch!" << std::endl;
        std::cout << p.what();
    }
}
}

```

성공적으로 컴파일 하였다면

```

1
Parent Catch!
Parent!
~/Desktop/CleaningCPP/12/12.1 ✘ master • ./12
2
Child Catch!
Child!

```

와 같이 잘 처리됨을 알 수 있습니다.

주의 사항

일반적으로 예외 객체는 `std::exception` 을 상속 받는 것이 좋습니다. 왜냐하면 표준 라이브러리의 유용한 함수들(`nested_exception` 등) 을 사용할 수 있기 때문이지요.

모든 예외 받기

만약에 어떤 예외를 `throw` 하였는데, 이를 받는 `catch` 가 없다면 어떻게 될까요?

```
#include <iostream>
#include <stdexcept>

int func() { throw std::runtime_error("error"); }

int main() {
    try {
        func();
    } catch (int i) {
        std::cout << "Catch int : " << i;
    }
}
```

성공적으로 컴파일 하였다면

```
terminate called after throwing an instance of 'std::runtime_error'
  what(): error
[1] 21618 abort (core dumped) ./12
```

와 같이 `runtime_error` 예외를 발생시키며 프로그램이 비정상적으로 종료되었다고 뜨게 됩니다. 따라서, 언제나 예외를 던지는 코드가 있다면 적절하게 받아내는 것이 중요합니다. 하지만, 때로는 예외 객체 하나 하나 처리할 필요 없이 그냥 나머지 전부다! 라고 쓰고 싶을 때가 있습니다. 마치 `switch` 문의 `default`이나 `if-else` 문에서 마지막 `else` 와 같이 말입니다.

재미있게도 `try .. catch` 문에서도 이를 잘 지원합니다.

```
#include <iostream>
#include <stdexcept>

int func(int c) {
    if (c == 1) {
        throw 1;
    } else if (c == 2) {
        throw "hi";
    } else if (c == 3) {
        throw std::runtime_error("error");
    }
    return 0;
}

int main() {
    int c;
    std::cin >> c;
```

```

try {
    func(c);
} catch (int e) {
    std::cout << "Catch int : " << e << std::endl;
} catch (...) {
    std::cout << "Default Catch!" << std::endl;
}
}

```

성공적으로 컴파일 하였다면

```

1
Catch int : 1
~/Desktop/ChewingCPP/12/12.1 > master ● ./12
2
Default Catch!
~/Desktop/ChewingCPP/12/12.1 > master ● ./12
3
Default Catch!

```

와 같이 나옵니다.

마지막 `catch(...)`에서 `try` 안에서 발생한 모든 예외들을 받게 됩니다. 당연히도, 어떠한 예외도 다 받을 수 있기 때문에 특정한 타입을 짊어서 객체에 대입 시킬 수는 없겠지요.

주의 사항

템플릿으로 정의되는 클래스의 경우 어떠한 방식으로 템플릿이 인스턴스화 되느냐에 따라서 던지는 예외의 종류가 달라질 수 있습니다. 이 때문에 해당 객체의 `catch`에서는 모든 예외 객체를 고려해야 합니다.

예외를 발생시키지 않는 함수 - noexcept

만약에 어떤 함수가 예외를 발생시키지 않는다면 `noexcept`를 통해 명시할 수 있습니다.

```
int foo() noexcept {}
```

`foo` 함수의 경우 예외를 발생시키지 않으므로 위와 같이 함수 정의 옆에 `noexcept`를 넣음으로써 나타낼 수 있습니다. 참고로, 함수에 `noexcept` 키워드를 붙였다고 해서, 함수가 예외를 절대로 던지지 않는다는 것은 아닙니다. 실제로

```
#include <iostream>

int foo() noexcept {}

int bar(int x) noexcept { throw 1; }

int main() { foo(); }
```

이라고 해도 (경고는 뜨지만) 문제 없이 컴파일 합니다. 즉 컴파일러는 noexcept 키워드가 붙은 함수가 이 친구는 예외를 발생시키지 않는구나 라고 곧이곧대로 믿고, 그대로 컴파일하게 됩니다.

대신 noexcept로 명시된 함수가 예외를 발생시키게 된다면 예외가 제대로 처리되지 않고 프로그램이 종료됩니다. 예를 들어서

```
#include <iostream>

int foo() noexcept {}

int bar() noexcept { throw 1; }

int main() {
    foo();
    try {
        bar();
    } catch (int x) {
        std::cout << "Error : " << x << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
terminate called after throwing an instance of 'int'
[1] 4331 abort (core dumped) ./test
```

와 같이 catch 문에서 예외가 제대로 처리되지 않고 프로그램이 종료됨을 알 수 있습니다.

그렇다면 이 noexcept 키워드를 왜 붙이는 것일까요? 이는 단순히 프로그래머가 컴파일러에게 주는 힌트라고 생각하시면 됩니다. 컴파일러가 어떤 함수가 절대로 예외를 발생시키지 않는다는 사실을 안다면, 여러가지 추가적인 최적화를 수행할 수 있습니다.

주의 사항

C++ 11 에서부터 소멸자들은 기본적으로 `noexcept` 입니다. 절대로 소멸자에서 예외를 던지면 안됩니다.

자 그럼 이것으로 C++ 에서의 예외 처리에 관한 강좌를 마치도록 하겠습니다. C++ 스타일 예외 처리를 통해 좀 더 안정적인 프로그램을 만들 수 있습니다!

생각 해보기

문제 1

C++ 표준 라이브러리에 자주 사용할만한 예외 객체들이 정의가 되어있습니다. [여기](#) 를 참고해서 읽어보세요!

우측값과 이동 연산

우측값 레퍼런스와 이동 생성자

안녕하세요 여러분! 지난번 STL 강좌는 어떠셨나요? 이번 강좌에서는 C++ 11에서 추가된 우측값 레퍼런스에 대해서 다루어보도록 하겠습니다. 처음에 보면 약간 생소할 수 있는데 천천히 읽어보시기 바랍니다.

복사 생략(Copy Elision)

아래 코드를 실행해보면 결과가 어떻게 나올까요?

```
#include <iostream>

class A {
    int data_;

public:
    A(int data) : data_(data) { std::cout << "일반 생성자 호출!" << std::endl; }

    A(const A& a) : data_(a.data_) {
        std::cout << "복사 생성자 호출!" << std::endl;
    }
};

int main() {
    A a(1); // 일반 생성자 호출
    A b(a); // 복사 생성자 호출

    // 그렇다면 이것은?
    A c(A(2));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
일반 생성자 호출!
복사 생성자 호출!
일반 생성자 호출!
```

와 같이 나옵니다.

뭔가 예상했던 것과 조금 다르지요?

```
// 그렇다면 이것은?
A c(A(2));
```

이 부분에서 "일반 생성자 호출!" 한번 만 출력되었습니다. 아마 정석대로 였다면,

```
A(2)
```

를 만들면서 "일반 생성자 호출!" 이 한 번 출력되어야 되고, 생성된 임시 객체로 `c` 가 복사 생성되면서 "복사 생성자 호출!" 이 될 것이기 때문이지요. 그런데 왜 "일반 생성자 호출!" 한 번 밖에 출력되지 않았을까요? 복사 생성자가 왜 불리지 않았을까요?

사실 생각해보면 굳이 임시 객체를 한 번 만들고, 이를 복사 생성할 필요가 없습니다. 어차피 `A(2)` 로 똑같이 `c` 를 만들거면, 차라리 `c` 자체를 `A(2)` 로 만들어진 객체로 해버리는 것이랑 똑같기 때문이지요.

따라서 똑똑한 컴파일러는 복사 생성을 굳이 수행하지 않고, 만들어진 임시로 만들어진 `A(2)` 자체를 `c` 로 만들어버립니다. 이렇게, 컴파일러 자체에서 복사를 생략해 버리는 작업을 복사 생략(copy elision) 이라고 합니다.

컴파일러가 복사 생략을 하는 경우는 (함수의 인자가 아닌) 함수 내부에서 생성된 객체를 그래도 리턴할 때, 수행할 수 있습니다. 물론 C++ 표준을 읽어보면 반드시 복사 생략을 해라 라는 식이 아니라, 복사 생략을 할 수도 있다 라는 뜻으로 써 있습니다.¹⁾ 즉, 경우에 따라서는 복사 생략을 해도 되는 경우에, 복사 생략을 하지 않을 수도 있다는 뜻이지요.

이전에 만들어 놓았던 `MyString` 클래스를 다시 살펴보도록 해봅시다.

```
#include <iostream>
#include <cstring>
```

1) C++ 17 부터 일부 경우에 대해서 (예를 들어서 함수 내부에서 객체를 만들어서 return 할 경우) 반드시 복사 생략을 해야되는 것으로 바뀌었습니다. 자세한 내용은 https://en.cppreference.com/w/cpp/language/copy_elision 를 참조해주세요.

```
class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();

    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    void reserve(int size);
    MyString operator+(const MyString &s);
    ~MyString();

    int length() const;

    void print();
    void println();
};

MyString::MyString() {
    std::cout << "생성자 호출!" << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = nullptr;
}

MyString::MyString(const char *str) {
    std::cout << "생성자 호출!" << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::~MyString() { delete[] string_content; }

void MyString::reserve(int size) {
```

```
if (size > memory_capacity) {
    char *prev_string_content = string_content;

    string_content = new char[size];
    memory_capacity = size;

    for (int i = 0; i != string_length; i++)
        string_content[i] = prev_string_content[i];

    if (prev_string_content != nullptr) delete[] prev_string_content;
}
}

MyString MyString::operator+(const MyString &s) {
    MyString str;
    str.reserve(string_length + s.string_length);
    for (int i = 0; i < string_length; i++)
        str.string_content[i] = string_content[i];
    for (int i = 0; i < s.string_length; i++)
        str.string_content[string_length + i] = s.string_content[i];
    str.string_length = string_length + s.string_length;
    return str;
}

int MyString::length() const { return string_length; }

void MyString::print() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];
}

void MyString::println() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];

    std::cout << std::endl;
}

int main() {
    MyString str1("abc");
    MyString str2("def");
    std::cout << "-----" << std::endl;
    MyString str3 = str1 + str2;
    str3.println();
}
```

성공적으로 컴파일 하였다면

```
C:\WINDOWS\system32\cmd.exe
생성자 호출!
생성자 호출!
-----
생성자 호출!
복사 생성자 호출!
abcdef
Press any key to continue . . .
```

와 같이 나옵니다.

```
string_content = nullptr;
```

`nullptr` 는 C++ 11 에 새로 추가된 키워드로, 기존의 `NULL` 대체합니다.

C 언어에서의 `NULL` 은 단순히 `#define` 으로 정의되어 있는 상수값 0 인데, 이 때문에 `NULL` 이 값 0 을 의미하는 것인지, 아니면 포인터 주소값 0 을 의미하는 것인지 구분할 수가 없었습니다.

하지만 `nullptr` 로 '포인터 주소값 0' 을 정확히 명시해 준다면 미연에 발생할 실수를 줄여 줄 수 있게 됩니다.

```
MyString str3 = str1 + str2;
```

이 부분에서 두 개의 문자열을 더한 새로운 문자열로 `str3` 를 생성하고 있습니다.

```
MyString MyString::operator+(const MyString &s) {
    MyString str;
    str.reserve(string_length + s.string_length);
    for (int i = 0; i < string_length; i++)
        str.string_content[i] = string_content[i];
    for (int i = 0; i < s.string_length; i++)
        str.string_content[string_length + i] = s.string_content[i];
    str.string_length = string_length + s.string_length;
    return str;
}
```

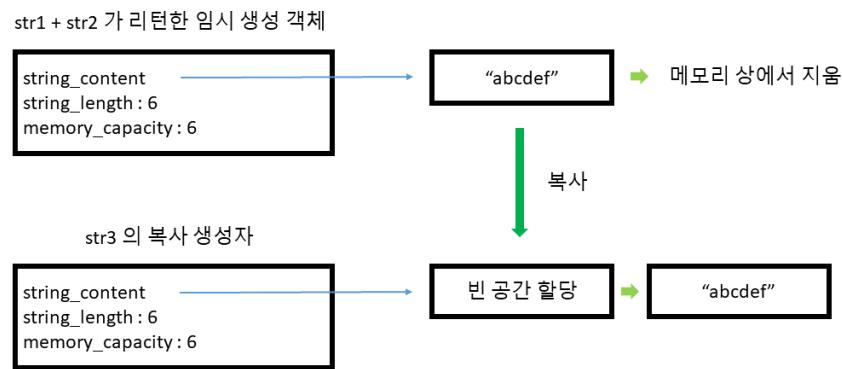
위 함수가 `str1 + str2` 를 실행 시에 호출되는데, 먼저 빈 `MyString` 객체인 `str` 을 생성합니다. (생성자 호출! 출력됨) 그 후에, `reserve` 함수를 이용해서 공간을 할당하고, `str1` 과 `str2` 를 더한 문자열을 복사하게 됩니다.

이렇게 리턴된 `str` 은 `str3` 을 생성하는데 전달되어서, `str3` 의 복사 생성자가 호출 됩니다.

하지만, 이미 예상했겠지만, 굳이 `str3` 의 복사 생성자를 또 호출할 필요가 없습니다. 왜냐하면, 어차피 똑같이 복사해서 생성할 것이면, 이미 생성된 (`str1 + str2`) 가 리턴한 객체를 `str3`

셈 치고 사용하면 되기 때문이지요. 이전의 예제에서는 컴파일러가 불필요한 복사 생성자 호출을 복사 생략을 통해 수행하지 않았지만, 이 예제의 경우, 컴파일러가 복사 생략 최적화를 수행하지 않았습니다.

위 과정을 그림으로 간단히 살펴보면 아래와 같습니다.



만약에 `str1` 과 `str2` 의 크기가 엄청 커다면 어떨까요? 쓸데 없는 복사를 두 번 하는데 상당한 자원이 소모될 것입니다.

그렇다면 이러한 문제를 C++ 에서는 어떠한 방식으로 해결하고 있을까요?

좌측값 (lvalue) 와 우측값 (rvalue)

모든 C++ 표현식 (expression) 의 경우 두 가지 카테고리로 구분할 수 있습니다. 하나는 이 구문이 어떤 타입을 가지냐이고, 다른 하나는 어떠한 종류의 '값' 을 가지냐입니다. 값에 종류가 있어? 라고 생각 하실 수 있는데, 아래 예시를 살펴보도록 합시다.

```
int a = 3;
```

위 표현식에서 먼저 '`a`' 를 살펴보도록 합시다. 우리는 `a` 가 메모리 상에서 존재하는 변수임을 알고 있습니다. 즉 '`a`' 의 주소값을 & 연산자를 통해 알아 낼 수 있다는 것입니다. 우리는 보통 이렇게 주소값을 취할 수 있는 값을 **좌측값 (lvalue)** 라고 부릅니다. 그리고 좌측값은 어떠한 표현식의 왼쪽 오른쪽 모두에 올 수 있습니다 (왼쪽에만 와야 하는게 아닙니다).

반면에 오른쪽에 있는 '`3`' 을 살펴보도록 합시다. 우리가 '`3`' 의 주소값을 취할 수 있나요? 아닙니다. '`3`' 은 왼쪽의 '`a`' 와는 다르게, 위 표현식을 연산할 때만 잠깐 존재할 뿐 위 식이 연산되고 나면 사라지는 값입니다. 즉, '`3`' 은 실체가 없는 값입니다.

이렇게, 주소값을 취할 수 없는 값을 **우측값 (rvalue)** 라고 부릅니다. 이름에도 알 수 있듯이, 우측값은 식의 오른쪽에만 항상 와야 합니다. 좌측값이 식의 왼쪽 오른쪽 모두 올 수 있는반면, 우측값은 식의 오른쪽에만 존재해야 합니다.

```
int a;           // a 는 좌측값
int& l_a = a; // l_a 는 좌측값 레퍼런스

int& r_b = 3; // 3 은 우측값. 따라서 오류
```

여태까지 우리가 다루어왔던 레퍼런스는 '좌측값'에만 레퍼런스를 가질 수 있습니다. 예를 들어서, a의 경우 좌측값이기 때문에, a의 좌측값 레퍼런스인 l_a를 만들 수 있습니다.

반면에 3의 경우 우측값이기 때문에, 우측값의 레퍼런스인 r_b를 만들 수 없습니다. 따라서 이 문장은 오류가 발생하게 됩니다.

이와 같이 & 하나를 이용해서 정의하는 레퍼런스를 좌측값 레퍼런스 (lvalue reference)라고 부르고, 좌측값 레퍼런스 자체도 좌측값이 됩니다.

그럼 다른 예제를 살펴보도록 합시다.

```
int& func1(int& a) { return a; }
int func2(int b) { return b; }

int main() {
    int a = 3;
    func1(a) = 4;
    std::cout << &func1(a) << std::endl;

    int b = 2;
    a = func2(b);           // 가능
    func2(b) = 5;           // 오류 1
    std::cout << &func2(b) << std::endl; // 오류 2
}
```

컴파일 하였다면 위 오류 1, 2, 줄에서 각각 다음과 같은 오류를 볼 수 있습니다.

컴파일 오류

```
Error C2106 '=': left operand must be l-value
Error C2102 '&' requires l-value
```

일단 func1의 경우 좌측값 레퍼런스를 리턴합니다. 앞서, 좌측값 레퍼런스의 경우 좌측값에 해당하기 때문에,

```
func1(a) = 4;
```

의 경우 'func(a)' 가 리턴하는 레퍼런스의 값을 4'로 해라 라는 의미로, 실제로 변수 a의 값이 바뀌게 됩니다. 또한, func1(a) 가 좌측값 레퍼런스를 리턴하므로, 그 리턴값의 주소값 역시 취할 수 있습니다.

하지만 func2 를 살펴볼까요? func2 의 경우, 레퍼런스가 아닌, 일반적인 int 값을 리턴하고 있습니다. 이 때 리턴되는 값은

```
a = func2(b);
```

이 문장이 실행 될 때 잠깐 존재할 뿐 그 문장 실행이 끝나면 사라지게 됩니다. 즉, 실체가 없는 값이라는 뜻이지요. 따라서 func2(b) 는 우측값이 됩니다. 따라서 위와 같이 우측값이 실제 표현식의 오른쪽에 오는 경우는 가능하지만,

```
func2(b) = 5;
```

위 문장 처럼 우측값이 왼쪽의 오는 경우는 가능하지 않습니다.

```
std::cout << &func2(b) << std::endl; // 오류 2
```

마찬가지로 우측값의 주소값을 취할 수 없기 때문에 위 문장은 허용되지 않습니다.

그렇다면 앞선 예제에서

```
MyString str3 = str1 + str2;
```

를 다시 살펴보도록 합시다. 위 문장은

```
MyString str3(str1.operator+(str2));
```

와 동일합니다. 그런데, operator+ 의 정의를 살펴보면,

```
MyString MyString::operator+(const MyString &s)
```

로 우측값을 리턴하고 있는데, 이 우측값이 어떻게 좌측값 레퍼런스를 인자로 받는,

```
MyString(const MyString &str);
```

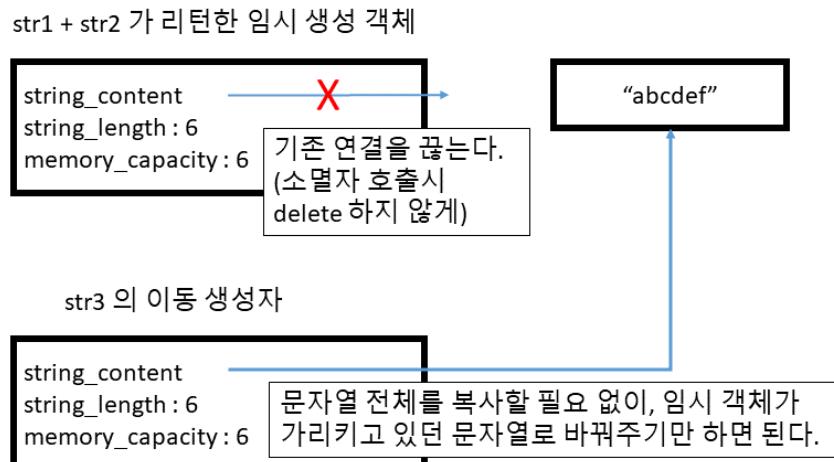
를 호출 시킬 수 있었을까요? 이는 & 가 좌측값 레퍼런스를 의미하지만, 예외적으로

const T&

의 타입의 한해서만, 우측값도 레퍼런스로 받을 수 있습니다. 그 이유는 **const** 레퍼런스 이기 때문에 임시로 존재하는 객체의 값을 참조만 할 뿐 이를 변경할 수 없기 때문입니다.

그렇다면 이동은 어떻게?

그렇다면 앞서 **MyString**에서 지적한 문제를 해결할 생성자의 경우 어떠한 방식으로 작동해야 할까요?



위와 같이 간단합니다. `str3` 생성 시에 임시로 생성된 객체의 `string_content` 가리키는 문자열의 주소값을 `str3`의 `string_content`로 해주면 됩니다.

문제는 이렇게 하게 되면, 임시 객체가 소멸 시에 `string_content`를 메모리에서 해제하게 되는데, 그렇게 되면 `str3` 가 가리키고 있던 문자열이 메모리에서 소멸되게 됩니다. 따라서 이를 방지하기 위해서는, 임시 생성된 객체의 `string_content`를 `nullptr`로 바꿔주고, 소멸자에서 `string_content` 가 `nullptr` 이면 소멸하지 않도록 해주면 됩니다. 매우 간단하지요?

하지만, 이 방법은 기존의 복사 생성자에서 사용할 수 없습니다. 왜냐하면 우리는 인자를 `const MyString&` 으로 받았기 때문에, 인자의 값을 변경할 수 없게 되지요. 즉 임시 객체의 `string_content` 값을 수정할 수 없기에 문제가 됩니다.

이와 같은 문제가 발생한 이유는 `const MyString&` 이 좌측값과 우측값 모두 받을 수 있다는 점에서 비롯되었습니다. 그렇다면, 좌측값 말고 우측값만 특이적으로 받을 수 있는 방법은 없을까요? 바로 C++ 11 부터 제공하는 우측값 레퍼런스를 이용하면 됩니다. (참고로 C++ 11 가 기본으로 설정되어 있지 않는 컴파일러는 사용 불가능 합니다. 비주얼 스튜디오 2017 버전의 경우 자동으로 사용 가능하게 설정 되어 있으니 걱정하실 필요 없습니다.)

우측값 레퍼런스

```
#include <iostream>
#include <cstring>

class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();

    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    // 이동 생성자
    MyString(MyString &&str);

    void reserve(int size);
    MyString operator+(const MyString &s);
    ~MyString();

    int length() const;

    void print();
    void println();
};

MyString::MyString() {
    std::cout << "생성자 호출!" << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = nullptr;
}

MyString::MyString(const char *str) {
    std::cout << "생성자 호출!" << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출!" << std::endl;
```

```
string_length = str.string_length;
string_content = new char[string_length];

for (int i = 0; i != string_length; i++)
    string_content[i] = str.string_content[i];
}

MyString::MyString(MyString &&str) {
    std::cout << "이동 생성자 호출 !" << std::endl;
    string_length = str.string_length;
    string_content = str.string_content;
    memory_capacity = str.memory_capacity;

    // 임시 객체 소멸 시에 메모리를 해제하지
    // 못하게 한다.
    str.string_content = nullptr;
}

MyString::~MyString() {
    if (string_content) delete[] string_content;
}

void MyString::reserve(int size) {
    if (size > memory_capacity) {
        char *prev_string_content = string_content;

        string_content = new char[size];
        memory_capacity = size;

        for (int i = 0; i != string_length; i++)
            string_content[i] = prev_string_content[i];

        if (prev_string_content != nullptr) delete[] prev_string_content;
    }
}

MyString MyString::operator+(const MyString &s) {
    MyString str;
    str.reserve(string_length + s.string_length);
    for (int i = 0; i < string_length; i++)
        str.string_content[i] = string_content[i];
    for (int i = 0; i < s.string_length; i++)
        str.string_content[string_length + i] = s.string_content[i];
    str.string_length = string_length + s.string_length;
    return str;
}

int MyString::length() const { return string_length; }

void MyString::print() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];
}

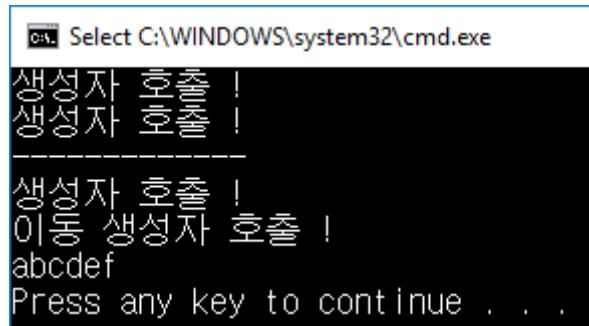
void MyString::println() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];

    std::cout << std::endl;
}
```

```
int main() {
    MyString str1("abc");
    MyString str2("def");

    std::cout << "-----" << std::endl;
    MyString str3 = str1 + str2;
    str3.println();
}
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

먼저 우측값 레퍼런스를 사용한 이동 생성자의 정의 부분부터 살펴봅시다.

```
MyString::MyString(MyString&& str) {
    std::cout << "이동 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = str.string_content;
    memory_capacity = str.memory_capacity;

    // 임시 객체 소멸 시에 메모리를 해제하지
    // 못하게 한다.
    str.string_content = nullptr;
}
```

우측값의 레퍼런스를 정의하기 위해서는 좌측값과는 달리 & 를 두 개 사용해서 정의해야 합니다. 즉, 위 생성자의 경우 `MyString` 타입의 우측값을 인자로 받고 있습니다.

그렇다면 한 가지 퀴즈! 과연 `str` 자체는 우측값 일까요? 좌측값 일까요? 당연히도 좌측값입니다. 실체가 있기 때문이지요 (`str`이라는 이름이 있잖아요). 다시 말해 `str`은 타입이 'MyString'의 우측값 레퍼런스'인 좌측값이라 보면 됩니다. 따라서 표현식의 좌측에 올 수도 있습니다. (마지막 줄처럼)

```
string_content = str.string_content;
```

이제 위와 같이 우리가 바라던 대로 임시 객체의 `string_content` 가 가리키는 메모리를 새로 생성되는 객체의 메모리로 옮겨주기만 하면 됩니다. 기존의 복사 생성자의 경우 문자열 전체를 새로 복사해야 했지만, 이동 생성자의 경우 단순히 주소값 하나만 달랑 복사해주면 끝이기 때문에 매우 간단합니다.

```
// 임시 객체 소멸 시에 메모리를 해제하지
// 못하게 한다.
str.string_content = nullptr;
```

한 가지 중요한 부분은 인자로 받은 임시 객체가 소멸되면서 자신이 가리키고 있던 문자열을 `delete` 하지 못하게 해야 합니다. 만약에 그 문자열을 지우게 된다면, 새롭게 생성된 문자열 `str3` 도 같은 메모리를 가리키고 있기 때문에 `str3` 의 문자열도 같이 사라지는 셈이 되기 때문입니다.

따라서 `str` 의 `string_content` 를 `nullptr` 로 바꿔줍니다.

```
MyString::~MyString() {
    if (string_content) delete[] string_content;
}
```

그리고 물론 소멸자 역시 바꿔줘야만 합니다. `string_content` 가 `nullptr` 가 아닐 때 예만 `delete` 를 하도록 말이죠.

일반적으로 우측값 레퍼런스는 아래와 같은 방식으로 사용할 수 있습니다.

```
int a;
int& l_a = a;
int& ll_a = 3; // 불가능

int&& r_b = 3;
int&& rr_b = a; // 불가능
```

일단 우측값 레퍼런스의 경우 반드시 우측값의 레퍼런스만 가능합니다. 따라서, `r_b` 의 경우 우측값 '3'의 레퍼런스가 될 수 있겠지만, `rr_b` 의 경우 `a` 가 좌측값이기 때문에 컴파일 되지 않습니다.

우측값 레퍼런스의 재미있는 특징으로 레퍼런스 하는 임시 객체가 소멸되지 않도록 붙들고 있는다는 점입니다. 예를 들어서,

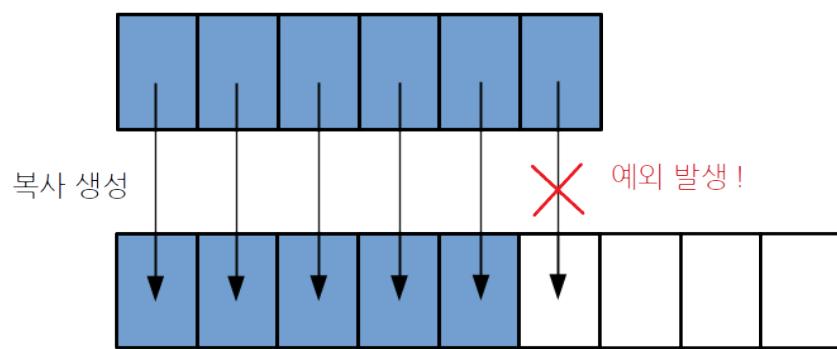
```
MyString&& str3 = str1 + str2;
str3.println();
```

의 경우 `str3` 이 `str1 + str2`에서 리턴되는 임시 객체의 레퍼런스가 되면서 그 임시 객체가 소멸되지 않도록 합니다. 실제로, 아래 `println` 함수에서 더해진 문자열이 잘 보여집니다.

이동 생성자 작성 시 주의할 점

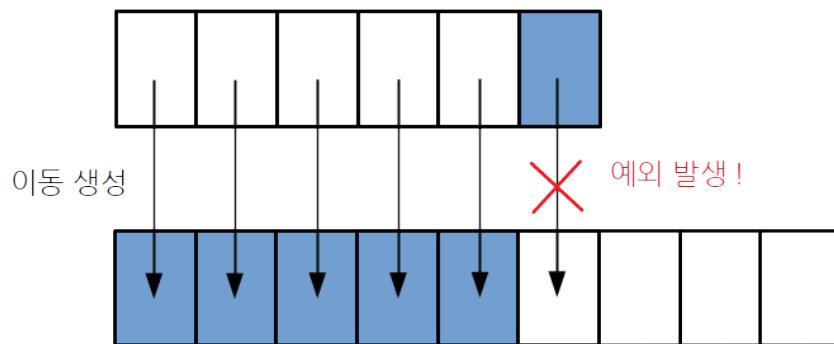
만약에 여러분이 `MyString` 을 C++ 컨테이너들, 예를 들어 `vector` 에 넣기 위해서는 한 가지 주의할 점이 있습니다. 바로 이동 생성자를 반드시 `noexcept` 로 명시해야 한다는 점입니다.

`vector` 를 예를 들어서 생각해봅시다. `vector` 는 새로운 원소를 추가 할 때, 할당해놓은 메모리가 부족하다면, 새로운 메모리를 할당한 후에, 기존에 있던 원소들을 새로운 메모리로 옮기게 됩니다.



복사 생성자를 사용 하였을 경우 위와 같이 원소가 하나씩 하나씩 복사 됩니다. 그런데 만약에 이 복사 생성하는 과정에서 예외가 발생하였다고 해봅시다.

해결책은 간단합니다. 새로 할당해놓은 메모리를 소멸시켜 버린 후, 사용자에게 예외를 전달하면 됩니다. 새로 할당한 메모리를 소멸 시켜 버리는 과정에서 이미 복사된 원소들도 소멸 되버리므로 자원이 낭비되는 일도 없을 것입니다.



반면에 이동 생성자를 사용하였을 경우는 어떨까요? 이동 생성하는 과정에서 예외가 발생했더라면, 꽤나 골치아파집니다. 복사 생성을 하였을 경우 새로 할당한 메모리를 소멸시켜 버려도, 기존의 메모리에 원소들이 존재하기 때문에 상관 없지만, 이동 생성의 경우 기존의 메모리에 원소들이 모두 이동되어서 사라져버렸기에, 새로 할당한 메모리를 셋불리 해제해버릴 수 없기 때문입니다.

따라서 `vector` 의 경우 이동 생성자에서 예외가 발생하였을 때 이를 제대로 처리할 수 없습니다. 이는 C++ 의 다른 컨테이너들도 동일합니다.

이 때문에 `vector` 는 이동 생성자가 `noexcept` 가 아닌 이상 이동 생성자를 사용하지 않습니다.

아래 실제 예제를 통해 살펴보겠습니다.

```
#include <iostream>
#include <cstring>
#include <vector>

class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length;    // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();

    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    // 이동 생성자
    MyString(MyString &&str);

    ~MyString();
};

MyString::MyString() {
    std::cout << "생성자 호출 ! " << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = nullptr;
}

MyString::MyString(const char *str) {
    std::cout << "생성자 호출 ! " << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출 ! " << std::endl;
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::MyString(MyString &&str) {
```

```

std::cout << "이동 생성자 호출 !" << std::endl;
string_length = str.string_length;
string_content = str.string_content;
memory_capacity = str.memory_capacity;

// 임시 객체 소멸 시에 메모리를 해제하지
// 못하게 한다.
str.string_content = nullptr;
}

MyString::~MyString() {
    if (string_content) delete[] string_content;
}

int main() {
    MyString s("abc");
    std::vector<MyString> vec;
    vec.resize(0);

    std::cout << "첫 번째 추가 ---" << std::endl;
    vec.push_back(s);
    std::cout << "두 번째 추가 ---" << std::endl;
    vec.push_back(s);
    std::cout << "세 번째 추가 ---" << std::endl;
    vec.push_back(s);
}

```

성공적으로 컴파일 하였다면

실행 결과

```

생성자 호출 !
첫 번째 추가 ---
복사 생성자 호출 !
두 번째 추가 ---
복사 생성자 호출 !
복사 생성자 호출 !
세 번째 추가 ---
복사 생성자 호출 !
복사 생성자 호출 !
복사 생성자 호출 !

```

위와 같이 기껏 이동 생성자를 만들어놓았는데, `vector` 가 확장할 때마다 복사 생성자를 이용하는 것을 볼 수 있습니다. 하지만 이동 생성자에 `noexcept` 를 추가하면 어떨까요.

```
#include <iostream>
```

```
#include <cstring>
#include <vector>

class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();
    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    // 이동 생성자
    MyString(MyString &&str) noexcept;

    ~MyString();
};

MyString::MyString() {
    std::cout << "생성자 호출!" << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = NULL;
}

MyString::MyString(const char *str) {
    std::cout << "생성자 호출!" << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::MyString(MyString &&str) noexcept {
    std::cout << "이동 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = str.string_content;
```

```

memory_capacity = str.memory_capacity;

// 임시 객체 소멸 시에 메모리를 해제하지
// 못하게 한다.
str.string_content = nullptr;
}

MyString::~MyString() {
    if (string_content) delete[] string_content;
}

int main() {
    MyString s("abc");
    std::vector<MyString> vec;
    vec.resize(0);

    std::cout << "첫 번째 추가 ---" << std::endl;
    vec.push_back(s);
    std::cout << "두 번째 추가 ---" << std::endl;
    vec.push_back(s);
    std::cout << "세 번째 추가 ---" << std::endl;
    vec.push_back(s);
}

```

성공적으로 컴파일 하였다면

실행 결과

```

생성자 호출 !
첫 번째 추가 ---
복사 생성자 호출 !
두 번째 추가 ---
복사 생성자 호출 !
이동 생성자 호출 !
세 번째 추가 ---
복사 생성자 호출 !
이동 생성자 호출 !
이동 생성자 호출 !

```

와 같이 제대로 이동 생성자를 호출함을 알 수 있습니다.

자 이것으로 이번 강좌는 여기서 마치도록 하겠습니다. 다음 강좌에서는 C++ 11 에 우측값 레퍼런스와 함께 새로 추가된 move 에 대해 살펴보도록 하겠습니다.

생각 해보기

문제 1

사실 C++ 에서 값의 종류로 좌측값 우측값 만이 있는게 아니라 조금 더 세부적으로 나뉘어집니다.
이에 대해 자세히 알아보고 싶으신 분들은 [여기를 참조해주세요](#)(난이도 : 상)

move 문법과 완벽한 전달

안녕하세요 여러분! 지난번의 우측값 레퍼런스 강의는 어떠셨나요? 우측값 레퍼런스를 통해서, 기존에는 불가능하였던 우측값에 대한 복사가 아닌 이동의 구현이 가능하게 되었습니다.

하지만, 만약에 좌측값도 이동을 시키고 싶다면 어떨까요? 예를 들어서 아래와 같이 두 변수의 값을 바꾸는 swap 함수를 생각해보세요.

```
template <typename T>
void my_swap(T &a, T &b) {
    T tmp(a);
    a = b;
    b = tmp;
}
```

위 my_swap 함수에서 tmp라는 임시 객체를 생성한 뒤에, b를 a에 복사하고, b에 a를 복사하게 됩니다. 문제는 무려 복사를 쓸데없이 3번이나 한다는 점입니다. 예를 들어서 T가 MyString인 경우를 생각해봅시다.

```
#include <iostream>
#include <cstring>

class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length; // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();

    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    // 이동 생성자
    MyString(MyString &&str);

    MyString &operator=(const MyString &s);
    ~MyString();

    int length() const;

    void println();
}
```

```
};

MyString::MyString() {
    std::cout << "생성자 호출!" << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = NULL;
}

MyString::MyString(const char *str) {
    std::cout << "생성자 호출!" << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::MyString(MyString &&str) {
    std::cout << "이동 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = str.string_content;
    memory_capacity = str.memory_capacity;

    // 임시 객체 소멸 시에 메모리를 해제하지
    // 못하게 한다.
    str.string_content = nullptr;
}

MyString::~MyString() {
    if (string_content) delete[] string_content;
}

MyString &MyString::operator=(const MyString &s) {
    std::cout << "복사!" << std::endl;
    if (s.string_length > memory_capacity) {
        delete[] string_content;
        string_content = new char[s.string_length];
        memory_capacity = s.string_length;
    }
    string_length = s.string_length;
    for (int i = 0; i != string_length; i++) {
        string_content[i] = s.string_content[i];
    }
}
```

```
    return *this;
}
int MyString::length() const { return string_length; }
void MyString::println() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];

    std::cout << std::endl;
}
template <typename T>
void my_swap(T &a, T &b) {
    T tmp(a);
    a = b;
    b = tmp;
}

int main() {
    MyString str1("abc");
    MyString str2("def");
    std::cout << "Swap 전 ----" << std::endl;
    str1.println();
    str2.println();

    std::cout << "Swap 후 ----" << std::endl;
    my_swap(str1, str2);
    str1.println();
    str2.println();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
생성자 호출 !
생성자 호출 !
Swap 전 -----
abc
def
Swap 후 -----
복사 생성자 호출 !
복사!
복사!
def
abc
```

와 같이 나옵니다.

```
template <typename T>
void my_swap(T &a, T &b) {
    T tmp(a);
    a = b;
    b = tmp;
}
```

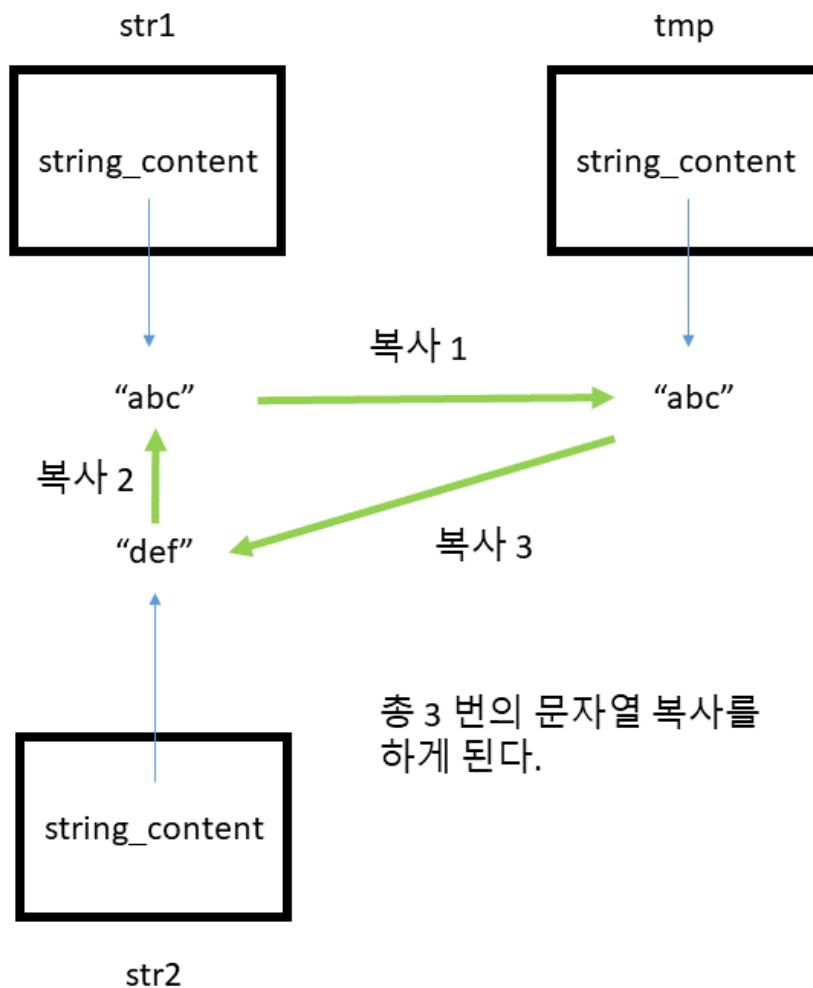
위 `my_swap` 함수를 살펴봅시다. 일단, 첫번째 줄에서, `a` 가 좌측값이기 때문에 `tmp` 의 복사 생성자가 호출됩니다. 따라서 1 차적으로 `a` 가 차지하는 공간 만큼 메모리 할당이 발생한 후 `a` 의 데이터가 복사됩니다.

```
a = b;
```

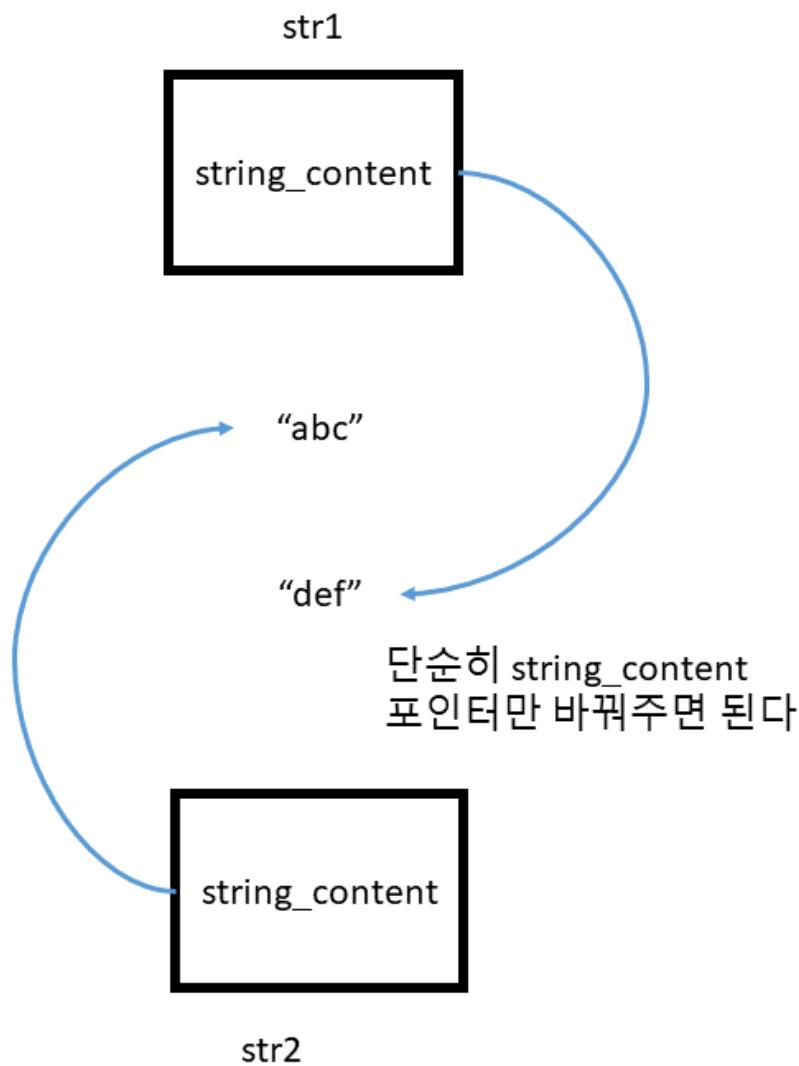
두 번째로 `a = b;` 에서 2 차적으로 복사가 발생합니다. 그리고 마지막으로,

```
b = tmp;
```

에서 또 한번 문자열 전체의 복사가 이루어지게 됩니다. 무려 `swap` 을 하기 위해 문자열 전체 복사를 3번이나 해야 합니다. 아래 그림처럼 말입니다.



하지만 우리는 굳이 문자열 내용을 복사할 필요 없이 각 `MyString` 객체의 `string_content` 주소값만 서로 바꿔주면 되는 것을 알고 있습니다. (물론 `string_length` 와 `memory_capacity`도 바꿔야겠지만, 이들은 단순히 4바이트 `int` 복사 이기 때문에 속도에 영향을 주지는 않습니다).



하지만 위를 `my_swap`에서 구현하기 위해서는 여러가지 문제가 있습니다. 일단 첫번째로 `my_swap` 함수는 임의의 타입을 받는 함수 (Generic)입니다. 다시 말해,

```

template <typename T>
void my_swap(T &a, T &b)

```

위 함수가 일반적인 타입 `T`에 대해 작동해야 한다는 의미이지요. 하지만 위 `string_content`의 경우 `MyString`에만 존재하는 필드이기 때문에 일반적인 타입 `T`에 대해서는 작동하지 않습니다. 물론 그렇다고 해서 불가능 한 것은 아닙니다. 아래처럼 템플릿 특수화를 이용하면 되기 때문이죠.

```

template <>
void my_swap(MyString &a, MyString &b) {
    // ...
}

```

문제는 `string_content` 가 `private` 이기 때문에, 이를 위해 `MyString` 내부에 `swap` 관련한 함수를 만들어야 된다는 것입니다. 사실 이렇게 된다면 굳이 `my_swap` 이라는 함수를 정의할 필요가 없게 됩니다.

위 문제를 원래의 `my_swap` 함수를 사용하면서 좀 더 깔끔하게 해결할 수 있는 방법은 없을까요?

```
T tmp(a);
```

먼저 기존의 `my_swap` 함수를 다시 살펴봅시다. 우리는 위 문장이 복사 생성자 대신에, 이동 생성자가 되기를 원합니다. 왜냐하면 `tmp` 를 복사생성 할 필요 없이, 단순히 `a` 를 잠깐 옮겨놓기만 하면 되기 때문이지요. 하지만 문제는 `a` 가 좌측값이라는 점입니다 (`a` 라는 실체가 있으므로). 따라서 지금 이 상태로는 우리가 무얼 해도 이동 생성자는 오버로딩 되지 않습니다.

그렇다면, 좌측값이 우측값으로 취급될 수 있게 바꿔주는 함수 같은 것이 있을까요?

move 함수 (move semantics)

다행이도 C++ 11 부터 `<utility>` 라이브러리에서 좌측값을 우측값으로 바꾸어주는 `move` 함수를 제공하고 있습니다. 아래 예시를 통해서 간단히 사용하는 방법을 살펴봅시다.

```
#include <iostream>
#include <utility>

class A {
public:
    A() { std::cout << "일반 생성자 호출!" << std::endl; }
    A(const A& a) { std::cout << "복사 생성자 호출!" << std::endl; }
    A(A&& a) { std::cout << "이동 생성자 호출!" << std::endl; }
};

int main() {
    A a;

    std::cout << "-----" << std::endl;
    A b(a);

    std::cout << "-----" << std::endl;
    A c(std::move(a));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
일반 생성자 호출!
```

```
-----
```

```
복사 생성자 호출!
```

```
-----
```

```
이동 생성자 호출!
```

와 같이 나옵니다.

일단 먼저 A 클래스에 아래와 같이 3 가지 형태의 생성자들을 정의하였습니다.

```
public:
A() { std::cout << "일반 생성자 호출!" << std::endl; }
A(const A& a) { std::cout << "복사 생성자 호출!" << std::endl; }
A(A&& a) { std::cout << "이동 생성자 호출!" << std::endl; }
```

그리고 이들 생성자를 호출하는 부분을 살펴봅시다.

```
A a;
std::cout << "-----" << std::endl;
A b(a);
```

먼저 여태까지 강좌를 잘 따라오신 분들이라면 위 부분에서 일반 생성자와 복사 생성자가 각각 호출되었음을 알 수 있습니다. 그 이유는 **b(a)** 를 했을 때 **a** 가 이름이 있는 좌측값 이므로 좌측값 레퍼런스가 참조하기 때문이죠.

그렇다면 바로 그 다음 줄을 볼까요?

```
A c(std::move(a));
```

놀랍게도 이번에는 이동 생성자가 호출되었습니다. 그 이유는 **std::move** 함수가 인자로 받은 객체를 우측값으로 변환해서 리턴해주기 때문입니다. 사실 이름만 보면 무언가 이동 시킬 것 같지만 실제로는 단순한 타입 변환 만 수행할 뿐입니다. ²⁾

2) C++ 의 원저자인 Bjarne Stroustrup 은 move 라고 이름을 지은 것을 후회했다고 합니다. 정확히 말하면 move 함수는 move 를 수행하지 않고 그냥 우측값으로 캐스팅만 하기 때문이죠! 더 적절한 이름은 rvalue 와 같은 것이 되겠습니다.

주의 사항

`std::move` 함수는 이동을 수행하지 않는다. 그냥 인자로 받은 객체를 우측값으로 변환할 뿐이다.

하지만 `std::move` 덕분에 강제적으로 우측값 레퍼런스를 인자로 받는 이동 생성자를 호출할 수 있었습니다. 그렇다면 이 아이디어를 바탕으로 우리의 `MyString`에 어떻게 적용할 수 있을지 살펴봅시다.

```
#include <iostream>
#include <cstring>

class MyString {
    char *string_content; // 문자열 데이터를 가리키는 포인터
    int string_length;    // 문자열 길이

    int memory_capacity; // 현재 할당된 용량

public:
    MyString();

    // 문자열로 부터 생성
    MyString(const char *str);

    // 복사 생성자
    MyString(const MyString &str);

    // 이동 생성자
    MyString(MyString &&str);

    // 일반적인 대입 연산자
    MyString &operator=(const MyString &s);

    // 이동 대입 연산자
    MyString& operator=(MyString&& s);

    ~MyString();

    int length() const;

    void println();
};

MyString::MyString() {
    std::cout << "생성자 호출!" << std::endl;
    string_length = 0;
    memory_capacity = 0;
    string_content = NULL;
}
```

```
MyString::MyString(const char *str) {
    std::cout << "생성자 호출!" << std::endl;
    string_length = strlen(str);
    memory_capacity = string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++) string_content[i] = str[i];
}

MyString::MyString(const MyString &str) {
    std::cout << "복사 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = new char[string_length];

    for (int i = 0; i != string_length; i++)
        string_content[i] = str.string_content[i];
}

MyString::MyString(MyString &&str) {
    std::cout << "이동 생성자 호출!" << std::endl;
    string_length = str.string_length;
    string_content = str.string_content;
    memory_capacity = str.memory_capacity;

    // 임시 객체 소멸 시에 메모리를 해제하지
    // 못하게 한다.
    str.string_content = nullptr;
    str.string_length = 0;
    str.memory_capacity = 0;
}

MyString::~MyString() {
    if (string_content) delete[] string_content;
}

MyString &MyString::operator=(const MyString &s) {
    std::cout << "복사!" << std::endl;
    if (s.string_length > memory_capacity) {
        delete[] string_content;
        string_content = new char[s.string_length];
        memory_capacity = s.string_length;
    }
    string_length = s.string_length;
    for (int i = 0; i != string_length; i++) {
        string_content[i] = s.string_content[i];
    }

    return *this;
}

MyString& MyString::operator=(MyString&& s) {
    std::cout << "이동!" << std::endl;
    string_content = s.string_content;
    memory_capacity = s.memory_capacity;
    string_length = s.string_length;
```

```
s.string_content = nullptr;
s.memory_capacity = 0;
s.string_length = 0;
return *this;
}

int MyString::length() const { return string_length; }

void MyString::println() {
    for (int i = 0; i != string_length; i++) std::cout << string_content[i];

    std::cout << std::endl;
}

template <typename T>
void my_swap(T &a, T &b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

int main() {
    MyString str1("abc");
    MyString str2("def");
    std::cout << "Swap 전 ----" << std::endl;
    std::cout << "str1 : ";
    str1.println();
    std::cout << "str2 : ";
    str2.println();

    std::cout << "Swap 후 ----" << std::endl;
    my_swap(str1, str2);
    std::cout << "str1 : ";
    str1.println();
    std::cout << "str2 : ";
    str2.println();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
생성자 호출 !
생성자 호출 !
Swap 전 -----
str1 : abc
str2 : def
Swap 후 -----
이동 생성자 호출 !
```

```
이동!
이동!
str1 : def
str2 : abc
```

와 같이 나옵니다.

먼저 우리의 my_swap 함수 부터 살펴봅시다.

```
template <typename T>
void my_swap(T &a, T &b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

먼저

```
T tmp(std::move(a));
```

를 통해서 tmp라는 임시 객체를 a로 부터 이동 생성하였습니다. 이동 생성이기 때문에 기존에 복사 생성하는 것 보다 훨씬 빠르게 수행됩니다.

```
a = std::move(b);
b = std::move(tmp);
```

그 다음에 a에 b를 이동 시켰고, b에 다시 tmp를 이동시킴으로써 swap을 수행하게 됩니다. 왜 여기서 일반적인 대입이 아니라 이동이 되는 것이냐면 우리가 아래와 같이 이동 대입 연산자를 정의하였기 때문입니다.

```
MyString& MyString::operator=(MyString&& s) {
    std::cout << "이동!" << std::endl;
    string_content = s.string_content;
    memory_capacity = s.memory_capacity;
    string_length = s.string_length;

    s.string_content = nullptr;
    s.memory_capacity = 0;
    s.string_length = 0;
    return *this;
}
```

이동 대입 연산자 역시 이동 생성자와 비슷하게 매우 간단합니다. 전체 문자열을 복사할 필요 없이 그냥 기존의 문자열을 가리키고 있던 `string_content` 만 복사하면 되기 때문이니까요.

여기서 알 수 있는 한 가지 사실은 실제로 데이터가 이동 되는 과정은 위와 같이 정의한 이동 생성자나 이동 대입 연산자를 호출할 때 수행 되는 것이지 `std::move` 를 한 시점에서 수행되는 것이 아니라는 점입니다.

만일 여러분이 `MyString& MyString::operator=(MyString&& s)` 를 정의하지 않았더라면 일반적인 대입 연산자가 오버로딩 되어서 매우 느린 복사가 수행됩니다. 실제로 이동 대입 연산자를 지원하고 실행하면

실행 결과

```
생성자 호출 !
생성자 호출 !
Swap 전 -----
str1 : abc
str2 : def
Swap 후 -----
이동 생성자 호출 !
복사!
복사!
str1 : def
str2 : abc
```

와 같이 그냥 일반적인 복사가 수행됩니다.

주의 사항

다시 한번 강조하지만 이동 자체는 `std::move` 를 실행함으로써 발생하는 것이 아니라 우측값을 받는 함수들이 오버로딩 되면서 수행되는 것입니다.

완벽한 전달 (perfect forwarding)

우측값 레퍼런스를 도입함으로써 해결할 수 있었던 또 다른 문제가 있습니다. 예를 들어서 아래와 같은 `wrapper` 함수를 생각해봅시다 C++ 11 에 우측값 레퍼런스가 도입되기 전 까지 해결할 수 없었던 문제가 있었습니다. 예를 들어서 아래와 같은 `wrapper` 함수를 생각해봅시다.

```
template <typename T>
void wrapper(T u) {
```

```

    g(u);
}

```

이 함수는 인자로 받은 `u` 를 그대로 `g` 라는 함수에 인자로 전달 해줍니다. 물론 왜 저런 함수가 필요하나고 생각할 수 있습니다. 그냥 저런 wrapper 함수를 만들지 말고 그냥 `g(u)` 를 호출하면 되잖아요?

하지만 실제로 저러한 형태의 전달 방식이 사용되는 경우가 종종 있습니다. 예를 들어 `vector` 에는 `emplace_back` 이라는 함수가 있습니다. 이 함수는 객체의 생성자에 전달하고 싶은 인자들을 함수에 전달하면, 알아서 생성해서 `vector` 맨 뒤에 추가해줍니다.

예를 들어서 클래스 `A` 를 원소로 가지는 벡터의 뒤에 원소를 추가하기 위해서는

```
vec.push_back(A(1, 2, 3));
```

과 같이 객체를 생성한 뒤에 인자로 전달해줘야만 합니다. 하지만 이 과정에서 불필요한 이동 혹은 복사가 발생하게 됩니다. 그 대신, `emplace_back` 함수를 사용하게 되면;

```
vec.emplace_back(1, 2, 3); // 위와 동일한 작업을 수행한다.
```

`emplace_back` 함수는 인자를 직접 전달받아서, 내부에서 `A` 의 생성자를 호출한 뒤에 이를 벡터 원소 뒤에 추가하게 되지요. 이 과정에서 불필요한 이동/복사 모두 발생하지 않습니다. 참고로 새로 생성한 객체를 벡터 뒤에 추가할 경우 위와 같이 `push_back` 을 이용하는 것 보다 `emplace_back` 을 이용하는 것이 권장되는 방식입니다.

주의 사항

`emplace_back` 을 사용 시에 주의할 점으로 어떤 생성자가 호출되는지 명확히 알고 있어야 합니다. 가끔 의도하지 않은 생성자가 호출되어서 디버깅 시 난항을 겪는 경우가 있습니다.

그렇다면 문제는 `emplace_back` 함수가 받은 인자들을 `A` 의 생성자에 제대로 전달해야 한다는 점입니다. 그렇지 않을 경우 사용자가 의도하지 않은 생성자가 호출될 수 있기 때문입니다. 그렇다면 위와 같은 wrapper 함수를 어떻게 하면 잘 정의할 수 있을까요?

```

#include <iostream>
#include <vector>

template <typename T>
void wrapper(T u) {
    g(u);
}

class A {};

```

```

void g(A& a) { std::cout << "좌측값 레퍼런스 호출" << std::endl; }
void g(const A& a) { std::cout << "좌측값 상수 레퍼런스 호출" << std::endl; }
void g(A&& a) { std::cout << "우측값 레퍼런스 호출" << std::endl; }

int main() {
    A a;
    const A ca;

    std::cout << "원본 -----" << std::endl;
    g(a);
    g(ca);
    g(A());
}

std::cout << "Wrapper ----" << std::endl;
wrapper(a);
wrapper(ca);
wrapper(A());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

원본 -----
좌측값 레퍼런스 호출
좌측값 상수 레퍼런스 호출
우측값 레퍼런스 호출
Wrapper ----
좌측값 레퍼런스 호출
좌측값 레퍼런스 호출
좌측값 레퍼런스 호출

```

와 같이 나옵니다.

```

std::cout << "원본 -----" << std::endl;
g(a);
g(ca);
g(A());
}

```

먼저 위 경우 우리의 예상대로 좌측값 레퍼런스, 좌측값 상수 레퍼런스, 우측값 레퍼런스가 각각 호출되었습니다. 반면에 `wrapper` 함수를 거쳐갔을 경우, 공교롭게도 위 세 경우 모두 좌측값 레퍼런스를 받는 `g` 함수가 호출되었습니다.

이러한 일이 발생한 이유는 C++ 컴파일러가 템플릿 타입을 추론할 때, 템플릿 인자 T 가 레퍼런스가 아닌 일반적인 타입이라면 `const` 를 무시하기 때문입니다. 다시 말해,

```
template <typename T>
void wrapper(T u) {
    g(u);
}
```

에서 T 가 전부 다 `class A` 로 추론됩니다. 따라서 위 세 경우 전부 다 좌측값 레퍼런스를 호출하는 `g` 를 호출하였습니다.

```
template <typename T>
void wrapper(T& u) {
    g(u);
}
```

그렇다면 위 경우는 어떨까요?

컴파일 오류

```
error: cannot bind non-const lvalue reference of type 'A&' to an
      rvalue of type 'A'
  wrapper(A());
          ^~~~
```

위와 같은 컴파일 오류가

```
g(A());
```

에서 발생합니다. (참고로 이 오류는 `gcc` 와 `clang` 컴파일러에서 모두 발생하는데, 비주얼 스튜디오 2017 이전 버전에서는 발생하지 않습니다. 하지만 원칙적으로 위와 같은 오류를 발생시켜야 하는 것이 맞습니다)

왜 위와 같은 오류가 발생하는지 생각해보자면 다음과 같습니다. 일단, `A()` 자체는 `const` 속성이 없으므로 템플릿 인자 추론에서 T 가 `class A` 로 추론됩니다. 하지만 `A&` 는 우측값의 레퍼런스가 될 수 없기 때문에 컴파일 오류가 발생하는 것입니다.

그렇다면 아예 우측값을 레퍼런스로 받을 수 있도록 `const A&` 와 `A&` 따로 만들어주는 방법이 있습니다. 아래와 같이 말이지요.

```
#include <iostream>
#include <vector>
```

```
template <typename T>
void wrapper(T& u) {
    std::cout << "T& 로 추론됨" << std::endl;
    g(u);
}

template <typename T>
void wrapper(const T& u) {
    std::cout << "const T&로 추론됨" << std::endl;
    g(u);
}

class A {};

void g(A& a) { std::cout << "좌측값 레퍼런스 호출" << std::endl; }
void g(const A& a) { std::cout << "좌측값 상수 레퍼런스 호출" << std::endl; }
void g(A&& a) { std::cout << "우측값 레퍼런스 호출" << std::endl; }

int main() {
    A a;
    const A ca;

    std::cout << "원본 -----" << std::endl;
    g(a);
    g(ca);
    g(A());

    std::cout << "Wrapper ----" << std::endl;
    wrapper(a);
    wrapper(ca);
    wrapper(A());
}
```

성공적으로 컴파일 하였다면

실행 결과

```
원본 -----
좌측값 레퍼런스 호출
좌측값 상수 레퍼런스 호출
우측값 레퍼런스 호출
Wrapper -----
T&로 추론됨
좌측값 레퍼런스 호출
const T&로 추론됨
```

좌측값 상수 레퍼런스 호출
const T& 로 추론됨
 좌측값 상수 레퍼런스 호출

와 같이 나옵니다.

일단 `a` 와 `ca` 의 경우 각각 `T&` 와 `const T&` 로 잘 추론되서 올바른 함수를 호출하고 있음을 알 수 있습니다. 반면에 `A()` 의 경우 `const T&`로 추론되면서 `g(const T&)` 함수를 호출하게 됩니다. 물론 이는 예상했던 일입니다. 우리가 무엇을 해도 `wrapper` 안에 `u` 가 좌측값이라는 사실은 변하지 않고 이에 언제나 좌측값 레퍼런스를 받는 함수들이 오버로딩 되겠지요.

뿐만이 아니라 다음과 같은 문제가 있습니다. 예를 들어서 함수 `g` 가 인자를 한 개가 아니라 2 개를 받는다고 가정합니다. 그렇다면 우리는 다음과 같은 모든 조합의 템플릿 함수들을 정의해야합니다.

```
template <typename T>
void wrapper(T& u, T& v) {
    g(u, v);
}

template <typename T>
void wrapper(const T& u, T& v) {
    g(u, v);
}

template <typename T>
void wrapper(T& u, const T& v) {
    g(u, v);
}

template <typename T>
void wrapper(const T& u, const T& v) {
    g(u, v);
}
```

매우 귀찮은 일입니다. 위와 같이 짜야하는 이유는 단순히 일반적인 레퍼런스가 우측값을 받을 수 없기 때문입니다. 그렇다고 해서 디폴트로 상수 레퍼런스만 받게 된다면, 상수가 아닌 레퍼런스도 상수 레퍼런스로 캐스팅되서 들어간다는 점이지요.

하지만 놀랍게도 C++ 11 에서는 이를 간단하게 해결할 수 있습니다.

보편적 레퍼런스 (Universal reference)

```
#include <iostream>

template <typename T>
void wrapper(T&& u) {
```

```

    g(std::forward<T>(u));
}

class A {};

void g(A& a) { std::cout << "좌측값 레퍼런스 호출" << std::endl; }
void g(const A& a) { std::cout << "좌측값 상수 레퍼런스 호출" << std::endl; }
void g(A&& a) { std::cout << "우측값 레퍼런스 호출" << std::endl; }

int main() {
    A a;
    const A ca;

    std::cout << "원본 -----" << std::endl;
    g(a);
    g(ca);
    g(A());
}

std::cout << "Wrapper ----" << std::endl;
wrapper(a);
wrapper(ca);
wrapper(A());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

원본 -----
좌측값 레퍼런스 호출
좌측값 상수 레퍼런스 호출
우측값 레퍼런스 호출

Wrapper -----
좌측값 레퍼런스 호출
좌측값 상수 레퍼런스 호출
우측값 레퍼런스 호출

```

와 같이 잘 작동함을 알 수 있습니다.

```

template <typename T>
void wrapper(T&& u) {
    g(std::forward<T>(u));
}

```

일단 우리의 wrapper 함수는 인자로 아예 `T &&`를 받아버리고 있습니다. 이렇게, 템플릿 인자 `T`에 대해서, 우측값 레퍼런스를 받는 형태를 보편적 레퍼런스(Universal reference)라고 합니다. 이 보편적 레퍼런스는 우측값만 받는 레퍼런스와 다릅니다.

예를 들어서 아래와 같은 코드를 봅시다.

```
#include <iostream>

void show_value(int&& t) { std::cout << "우측값 : " << t << std::endl; }

int main() {
    show_value(5); // 우측값 ok!

    int x = 3;
    show_value(x); // 애러
}
```

컴파일 하였다면

컴파일 오류

```
test2.cc: In function ‘int main()’:
test2.cc:9:15: error: cannot bind rvalue reference of type ‘int&&’
→ to lvalue of type ‘int’
    show_value(x);
           ^
test2.cc:3:6: note:   initializing argument 1 of ‘void
→ show_value(int&&)
void show_value(int&& t) { std::cout << "우측값 : " << t <<
→ std::endl; }
           ^~~~~~
```

와 같이 컴파일 오류가 발생합니다. 위처럼, 그냥 `int&& t` 형태의 함수는 우측값만을 인자로 받을 수 있습니다..

```
template <typename T>
void wrapper(T&& u) {
```

하지만 위와 같이 템플릿 타입의 우측값 레퍼런스는 다릅니다. 이 보편적 레퍼런스는 우측값 뿐만이 아니라 좌측값 역시 받아낼 수 있습니다. 그렇다면 좌측값이 왔을 때 `T`의 타입은 어떻게 해석될까요?

C++ 11에서는 다음과 같은 레퍼런스 겹침 규칙(reference collapsing rule)에 따라 `T`의 타입을 추론하게 됩니다.

```
typedef int& T;
T& r1; // int& &; r1 은 int&
T&& r2; // int &&; r2 는 int&

typedef int&& U;
U& r3; // int && &; r3 는 int&
U&& r4; // int && &&; r4 는 int&&
```

즉 쉽게 생각하면 & 는 1이고 && 은 0이라 둘 뒤에, OR 연산을 한다고 보면 됩니다.

그렇다면,

```
wrapper(a);
wrapper(ca);
```

위 두 개의 호출의 경우 T 가 각각 A& 와 const A& 로 추론될 것이고,

```
wrapper(A());
```

의 경우에는 T 가 단순히 A 로 추론되겠지요.

그런데 문제는 이제 직접 g 에 이 인자를 전달하는 방법입니다. 왜 그냥

```
g(u)
```

로 하지 않았는지 생각해봅시다. 앞서도 말했듯이 여기서 u 는 좌측값입니다. 따라서 우리는 int&& 를 오버로딩 하는 g 를 호출하고 싶었겠지만 실제로는 const int& 를 오버로딩하는 g 가 호출되게 됩니다. 따라서 이 경우 move 를 통해 u 를 다시 우측값으로 변환해야 합니다.

하지만 당연히도 아무때나 move 를 하면 안됩니다. 인자로 받은 u 가 우측값 레퍼런스 일 때 에만 move 를 해줘야 합니다. 만일 좌측값 레퍼런스일 때 move 를 해버린다면 좌측값에 오버로딩 되는 g 가 아닌 우측값에 오버로딩 되는 g 가 호출되겠지요.

```
g(forward<T>(u));
```

이 문제를 해결해주는 것이 forward 함수입니다. 이 함수는 u 가 우측값 레퍼런스 일 때 에만 마치 move 를 적용한 것처럼 작동합니다. 실제로 forward 가 어떻게 생겼나면,

```
template <class S>
S&& forward(typename std::remove_reference<S>::type& a) noexcept {
    return static_cast<S&&>(a);
}
```

와 같이 생겼는데, S 가 A& 라면 (참고로 `std::remove_reference` 는 타입의 레퍼런스를 지워주는 템플릿 메타 함수입니다)

```
A&&& forward(typename std::remove_reference<A&>::type& a) noexcept {  
    return static_cast<A&&&>(a);  
}
```

가 되어 레퍼런스 겹침 규칙에 따라

```
A& forward(A& a) noexcept { return static_cast<A&>(a); }
```

가 되버리고, S 가 A 라면, (퀴즈! 여기서 왜 `forward` 의 인자가 A&& 가 아니라 A& 일까요?)

```
A&& forward(A& a) noexcept { return static_cast<A>(a); }
```

가 되어 성공적으로 우측값으로 캐스팅해줍니다. 따라서 결과적으로 위 그림처럼 원본과 Wrapper 을 사용했을 때 모두 호출되는 함수가 동일함을 알 수 있습니다. 성공적으로 인자를 전달한 것이지요! 자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. 다음 강좌에서는 여태까지 배운 내용을 바탕으로 스마트 포인터를 사용하는 방법에 대해서 다루어보도록 하겠습니다.

생각 해보기

문제 1

실제로 `move` 와 `forward` 가 어떠한 방식으로 구현되어 있는지 궁금하신 분들은 [여기를 참고하시면 됩니다](#) 한 번 코드를 보시고 왜 이런 방식으로 구현되어 있는지 생각해보세요. (난이도 : 중)

스마트 포인터

안녕하세요 여러분! 지난번 강좌에서 다루었던 *move semantics* 와 *perfect forwarding* 은 이해가 잘 되셨나요? C++ 에서 이와 같이 우측값들을 직접 다룰 수 있는 툴들을 많이 추가해준 덕분에 좀더 세심한 최적화와, 기존에 불가능 하였던 많은 일들이 가능하게 되었습니다.

이번 강좌에서는 C++ 11 에서 자원을 관리하는 방법에 대해서 다루도록 할 것입니다. 컴퓨터에서 자원 (resource) 라 하면 여러 가지를 꼽을 수 있지만 예를 들어보자면 여러분이 할당한 메모리도 자원이고, `open` 한 파일 역시 하나의 자원이라고 볼 수 있습니다.

중요한 점은, 자원의 양은 프로그램마다 한정되어 있기 때문에 관리를 잘 해주어야 합니다. 이말은 즉슨, 사용이 끝난 자원은 반드시 반환을 해서 다른 작업 때 사용할 수 있도록 해야 합니다. 예를 들어서 메모리를 할당만 하고 해제를 하지 않는다면, 결국 메모리 부족으로 프로그램이 `crash` 될 수도 있습니다.

자원(resource) 관리의 중요성

C++ 이후에 나온 많은 언어 (Java 등등) 들은 대부분은 가비지 컬렉터 (Garbage Collector - GC) 라 불리는 자원 청소기가 기본적으로 내장되어 있습니다. 이 가비지 컬렉터의 역할은 프로그램 상에서 더 이상 쓰이지 않는 자원을 자동으로 해제해 주는 역할을 합니다. 따라서 프로그래머들이 코드를 작성할 때, 자원을 해제하는 일에 대해 크게 신경 쓸 필요는 없습니다.

하지만 C++ 의 경우는 다릅니다. 여러분이 한 번 획득한 자원은, 직접 해제해주지 않는 이상 프로그램이 종료되기 전 까지 영원히 남아있게 됩니다. (프로그램이 종료되면 운영체제가 알아서 해제해줍니다.) 예를 들어서;

```
#include <iostream>

class A {
    int *data;

public:
```

```

A() {
    data = new int[100];
    std::cout << "자원을 획득함!" << std::endl;
}

~A() {
    std::cout << "소멸자 호출!" << std::endl;
    delete[] data;
}
};

void do_something() { A *pa = new A(); }

int main() {
    do_something();

    // 할당된 객체가 소멸되지 않음!
    // 즉, 400 바이트 (4 * 100) 만큼의 메모리 누수 발생
}

```

성공적으로 컴파일 하였다면

실행 결과

자원을 획득함!

와 같이 나옵니다.

즉 자원을 획득만 하고, 소멸자는 호출되지 않은 점을 확인할 수 있습니다. 그 이유는 까먹고

```
delete pa;
```

를 해주지 않았기 때문이지요 (아시다실이 `delete` 는 메모리를 해제하기 직전 가리키는 객체의 소멸자를 호출해줍니다).

만약에 `delete` 를 `do_something` 함수 안에서 해주지 않는다면, 생성된 객체를 가리키던 `pa` 는 메모리에서 사라지게 됩니다. 따라서 Heap 어딘가에 클래스 A 의 객체가 남아있지만, 그 주소값을 가지고 있는 포인터는 메모리 상에 존재하지 않게 됩니다. 그 객체는 영원히 해제되지 못한 채 힙에서 자리만 차지하고 있게 됩니다. 위 경우 400 바이트의 메모리 누수가 발생하게 되겠네요.

언뜻 생각하기에 아 그럼 항상 잊지 말고 자원 해제를 꼭 해주면 되잖아! 라고 하실 분들이 계실 것입니다. 하지만 프로그램의 크기가 커지면, 자원을 해제하는 위치가 애매한 경우가 많아서 놓치기 십상입니다. 물론 그래도 그건 프로그래머 책임 아님? 이라고 반문 하실 수 있습니다.

그런데, 다음과 같은 상황을 생각해보세요

```
#include <iostream>

class A {
    int *data;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }

    ~A() {
        std::cout << "자원을 해제함!" << std::endl;
        delete[] data;
    }
};

void thrower() {
    // 예외를 발생시킴!
    throw 1;
}

void do_something() {
    A *pa = new A();
    thrower();

    // 발생된 예외로 인해 delete pa 가 호출되지 않는다!
    delete pa;
}

int main() {
    try {
        do_something();
    } catch (int i) {
        std::cout << "예외 발생!" << std::endl;
        // 예외 처리
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
자원을 획득함!
예외 발생!
```

와 같이 나옵니다.

보시다 싶이, `thrower()` 로에서 발생된 예외로 인해, 밑에 있는 `delete pa` 가 실행 되지 않고 넘어가버렸습니다. 물론 예외는 정상적으로 처리되었지만, 이로 인해 메모리 누수는 피할 수 없게 됩니다.

그렇다면 이 상황을 어떻게 해결할까요?

Resource Acquisition Is Initialization - RAI

C++ 창시자인 비야네 스트로스트룹은 C++에서 자원을 관리하는 방법으로 다음과 같은 디자인 패턴을 제안하였습니다. 바로 흔히 *RAII* 라 불리는 자원의 획득은 초기화다 - **Resource Acquisition Is Initialization** 입니다. 이는 자원 관리를 스택에 할당한 객체를 통해 수행하는 것입니다.

지난 강좌에서 예외가 발생해서 함수를 빠져나가더라도, 그 함수의 스택에 정의되어 있는 모든 객체들은 빠짐없이 소멸자가 호출된다고 하였습니다 (이를 *stack unwinding* 이라 한다고 했었죠). 물론 예외가 발생하지 않는다면, 함수가 종료될 때 당연히 소멸자들이 호출되지요.

그렇다면 생각을 조금 바꿔서 만약에 이 소멸자들 안에 다 사용한 자원을 해제하는 루틴을 넣으면 어떨까요?

예를 들어서 위 포인터 `pa` 의 경우 객체가 아니기 때문에 소멸자가 호출되지 않습니다. 그렇다면 그 대신에, `pa` 를 일반적인 포인터가 아닌, 포인터 '객체'로 만들어서 자신이 소멸 될 때 자신이 가리키고 있는 데이터도 같이 `delete` 하게 하면 됩니다. 즉, 자원 (이 경우 메모리) 관리를 스택의 객체 (포인터 객체)를 통해 수행하게 되는 것입니다.

이렇게 똑똑하게 작동하는 포인터 객체를 스마트 포인터(smart pointer)라고 부릅니다. C++ 11 이전에, 이러한 문제를 해결하기 위해 `auto_ptr` 란게 잠시 등장했었지만, **너무나 많은 문제들이 있었기에** 사용을 권장하지 않습니다. (아니, 아예 사용을 금지한다고 보는 것이 맞습니다.)¹⁾

C++ 11에서는 `auto_ptr` 를 보완한 두 가지 형태의 새로운 스마트 포인터를 제공하고 있습니다. 바로 `unique_ptr` 과 `shared_ptr` 입니다.

객체의 유일한 소유권 - `unique_ptr`

C++에서 메모리를 잘못된 방식으로 관리하였을 때 크게 두 가지 종류의 문제점이 발생할 수 있습니다.

첫 번째로 앞서도 이야기한 메모리를 사용한 후에 해제하지 않은 경우입니다 (이를 메모리 누수 (memory leak) 이라고 부릅니다) 간단한 프로그램의 경우 크게 문제될 일이 없지만, 서버처럼 장시

1) 혹시라도 지금 `auto_ptr` 을 이용하고 있다면 빨리 바꾸시기를 바랍니다. C++ 11에서는 deprecated 되었고, C++ 17에서는 아예 삭제될 예정입니다.

간 작동하는 프로그램의 경우 시간이 지남에 따라 점점 사용하는 메모리 양의 늘어나서 결과적으로 나중에 시스템 메모리가 부족해져서 서버가 죽어버리는 상황이 발생할 수 도 있습니다.

다행이도 위 문제는 위에서 이야기한 RAI^I 패턴을 사용하면 해결 할 수 있습니다. RAI^I 를 통해서 사용이 끝난 메모리는 항상 해제시켜 버리면 메모리 누수 문제를 사전에 막을 수 있습니다.

두 번째로 발생 가능한 문제는, 이미 해제된 메모리를 다시 참조하는 경우입니다. 예를 들어서

```
Data* data = new Data();
Data* data2 = data;

// data 의 입장 : 사용 다 했으니 소멸시켜야지.
delete data;

// ...

// data2 의 입장 : 나도 사용 다 했으니 소멸시켜야지
delete data2;
```

위 경우 data 와 data2 가 동시에 한 객체를 가리키고 있는데, delete data 를 통해 그 객체를 소멸시켜주었습니다. 그런데, data2 가 이미 소멸된 객체를 다시 소멸시키려고 합니다. 보통 이럴 경우 메모리 오류가 나면서 프로그램이 죽게 됩니다. 이렇게 이미 소멸된 객체를 다시 소멸시켜서 발생하는 버그를 double free 버그라고 부릅니다.

위와 같은 문제가 발생한 이유는 만들어진 객체의 소유권이 명확하지 않아서입니다. 만약에, 우리가 어떤 포인터에 객체의 유일한 소유권을 부여해서, 이 포인터 말고는 객체를 소멸시킬 수 없다! 라고 한다면, 위와 같이 같은 객체를 두 번 소멸시켜버리는 일은 없을 것입니다.

위 경우 data 에 new Data() 로 생성된 객체의 소유권을 보유한다면, delete data 만 가능하고, delete data2 는 불가능하게 됩니다.

C++ 에서는 이렇게, 특정 객체에 유일한 소유권을 부여하는 포인터 객체를 unique_ptr 라고 합니다.

```
#include <iostream>
#include <memory>

class A {
    int *data;

public:
    A() {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
    }

    void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }
```

```

~A() {
    std::cout << "자원을 해제함!" << std::endl;
    delete[] data;
}

void do_something() {
    std::unique_ptr<A> pa(new A());
    pa->some();
}

int main() { do_something(); }

```

성공적으로 컴파일 하였다면

실행 결과

```

자원을 획득함!
일반 포인터와 동일하게 사용가능!
자원을 해제함!

```

와 같이 나옵니다.

```
std::unique_ptr<A> pa(new A());
```

먼저 `unique_ptr` 을 정의하는 부분부터 살펴봅시다. `unique_ptr` 를 정의하기 위해서는 템플릿에 인자로, 포인터가 가리킬 클래스를 전달하면 됩니다. 위 경우 `pa` 는 `A` 클래스의 객체를 가리키는 포인터가 되겠지요. 위는 마치

```
A* pa = new A();
```

와 동일한 문장이라 생각하시면 됩니다.

```
pa->some();
```

그렇다면 이제 위와 같이 `pa` 가 포인터인것처럼 사용하시면 됩니다. `unique_ptr` 은 `->` 연산자를 오버로드해서 마치 포인터를 다루는 것과 같이 사용할 수 있게 하였습니다.

또한 이 `unique_ptr` 덕분에 RAI^I 패턴을 사용할 수 있습니다. `pa` 는 스택에 정의된 객체이기 때문에, `do_something()` 함수가 종료될 때 자동으로 소멸자가 호출됩니다. 그리고 이 `unique_ptr` 는 소멸자 안에서 자신이 가리키고 있는 자원을 해제해 주기 때문에, 자원이 잘 해제될 수 있었습니다.

만약에 unique_ptr 를 복사하려고 한다면 어떨까요?

```
#include <iostream>
#include <memory>

class A {
    int *data;

public:
    A() {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
    }

    void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }

    ~A() {
        std::cout << "자원을 해제함!" << std::endl;
        delete[] data;
    }
};

void do_something() {
    std::unique_ptr<A> pa(new A());

    // pb 도 객체를 가리키게 할 수 있을까?
    std::unique_ptr<A> pb = pa;
}

int main() { do_something(); }
```

컴파일 하였다면

컴파일 오류

```
'std::unique_ptr<A, std::default_delete<_Ty>>::unique_ptr(const
↳ std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': attempting
↳ to reference a deleted function
```

위와 같은 오류가 나오게 됩니다. 위 오류는, 삭제된 함수를 사용하려고 했다는 뜻인데, 삭제된 함수가 도대체 무슨 말일까요?

삭제된 함수

사용을 원치 않는 함수를 삭제시키는 방법은 C++ 11 에 추가된 기능입니다. 아래와 같은 코드를 살펴봅시다.

```
#include <iostream>

class A {
public:
    A(int a){};
    A(const A& a) = delete;
};

int main() {
    A a(3); // 가능
    A b(a); // 불가능 (복사 생성자는 삭제됨)
}
```

컴파일 하게 된다면 복사 생성자를 호출하는 부분에서 오류가 발생함을 알 수 있습니다. 왜냐하면,

```
A(const A& a) = delete;
```

위와 같이 복사 생성자를 명시적으로 삭제하였기 때문이지요. 따라서, 클래스 A 의 복사 생성자는 존재하지 않습니다. 위와 같이 = delete; 를 사용하게 되면, 프로그래머가 명시적으로 '이 함수는 쓰지 마!' 라고 표현할 수 있습니다. 혹시나 사용하더라도 컴파일 오류가 발생하게 됩니다.

unique_ptr 도 마찬가지로 unique_ptr 의 복사 생성자가 명시적으로 삭제되었습니다. 그 이유는 unique_ptr 는 어떠한 객체를 유일하게 소유해야 하기 때문이지요. 만일 unique_ptr 를 복사 생성할 수 있게 된다면, 특정 객체를 여러 개의 unique_ptr 들이 소유하게 되는 문제가 발생합니다. 따라서, 각각의 unique_ptr 들이 소멸될 때 전부 객체를 delete 하려 해서 앞서 말한 double free 버그가 발생하게 됩니다.

unique_ptr 소유권 이전하기

앞서 unique_ptr 는 복사가 되지 않는다고 하였지만, 소유권은 이전할 수 있습니다.

```
#include <iostream>
#include <memory>

class A {
    int *data;
```

```

public:
A() {
    std::cout << "자원을 획득함!" << std::endl;
    data = new int[100];
}

void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }

~A() {
    std::cout << "자원을 해제함!" << std::endl;
    delete[] data;
}
};

void do_something() {
    std::unique_ptr<A> pa(new A());
    std::cout << "pa : ";
    pa->some();

    // pb에 소유권을 이전.
    std::unique_ptr<A> pb = std::move(pa);
    std::cout << "pb : ";
    pb->some();
}

int main() { do_something(); }

```

성공적으로 컴파일 하였다면

실행 결과

```

자원을 획득함!
pa : 일반 포인터와 동일하게 사용가능!
pb : 일반 포인터와 동일하게 사용가능!
자원을 해제함!

```

와 같이 나옵니다.

`unique_ptr`은 복사 생성자는 정의되어 있지 않지만, 이동 생성자는 가능 합니다. 왜냐하면, 마치 소유권을 이동시킨다 라는 개념으로 생각하면 되기 때문이지요.

```
std::unique_ptr<A> pb = std::move(pa);
```

에서 위와 같이 `pa`를 `pb`에 강제로 이동시켜버립니다. (여기서 퀴즈! `std::move`가 왜 필요할까요?) 이제 `pb`가 `new A`로 생성된 객체의 소유권을 갖게 되고, `pa`는 아무 것도 가리키고 있지

않게 됩니다. 실제로,

```
pa.get()
```

으로 pa 가 가리키고 있는 실제 주소값을 확인해보면 0 (nullptr) 이 나옵니다. 따라서 pa 를 이동시켜버린 이후에 pa->some() 을 하게 되면 문제가 발생하게 되겠지요!

따라서 소유권을 이동 시킨 이후에 기존의 unique_ptr 을 접근하지 않도록 조심해야 합니다.

주의 사항

소유권이 이전된 unique_ptr 를 맹글링 포인터(dangling pointer) 라고 하며 이를 재 참조할 시에 런타임 오류가 발생하도록 합니다. 따라서 소유권 이전은, 맹글링 포인터를 절대 다시 참조하지 않겠다는 확신 하에 이동해야 합니다.

unique_ptr 를 함수 인자로 전달하기

만약에 어떠한 unique_ptr 를 함수 인자로 전달하고 싶다면 어떨까요? 앞서 말했듯이, unique_ptr 는 복사 생성자가 없다고 하였습니다. 그렇다면, 그냥 함수에 레퍼런스에 전달하면 어떨까요?

```
#include <iostream>
#include <memory>

class A {
    int* data;

public:
    A() {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
    }

    void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }

    void do_sth(int a) {
        std::cout << "무언가를 한다!" << std::endl;
        data[0] = a;
    }

    ~A() {
        std::cout << "자원을 해제함!" << std::endl;
        delete[] data;
    }
};
```

```
// 올바르지 않은 전달 방식
void do_something(std::unique_ptr<A>& ptr) { ptr->do_sth(3); }

int main() {
    std::unique_ptr<A> pa(new A());
    do_something(pa);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
자원을 획득함!
무언가를 한다!
자원을 해제함!
```

와 같이 나옵니다.

일단, 함수 내부로 `unique_ptr` 가 잘 전달 되었음을 알 수 있습니다. 하지만, 위와 같이 함수에 `unique_ptr` 을 전달하는 것이 문맥 상 맞는 코드 일까요?

앞서 말했듯이 `unique_ptr` 은 어떠한 객체의 소유권 을 의미한다고 말했습니다. 하지만, 위와 같이 레퍼런스로 `unique_ptr` 을 전달했다면, `do_something` 함수 내부에서 `ptr` 은 더이상 유일한 소유권을 의미하지 않습니다.

물론 `ptr` 은 레퍼런스 이기 때문에, `do_something` 함수가 종료되면서 `pa` 가 가리키고 있는 객체를 파괴하지는 않습니다. 하지만, `pa` 가 유일하게 소유하고 있던 객체는 이제, 적어도 `do_something` 함수 내부에서는 `ptr` 을 통해서도 소유할 수 있게 된다는 것입니다. 즉, `unique_ptr` 은 소유권을 의미한다는 원칙에 위배되는 것이지요.

따라서, `unique_ptr` 의 레퍼런스를 사용하는 것은 `unique_ptr` 를 소유권 이라는 중요한 의미를 망각한 채 단순히 포인터의 단순한 `Wrapper` 로 사용하는 것에 불과합니다.

그렇다면, 함수에 올바르게 `unique_ptr` 를 전달하는 방법이 있을까요? 이는 단순합니다. 그냥 원래의 포인터 주소값을 전달해주면 됩니다.

```
#include <iostream>
#include <memory>

class A {
    int* data;

public:
    A() {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
```

```

}

void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }

void do_sth(int a) {
    std::cout << "무언가를 한다!" << std::endl;
    data[0] = a;
}

~A() {
    std::cout << "자원을 해제함!" << std::endl;
    delete[] data;
}
};

void do_something(A* ptr) { ptr->do_sth(3); }

int main() {
    std::unique_ptr<A> pa(new A());
    do_something(pa.get());
}

```

성공적으로 컴파일 하였다면

실행 결과

```

자원을 획득함!
무언가를 한다!
자원을 해제함!

```

와 같이 나옵니다.

`unique_ptr`의 `get` 함수를 호출하면, 실제 객체의 주소값을 리턴해줍니다. 위 경우 `do_something` 함수가 일반적인 포인터를 받고 있습니다. 이렇게 된다면, 소유권이라는 의미는 버린 채, `do_something` 함수 내부에서 객체에 접근할 수 있는 권한을 주는 것입니다.

정리해보자면,

- `unique_ptr`은 어떤 객체의 유일한 소유권을 나타내는 포인터이며, `unique_ptr` 가 소멸될 때, 가리키던 객체 역시 소멸된다.
- 만약에 다른 함수에서 `unique_ptr` 가 소유한 객체에 일시적으로 접근하고 싶다면, `get` 을 통해 해당 객체의 포인터를 전달하면 된다.
- 만약에 소유권을 이동하고자 한다면, `unique_ptr` 를 `move` 하면 된다.

unique_ptr 을 쉽게 생성하기

C++ 14 부터 unique_ptr 을 간단히 만들 수 있는 std::make_unique 함수를 제공합니다.

```
#include <iostream>
#include <memory>

class Foo {
    int a, b;

public:
    Foo(int a, int b) : a(a), b(b) { std::cout << "생성자 호출!" << std::endl; }
    void print() { std::cout << "a : " << a << ", b : " << b << std::endl; }
    ~Foo() { std::cout << "소멸자 호출!" << std::endl; }
};

int main() {
    auto ptr = std::make_unique<Foo>(3, 5);
    ptr->print();
}
```

성공적으로 컴파일 하였다면

실행 결과

```
생성자 호출!
a : 3, b : 5
소멸자 호출!
```

와 같이 잘 작동함을 알 수 있습니다. make_unique 함수는 아예 템플릿 인자로 전달된 클래스의 생성자에 인자들에 직접 완벽한 전달 을 수행합니다. 따라서 기존 처럼 불필요하게

```
std::unique_ptr<Foo> ptr(new Foo(3, 5));
```

할 필요 없이 간단히 make_unique 로 만들 수 있습니다.

unique_ptr 를 원소로 가지는 컨테이너

자 이제 마지막으로, unique_ptr 를 원소로 가지는 STL 컨테이너에 대해 알아보도록 합시다. 사실, unique_ptr 은 다른 타입들과 큰 차이는 없지만, 복사 생성자가 없다 라는 특성 때문에 처음에 사용하시는 분들이 많은 애를 먹는 경우를 종종 보았습니다.

```
#include <iostream>
#include <memory>
#include <vector>

class A {
    int *data;

public:
    A(int i) {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
        data[0] = i;
    }

    void some() { std::cout << "일반 포인터와 동일하게 사용가능!" << std::endl; }

    ~A() {
        std::cout << "자원을 해제함!" << std::endl;
        delete[] data;
    }
};

int main() {
    std::vector<std::unique_ptr<A>> vec;
    std::unique_ptr<A> pa(new A(1));

    vec.push_back(pa); // ??
}
```

컴파일 하였다면 아래와 같은 무시무시한 컴파일 오류를 맛보게 될 것입니다.

컴파일 오류

```
In file included from
→ /usr/include/x86_64-linux-gnu/c++/7/bits/c++allocator.h:33:0,
     from /usr/include/c++/7/bits/allocator.h:46,
     from /usr/include/c++/7/string:41,
     from /usr/include/c++/7/bits/locale_classes.h:40,
     from /usr/include/c++/7/bits/ios_base.h:41,
     from /usr/include/c++/7/ios:42,
     from /usr/include/c++/7/ostream:38,
     from /usr/include/c++/7/iostream:39,
     from 13.1.7.cc:1:
```

```

/usr/include/c++/7/ext/new_allocator.h: In instantiation of 'void
 $\sim$  __gnu_cxx::new_allocator<_Tp>::construct(_Up*, _Args&& ...)
 $\sim$  [with _Up = std::unique_ptr<A>; _Args = {const
 $\sim$  std::unique_ptr<A, std::default_delete<A> &}]; _Tp =
 $\sim$  std::unique_ptr<A>]':
/usr/include/c++/7/bits/allocator_traits.h:475:4:   required from
 $\sim$  'static void std::allocator_traits<std::allocator<_CharT>
 $\sim$  >::construct(std::allocator_traits<std::allocator<_CharT>
 $\sim$  >::allocator_type&, _Up*, _Args&& ...) [with _Up =
 $\sim$  std::unique_ptr<A>; _Args = {const std::unique_ptr<A,
 $\sim$  std::default_delete<A> &}]; _Tp = std::unique_ptr<A>;
 $\sim$  std::allocator_traits<std::allocator<_CharT> >::allocator_type
 $\sim$  = std::allocator<std::unique_ptr<A> >]'
/usr/include/c++/7/bits/stl_vector.h:943:30:   required from 'void
 $\sim$  std::vector<_Tp, _Alloc>::push_back(const value_type&) [with
 $\sim$  _Tp = std::unique_ptr<A>; _Alloc =
 $\sim$  std::allocator<std::unique_ptr<A> >; std::vector<_Tp,
 $\sim$  _Alloc>::value_type = std::unique_ptr<A>]'
13.1.7.cc:32:19:   required from here
/usr/include/c++/7/ext/new_allocator.h:136:4: error: use of
 $\sim$  deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const
 $\sim$  std::unique_ptr<_Tp, _Dp>&) [with _Tp = A; _Dp =
 $\sim$  std::default_delete<A>]'

{ ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
^~~~~~
In file included from /usr/include/c++/7/memory:80:0,
                 from 13.1.7.cc:2:
/usr/include/c++/7/bits/unique_ptr.h:388:7: note: declared here
      unique_ptr(const unique_ptr&) = delete;
      ^~~~

```

이와 같은 오류가 발생하는 이유는 당연합니다. 역시, 삭제된 `unique_ptr`의 복사 생성자에 접근하였기 때문이지요. 기본적으로 `vector`의 `push_back` 함수는 전달된 인자를 복사해서 집어 넣기 때문에 위와 같은 문제가 발생하게 되는 것이지요.

이를 방지하기 위해서는 명시적으로 `pa`를 `vector` 안으로 이동 시켜주어야만 합니다. 즉 `push_back`의 우측값 레퍼런스를 받는 버전이 오버로딩 될 수 있도록 말이지요.

```

int main() {
    std::vector<std::unique_ptr<A>> vec;
}

```

```

    std::unique_ptr<A> pa(new A(1));

    vec.push_back(std::move(pa)); // 잘 실행됨
}

```

와 같이 하면 잘 컴파일 됩니다.

하지만 재미있게도, `emplace_back` 함수를 이용하면, `vector` 안에 `unique_ptr` 을 직접 생성하면서 집어넣을 수도 있습니다. 즉, 불필요한 이동 과정을 생략할 수 있다는 것입니다.

```

#include <iostream>
#include <memory>
#include <vector>

class A {
    int *data;

public:
    A(int i) {
        std::cout << "자원을 획득함!" << std::endl;
        data = new int[100];
        data[0] = i;
    }

    void some() { std::cout << "값 : " << data[0] << std::endl; }

    ~A() {
        std::cout << "자원을 해제함!" << std::endl;
        delete[] data;
    }
};

int main() {
    std::vector<std::unique_ptr<A>> vec;

    // vec.push_back(std::unique_ptr<A>(new A(1))); 과 동일
    vec.emplace_back(new A(1));

    vec.back()->some();
}

```

성공적으로 컴파일 하였다면

실행 결과

자원을 획득함!

값 : 1

자원을 해제함!

와 같이 나옵니다.

`emplace_back` 함수는 전달된 인자를 완벽한 전달(perfect forwarding)을 통해, 직접 `unique_ptr<A>`의 생성자에 전달 해서, `vector` 맨 뒤에 `unique_ptr<A>` 객체를 생성해버리게 됩니다. 따라서, 위에서처럼 불필요한 이동 연산이 필요 없게 됩니다 (왜냐하면 `vector` 맨 뒤에 생성하기 때문에!)

참고로 `emplace_back` 을 사용 시에 어떠한 생성자가 호출되는지 주의 하셔야 합니다. 예를 들어서

```
std::vector<int> v;  
v.emplace_back(100000);
```

을 하게 되면, 100000 이란 `int` 값을 `v` 에 추가하게 되지만

```
std::vector<std::vector<int>> v;  
v.emplace_back(100000);
```

를 하게 되면 원소가 100000 개 들어있는 벡터를 `v` 에 추가하게 됩니다.

생각 해보기

문제 1

`unique_ptr` 을 어떤식으로 구현할 수 있을지 생각해보세요 (난이도 : 중상)

여러 객체가 소유할 수 있는 포인터

안녕하세요 여러분! 지난 강좌에서는 객체를 유일하게 소유하는 스마트 포인터인 `unique_ptr`에 대해서 다루어 보았습니다. 대부분의 경우 하나의 자원은 한 개의 스마트 포인터에 의해 소유되는 것이 바람직하고, 나머지 접근은 (소유가 아닌) 그냥 일반 포인터로 처리하면 됩니다.

하지만, 때에 따라서는 여러 개의 스마트 포인터가 하나의 객체를 같이 소유해야 하는 경우가 발생합니다. 예를 들어서 여러 객체에서 하나의 자원을 사용하고자 합니다. 후에 자원을 해제하기 위해서는 이 자원을 사용하는 모든 객체들이 소멸되어 하는데, 어떤 객체가 먼저 소멸되는지 알 수 없기 때문에 이 자원 역시 어느 타이밍에 해제 시켜야 할지 알 수 없게 됩니다.

따라서 이 경우, 좀 더 스마트한 포인터가 있어서, 특정 자원을 **몇 개의 객체에서 가리키는지를 추적**한 다음에, 그 수가 0이 되어야만 비로소 해제를 시켜주는 방식의 포인터가 필요합니다.

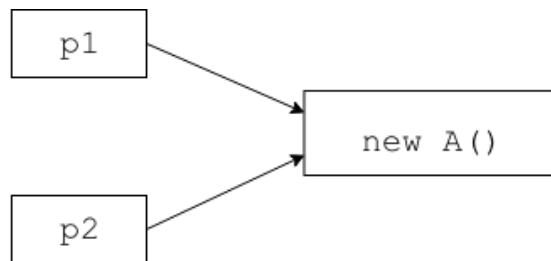
`shared_ptr`

`shared_ptr`은 앞서 이야기한 방식을 정확히 수행하는 스마트 포인터입니다. 기존에 유일하게 객체를 소유하는 `unique_ptr`와는 다르게, `shared_ptr`로 객체를 가리킬 경우, 다른 `shared_ptr` 역시 그 객체를 가리킬 수 있습니다. 예를 들어서;

```
std::shared_ptr<A> p1(new A());
std::shared_ptr<A> p2(p1); // p2 역시 생성된 객체 A 를 가리킨다.

// 반면에 unique_ptr의 경우
std::unique_ptr<A> p1(new A());
std::unique_ptr<A> p2(p1); // 컴파일 오류!
```

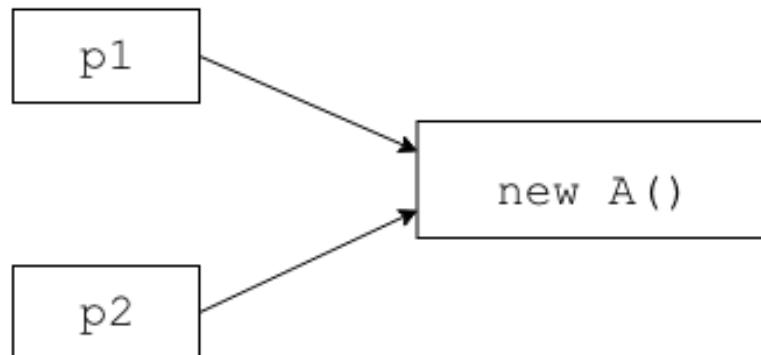
`p1`과 `p2`의 경우 같이 동일한 객체인 `A()`를 가리키지만, `unique_ptr`의 경우 유일한 소유권만 인정되므로 컴파일 오류가 발생하게 됩니다.



여러 개의 `shared_ptr` 가 같은 객체를 가리킬 수 있다

위 그림과 같이 `shared_ptr` 는 같은 객체를 가리킬 수 있습니다. 이를 위해서는, 앞서 말했듯이, 몇 개의 `shared_ptr` 들이 원래 객체를 가리키는지 알아야만 합니다. 이를 참조 개수 (reference count) 라고 하는데, 참조 개수가 0 이 되어야 가리키고 있는 객체를 해제할 수 있겠지요.

`ref count : 2`



`ref count : 2`

`p1` 과 `p2` 의 참조 카운트는 2 이다.

위 그림의 경우 `p1` 과 `p2` 가 같은 객체를 가리키고 있으므로, 참조 개수가 2 가 됩니다.

한번 아래 예제를 살펴보실까요.

```

#include <iostream>
#include <memory>
#include <vector>

class A {
    int *data;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }

    ~A() {
        std::cout << "소멸자 호출!" << std::endl;
        delete[] data;
    }
};

int main() {
    std::vector<std::shared_ptr<A>> vec;

    vec.push_back(std::shared_ptr<A>(new A()));
}
  
```

```
vec.push_back(std::shared_ptr<A>(vec[0]));
vec.push_back(std::shared_ptr<A>(vec[1]));

// 벡터의 첫번째 원소를 소멸 시킨다.
std::cout << "첫 번째 소멸!" << std::endl;
vec.erase(vec.begin());

// 그 다음 원소를 소멸 시킨다.
std::cout << "다음 원소 소멸!" << std::endl;
vec.erase(vec.begin());

// 마지막 원소 소멸
std::cout << "마지막 원소 소멸!" << std::endl;
vec.erase(vec.begin());

std::cout << "프로그램 종료!" << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
자원을 획득함!
첫 번째 소멸!
다음 원소 소멸!
마지막 원소 소멸!
소멸자 호출!
프로그램 종료!
```

와 같이 나옵니다.

위 예제의 경우 `shared_ptr` 를 원소로 가지는 벡터 `vec` 을 정의한 후, `vec[0]`, `vec[1]`, `vec[2]` 가 모두 같은 A 객체를 가리키는 `shared_ptr` 를 생성하였습니다.

```
// 벡터의 첫번째 원소를 소멸 시킨다.
std::cout << "첫 번째 소멸!" << std::endl;
vec.erase(vec.begin());

// 그 다음 원소를 소멸 시킨다.
std::cout << "다음 원소 소멸!" << std::endl;
vec.erase(vec.begin());

// 마지막 원소 소멸
std::cout << "마지막 원소 소멸!" << std::endl;
vec.erase(vec.begin());
```

그 다음에 위 부분에서, `vec` 의 첫 번째 원소 부터 차례대로 지워나갔는데, `unique_ptr` 와는 다르게 `shared_ptr` 의 경우 객체를 가리키는 모든 스마트 포인터 들이 소멸되어야만 객체를 파괴하기 때문에, 처음 두 번의 `erase` 에서는 아무것도 하지 않다가 마지막의 `erase` 에서 비로소 A 의 소멸자를 호출하는 것을 볼 수 있습니다.

즉 참조 개수가 처음에는 3 이 였다가, 2, 1, 0 순으로 줄어들게 되겠지요.

현재 `shared_ptr` 의 참조 개수가 몇 개 인지는 `use_count` 함수를 통해 알 수 있습니다. 예를 들어서

```
std::shared_ptr<A> p1(new A());
std::shared_ptr<A> p2(p1); // p2 역시 생성된 객체 A 를 가리킨다.

std::cout << p1.use_count(); // 2
std::cout << p2.use_count(); // 2
```

와 같이 출력 되겠지요.

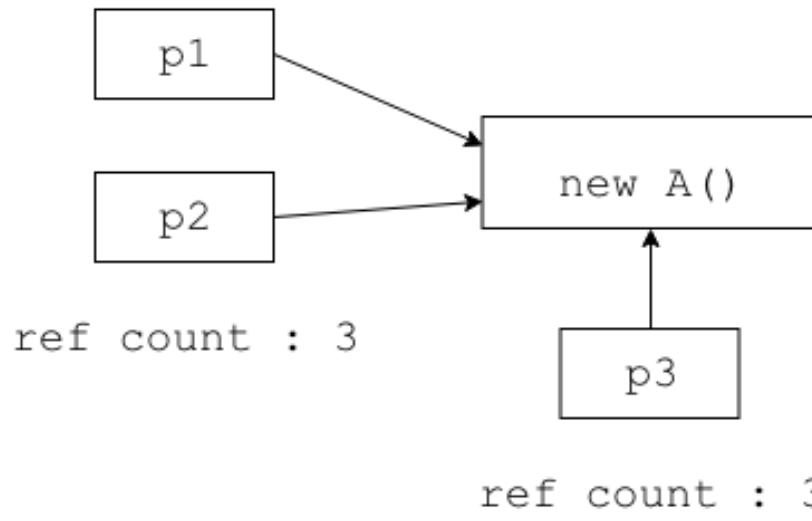
그렇다면 퀴즈 하나! 위에서도 보시다시피 개개의 `shared_ptr` 들은 참조 개수가 몇 개 인지 알고 있어야만 합니다. 이 경우 어떻게 하면 같은 객체를 가리키는 `shared_ptr` 끼리 동기화를 시킬 수 있을까요?

만약에, `shared_ptr` 내부에 참조 개수를 저장한다면 아래와 같은 문제가 생길 수 있습니다. 만약에 아래와 같이 한 개의 `shared_ptr` 가 추가적으로 해당 객체를 가리킨다면 어떨까요?

```
std::shared_ptr<A> p3(p2);
```

와 같이 말이지요. 그렇다면 여차저차 해서 `p2` 의 참조 카운트 개수는 증가시킬 수 있다고 해도, `p1`에 저장되어 있는 참조 개수를 건드릴 수 없습니다. 즉 아래와 같은 상황이 발생하겠지요.

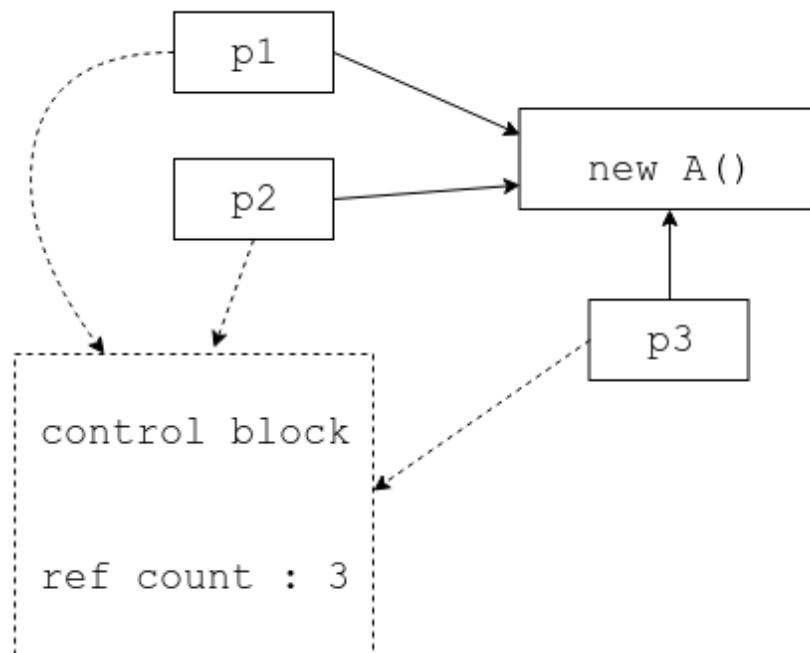
ref count : 2



ref count : 3

p1 의 참조 카운트를 바꿀 수 없다

따라서 이와 같은 문제를 방지하기 위해 처음으로 실제 객체를 가리키는 `shared_ptr` 가 제어 블록(control block) 을 동적으로 할당한 후, `shared_ptr` 들이 이 제어 블록에 필요한 정보를 공유하는 방식으로 구현됩니다. 아래 그림과 같이 말이지요.



p1, p2, p3 가 공통된 제어 블록을 공유한다

`shared_ptr` 는 복사 생성할 때마다 해당 제어 블록의 위치만 공유하면 되고, `shared_ptr` 가 소멸할 때마다 제어 블록의 참조 개수를 하나 줄이고, 생성할 때마다 하나 늘리는 방식으로 작동할

것입니다.

make_shared로 생성하자

앞서 `shared_ptr`를 처음 생성할 때 아래와 같이 하였습니다.

```
std::shared_ptr<A> p1(new A());
```

하지만 사실 이는 바람직한 `shared_ptr`의 생성 방법은 아닙니다. 왜냐하면 일단 `A`를 생성하기 위해서 동적 할당이 한 번 일어나야 하고, 그 다음 `shared_ptr`의 제어 블록 역시 동적으로 할당해야 하기 때문이지요. 즉 두 번의 동적 할당이 발생해야 합니다.

동적 할당은 상당히 비싼 연산입니다. 어차피 동적 할당을 두 번 할 것이라는 것을 알고 있다면, 아예 두 개 합친 크기로 한 번 할당 하는 것이 훨씬 빠릅니다.

```
std::shared_ptr<A> p1 = std::make_shared<A>();
```

`make_shared` 함수는 `A`의 생성자에 인자들을 받아서 이를 통해 객체 `A`와 `shared_ptr`의 제어 블록 까지 한 번에 동적 할당 한 후에 만들어진 `shared_ptr`을 리턴합니다.

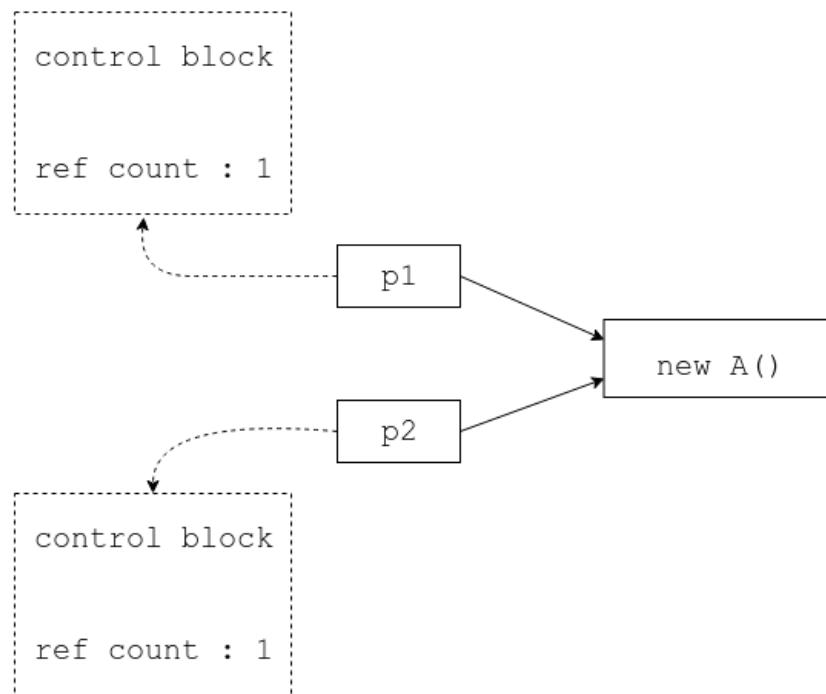
위 경우 `A`의 생성자에 인자가 없어서 `make_shared`에 아무 것도 전달하지 않았지만, 만약에 `A`의 생성자에 인자가 있다면 `make_shared`에 인자로 전달해 주면 됩니다. (그리고 `make_shared`가 `A`의 생성자에 완벽한 전달을 해주겠지요!)

shared_ptr 생성 시 주의 할 점

`shared_ptr`은 인자로 주소값이 전달된다면, 마치 자기가 해당 객체를 첫번째로 소유하는 `shared_ptr`인 것 마냥 행동합니다. 예를 들어서

```
A* a = new A();
std::shared_ptr<A> pa1(a);
std::shared_ptr<A> pa2(a);
```

를 하게 된다면 아래와 같이 이 두 개의 제어 블록이 따로 생성됩니다.



따라서 위와 같이 각각의 제어 블록들은, 다른 제어 블록들의 존재를 모르고 참조 개수를 1로 설정하게 되겠지요. 만약에 pa1이 소멸된다면, 참조 카운트가 0이 되어서 자신이 가리키는 객체 A를 소멸시켜 버립니다. pa2가 아직 가리키고 있는데도 말이지요!

물론 pa2의 참조 카운트는 계속 1이기 때문에 자신이 가리키는 객체가 살아 있을 것이라 생각할 것입니다. 설사 운 좋게도 pa2를 사용하지 않아도, pa2가 소멸되면 참조 개수가 0으로 떨어지고 자신이 가리키고 있는 (이미 해제된) 객체를 소멸시키기 때문에 오류가 발생합니다.

아래 예제를 보면 쉽게 알 수 있습니다.

```
#include <iostream>
#include <memory>

class A {
    int* data;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }

    ~A() {
        std::cout << "소멸자 호출!" << std::endl;
        delete[] data;
    }
};
```

```

int main() {
    A* a = new A();

    std::shared_ptr<A> pa1(a);
    std::shared_ptr<A> pa2(a);

    std::cout << pa1.use_count() << std::endl;
    std::cout << pa2.use_count() << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

자원을 획득함!
1
1
소멸자 호출!
소멸자 호출!
test(37272,0x11254d5c0) malloc: *** error for object
  ↳ 0x4000000000000000: pointer being freed was not allocated
test(37272,0x11254d5c0) malloc: *** set a breakpoint in
  ↳ malloc_error_break to debug
[1] 37272 abort      ./test

```

위와 같이 소멸자가 두 번 호출되면서 오류가 나게 됩니다. 오류 내용 역시, 이미 해제한 메모리를 또 해제 한다는 뜻이네요.

이와 같은 상황을 방지하려면 `shared_ptr`를 주소값을 통해서 생성하는 것을 지양해야 합니다.

하지만, 어쩔 수 없는 상황도 있습니다. 바로 객체 내부에서 자기 자신을 가리키는 `shared_ptr`를 만들 때를 생각해봅시다.

```

#include <iostream>
#include <memory>

class A {
    int *data;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }
}

```

```

~A() {
    std::cout << "소멸자 호출!" << std::endl;
    delete[] data;
}

std::shared_ptr<A> get_shared_ptr() { return std::shared_ptr<A>(this); }

int main() {
    std::shared_ptr<A> pa1 = std::make_shared<A>();
    std::shared_ptr<A> pa2 = pa1->get_shared_ptr();

    std::cout << pa1.use_count() << std::endl;
    std::cout << pa2.use_count() << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

자원을 획득함!
1
1
소멸자 호출!
소멸자 호출!
test(38479,0x10e0945c0) malloc: *** error for object
  ↳ 0x7fa1e0e02700: pointer being freed was not allocated
test(38479,0x10e0945c0) malloc: *** set a breakpoint in
  ↳ malloc_error_break to debug
[1] 38479 abort      ./test

```

위와 같이 이전과 같은 이유로 오류가 발생하게 됩니다. `get_shared_ptr` 함수에서 `shared_ptr`을 생성할 때, 이미 자기 자신을 가리키는 `shared_ptr`가 있다는 사실을 모른채 새로운 제어블록을 생성하기 때문입니다.

이 문제는 `enable_shared_from_this`를 통해 깔끔하게 해결할 수 있습니다.

`enable_shared_from_this`

우리가 `this`를 사용해서 `shared_ptr`을 만들고 싶은 클래스가 있다면, `enable_shared_from_this`를 상속 받으면 됩니다. 아래 사용 예시를 보실까요.

```

#include <iostream>
#include <memory>

```

```

class A : public std::enable_shared_from_this<A> {
    int *data;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }

    ~A() {
        std::cout << "소멸자 호출!" << std::endl;
        delete[] data;
    }

    std::shared_ptr<A> get_shared_ptr() { return shared_from_this(); }
};

int main() {
    std::shared_ptr<A> pa1 = std::make_shared<A>();
    std::shared_ptr<A> pa2 = pa1->get_shared_ptr();

    std::cout << pa1.use_count() << std::endl;
    std::cout << pa2.use_count() << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

자원을 획득함!

2

2

소멸자 호출!

와 같이 제대로 작동하는 것을 볼 수 있습니다.

`enable_shared_from_this` 클래스에는 `shared_from_this`라는 멤버 함수를 정의하고 있는데, 이 함수는 이미 정의되어 있는 제어 블록을 사용해서 `shared_ptr`을 생성합니다.

따라서 이전처럼 같은 객체에 두 개의 다른 제어 블록이 생성되는 일을 막을 수 있습니다.

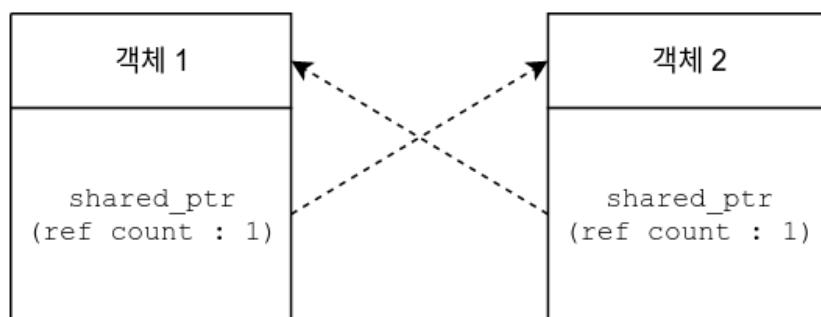
한 가지 중요한 점은 `shared_from_this`가 잘 작동하기 위해서는 해당 객체의 `shared_ptr`가 반드시 먼저 정의되어 있어야만 합니다. 즉 `shared_from_this`는 있는 제어 블록을 확인만 할 뿐, 없는 제어 블록을 만들지는 않습니다. 쉽게 말해 아래 코드는 오류가 발생합니다.

```
A* a = new A();
```

```
std::shared_ptr<A> pa1 = a->get_shared_ptr();
```

서로 참조하는 shared_ptr

앞서 `shared_ptr` 는 참조 개수가 0 이 되면 가리키는 객체를 메모리에서 해제 시킨다고 했습니다. 그런데, 객체들을 더이상 사용하지 않는되도 불구하고 참조 개수가 절대로 0 이 될 수 없는 상황이 있습니다. 아래 그림을 살펴보실까요.



이 같은 형태를 순환 참조라고 합니다.

위 그림의 경우 각 객체는 `shared_ptr` 를 하나 씩 가지고 있는데, 이 `shared_ptr` 가 다른 객체를 가리키고 있습니다. 즉 객체 1 의 `shared_ptr` 은 객체 2 를 가리키고 있고, 객체 2 의 `shared_ptr` 는 객체 1 을 가리키고 있지요.

만약에 객체 1 이 파괴가 되기 위해서는 객체 1 을 가리키고 있는 `shared_ptr` 의 참조 개수가 0 이 되어야만 합니다. 즉, 객체 2 가 파괴가 되어야 하겠지요. 하지만 객체 2 가 파괴 되기 위해서는 마찬가지로 객체 2 를 가리키고 있는 `shared_ptr` 의 참조 개수가 0 이 되어야 하는데, 그러기 위해서는 객체 1 이 파괴되어야만 합니다.

즉 이러지도 저러지도 못하는 상황이 된것입니다.

```
#include <iostream>
#include <memory>

class A {
    int *data;
    std::shared_ptr<A> other;

public:
    A() {
        data = new int[100];
        std::cout << "자원을 획득함!" << std::endl;
    }

    ~A() {
```

```

    std::cout << "소멸자 호출!" << std::endl;
    delete[] data;
}

void set_other(std::shared_ptr<A> o) { other = o; }

int main() {
    std::shared_ptr<A> pa = std::make_shared<A>();
    std::shared_ptr<A> pb = std::make_shared<A>();

    pa->set_other(pb);
    pb->set_other(pa);
}

```

성공적으로 컴파일 하였다면

실행 결과

자원을 획득함!

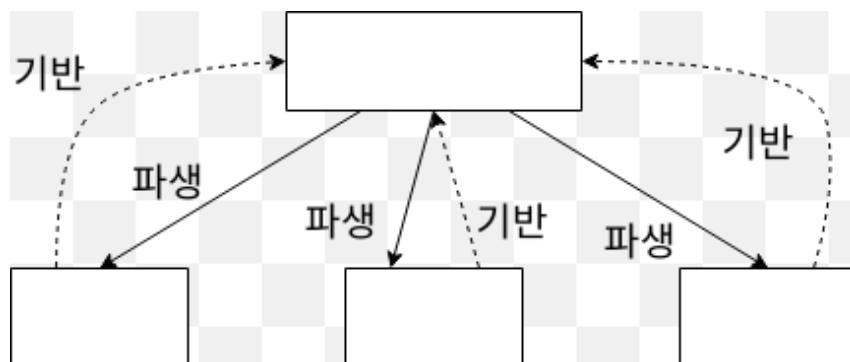
자원을 획득함!

위와 같이 소멸자가 제대로 호출되지 않음을 알 수 있습니다.

이 문제는 `shared_ptr` 자체에 내재되어 있는 문제이기 때문에 `shared_ptr`를 통해서는 이를 해결할 수 없습니다. 이러한 순환 참조 문제를 해결하기 위해 나타난 것이 바로 `weak_ptr`입니다.

weak_ptr

우리는 트리 구조를 지원하는 클래스를 만들려고 합니다. 트리 구조라 함은 아래와 가계도와 비슷하다고 생각하시면 됩니다.



즉, 한 개의 노드는 여러개의 자식 노드를 가질 수 있지만, 단 한 개의 부모 노드를 가집니다. 위

그림에서 부모 노드는 자식 노드들을 가리키고 있고 (실선), 자식 노드들은 부모 노드를 가리키고 있습니다 (점선).

위와 같은 형태를 자료 구조로 나타낸다면 어떻게 할 수 있을까요?

```
class Node {
    std::vector<std::shared_ptr<Node>> children;
    /* 어떤 타입이 와야할까? */ parent;

public:
    Node() {};
    void AddChild(std::shared_ptr<Node> node) { children.push_back(node); }
};
```

일단 기본적으로 위와 같은 형태를 취한다고 볼 수 있습니다. 부모가 여러개의 자식 노드들을 가지므로 `shared_ptr` 들의 벡터로 나타낼 수 있고, 그 노드 역시 부모 노드가 있으므로 부모 노드를 가리키는 포인터를 가집니다.

여기서 질문은 과연 `parent` 의 타입을 무엇으로 하냐 입니다.

- 만약에 일반 포인터(`Node *`)로 하게 된다면, 메모리 해제를 까먹고 하지 않을 경우 혹은 예외가 발생하였을 경우 적절하게 자원을 해제하기 어렵습니다. 물론 이미 해제된 메모리를 계속 가리키고 있을 위험도 있습니다.
- 하지만 이를 `shared_ptr`로 하게 된다면 앞서 본 순환 참조 문제가 생깁니다. 부모와 자식이 서로를 가리키기 때문에 참조 개수가 절대로 0이 될 수 없습니다. 따라서, 이들 객체들은 프로그램 끝날 때 까지 절대로 소멸되지 못하고 남아있게 됩니다.

`weak_ptr` 는 일반 포인터와 `shared_ptr` 사이에 위치한 스마트 포인터로, 스마트 포인터처럼 객체를 안전하게 참조할 수 있게 해주지만, `shared_ptr` 와는 다르게 참조 개수를 늘리지는 않습니다. 이를 그대로 약한 포인터 인것이지요.

따라서 설사 어떤 객체를 `weak_ptr` 가 가리키고 있다고 하더라도, 다른 `shared_ptr` 들이 가리키고 있지 않다면 이미 메모리에서 소멸되었을 것입니다.

이 때문에 `weak_ptr` 자체로는 원래 객체를 참조할 수 없고, 반드시 `shared_ptr` 로 변환해서 사용해야 합니다. 이 때 가리키고 있는 객체가 이미 소멸되었다면 빈 `shared_ptr` 로 변환되고, 아닐경우 해당 객체를 가리키는 `shared_ptr` 로 변환됩니다.

아래 예제를 통해 `weak_ptr` 을 어떻게 활용할 수 있는지 알아봅시다.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
```

```

class A {
    std::string s;
    std::weak_ptr<A> other;

public:
    A(const std::string& s) : s(s) { std::cout << "자원을 획득함!" << std::endl; }

    ~A() { std::cout << "소멸자 호출!" << std::endl; }

    void set_other(std::weak_ptr<A> o) { other = o; }
    void access_other() {
        std::shared_ptr<A> o = other.lock();
        if (o) {
            std::cout << "접근 : " << o->name() << std::endl;
        } else {
            std::cout << "이미 소멸됨 뀌" << std::endl;
        }
    }
    std::string name() { return s; }
};

int main() {
    std::vector<std::shared_ptr<A>> vec;
    vec.push_back(std::make_shared<A>("자원 1"));
    vec.push_back(std::make_shared<A>("자원 2"));

    vec[0]->set_other(vec[1]);
    vec[1]->set_other(vec[0]);

    // pa 와 pb 의 ref count 는 그대로다.
    std::cout << "vec[0] ref count : " << vec[0].use_count() << std::endl;
    std::cout << "vec[1] ref count : " << vec[1].use_count() << std::endl;

    // weak_ptr 로 해당 객체 접근하기
    vec[0]->access_other();

    // 벡터 마지막 원소 제거 (vec[1] 소멸)
    vec.pop_back();
    vec[0]->access_other(); // 접근 실패!
}

```

성공적으로 컴파일 하였다면

실행 결과

자원을 획득함!

자원을 획득함!

```

vec[0] ref count : 1
vec[1] ref count : 1
접근 : 자원 2
소멸자 호출!
이미 소멸됨ㅠ
소멸자 호출!

```

와 같이 나옵니다.

일단 `weak_ptr` 을 정의하는 부분 부터 살펴봅시다.

```

vec[0]->set_other(vec[1]);
vec[1]->set_other(vec[0]);

```

`set_other` 함수는 `weak_ptr<A>` 를 인자로 받고 있었는데, 여기에 `shared_ptr` 을 전달하였습니다. 즉, `weak_ptr` 는 생성자로 `shared_ptr` 나 다른 `weak_ptr` 를 받습니다. 또한 `shared_ptr` 과는 다르게, 이미 제어 블록이 만들어진 객체만이 의미를 가지기 때문에, 평범한 포인터 주소값으로 `weak_ptr` 를 생성할 수는 없습니다.

그 다음으로 살펴볼 부분은 실제 `weak_ptr` 를 `shared_ptr` 로 변환하는 과정입니다.

```

void access_other() {
    std::shared_ptr<A> o = other.lock();
    if (o) {
        std::cout << "접근 : " << o->name() << std::endl;
    } else {
        std::cout << "이미 소멸됨ㅠ" << std::endl;
    }
}

```

앞서 말했듯이 `weak_ptr` 그 자체로는 원소를 참조할 수 없고, `shared_ptr` 로 변환해야 한다고 하였습니다. 이 작업은 `lock` 함수를 통해 수행할 수 있습니다.

`weak_ptr` 에 정의된 `lock` 함수는 만일 `weak_ptr` 가 가리키는 객체가 아직 메모리에서 살아 있다면 (즉 참조 개수가 0 이 아니라면) 해당 객체를 가리키는 `shared_ptr` 을 반환하고, 이미 해제가 되었다면 아무것도 가리키지 않는 `shared_ptr` 을 반환 합니다.

```

std::shared_ptr<A> o = other.lock();
if (o) {
    std::cout << "접근 : " << o->name() << std::endl;
}

```

참고로 아무것도 가리키지 않는 `shared_ptr` 은 `false` 로 형변환 되므로 위와 같이 `if` 문으로 간단히 확인할 수 있습니다.

앞서 제어 블록에는 몇 개의 `shared_ptr` 가 가리키고 있는지를 나타내는 참조 개수(ref count)가 있다고 하였습니다. 그리고 참조 개수가 0 이 되면 해당 객체를 메모리에서 해제하는 것도 알고 있지요. 그렇다면 참조 개수가 0 이 될때 제어 블록 역시 메모리에서 해제해야 할까요?

아닙니다. 만약에 가리키는 `shared_ptr` 은 0 개지만 아직 `weak_ptr` 가 남아있다고 해봅시다. 물론 이 상태에서는 이미 객체는 해제 되어 있을 것입니다. 하지만 제어 블록 마지 해제해 버린다면, 제어 블록에서 참조 카운트가 0 이라는 사실을 알 수 없게 됩니다. ²⁾

즉, 제어 블록을 메모리에서 해제해야 하기 위해서는 이를 가리키는 `weak_ptr` 역시 0 개여야 합니다. 따라서 제어 블록에는 참조 개수와 더불어 약한 참조 개수 (weak count) 기록하게 됩니다.

자 그럼 이것으로 스마트 포인터 삼형제 (`unique_ptr`, `shared_ptr`, `weak_ptr`) 에 관한 강좌를 마치도록 하겠습니다. 스마트 포인터를 도입함으로써 골치 아픈 메모리 문제를 많이 해결 할 수 있을 것이라 생각합니다.

생각 해보기

문제 1

가계도를 관리하는 라이브러리를 만들어보세요. 기본적으로 다음과 같이 생겼을 것입니다. (난이도 : 상)

```
class Member {
private:
    std::vector<std::shared_ptr<Member>> children;
    std::vector<std::weak_ptr<Member>> parents;
    std::vector<std::weak_ptr<Member>> spouse;

public:
    void AddParent(const std::shared_ptr<Member>& parent);
    void AddSpouse(const std::shared_ptr<Member>& spouse);
    void AddChild(const std::shared_ptr<Member>& child);
};

class FamilyTree {
private:
    std::vector<std::shared_ptr<Member>> entire_family;

public:
    // 두 사람 사이의 촌수를 계산한다.
}
```

2) 메모리가 해제된 이후에, 같은 자리가 다른 용도로 할당 될 수 있습니다. 이 때문에 참조 카운트 위치에 있는 메모리가 다른 값으로 덮어 써어질 수도 있습니다.

```
int CalculateChon(Member* mem1, Member* mem2);  
};
```

함수 객체

안녕하세요 여러분! 지난 강좌에서 배우신 스마트 포인터 삼형제 (`unique_ptr`, `shared_ptr`, `weak_ptr`) 들은 어떠셨나요? 스마트 포인터를 도입함으로써 C++에서 메모리 제어를 훨씬 더 쉽게 수행할 수 있습니다.

이번 강좌에서는 C++에서 호출 가능한 모든 것을 포괄해서 나타내는 *Callable*에 대해서 알아보도록 하겠습니다. 이번 강좌는 다음 강좌에서 쓰레드를 배우기 전에 짧게 거쳐가는 징검다리라고 보시면 됩니다.

Callable

Callable 이란, 이름 그대로 나타내듯이 호출(Call) 할 수 있는 모든 것을 의미합니다. 대표적인 예시로 함수를 들 수 있겠지요.

하지만 C++에서는 ()를 붙여서 호출할 수 있는 모든 것을 *Callable*이라고 정의합니다. 예를 들어서

```
#include <iostream>

struct S {
    void operator()(int a, int b) { std::cout << "a + b = " << a + b << std::endl; }
};

int main() {
    S some_obj;
    some_obj(3, 5);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a + b = 8
```

와 같이 나옵니다. 그렇다면 여기서 `same_obj` 는 함수 일까요? 아닙니다. 하지만 `same_obj` 클래스 S 의 객체이죠. 하지만, `same_obj` 는 마치 함수처럼 () 를 이용해서 호출할 수 있습니다. (실제로는 `same_obj.operator()(3, 5)` 를 한 것이죠.)

또 다른 예시로 람다 함수를 생각해봅시다.

```
#include <iostream>

int main() {
    auto f = [] (int a, int b) { std::cout << "a + b = " << a + b << std::endl; };
    f(3, 5);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a + b = 8
```

와 같이 나옵니다. f 역시 일반적인 함수의 꼴을 하고 있지는 않지만, () 를 통해서 호출할 수 있기에 Callable 이라 할 수 있습니다.

std::function

C++ 에서는 이러한 Callable 들을 객체의 형태로 보관할 수 있는 `std::function` 이라는 클래스를 제공합니다. C 에서의 함수 포인터는 진짜 함수들만 보관할 수 있는 객체라고 볼 수 있다면 이 `std::function` 의 경우 함수 뿐만이 아니라 모든 Callable 들을 보관할 수 있는 객체입니다. 이 `std::function` 을 어떻게 사용할 수 있는지 아래의 예시를 통해 보겠습니다.

```
#include <functional>
#include <iostream>
#include <string>

int some_func1(const std::string& a) {
    std::cout << "Func1 호출! " << a << std::endl;
    return 0;
}
```

```

struct S {
    void operator()(char c) { std::cout << "Func2 호출! " << c << std::endl; }

};

int main() {
    std::function<int(const std::string&)> f1 = some_func1;
    std::function<void(char)> f2 = S();
    std::function<void()> f3 = []() { std::cout << "Func3 호출! " << std::endl; };

    f1("hello");
    f2('c');
    f3();
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Func1 호출! hello
Func2 호출! c
Func3 호출!

```

와 같이 나옵니다.

```

std::function<int(const string&)> f1 = some_func1;
std::function<void(char)> f2 = S();
std::function<void()> f3 = []() { std::cout << "Func3 호출! " << std::endl; };

```

일단 위와 같이 `function` 객체를 정의하는 부분부터 살펴봅시다. `function` 객체는 템플릿 인자로 전달 받을 함수의 타입을 갖게 됩니다. 여기서 함수의 타입이라 하면, 리턴값과 함수의 인자들을 말합니다.

따라서 예를 들어 `some_func1`의 경우 `int`를 리턴하며, 인자로 `const string&`을 받기 때문에 위와 같이 `std::function<int(const string&)>`의 형태로 정의 됩니다.

한편 `Functor` 인 클래스 `S`의 객체의 경우 단순히 `S`의 객체를 전달해도 이를 마치 함수 인자로 받게 됩니다. `S`의 경우 `operator()` 가 인자로 `char`을 받고 리턴타입이 `void` 이므로 `std::function<void(char)>` 의 꼴로 표현할 수 있게 됩니다.

마지막으로 람다 함수의 경우 마찬가지로 리턴값이 없고 인자를 받지 않기 때문에 `std::function<void()>` 로 정의되겠지요.

이렇든 `std::function` 은 C++ 의 모든 Callable 을 마음대로 보관할 수 있는 유용한 객체입니다. 만약에 함수 포인터로 이를 구현하려고 했다면 Functor 와 같은 경우를 성공적으로 보관할 수 없었겠지요.

멤버 함수를 가지는 `std::function`

앞서 `function` 은 일반적인 Callable 들을 쉽게 보관할 수 있었지만, 멤버 함수들의 경우 이야기가 조금 달라집니다. 왜냐하면, 멤버 함수 내에서 `this` 의 경우 자신을 호출한 객체를 의미하기 때문에, 만일 멤버 함수를 그냥 `function` 에 넣게 된다면 `this` 가 무엇인지 알 수 없는 문제가 발생하게 됩니다.

아래의 예시를 보실까요.

```
#include <functional>
#include <iostream>
#include <string>

class A {
    int c;

public:
    A(int c) : c(c) {}
    int some_func() { std::cout << "내부 데이터 : " << c << std::endl; }
};

int main() {
    A a(5);
    std::function<int()> f1 = a.some_func;
}
```

컴파일 한다면 아래와 같은 컴파일 오류가 나게 됩니다.

컴파일 오류

```
test2.cc: In function 'int main()':
test2.cc:17:26: error: invalid use of non-static member function
  ↵ 'int A::some_func()'
    std::function<int()> f1 = a.some_func;
                           ~~~^~~~~~
test2.cc:10:9: note: declared here
    int some_func() {
        ^~~~~~
```

왜냐하면 `f1` 을 호출하였을 때, 함수의 입장에서 자신을 호출하는 객체가 무엇인지 알 길이 없기 때문에 `c` 를 참조 하였을 때 어떤 객체의 `c` 인지를 알 수 없겠지요. 따라서 이 경우 `f1` 에 `a` 에 관한 정보도 추가로 전달해야 합니다.

그렇다면 이를 어떻게 할까요? 사실 멤버 함수들은 구현 상 자신을 호출한 객체를 인자로 암묵적으로 받고 있었습니다.¹⁾

따라서 이를 받는 `function` 은 아래와 같은 형태로 나타나야 합니다.

```
#include <functional>
#include <iostream>
#include <string>

class A {
    int c;

public:
    A(int c) : c(c) {}
    int some_func() {
        std::cout << "비상수 함수: " << ++c << std::endl;
        return c;
    }

    int some_const_function() const {
        std::cout << "상수 함수: " << c << std::endl;
        return c;
    }

    static void st() {}
};

int main() {
    A a(5);
    std::function<int(A&)> f1 = &A::some_func;
    std::function<int(const A&)> f2 = &A::some_const_function;

    f1(a);
    f2(a);
}
```

성공적으로 컴파일 하였다면

실행 결과

비상수 함수: 6

1) 혹시라도 파이썬을 써보신 분들은 알겠지만, 멤버 함수들이 `def func(self)` 이런 식으로 정의되는 것과 마찬가지입니다.

상수 함수: 6

와 같이 나옵니다.

```
std::function<int(A&)> f1 = &A::some_func;
std::function<int(const A&)> f2 = &A::some_const_function;
```

위와 같이 원래 인자에 추가적으로 객체를 받는 인자를 전달해주면 됩니다. 이 때 상수 함수의 경우 당연히 상수 형태로 인자를 받아야 하고 (`const A&`), 반면에 상수 함수가 아닌 경우 단순히 `A&`의 형태로 인자를 받으면 되겠습니다.

참고로 이전의 함수들과는 다르게 `&A::some_func` 와 같이 함수의 이름만으로는 그 주소값을 전달할 수 없습니다. 이는 C++ 언어 규칙에 때문에 그런데, 멤버 함수가 아닌 모든 함수들의 경우 함수의 이름이 함수의 주소값으로 암시적 변환이 일어나지만, 멤버 함수들의 경우 암시적 변환이 발생하지 않으므로 `&` 연산자를 통해 명시적으로 주소값을 전달해줘야 합니다.

따라서 아래와 같이 호출하고자 하는 객체를 인자로 전달해주면 마치 해당 객체의 멤버 함수를 호출한 것과 같은 효과를 낼 수 있습니다.

```
f1(a);
f2(a);
```

위와 같이 말이지요.

멤버 함수들을 함수 객체로 - `mem_fn`

예를 들어서 `vector` 들을 가지는 `vector` 가 있을 때, 각각의 `vector` 들의 크기들을 벡터로 만들어주는 코드를 생각해봅시다.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
using std::vector;

int main() {
    vector<int> a(1);
    vector<int> b(2);
    vector<int> c(3);
    vector<int> d(4);

    vector<vector<int>> container;
    container.push_back(b);
```

```

container.push_back(d);
container.push_back(a);
container.push_back(c);

vector<int> size_vec(4);
std::transform(container.begin(), container.end(), size_vec.begin(),
    &vector<int>::size);
for (auto itr = size_vec.begin(); itr != size_vec.end(); ++itr) {
    std::cout << "벡터 크기 :: " << *itr << std::endl;
}
}
}

```

`transform` 함수는 `<algorithm>` 라이브러리에 있는 함수인데, 각 원소들에 대해 인자로 전달된 함수를 실행시킨 다음 그 결과를 전달된 컨테이너에 넣어줍니다. 함수 정의를 살짝 살펴보면 아래와 같습니다.

```

template <class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first,
    UnaryOperation unary_op) {
    while (first1 != last1) {
        *d_first++ = unary_op(*first1);
        first1++;
    }
    return d_first;
}

```

여기서 문제는 해당 함수를 아래와 같이 호출한다는 점입니다.

```
*d_first++ = unary_op(*first1);
```

`unary_op` 가 멤버 함수가 아닐 경우 위와 같이 호출해도 괜찮습니다. 하지만 문제는 `unary_op` 가 멤버함수 일 경우입니다.

사실 위 코드를 컴파일 하면 아래와 같은 컴파일 오류가 나게 됩니다.

컴파일 오류

```
In file included from /usr/include/c++/7/algorithm:62:0,
                 from test2.cc:5:
```

```
/usr/include/c++/7/bits/stl_algo.h: In instantiation of '_OIter
→ std::transform(_IIter, _IIter, _OIter, _UnaryOperation) [with
→ _IIter = __gnu_cxx::__normal_iterator<std::vector<int>*>;
→ std::vector<std::vector<int> > >; _OIter =
→ __gnu_cxx::__normal_iterator<int*, std::vector<int> >;
→ _UnaryOperation = long unsigned int (std::vector<int>::*)()
→ const noexcept]':
test2.cc:21:85:   required from here
/usr/include/c++/7/bits/stl_algo.h:4306:24: error: must use '.*'
→ or '->*' to call pointer-to-member function in '__unary_op
→ (...)', e.g. '(... ->* __unary_op) (...)'
*__result = __unary_op(*__first);
~~~~~^~~~~~
```

왜 그럴까요? 이 역시 전달된 `size` 함수가 멤버 함수여서 발생하는 문제입니다. 위 템플릿에 `&vector<int>::size` 가 들어간다면 해당 `unary_op` 를 호출하는 부분은 아래와 같이 변환됩니다.

```
unary_op(*first1);
```

가

```
&vector<int>::size(*first);
```

꼴로 되는데, 멤버 함수의 경우

```
(*first).(*&vector<int>::size)
```

혹은

```
first->(*&vector<int>::size)
```

와 같이 호출해야 하기 때문입니다. (이는 C++ 의 규칙이라 생각하시면 됩니다. 위 컴파일러 오류 메세지를 읽어보세요!) 따라서 이를 위해서는 제대로 `std::function` 으로 변환해서 전달해줘야 합니다.

```
#include <algorithm>
#include <functional>
```

```

#include <iostream>
#include <vector>
using std::vector;

int main() {
    vector<int> a(1);
    vector<int> b(2);
    vector<int> c(3);
    vector<int> d(4);

    vector<vector<int>> container;
    container.push_back(a);
    container.push_back(b);
    container.push_back(c);
    container.push_back(d);

    std::function<size_t(const vector<int>&)> sz_func = &vector<int>::size;

    vector<int> size_vec(4);
    std::transform(container.begin(), container.end(), size_vec.begin(), sz_func);
    for (auto itr = size_vec.begin(); itr != size_vec.end(); ++itr) {
        std::cout << "벡터 크기 :: " << *itr << std::endl;
    }
}
}

```

성공적으로 컴파일 했다면

실행 결과

```

벡터 크기 :: 1
벡터 크기 :: 2
벡터 크기 :: 3
벡터 크기 :: 4

```

와 같이 잘 나옴을 알 수 있습니다.

하지만 매번 위처럼 `function` 객체를 따로 만들어서 전달하는 것은 매우 귀찮습니다. 따라서 C++ 개발자들은 라이브러리에 위 `function` 객체를 리턴해버리는 함수를 추가하였습니다.

```

#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
using std::vector;

int main() {
    vector<int> a(1);

```

```
vector<int> b(2);
vector<int> c(3);
vector<int> d(4);

vector<vector<int>> container;
container.push_back(a);
container.push_back(b);
container.push_back(c);
container.push_back(d);

vector<int> size_vec(4);
transform(container.begin(), container.end(), size_vec.begin(),
         std::mem_fn(&vector<int>::size));
for (auto itr = size_vec.begin(); itr != size_vec.end(); ++itr) {
    std::cout << "벡터 크기 :: " << *itr << std::endl;
}
}
```

성공적으로 컴파일 하였다면

실행 결과

```
벡터 크기 :: 1
벡터 크기 :: 2
벡터 크기 :: 3
벡터 크기 :: 4
```

와 같이 잘 나옵니다. `mem_fn` 함수는 이름 그대로, 전달된 멤버 함수를 `function` 객체로 예쁘게 만들어서 리턴해줍니다.

주의 사항

참고로 `mem_fn`은 그리 자주 쓰이지는 않는데, 람다 함수로도 동일한 작업을 수행할 수 있기 때문입니다. 위 코드의 경우 `mem_fn(&vector<int>::size)` 대신에 `[](const auto& v){ return v.size();}`를 전달해도 동일한 작업을 수행합니다.

`mem_fn`을 사용하기 위해서는 `<functional>` 헤더를 추가해야 하지만 람다함수는 그냥 쓸 수 있으니 좀 더 편리한 면이 있습니다. 물론, 코드 길이 면에서는 `mem_fn`을 사용하는 것이 좀 더 깔끔한 편입니다.

std::bind

재미있게도 함수 객체 생성 시에 인자를 특정한 것으로 지정할 수 도 있습니다. 아래 예제를 보실까요.²⁾

```
#include <functional>
#include <iostream>

void add(int x, int y) {
    std::cout << x << " + " << y << " = " << x + y << std::endl;
}

void subtract(int x, int y) {
    std::cout << x << " - " << y << " = " << x - y << std::endl;
}
int main() {
    auto add_with_2 = std::bind(add, 2, std::placeholders::_1);
    add_with_2(3);

    // 두 번째 인자는 무시된다.
    add_with_2(3, 4);

    auto subtract_from_2 = std::bind(subtract, std::placeholders::_1, 2);
    auto negate =
        std::bind(subtract, std::placeholders::_2, std::placeholders::_1);

    subtract_from_2(3);    // 3 - 2 를 계산한다.
    negate(4, 2);          // 2 - 4 를 계산한다
}
```

성공적으로 컴파일 하였다면

실행 결과

```
2 + 3 = 5
2 + 3 = 5
3 - 2 = 1
2 - 4 = -2
```

와 같이 나옵니다.

`bind` 함수는 이름 그대로 원래 함수에 특정 인자를 붙여(bind) 줍니다. 예를 들어서

2) 참고로 자바스크립트를 배우신 분들은 알겠지만 자바스크립트에도 똑같은 작업을 수행하는 `bind` 함수가 있습니다 :)

```
std::bind(add, 2, std::placeholders::_1);
```

위 예시의 경우 `add`라는 함수에 첫 번째 인자로 2를 `bind` 시켜주고, 두 번째 인자로는 새롭게 만들어진 함수 객체의 첫 번째 인자를 전달해줍니다. 따라서;

```
add_with_2(3);
```

를 하였을 때, 원래 `add` 함수의 첫 번째 인자로는 2가 들어가게 되고, 두 번째 인자로는 `add_with_2`의 첫 번째 인자인 3이 들어가겠지요. 만약에

```
add_with_2(3, 4);
```

처럼 인자를 여러개 전달하더라도 뒤에 것들은 무시 됩니다.

```
auto negate = std::bind(subtract, std::placeholders::_2, std::placeholders::_1);
```

위 경우는 어떨까요? `negate` 함수는 첫 번째 인자와 두 번째 인자의 순서를 바꿔서 `subtract` 함수를 호출하게 됩니다. 즉 `negate(3, 5)`를 호출할 경우 실제로는 `subtract(5, 3)`이 호출되겠지요.

`placeholders`의 `_1`, `_2`들은 일일히 정의된 객체들입니다. 그 개수는 라이브러리마다 다른데, `libstdc++`의 경우 (`g++`에서 사용하는 C++ 라이브러리입니다.) `_1`부터 `_29`까지 정의되어 있습니다.³⁾

한 가지 주의할 점은, 레퍼런스를 인자로 받는 함수들의 경우입니다.

```
#include <functional>
#include <iostream>

struct S {
    int data;
    S(int data) : data(data) { std::cout << "일반 생성자 호출!" << std::endl; }
    S(const S& s) {
        std::cout << "복사 생성자 호출!" << std::endl;
        data = s.data;
    }
    S(S&& s) {
        std::cout << "이동 생성자 호출!" << std::endl;
    }
}
```

3) 인자를 30개 보다 많이 받는 함수들의 경우 `bind`를 제대로 사용할 수 없겠지만, 인자를 30개 받는 함수를 만들었다는 사실은 무언가 코드를 잘못 짰다는 뜻이기도 합니다 :)

```

        data = s.data;
    }
};

void do_something(S& s1, const S& s2) { s1.data = s2.data + 3; }

int main() {
    S s1(1), s2(2);

    std::cout << "Before : " << s1.data << std::endl;

    // s1 이 그대로 전달된 것이 아니라 s1 의 복사본이 전달됨!
    auto do_something_with_s1 = std::bind(do_something, s1, std::placeholders::_1);
    do_something_with_s1(s2);

    std::cout << "After :: " << s1.data << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

일반 생성자 호출!
일반 생성자 호출!
Before : 1
복사 생성자 호출!
After :: 1

```

와 같이 나옵니다.

보시다시피 do_something 함수의 경우 첫 번째 인자의 data 를 두 번째 인자의 data + 3 으로 만들어주지만, 실제로 do_something_with_s1 함수를 실행하였을 때 첫 번째 인자로 s1 을 전달했음에도 불구하고 s1 의 data 가 바뀌지 않음을 알 수 있습니다.

그 이유는 위 생성자 호출 메세지에서 확인할 수 있듯이 bind 함수로 인자가 복사 되서 전달되기 때문입니다. 따라서 이를 해결 하기 위해서는 명시적으로 s1 의 레퍼런스를 전달해줘야 합니다.

```

#include <functional>
#include <iostream>

struct S {
    int data;
    S(int data) : data(data) { std::cout << "일반 생성자 호출!" << std::endl; }
    S(const S& s) {
        std::cout << "복사 생성자 호출!" << std::endl;
        data = s.data;
    }
};

```

```

}

S(S&& s) {
    std::cout << "이동 생성자 호출!" << std::endl;
    data = s.data;
}
;

void do_something(S& s1, const S& s2) { s1.data = s2.data + 3; }

int main() {
    S s1(1), s2(2);

    std::cout << "Before : " << s1.data << std::endl;

    // s1 이 그대로 전달된 것이 아니라 s1 의 복사본이 전달됨!
    auto do_something_with_s1 =
        std::bind(do_something, std::ref(s1), std::placeholders::_1);
    do_something_with_s1(s2);

    std::cout << "After :: " << s1.data << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

일반 생성자 호출!
일반 생성자 호출!
Before : 1
After :: 5

```

와 같이 실제로 s1의 값이 잘 바뀌었음을 알 수 있습니다. `ref` 함수는 전달받은 인자를 복사 가능한 레퍼런스로 변환해줍니다. 따라서 `bind` 함수 안으로 s1의 레퍼런스가 잘 전달 될 수 있게 됩니다.

참고로 `const` 레퍼런스의 경우 `cref` 함수를 호출하면 됩니다.

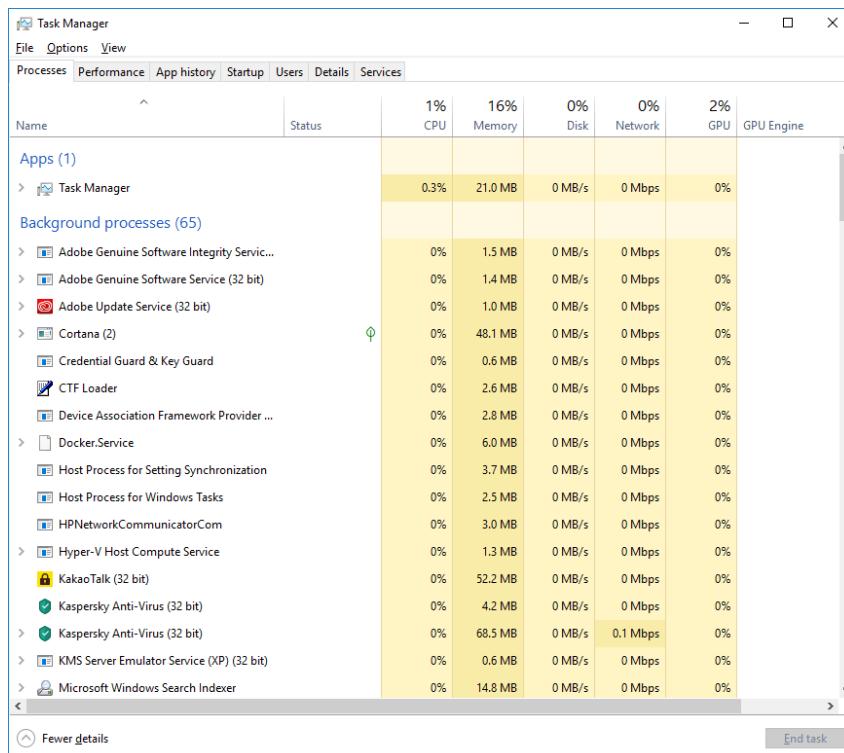
그럼 이것으로 이번 강좌를 마치도록 하겠습니다. `function`, `mem_fn`, `bind`들을 적재 적소에 잘 쓴다면, C++의 강력한 라이브러리를 좀 더 풍요롭게 사용할 수 있을 것입니다.

C++ 쓰레드

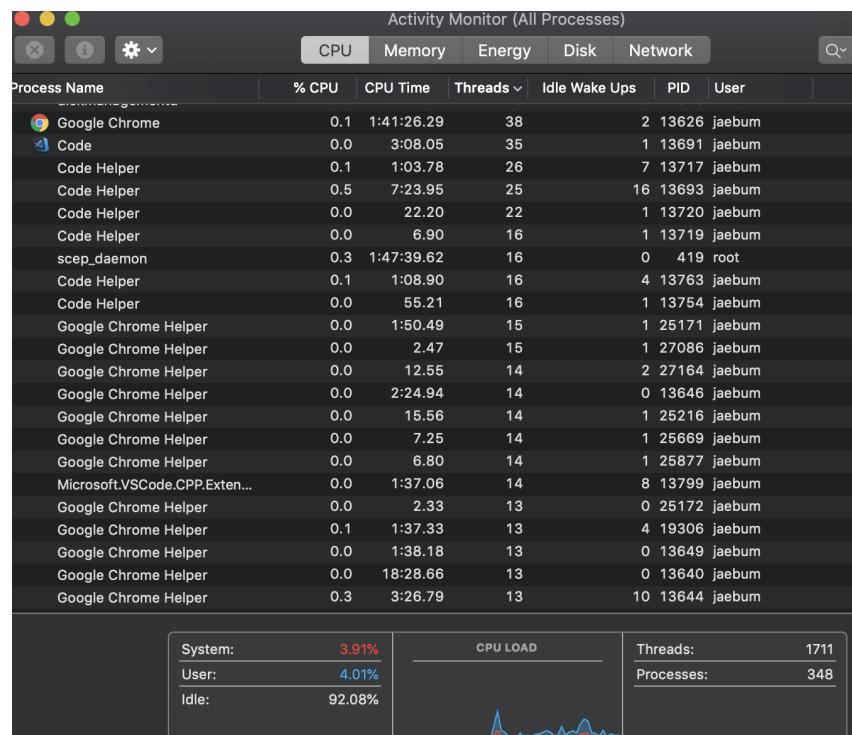
안녕하세요 여러분! 이번 강좌에서는 여태까지 작성하였던 프로그램과 차원이 다른 프로그램을 만들어볼 것입니다.

멀티 쓰레드 프로그램

여러분은 작업 관리자 를 실행해 보신적이 있으신가요? 아마 실행했다면, 아래와 같은 화면을 보셨을 것입니다.



윈도우에서 나오는 작업 관리자 모습



맥에서 나오는 작업 관리자 모습

한 가지 눈여겨 볼 점은, 막대한 개수의 프로세스의 개수입니다. 프로세스란, 운영체제에서 실행되는 프로그램의 최소 단위라고 보시면 됩니다. 즉, 우리가 1 개의 프로그램을 가리킬 때 보통 1 개의 프로세스를 의미하는 경우가 많습니다.¹⁾

그렇다면 이 프로세스들은 어디에서 실행될까요? 바로 컴퓨터의 두뇌라 하는 CPU의 코어(연산하는 부분)에서 실행되고 있습니다. 옛날(2005년 이전)에는 서버용이 아닌 일반 소비자용 CPU의 경우 1 개의 코어를 가지는 것이 대부분이었습니다. 대표적으로 펜티엄 4가 있지요. 이 말은 즉슨, CPU가 한 번에 한 개의 연산을 수행한다는 것입니다.

근데 CPU가 한 번에 한 가지 연산 밖에 못한다면, 도대체 그 시절에는 인터넷을 하면서 음악을 듣고, 아니면 게임을 하는 등 여러가지 일들을 어떻게 한꺼번에 하였을까요? 분명히 제 기억에는 이러한 일들이 가능했던 것 같기 때문이지요. 그 비밀은 컨텍스트 스위칭(Context switching)이라는 기술에 숨어 있습니다.

컴퓨터에서 프로그램이 실행될 때 겉으로 보기에는 프로그램이 연속적으로 주르륵 작동하는 것처럼 보이지만 실제로는 그렇지 않습니다. 아래 그림을 보면 CPU 코어 하나에서 프로그램들이 어떻게 실행되는지 알 수 있습니다.

1) 물론 구글 크롬처럼 한 개의 탭이 한 개의 프로세스를 차지해서, 프로그램 자체가 여러개의 프로세스로 이루어진 경우도 있습니다.



코어 하나에서 프로그램들의 실행 모습

보시다시피, 프로그램 하나가 주르륵 작동하는 것이 아니라, 프로그램 하나가 잠시 실행되었다가, 다른 프로그램으로 스위칭 되는 것을 볼 수 있습니다. 즉, CPU는 한 프로그램을 통째로 쭉 실행시키는 것이 아니라, 이 프로그램 조금, 저 프로그램 조금씩 골라서 차례를 돌며 실행시킨다는 것을 알 수 있습니다.

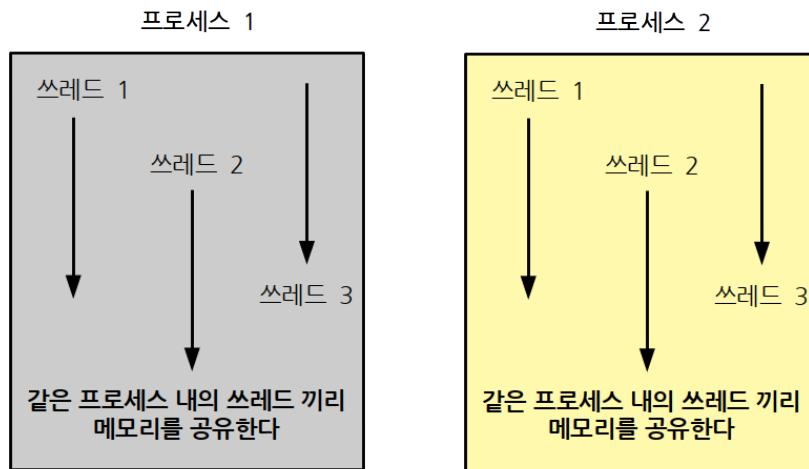
정확히 말하자면, CPU는 그냥 운영체제가 처리하라고 시키는 명령어들을 실행할 뿐, 어떤 프로그램을 실행시키고, 얼마 동안 실행시키고, 또 다음에 무슨 프로그램으로 스위치 할지는 운영체제의 스케줄러(scheduler) 알아서 결정하게 됩니다.

쓰레드

한 가지 중요한 점은, 이 CPU 코어에서 돌아가는 프로그램 단위를 쓰레드 (thread) 라고 부릅니다. 즉, CPU의 코어 하나에서는 한 번에 한 개의 쓰레드의 명령을 실행시키게 됩니다.

한 개의 프로세스는 최소 한 개 쓰레드로 이루어져 있으며, 여러 개의 쓰레드로 구성될 수 있게 됩니다. 이렇게 여러개의 쓰레드로 구성된 프로그램을 멀티 쓰레드 (multithread) 프로그램이라 합니다.

쓰레드와 프로세스의 가장 큰 차이점은 프로세스들은 서로 메모리를 공유하지 않습니다. 다시 말해, 프로세스 1과 프로세스 2가 있을 때, 프로세스 1은 프로세스 2의 메모리에 접근할 수 없고, 마찬가지로 프로세스 2도 프로세스 1의 메모리에 접근할 수 없습니다.



프로세스는 서로의 메모리를 접근할 수 없지만, 같은 프로세스 내에 쓰레드끼리는 메모리를 공유한다

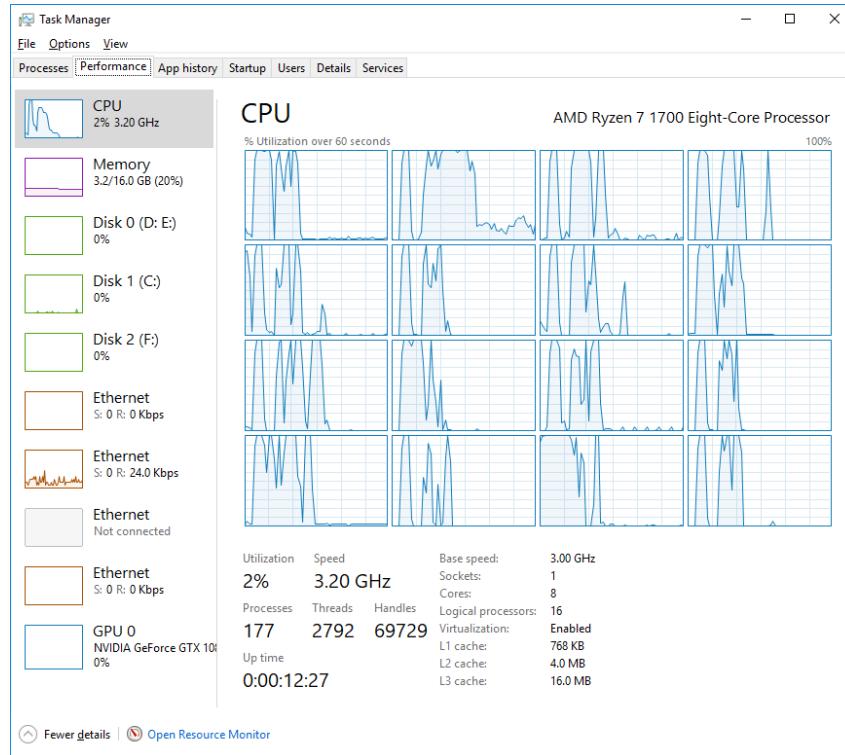
하지만 쓰레드의 경우는 다릅니다. 만일 한 프로세스 안에 쓰레드 1과 쓰레드 2가 있다면, 서로 같은 메모리를 공유하게 됩니다. 예컨대, 쓰레드 1과 쓰레드 2가 같은 변수에 값에 접근할 수 있습니다.

여태까지 여러분이 작성 하였던 프로그램들은 모두 한 개의 쓰레드로 구성된 싱글 쓰레드 프로그램입니다. 하지만, 많은 프로그램들이 멀티쓰레드 프로그램으로 구성되어 있는데, 맥에서 나오는 작업 관리자 사진의 우측 하단에, 현재 시스템의 쓰레드 개수와 프로세스에서도 알 수 있듯이, 프로세스 개수는 348 개 인데, 총 쓰레드 수는 1711 개로 써잇습니다. 대량 프로세스 하나당 5 개의 쓰레드들로 구성되어 있다고 생각하면 되겠네요.

CPU 의 코어는 한 개가 아니다.

요 근래 들어서는 CPU 의 발전 방향이 코어 하나의 동작 속도를 높이기 보다는, CPU 에 장착된 코어 개수를 늘려가는 식으로 발전해왔습니다.

예를 들어서 인텔의 i5 모델의 경우 4 개의 코어가 장착되어 있습니다. 제가 사용하는 AMD 의 라이젠 모델의 경우 아래 그림과 같이 8 개의 코어를 가지고 있습니다. 참고로 SMT 라는 기술을 통해서 마치 16 개의 코어인 것 처럼 보이지만 하단에 **Cores** 를 보면 실제로는 8 개의 코어만 있다는 점을 확인할 수 있습니다.



실제론 8 코어 CPU 이지만, SMT를 통해서 16개인 것 처럼 보입니다.

그렇다면 실제 이 8 개의 코어에서 프로그램이 실행되는 모습은 아래와 같을 것입니다.



여러 코어들에서 쓰레드들이 실행되는 모습

따라서 이전에 싱글 코어 CPU 에서 아무리 멀티 쓰레드 프로그램이라 하더라도 결국에는 한 번에 한 쓰레드만 실행할 수 있었겠지만, 멀티 코어 CPU 에서는 여러개의 코어에 각기 다른 쓰레드들이 들어가 동시에 여러개의 쓰레드들을 효율적으로 실행할 수 있습니다.

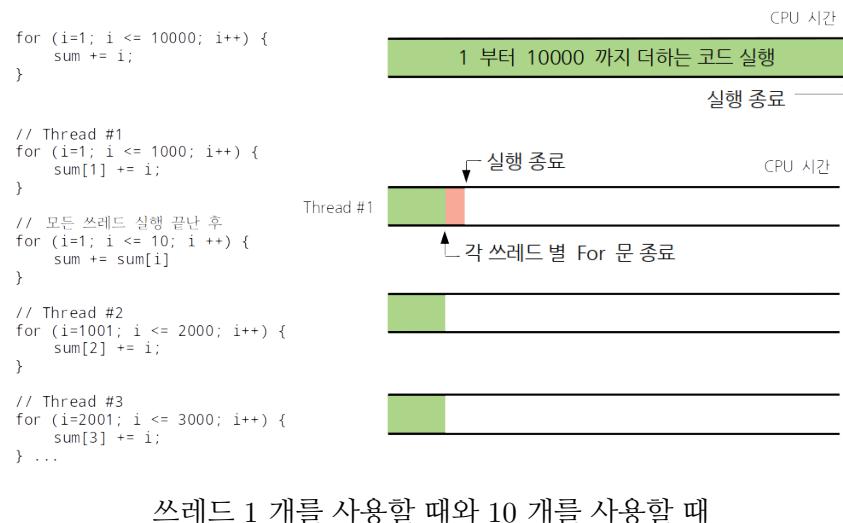
그래서 왜 멀티 쓰레드 인데?

앞서 현대의 CPU 가 여러개의 코어를 지원함으로써 여러개의 쓰레드를 동시에 실행시킬 수 있다고 하였습니다. 그렇다면 어떨 때 프로그램을 멀티 쓰레드로 만드는 것이 유리할까요? 이에 대해 크게 두 가지 이유를 생각할 수 있습니다.

병렬 가능한 (Parallelizable) 작업들

예를 들어서 1 부터 10000 까지 더하는 작업을 생각해봅시다. 만약에 단일 쓰레드 프로그램으로 짠다면 단순히 `for` 문으로 1 부터 10000 까지 더하는 코드를 쓰면 됩니다.

반면에 이를 쓰레드 10 개로 만든다면 어떨까요. 예를 들어서 쓰레드 1에서 1 부터 1000 까지 더하고, 쓰레드 2에서 1001 부터, 2000 까지 더하고, ... 쓰레도 10에서 9001 부터 10000 까지 더하게 한다면 어떨까요? 모든 쓰레드의 작업이 완료된 후에, 각각의 결과를 합치는 식으로 말이지요.



CPU 코어에서 덧셈 한 번에 1 초가 걸린다고 가정해봅시다. 그렇다면 단일 쓰레드의 경우 10000 초가 걸리게 됩니다.

하지만, 멀티 쓰레드를 사용하였을 경우 CPU 에 코어가 10 개가 있어서 각 쓰레드들이 동시에 실행될 수 만 있다면, 각 쓰레드에서 덧셈은 1000 초가 걸리고, 마지막으로 다 합칠 때 10 초가 걸려서 총 1010 초가 걸리게 됩니다.

싱글 쓰레드의 경우보다 속도가 무려 10 배가 향상된 수치입니다!

이렇게, 어떠한 작업을 여러개의 다른 쓰레드를 이용해서 좀 더 빠르게 수행하는 것을 병렬화(parallelize) 라고 합니다. 하지만 모든 작업들이 이렇게 병렬화가 가능한 것이 아닙니다. 예를 들어서 피보나치 수열을 계산하는 프로그램을 생각해봅시다. 아마 아래와 같이 작성할 것입니다.

```

int main() {
    int bef = 1, cur = 1;

```

```
// 물론 100 번째 피보나치 항을 구한다면, int 오버플로우가 나겠지만 일단 그
// 점은 여기서 무시하도록 합시다.
for (int i = 0; i < 98; i++) {
    int temp = cur;
    cur = cur + bef;
    bef = temp;
}
std::cout << "F100 : " << cur << std::endl;
}
```

위와 같은 프로그램을 여러 쓰레드를 사용하는 방식으로 실행 속도를 높일 수 있을까요?

피보나치의 n 번째 항인 F_n 을 계산하기 위해서는 F_{n-1} 과 F_{n-2} 을 알아야 합니다. 다시 말해 F_3 을 구하기 위해서는 F_1 과 F_2 를 알아야 하고, F_4 를 구하기 위해서는 F_3 과 F_2 를 알아야 합니다.

예를 들어서 F_3 을 쓰레드 1, F_4 를 쓰레드 2 에서 계산한다고 생각해봅시다. 쓰레드 2 가 값을 계산하기 위해서는 F_3 의 값이 필요합니다. 그런데, F_3 은 쓰레드 1 에서 계산되고 있으므로, 쓰레드 1 의 연산이 끝날 때 까지 쓰레드 2 가 기다려야 합니다. 따라서 최종 실행 속도는 그냥 쓰레드 1 에서 F_3 과 F_4 모두를 계산하는 것과 차이가 없게 됩니다.

결과적으로 이와 같은 방법으로 피보나치 수열을 계산하는 프로그램은 병렬화 하는 것이 매우 까다롭습니다. 이러한 문제가 발생하는 근본적인 이유는 어떠한 연산 (연산 A) 을 수행하기 위해 다른 연산 (연산 B)의 결과가 필요하기 때문이라 볼 수 있습니다. 이와 같은 상황을 A 가 B 에 의존(dependent) 한다 라고 합니다.

프로그램 논리 구조 상에서 연산들 간의 의존 관계가 많을 수록 병렬화가 어려워지고, 반대로, 다른 연산의 결과와 관계없이 독립적으로 수행할 수 있는 구조가 많을 수록 병렬화가 매우 쉬워집니다.

대기시간이 긴 작업들

인터넷에서 웹사이트들을 긁어 모으는 프로그램을 생각해봅시다. 아마 아래와 같이 구성할 수 있을 것입니다.

```
int main() {
    // 다운 받으려는 웹사이트와 내용을 저장하는 맵
    map<string, string> url_and_content;
    for (auto itr = url_and_content.begin(); itr != url_and_content.end();
         ++itr) {
        const string& url = itr->first;

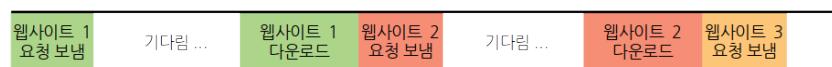
        // download 함수는 인자로 전달받은 url 에 있는 사이트를 다운받아 리턴한다.
        itr->second = download(url);
    }
}
```

이 임의로 만든 `download` 함수는 인자로 전달한 `url`에 위치한 웹사이트를 다운 받아서 리턴합니다.

문제는 우리의 CPU의 처리 속도에 비해 인터넷은 매우 느리다는 점입니다.

우리가 흔히 `ping`이라고 부르는 것은, 내가 보낸 요청이 상대 서버에 도착해서 다시 나에게 돌아오는데 걸리는 시간을 의미 합니다. 보통 우리나라 안에서 웹사이트에 요청을 보낼 시에 `ping`이 30 밀리초 정도 나오고, 해외의 경우 (예컨대 미국), 150 밀리초에서 멀면 300 밀리초 까지 걸리게 됩니다.²⁾

150 밀리초라 한다면 사람 기준에서 얼마 안되는 시간처럼 보입니다. 0.15초 이기 때문이지요. 하지만, 실제로 컴퓨터는 0.15초 동안 정말 많은 일들을 할 수 있습니다. 보통의 CPU는 1초에 10^9 번 연산을 할 수 있기 때문에 0.15초 동안 응답을 단순히 기다리기만 한다면, 1.5×10^8 번 연산을 수행할 수 있는 시간을 버리게 되는 것입니다. 즉 CPU 코어를 비효율적으로 사용하게 되는 셈이지요. 한창 일해야될 CPU를 놀게 놔둔다니요!



쓰레드 1 개 만을 사용할 때

하지만 만일 `download` 함수를 호출하는 것을 여러 쓰레드에서 부르면 어떨까요?



쓰레드 여러개 만을 사용할 때

위 그림은 같은 코어 안에서 쓰레드들이 컨텍스트 스위칭을 통해 기다리는 시간 없이 CPU를 최대한으로 사용하는 것을 볼 수 있습니다. 초록색 쓰레드에서 웹사이트 1에 요청을 보낸 후, 이전에는 웹사이트 1에서 데이터를 다운로드를 시작하기 까지 기다려야 했지만, 이 경우 분홍색 쓰레드로 컨텍스트 스위칭 되어서, 기다리는 시간을 낭비하지 않고 바로 웹사이트 2에 요청을 보내는 것을 볼 수 있습니다.

위와 같이 처리하게 된다면 CPU 시간을 낭비하지 않고 효율적으로 작업을 처리할 수 있게 됩니다.

2) 이 시간은 웹사이트 전체를 다운 받는데 걸리는 시간을 말하는 것이 아닙니다. 내가 다운로드 요청을 보내서, 첫 번째 응답이 돌아올 때 까지 걸리는 시간을 말합니다.

C++에서 쓰레드 생성하기

이전에는 C++ 표준에 쓰레드가 없어서, 각 플랫폼마다 다른 구현을 사용해야만 했습니다. (예를 들어서 윈도우즈에서는 `CreateThread`로 쓰레드를 만들지만 리눅스에서는 `pthread_create`로 만듭니다)

하지만 C++ 11에서부터 표준에 쓰레드가 추가되면서, 쓰레드 사용이 매우 편리해졌습니다.

이제 첫 번째 멀티 쓰레드 프로그램을 만들어보겠습니다.

```
// 내 생애 첫 쓰레드
#include <iostream>
#include <thread>
using std::thread;

void func1() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 1 작동중! \n";
    }
}

void func2() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 2 작동중! \n";
    }
}

void func3() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 3 작동중! \n";
    }
}

int main() {
    thread t1(func1);
    thread t2(func2);
    thread t3(func3);

    t1.join();
    t2.join();
    t3.join();
}
```

성공적으로 컴파일 하였다면 (참고로 리눅스에서 컴파일 하는 분은 컴파일 옵션에 `-pthread`를 추가로 넣어야 합니다.)

실행 결과

```
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 3 작동중!
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 1 작동중!
쓰레드 2 작동중!
쓰레드 2 작동중!
쓰레드 1 작동중!
쓰레드 3 작동중!
쓰레드 2 작동중!
```

와 같이 나옵니다.

C++ 11에서 쓰레드를 생성하는 방법은 매우 간단합니다.

```
#include <thread>
```

일단 위처럼 `thread` 헤더파일을 추가하고,

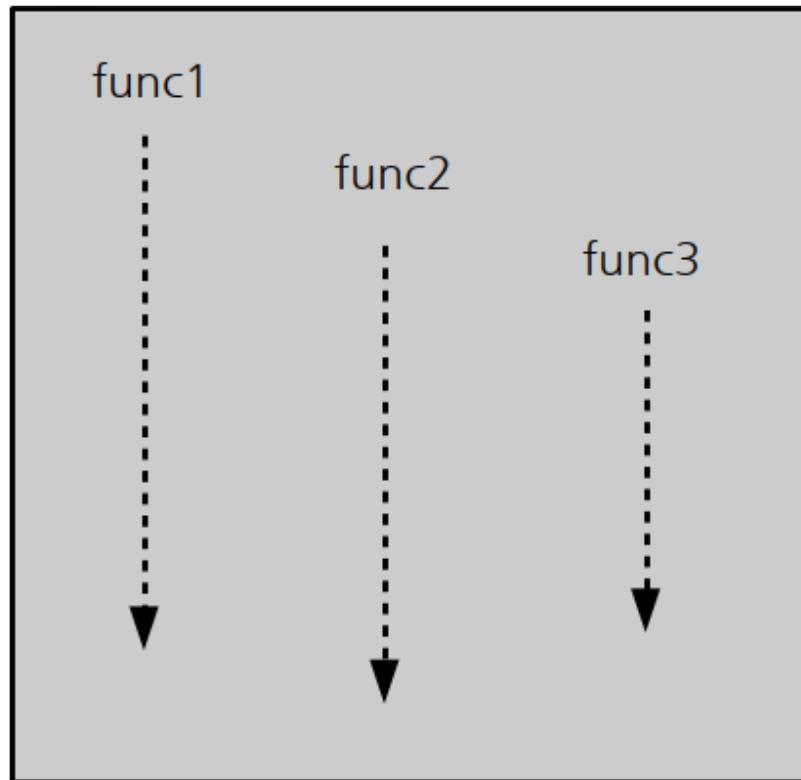
```
thread t1(func1);
```

`thread` 객체를 생성하는 순간 끝입니다. 이렇게 생성된 `t1`은 인자로 전달받은 함수 `func1`을 새로운 쓰레드에서 실행하게 됩니다.

즉

```
thread t1(func1);
thread t2(func2);
thread t3(func3);
```

를 실행하게 되면, `func1`, `func2`, `func3`가 각기 다른 쓰레드 상에서 실행되게 됩니다.



`func1`, `func2`, `func3`가 각기 다른 쓰레드에서 실행된다.

한 가지 중요한 사실은 이 쓰레드들이 CPU 코어에 어떻게 할당되고, 또 언제 컨텍스트 스위치를 할 지는 전적으로 운영체제의 마음에 달려있다는 점입니다.³⁾ 예를 들어서 우리의 실행 결과를 살펴봅시다.

3) 쓰레드 3 개를 만들었다고 해서 반드시 3 개의 각기 다른 코어에 할당되는 것이 아닙니다. 운이 좋으면 그렇게 되겠지만, 그냥 한 코어에 쓰레드 3 개가 컨텍스트 스위칭을 하면서 돌아갈 수도 있습니다.

처음에 쓰레드 1 작동중! 이 조금 나오다가, 쓰레드 3 작동중! 이 나옵니다. 그 다음에 쓰레드 2 작동중! 또 나오다가, 뒤죽박죽 순서가 바뀌어서 나오는 것을 볼 수 있습니다. 한 가지 더 재미있는 점은, 프로그램을 실행 할 때마다 그 결과가 달라진다는 점입니다. 운영체제가 쓰레드들을 어떤 코어에 할당하고, 또 어떤 순서로 스케줄 할지는 그 때 그 때마다 상황에 맞게 바뀌기 때문에 그 결과를 정확히 예측할 수 없습니다.

아무튼 쓰레드 작동중! 메세지를 통해 그때 그때 어떠한 쓰레드의 코드가 출력되는지 짐작할 수 있습니다.

```
t1.join();
t2.join();
t3.join();
```

마지막으로 `join` 은, 해당하는 쓰레드들이 실행을 종료하면 리턴하는 함수입니다. 따라서 `t1.join()` 의 경우 `t1` 이 종료하기 전 까지 리턴하지 않습니다.

그렇다면 만약에 `t2` 가 `t1` 보다 먼저 종료된다면 어떨까요? 상관 없습니다. `t1.join()` 이 끝나고 `t2.join()` 을 하였을 때 쓰레드 `t2` 가 이미 종료된 상태라면 바로 함수가 리턴하게 됩니다.

그렇다면 만약에 `join` 을 하지 않는다면 어떻게 될까요?

```
#include <iostream>
#include <thread>
using std::thread;

void func1() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 1 작동중! \n";
    }
}

void func2() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 2 작동중! \n";
    }
}

void func3() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 3 작동중! \n";
    }
}

int main() {
    thread t1(func1);
    thread t2(func2);
    thread t3(func3);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
terminate called without an active exception
쓰레드 2 작동중!
[1] 1871 abort (core dumped) ./test
```

와 같이 나옵니다. 일단, 보시다시피 쓰레드들의 내용이 채 실행되기 전에 `main` 함수가 종료되어서 쓰레드 객체들 (`t1`, `t2`, `t3`)의 소멸자가 호출되었음을 알 수 있습니다.

C++ 표준에 따르면, `join` 되거나 `detach` 되지 않는 쓰레드들의 소멸자가 호출된다면 예외를 발생시키도록 명시되어 있습니다. 따라서, 우리의 쓰레드 객체들이 `join`이나 `detach` 모두 되지 않았으므로 위와 같은 문제가 발생하게 됩니다.

아, 그렇다면 `detach` 가 무엇일까요? `detach` 는 말 그대로, 해당 쓰레드를 실행 시킨 후, 잊어 버리는 것이라 생각하시면 됩니다. 대신 쓰레드는 알아서 백그라운드에서 돌아가게 됩니다. 아래 예제를 통해 살펴보겠습니다.

```
#include <iostream>
#include <thread>
using std::thread;

void func1() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 1 작동중! \n";
    }
}

void func2() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 2 작동중! \n";
    }
}

void func3() {
    for (int i = 0; i < 10; i++) {
        std::cout << "쓰레드 3 작동중! \n";
    }
}

int main() {
    thread t1(func1);
    thread t2(func2);
    thread t3(func3);

    t1.detach();
    t2.detach();
    t3.detach();
}
```

```
    std::cout << "메인 함수 종료 \n";  
}
```

성공적으로 컴파일 하였다면

실행 결과

메인 함수 종료

혹은

실행 결과

```
쓰레드 1 작동중!  
쓰레드 1 작동중!  
쓰레드 1 작동중!  
쓰레드 2 작동중!  
메인 함수 종료  
쓰레드 3 작동중!  
쓰레드 3 작동중!
```

등등 여러가지 결과가 나옵니다.

기본적으로 프로세스가 종료될 때, 해당 프로세스 안에 있는 모든 쓰레드들은 종료 여부와 상관없이 자동으로 종료됩니다. 즉 `main` 함수에서 메인 함수 종료! 를 출력하고, 프로세스가 종료하게 되면, `func1`, `func2`, `func3` 모두 더 이상 쓰레드 작동중! 을 출력할 수 없게 됩니다.

먼저 첫번째 출력 결과가 왜 저런 식으로 나왔는지 생각해봅시다. 쓰레드를 `detach` 하게 된다면 `main` 함수에서는 더이상 쓰레드들이 종료될 때 까지 기다리지 않습니다.

따라서

```
t1.detach();
t2.detach();
t3.detach();

std::cout << "메인 함수 종료 \n";
```

위 부분이 그냥 주르륵 실행되어서 쓰레드들이 채 문자열을 표시하기도 전에 프로세스가 종료된 것이지요.

반면에 후자의 경우에는 프로세스가 종료되기 전에 운이 좋게도 생성된 쓰레드들에서 적당히 메세지를 출력하고 프로세스가 종료되었습니다. 그래도 쓰레드 1 의 경우 메세지를 3 개 밖에 작성하지 못하고 종료된 것을 볼 수 있습니다.

쓰레드에 인자 전달하기

이번 예제에서는 이전에 이야기한 1 부터 10000 까지의 합을 여러 쓰레드들을 소환해서 빠르게 계산하는 방법을 살펴보도록 하겠습니다.

```
#include <cstdio>
#include <iostream>
#include <thread>
#include <vector>
using std::thread;
using std::vector;

void worker(vector<int>::iterator start, vector<int>::iterator end,
            int* result) {
    int sum = 0;
    for (auto itr = start; itr < end; ++itr) {
        sum += *itr;
    }
    *result = sum;

    // 쓰레드의 id 를 구한다.
    thread::id this_id = std::this_thread::get_id();
    printf("쓰레드 %x 에서 %d 부터 %d 까지 계산한 결과 : %d \n", this_id, *start,
```

```

        *(end - 1), sum);
}

int main() {
    vector<int> data(10000);
    for (int i = 0; i < 10000; i++) {
        data[i] = i;
    }

    // 각 쓰레드에서 계산된 부분 합들을 저장하는 벡터
    vector<int> partial_sums(4);

    vector<thread> workers;
    for (int i = 0; i < 4; i++) {
        workers.push_back(thread(worker, data.begin() + i * 2500,
                                 data.begin() + (i + 1) * 2500, &partial_sums[i]));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    int total = 0;
    for (int i = 0; i < 4; i++) {
        total += partial_sums[i];
    }
    std::cout << "전체 합 : " << total << std::endl;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

쓰레드 a754700 에서 0 부터 2499 까지 계산한 결과 : 3123750
쓰레드 9752700 에서 5000 부터 7499 까지 계산한 결과 : 15623750
쓰레드 9f53700 에서 2500 부터 4999 까지 계산한 결과 : 9373750
쓰레드 8f51700 에서 7500 부터 9999 까지 계산한 결과 : 21873750
전체 합 : 49995000

```

와 같이 나옵니다.

```

void worker(vector<int>::iterator start, vector<int>::iterator end,
            int* result);

```

먼저 `worker` 함수는 덧셈을 수행할 데이터의 시작점과 끝점을 받아서 해당 범위 내의 원소들을 모두 더한 후, 그 결과를 `result`에 저장하게 됩니다.

참고로 쓰레드는 리턴값이란것이 없기 때문에 만일 어떠한 결과를 반환하고 싶다면 포인터의 형태로 전달하면 됩니다.

```
vector<thread> workers;
for (int i = 0; i < 4; i++) {
    workers.push_back(thread(worker, data.begin() + i * 2500,
                             data.begin() + (i + 1) * 2500, &partial_sums[i]));
}
```

다음에 `main` 함수 안에서 각 쓰레드에게 임무를 할당하고 있는 모습입니다. 보시다시피, 각 `worker`들이 덧셈을 수행해야 할 범위는 `data.begin() + i * 2500`, `data.begin() + (i + 1) * 2500` 임을 알 수 있습니다. 즉, 첫 번째 쓰레드는 0 부터 2499 까지, 두 번째 쓰레드는 2500 부터 4999 까지 주르륵 할당하게 됩니다.

쓰레드를 생성할 때 함수에 인자들을 전달하는 방법은 매우 간단합니다. 우리가 이전에 `std::bind`를 사용했던 방법을 떠올리면 됩니다.

```
thread(worker, data.begin() + i * 2500, data.begin() + (i + 1) * 2500,
       &partial_sums[i])
```

`thread` 생성자의 첫번째 인자로 함수 (정확히는 `Callable` 은 다 됩니다) 를 전달하고, 이어서 해당 함수에 전달할 인자들을 주르륵 써주면 됩니다.

자 이제 그렇다면;

```
int sum = 0;
for (auto itr = start; itr < end; ++itr) {
    sum += *itr;
}
*result = sum;
```

실제로 `worker` 함수의 내부를 보면 정확히 해당 범위의 원소들의 덧셈을 수행하고 있음을 알 수 있습니다.

```
thread::id this_id = std::this_thread::get_id();
```

각 쓰레드에는 고유 아이디 번호가 할당 됩니다. 만약에 우리가 지금 어떤 쓰레드에서 작업중인지 보고싶다면 `this_thread::get_id` 함수를 통해서 현재 내가 돌아가고 있는 쓰레드의 아이디를 알 수 있습니다.

```
printf("쓰레드 %x 에서 %d 부터 %d 까지 계산한 결과 : %d \n", this_id, *start,
      *(end - 1), sum);
```

그리고 마지막으로 `printf` 함수를 통해 부분합 결과를 출력해주고 있습니다.

여기서 한 가지 궁금한 점이 있습니다. 왜 난데없이 `printf` 함수를 사용하였을까요?

한 번 위 출력 부분을 그대로 `std::cout` 으로 바꿔서 실행해보도록 하겠습니다.

```
std::cout << "쓰레드 " << hex << this_id << " 에서 " << dec << *start << " 부터 "
<< *(end - 1) << " 까지 계산한 결과 : " << sum << std::endl;
```

로 치환해서 실행한다면 아래와 같이 나옵니다.

실행 결과

```
쓰레드 쓰레드 쓰레드 쓰레드 7f2d6ea5c700 에서 7f2d6f25d700 에서
↪ 7f2d6fa5e70075005000 에서 부터 부터 2500 부터 4999 까지 계산한 결과 :
↪ 93737509999 까지 계산한 결과 : 218737507499 까지 계산한 결과 :
↪ 156237507f2d7025f700 에서
0 부터 2499 까지 계산한 결과 : 3123750
```

전체 합 : 49995000

왜 이런일이 발생하였을까요? 한 번 여러분이 컴퓨터라고 생각하고 위 `std::cout` 명령을 실행한다고 생각해보세요. 만약에 `std::cout << "쓰레드 "` 까지 딱 실행했는데 운영체제가 갑자기 다른 쓰레드를 실행시키면 어떨까요? 그렇다면 화면에는 쓰레드 만 딱 나오고 그 뒤로 다른 쓰레드의 메세지가 표시될 것입니다.

따라서 위와 같이 `std::cout` 의 `<<` 를 실행하는 과정 중간 중간에 계속 실행되는 쓰레드들이 바뀌면서 결과적으로 메세지가 뒤섞여서 나타나게 됩니다.

`std::cout` 의 경우 `std::cout << A;` 를 하게 된다면 A의 내용이 출력되는 동안 중간에 다른 쓰레드가 내용을 출력할 수 없게 보장을 해줍니다 (그 사이에 컨텍스트 스위치가 되더라도 말이지요). 하지만 `std::cout << A << B;` 를 하게 되면 A를 출력한 이후에 B를 출력하기 전에 다른 쓰레드가 내용을 출력할 수 있습니다.

반면에 `printf` 는 조금 다릅니다. `printf` 는 "..." 안에 있는 문자열을 출력할 때, 컨텍스트 스위치가 되더라도 다른 쓰레드들이 그 사이에 메세지를 집어넣지 못하게 막습니다. (자세한 내용은 여기 [참고](#))

따라서, 방해받지 않고 전체 메세지를 제대로 출력할 수 있게 해줍니다.

```
for (int i = 0; i < 4; i++) {
    workers[i].join();
}
```

```

int total = 0;
for (int i = 0; i < 4; i++) {
    total += partial_sums[i];
}

```

마지막으로 `main` 함수에서 위와 같이 모든 쓰레드들이 종료될 때 까지 기다립니다. 각 쓰레드에서 계산한 결과는 `partial_sums`의 각 원소들에 저장되어 있습니다.

모든 쓰레드에서 연산이 끝난 후에, 최종적으로 `main` 함수에서 부분 합들을 모두 더해서 최종 결과를 얻을 수 있겠네요.

앞에서도 이야기 했지만 쓰레드들은 서로 메모리를 공유한다고 하였습니다. 실제로 각 쓰레드들에서 `data` 와 `partial_sums`에 (다른 부분이긴 했지만) 서로 접근할 수 있었습니다.

그렇다면 여기서 궁금한게 있습니다. 만약에, 서로 다른 쓰레드들이, 같은 메모리에 서로 접근하고 데이터를 쓴다면 어떠한 일이 발생할까요?

메모리를 같이 접근한다면?

아래 예제는 서로 다른 쓰레드들에서 `counter`라는 변수의 값을 1씩 계속 증가시키는 연산을 수행합니다.

```

#include <iostream>
#include <thread>
#include <vector>
using std::thread;
using std::vector;

void worker(int& counter) {
    for (int i = 0; i < 10000; i++) {
        counter += 1;
    }
}

int main() {
    int counter = 0;

    vector<thread> workers;
    for (int i = 0; i < 4; i++) {
        // 레퍼런스로 전달하려면 ref 함수로 감싸야 한다 (지난 강좌 bind 함수 참조)
        workers.push_back(thread(worker, std::ref(counter)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }
}

```

```
    }  
  
    std::cout << "Counter 최종 값 : " << counter << std::endl;  
}
```

성공적으로 컴파일 하였다면⁴⁾

실행 결과

```
Counter 최종 값 : 26459
```

흠 결과가 조금 이상하네요? 분명히 각 쓰레드에서 10000 씩 더했기 때문에 정상적인 상황이였다면 40000 이 출력되어야 했을 것입니다. 그런데, 모든 쓰레드들이 종료되고 최종적으로 Counter 에 써진 값은 10000 이 되었습니다.

```
for (int i = 0; i < 10000; i++) {  
    counter += 1;  
}
```

이 부분을 살펴봅시다. 틀림없이 counter 에 1 을 10000 번 더하는 코드 입니다. 그렇다면 counter += 1 이 문제였을까요?

바로 다음 강좌에서 알아보도록 하겠습니다!

생각 해보기

문제 1

피보나치 수열을 멀티 쓰레딩을 활용해서 빠르게 계산할 수 있는 방법은 없을까요?

4) 참고로 컴파일러 최적화를 키면, 위 for 문을 그냥 counter += 10000; 으로 대체해버리는 경우도 있으니, 정확한 효과를 보기 위해서는 컴파일러 최적화 옵션을 꺼야 합니다.

뮤텍스와 조건변수

안녕하세요 여러분!

지난 강좌에서 보았듯이, 서로 다른 쓰레드에서 같은 메모리를 공유할 때 발생할 수 있는 문제를 보았습니다. 이와 같이 서로 다른 쓰레드들이 동일한 자원을 사용할 때 발생하는 문제를 경쟁 상태 (race condition) 이라 부릅니다. 이 경우 counter라는 변수에 race condition이 있었습니다.

Race Condition

그 코드를 다시 가져오면 아래와 같습니다.

```
#include <iostream>
#include <thread>
#include <vector>

void worker(int& counter) {
    for (int i = 0; i < 10000; i++) {
        counter += 1;
    }
}

int main() {
    int counter = 0;

    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++) {
        // 레퍼런스로 전달하려면 ref 함수로 감싸야 한다 (지난 강좌 bind 함수 참조)
        workers.push_back(std::thread(worker, std::ref(counter)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    std::cout << "Counter 최종 값 : " << counter << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

Counter 최종 값 : 26459

왜 이런 문제가 발생하였을까요?

```
counter += 1;
```

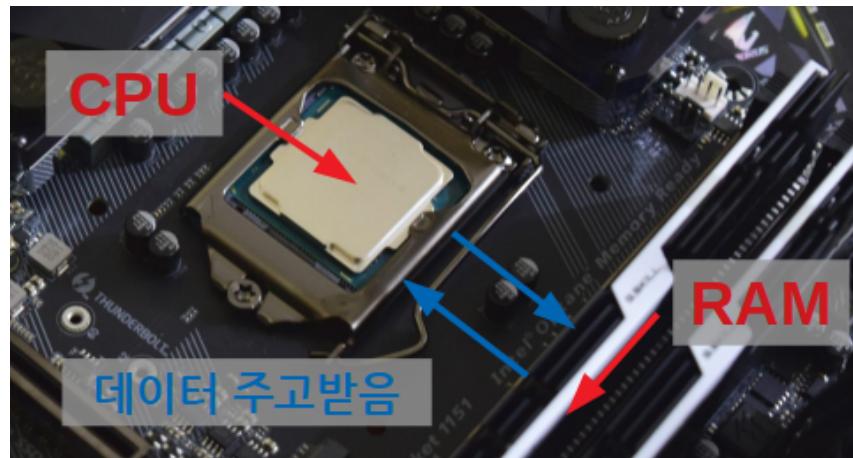
문제는 위 명령에 있습니다. 컴퓨터에 입장에서 생각해봅시다. `counter += 1;` 을 하기 위해서는 어떠한 과정이 필요할까요?

이를 이해하기 위해서는 CPU에서 연산을 어떻게 처리하는지 알아야 합니다.

CPU 간단 소개

CPU는 말했듯이 컴퓨터의 모든 연산이 발생하는 두뇌라고 볼 수 있습니다. CPU에서 연산을 수행하기 위해서는, CPU의 레지스터(register)라는 곳에 데이터를 기록한 다음에 연산을 수행해야 합니다.

레지스터의 크기는 매우 작습니다. 64비트 컴퓨터의 경우, 레지스터의 크기들이 8바이트에 불과합니다. 뿐만 아니라 레지스터의 개수는 그리 많지 않습니다. 일반적인 연산에서 사용되는 범용 레지스터의 경우 불과 16개밖에 없습니다.⁵⁾



메인보드를 보면 CPU 바로 옆에 메모리가 있습니다

즉, 모든 데이터들은 메모리에 저장되어 있고, 연산 할 때 할 때마다 메모리에서 레지스터로 값을 가져온 뒤에, 빠르게 연산을 하고, 다시 메모리에 가져다 놓는 식으로 작동을 한다고 보시면 됩니다.

쉽게 말하자면, 메모리는 냉장고이고 CPU의 레지스터는 도마라고 보시면 됩니다. 냉장고(RAM)에서 재료를 도마 위에 하나(레지스터) 꺼내서 후다닥 썰고(연산) 다시 냉장고로 가져다 놓는 거라 생각하면 됩니다.

그렇다면 `counter += 1` 이 실제로 어떠한 코드로 컴파일되는지 살펴봅시다.

5) 32비트 시절에는 8개밖에 없었지만, x86-64로 넘어오면서 8개가 추가되어 총 16개가 되었습니다.

```
mov rax, qword ptr [rbp - 8] mov ecx, dword ptr [rax] add ecx, 1 mov dword ptr [rax], ecx
```

흠, 조금 무섭게 생겼습니다. 위와 같은 코드를 어셈블리(Assembly) 코드라고 부릅니다. 어셈블리 코드는 CPU 가 실제로 실행하는 기계어와 1 대 1 대응이 되어 있습니다. 따라서, 위 명령을 한줄 한줄 CPU 가 처리한다고 생각해도 무방합니다.

이해하기 매우 어렵게 생겼지만 사실 하나씩 뜯어보면 크게 어렵지 않습니다. 먼저 첫번째 줄부터 살펴봅시다.

```
mov rax, qword ptr [rbp - 8]
```

`rax` 와 `rbp` 모두 CPU 의 레지스터를 의미합니다. `mov` 는 이 문장이 어떤 명령을 하는지 나타내는데, 이름에서도 짐작할 수 있듯이 대입(move) 명령입니다. 즉, `[rbp - 8]` 이 `rax` 에 대입됩니다.

이 때 `[]` 의 의미는 역참조, 즉 `rbp - 8` 을 주소값이라 생각했을 때 해당 주소에 있는 값을 읽어라라는 의미가 되겠습니다. C++ 에서 포인터에 `*` 연산을 하는 것과 동일합니다. 그런데, 이 때 값을 읽기 위해 해당 주소부터 얼마나 읽어야 하는지 명시해야합니다. 이는 `qword` 라는 단어에서 알 수 있는데, `qword` 는 8 바이트를 의미합니다. (주소값의 크기가 8 바이트 이지요!)

즉, C++ 의 언어로 풀어 쓰자면

```
rax = *(int**)(rbp - 8)
```

가 되겠습니다.

실제로 위 명령에서 무슨 짓을 하고 있는 것이냐면 현재 `rbp - 8` 에는 `counter` 의 주소값이 담겨 있어서 `rax` 에 `counter` 의 주소값을 복사하고 있는 과정입니다. 그렇다면 그 아래 문장이 바로 이해가 되시겠지요?

```
mov ecx, dword ptr [rax]
```

현재 `rax` 에는 `result` 의 주소값이 담겨 있습니다. 따라서 `ecx` 에는 `result` 의 현재 값이 들어가게 되니다. 위 문장은

```
ecx = *(int*)(rax); // rax 에는 &result 가 들어가 있음
```

와 동일합니다.

자 이제 그 다음 문장입니다.

```
add ecx, 1
```

언뜻 봐도 알 수 있듯이 ecx 에 1 을 더하는 명령입니다. 즉, result 에 1 이 더해집니다.

```
mov dword ptr [rax], ecx
```

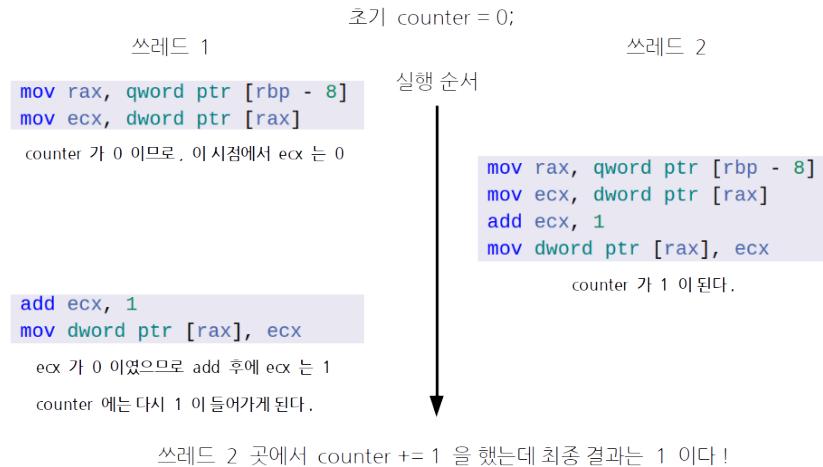
마지막으로 result 에 이전의 result 에 1 이 더해진 값이 저장됩니다.

참고로 ecx 없이

```
mov rax, qword ptr [rbp - 8] add dword ptr [rax], 1
```

이렇게 하면 안되냐고 생각할 수 있는데, 이는 CPU 의 구조상 add 명령은 역참조한 메모리에서 직접 사용할 수 없고 반드시 레지스터에만 내릴 수 있습니다. (냉장고 안에서 직접 요리를 할 수 없으니까요!)

자 그러면, 왜 이제 counter 의 값이 이상하게 나왔는지 짐작하실 수 있나요?



counter += 1 을 두 번 했는데, 결과는 1이 되었다고?

위 그림과 같은 상황을 생각해봅시다. 처음에 counter 가 0 이였다고 가정하고, 쓰레드 1에서

```
mov rax, qword ptr [rbp - 8] mov ecx, dword ptr [rax]
```

딱 여기 까지 실행하였다고 생각해봅시다. 그러면 이 시점에서 쓰레드 1 의 ecx 레지스터에는 counter 의 초기값인 0 이 들어가게 됩니다.

다음에 쓰레드 2에서 전체 명령을 모두 실행합니다. 현재 쓰레드 1 이 counter 의 값을 바꾸지 않은 상태이기 때문에 쓰레드 2에서 읽은 counter 의 값도 역시 0 입니다. 따라서 쓰레드 2가

counter += 1 을 마쳤을 때에는 counter 에는 1 이 들어가 있겠지요.⁶⁾

다시 쓰레드 1 의 차례입니다. 쓰레드 1에서 나머지

```
add ecx, 1 mov dword ptr [rax], ecx
```

부분을 실행하였습니다. 이 때 쓰레드 1의 ecx 는 0 이였으므로, add ecx, 1 후에 ecx 역시 1 이 됩니다. 결국 counter 에는 2 가 아닌 1 이 기록됩니다.

물론 운이 좋다면 쓰레드 1에서 중간에 쓰레드 2가 실행되는 일 없이 쭉 실행해서 정상적으로 counter 에 2가 들어갔을 수도 있습니다. 하지만, 쓰레드를 어떻게 스케줄링 할지는 운영체제가 마음대로 결정하는 것이기 때문에 우리는 그런 행운을 항상 바랄 수 없습니다.

이게 멀티쓰레딩의 재밌는 점입니다. 여태까지 여러분이 실행한 모든 프로그램은 몇 번을 실행 하건 결과가 동일하게 나왔습니다. 하지만, 멀티쓰레드 프로그램의 경우 프로그램 실행 마다 그 결과가 달라질 수 있습니다.

이게 무슨 말일까요? 제대로 프로그램을 만들지 않았을 경우 디버깅이 겁나 어렵다는 뜻입니다.

뮤텍스 (mutex)

그렇다면 위 문제를 어떻게 하면 해결할 수 있을까요? 위 문제가 발생한 근본적인 이유는

```
counter += 1;
```

위 부분을 여러 쓰레드에서 동시에 실행시켰기 때문이지요. 그렇다면 만약에 어떤 경찰관 같은 역할을 하는 것이 있어서, 한 번에 한 쓰레드에서만 위 코드를 실행시킬 수 있다면 어떨까요?

6) 참고로 각 쓰레드는 메모리를 공유할 지언정, 레지스터는 공유하지 않습니다. 따라서 각 쓰레드 별로 고유의 레지스터들을 가지고 있다고 생각하셔도 됩니다. 즉, 쓰레드 1의 ecx 와 쓰레드 2의 ecx 는 서로 다르다고 보시면 됩니다.



쓰레드 한 개만 들어와!

그렇다면 우리가 앞서 말한 문제를 완벽히 해결할 수 있을 것입니다. 그리고 다행이도 C++ 에선 이러한 기능을 하는 객체를 제공하고 있습니다. 바로 뮤텍스(mutex) 라고 불리는 것입니다.

```
#include <iostream>
#include <mutex> // mutex 를 사용하기 위해 필요
#include <thread>
#include <vector>

void worker(int& result, std::mutex& m) {
    for (int i = 0; i < 10000; i++) {
        m.lock();
        result += 1;
        m.unlock();
    }
}

int main() {
    int counter = 0;
    std::mutex m; // 우리의 mutex 객체

    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++) {
        workers.push_back(std::thread(worker, std::ref(counter), std::ref(m)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    std::cout << "Counter 최종 값 : " << counter << std::endl;
```

```
}
```

성공적으로 컴파일 하였다면

실행 결과

Counter 최종 값 : 40000

와 같이 제대로 나오는 것을 알 수 있습니다.

```
std::mutex m; // 우리의 mutex 객체
```

일단 위와 같이 뮤텍스 객체를 정의 하였습니다. `mutex`라는 단어는 영어의 상호 배제 (mutual exclusion)라는 단어에서 따온 단어입니다.⁷⁾

```
void worker(int& result, std::mutex& m)
```

뮤텍스를 각 쓰레드에서 사용하기 위해 위와 같이 전달하였고;

```
m.lock();
result += 1;
m.unlock();
```

실제 사용하는 것은 위와 같습니다.

`m.lock()`은 뮤텍스 `m`을 내가 쓰게 달라!라고 이야기 하는 것입니다. 이 때 중요한 사실은, 한번에 한 쓰레드에서만 `m`의 사용 권한을 갖는다는 것입니다. 그렇다면, 다른 쓰레드에서 `m.lock()`을 하였다면 어떻게 될까요? 이는 `m`을 소유한 쓰레드가 `m.unlock()`을 통해 `m`을 반환할 때 까지 무한정 기다리게 됩니다.

따라서, `result += 1;`은 아무리 많은 쓰레드들이 서로 다른 코어에서 돌아가고 있더라도, 결국 `m`은 한 번에 한 쓰레드만 얻을 수 있기 때문에, `result += 1;`은 결국 한 쓰레드만 유일하게 실행할 수 있게 됩니다.

이렇게 `m.lock()`과 `m.unlock()` 사이에 한 쓰레드만이 유일하게 실행할 수 있는 코드 부분을 임계 영역(critical section)이라고 부릅니다.

만약에 까먹고 `m.unlock()`을 하지 않는다면 어떻게 될까요?

7) 사실 한국말로 해석한 것이 영어로 이해하는 것보다 어려운데, 영어 단어에 뜻을 살펴보자면; mutual - 여러 사람들이 동시에 느끼는 감정; exclusion - 배제하다; 와 같은 뜻입니다. 즉, 여러 쓰레드들이 동시에 어떠한 코드에 접근하는 것을 배제한다는 의미를 담고 있다고 보면 됩니다.

```

#include <iostream>
#include <mutex> // mutex 를 사용하기 위해 필요
#include <thread>
#include <vector>

void worker(int& result, std::mutex& m) {
    for (int i = 0; i < 10000; i++) {
        m.lock();
        result += 1;
    }
}

int main() {
    int counter = 0;
    std::mutex m; // 우리의 mutex 객체

    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++) {
        workers.push_back(std::thread(worker, std::ref(counter), std::ref(m)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    std::cout << "Counter 최종 값 : " << counter << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

(끝나지 않아서 강제 종료)

와 같이 나옵니다. 위와 같이 프로그램이 끝나지 않아서 강제로 종료해야만 합니다.

뮤텍스를 취득한 쓰레드가 `unlock` 을 하지 않으므로, 다른 모든 쓰레드들이 기다리게 됩니다. 심지어 본인도 마찬가지로 `m.lock()` 을 다시 호출하게 되고, `unlock` 을 하지 않았기에 본인 역시 기다리게 되죠.

결국 아무 쓰레드도 연산을 진행하지 못하게 됩니다. 이러한 상황을 데드락(deadlock) 이라고 합니다.

위와 같은 문제를 해결하기 위해서는 취득한 뮤텍스는 사용이 끝나면 반드시 반환을 해야 합니다. 하지만 코드 길이가 길어지게 된다면 반환하는 것을 까먹을 수 있기 마련입니다.

곰곰히 생각해보면 이전에 비슷한 문제를 해결한 기억이 있습니다. `unique_ptr` 를 왜 도입을 하였는지 생각을 해보자면, 메모리를 할당 하였으면 사용 후에 반드시 해제를 해야 하므로, 아예 이 과정을 `unique_ptr` 의 소멸자에서 처리하도록 했었습니다.

뮤텍스도 마찬가지로 사용 후 해제 패턴을 따르기 때문에 동일하게 소멸자에서 처리할 수 있습니다.

```
#include <iostream>
#include <mutex> // mutex 를 사용하기 위해 필요
#include <thread>
#include <vector>

void worker(int& result, std::mutex& m) {
    for (int i = 0; i < 10000; i++) {
        // lock 생성 시에 m.lock() 을 실행한다고 보면 된다.
        std::lock_guard<std::mutex> lock(m);
        result += 1;

        // scope 를 빠져 나가면 lock 이 소멸되면서
        // m 을 알아서 unlock 한다.
    }
}

int main() {
    int counter = 0;
    std::mutex m; // 우리의 mutex 객체

    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++) {
        workers.push_back(std::thread(worker, std::ref(counter), std::ref(m)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    std::cout << "Counter 최종 값 : " << counter << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

Counter 최종 값 : 40000

와 같이 나옵니다.

```
std::lock_guard<std::mutex> lock(m);
```

`lock_guard` 객체는 뮤텍스를 인자로 받아서 생성하게 되는데, 이 때 생성자에서 뮤텍스를 `lock`하게 됩니다. 그리고 `lock_guard` 가 소멸될 때 알아서 `lock` 했던 뮤텍스를 `unlock` 하게 됩니다. 따라서 사용자가 따로 `unlock` 을 신경쓰지 않아도 되서 매우 편리하죠.

그렇다면 `lock_guard` 만 있다면 이제 더이상 데드락 상황은 신경쓰지 않아도 되는 것일까요?

데드락 (deadlock)

아래와 같은 상황을 생각해봅시다.

```
#include <iostream>
#include <mutex> // mutex 를 사용하기 위해 필요
#include <thread>

void worker1(std::mutex& m1, std::mutex& m2) {
    for (int i = 0; i < 10000; i++) {
        std::lock_guard<std::mutex> lock1(m1);
        std::lock_guard<std::mutex> lock2(m2);
        // Do something
    }
}

void worker2(std::mutex& m1, std::mutex& m2) {
    for (int i = 0; i < 10000; i++) {
        std::lock_guard<std::mutex> lock2(m2);
        std::lock_guard<std::mutex> lock1(m1);
        // Do something
    }
}

int main() {
    int counter = 0;
    std::mutex m1, m2; // 우리의 mutex 객체

    std::thread t1(worker1, std::ref(m1), std::ref(m2));
    std::thread t2(worker2, std::ref(m1), std::ref(m2));

    t1.join();
    t2.join();

    std::cout << "끝!" << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

(끝나지 않아서 강제 종료)

와 같이 나옵니다. 위와 같이 프로그램이 끝나지 않아서 강제로 종료해야만 합니다.

왜 이런 일이 발생하였을까요? `worker1` 과 `worker2`에서 뮤텍스를 얻는 순서를 살펴봅시다.

`worker1`에서는

```
std::lock_guard<std::mutex> lock1(m1);
std::lock_guard<std::mutex> lock2(m2);
```

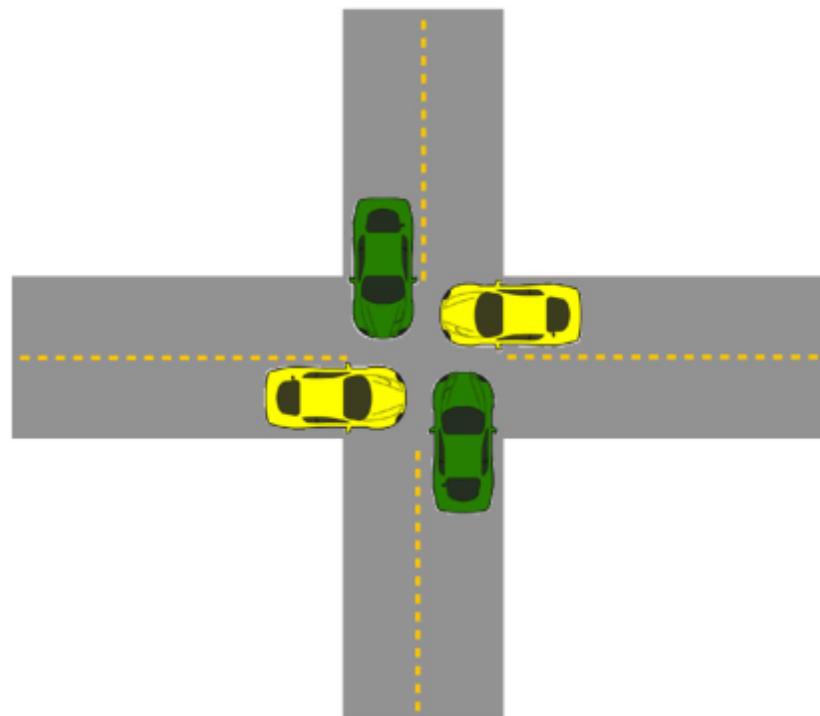
와 같이 `m1` 을 먼저 `lock` 한 후 `m2` 를 `lock` 하게 됩니다. 반면에 `worker2` 의 경우

```
std::lock_guard<std::mutex> lock2(m2);
std::lock_guard<std::mutex> lock1(m1);
```

`m2` 를 먼저 `lock` 한 후 `m1` 을 `lock` 하게 됩니다.

그렇다면 다음과 같은 상황을 생각해보세요. 만약에 `worker1`에서 `m1` 을 `lock` 하고, `worker2`에서 `m2` 를 `lock` 했습니다. 그렇다면 `worker1`에서 `m2` 를 `lock` 할 수 있을까요?

아닙니다. `worker1`에서 `m2` 를 `lock` 하기 위해서는 `worker2`에서 `m2` 를 `unlock` 해야 됩니다. 하지만 그러기 위해서는 `worker2`에서 `m1` 을 `lock` 해야 합니다. 그런데 이 역시 불가능합니다. 왜냐하면 `worker1`에 `m1` 을 `lock` 하고 있기 때문이지요!



데드락은 실생활에도 존재합니다.

즉 `worker1` 과 `worker2` 모두 이러지도 저러지도 못하는 데드락 상황에 빠지게 됩니다. 분명히 뮤텍스를 `lock` 하면 반드시 `unlock` 한다라는 원칙을 지켰음에도 불구하고 데드락을 피할 수 없습니다.

여기에서 보면 데드락이 발생하는 조건이 잘 나타나 있습니다. 물론 만족해야 할 조건이 꽤나 많지만, 일어날 수 있는 일은 반드시 일어나고, 데드락 때문에 디버깅 하는 것 만큼 골때리는 것도 없습니다.

그렇다면 데드락이 가능한 상황을 어떻게 해결할 수 있을까요? 한 가지 방법으로는 한 쓰레드에게 우선권을 주는 것입니다. 위 자동차 그림으로 보자면 초록색 차가 노란색 차보다 항상 먼저 지나가도록 우선권을 주는 것이지요. 만약에 노란색 차가 교차로에 있는데 초록색 차가 들어온다면 초록색 차가 노란색 차에게 "야 차 빼~!"라고 요구할 수도 있지요.

물론 노란색 차는 억울하겠지만, 적어도 차들이 뒤엉켜서 아무도 전진하지 못하는 상황은 막을 수 있습니다. 쓰레드로 비유하자면, 한 쓰레드가 다른 쓰레드에 의해 우위를 갖게 된다면, 한 쓰레드만 열심히 일하고 다른 쓰레드는 일할 수 없는 기아 상태(starvation)가 발생할 수 있습니다.

위에서 말한 해결 방식을 코드로 옮기자면 아래와 같습니다.

```
#include <iostream>
#include <mutex> // mutex 를 사용하기 위해 필요
#include <thread>

void worker1(std::mutex& m1, std::mutex& m2) {
    for (int i = 0; i < 10; i++) {
```

```
m1.lock();
m2.lock();
std::cout << "Worker1 Hi! " << i << std::endl;

m2.unlock();
m1.unlock();
}

}

void worker2(std::mutex& m1, std::mutex& m2) {
    for (int i = 0; i < 10; i++) {
        while (true) {
            m2.lock();

            // m1 이 이미 lock 되어 있다면 "야 차 빼" 를 수행하게 된다.
            if (!m1.try_lock()) {
                m2.unlock();
                continue;
            }

            std::cout << "Worker2 Hi! " << i << std::endl;
            m1.unlock();
            m2.unlock();
            break;
        }
    }
}

int main() {
    std::mutex m1, m2; // 우리의 mutex 객체

    std::thread t1(worker1, std::ref(m1), std::ref(m2));
    std::thread t2(worker2, std::ref(m1), std::ref(m2));

    t1.join();
    t2.join();

    std::cout << "끝!" << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Worker1 Hi! 0
Worker1 Hi! 1
Worker1 Hi! 2
Worker1 Hi! 3
```

```

Worker1 Hi! 4
Worker1 Hi! 5
Worker1 Hi! 6
Worker1 Hi! 7
Worker1 Hi! 8
Worker1 Hi! 9
Worker2 Hi! 0
Worker2 Hi! 1
Worker2 Hi! 2
Worker2 Hi! 3
Worker2 Hi! 4
Worker2 Hi! 5
Worker2 Hi! 6
Worker2 Hi! 7
Worker2 Hi! 8
Worker2 Hi! 9
끝!

```

데드락 상황 없이 잘 실행됨을 알 수 있습니다. (물론 출력하는 개수가 적어서 그럴 수도 있습니다.
for 문에서 10 을 10000 정도로 바꿔보세요. 그럼에도 잘 실행됨을 알 수 있습니다.)

```

m1.lock();
m2.lock();
std::cout << "Worker1 Hi! " << i << std::endl;

m2.unlock();
m1.unlock();

```

일단 worker1 은 lock_guard 를 통해 구현한 부분을 그대로 옮겨왔습니다. worker1 이 뮤텍스를 갖고 경쟁할 때 우선권을 가지므로 굳이 코드를 바꿀 필요가 없습니다. 차를 빼야 하는 것은 worker2 이니까요.

```

while (true) {
    m2.lock();

    // m1 이 이미 lock 되어 있다면 "야 차 빼" 를 수행하게 된다.
    if (!m1.try_lock()) {
        m2.unlock();
        continue;
    }

    std::cout << "Worker2 Hi! " << i << std::endl;
}

```

```
m1.unlock();  
m2.unlock();  
break;  
}
```

worker2의 경우 사뭇 다릅니다. 일단 m2는 아무 문제 없이 lock 할 수 있습니다. 하지만 문제는 m1을 lock하는 과정입니다.

만약에 worker1이 m1을 lock하고 있다면 어떨까요? m1.lock을 호출한 순간 서로 교차로 끼어서 이도저도 못하는 상황이 되는 것이겠지요.

C++에서는 try_lock이라는 함수를 제공하는데, 이 함수는 만일 m1을 lock할 수 있다면 lock을 하고 true를 리턴합니다. 그런데 lock() 함수와는 다르게, lock을 할 수 없다면 기다리지 않고 그냥 false를 리턴합니다.

따라서 m1.try_lock()이 true를 리턴하였다면 worker2가 m1과 m2를 성공적으로 lock한 상황이므로 (교차로에 노란차만 있는 상황) 그대로 처리하면 됩니다.

반면에 m1.try_lock()이 false를 리턴하였다면 worker1이 이미 m1을 lock했다는 의미이지요. 이 경우 worker1에서 우선권을 줘야 하기 때문에 자신이 이미 얻은 m2 역시 worker1에게 제공해야 합니다. 쉽게 말해 교차로에서 노란차가 후진한다고 보시면 됩니다.

그 후에 while을 통해 m1과 m2 모두 lock하는 것을 성공할 때 까지 계속 시도하게 되며, 성공하게 되면 while을 빠져나가겠지요.

이와 같이 데드락을 해결하는 것은 매우 복잡합니다 (또한 완벽하지 않지요). 애초에 데드락 상황이 발생할 수 없게 프로그램을 잘 설계하는 것이 중요합니다.

C++ Concurrency In Action 이란 책에선 데드락 상황을 피하기 위해 다음과 같은 가이드라인을 제시하고 있습니다.

중첩된 Lock을 사용하는 것을 피해라

모든 쓰레드들이 최대 1개의 Lock만을 소유한다면 (일반적인 경우에) 데드락 상황이 발생하는 것을 피할 수 있습니다. 또한 대부분의 디자인에서는 1개의 Lock으로도 충분합니다. 만일 여러개의 Lock을 필요로 한다면 정말 필요로 하는지 를 되물어보는 것이 좋습니다.

Lock을 소유하고 있을 때 유저 코드를 호출하는 것을 피해라

사실 이 가이드라인 역시 위에서 말한 내용과 자연스럽게 따라오는 것이긴 한데, 유저 코드에서 Lock을 소유할 수도 있기에 중첩된 Lock을 얻는 것을 피하려면 Lock 소유시 유저 코드를 호출하는 것을 지양해야 합니다.

Lock들을 언제나 정해진 순서로 획득해라

만일 여러개의 Lock들을 획득해야 할 상황이 온다면, 반드시 이 Lock들을 정해진 순서로 획득해야 합니다. 우리가 앞선 예제에서 데드락이 발생했던 이유 역시, `worker1`에서는 `m1`, `m2` 순으로 `lock`을 하였지만 `worker2`에서는 `m2`, `m1` 순으로 `lock`을 하였기 때문이지요. 만일 `worker2`에서 역시 `m1`, `m2` 순으로 `lock`을 하였다면 데드락은 발생하지 않았을 것입니다.

생산자(Producer) 와 소비자(Consumer) 패턴

다음으로 멀티 쓰레드 프로그램에서 가장 많이 등장하는 생산자(producer)-소비자(consumer) 패턴에 대해서 살펴보겠습니다.



생산자는 여러분의 상사, 소비자는 바로 일을 처리하는 여러분입니다!

생산자의 경우, 무언가 처리할 일을 받아오는 쓰레드를 의미합니다. 예를 들어서, 여러분이 인터넷에서 페이지를 긁어서 분석하는 프로그램을 만들었다고 생각해봅시다. 이 경우 페이지를 긁어 오는 쓰레드가 바로 생산자가 되겠지요.

소비자의 경우, 받은 일을 처리하는 쓰레드를 의미합니다. 앞선 예제의 경우 긁어온 페이지를 분석하는 쓰레드가 해당 역할을 하겠습니다.

그렇다면 이와 같은 상황을 쓰레드로 어떻게 구현할지 살펴보겠습니다.

```
#include <chrono> // std::chrono::milliseconds  
#include <iostream>  
#include <mutex>
```

```
#include <queue>
#include <string>
#include <thread>
#include <vector>

void producer(std::queue<std::string*>* downloaded_pages, std::mutex* m,
              int index) {
    for (int i = 0; i < 5; i++) {
        // 웹사이트를 다운로드하는데 걸리는 시간이라 생각하면 된다.
        // 각 쓰레드 별로 다운로드하는데 걸리는 시간이 다르다.
        std::this_thread::sleep_for(std::chrono::milliseconds(100 * index));
        std::string content = "웹사이트 : " + std::to_string(i) + " from thread(" +
                             std::to_string(index) + ")\\n";
        // data 는 쓰레드 사이에서 공유되므로 critical section 에 넣어야 한다.
        m->lock();
        downloaded_pages->push(content);
        m->unlock();
    }
}

void consumer(std::queue<std::string*>* downloaded_pages, std::mutex* m,
              int* num_processed) {
    // 전체 처리하는 페이지 개수가 5 * 5 = 25 개.
    while (*num_processed < 25) {
        m->lock();
        // 만일 현재 다운로드한 페이지가 없다면 다시 대기.
        if (downloaded_pages->empty()) {
            m->unlock(); // (Quiz) 여기서 unlock 을 안한다면 어떻게 될까요?

            // 10 밀리초 뒤에 다시 확인한다.
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            continue;
        }

        // 맨 앞의 페이지를 읽고 대기 목록에서 제거한다.
        std::string content = downloaded_pages->front();
        downloaded_pages->pop();

        (*num_processed)++;
        m->unlock();

        // content 를 처리한다.
        std::cout << content;
        std::this_thread::sleep_for(std::chrono::milliseconds(80));
    }
}

int main() {
    // 현재 다운로드한 페이지들 리스트로, 아직 처리되지 않은 것들이다.
    std::queue<std::string> downloaded_pages;
```

```
std::mutex m;

std::vector<std::thread> producers;
for (int i = 0; i < 5; i++) {
    producers.push_back(std::thread(producer, &downloaded_pages, &m, i + 1));
}

int num_processed = 0;
std::vector<std::thread> consumers;
for (int i = 0; i < 3; i++) {
    consumers.push_back(
        std::thread(consumer, &downloaded_pages, &m, &num_processed));
}

for (int i = 0; i < 5; i++) {
    producers[i].join();
}
for (int i = 0; i < 3; i++) {
    consumers[i].join();
}
}
```

성공적으로 컴파일 하였다면

실행 결과

```
웹사이트 : 0 from thread(1)
웹사이트 : 0 from thread(2)
웹사이트 : 1 from thread(1)
웹사이트 : 0 from thread(3)
웹사이트 : 2 from thread(1)
웹사이트 : 0 from thread(4)
웹사이트 : 1 from thread(2)
웹사이트 : 3 from thread(1)
웹사이트 : 0 from thread(5)
웹사이트 : 4 from thread(1)
웹사이트 : 1 from thread(3)
웹사이트 : 2 from thread(2)
웹사이트 : 1 from thread(4)
웹사이트 : 3 from thread(2)
웹사이트 : 2 from thread(3)
웹사이트 : 1 from thread(5)
웹사이트 : 4 from thread(2)
```

```
웹사이트 : 2 from thread(4)
웹사이트 : 3 from thread(3)
웹사이트 : 2 from thread(5)
웹사이트 : 4 from thread(3)
웹사이트 : 3 from thread(4)
웹사이트 : 3 from thread(5)
웹사이트 : 4 from thread(4)
웹사이트 : 4 from thread(5)
```

와 같이 나옵니다. 일단 위 코드가 어떻게 생산자-소비자 패턴을 구현하였는지 살펴봅시다.

```
std::queue<std::string> downloaded_pages;
```

먼저 **producer** 쓰레드에서는 웹사이트에서 페이지를 계속 다운로드 하는 역할을 하게 됩니다. 이 때, 다운로드한 페이지들을 **downloaded_pages** 라는 큐에 저장하게 됩니다.



왜 굳이 큐를 사용하였나면 큐가 바로 먼저 들어온 것이 먼저 나간다(**First In First Out - FIFO**)라는 특성이 있기 때문입니다. 쉽게 말해, 먼저 다운로드한 페이지를 먼저 처리하기 위함이지요.

물론 **vector**로 구현해도 상관은 없습니다. 하지만 **vector**를 사용하였을 경우, 가장 먼저 도착한 페이지가 벡터 첫번째 원소로 있을터인데, 이를 제거하는 작업이 꽤나 느리기 때문에 권장하지 않습니다. (맨 앞의 원소를 제거하면, 나머지 모든 원소들을 앞으로 한 칸씩 땡겨줘야 하지요)

하지만 큐의 경우 해당 연산들이 매우 빠르게 이루어질 수 있습니다.

producer 를 살펴보면 아래와 같습니다.

```
// 웹사이트를 다운로드 하는데 걸리는 시간이라 생각하면 된다.
// 각 쓰레드 별로 다운로드 하는데 걸리는 시간이 다르다.
std::this_thread::sleep_for(std::chrono::milliseconds(100 * index));
std::string content = "웹사이트 : " + std::to_string(i) + " from thread(" +
                     std::to_string(index) + ")\\n";
```

```
// downloaded_pages 는 쓰레드 사이에서 공유되므로 critical section에 넣어야
// 한다.
m->lock();
downloaded_pages->push(content);
m->unlock();
```

일단 기본적으로 C++ 표준 라이브러리 상에서는 인터넷 페이지를 다운받는 기능을 제공하지 않기 때문에, 대략 비슷한 상황을 가정하고 시뮬레이션 하였습니다.

`std::this_thread::sleep_for` 함수는 인자로 전달된 시간 만큼 쓰레드를 `sleep` 시키는데, 이 때 해당 인자로 `chrono` 의 시간 객체를 받게 됩니다. `chrono` 는 C++ 11 에 새로 추가된 시간 관련 라이브러리로 기존의 C 의 `time.h` 보다 훨씬 편리한 기능을 제공하고 있습니다. 이에 대해서는 나중 강좌에서 자세히 다루어 보도록 하고, 일단 `100 * index` 밀리초 만큼 쓰레드를 재우기 위해서는 `std::chrono::milliseconds(100 * index)` 와 같이 전달하면 됩니다.

그리고 다운 받은 웹사이트 내용이 `content` 라고 생각해봅시다.

그렇다면, 이제 다운 받은 페이지를 작업 큐에 집어 넣어야 합니다. 이 때 주의할 점으로, `producer` 쓰레드가 1 개가 아니라 5 개나 있다는 점입니다. 따라서 `downloaded_pages`에 접근하는 쓰레드들 사이에 *race condition* 이 발생할 수 있습니다.

이를 방지하기 위해서 뮤텍스 `m` 으로 해당 코드를 감싸서 문제가 발생하지 않게 해줍니다.

자 그럼 `consumer` 의 경우 어떤 식으로 구현할 지 생각해봅시다.

먼저 `consumer` 쓰레드의 입장에서는 언제 일이 올지 알 수 없습니다. 따라서 `downloaded_pages` 가 비어있지 않을 때 까지 계속 `while` 루프를 돌아야겠지요. 한 가지 문제는 컴퓨터 CPU 의 속도에 비해 웹사이트 정보가 큐에 추가되는 속도는 매우 느리다는 점입니다.

우리의 `producer` 의 경우 대충 100ms 마다 웹사이트 정보를 큐에 추가하게 되는데, 이 시간 동안 `downloaded_pages->empty()` 이 문장을 수십 번 호출할 수 있을 것입니다. 이는 상당한 CPU 자원의 낭비가 아닐 수 없지요.

따라서, 실제로는 아래와 같이 구현하였습니다.

```
m->lock();
// 만일 현재 다운로드한 페이지가 없다면 다시 대기.
if (downloaded_pages->empty()) {
    m->unlock(); // (Quiz) 여기서 unlock 을 안한다면 어떻게 될까요?

    // 10 밀리초 뒤에 다시 확인한다.
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    continue;
}
```

`downloaded_pages->empty()` 라면, 강제로 쓰레드를 `sleep` 시켜서 10 밀리초 뒤에 다시 확

인하는 식으로 말이지요.

참고로 `m->unlock` 을 위 `if` 문 안에서 호출하지 않는다면 데드락이 발생하게 됩니다. (왜 인지는 생각해보세요!)

```
// 맨 앞의 페이지를 읽고 대기 목록에서 제거한다.
std::string content = downloaded_pages->front();
downloaded_pages->pop();

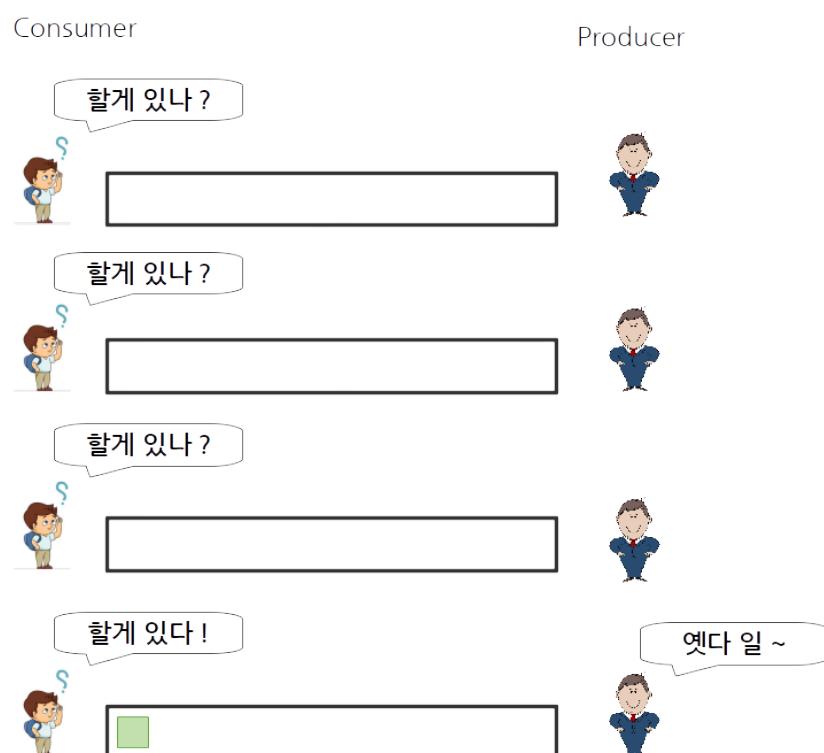
(*num_processed)++;
m->unlock();

// content 를 처리한다.
std::cout << content;
std::this_thread::sleep_for(std::chrono::milliseconds(80));
```

마지막으로 `content` 를 처리하는 과정은 간단합니다. `front` 를 통해서 맨 앞의 원소를 얻은 뒤에, `pop` 을 호출하면 맨 앞의 원소를 큐에서 제거하게 됩니다.

이 때 `m->unlock` 을 함으로써 다른 쓰레드에서도 다음 원소를 바로 처리할 수 있도록 해야되죠. `content` 를 처리하는 시간은 대충 80 밀리초가 소모된다고 시뮬레이션 하였습니다.

우리의 `producer` 와 `consumer` 를 관계를 그림으로 보자면 아래와 같습니다.

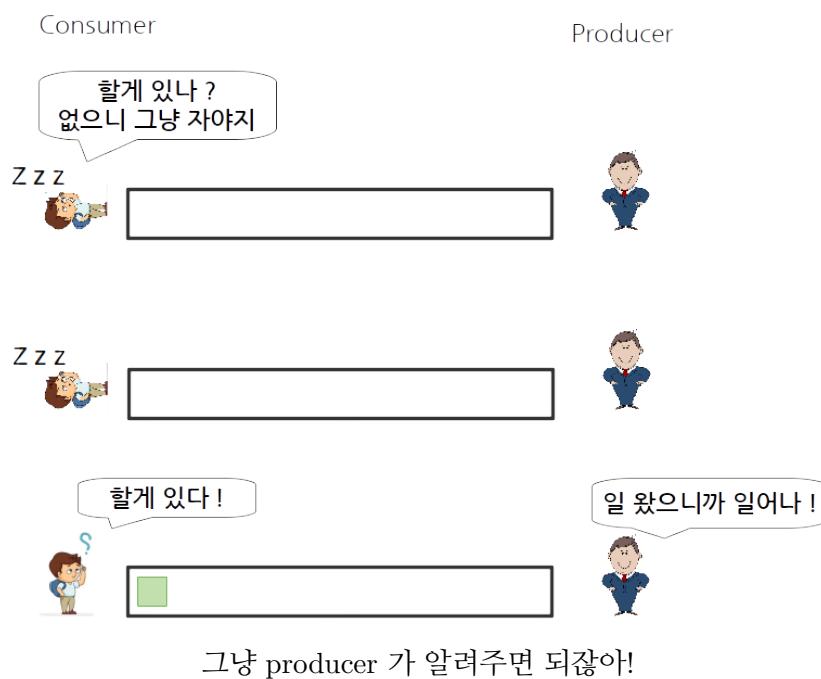


굳이 consumer 가 계속 물어봐야 할까?

위 그림처럼 우리의 구현에서 `consumer` 쓰레드가 10 밀리초마다 `downloaded_pages`에 할 일이 있는지 확인하고 없으면 다시 기다리는 형태를 취하고 있습니다.

이는 매우 비효율적입니다. 매 번 언제 올지 모르는 데이터를 확인하기 위해 지속적으로 `mutex`를 `lock`하고, 큐를 확인해야 하기 때문이지요.

차라리 `producer`에서 데이터가 뜸하게 오는 것을 안다면 그냥 `consumer`는 아예 재워놓고, `producer`에서 데이터가 온다면 `consumer`를 깨우는 방식이 낫지 않을까요? 쓰레드를 재워놓게 되면, 그 사이에 다른 쓰레드들이 일을 할 수 있기 때문에 CPU를 더 효율적으로 쓸 수 있을 것입니다.



C++에서는 위와 같은 형태로 생산자 소비자 패턴을 구현할 수 있도록 여러 가지 도구들을 제공하고 있습니다.

condition_variable

위와 같은 상황에서 쓰레드들을 10 밀리초마다 재웠다 깨웠다 할 수 밖에 없었던 이유는 어떠한 조건을 만족할 때 까지 자라!라는 명령을 내릴 수 없었기 때문입니다.

위 경우 `downloaded_pages`가 `empty()`가 참이 아닐 때 까지 자라 라는 명령을 내리고 싶었겠지요.

이는 조건 변수(**condition_variable**)를 통해 해결할 수 있습니다.

```
#include <chrono>           // std::chrono::milliseconds
#include <condition_variable> // std::condition_variable
#include <iostream>
```

```
#include <mutex>
#include <queue>
#include <string>
#include <thread>
#include <vector>

void producer(std::queue<std::string>* downloaded_pages, std::mutex* m,
              int index, std::condition_variable* cv) {
    for (int i = 0; i < 5; i++) {
        // 웹사이트를 다운로드하는데 걸리는 시간이라 생각하면 된다.
        // 각 쓰레드 별로 다운로드하는데 걸리는 시간이 다르다.
        std::this_thread::sleep_for(std::chrono::milliseconds(100 * index));
        std::string content = "웹사이트 : " + std::to_string(i) + " from thread(" +
                             std::to_string(index) + ")\\n";
        // data 는 쓰레드 사이에서 공유되므로 critical section 에 넣어야 한다.
        m->lock();
        downloaded_pages->push(content);
        m->unlock();

        // consumer 에게 content 가 준비되었음을 알린다.
        cv->notify_one();
    }
}

void consumer(std::queue<std::string>* downloaded_pages, std::mutex* m,
              int* num_processed, std::condition_variable* cv) {
    while (*num_processed < 25) {
        std::unique_lock<std::mutex> lk(*m);

        cv->wait(
            lk, [&] { return !downloaded_pages->empty() || *num_processed == 25; });

        if (*num_processed == 25) {
            lk.unlock();
            return;
        }

        // 맨 앞의 페이지를 읽고 대기 목록에서 제거한다.
        std::string content = downloaded_pages->front();
        downloaded_pages->pop();

        (*num_processed)++;
        lk.unlock();

        // content 를 처리한다.
        std::cout << content;
        std::this_thread::sleep_for(std::chrono::milliseconds(80));
    }
}
```

```
int main() {
    // 현재 다운로드한 페이지들 리스트로, 아직 처리되지 않은 것들이다.
    std::queue<std::string> downloaded_pages;
    std::mutex m;
    std::condition_variable cv;

    std::vector<std::thread> producers;
    for (int i = 0; i < 5; i++) {
        producers.push_back(
            std::thread(producer, &downloaded_pages, &m, i + 1, &cv));
    }

    int num_processed = 0;
    std::vector<std::thread> consumers;
    for (int i = 0; i < 3; i++) {
        consumers.push_back(
            std::thread(consumer, &downloaded_pages, &m, &num_processed, &cv));
    }

    for (int i = 0; i < 5; i++) {
        producers[i].join();
    }

    // 나머지 자고 있는 쓰레드들을 모두 깨운다.
    cv.notify_all();

    for (int i = 0; i < 3; i++) {
        consumers[i].join();
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
웹사이트 : 0 from thread(1)
웹사이트 : 0 from thread(2)
웹사이트 : 1 from thread(1)
웹사이트 : 0 from thread(3)
웹사이트 : 2 from thread(1)
웹사이트 : 1 from thread(2)
웹사이트 : 0 from thread(4)
웹사이트 : 3 from thread(1)
웹사이트 : 0 from thread(5)
웹사이트 : 4 from thread(1)
```

```
웹사이트 : 1 from thread(3)
웹사이트 : 2 from thread(2)
웹사이트 : 1 from thread(4)
웹사이트 : 3 from thread(2)
웹사이트 : 2 from thread(3)
웹사이트 : 1 from thread(5)
웹사이트 : 4 from thread(2)
웹사이트 : 2 from thread(4)
웹사이트 : 3 from thread(3)
웹사이트 : 2 from thread(5)
웹사이트 : 4 from thread(3)
웹사이트 : 3 from thread(4)
웹사이트 : 3 from thread(5)
웹사이트 : 4 from thread(4)
웹사이트 : 4 from thread(5)
```

와 같이 나옵니다.

```
condition_variable cv;
```

먼저 뮤텍스를 정의할 때와 같이 `condition_variable` 을 정의하였습니다. 이 `condition_variable` 이 어떻게 사용되는지 `consumer` 쓰레드 부터 살펴봅시다.

```
std::unique_lock<std::mutex> lk(*m);
cv->wait(lk,
          [&] { return !downloaded_pages->empty() || *num_processed == 25; });
```

대충 코드를 보면 느낌이 오겠지만, `condition_variable` 의 `wait` 함수에 어떤 조건이 참이 될 때 까지 기다릴지 해당 조건을 인자로 전달해야 합니다. 우리의 경우 조건으로

```
!downloaded_pages->empty() || *num_processed == 25;
```

를 전달하였는데, 이는 `downloaded_pages` 에 원소들이 있거나, 전체 처리된 페이지의 개수가 25개 일 때 `wait` 을 중지하도록 하였습니다.

조건 변수는 만일 해당 조건이 거짓이라면, `lk` 를 `unlock` 한 뒤에, 영원히 `sleep` 하게 됩니다. 이 때 이 쓰레드는 다른 누가 깨워주기 전까지 계속 `sleep` 된 상태로 기다리게 됩니다. 한 가지 중요한 점이라면 `lk` 를 `unlock` 한다는 점입니다.

반면에 해당 조건이 참이라면 `cv.wait` 는 그대로 리턴해서 `consumer` 의 `content` 를 처리하는 부분이 그대로 실행되게 됩니다.

```
std::unique_lock<std::mutex> lk(*m);
```

참고로 기존의 `lock_guard` 와는 다르게 `unique_lock` 을 정의하였는데, 사실 `unique_lock` 은 `lock_guard` 와 거의 동일합니다. 다만, `lock_guard` 의 경우 생성자 말고는 따로 `lock` 을 할 수 없는데, `unique_lock` 은 `unlock` 후에 다시 `lock` 할 수 있습니다.

덧붙여 `unique_lock` 을 사용한 이유는 `cv->wait` 가 `unique_lock` 을 인자로 받기 때문입니다.

```
if (*num_processed == 25) {
    lk.unlock();
    return;
}
```

`cv.wait` 후에 아래 `num_processed` 가 25 인지 확인하는 구문이 추가되었는데, 이는 `wait` 에서 탈출한 이유가 모든 페이지 처리를 완료해서인지, 아니면 정말 `downloaded_pages` 에 페이지가 추가되었는지 알 수 없기 때문입니다. 만일 모든 페이지 처리가 끝나서 탈출한 것였다면, 그냥 쓰레드를 종료해야 합니다.

자 그렇다면 `producer` 를 확인해보겠습니다.

```
// consumer에게 content가 준비되었음을 알린다.
cv->notify_one();
```

만약에 페이지를 하나 다운 받았다면, 잠자고 있는 쓰레드들 중 하나를 깨워서 일을 시켜야겠죠? (만약에 모든 쓰레드들이 일을 하고 있는 상태라면 아무 일도 일어나지 않습니다.) `notify_one` 함수는 말 그대로, 조건이 거짓인 바람에 자고 있는 쓰레드 중 하나를 깨워서 조건을 다시 검사하게 해줍니다. 만일 조건이 참이 된다면 그 쓰레드가 다시 일을 시작하겠지요.

```
for (int i = 0; i < 5; i++) {
    producers[i].join();
}

// 나머지 자고 있는 쓰레드들을 모두 깨운다.
cv.notify_all();
```

`producer` 들이 모두 일을 끝낸 시점을 생각해본다면, 자고 있는 일부 `consumer` 쓰레드들이 있을 것입니다. 만약에 `cv.notify_all()` 을 하지 않는다면, 자고 있는 `consumer` 쓰레드들의 경우 `join` 되지 않는 문제가 발생합니다.

따라서 마지막으로 `cv.notify_all()` 을 통해서 모든 쓰레드를 깨워서 조건을 검사하도록 합니다. 해당 시점에선 이미 `num_processed` 가 25 가 되어 있을 것이므로, 모든 쓰레드들이 잠에서 깨어나 종료하게 됩니다.

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. 다음 강좌에서는 C++ 에서 제공하는 또 다른 기능인 `atomic` 객체에 대해 다루어 볼 것입니다.

뭘 배웠지?

- 여러 쓰레드에서 같은 객체의 값을 수정한다면 Race Condition 이 발생합니다. 이를 해결하기 위해서는 여러가지 방법이 있지만, 한 가지 방법으로 뮤텍스를 사용하는 방법이 있습니다.
- 뮤텍스는 한 번에 한 쓰레드에서만 획득할 수 있습니다. 획득한 뮤텍스는 반드시 반환해야 합니다.
- `lock_guard` 나 `unique_lock` 등을 이용하면 뮤텍스의 획득-반환을 손쉽게 처리할 수 있습니다.
- 뮤텍스를 사용할 때 데드락이 발생하지 않도록 주의해야 합니다. 데드락을 디버깅하는 것은 매우 어렵습니다.
- `condition_variable` 을 사용하면 생산자-소비자 패턴을 쉽게 구현할 수 있습니다.

생각 해보기

문제 1

`condition_variable`에서 `wait` 말고도 `wait_for` 과 `wait_until`이라는 다른 유용한 함수들이 있습니다. 여기서 읽어보세요!

atomic 객체와 명령어 재배치

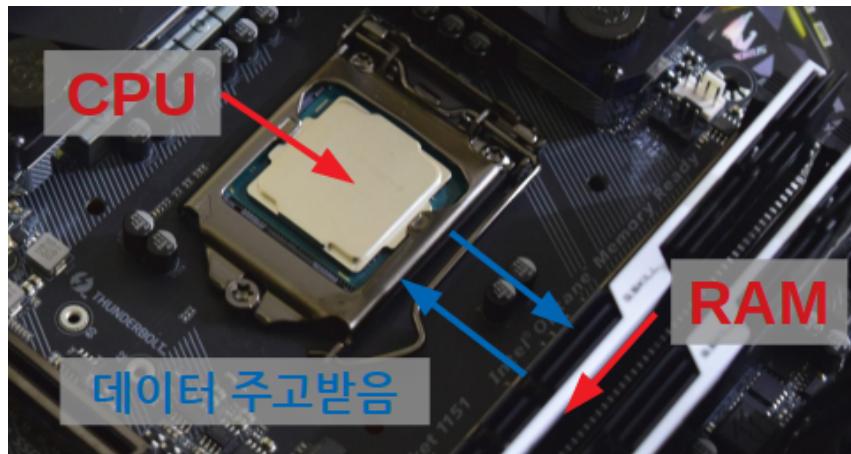
안녕하세요 여러분!

지난 두 강좌를 통해 C++에서 멀티 쓰레딩을 위해 제공하는 기본적인 요소들인 쓰레드(thread), 뮤텍스(mutex), 조건변수(condition variable)들의 사용법에 대해 배웠습니다. 이번 강좌에서는 이러한 기본 요소들을 조금 더 세밀하게 컨트롤 할 수 있는 몇 가지 요소들에 대해 살펴볼 것입니다.

이번 강좌는 C++의 매우 세세한 디테일을 다루기 때문에, C++을 처음 배우시는 분들은 넘어가셔도 좋습니다.

메모리는 엄청 느리다.

강좌를 진행하기 전에, 컴퓨터 메모리에 관련한 몇 가지 중요한 사실들을 짚고 넘어갈 것입니다.



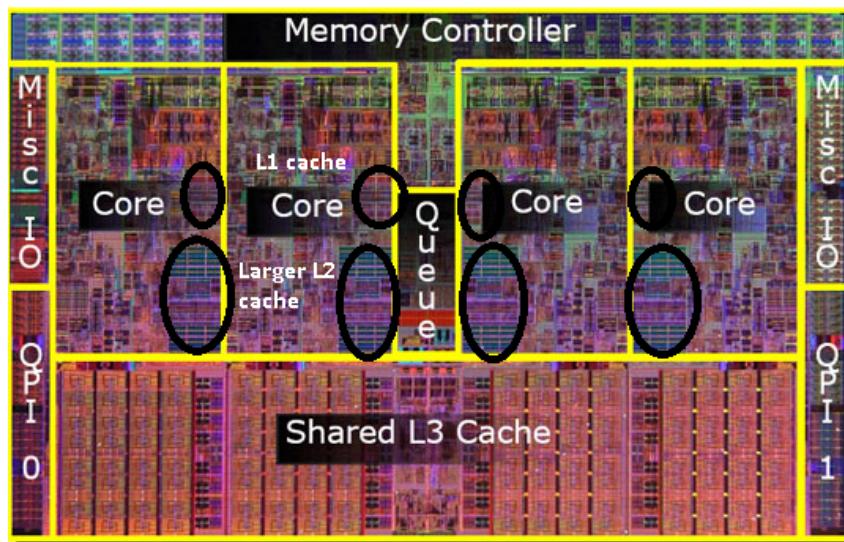
CPU 와 RAM

기본적으로 CPU와 컴퓨터 메모리인 RAM은 물리적으로 떨어져 있습니다. 따라서 CPU가 메모리에서 데이터를 읽어 오기 위해서는 꽤 많은 시간이 걸립니다. 실제로, 인텔의 i7-6700 CPU의 경우 최소 42 사이클 정도 걸린다고 보시면 됩니다. CPU에서 덧셈 한 번을 1 사이클에 끝낼 수 있는데, 메모리에서 데이터 오는 것을 기다리느라, 42 번 덧셈을 연산할 시간을 놓치게 되는 것입니다.

이는 CPU 입장에 굉장히 손해가 아닐 수 없습니다. 메모리에서 데이터 한 번 읽을 때마다 42 사이클 동안 아무것도 못한다니 말입니다.

캐시

따라서 CPU 개발자들은, 이를 보완하기 위해 캐시(Cache)라는 것을 도입하였습니다. 캐시란, CPU 칩 안에 있는 조그마한 메모리라고 보시면 됩니다. 여기서 중요한 점은 램과는 다르게 CPU에서 연산을 수행하는 부분이랑 거의 붙어 있다 싶이 해서, 읽기 / 쓰기 속도가 매우 빠르다는 점입니다.



위 그림에서 L1, L2, L3 라 표시된 것이 모두 캐시입니다. CPU에서 연산하는 부분 (Core) 보다 캐시가 더 큽니다.

캐시의 크기는 그렇게 크지 않습니다. 요즈음 컴퓨터 램 크기가 적어도 8 GB 정도는 달고 나오는데에 비해, 인텔의 하스웰 아키텍쳐인 i7-4770 CPU의 경우, L1 캐시는 32KB, L2 캐시는 256 KB, L3 캐시는 8 MB 정도 됩니다. 여러분이 다른 CPU를 쓰고 있다 하더라도 아마 큰 차이는 없을 것입니다.

하지만, L1 읽기 쓰기의 경우 단 4 사이클이면 충분하고, L2 캐시는 12 사이클, L3 캐시는 36 사이클 정도로 메모리를 왔다 갔다 하는 것 보다 훨씬 빠른 속도로 접근할 수 있게 됩니다.

따라서, 실제로는 다음과 같이 작동합니다. CPU에서 가장 많이 접근하는 메모리 영역은 L1 캐시에 가져다 놓게 되고, 그 다음으로, 자주 접근하는 부분은 L2, 마지막으로 L3 캐시 순으로 놓게 된다는 것이지요.

CPU가 특정한 주소에 있는 데이터에 접근하려 한다면, 일단 캐시에 있는지 확인한 후, 캐시에 있다면 해당 값을 읽고, 없다면 메모리 까지 갔다 오는 방식으로 진행됩니다. 이렇게 캐시에 있는 데이터를 다시 요청해서 시간을 절약하는 것을 *Cache hit* 이라고 하며 반대로 캐시에 요청한 데이터가 없어서 메모리 까지 갔다 오는 것을 *Cache miss*라고 부릅니다.

하지만 여기서 문제가 있습니다. CPU가 어떻게 어느 영역의 메모리에 자주 접근할 지 어떻게 아는 것일까요? 답은 알 수 없다 입니다. 따라서 보통 CPU에서 캐시가 작동하는 방식은 다음과 같습니다.

- 메모리를 읽으면 일단 캐시에 저장해놓는다.
- 만일 캐시가 다 찼다면 특정한 방식에 따라 처리한다.

이 때 여기서 말하는 특정한 방식은 CPU마다 다른데, 대표적인 예로 가장 이전에 쓴(LRU - Least Recently Used) 캐시를 날려버리고 그 자리에 새로운 캐시를 기록하는 방식이 있습니다. 이 LRU 방식의 가장 큰 특징으로는, 최근에 접근한 데이터를 자주 반복해서 접근한다면 매우 유리하다는 점이 있습니다.

예를 들어서 캐시 크기가 1 KB 밖에 안되고 LRU 방식을 사용하는 CPU가 있다고 했을 때 첫 번째 코드가 더 빠르게 작동할까요? 아니면 두 번째 코드가 더 빨리 작동할까요? 두 코드는 동일한 연산을 수행합니다.

```
for (int i = 0; i < 1000; i++) {
    for (int j = 0; j < 10000; j++) {
        s += data[j];
    }
}
```

와

```
for (int j = 0; j < 10000; i++) {
    for (int i = 0; i < 10000; i++) {
        s += data[j];
    }
}
```

답은 두 번째 방식입니다. 왜냐하면 첫 번째 경우에서 `data[0]`를 접근하는 것을 생각해봅시다. 일단 첫 번째 루프에서 `data[0]`는 캐시에 들어가게 됩니다. 하지만, CPU 캐시가 매우 작기 때문에 `j = 256`이 되었을 때 `data[0]`는 캐시에서 사라지게 되지요 ($1\text{KB} = 1024 \text{ byte} = \text{int} 256$ 개).

따라서 `i = 1`인 루프에서 `data[0]`에 다시 접근했을 때 이미 `data[0]`는 캐시에서 사라진 이후기에 Cache Miss가 발생하게 됩니다. 따지고 보면 `data` 원소의 모든 접근이 Cache Miss가 되어서 느리겠지요.

반면에 후자의 경우 `data[0]`을 10000번 연속으로 접근하므로, 처음에 접근할 때 빼고 나머지 9999번 접근이 Cache hit이 되어서 빠르게 덧셈을 수행할 수 있게 됩니다.

캐시에 대해서는 이 정도로 줄이겠습니다. 사실 캐시에 대해서만 이야기해도 한 보따리는 풀 수 있지만, 이는 나중에 컴퓨터 시스템에 관한 강좌를 하게 되면다면 더 깊게 다루도록 하겠습니다.

컴퓨터는 사실 여러분이 시키는 대로 하지 않는다.

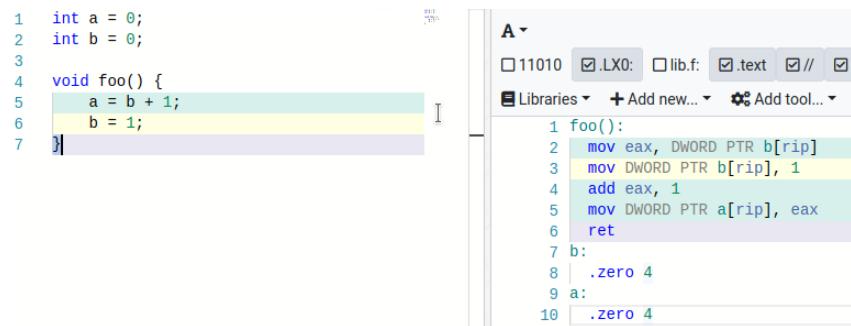
여태까지 여러분이 코드를 작성하면, 컴파일러가 그 코드를 그대로 기계어로 번역한 다음, CPU 가 해당 번역된 기계어를 그대로 실행시킨다고 생각하셨을 것입니다.

그런데 이게 사실이 아니라 한다면 여러분은 믿을 수 있으신가요?

```
int a = 0;
int b = 0;

void foo() {
    a = b + 1;
    b = 1;
}
```

위 코드를 그대로 컴파일 하였을 때, 생성되는 어셈블리는 아래와 같습니다.



The screenshot shows a debugger interface with assembly code. The assembly code is:

```

1  int a = 0;
2  int b = 0;
3
4  void foo() {
5      a = b + 1;           ; Line 5 highlighted in yellow
6      b = 1;              ; Line 6 highlighted in yellow
7  }

```

On the right, the assembly output is shown:

```

A▼
1 11010 LX0: lib.f: .text // \
2
3 Libraries + Add new... ⚙ Add tool...
4
5 foo():
6     mov eax, DWORD PTR b[rip]
7     mov DWORD PTR b[rip], 1
8     add eax, 1
9     mov DWORD PTR a[rip], eax
10    ret
11 b:
12     .zero 4
13 a:
14     .zero 4

```

같은 색깔로 나타낸 부분이, 해당 부분의 코드가 어셈블리로 컴파일 된 결과입니다

놀랍게도 `a = b + 1` 부분의 실행이 채 끝나기 전에 `b = 1` 이 먼저 실행이 끝나게 됩니다.



그런데 그것이 실제로 일어났습니다.

물론, 위 `foo` 함수의 입장에선 크게 문제는 없습니다. 왜냐하면, 최종적으로는 `a`에는 1이, `b`도 1이 들어 있을테니 말이지요.

하지만, 만약에 다른 쓰레드가 있어서 `a`와 `b`의 값을 확인하였을 때, 코드가 순서대로 실행되었더라면 `b`가 1이면 `a`도 1이어야하지만, `a`가 0인데, `b`가 1일 수 있다는 말입니다!

그렇다면 컴파일러는 도대체 왜 위와 같이 명령어를 재배치 한 것일까요? 이는 현대의 CPU 한 번에 한 명령어씩 실행하는 것이 아니기 때문입니다.

CPU 파이프라이닝 (pipelining)

여러분이 빨래를 하는 과정을 생각해봅시다.

먼저 빨래를 세탁기에 넣어야 하고, 세탁이 끝나면, 건조기에 넣어야 하고, 마지막으로 건조가 끝나면 빨래를 잘 개어야 겠지요. 위와 같이 빨래라는 과정은 여러 단계를 거쳐야 합니다.

여러 바구니의 빨래를 효율적으로 하려면 ?



다른 작업들을 같이 할 수는 없을까?

비효율적으로 빨래를 하는 방법

그렇다면 여러 바구니의 빨래를 있다고 해봅시다. 한 가지 방법은 위 그림처럼 한 단계씩 차례대로 하는 방법입니다. 한 바구니 빨래를 세탁 - 건조 - 빨래 개기 한 후에, 다른 바구니의 빨래를 하는 것이지요.

하지만 위와 같은 방식은 효율적이지 않습니다. 왜냐하면 빨래를 건조기에 넣게 된다면, 세탁기가 비어 있으므로, 그 사이에 다른 빨래를 또 세탁할 수 있기 때문이지요! 따라서 효율적으로 빨래를 하는 방식은 아래와 같을 것입니다.

여러 바구니의 빨래를 효율적으로 하려면 ?

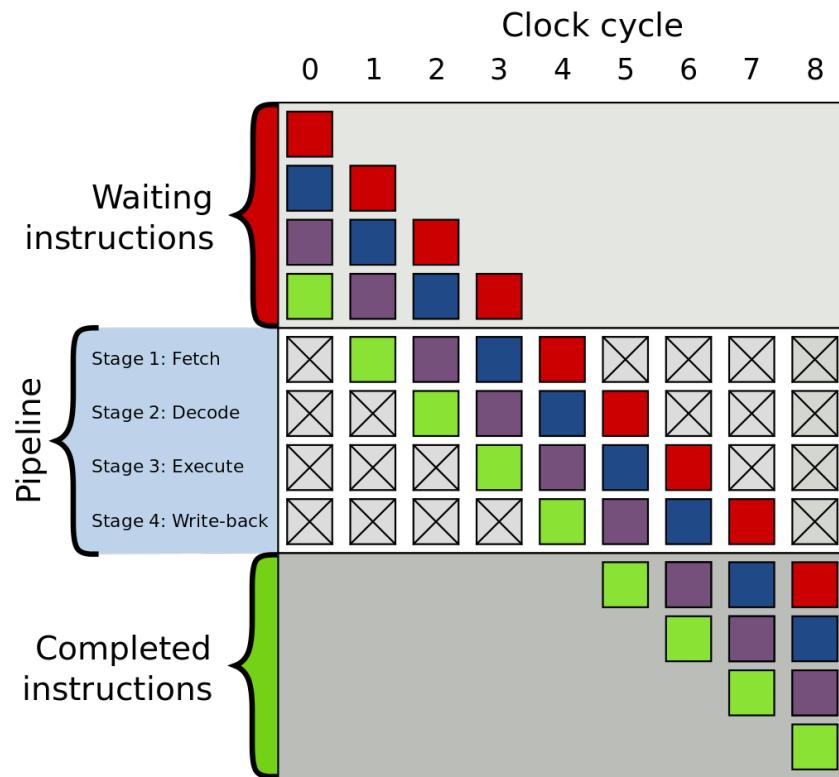


효율적으로 빨래를 하는 방법

위와 같이 모든 단계의 작업들을 쉬지 않고 계속 돌릴 수 있습니다. 즉, 이전의 방식은 효율이 33% 였다면, 새로운 방식의 경우 모든 단계를 100% 사용할 수 있게 되지요. 이와 같이, 한 작업(세탁 - 건조 - 개기)이 끝나기 전에, 다음 작업을 시작하는 방식으로 동시에 여러 개의 작업을 동시에 실행하는 것을 파이프라인ning(pipelining)이라고 합니다.

CPU도 마찬가지입니다. 실제 CPU에서 명령어를 실행할 때 여러 단계를 거치게 됩니다. 명령어를 읽어야 하고 (fetch), 읽은 명령어가 무엇인지 해석해야 하고 (decode), 해석된 명령어를 실행하고 (execute), 마지막으로 결과를 써야 하지요 (write).

CPU 역시 정확히 동일한 방법으로 명령어를 처리합니다.



CPU 의 파이프라이닝; 알고보면 빨래하는 것과 다를바가 없다.

위 그림에서는 각 단계의 실행 속도가 동일한 것 처럼 나타났지만, 실제로는 실행 부분의 실행 속도는 명령어마다 천차 만별입니다. 따라서, 만일 매우 실행 시간이 오래 걸리는 명령어가 있다면, 해당 작업 때문에 다른 명령어들이 꽉 밀리게 되겠지요.

예컨대, 세탁이나 빨래 개기는 30 분 밖에 안 걸리는데 건조가 3시간이 걸린다면, 건조기 기다리느라 빨래를 계속 할 수 없는 것과 마찬가지 입니다 (세탁이 끝난 빨래를 쌓아 놓을 수 없다는 전제하에)

따라서, 컴파일러는 우리가 어떠한 최대한 CPU 의 파이프라인을 효율적으로 활용할 수 있도록 명령어를 재배치하게 됩니다. 물론 전제 조건은 명령어를 재배치 하더라도 최종 결과물은 당연히도 달라지면 안되겠지요. 문제는 컴파일러가 명령어를 재배치 할 때, 다른 쓰레드들을 고려하지 않는다는 점입니다. 따라서 우리의 `foo` 함수 처럼, 멀티 쓰레드 환경에서는 예상치 못한 결과가 나올 수도 있습니다.

과연 컴파일러만 재배치를 할까?

한 가지 더 재미있는 점은, 꼭 컴파일러만이 명령어를 재배치하는게 아니라는 점입니다. 예를 들어서 다음과 같은 두 명령을 생각해봅시다.

```
// 현재 a = 0, b = 0;
a = 1; // 캐시에 없음
b = 1; // 캐시에 있음
```

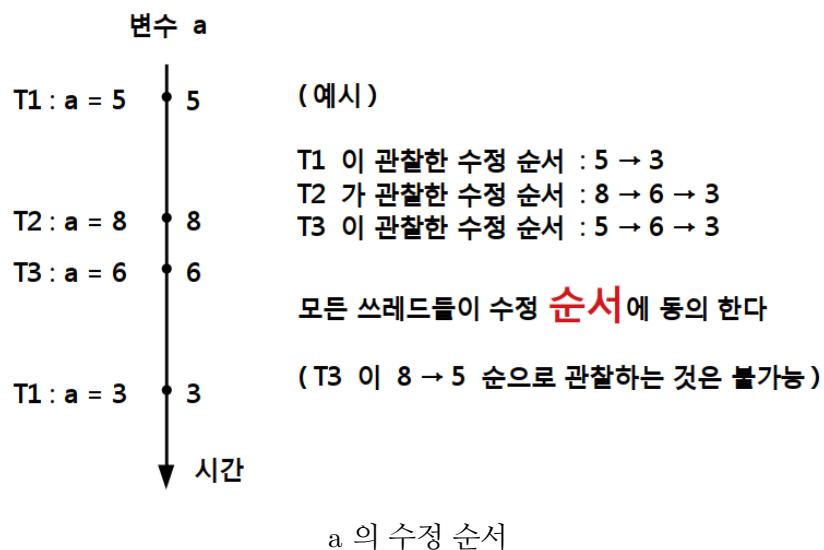
`a = 1`의 경우 현재 `a` 가 캐시에 없으므로, 매우 오래 걸립니다. 반면에 `b = 1;` 의 경우 현재 `b` 가 캐시에 있기 때문에 빠르게 처리할 수 있겠지요. 따라서 CPU에서 위 코드가 실행될 때, `b = 1;` 가 `a = 1;` 보다 먼저 실행될 수 있습니다.

따라서, 다른 쓰레드에서 `a` 는 0 인데, `b` 가 1 인 순간을 관찰할 수 있다는 것입니다.⁸⁾

무엇을 믿어야 하는가?

아니, 이렇게 명령어 순서도 뒤죽박죽 바꾸고 심지어 CPU에서도 명령어들을 제대로 된 순서로 실행하지 않는다면, 도대체 무엇을 믿을 수 있을까요?

C++의 모든 객체들은 수정 순서(modification order)라는 것을 정의할 수 있습니다. 수정 순서라 하는 것은, 만약에 어떤 객체의 값을 실시간으로 확인할 수 있는 전지전능한 무언가가 있다고 하였을 때, 해당 객체의 값의 변화를 기록한 것이라 보면 됩니다. (물론 실제로 존재하지 않고, 가상의 수정 순서가 있다고 생각해봅시다.)



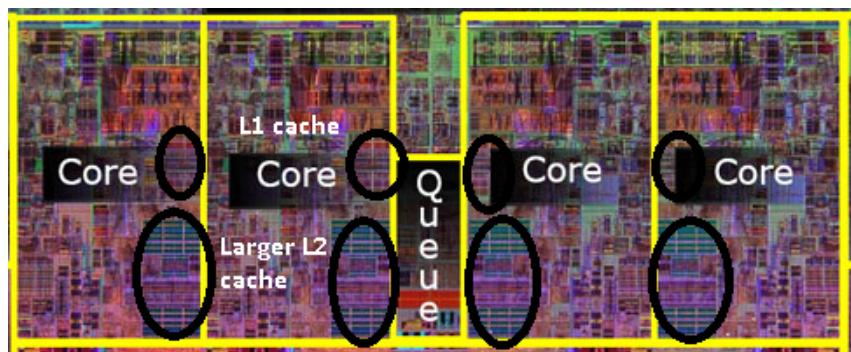
만약에 위처럼, 어떤 변수 `a`의 값이 위와 같은 형태로 변화해왔다고 해봅시다. C++에서 보장하는 사실은, 원자적 연산을 할 경우에 모든 쓰레드에서 같은 객체에 대해서 동일한 수정 순서를 관찰할 수 있다는 사실입니다.

여기서 강조할 점은 순서가 동일하다는 것이라는 점입니다. 쉽게 말해 어떤 쓰레드가 `a`의 값을 읽었을 때, 8로 읽었다면, 그 다음에 읽어지는 `a`의 값은 반드시 8, 6, 3 중에 하나여야 할 것입니다. 수정 순서를 거꾸로 거슬러 올라가서 5를 읽는 일은 없습니다.

8) 참고로 위 명령어가 읽기 - 읽기 순으로 나오기 때문에 이를 RR(Read - Read) 순서라고 합니다. 인텔 CPU에서는 RR의 명령어 재배치를 금지하고 있기에, 위에서 말한 상황은 발생하지 않습니다. 하지만 ARM의 CPU의 경우 RR 명령어 재배치가 가능하므로, 위와 같은 상황이 발생할 수 있습니다. 참고로 인텔 CPU의 경우 유일하게 쓰기 - 읽기 순은 재배치가 가능합니다. 즉, 쓰기가 채 끝나기 전에 다음 읽기가 먼저 실행될 수 있다는 뜻입니다.

모든 쓰레드에서 변수의 수정 순서에 동의만 한다면 문제될 것이 없습니다. 이 말이 무슨 말이냐면, 같은 시간에 변수 a 의 값을 관찰했다고 해서 굳이 모든 쓰레드들이 동일한 값을 관찰할 필요는 없다라는 점입니다. 예를 들어서 정확히 같은 시간에 쓰레드 1 과 2 에서 a 의 값을 관찰하였을 때 쓰레드 1 에서는 5 를 관찰하고, 쓰레드 2 에서는 8 을 관찰해도 문제될 것이 없습니다. 심지어, 동일한 코드를 각기 다른 쓰레드에서 실행하였을 때, 실행하는 순서가 달라도 (결과만 같다면) 문제가 안됩니다.

쓰레드 간에서 같은 시간에 변수의 값을 읽었을 때 다른 값을 리턴해도 된다는 점은 조금 충격적입니다. 하지만, 이 조건을 강제할 수 없는 이유는 그 이유는 아래 그림처럼 CPU 캐시가 각 코어별로 존재하기 때문입니다.



코어 각각 L1, L2 캐시를 가지고 있다.

보시다시피, 각 코어가 각각 자신들의 L1, L2 캐시들을 갖고 있는 것을 알 수 있습니다. 따라서, 만약에 쓰레드 1 에서 $a = 5$ 을 한 후에 자신들의 캐시에만 기록해 놓고 다른 코어들에게 알리지 않는다면, 쓰레드 3 에서 a 의 값을 확인할 때, 5 를 얻는다는 보장이 없다는 이야기입니다.

물론, 매번 값을 기록할 때마다, 모든 캐시에 동기화를 시킬 수 있겠지만 동기화 작업은 시간을 꽤나 잡아먹는 일입니다. 다행이도, C++ 에서는 로우 레벨 언어 답게, 여러분들이 이를 세밀하게 조정할 수 있는 여러가지 도구들을 제공하고 있습니다.

원자성(atomicity)

앞서 이야기 했듯이, C++ 에서 모든 쓰레드들이 수정 순서에 동의해야만 하는 경우는 바로 모든 연산들이 원자적 일 때 라고 하였습니다.

원자적인 연산이 아닌 경우에는 모든 쓰레드에서 같은 수정 순서를 관찰할 수 있음이 보장되지 않기에 여러분이 직접 적절한 동기화 방법을 통해서 처리해야 합니다. 만일 이를 지키지 않는다면, 프로그램이 정의되지 않은 행동(undefined behavior)을 할 수 있습니다.

그렇다면 원자적이라는 것이 무엇일까요?

이미 이름에서 짐작하셨겠지만, 원자적 연산이란, CPU 가 명령어 1 개로 처리하는 명령으로, 중간에 다른 쓰레드가 끼어들 여지가 전혀 없는 연산을 말합니다. 즉, 이 연산을 반 정도 했다는 있을

수 없고 이 연산을 했다 혹은 안 했다 만 존재할 수 있습니다. 마치 원자처럼 쪼갤 수 없다 해서 **원자적(atomic)** 이라고 합니다.⁹⁾

C++ 에서는 몇몇 타입들에 원자적인 연산을 쉽게 할 수 있도록 여러가지 도구들을 지원하고 있습니다. 또한 이러한 원자적 연산들은 올바른 연산을 위해 굳이 뮤텍스가 필요하지 않습니다! 즉 속도가 더 빠릅니다.

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

void worker(std::atomic<int>& counter) {
    for (int i = 0; i < 10000; i++) {
        counter++;
    }
}

int main() {
    std::atomic<int> counter(0);

    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++) {
        workers.push_back(std::thread(worker, ref(counter)));
    }

    for (int i = 0; i < 4; i++) {
        workers[i].join();
    }

    std::cout << "Counter 최종 값 : " << counter << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

Counter 최종 값 : 40000

와 같이 잘 나옴을 알 수 있습니다.

```
std::atomic<int> counter(0);
```

9) 실제로는 원자를 쪼갤 수 있습니다.

`atomic`의 템플릿 인자로 원자적으로 만들고 싶은 타입을 전달하면 됩니다. 위 경우 0으로 초기화된 원자적인 변수를 정의하였습니다. `atomic` 객체에서 제공하는 함수들을 통해서, 여러가지 원자적인 연산들을 손쉽게 수행할 수 있습니다.

```
counter++;
```

놀랍게도 `counter ++;` 을 아무런 뮤텍스로 보호하지 않았음에도 불구하고, 정확히 Counter 가 40000으로 출력되었습니다. 원래 `counter ++` 을 하기 위해서는 CPU가 메모리에서 `counter`의 값을 읽고 -1 더하고 - 쓰는 총 3개의 단계를 거쳐야만 했습니다. 그런데, 여기서는 `lock` 없이도, 제대로 계산하였지요.

그렇다면 컴파일러는 이를 어떻게 원자적 연산으로 만들었을까요? 이를 알기 위해서는 다시 컴파일러가 어떤 어셈블리 코드를 생성했는지 살펴봐야 합니다.

```

1 std::thread::_State_impl<std::th
2 | mov rax, rdi
3 | mov rdi, QWORD PTR [rdi+8]
4 | jmp [QWORD PTR [rax+16]]
5 worker(std::atomic<int> &):
6 | mov eax, 10000
7 .L4:
8 lock add DWORD PTR [rdi], 1
9 sub eax, 1
10 jne .L4
11 ret
12 std::thread::_State_impl<std::th

```

붉은색 테두리가 `counter ++` 부분이다.

놀랍게도 `counter ++` 부분이 실제로 어셈블리 명령 한 줄인

```
lock add DWORD PTR [rdi], 1
```

로 나타남을 알 수 있습니다. 원래 CPU는 한 명령어에서 메모리에 읽기 혹은 쓰기 둘 중 하나 밖에 하지 못합니다. 메모리에 읽기 쓰기를 동시에 하는 명령은 없습니다. 하지만, 이 `lock add`의 경우 `rdi`에 위치한 메모리를 읽고 -1 더하고 - 다시 `rdi`에 위치한 메모리에 쓰기를 모두 해버립니다.

참고로 이러한 명령어를 컴파일러가 사용할 수 있었던 이유는 우리가 어느 CPU에서 실행할지 (x86) 컴파일러가 알고 있기 때문에 이런 CPU 특이적인 명령어를 제공할 수 있던 것입니다. 물론, CPU에 따라 위와 같은 명령이 없는 경우도 있습니다.

이 경우 CPU는 위와 같은 원자적인 코드를 생성할 수 없습니다. 이는 해당 `atomic` 객체의 연산들이 과연 정말로 원자적으로 구현될 수 있는지 확인하는 `is_lock_free()` 함수를 호출해보면 됩니다. 예를 들어서

```
std::atomic<int> x;
std::cout << "is lock free ? : " << boolalpha << x.is_lock_free() << std::endl;
```

를 실행해보면

실행 결과

```
Is lock free ? : true
```

와 같이 나옵니다. 여기서 *lock free* 의 `lock` 과 실제 어셈블리 명령에서의 `lock` 과는 다른 `lock` 을 의미합니다.

위 어셈블리 명령어에서의 `lock` 은 해당 명령을 원자적으로 수행하라는 의미로 사용되고, *lock free*에서의 `lock` 이 없다 라는 의미는 뮤텍스와 같은 객체들의 `lock`, `unlock` 없이도 해당 연산을 올바르게 수행할 수 있다는 뜻입니다.

memory_order

`atomic` 객체들의 경우 원자적 연산 시에 메모리에 접근할 때 어떠한 방식으로 접근하는지 지정할 수 있습니다.

memory_order_relaxed

`memory_order_relaxed` 는 가장 느슨한 조건입니다. 다시 말해, `memory_order_relaxed` 방식으로 메모리에서 읽거나 쓸 경우, 주위의 다른 메모리 접근들과 순서가 바뀌어도 무방합니다. 예를 들어서 아래 예제를 살펴봅시다.

```
#include <atomic>
#include <cstdio>
#include <thread>
#include <vector>
using std::memory_order_relaxed;

void t1(std::atomic<int>* a, std::atomic<int>* b) {
    b->store(1, memory_order_relaxed);      // b = 1 (쓰기)
    int x = a->load(memory_order_relaxed); // x = a (읽기)

    printf("x : %d \n", x);
}

void t2(std::atomic<int>* a, std::atomic<int>* b) {
    a->store(1, memory_order_relaxed);      // a = 1 (쓰기)
    int y = b->load(memory_order_relaxed); // y = b (읽기)

    printf("y : %d \n", y);
}

int main() {
    std::vector<std::thread> threads;
```

```

std::atomic<int> a(0);
std::atomic<int> b(0);

threads.push_back(std::thread(t1, &a, &b));
threads.push_back(std::thread(t2, &a, &b));

for (int i = 0; i < 2; i++) {
    threads[i].join();
}
}

```

성공적으로 컴파일 하였다면 아래와 같은 결과들을 확인할 수 있습니다.

실행 결과

```

x : 1
y : 0

```

혹은

실행 결과

```

x : 0
y : 1

```

혹은

실행 결과

```

y : 1
x : 1

```

을 말이지요.

```

b->store(1, memory_order_relaxed);      // b = 1 (쓰기)
int x = a->load(memory_order_relaxed); // x = a (읽기)

```

`store` 과 `load` 는 `atomic` 객체들에 대해서 원자적으로 쓰기와 읽기를 지원해주는 함수입니다. 이 때, 추가적인 인자로, 어떠한 형태로 `memory_order` 을 지정할 것인지 전달할 수 있는데, 우리의 경우 가장 느슨한 방식인 `memory_order_relaxed` 를 전달하였습니다.

여기서 잠깐 궁금한게 있습니다. 과연 아래와 같은 결과를 볼 수 있을까요?

실행 결과

```
x : 0
y : 0
```

상식적으로는 불가능 합니다. 왜냐하면 x , y 둘다 0 이 나오기 위해서는 $x = a$ 와 $y = b$ 시점에서 a 와 b 모두 0 이어야만 합니다. 하지만 위 명령어들이 순서대로 실행된다면 이는 불가능 하다는 사실을 알 수 있습니다.

예를 들어서 x 에 0이 들어가려면 a 가 0이어야 합니다. 이 말은 즉슨, $x = a$ 가 실행된 시점에서 $a = 1$ 이 실행되지 않았어야만 합니다. 따라서 $t2$ 에서 $y = b$ 를 할 때 이미 b 는 1인 상태이므로, y 는 반드시 1이 출력되어야 하지요.

하지만, 실제로는 그렇지 않습니다. `memory_order_relaxed` 는 앞서 말했듯이, 메모리 연산들 사이에서 어떠한 제약조건도 없다고 하였습니다. 다시 말해 서로 다른 변수의 `relaxed` 메모리 연산은 CPU 마음대로 재배치 할 수 있습니다 (단일 쓰레드 관점에서 결과가 동일하다면).

예를 들어서

```
int x = a->load(memory_order_relaxed); // x = a (읽기)
b->store(1, memory_order_relaxed); // b = 1 (쓰기)
```

순으로 CPU 가 순서를 재배치 하여 실행해도 무방하다는 뜻입니다.

그렇다면 이 경우 x 와 y 에 모두 0 이 들어가겠네요. `memory_order_relaxed` 는 CPU 에서 메모리 연산 순서에 관련해서 무한한 자유를 주는 것과 같습니다. 덕분에 CPU 에서 매우 빠른 속도로 실행할 수 있게됩니다.

이렇게 `relaxed` 메모리 연산을 사용하면 예상치 못한 결과를 나을 수 있지만, 종종 사용할 수 있는 경우가 있습니다.

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>
using std::memory_order_relaxed;

void worker(std::atomic<int>* counter) {
    for (int i = 0; i < 10000; i++) {
        // 다른 연산들 수행

        counter->fetch_add(1, memory_order_relaxed);
    }
}

int main() {
```

```

std::vector<std::thread> threads;

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

for (int i = 0; i < 4; i++) {
    threads.push_back(std::thread(worker, &counter));
}

for (int i = 0; i < 4; i++) {
    threads[i].join();
}

std::cout << "Counter : " << counter << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

Counter : 40000

와 같이 나옵니다. 여기서 중요한 부분은

```
counter->fetch_add(1, memory_order_relaxed);
```

로 이는 `counter ++` 와 정확히 하는 일이 동일하지만, `counter++` 와는 다르게 메모리 접근 방식을 설정할 수 있습니다. 위 문장 역시 원자적으로 `counter` 의 값을 읽고 - 1 을 더하고 - 다시 그 결과를 씁니다.

다만 `memory_order_relaxed` 를 사용할 수 있었던 이유는, 다른 메모리 연산들 사이에서 굳이 `counter` 를 증가시키는 작업을 재배치 못하게 막을 필요가 없기 때문입니다. 비록 다른 메모리 연산들 보다 `counter ++` 이 늦게 된다고 하더라도 결과적으로 증가 되기만 하면 문제 될 게 없기 때문입니다.

`memory_order_acquire` 과 `memory_order_release`

`memory_order_relaxed` 가 사용되는 경우가 있다고 하더라도 너무나 CPU 에 많은 자유를 부여하기에 그 사용 용도는 꽤나 제한적입니다. 이번에 살펴볼 것들은 `memory_order_relaxed` 보다 살짝 더 엄격한 친구들입니다.

아래와 같은 producer - consumer 관계를 생각해봅시다.

```
#include <atomic>
#include <iostream>
```

```
#include <thread>
#include <vector>
using std::memory_order_relaxed;

void producer(std::atomic<bool>* is_ready, int* data) {
    *data = 10;
    is_ready->store(true, memory_order_relaxed);
}

void consumer(std::atomic<bool>* is_ready, int* data) {
    // data 가 준비될 때 까지 기다린다.
    while (!is_ready->load(memory_order_relaxed)) {
    }

    std::cout << "Data : " << *data << std::endl;
}

int main() {
    std::vector<std::thread> threads;

    std::atomic<bool> is_ready(false);
    int data = 0;

    threads.push_back(std::thread(producer, &is_ready, &data));
    threads.push_back(std::thread(consumer, &is_ready, &data));

    for (int i = 0; i < 2; i++) {
        threads[i].join();
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

Data : 10

일반적인 경우 위와 같이 나옵니다. 하지만, 아래와 같은 결과를 얻을 수 도 있을까요?

실행 결과

Data : 0

있습니다! 왜냐하면 producer 쓰레드를 살펴보자면;

```
*data = 10;
is_ready->store(true, memory_order_relaxed);
```

위 `is_ready`에 쓰는 연산이 `relaxed`이기 때문에 위의 `*data = 10`과 순서가 바뀌어서 실행된다면 `is_ready`가 `true`임에도 `*data = 10`이 채 실행이 끝나지 않을 수 있다는 것이지요.

따라서 `consumer` 쓰레드에서 `is_ready`가 `true`가 되었음에도 제대로된 `data`를 읽을 수 없는 상황이 벌어집니다.

`consumer` 쓰레드에서도 마찬가지입니다.

```
while (!is_ready->load(memory_order_relaxed)) {
}

std::cout << "Data : " << *data << std::endl;
```

아래에 `data`를 읽는 부분과 위 `is_ready`에서 읽는 부분이 순서가 바뀌어 버린다면, `is_ready`가 `true`가 되기 이전의 `data` 값을 읽어버릴 수 있다는 문제가 생깁니다. 따라서 위와 같은 생산자 - 소비자 관계에서는 `memory_order_relaxed`를 사용할 수 없습니다.

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

void producer(std::atomic<bool>* is_ready, int* data) {
    *data = 10;
    is_ready->store(true, std::memory_order_release);
}

void consumer(std::atomic<bool>* is_ready, int* data) {
    // data 가 준비될 때 까지 기다린다.
    while (!is_ready->load(std::memory_order_acquire)) {
    }

    std::cout << "Data : " << *data << std::endl;
}

int main() {
    std::vector<std::thread> threads;

    std::atomic<bool> is_ready(false);
    int data = 0;

    threads.push_back(std::thread(producer, &is_ready, &data));
    threads.push_back(std::thread(consumer, &is_ready, &data));
}
```

```

for (int i = 0; i < 2; i++) {
    threads[i].join();
}
}

```

성공적으로 컴파일 하였다면

실행 결과

Data : 10

와 같이 나옵니다. 이 경우 data에 0이 들어가는 일은 불가능 합니다. 이유는 아래와 같습니다.

```

*data = 10;
is_ready->store(true, std::memory_order_release);

```

`memory_order_release`는 해당 명령 이전의 모든 메모리 명령들이 해당 명령 이후로 재배치 되는 것을 금지 합니다. 또한, 만약에 같은 변수를 `memory_order_acquire`으로 읽는 쓰레드가 있다면, `memory_order_release` 이전에 오는 모든 메모리 명령들이 해당 쓰레드에 의해서 관찰 될 수 있어야 합니다.

쉽게 말해 `is_ready->store(true, std::memory_order_release);` 밑으로 `*data = 10`이 올 수 없게 됩니다. 또한 `is_ready`가 `true`가 된다면, `memory_order_acquire`로 `is_ready`를 읽는 쓰레드에서 `data`의 값을 확인했을 때 10임을 관찰할 수 있어야 하죠.

```

while (!is_ready->load(std::memory_order_acquire)) {
}

```

실제로 consumer 쓰레드에서 `is_ready`를 `memory_order_acquire`로 `load`하고 있기에, `is_ready`가 `true`가 된다면, `data`는 반드시 10이어야만 합니다.

`memory_order_acquire`의 경우, `release`와는 반대로 해당 명령 뒤에 오는 모든 메모리 명령들이 해당 명령 위로 재배치 되는 것을 금지 합니다.

이와 같이 두 개의 다른 쓰레드들이 같은 변수의 `release`와 `acquire`를 통해서 동기화 (`synchronize`)를 수행하는 것을 볼 수 있습니다.

아래 예시를 보시면 좀더 이해가 잘 되실 것입니다.

```

#include <atomic>
#include <iostream>
#include <thread>

```

```
#include <vector>
using std::memory_order_relaxed;

std::atomic<bool> is_ready;
std::atomic<int> data[3];

void producer() {
    data[0].store(1, memory_order_relaxed);
    data[1].store(2, memory_order_relaxed);
    data[2].store(3, memory_order_relaxed);
    is_ready.store(true, std::memory_order_release);
}

void consumer() {
    // data 가 준비될 때 까지 기다린다.
    while (!is_ready.load(std::memory_order_acquire)) {
    }

    std::cout << "data[0] : " << data[0].load(memory_order_relaxed) << std::endl;
    std::cout << "data[1] : " << data[1].load(memory_order_relaxed) << std::endl;
    std::cout << "data[2] : " << data[2].load(memory_order_relaxed) << std::endl;
}

int main() {
    std::vector<std::thread> threads;

    threads.push_back(std::thread(producer));
    threads.push_back(std::thread(consumer));

    for (int i = 0; i < 2; i++) {
        threads[i].join();
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
data[0] : 1
data[1] : 2
data[2] : 3
```

와 같이 나옵니다.

```
data[0].store(1, memory_order_relaxed);
data[1].store(2, memory_order_relaxed);
```

```
data[2].store(3, memory_order_relaxed);
is_ready.store(true, std::memory_order_release);
```

여기서 `data`의 원소들을 `store` 하는 명령들은 모두 `relaxed` 때문에 자기들 끼리는 CPU에서 마음대로 재배치될 수 있겠지만, 아래 `release` 명령을 넘어가서 재배치될 수는 없습니다.



release - acquire 동기화

따라서 `consumer`에서 `data`들의 값을 확인했을 때 언제나 정확히 1, 2, 3이 들어있게 됩니다.

memory_order_acq_rel

`memory_order_acq_rel`은 이름에서도 알 수 있듯이, `acquire`와 `release`를 모두 수행하는 것입니다. 이는, 읽기와 쓰기를 모두 수행하는 명령들, 예를 들어서 `fetch_add`와 같은 함수에서 사용될 수 있습니다.

memory_order_seq_cst

`memory_order_seq_cst`는 메모리 명령의 순차적 일관성(sequential consistency)을 보장해줍니다.

순차적 일관성이란, 메모리 명령 재배치도 없고, 모든 쓰레드에서 모든 시점에 동일한 값을 관찰할 수 있는, 여러분이 생각하는 그대로 CPU가 작동하는 방식이라 생각하면 됩니다.

`memory_order_seq_cst`를 사용하는 메모리 명령들 사이에선 이러한 순차적 일관성을 보장해줍니다.

```
#include <atomic>
#include <iostream>
#include <thread>

std::atomic<bool> x(false);
std::atomic<bool> y(false);
std::atomic<int> z(0);

void write_x() { x.store(true, std::memory_order_release); }
```

```

void write_y() { y.store(true, std::memory_order_release); }

void read_x_then_y() {
    while (!x.load(std::memory_order_acquire)) {
    }
    if (y.load(std::memory_order_acquire)) {
        ++z;
    }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_acquire)) {
    }
    if (x.load(std::memory_order_acquire)) {
        ++z;
    }
}

int main() {
    thread a(write_x);
    thread b(write_y);
    thread c(read_x_then_y);
    thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    std::cout << "z : " << z << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

z : 2

혹은

실행 결과

z : 1

과 같이 나옵니다. 그렇다면

실행 결과

```
z : 0
```

은 발생할 수 있을까요?

일단, `write_x` 와 `read_x_then_y` 사이의 `release - acquire` 동기화와, `write_y` 와 `read_y_then_x` 사이의 `release - acquire` 동기화가 이루어지고 있음을 알 수 있습니다.

그렇다고 해서, `read_x_then_y` 와 `read_y_then_x` 두 쓰레드가 같은 순서로 `x.store` 와 `y.store` 를 관찰한다는 보장이 없습니다. 다시 말해 `read_x_then_y` 의 입장에서는 `x.store` 가 `y.store` 보다 먼저 발생해도 되고, `read_y_then_x` 입장에서는 `y.store` 가 `x.store` 보다 먼저 발생해도 된다는 것입니다.

이 경우 두 `if` 문 안의 `load` 가 `false` 가 되어서 `z` 가 0 이 되겠지요.

하지만 `memory_order_seq_cst` 를 사용하게 된다면, 해당 명령을 사용하는 메모리 연산들끼리는 모든 쓰레드에서 동일한 연산 순서를 관찰할 수 있도록 보장해줍니다. 참고로 우리가 `atomic` 객체를 사용할 때, `memory_order` 를 지정해주지 않는다면 디폴트로 `memory_order_seq_cst` 가 지정이 됩니다. 예컨대 이전에 `counter ++` 은 사실 `counter.fetch_add(1, memory_order_seq_cst)` 와 동일한 연산입니다.

문제는 멀티 코어 시스템에서 `memory_order_seq_cst` 가 꽤나 비싼 연산이라는 것입니다. 인텔 혹은 AMD 의 x86(-64) CPU 의 경우에는 사실 거의 순차적 일관성이 보장되서 `memory_order_seq_cst` 를 강제하더라도 그 차이가 그렇게 크지 않습니다. 하지만 ARM 계열의 CPU 와 같은 경우 순차적 일관성을 보장하기 위해서는 CPU 의 동기화 비용이 매우 큽니다. 따라서 해당 명령은 정말 꼭 필요 할 때만 사용해야 합니다.

```
#include <atomic>
#include <iostream>
#include <thread>
using std::atomic<bool>;
using std::thread;

std::atomic<bool> x(false);
std::atomic<bool> y(false);
std::atomic<int> z(0);

void write_x() { x.store(true, memory_order_seq_cst); }

void write_y() { y.store(true, memory_order_seq_cst); }

void read_x_then_y() {
    while (!x.load(memory_order_seq_cst)) {
    }
    if (y.load(memory_order_seq_cst)) {
        ++z;
    }
}
```

```
void read_y_then_x() {
    while (!y.load(memory_order_seq_cst)) {
    }
    if (x.load(memory_order_seq_cst)) {
        ++z;
    }
}

int main() {
    thread a(write_x);
    thread b(write_y);
    thread c(read_x_then_y);
    thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    std::cout << "z : " << z << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

z : 2

혹은

실행 결과

z : 1

과 같이 나옵니다. x.store 와 y.store 가 모두 memory_order_seq_cst 이므로, read_x_then_y 와 read_y_then_x 에서 관찰했을 때 x.store 와 y.store 가 같은 순서로 발생해야 합니다. 따라서 z 의 값이 0 이 되는 경우는 발생하지 않습니다.

정리해보자면 다음과 같습니다.

연산	허용된 memory order
쓰기 (store)	<code>memory_order_relaxed, memory_order_release, memory_order_seq_cst</code>
읽기 (load)	<code>memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_seq_cst</code>
읽고 - 수정하고 - 쓰기 (read - modify - write)	<code>memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel, memory_order_seq_cst</code>

참고로 `memory_order_consume` 은 다루지 않았는데 C++ 17 현재, `memory_order_consume`의 정의가 살짝 수정 중에 있기에 `memory_order_consume` 의 사용이 권장되지 않습니다.

이렇게 C++ 에서 `atomic` 연산들에 대해 `memory_order` 을 지정하는 방법에 대해 알아보았습니다. C++ `atomic` 객체들의 경우 따로 지정하지 않는다면 기본으로 `memory_order_seq_cst` 로 설정되는데, 이는 일부 CPU 에서 매우 값비싼 명령입니다. 만약에 제약 조건을 좀 더 느슨하게 할 수 있을 때 더 약한 수준의 `memory_order` 을 사용한다면 프로그램의 성능을 더 크게 향상 시킬 수 있습니다.

생각 해보기

문제 1

`std::atomic<bool>` 을 사용해서 `lock()` 과 `unlock()` 을 만들어보세요. 참고로 `compare_exchange_strong` 함수를 사용하는 것이 도움이 됩니다. `compare_exchange_strong` 은 아래와 같이 생겼습니다.

```
bool compare_exchange_strong(
    T& expected, T desired, std::memory_order order = std::memory_order_seq_cst);
```

만일 현재 `atomic` 객체의 값이 `expected` 와 같다면 `desired` 로 바꾸고 `true` 를 리턴합니다. `expected` 와 다르다면 `desired` 로 바꾸지 않고 그냥 `false` 를 리턴합니다. 물론 이 읽기 - 수정하기 - 쓰기 명령은 `atomic` 하게 실행됩니다.

문제 2

위는 `atomic_flag` 의 `test_and_set` 함수를 이용해서도 동일하게 만들 수 있습니다. 한 번 다시 만들어보세요! `atomic_flag` 는 `std::atomic<bool>` 과 비슷하게 `true` 혹은 `false` 만 가질 수 있지만, `atomic_flag` 는 `is_lock_free` 가 언제나 참임이 보장됩니다. 반면에 `std::atomic<bool>` 은 그렇지 않습니다. (정확히 말하자면 모든 `atomic` 객체들은 `is_lock_free` 가 참인 것이 보장되지 않습니다.)

문제 3

C++ `memory_order` 에 관련해서 매우 훌륭한 자료들이 인터넷에 많이 있습니다.

- [Preshing](#) 의 [블로그](#) 에는 `memory_order` 관련한 주옥같은 글들이 많이 있습니다.
- [cppreference](#) 도 꽤나 도움이 됩니다.
- [GCC Wiki](#) 도 설명을 잘해놓았습니다.

뭘 배웠지?

- 여러분의 코드는 여러분이 생각하는 순서로 작동하지 않습니다. (단일 쓰레드 관점에서) 결과값이 동일하다면 컴파일러와 CPU 는 명령어의 순서를 재배치 할 수 있습니다.
- 문제는 이렇게 마음대로 메모리 접근 명령어의 순서를 재배치 한다면 멀티 쓰레드 환경에서 그 결과가 달라질 수 있다는 점입니다. C++ 에서는 이와 같은 상황을 막기 위해서 메모리 재배치 순서를 강제할 수 있는 `memory_order` 라는 것을 제공합니다.
- C++ 에서 원자적 연산을 쉽게 할 수 있는 도구로 `atomic` 이란 것을 제공합니다.
- `atomic` 의 메모리 관련 연산에 적절한 `memory_order` 를 지정해서 올바른 결과를 이끌 어낼 수 있습니다.

비동기 연산을 위한 도구들

안녕하세요 여러분! 앞선 강좌를 통해서 C++에서 어떻게 쓰레드를 생성하고, 뮤텍스를 통해서 공유된 자원에서 경쟁 상태 (race condition)을 방지하고, 조건 변수 (`condition_variable`)을 통해서 생산자 - 소비자 패턴을 어떻게 구현하는지 알아보았습니다.

뿐만 아니라 `atomic` 객체를 통해 쉽게 원자적 연산을 수행하는 방법을 배웠고, 더 나아가서 `memory_order`을 통해 컴파일러가 어떤 식으로 명령어를 재배치 할지 설정하는 방법도 다루었습니다.

이번 강좌에서는 멀티 쓰레딩의 강력함을 더 쉽게 활용할 수 있게 해주는 몇 가지 도구들을 살펴보도록 하겠습니다.

동기 (synchronous) 와 비동기 (asynchronous) 실행

자바스크립트로 프로그램을 한 번이라도 짜신 분들은 비동기 (asynchronous) 작업이라는 단어를 수 없이 들어봤을 것입니다. 하지만 C++ 만 배우신 분들은 아직 많이 생소할 텐데 간단히 설명하면 다음과 같습니다.

예를 들어서 여러분이 하드 디스크에서 파일을 읽는다고 생각해봅시다. SSD 가 아니라, 하드 디스크를 사용한다면, 임의의 위치에 쓰여져 있는 파일을 읽는데 시간이 상당해 오래 걸립니다.



뾰족하게 생긴 장치가 바로 헤드이다.

왜냐하면 하드 디스크의 경우 헤드라고 부르는 장치가 디스크에 파일이 쓰여져 있는 실제 위치 까지 가야하기 때문이죠. 이는 하드 디스크에 있는 모터가 디스크를 돌려서 헤드를 정해진 구역에 위치 시킵니다.

보통 사용하는 7200rpm 하드 디스크의 경우 (여기서 rpm은 모터가 돌아가는 속도를 말합니다), 평균 4.17 밀리초가 걸린다고 합니다. 램에서 데이터를 읽어내는데 50 나노초가 걸리는 것에 비해 대략 8만배 정도 느린 셈입니다.

따라서 아래와 같은 코드를 생각해보면

```
string txt = read("a.txt");           // 5ms
string result = do_something_with_txt(txt); // 5ms

do_other_computation(); // 5ms 걸림 (CPU로 연산을 수행함)
```

만일 순차적으로 실행한다고 했을 때, 위 작업들이 모두 종료되는데 총 $5 + 5 + 5 = 15$ 밀리초가 걸리게 됩니다.

이러한 작업이 비효율적인 이유는 `read` 함수가 파일이 하드 디스크에서 읽어지는 동안 기다리기 때문입니다. 다시 말해 `read` 함수는 파일 읽기가 끝나기 전 까지 리턴하지 않고, CPU는 아무것도 하지 않은 채 가만히 기다리게 됩니다.

이렇게, 한 번에 하나씩 순차적으로 실행되는 작업을 동기적 (*synchronous*)으로 실행된다고 부릅니다. 동기적인 작업들은 한 작업이 끝날 때 까지 다음 작업으로 이동하지 않기 때문이지요.

만일 `read` 함수가 CPU를 계속 사용한다면, 동기적으로 작업을 수행해도 문제될 것이 없습니다. 하지만 실제로는 `read` 함수가 하드 디스크에서 데이터를 읽어오는 동안 CPU는 아무런 작업도 하지 않기 때문에, 그 시간에 오히려 CPU를 놀리지 않고 `do_other_computation`과 같은 작업을 수행하는 것이 더 바람직합니다.

그렇다면 이를 C++에서는 어떻게 구현할 수 있을까요? 아마 쓰레드를 배우신 여러분들은 아래와 같이 코드를 짤 수 있을 것입니다.

```
void file_read(string* result) {
    string txt = read("a.txt"); // (1)
    *result = do_something_with_txt(txt);
}

int main() {
    string result;
    thread t(file_read, &result);
    do_other_computation(); // (2)

    t.join();
}
```

위 코드의 수행 시간은 어떻게 될까요? 예를 들어서 쓰레드 `t`를 생성한 뒤에 바로 새로운 쓰레드에서 `file_read` 함수를 실행한다고 해봅시다.

`file_read` 함수 안에서 `read("a.txt")` 가 실행이 되는데, 이 때 CPU 는 하드 디스크에서 데이터를 기다리지 않고, 바로 다시 원래 `main` 함수 쓰레드로 넘어와서 `do_other_computation()` 을 수행하게 되겠지요.

5 밀리초 후에 `do_other_computation()` 이 끝나게 된다면, `t.join` 을 수행하면서 다시 `file_read` 쓰레드를 실행할 텐데, 이미 하드 디스크에서 `a.txt` 파일의 내용이 도착해있을 것이므로, `do_something_with_txt` 를 바로 실행하게 됩니다. 이 경우, 총 $5 + 5 = 10$ 밀리초 만에 수행이 끝나게 됩니다. CPU 는 단 한 순간도 놀지 않았습니다.

이와 같이 프로그램의 실행이, 한 갈래가 아니라 여러 갈래로 갈라져서 동시에 진행되는 것을 비동기적(asynchronous) 실행 이라고 부릅니다. 자바스크립트와 같은 언어들은 언어 차원에서 비동기적 실행을 지원하지만, C++ 의 경우 위와 같이 명시적으로 쓰레드를 생성해서 적절히 수행해야 했습니다.

하지만 C++ 11 표준 라이브러리를 통해 매우 간단히 비동기적 실행을 할 수 있게 해주는 도구를 제공하고 있습니다.

std::promise 와 std::future

결국 비동기적 실행으로 하고 싶은 일은, 어떠한 데이터를 다른 쓰레드를 통해 처리해서 받아내는 것입니다.

내가 어떤 쓰레드 `T` 를 사용해서, 비동기적으로 값을 받아내겠다 라는 의미는, 미래에 (future) 쓰레드 `T` 가 원하는 데이터를 돌려 주겠다 라는 약속 (promise) 라고 볼 수 있습니다.

이 문장을 그대로 코드로 옮겨보면 아래와 같습니다.

```
#include <future>
#include <iostream>
#include <string>
#include <thread>
using std::string;

void worker(std::promise<string>* p) {
    // 약속을 이행하는 모습. 해당 결과는 future 에 들어간다.
    p->set_value("some data");
}

int main() {
    std::promise<string> p;

    // 미래에 string 데이터를 돌려 주겠다는 약속.
    std::future<string> data = p.get_future();

    std::thread t(worker, &p);

    // 미래에 약속된 데이터를 받을 때 까지 기다린다.
}
```

```

    data.wait();

    // wait 이 리턴했다는 뜻이 future 에 데이터가 준비되었다는 의미.
    // 참고로 wait 없이 그냥 get 해도 wait 한 것과 같다.
    std::cout << "받은 데이터 : " << data.get() << std::endl;

    t.join();
}

```

성공적으로 컴파일 하였다면

실행 결과

받은 데이터 : some data

와 같이 나옵니다.

```
std::promise<string> p;
```

먼저 promise 객체를 살펴봅시다. promise 객체를 정의할 때, 연산을 수행 후에 돌려줄 객체의 타입을 템플릿 인자로 받습니다. 우리의 경우 string 객체를 돌려줄 예정이므로 string 을 전달하였습니다.

연산이 끝난 다음에 promise 객체는 자신이 가지고 있는 future 객체에 값을 넣어주게 됩니다. 이 때 promise 객체에 대응되는 future 객체는

```
std::future<string> data = p.get_future();
```

위와 같이 get_future 함수를 통해서 얻을 수 있습니다. 하지만 data 가 아직은 실제 연산 결과를 포함하고 있는 것은 아닙니다. data 가 실제 결과를 포함하기 위해서는

```
p->set_value("some data");
```

위와 같이 promise 객체가 자신의 future 객체에 데이터를 제공한 후에;

```

// 미래에 약속된 데이터를 받을 때 까지 기다린다.
data.wait();

// wait 이 리턴했다는 뜻이 future 에 데이터가 준비되었다는 의미.
// 참고로 wait 없이 그냥 get 해도 wait 한 것과 같다.
std::cout << "받은 데이터 : " << data.get() << std::endl;

```

대응되는 `future` 객체의 `get` 함수를 통해 얻어낼 수 있습니다. 한 가지 중요한 점은 `promise`가 `future`에 값을 전달하기 전 까지 `wait` 함수가 기다린다는 점입니다. `wait` 함수가 리턴을 하였다면 `get`을 통해서 `future`에 전달된 객체를 얻을 수 있습니다.

참고로 굳이 `wait` 함수를 따로 호출할 필요는 없는데, `get` 함수를 바로 호출하더라도 알아서 `promise`가 `future`에 객체를 전달할 때 까지 기다린 다음에 리턴합니다. 참고로 `get`을 호출하면 `future` 내에 있던 데이터가 이동 됩니다. 따라서 `get`을 다시 호출하면 안됩니다.

주의 사항

`future`에서 `get`을 호출하면, 설정된 객체가 이동 됩니다. 따라서 절대로 `get`을 두 번 호출하면 안됩니다.

정리해 보자면 **promise**는 생산자-소비자 패턴에서 마치 생산자 (producer)의 역할을 수행하고, **future**는 소비자 (consumer)의 역할을 수행한다고 보면 됩니다.

따라서 아래와 같이 조건 변수를 통해서도 `promise` - `future` 패턴을 구현할 수 있습니다.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

std::condition_variable cv;
std::mutex m;
bool done = false;
std::string info;

void worker() {
    {
        std::lock_guard<std::mutex> lk(m);
        info = "some data"; // 위의 p->set_value("some data")에 대응
        done = true;
    }
    cv.notify_all();
}

int main() {
    std::thread t(worker);

    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [] { return done; }); // 위의 data.wait()이라 보면 된다.
    lk.unlock();

    std::cout << "받은 데이터 : " << info << std::endl;
    t.join();
```

{}

하지만, `promise` 와 `future` 를 이용하는 것이 훨씬 더 깔끔하고 더 이해하기도 쉽습니다. 또한 위 조건 변수를 사용한 것 보다 더 우수한 점은 `future` 에 예외도 전달할 수 있기 때문입니다. 예를 들어서 아래와 같은 코드를 살펴봅시다.

```
#include <exception>
#include <future>
#include <iostream>
#include <string>
#include <thread>
using std::string;

void worker(std::promise<string>* p) {
    try {
        throw std::runtime_error("Some Error!");
    } catch (...) {
        // set_exception 에는 exception_ptr 를 전달해야 한다.
        p->set_exception(std::current_exception());
    }
}
int main() {
    std::promise<string> p;

    // 미래에 string 데이터를 돌려 주겠다는 약속.
    std::future<string> data = p.get_future();

    std::thread t(worker, &p);

    // 미래에 약속된 데이터를 받을 때 까지 기다린다.
    data.wait();

    try {
        data.get();
    } catch (const std::exception& e) {
        std::cout << "예외 : " << e.what() << std::endl;
    }
    t.join();
}
```

성공적으로 컴파일 하였다면

실행 결과

예외 : Some Error!

위와 같이 예외가 제대로 전달되었음을 알 수 있습니다.

```
p->set_exception(current_exception());
```

참고로 `set_exception`에는 예외 객체가 아니라 `exception_ptr`을 전달해야 합니다. 이 `exception_ptr`는 `catch`로 받은 예외 객체의 포인터가 아니라, 현재 `catch` 된 예외에 관한 정보를 반환하는 `current_exception` 함수가 리턴하는 객체입니다.

물론, `catch`로 전달받은 예외 객체를 `make_exception_ptr` 함수를 사용해서 `exception_ptr`로 만들 수도 있지만, 그냥 편하게 `current_exception`을 호출하는 것이 더 간단합니다. 이렇게 `future`에 전달된 예외 객체는

```
try {
    data.get();
} catch (const std::exception& e) {
    std::cout << "예외 : " << e.what() << std::endl;
}
```

위와 같이 `get` 함수를 호출하였을 때, 실제로 `future`에 전달된 예외 객체가 던져지고, 마치 `try`와 `catch` 문을 사용한 것처럼 예외를 처리할 수 있게 됩니다. 매우 간단하지요.

wait_for

그냥 `wait`을 하였다면 `promise`가 `future`에 전달할 때 까지 기다리게 됩니다. 하지만 `wait_for`을 사용하면, 정해진 시간 동안만 기다리고 그냥 진행할 수 있습니다.

```
#include <chrono>
#include <exception>
#include <future>
#include <iostream>
#include <string>
#include <thread>

void worker(std::promise<void>* p) {
    std::this_thread::sleep_for(std::chrono::seconds(10));
    p->set_value();
}

int main() {
    // void의 경우 어떠한 객체도 전달하지 않지만, future 가 set 이 되었나
    // 안되었느냐의 유무로 마치 플래그의 역할을 수행할 수 있습니다.
    std::promise<void> p;

    // 미래에 string 데이터를 돌려 주겠다는 약속.
    std::future<void> data = p.get_future();

    std::thread t(worker, &p);
```

```
// 미래에 약속된 데이터를 받을 때 까지 기다린다.  
while (true) {  
    std::future_status status = data.wait_for(std::chrono::seconds(1));  
  
    // 아직 준비가 안됨  
    if (status == std::future_status::timeout) {  
        std::cerr << ">";  
    }  
    // promise 가 future 를 설정함.  
    else if (status == std::future_status::ready) {  
        break;  
    }  
}  
t.join();  
}
```

성공적으로 컴파일 하였다면



와 같이 나옵니다.

```
std::future_status status = data.wait_for(std::chrono::seconds(1));  
  
// 아직 준비가 안됨  
if (status == std::future_status::timeout) {  
    cerr << ">";  
}  
// promise 가 future 를 설정함.  
else if (status == std::future_status::ready) {  
    break;  
}
```

`wait_for` 함수는 `promise` 가 설정될 때 까지 기다리는 대신에 `wait_for` 에 전달된 시간 만큼 기다렸다가 바로 리턴해버립니다. 이 때 리턴하는 값은 현재 `future` 의 상태를 나타내는 `future_status` 객체입니다.

`future_status` 는 총 3 가지 상태를 가질 수 있습니다. 먼저 `future`에 값이 설정 됐을 때 나타나는 `future_status::ready` 가 있고, `wait_for`에 지정한 시간이 지났지만 값이 설정되지 않아서 리턴한 경우에는 `future_status::timeout` 이 리턴됩니다.

마지막으로 `future_status::deferred` 가 있는데 이는 결과값을 계산하는 함수가 채 실행되지 않았다는 의미인데, 뒤에서 좀더 자세히 다루도록 하겠습니다.

shared_future

앞서 `future`의 경우 딱 한 번만 `get` 을 할 수 있다고 하였습니다. 왜냐하면 `get` 을 호출하면 `future` 내부의 객체가 이동되기 때문이지요. 하지만, 종종 여러 개의 다른 쓰레드에서 `future` 를 `get` 할 필요성이 있습니다.

이 경우 `shared_future` 를 사용하면 됩니다. 아래 예제는 달리기를 하는 것을 C++ 프로그램으로 나타내본 것입니다. `main` 함수에서 출발 신호를 보내면 각 `runner` 쓰레드들에서 달리기를 시작하게 됩니다.

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>
using std::thread;

void runner(std::shared_future<void>* start) {
    start->get();
    std::cout << "출발!" << std::endl;
}

int main() {
    std::promise<void> p;
    std::shared_future<void> start = p.get_future();

    thread t1(runner, &start);
    thread t2(runner, &start);
    thread t3(runner, &start);
    thread t4(runner, &start);

    // 참고로 cerr 는 std::cout 과는 다르게 버퍼를 사용하지 않기 때문에 터미널에 바로 출력된다.
    std::cerr << "준비..." ;
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cerr << "땅!" << std::endl;

    p.set_value();

    t1.join();
    t2.join();
    t3.join();
}
```

```
t4.join();
}
```

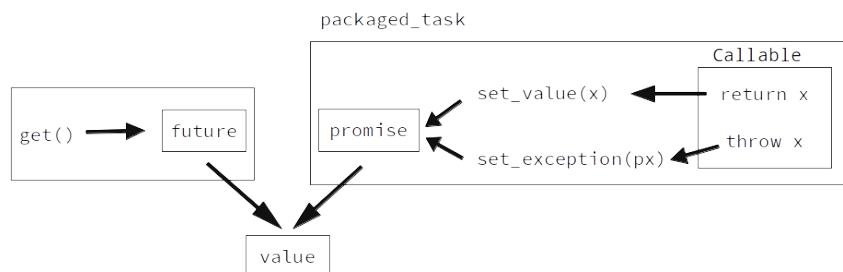
성공적으로 컴파일 하였다면



와 같이 잘 나옵니다. 위 코드 역시 `condition_variable` 을 이용해서 동일하게 작성할 수 있습니다. 하지만 보시다시피, `future` 를 사용하는 것이 훨씬 편리합니다.

packaged_task

C++ 에서는 위 `promise-future` 패턴을 비동기적 함수(정확히는 Callable - 즉 람다 함수, Functor 포함)의 리턴값에 간단히 적용할 수 있는 `packaged_task` 라는 것을 지원합니다.



TCPL에서 가져옴

`packaged_task` 에 전달된 함수가 리턴할 때, 그 리턴값을 `promise` 에 `set_value` 하고, 만약에 예외를 던졌다면 `promise` 에 `set_exception` 을 합니다. 해당 `future` 는 `packaged_task` 가 리턴하는 `future` 에서 접근할 수 있습니다. 아래 예제를 한편 살펴봅시다.

```
#include <future>
#include <iostream>
#include <thread>
```

```

int some_task(int x) { return 10 + x; }

int main() {
    // int(int) : int 를 리턴하고 인자로 int 를 받는 함수. (std::function 참조)
    std::packaged_task<int(int)> task(some_task);

    std::future<int> start = task.get_future();

    std::thread t(std::move(task), 5);

    std::cout << "결과값 : " << start.get() << std::endl;
    t.join();
}

```

성공적으로 컴파일 하였다면

실행 결과

결과값 : 15

와 같이 잘 나옵니다.

```

std::packaged_task<int(int)> task(some_task);
std::future<int> start = task.get_future();

```

`packaged_task` 는 비동기적으로 수행할 함수 자체를 생성자의 인자로 받습니다. 또한 템플릿 인자로 해당 함수의 타입을 명시해야 합니다. `packaged_task` 는 전달된 함수를 실행해서, 그 함수의 리턴값을 `promise` 에 설정합니다.

해당 `promise` 에 대응되는 `future` 는 위와 같이 `get_future` 함수로 얻을 수 있습니다.

```
thread t(std::move(task), 5);
```

생성된 `packaged_task` 를 쓰레드에 전달하면 됩니다. 참고로 `packaged_task` 는 복사 생성이 불가능하므로 (`promise` 도 마찬가지입니다.) 명시적으로 `move` 해줘야만 합니다.

```
std::cout << "결과값 : " << start.get() << std::endl;
```

비동기적으로 실행된 함수의 결과값은 추후에 `future` 의 `get` 함수로 받을 수 있게 됩니다. 이와 같이 `packaged_task` 를 사용하게 된다면 쓰레드에 굳이 `promise` 를 전달하지 않아도 알아서 `packaged_task` 가 함수의 리턴값을 처리해줘서 매우 편리합니다.

std::async

앞서 `promise` 나 `packaged_task` 는 비동기적으로 실행을 하기 위해서는, 쓰레드를 명시적으로 생성해서 실행해야만 했습니다. 하지만 `std::async` 에 어떤 함수를 전달한다면, 아예 쓰레드를 알아서 만들어서 해당 함수를 비동기적으로 실행하고, 그 결과값을 `future` 에 전달합니다.

```
#include <future>
#include <iostream>
#include <thread>
#include <vector>

// std::accumulate 와 동일
int sum(const std::vector<int>& v, int start, int end) {
    int total = 0;
    for (int i = start; i < end; ++i) {
        total += v[i];
    }
    return total;
}

int parallel_sum(const std::vector<int>& v) {
    // lower_half_future 는 1 ~ 500 까지 비동기적으로 더함
    // 참고로 람다 함수를 사용하면 좀 더 깔끔하게 표현할 수도 있다.
    // --> std::async([&v]() { return sum(v, 0, v.size() / 2); });
    std::future<int> lower_half_future =
        std::async(std::launch::async, sum, cref(v), 0, v.size() / 2);

    // upper_half 는 501 부터 1000 까지 더함
    int upper_half = sum(v, v.size() / 2, v.size());

    return lower_half_future.get() + upper_half;
}

int main() {
    std::vector<int> v;
    v.reserve(1000);

    for (int i = 0; i < 1000; ++i) {
        v.push_back(i + 1);
    }

    std::cout << "1 부터 1000 까지의 합 : " << parallel_sum(v) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1 부터 1000 까지의 합 : 500500
```

와 같이 잘 나옵니다.

```
std::future<int> lower_half_future =
    std::async(std::launch::async, sum, cref(v), 0, v.size() / 2);
```

`async` 함수는 인자로 받은 함수를 비동기적으로 실행한 후에, 해당 결과값을 보관할 `future` 를 리턴합니다. 첫 번째 인자로는 어떠한 형태로 실행할지를 전달하는데 두 가지 값이 가능합니다.

- `std::launch::async` : 바로 쓰레드를 생성해서 인자로 전달된 함수를 실행한다.
- `std::launch::deferred` : `future` 의 `get` 함수가 호출되었을 때 실행한다. (새로운 쓰레드를 생성하지 않음)

즉 `launch::async` 옵션을 주면 바로 그 자리에서 쓰레드를 생성해서 실행하게 되고, `launch::deferred` 옵션을 주면, `future` 의 `get` 을 하였을 때 비로소 (동기적으로) 실행하게 됩니다. 다시 말해, 해당 함수를 굳이 바로 당장 비동기적으로 실행할 필요가 없다면 `deferred` 옵션을 주면 됩니다.

`async` 함수는 실행하는 함수의 결과값을 포함하는 `future` 를 리턴합니다. 그 결과값은

```
return lower_half_future.get() + upper_half;
```

`async` 함수가 리턴한 `future` 에 `get` 을 통해 얻어낼 수 있습니다.

위 `parallel` 함수는 1 부터 1000 까지의 덧셈을 총 2 개의 쓰레드에서 실행한다고 보면 됩니다. 1 부터 500 까지의 합은, `async` 를 통해 생성된 새로운 쓰레드에서 더하게 되고, 나머지 501 부터 1000 까지의 합은 원래의 쓰레드에서 처리하게 되죠.

물론 위 1 부터 1000 까지의 합은 금방 처리되기 때문에 큰 차이는 나지 않지만, CPU 를 많이 사용하는 작업을 두 개의 쓰레드에서 나눠 처리한다면 (CPU 가 멀티 코어임을 가정할 때) 2 배 빠르게 작업을 수행할 수 있습니다.

예를 들어서 아래 예제를 살펴봅시다.

```
#include <future>
#include <iostream>
#include <thread>

int do_work(int x) {
    // x 를 가지고 무슨 일을 한다.
```

```

    std::this_thread::sleep_for(std::chrono::seconds(3));
    return x;
}

void do_work_parallel() {
    auto f1 = std::async([]() { do_work(3); });
    auto f2 = std::async([]() { do_work(3); });
    do_work(3);

    f1.get();
    f2.get();
}

void do_work_sequential() {
    do_work(3);
    do_work(3);
    do_work(3);
}

int main() { do_work_parallel(); }

```

성공적으로 컴파일 하였다면

```

~/Teach ➤ time ./test
./test 0.00s user 0.00s system 0% cpu 3.004 total

```

time 은 프로그램의 실행 시간과 CPU 사용률을 간단히 측정할 수 있는 프로그램입니다. 맨 마지막의 total 이 프로그램 총 실행 시간입니다.

위와 같이 실행하는데 총 3초가 걸리는 것을 알 수 있습니다. 이는 총 실행하는데 3 초가 걸리는 do_work 함수를 아래와 같이 비동기적으로 호출하였기 때문입니다.

```

auto f1 = std::async([]() { do_work(3); });
auto f2 = std::async([]() { do_work(3); });
do_work(3);

```

즉 3 개의 do_work 함수를 동시에 각기 다른 쓰레드에서 실행한 덕분에 3 초 만에 끝났습니다. 반면에 동기적으로 하나씩 실행하였다면

```

#include <future>
#include <iostream>
#include <thread>

int do_work(int x) {
    // x 를 가지고 무슨 일을 한다.
    std::this_thread::sleep_for(std::chrono::seconds(3));
    return x;
}

```

```

}

void do_work_parallel() {
    auto f1 = std::async([]() { do_work(3); });
    auto f2 = std::async([]() { do_work(3); });
    do_work(3);

    f1.get();
    f2.get();
}

void do_work_sequential() {
    do_work(3);
    do_work(3);
    do_work(3);
}

int main() { do_work_sequential(); }

```

성공적으로 컴파일 하였다면

```

~/Teach ➤ time ./test
./test 0.00s user 0.00s system 0% cpu 9.004 total

```

위와 같이 총 $3 + 3 + 3 = 9$ 초가 걸림을 알 수 있습니다.

이처럼 C++에서 제공하는 `promise`, `future`, `packaged_task`, `async`를 잘 활용하면 귀찮게 `mutex`나 `condition_variable`을 사용하지 않고도 매우 편리하게 비동기적 작업을 수행할 수 있습니다. 그렇다면 이번 강좌는 여기서 마치도록 하겠습니다.

다음 강좌에서는 여태까지 배운 것들을 총 동원해서 `ThreadPool`을 만들어보겠습니다.

생각 해보기

문제 1

`async`를 사용해서 기존의 `find$algorithm`를 더 빠르게 수행하는 함수를 만들어보세요.

문제 2

쓰레드풀(ThreadPool)이란 말 그대로 쓰레드들의 작업 소개소라고 보면 됩니다. 여기에 일거리(함수)를 던져주면, 작업 소개소에 있던 쓰레드 하나가 그 일감을 받아서 수행하게 됩니다. 그 일을 다 수행한 쓰레드는 다시 작업 소개소로 돌아오죠.

쓰레드풀의 사용자는 원하는 만큼의 쓰레드들을 생성해놓고, 무언가 수행하고 싶은 일이 있다면 그냥 쓰레드풀에 추가하면 됩니다.

그렇다면 한 번 `ThreadPool` 클래스를 설계하고 만들어보세요. 다음 강좌에서 같이 만들겠지만, 먼저 혼자 자기 힘으로 만들어보는 것이 중요합니다.

뭘 배웠지?

- 한 번 발생하는 이벤트에 대해서 `promise - future` 패턴을 이용하면 간단하게 처리할 수 있습니다.
- `shared_future` 를 사용해서 여러 개의 쓰레드를 한꺼번에 관리할 수 있습니다.
- `packaged_task` 를 통해서 원하는 함수의 `promise` 와 `future` 패턴을 손쉽게 생성할 수 있습니다.
- `async` 를 사용하면 원하는 함수를 비동기적으로 실행할 수 있습니다.

ThreadPool 만들기

안녕하세요 여러분! 이번 강좌에서는 여태까지 배운 내용들을 총 활용해서 쓰레드풀(ThreadPool)을 만들어보겠습니다. 이 쓰레드풀 구현은 [여기](#)를 기반으로 작성하였습니다.¹⁰⁾

쓰레드풀이란, 쓰레드들을 위한 작업 소개소라고 보시면 됩니다. 여러 개의 쓰레드들이 대기하고 있다가, 할 일이 들어오게 되면, 대기하고 있던 쓰레드들 중 하나가 이를 받아서 실행하게 됩니다.

예를 들어서 서버의 경우, 클라이언트 (사용자) 에서 요청이 들어오면 해당 요청에 대한 처리를 쓰레드풀에 추가만 하면 됩니다. 그러면 나중에 쓰레드들 중 하나가 처리를 하게 되겠지요. 물론 모든 쓰레드들이 다 다른 것들을 처리하고 있어도 괜찮습니다.

보통 이를 구현하는 아이디어는 간단합니다. 처리해야 될 작업들을 큐(queue)에 추가하는 것입니다. 큐는 그냥 링크드리스트 라고 생각하면 편한데, push 를 하게 되면 큐 맨 뒤에 작업을 추가하게 됩니다. 그 다음에 pop 을 하면 맨 앞에 있는 작업을 빼버리게 됩니다.

참고로 C++ 에서 제공하는 queue 의 경우 pop 을 하면 맨 앞의 원소를 제거하지만 해당 원소를 리턴하지 않습니다. 해당 원소에 접근하기 위해서는 front 를 호출해야 합니다.

큐를 사용하면 가장 먼저 추가된 작업을 가장 먼저 처리를 시작할 수 있습니다. 다시 말해 가장 오래된 작업 요청을 먼저 처리하는 방식이라고 보면 됩니다. 가장 상식적인 방식이기도 한데, 때때론 가장 최근에 추가된 작업 요청을 먼저 처리해야 할 때도 있습니다. 이 경우 queue 대신에 다른 자료 구조를 이용하는 것이 좋습니다.

클래스 설계 하기

그렇다면 먼저 이 ThreadPool 클래스에 무엇이 필요할지 생각해봅시다.

먼저 당연하게도 쓰레드들을 잔뜩 보관할 컨테이너가 필요 합니다.

```
// 총 Worker 쓰레드의 개수.
size_t num_threads_;
// Worker 쓰레드를 보관하는 벡터.
std::vector<std::thread> worker_threads_;
```

위와 같이 쓰레드들을 보관하는 worker_threads_ 라는 벡터를 만듭시다. 참고로 우리의 쓰레드풀에서 돌아가는 쓰레드들을 편의상 Worker 쓰레드 라고 부르도록 하겠습니다. num_threads_ 는 전체 쓰레드의 개수를 보관하는 멤버 변수입니다. 물론 해당 값은 worker_threads_.size() 와 같겠지요.

10) 이 강좌에서 정말 여태까지 배운 내용들을 주르륵 활용하므로 좋은 복습이 될 것이라 생각합니다.

그렇다면 이제 작업들을 어떻게 저장할지 생각해야 합니다. 쓰레드풀 사용자는 실행을 원하는 함수들을 쓰레드풀에 전달할 것입니다. 하지만 C++에는 안타깝게도 일반적인 타입의 함수 포인터를 저장할 수 있는 컨테이너는 없습니다.

따라서 일단은 `void` 형의 인자를 받지 않는 함수를 전달한다고 가정하겠습니다. 강좌 뒷부분에서 어떻게 하면 임의의 타입을 받는 함수들도 처리할 수 있을지 다룰 것입니다.

작업을 보관할 컨테이너는 아래와 같습니다.

```
// 할일들을 보관하는 job 큐.  
std::queue<std::function<void()>> jobs_;
```

앞서 말했듯이 작업을 보관하는 컨테이너는 큐를 사용한다고 하였습니다. 큐를 사용해서 가장 오래 전에 추가된 작업을 쉽게 알아낼 수 있습니다.

해당 큐는 모든 작업 쓰레드들에서 접근 가능한 큐입니다. 또한, 쓰레드풀 사용자들도 작업들을 각기 다른 쓰레드들에서 쓰레드풀에 추가할 수도 있습니다. 하지만 `queue`는 멀티 쓰레드 환경에서 안전하지 않기 때문에 이 `queue`를 `race condition`에서 보호할 장치들이 필요합니다.

```
std::condition_variable cv_job_q_;  
std::mutex m_job_q_;
```

`cv_job_q_` 와 `m_job_q_` 는 생산자-소비자 패턴을 구현할 때 사용됩니다. 여기서 생산자 역할은 쓰레드풀을 사용하는 사용자들이고 (`jobs_`에 작업을 추가하는 사람들), 소비자들은 `Worker` 쓰레드들이겠지요.

마지막으로 `Worker` 쓰레드들을 종료시킬 조건을 나타내는 멤버 변수인

```
// 모든 쓰레드 종료  
bool stop_all;
```

가 필요 합니다. `Worker` 쓰레드들은 기본적으로 `jobs_` 들을 처리하는 동안 무한 루프를 돌고 있는데, 위 `stop_all` 이 설정 된다면 무한 루프를 빠져나가게 됩니다.

ThreadPool 첫 번째 버전

그렇다면 `ThreadPool` 의 구현을 먼저 살펴보도록 하겠습니다. 먼저 생성자는 간단합니다. `worker_threads_` 에 쓰레드를 시작시켜주기만 하면 됩니다.

```
ThreadPool::ThreadPool(size_t num_threads)  
: num_threads_(num_threads), stop_all(false) {
```

```

    worker_threads_.reserve(num_threads_);
    for (size_t i = 0; i < num_threads_; ++i) {
        worker_threads_.emplace_back([this]() { this->WorkerThread(); });
    }
}

```

위와 같이 `num_threads_` 개의 쓰레드를 생성하게 됩니다. 이 때 각 쓰레드들은 `ThreadPool`에 정의된 `WorkerThread` 함수를 실행하게 됩니다. 참고로, 외부에서 멤버 함수에 접근하기 위해서는 이전에 이야기 하였듯이 `mem_fn` 으로 감싸거나, 람다 함수를 이용하면 되는데 여기서는 간단히 멤버 함수를 사용하였습니다.

물론 람다 안에서 멤버 함수에 접근하기 위해서는 `this` 를 전달해줘야 합니다. 그리고 람다 함수 안에서 `this->WorkerThread()` 를 통해 멤버 함수를 실행할 수 있습니다.

그렇다면 `WorkerThread` 에서는 무슨 일을 해야 할까요? 간단합니다. `jobs_` 에 작업이 추가될 때 까지 대기하고 있다가, 작업이 추가되면 받아서 처리하면 됩니다. 따라서 아래와 같이 구현할 수 있습니다.

```

void ThreadPool::WorkerThread() {
    while (true) {
        std::unique_lock<std::mutex> lock(m_job_q_);
        cv_job_q_.wait(lock, [this]() { return !this->jobs_.empty() || stop_all; });
        if (stop_all && this->jobs_.empty()) {
            return;
        }

        // 맨 앞의 job 을 뺀다.
        std::function<void()> job = std::move(jobs_.front());
        jobs_.pop();
        lock.unlock();

        // 해당 job 을 수행한다 :
        job();
    }
}

```

조건 변수 `cv_job_q_` 에서 `jobs_` 에 원소가 있거나, `stop_all` 이 설정될 때 까지 기다립니다. 만약에 모든 작업들이 설정되어 있고 `jobs_` 에 대기중인 작업이 없을 때 비로소 쓰레드를 종료하게 됩니다 (일이 없을 때 까지 퇴근을 못하는 슬픈 현실을 감안한 구현입니다.)

처리할 일이 있다면 간단히 `jobs_.front()` 를 통해 가장 오래전에 추가된 작업을 얻은 뒤에 해당 작업을 실행하면 됩니다.

그렇다면 작업을 추가하는 함수를 어찌까요?

```

void ThreadPool::EnqueueJob(std::function<void()> job) {
    if (stop_all) {

```

```

    throw std::runtime_error("ThreadPool 사용 중지됨");
}
{
    std::lock_guard<std::mutex> lock(m_job_q_);
    jobs_.push(std::move(job));
}
cv_job_q_.notify_one();
}

```

크게 복잡하지 않습니다. 일단 이미 `stop_all` 이 설정된 상태라면 더이상 작업을 추가하면 안되기에 예외를 던지도록 하였습니다. 그렇지 않을 경우 간단히 작업을 추가한 뒤에 자고 있는 쓰레드 하나만 깨워주면 됩니다.

마지막으로 소멸자는 아래와 같습니다.

```

ThreadPool::~ThreadPool() {
    stop_all = true;
    cv_job_q_.notify_all();

    for (auto& t : worker_threads_) {
        t.join();
    }
}

```

`stop_all` 을 설정한 뒤에, 모든 `Worker` 쓰레드들에 알려줍니다. 그 후 모든 쓰레드들을 `join`하면 됩니다.

전체 코드를 보면 아래와 같습니다.

전체 구현 (1)

```

#include <chrono>
#include <condition_variable>
#include <cstdio>
#include <functional>
#include <mutex>
#include <queue>
#include <thread>
#include <vector>

namespace ThreadPool {
class ThreadPool {
public:
    ThreadPool(size_t num_threads);
    ~ThreadPool();
}

```

```
// job 을 추가한다.
void EnqueueJob(std::function<void()> job);

private:
// 총 Worker 쓰레드의 개수.
size_t num_threads_;
// Worker 쓰레드를 보관하는 벡터.
std::vector<std::thread> worker_threads_;
// 할일들을 보관하는 job 큐.
std::queue<std::function<void()>> jobs_;
// 위의 job 큐를 위한 cv 와 m.
std::condition_variable cv_job_q_;
std::mutex m_job_q_;

// 모든 쓰레드 종료
bool stop_all;

// Worker 쓰레드
void WorkerThread();
};

ThreadPool::ThreadPool(size_t num_threads)
: num_threads_(num_threads), stop_all(false) {
worker_threads_.reserve(num_threads_);
for (size_t i = 0; i < num_threads_; ++i) {
    worker_threads_.emplace_back([this]() { this->WorkerThread(); });
}
}

void ThreadPool::WorkerThread() {
while (true) {
    std::unique_lock<std::mutex> lock(m_job_q_);
    cv_job_q_.wait(lock, [this]() { return !this->jobs_.empty() || stop_all; });
    if (stop_all && this->jobs_.empty()) {
        return;
    }

    // 맨 앞의 job 을 뺀다.
    std::function<void()> job = std::move(jobs_.front());
    jobs_.pop();
    lock.unlock();

    // 해당 job 을 수행한다 :)
    job();
}
}

ThreadPool::~ThreadPool() {
stop_all = true;
cv_job_q_.notify_all();
```

```
for (auto& t : worker_threads_) {
    t.join();
}
}

void ThreadPool::EnqueueJob(std::function<void()> job) {
    if (stop_all) {
        throw std::runtime_error("ThreadPool 사용 중지됨");
    }
    {
        std::lock_guard<std::mutex> lock(m_job_q_);
        jobs_.push(std::move(job));
    }
    cv_job_q_.notify_one();
}

} // namespace ThreadPool

void work(int t, int id) {
    printf("%d start \n", id);
    std::this_thread::sleep_for(std::chrono::seconds(t));
    printf("%d end after %ds\n", id, t);
}

int main() {
    ThreadPool::ThreadPool pool(3);

    for (int i = 0; i < 10; i++) {
        pool.EnqueueJob([i]() { work(i % 3 + 1, i); });
    }
}
```

성공적으로 컴파일 하였다면



와 같이 잘 실행됨을 알 수 있습니다.

쓰레드풀에 작업을 추가하는 것은 아래와 같습니다.

```
pool.EnqueueJob([i]() { work(i % 3 + 1, i); });
```

앞서 쓰레드풀이 받는 함수의 형태가 리턴 타입이 `void`이고 인자를 받지 않는다고 하였습니다. 따라서 `work` 함수를 그대로 전달할 수는 없습니다. 왜냐하면 `int` 타입 인자 두 개를 받기 때문이지요. 하지만 크게 문제될 것은 없습니다. 위와 같이 `void()` 형태의 람다 함수로 감싸서 전달하면 되기 때문이지요.

임의의 함수 받기

안타깝게도 현재 구현한 `ThreadPool`의 경우 부족한 점이 하나 있습니다. 바로 우리가 전달한 함수가 어떠한 값을 리턴할 때 입니다. 물론 그 함수에 포인터로 리턴값을 저장할 변수를 전달하면 되기는 합니다. 하지만, 기존의 `future`처럼 그 값이 설정될 때 까지 기다리는 것은 불가능 합니다.

따라서 더 나은 구조로는 `EnqueueJob` 함수가 임의의 형태의 함수를 받고, 그 함수의 리턴값을 보관하는 `future`를 리턴하는 꼴이면 더 좋을 것 같습니다.

```
// job 을 추가한다.  
template <class F, class... Args>
```

```
std::future<typename std::result_of<F(Args...)>::type> EnqueueJob(F f,
    Args... args);
```

이를 반영한 것이 바로 위 `EnqueueJob` 함수입니다. 엄청 복잡해 보이지만 차근차근 뜯어보면 간단합니다.

```
template <class F, class... Args>
```

위 `class...` 은 가변 길이 템플릿으로 임의의 길이의 인자들을 받을 수 있습니다. 예를 들어서

```
EnqueueJob(func, 1, 2, 3);
```

와 같이 함수를 호출하였을 때 첫 번째 인자인 `func`는 `f`에 들어가게 되고, 나머지 `1, 2, 3`이 `args...` 부분에 들어가게 됩니다. 그렇다면 이 `EnqueueJob` 함수는 무엇을 리턴할까요?

간단히 생각해보면 전달받은 함수 `f`의 리턴값을 가지는 `future`를 리턴해야 할 것입니다. 함수 `F`의 리턴값은 `std::result_of`를 사용하면 알 수 있습니다.

```
typename std::result_of<F(Args...)>::type // f 의 리턴값
```

따라서 `EnqueueJob`의 정의는 그냥

```
// job 을 추가한다.
template <class F, class... Args>
std::future</* f 의 리턴 타입 */> EnqueueJob(F f, Args... args);
```

이라고 생각하시면 됩니다.

그런데 임의의 함수와 원소들을 받을 수 있다고 해서, 이를 컨테이너에 추가할 수 있다는 것은 아닙니다. 어떻게 하면 해당 함수의 실행을 `void()` 꼴의 함수만 저장할 수 있는 컨테이너에 넣을 수 있을까요?

그야 간단합니다. 그냥

```
jobs_.push([f, args...]() { f(args...); });
```

을 한다면 `Worker` 쓰레드 안에서 `f(args...)`를 실행 할 수 있습니다. 그런데 이와 같은 형태는 한 가지 문제점이 있습니다. 바로 `f(args...)`의 리턴값을 얻을 길이 없어진다는 것입니다.

하지만 우리는 이전 강좌를 통해 비동기적으로 실행되는 함수의 리턴값 (더 나아가 예외 까지) 받아내는 법을 알고 있습니다. 바로 `packaged_task`를 이용하는 것입니다!

```
using return_type = typename std::result_of<F(Args...)>::type;
std::packaged_task<return_type()> job(std::bind(f, args...));
```

편의상 `return_type` 라는 `f` 의 리턴타입을 보관하는 타입을 정의하였고, 그 밑에 `f` 의 실행 결과를 저장하는 `packaged_task` 인 `job` 객체를 정의하였습니다.

한 가지 중요한 점은 `packaged_task` 의 생성자는 함수 만을 받기 때문에, 실제 `job` 을 수행하기 위해서는 `job(args...)` 와 같이 호출하거나, 아니면 위처럼 그냥 인자들을 `f` 에 `bind` 시켜주면 됩니다. 우리의 경우 `bind` 를 하는 것으로 선택하였습니다.

```
std::future<return_type> job_result_future = job.get_future();
{
    std::lock_guard<std::mutex> lock(m_job_q_);
    jobs_.push([&job](){ job(); });
}
```

그 후에 `job` 의 실행 결과를 보관하는 `job_result_future` 를 정의하였고, 마지막으로 `jobs_` 에 `job` 을 실행하는 람다 함수를 추가하였습니다. `job` 이 실행된다면, `f` 의 리턴값이 `job_result_future` 에 들어가게 되고, 이는 쓰레드풀 사용자가 접근할 수 있게 됩니다.

```
#include <chrono>
#include <condition_variable>
#include <cstdio>
#include <functional>
#include <future>
#include <mutex>
#include <queue>
#include <thread>
#include <vector>

namespace ThreadPool {
class ThreadPool {
public:
    ThreadPool(size_t num_threads);
    ~ThreadPool();

    // job 을 추가한다.
    template <class F, class... Args>
    std::future<typename std::result_of<F(Args...)>::type> EnqueueJob(
        F f, Args... args);

private:
    // 총 Worker 쓰레드의 개수.
    size_t num_threads_;
    // Worker 쓰레드를 보관하는 벡터.
    std::vector<std::thread> worker_threads_;
```

```
// 할일들을 보관하는 job 큐.
std::queue<std::function<void()>> jobs_;
// 위의 job 큐를 위한 cv 와 m.
std::condition_variable cv_job_q_;
std::mutex m_job_q_;

// 모든 쓰레드 종료
bool stop_all;

// Worker 쓰레드
void WorkerThread();
};

ThreadPool::ThreadPool(size_t num_threads)
    : num_threads_(num_threads), stop_all(false) {
    worker_threads_.reserve(num_threads_);
    for (size_t i = 0; i < num_threads_; ++i) {
        worker_threads_.emplace_back([this]() { this->WorkerThread(); });
    }
}

void ThreadPool::WorkerThread() {
    while (true) {
        std::unique_lock<std::mutex> lock(m_job_q_);
        cv_job_q_.wait(lock, [this]() { return !this->jobs_.empty() || stop_all; });
        if (stop_all && this->jobs_.empty()) {
            return;
        }

        // 맨 앞의 job 을 뺀다.
        std::function<void()> job = std::move(jobs_.front());
        jobs_.pop();
        lock.unlock();

        // 해당 job 을 수행한다 :
        job();
    }
}

ThreadPool::~ThreadPool() {
    stop_all = true;
    cv_job_q_.notify_all();

    for (auto& t : worker_threads_) {
        t.join();
    }
}

template <class F, class... Args>
std::future<typename std::result_of<F(Args...)>::type> ThreadPool::EnqueueJob(
    F f, Args... args)
```

```

    if (stop_all) {
        throw std::runtime_error("ThreadPool 사용 중지됨");
    }

    using return_type = typename std::result_of<F(Args...)>::type;
    std::packaged_task<return_type()> job(std::bind(f, args...));

    std::future<return_type> job_result_future = job.get_future();
    {
        std::lock_guard<std::mutex> lock(m_job_q_);
        jobs_.push([&job] () { job(); });
    }
    cv_job_q_.notify_one();

    return job_result_future;
}

} // namespace ThreadPool

int work(int t, int id) {
    printf("%d start \n", id);
    std::this_thread::sleep_for(std::chrono::seconds(t));
    printf("%d end after %ds\n", id, t);
    return t + id;
}

int main() {
    ThreadPool::ThreadPool pool(3);

    std::vector<std::future<int>> futures;
    for (int i = 0; i < 10; i++) {
        futures.emplace_back(pool.EnqueueJob(work, i % 3 + 1, i));
    }
    for (auto& f : futures) {
        printf("result : %d \n", f.get());
    }
}
}

```

성공적으로 컴파일 후 실행하였다면 아래와 같이 런타임 오류가 발생합니다.

실행 결과

```

0 start
2 start
terminate called after throwing an instance of '4 start
std::future_error'
what(): std::future_error: Broken promise

```

```
[1] 28513 abort (core dumped) ./threadpool
```

보시다시피 **Broken promise** 예외가 던져졌습니다. **Broken promise** 예외는 **promise**에 **set_value**를 하기 전에 이미 **promise**의 **future** 객체가 파괴되었다면 발생하는 예외입니다. 그렇다면 왜 **future** 객체가 파괴되었을까요? 그 이유는 간단합니다.

```
std::packaged_task<return_type()> job(std::bind(f, args...));
```

EnqueueJob 함수에 정의된 **job** 객체는 지역 변수입니다. 즉, **EnqueueJob** 함수가 리턴하면 파괴되는 객체입니다. 따라서 `[&job]() { job(); }` 안에서 **job** 을 접근할 때 이미 그 객체는 파괴되고 없어져있을 것입니다.

이 문제를 해결하는 방법으로 크게 두 가지를 생각해볼 수 있습니다.

1. **packaged_task** 를 따로 컨테이너에 저장해서 보관한다.
2. **shared_ptr** 에 **packaged_task** 를 보관한다.

(1) 번 방식의 경우 더 이상 **packaged_task** 를 사용하지 않을 때에도 컨테이너에 남아있다는 문제가 있습니다. 하지만 (2) 의 경우 **packaged_task** 를 사용하는 것이 없을 때 알아서 **shared_ptr** 가 객체를 소멸시켜주므로 훨씬 관리하기 편합니다. 따라서 후자를 택하도록 하겠습니다. 이를 구현하면 아래와 같습니다.

```
auto job =
    std::make_shared<std::packaged_task<return_type()>>(std::bind(f, args...));
std::future<return_type> job_result_future = job->get_future();
{
    std::lock_guard<std::mutex> lock(m_job_q_);
    jobs_.push([job]() { (*job)(); });
}
```

위와 같이 간단히 **make_shared** 를 통해서 **shared_ptr** 을 생성하였고, 대신에 람다 함수에 **shared_ptr** 의 복사본을 전달해서 람다 함수 안에서도 **packaged_task** 의 **shared_ptr** 하나를 붙들고 있게 되었습니다.

따라서 **job** 을 실행하는 시점에서도 **packaged_task** 객체는 계속 살아있게 됩니다.

```
#include <chrono>
#include <condition_variable>
#include <cstdio>
#include <functional>
#include <future>
#include <mutex>
```

```
#include <queue>
#include <thread>
#include <vector>

namespace ThreadPool {
    class ThreadPool {
        public:
            ThreadPool(size_t num_threads);
            ~ThreadPool();

            // job 을 추가한다.
            template <class F, class... Args>
            std::future<typename std::result_of<F(Args...)>::type> EnqueueJob(
                F f, Args... args);

        private:
            // 총 Worker 쓰레드의 개수.
            size_t num_threads_;
            // Worker 쓰레드를 보관하는 벡터.
            std::vector<std::thread> worker_threads_;
            // 할일들을 보관하는 job 큐.
            std::queue<std::function<void()>> jobs_;
            // 위의 job 큐를 위한 cv 와 m.
            std::condition_variable cv_job_q_;
            std::mutex m_job_q_;

            // 모든 쓰레드 종료
            bool stop_all;

            // Worker 쓰레드
            void WorkerThread();
    };

    ThreadPool::ThreadPool(size_t num_threads)
        : num_threads_(num_threads), stop_all(false) {
        worker_threads_.reserve(num_threads_);
        for (size_t i = 0; i < num_threads_; ++i) {
            worker_threads_.emplace_back([this]() { this->WorkerThread(); });
        }
    }

    void ThreadPool::WorkerThread() {
        while (true) {
            std::unique_lock<std::mutex> lock(m_job_q_);
            cv_job_q_.wait(lock, [this]() { return !this->jobs_.empty() || stop_all; });
            if (stop_all && this->jobs_.empty()) {
                return;
            }

            // 맨 앞의 job 을 뺀다.
            std::function<void()> job = std::move(jobs_.front());
        }
    }
}
```

```
    jobs_.pop();
    lock.unlock();

    // 해당 job 을 수행한다 :
    job();
}
}

ThreadPool::~ThreadPool() {
    stop_all = true;
    cv_job_q_.notify_all();

    for (auto& t : worker_threads_) {
        t.join();
    }
}

template <class F, class... Args>
std::future<typename std::result_of<F(Args...)>::type> ThreadPool::EnqueueJob(
    F f, Args... args) {
    if (stop_all) {
        throw std::runtime_error("ThreadPool 사용 중지됨");
    }

    using return_type = typename std::result_of<F(Args...)>::type;
    auto job =
        std::make_shared<std::packaged_task<return_type()>>(std::bind(f, args...));
    std::future<return_type> job_result_future = job->get_future();
    {
        std::lock_guard<std::mutex> lock(m_job_q_);
        jobs_.push([job]() { (*job)(); });
    }
    cv_job_q_.notify_one();

    return job_result_future;
}

} // namespace ThreadPool

int work(int t, int id) {
    printf("%d start \n", id);
    std::this_thread::sleep_for(std::chrono::seconds(t));
    printf("%d end after %ds\n", id, t);
    return t + id;
}

int main() {
    ThreadPool::ThreadPool pool(3);

    std::vector<std::future<int>> futures;
    for (int i = 0; i < 10; i++) {
```

```
    futures.emplace_back(pool.EnqueueJob(work, i % 3 + 1, i));
}
for (auto& f : futures) {
    printf("result : %d \n", f.get());
}
}
```

성공적으로 컴파일 하였다면



와 같이 잘 나옵니다.

완벽한 전달

자 이제 거의 다 왔습니다. 우리의 `.EnqueueJob` 함수의 경우 다 좋지만 한 가지 문제점이 있는데 바로

```
ThreadPool::EnqueueJob(F f, Args... args);
```

위와 같이 인자들의 복사본을 받는다는 것입니다. 하지만 이는 불필요한 복사를 야기하므로 [완벽한 전달](#) 패턴을 사용하는 것이 좋겠습니다.

이는 크게 어렵지 않습니다. 먼저 `.EnqueueJob` 함수의 인자들을 우측값 레퍼런스로 바꾼 뒤에;

```
template <class F, class... Args>
std::future<typename std::result_of<F(Args...)>::type> EnqueueJob(
    F&& f, Args&&... args);
```

bind 함수에 forward로 인자를 전달해주면 됩니다.

```
auto job = std::make_shared<std::packaged_task<return_type()>>(
    std::bind(std::forward<F>(f), std::forward<Args>(args)...));
```

그렇다면 불필요한 복사 없이 Enqueue 함수에 인자들을 완벽히 전달할 수 있게 됩니다. 따라서 최종 우리의 ThreadPool은 아래와 같습니다.

최종 ThreadPool 구현 버전

```
#include <chrono>
#include <condition_variable>
#include <cstdio>
#include <functional>
#include <future>
#include <mutex>
#include <queue>
#include <thread>
#include <vector>

namespace ThreadPool {
    class ThreadPool {
        public:
            ThreadPool(size_t num_threads);
            ~ThreadPool();

            // job 을 추가한다.
            template <class F, class... Args>
            std::future<typename std::result_of<F(Args...)>::type> EnqueueJob(
                F&& f, Args&&... args);

        private:
            // 총 Worker 쓰레드의 개수.
            size_t num_threads_;
            // Worker 쓰레드를 보관하는 벡터.
            std::vector<std::thread> worker_threads_;
            // 할일들을 보관하는 job 큐.
            std::queue<std::function<void()>> jobs_;
            // 위의 job 큐를 위한 cv 와 m.
            std::condition_variable cv_job_q_;
```

```
    std::mutex m_job_q_;

    // 모든 쓰레드 종료
    bool stop_all;

    // Worker 쓰레드
    void WorkerThread();
};

ThreadPool::ThreadPool(size_t num_threads)
    : num_threads_(num_threads), stop_all(false) {
    worker_threads_.reserve(num_threads_);
    for (size_t i = 0; i < num_threads_; ++i) {
        worker_threads_.emplace_back([this]() { this->WorkerThread(); });
    }
}

void ThreadPool::WorkerThread() {
    while (true) {
        std::unique_lock<std::mutex> lock(m_job_q_);
        cv_job_q_.wait(lock, [this]() { return !this->jobs_.empty() || stop_all; });
        if (stop_all && this->jobs_.empty()) {
            return;
        }

        // 맨 앞의 job 을 뺀다.
        std::function<void()> job = std::move(jobs_.front());
        jobs_.pop();
        lock.unlock();

        // 해당 job 을 수행한다 :)
        job();
    }
}

ThreadPool::~ThreadPool() {
    stop_all = true;
    cv_job_q_.notify_all();

    for (auto& t : worker_threads_) {
        t.join();
    }
}

template <class F, class... Args>
std::future<typename std::result_of<F(Args...)>::type> ThreadPool::EnqueueJob(
    F&& f, Args&&... args) {
    if (stop_all) {
        throw std::runtime_error("ThreadPool 사용 중지됨");
    }
}
```

```
using return_type = typename std::result_of<F(Args...)>::type;
auto job = std::make_shared<std::packaged_task<return_type()>>(
    std::bind(std::forward<F>(f), std::forward<Args>(args)...));
std::future<return_type> job_result_future = job->get_future();
{
    std::lock_guard<std::mutex> lock(m_job_q_);
    jobs_.push([job]() { (*job)(); });
}
cv_job_q_.notify_one();

return job_result_future;
}

} // namespace ThreadPool

// 사용 예시
int work(int t, int id) {
    printf("%d start \n", id);
    std::this_thread::sleep_for(std::chrono::seconds(t));
    printf("%d end after %ds\n", id, t);
    return t + id;
}

int main() {
    ThreadPool::ThreadPool pool(3);

    std::vector<std::future<int>> futures;
    for (int i = 0; i < 10; i++) {
        futures.emplace_back(pool.EnqueueJob(work, i % 3 + 1, i));
    }
    for (auto& f : futures) {
        printf("result : %d \n", f.get());
    }
}
```

성공적으로 컴파일 하였다면



와 같이 잘 실행됩니다 :)

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다.

C++ 강좌도 점점 마무리를 향해 가는것 같습니다. 다음 강좌들에서는 이전 강좌들에서 채 다루지 못했던 C++ 11 에서 새로 추가된 문법 요소와, 더 나아가 몇몇 새로운 라이브러리들을 다룰 예정입니다.

C++ 유니폼 초기화

안녕하세요 여러분! 이번 강좌에서는 C++ 11 에서 추가된 기능인 균일한 초기화(Uniform Initialization)에 대해 살펴보도록 하겠습니다.

아마도 여러분들은 아래와 같은 실수를 한 번쯤 하셨을 것이라 생각합니다.

```
#include <iostream>

class A {
public:
    A() { std::cout << "A 의 생성자 호출!" << std::endl; }

int main() {
    A a(); // ?
}
```

성공적으로 컴파일 하였다면

실행 결과

놀랍게도 아무것도 출력되지 않습니다. 왜일까요?

```
A a(); // ?
```

왜냐하면 사실은 위 코드가 A 의 객체 a 를 만든것이 아니라, A 를 리턴하고, 인자를 받지 않는 함수 a 를 정의한 것이기 때문입니다. 왜냐하면 C++ 의 컴파일러는 함수의 정의처럼 보이는 것들은 모두 함수의 정의로 해석 하기 때문입니다.

심지어 아래와 같은 코드는 더 헷갈립니다.

```
#include <iostream>

class A {
public:
    A() { std::cout << "A 의 생성자 호출!" << std::endl; }

};

class B {
public:
    B(A a) { std::cout << "B 의 생성자 호출!" << std::endl; }

};

int main() {
    B b(A()); // 뭐가 출력될까요?
}
```

성공적으로 컴파일 하였다면

실행 결과

와 같이 아무 것도 출력되지 않습니다.

사실 위 코드를 보면 마치 b라는 클래스 B의 객체를 생성하는 것 같아 보이지만, 사실은 인자로 A를 리턴하고 인자가 없는 함수를 받으며, 리턴 타입이 B인 함수 b를 정의한 것입니다.

상당히 골치 아픈 일입니다. 이러한 문제가 발생하는 것은 () 가 함수의 인자들을 정의하는데도 사용되고, 그냥 일반적인 객체의 생성자를 호출하는데에도 사용되기 때문입니다.

따라서 C++ 11에서는 이러한 문제를 해결하기 위해 **균일한 초기화(Uniform Initialization)**라는 것을 도입하였습니다.

균일한 초기화 (Uniform Initialization)

균일한 초기화 문법을 사용하기 위해서는 생성자를 호출하기 위해 () 를 사용하는 대신에 {} 를 사용하면 끝입니다.

```
#include <iostream>

class A {
public:
    A() { std::cout << "A 의 생성자 호출!" << std::endl; }

};
```

```
int main() {
    A a{}; // 균일한 초기화!
}
```

성공적으로 컴파일 하였다면

실행 결과

A 의 생성자 호출!

위와 같이 제대로 생성자가 호출되었음을 알 수 있습니다.

중괄호를 이용해서 생성자를 호출하는 문법은 동일합니다. 그냥 기존에 () 자리를 {}로 바꿔주기만 하면 됩니다. 하지만, ()를 이용한 생성과 {}를 이용한 생성의 경우 한 가지 큰 차이가 있는데 바로 일부 암시적 타입 변환들을 불허하고 있다는 점입니다.

예를 들어서 아래 코드를 살펴봅시다.

```
#include <iostream>

class A {
    public:
        A(int x) { std::cout << "A 의 생성자 호출!" << std::endl; }
};

int main() {
    A a(3.5); // Narrow-conversion 가능
    A b{3.5}; // Narrow-conversion 불가
}
```

컴파일 하였다면 아래와 같은 오류가 발생합니다.

컴파일 오류

```
test.cc: In function ‘int main()’:
test.cc:10:10: error: narrowing conversion of ‘3.5e+0’ from
→ ‘double’ to ‘int’ inside { } [-Wnarrowing]
    A b{3.5}; // Narrow-conversion 불가
          ^
A 의 생성자 호출!
```

보시다시피

```
A a(3.5); // Narrow-conversion 가능
```

위 코드는 성공적으로 컴파일 되었고 x에는 3.5의 정수 캐스팅 버전이 3이 전달됩니다. (한 번 아래 b 생성을 지원하고 실행해보세요.)

반면에

```
A b{3.5}; // Narrow-conversion 불가
```

의 경우 위와 같이 double인 3.5를 int로 변환할 수 없다는 오류가 발생하였습니다.

그 이유는 중괄호를 이용해서 생성자를 호출하는 경우 아래와 같은 암시적 타입 변환들이 불가능해집니다. 이들은 전부 데이터 손실이 있는(Narrowing) 변환입니다.

- 부동 소수점 타입에서 정수 타입으로의 변환 (우리의 예시지요)
- long double에서 double 혹은 float으로의 변환, double에서 float으로의 변환
- 정수 타입에서 부동 소수점 타입으로의 변환

등등이 있습니다. 자세한 예시들은 [여기](#)에서 확인할 수 있습니다.

따라서 {}를 사용하게 된다면, 위와 같이 원하지 않는 타입 캐스팅을 방지해서 미연에 오류를 잡아낼 수 있습니다.

{ }를 이용한 생성의 또 다른 쓰임새로 함수 리턴 시에 굳이 생성하는 객체의 타입을 다시 명시 하지 않아도 됩니다.

```
#include <iostream>

class A {
public:
    A(int x, double y) { std::cout << "A 생성자 호출" << std::endl; }
};

A func() {
    return {1, 2.3}; // A(1, 2.3) 과 동일
}

int main() { func(); }
```

성공적으로 컴파일 하였다면

실행 결과

A 생성자 호출

와 같이 잘 나옵니다. {} 를 이용해서 생성하지 않았더라면 A(1, 2, 3) 과 같이 클래스를 명시해 줘야만 했지만, {} 를 이용할 경우 컴파일러가 알아서 함수의 리턴타입을 보고 추론해줍니다.

{ } 의 쓰임새는 이 뿐만이 아닙니다. 아래를 보시지요.

초기화자 리스트 (Initializer list)

배열을 정의할 때 우리는 다음과 같이 작성하였습니다.

```
int arr[] = {1, 2, 3, 4};
```

그렇다면 중괄호를 이용해서 마찬가지 효과를 낼 수 없을까요? 예를 들면

```
vector<int> v = {1, 2, 3, 4};
```

와 같이 말이지요. 근데, 놀랍게도 C++ 11 에서부터 이와 같은 문법을 사용할 수 있게 되었습니다.

```
#include <iostream>

class A {
public:
    A(std::initializer_list<int> l) {
        for (auto itr = l.begin(); itr != l.end(); ++itr) {
            std::cout << *itr << std::endl;
        }
    }
};

int main() { A a = {1, 2, 3, 4, 5}; }
```

성공적으로 컴파일 하였다면

실행 결과

```
1
2
3
```

```
4
5
```

와 같이 나옵니다.

`initializer_list` 는 우리가 `{}` 를 이용해서 생성자를 호출할 때, 클래스의 생성자들 중에 `initializer_list` 를 인자로 받는 생성자가 있다면 전달됩니다.

주의 사항

(`)` 를 사용해서 생성자를 호출한다면 `initializer_list` 가 생성되지 않습니다.

`initializer_list` 를 이용하면 컨테이너들을 간단하게 정의할 수 있습니다. 예를 들어서

```
#include <iostream>
#include <map>
#include <string>
#include <vector>

template <typename T>
void print_vec(const std::vector<T>& vec) {
    std::cout << "[";
    for (const auto& e : vec) {
        std::cout << e << " ";
    }
    std::cout << "]" << std::endl;
}

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    for (const auto& kv : m) {
        std::cout << kv.first << " : " << kv.second << std::endl;
    }
}

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    print_vec(v);

    std::cout << "-----" << std::endl;
    std::map<std::string, int> m = {
        {"abc", 1}, {"hi", 3}, {"hello", 5}, {"c++, 2}, {"java", 6}};
    print_map(m);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
[1 2 3 4 5 ]
-----
abc : 1
c++ : 2
hello : 5
hi : 3
java : 6
```

와 같이 나옵니다.

```
std::vector<int> v = {1, 2, 3, 4, 5};
```

`vector`의 경우 생각했던대로, `vector`의 원소들을 그냥 나열해주면 됩니다. 마치 이전에 배열을 정의할 때처럼 말이지요.

```
std::map<std::string, int> m = {
    {"abc", 1}, {"hi", 3}, {"hello", 5}, {"c++", 2}, {"java", 6}};
```

`map`의 경우도 비슷합니다. `map`의 경우 `vector`와는 다르게 `pair<Key, Value>` 원소들을 초기화자 리스트의 원소들로 받습니다. `pair`는 C++ STL에서 지원하는 간단한 클래스로 그냥 두 개의 원소를 보관하는 객체라고 보시면 됩니다. `map`에는 `pair`의 첫 번째 원소로 키를, 두 번째 원소로 값을 전달해주면 됩니다.

initializer_list 사용 시 주의할 점

생성자들 중에서 `initializer_list`를 받는 생성자가 있다면 한 가지 주의해야 할 점이 있습니다. 만일 {} 를 이용해서 객체를 생성할 경우 생성자 오버로딩 시에 해당 함수가 최우선으로 고려된다는 점입니다.

예를 들어서 `vector`의 경우 아래와 같은 형태의 생성자가 존재합니다.

```
vector(size_type count);
```

이 생성자는 `count` 개수 만큼의 원소 자리를 미리 생성해놓습니다. 그렇다면

```
vector v{10};
```

은 해당 생성자를 호출할까요? 아닙니다. 그냥 원소 1 개 짜리 `initializer_list` 라고 생각해서 10 을 보관하고 있는 벡터를 생성하게 됩니다.

따라서, 이러한 불상사를 막기 위해서는 {} 로 생성하기 보다는 () 를 이용해서

```
vector v(10);
```

과 같이 v 를 생성한다면 우리가 원하는 생성자를 호출할 수 있게 됩니다.

`initializer_list` 를 받는 생성자가 최우선적으로 고려된다는 말은, 컴파일러가 최선을 다해서 해당 생성자와 매칭시키려고 노력한다는 의미입니다. 예를 들어서 아래와 같은 코드를 살펴봅시다.

```
#include <initializer_list>
#include <iostream>

class A {
public:
    A(int x, double y) { std::cout << "일반 생성자! " << std::endl; }

    A(std::initializer_list<int> lst) {
        std::cout << "초기화자 사용 생성자! " << std::endl;
    }
};

int main() {
    A a(3, 1.5); // Good
    A b{3, 1.5}; // Bad!
}
```

컴파일 하였다면 아래와 같은 오류가 발생합니다.

컴파일 오류

```
test.cc: In function ‘int main()’:
test.cc:15:13: error: narrowing conversion of ‘1.5e+0’ from
→ ‘double’ to ‘int’ inside { } [-Wnarrowing]
    A b{3, 1.5}; // Bad!
          ^
```

일단 보시다시피

```
A a(3, 1.5); // Good
```

이 문장은 아무런 문제가 없습니다. () 를 이용해서 생성자를 호출하였기 때문에, A 의 첫 번째 생성자인 일반 생성자가 호출됩니다.

하지만 그 다음 문장을 살펴봅시다.

```
A b{3, 1.5}; // Bad!
```

앞서 C++ 컴파일러는 {} 를 이용해서 생성자를 호출하였을 경우 `initializer_list` 를 받는 생성자를 최우선으로 고려한다고 하였습니다. 따라서, 컴파일러는 `initializer_list` 를 이용하도록 최대한 노력하려고 하는데, 1.5 는 `int` 가 아니지만, `double` 에서 `int` 로 암시적 변환을 할 수 있으므로 이를 택하게 됩니다.

그런데 문제는 앞서 {} 는 데이터 손실이 있는 변환을 할 수 없다고 하였습니다. 그런데 `double` 에서 `int` 로의 타입 변환은 데이터 손실이 있는 변환이므로, 오류가 발생하게 됩니다. 사실 `A(int x, double y)` 이 생성자가 좀 더 나은 매칭이지만, C++ 컴파일러는 `initializer_list` 를 이용한 생성자를 최대한 고려하려고 합니다. 이러한 문제가 발생하지 않는 경우는 `initializer_list` 의 원소 타입으로 타입 변환 자체가 불가능한 경우여야만 합니다.

```
#include <initializer_list>
#include <iostream>
#include <string>

class A {
public:
    A(int x, double y) { std::cout << "일반 생성자! " << std::endl; }

    A(std::initializer_list<std::string> lst) {
        std::cout << "초기화자 사용 생성자! " << std::endl;
    }
};

int main() {
    A a(3, 1.5);           // 일반
    A b{3, 1.5};           // 일반
    A c{"abc", "def"};     // 초기화자
}
```

성공적으로 컴파일 하였다면

실행 결과

```
일반 생성자!
일반 생성자!
초기화자 사용 생성자!
```

와 같이 잘 나옵니다. 위 경우 `int` 나 `double` 이 `string` 으로 변환될 수 없기 때문에 `initializer_list` 를 받는 생성자는 아예 고려 대상에서 제외됩니다.

initializer_list 와 auto

만일 {} 를 이용해서 생성할 때 타입으로 `auto` 를 지정한다면 `initializer_list` 객체가 생성 됩니다. 예를 들어서

```
auto list = {1, 2, 3};
```

을 하게 되면 `list` 는 `initializer_list<int>` 가 되겠지요.

그렇다면 아래는 어떨까요? 참고로 이 예제는 여기에서 가져왔습니다.

```
auto a = {1};      // std::initializer_list<int>
auto b{1};         // std::initializer_list<int>
auto c = {1, 2};   // std::initializer_list<int>
auto d{1, 2};       // std::initializer_list<int>
```

상식적으로 적어도 `b` 는 `int` 로 추론되어야 할 것 같지만, C++ 11 에서는 위 `a, b, c, d` 모두 `std::initializer_list<int>` 로 정의됩니다.

하지만 이는 꽤 비상식적이기 때문에 C++ 17 부터 아래와 같이 두 가지 형태로 구분해서 `auto` 타입이 추론됩니다.

- `auto x = {arg1, arg2...}` 형태의 경우 `arg1, arg2 ...` 들이 모두 같은 타입이라면 `x` 는 `std::initializer_list<T>` 로 추론됩니다.
- `auto x {arg1, arg2, ...}` 형태의 경우 만일 인자가 단 1 개라면 인자의 타입으로 추론되고, 여러 개일 경우 오류를 발생시킵니다.

따라서 C++ 17 부터는 아래와 같습니다.

```
auto a = {1};      // 첫 번째 형태이므로 std::initializer_list<int>
auto b{1};         // 두 번째 형태 이므로 그냥 int
auto c = {1, 2};   // 첫 번째 형태이므로 std::initializer_list<int>
auto d{1, 2};       // 두 번째 형태 인데 인자가 2 개 이상이므로 컴파일 오류
```

좀 더 직관적으로 바뀐 것을 알 수 있습니다.

유니폼 초기화와 `auto` 를 같이 사용 할 때 또 한 가지 주의할 점은, 문자열을 다룰 때

```
auto list = {"a", "b", "cc"};
```

를 하게 된다면 `list` 는 `initializer_list<std::string>` 이 아닌 `initializer_list<const char*>` 이 된다는 점입니다. 물론 이 문제는 C++ 14 에서 추가된 리터럴 연산자를 통해 해결할 수 있습니다.

```
using namespace std::literals; // 문자열 리터럴 연산자를 사용하기 위해
                                // 추가해줘야함.
auto list = {"a"s, "b"s, "c"s};
```

와 같이 하면, `initializer_list<std::string>` 으로 추론됩니다.

자 그럼 이것으로 이번 강좌를 마무리 하도록 하겠습니다. 다음 강좌에서는 C++ 11 에서 추가된 `constexpr` 와 가변 길이 템플릿에 대해 다루어보도록 하겠습니다.

뭘 배웠지?

- 유니폼 초기화 (`{}` 를 이용한 생성자 호출) 를 통해서 인자 없는 생성자가 함수의 정의로 오해되는 일을 막을 수 있으며 `initializer_list` 를 만들어 전달할 수 있습니다.
- `initializer_list` 를 통해서 객체를 간단하게 생성할 수 있습니다.

컴파일 타임 상수 constexpr

안녕하세요 여러분! 이번 강좌에서는 C++ 11 에서 새롭게 도입된 `constexpr` 키워드에 대해 알아보도록 하겠습니다. `constexpr` 키워드는 객체나 함수 앞에 붙일 수 있는 키워드로, 해당 객체나 함수의 리턴값을 컴파일 타임에 값을 알 수 있다 라는 의미를 전달하게 됩니다.

컴파일러가 컴파일 타임에 어떠한 식의 값을 결정할 수 있다면 해당 식을 상수식 (Constant expression) 이라고 표현합니다. 그리고 이러한 상수식들 중에서 값이 정수인 것을 정수 상수식 (Integral constant expression) 이라고 하게 되는데, 정수 상수식들은 매우 쓰임새가 많습니다.

예를 들어서

```
int arr[size];
```

위 배열 선언식이 컴파일 되기 위해서는 `size` 가 정수 상수식이여야 하고,

```
template <int N>
struct A {
    int operator()() { return N; }
};

A<number> a;
```

템플릿 타입 인자의 경우도 마찬가지로 `number` 가 정수 상수식이여야만 합니다. 그 외에도,

```
enum A { a = number, b, c };
```

`enum` 에서 값을 지정해줄 때에 오는 `number` 역시 정수 상수식이여야만 합니다. 이처럼 C++ 언어 상 정수 상수식이 등장하는 곳은 매우 많습니다.

constexpr

`constexpr` 은 앞서 말한 대로, 어떠한 식이 상수식 이라고 명시해주는 키워드입니다. 만일, 객체의 정의에 `constexpr` 이 오게 된다면, 해당 객체는 어떠한 상수식에도 사용될 수 있습니다. 아래 예시를 보실까요.

```
#include <iostream>

template <int N>
struct A {
    int operator()() { return N; }
};

int main() {
    constexpr int size = 3;
    int arr[size]; // Good!

    constexpr int N = 10;
    A<N> a; // Good!
    std::cout << a() << std::endl;

    constexpr int number = 3;
    enum B { x = number, y, z }; // Good!
    std::cout << B::x << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
10
3
```

와 같이 잘 나옵니다.

`constexpr` 은 언뜻 보기에도 `const` 와 큰 차이가 없어 보입니다. `constexpr` 로 정의된 변수들도 마찬가지로 상수이므로 수정할 수 없기 때문이죠. 하지만 둘은 큰 차이가 있습니다.

constexpr vs const

`const` 로 정의된 상수들은 굳이 컴파일 타임에 그 값을 알 필요가 없습니다. 예를 들어서;

```
int a;
```

```
// Do something...
const int b = a;
```

위 코드를 볼 때 b의 값을 컴파일 타임에 알 수는 없지만, b의 값을 지정해주면 바꿀 수 없다는 점은 확실합니다.

반면에

```
int a;
// Do something...
constexpr int b = a; // ??
```

반면에 **constexpr** 변수의 경우 반드시 오른쪽에 다른 상수식이 와야 합니다. 하지만 컴파일러 입장에서 컴파일 타임 시에 a가 뭐가 올지 알 수 없습니다. 따라서 위 코드는 컴파일 오류가 됩니다. 정리하자면, **constexpr** 은 항상 **const** 이지만, **const** 는 **constexpr** 이 아닙니다!

여담으로 **const** 객체가 만일 상수식으로 초기화 되었다 하더라도 컴파일러에 따라 이를 런타임에 초기화 할지, 컴파일에 초기화할지 다를 수 있습니다. 예컨대

```
const int i = 3;
```

위의 경우 i는 컴파일 타임에 초기화될 수도, 런타임에 초기화될 수도 있습니다. 따라서 컴파일 타임에 상수를 확실히 사용하고 싶다면 **constexpr** 키워드를 꼭 사용해야 합니다.

constexpr 함수

앞서 **constexpr**로 객체를 선언한다면 해당 객체는 컴파일 타임 상수로 정의된다고 하였습니다. 그렇다면 컴파일 타임 상수인 객체들을 만들어내는 함수를 정의할 수는 없을까요?

constexpr 키워드가 등장하기 이전에는 컴파일 타임 상수인 객체를 만드는 함수를 작성하는 것이 불가능 하였습니다. 예를 들어서;

```
#include <iostream>

int factorial(int N) {
    int total = 1;
    for (int i = 1; i <= N; i++) {
        total *= i;
    }
```

```

    return total;
}

template <int N>
struct A {
    int operator()() { return N; }
};

int main() { A<factorial(5)> a; }

```

컴파일 하였다면

컴파일 오류

```

test2.cc: In function ‘int main()’:
test2.cc:17:14: error: call to non-constexpr function ‘int
→   factorial(int)’
A<factorial(5)> a;
~~~~~^~~

test2.cc:17:14: error: call to non-constexpr function ‘int
→   factorial(int)’
test2.cc:17:17: note: in template argument for type ‘int’
A<factorial(5)> a;
^

```

와 같은 오류가 발생하게 됩니다. 왜냐하면 `factorial(5)` 는 컴파일 타임 상수가 아니기 때문이지요. 물론 우리는 똑똑하기 때문에 `factorial(5)` 를 컴파일 타임에 계산해서 그냥 `A<120>a;` 로 컴파일 해도 된다는 것을 알고 있지만, 컴파일러는 그렇지 않습니다.

따라서 이와 같은 문제를 해결하기 위해 기존에는 난해한 템플릿 메타프로그래밍을 사용해야했습니다. 예를 들어서 위 `factorial` 함수를 TMP 방식으로 접근한 코드를 살펴봅시다.

```

#include <iostream>

template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static const int value = 1;
};

```

```
template <int N>
struct A {
    int operator()() { return N; }
};

int main() {
    // 컴파일 타임에 값이 결정되므로 템플릿 인자로 사용 가능!
    A<Factorial<10>::value> a;

    std::cout << a() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

3628800

와 같이 잘 나옵니다.

```
template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static const int value = 1;
};
```

아무래도 이전에 템플릿 메타프로그래밍 강좌를 보신 분들은 위 코드가 무엇을 하는지 단박에 이해하실 수 있을 것이라 생각합니다. N 부터 1 까지의 곱을 TMP 의 형태로 수행하게 됩니다.

```
// 컴파일 타임에 값이 결정되므로 템플릿 인자로 사용 가능!
A<Factorial<10>::value> a;
```

그렇다면 우리의 Factorial 클래스를 통해 계산한 $10!$ 의 값을 사용해서 배열을 정의할 수도 있습니다. 이 경우 위 코드는 $A<3628800>$ a 와 동일합니다.

위와 같이 배열의 크기를 정의할 수 있는 이유는 $Factorial<10>::value$ 의 값을 컴파일러가 컴파일 타임에 알아낼 수 있기 때문입니다.

하지만, 위 Factorial 클래스처럼 간단한 경우를 빼면 사실 TMP 를 이용해서 구현된 코드는 딱히 이해하기 쉽지 않습니다. 왜냐하면 템플릿의 특성상 조건문들은 대개 템플릿 특수화를 통해 구현되고, 반복문들은 재귀 호출의 형태로 구현되어야 하기 때문에 복잡하기 때문입니다.

하지만 함수의 리턴 타입에 `constexpr` 을 추가한다면 조건이 맞을 때, 해당 함수의 리턴값을 컴파일 타임 상수로 만들어버릴 수 있습니다.

그럼 아래 코드를 살펴보겠습니다.

```
#include <iostream>

constexpr int Factorial(int n) {
    int total = 1;
    for (int i = 1; i <= n; i++) {
        total *= i;
    }
    return total;
}

template <int N>
struct A {
    int operator()() { return N; }
};

int main() {
    A<Factorial(10)> a;

    std::cout << a() << std::endl;
}
```

성공적으로 컴파일 하였다면

```
3628800
```

와 같이 잘 나옵니다! 놀랍게도 `Factorial(10)` 이 컴파일 타임에 계산되어서 클래스 A 의 템플릿 인자로 들어가게 되었습니다.

```
constexpr int Factorial(int n) {
    int total = 1;
    for (int i = 1; i <= n; i++) {
        total *= i;
    }
    return total;
}
```

위 `constexpr` 함수를 살펴보면 그냥 일반 함수와는 다를 바가 없습니다. 사실 C++ 11 에 `constexpr` 이 처음 도입되었을 때에는 `constexpr` 함수에는 여러 제약 조건이 많았습니다. 예를 들어서 함수 내부에서 변수들을 정의할 수 없고, `return` 문은 딱 하나만 있어야만 했습니다.

하지만 C++ 14 부터 위와 같은 제약 조건들이 완화되어서 아래와 제약 조건들 빼고는 모두 `constexpr` 함수 내부에서 수행할 수 있습니다.

- `goto` 문 사용
- 예외 처리 (`try` 문; C++ 20 부터 가능하게 바뀌었습니다.)
- 리터럴 타입이 아닌 변수의 정의
- 초기화 되지 않는 변수의 정의
- 실행 중간에 `constexpr` 이 아닌 함수를 호출하게 됨

따라서 위와 같은 작업들을 하지 않는 이상 `constexpr` 키워드를 함수에 붙일 수 있게 됩니다. 만일 조건을 만족하지 않는 작업을 함수 내에서 하게 된다면 컴파일 타임 오류가 발생하게 됩니다. 예를 들어서

```
int not_constexpr(int x) { return x++; }
constexpr int Factorial(int n) {
    int total = 1;
    for (int i = 1; i <= n; i++) {
        total *= i;
    }

    not_constexpr(total);
    return total;
}
```

위 경우 중간에 `constexpr` 함수가 아닌 함수를 호출하게 되므로

컴파일 오류

```
test2.cc: In function ‘constexpr int Factorial(int)’:
test2.cc:28:16: error: call to non-constexpr function ‘int
→ not_constexpr(int)’
not_constexpr(total);
~~~~~^~~~~~
```

위와 같은 오류가 발생하게 됩니다.

성공적으로 `constexpr` 함수를 정의하였다면 이를 이용해서 `constexpr` 상수들을 생성할 수 있습니다. `constexpr` 함수에 인자로 컴파일 타임 상수들을 전달하면, 그 반환값 역시 컴파일 타임 상수가 됩니다. 우리의 사용 예시도 마찬가지로

```
A<Factorial(10)> a;
```

위 처럼 컴파일 타임 상수인 10 을 전달하였기 때문에 `Factorial(10)` 의 반환값은 컴파일 타임 상수가 되어서 위처럼 템플릿 인자로 전달 가능하게 됩니다. 당연하게도 `constexpr` 으로 정의된 상수들 역시 컴파일 타임 상수 이므로;

```
constexpr int ten = 10;
A<Factorial(ten)> a;
```

위 역시 마찬가지로 동작합니다. 그렇다면 `constexpr` 함수는 컴파일 타임 상수들만 인자로 받을 수 있는 것일까요? 아닙니다! `constexpr` 함수에 인자로 컴파일 타임 상수가 아닌 값을 전달하였다면 그냥 일반 함수처럼 동작하게 됩니다.

```
#include <iostream>

constexpr int Factorial(int n) {
    int total = 1;
    for (int i = 1; i <= n; i++) {
        total *= i;
    }
    return total;
}

int main() {
    int num;
    std::cin >> num;
    std::cout << Factorial(num) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
5
120
```

와 같이 잘 실행됩니다. 위 경우 `Factorial` 에 컴파일 타임 상수가 아닌 일반 값을 전달하였지만, 컴파일 타입이 아닌 런타임에 `Factorial` 이 실행되어서 잘 작동합니다.

따라서 `constexpr` 을 함수에 붙일 수 있다면 붙여주는 것이 좋습니다. 왜냐하면 `constexpr` 처럼 동작하지 못한다면 그냥 일반 함수처럼 동작할 테이고, 컴파일 타임 상수를 생성할 수 있는 상황이라면 간단히 이용할 수 있기 때문이지요.

리터럴 타입?

앞서 `constexpr` 함수 내부에서 불가능한 작업으로 리터럴(Literal) 타입이 아닌 변수의 정의라고 이야기 하였습니다. 리터럴 타입은 쉽게 생각하면 컴파일러가 컴파일 타임에 정의할 수 있는 타입이라고 생각하시면 됩니다. C++ 에서 정의하는 바로는;

- `void` 형
- 스칼라 타입 (`char`, `int`, `bool`, `long`, `float`, `double`) 등등
- 레퍼런스 타입
- 리터럴 타입의 배열
- 혹은 아래 조건들을 만족하는 타입
 - 디폴트 소멸자를 가지고
 - 다음 중 하나를 만족하는 타입
 - * 람다 함수
 - * Aggregate 타입 (사용자 정의 생성자, 소멸자가 없으며 모든 데이터 멤버들이 `public`)
쉽게 말해 `pair` 같은 애들을 이야기함
 - * `constexpr` 생성자를 가지며 복사 및 이동 생성자가 없음

들을 리터럴 타입이라 의미하며 해당 객체들만이 `constexpr` 로 선언되던지 `constexpr` 함수 내부에서 사용될 수 있습니다. 이전에는 리터럴 타입으로 정의되어 있는 것들이 매우 한정적이였는데 (대부분 스칼라 타입), C++ 14 부터 `constexpr` 생성자를 지원함으로써 사용자들이 리터럴 타입들을 직접 만들 수 있게 되었습니다.

그렇다면 `constexpr` 생성자를 어떻게 사용하는지 살펴보겠습니다.

`constexpr` 생성자

`constexpr`로 생성자의 경우 일반적인 `constexpr` 함수에서 적용되는 제약조건들이 모두 적용됩니다. 또한 `constexpr` 생성자의 인자들은 반드시 리터럴 타입이여야만 하고, 해당 클래스는 다른 클래스를 가상 상속 받을 수 없습니다.

예를 들어서 아래와 같은 클래스를 생각해봅시다.

```
class Vector {  
public:
```

```
constexpr Vector(int x, int y) : x_(x), y_(y) {}

constexpr int x() const { return x_; }
constexpr int y() const { return y_; }

private:
    int x_;
    int y_;
};
```

위 `Vector` 클래스는 벡터를 나타내는 클래스입니다 (`std::vector` 가 아닙니다!). `Vector` 의 생성자는 리터럴인 `int` 두 개를 인자로 받고 있습니다. 따라서 이는 적합한 `constexpr` 생성자가 되겠지요.

```
constexpr int x() const { return x_; }
constexpr int y() const { return y_; }
```

마찬가지로 두 멤버 변수를 접근하는 함수 역시 `constexpr` 로 정의해주었습니다. 따라서 `x()` 와 `y()` 역시 `constexpr` 함수 내부에서 사용할 수 있게 됩니다.

그렇다면 실제 컴파일 시에 어떻게 작동하는지 살펴봅시다.

```
#include <iostream>

class Vector {
public:
    constexpr Vector(int x, int y) : x_(x), y_(y) {}

    constexpr int x() const { return x_; }
    constexpr int y() const { return y_; }

private:
    int x_;
    int y_;
};

constexpr Vector AddVec(const Vector& v1, const Vector& v2) {
    return {v1.x() + v2.x(), v1.y() + v2.y()};
}

template <int N>
struct A {
    int operator()() { return N; }
};

int main() {
    constexpr Vector v1{1, 2};
```

```
constexpr Vector v2{2, 3};

// constexpr 객체의 constexpr 멤버 함수는 역시 constexpr!
A<v1.x()> a;
std::cout << a() << std::endl;

// AddVec 역시 constexpr 을 리턴한다.
A<AddVec(v1, v2).x()> b;
std::cout << b() << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1
3
```

와 같이 잘 나옵니다.

```
constexpr Vector v1{1, 2};
constexpr Vector v2{2, 3};
```

먼저 위처럼 우리가 만든 클래스인 `Vector` 를 `constexpr` 로 선언할 수 있었습니다. 왜냐하면 `constexpr` 로 생성자를 만들었기 때문이지요.

```
A<v1.x()> a;
```

그리고 `v1` 의 `constexpr` 멤버 함수인 `x` 를 호출하였는데, `x` 역시 `constexpr` 함수이므로 위 코드는 결국 `A<1> a` 와 다름이 없게 됩니다. 만일 `v1` 이나 `x` 가 하나라도 `constexpr` 이 아니라면 위 코드는 컴파일 되지 않습니다. `constexpr` 객체의 `constexpr` 멤버 함수만이 `constexpr` 을 줍니다!

```
// AddVec 역시 constexpr 을 리턴한다.
A<AddVec(v1, v2).x()> b;
```

그렇다면 우리의 `AddVec` 함수는 어떨까요? 마찬가지로 `v1` 과 `v2` 를 인자로 받아서 `constexpr` 객체를 리턴하게 됩니다. 이것이 가능한 이유는 `AddVec` 이 `constexpr` 함수이고, `Vector` 가 리터럴 타입이여서 그렇겠지요.

if constexpr

만약에 타입에 따라 형태가 달라지는 함수를 짜고 싶다면 어떻게 하시나요? 예를 들어서 `get_value`라는 함수가 있는데, 이 함수는 인자가 포인터 타입이면 `*` 을 한 것을 리턴하고 아니면 그냥 원래의 인자를 리턴하는 함수입니다.

템플릿 타입 유추를 이용하면 해당 함수는 다음과 같이 작성할 수 있습니다.

```
#include <iostream>

template <typename T>
void show_value(T t) {
    std::cout << "포인터가 아니다 : " << t << std::endl;
}

template <typename T>
void show_value(T* t) {
    std::cout << "포인터이다 : " << *t << std::endl;
}

int main() {
    int x = 3;
    show_value(x);

    int* p = &x;
    show_value(p);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
포인터가 아니다 : 3
포인터이다 : 3
```

와 같이 잘 나옵니다. 아주 좋습니다. 하지만 문제는 1) `show_value` 함수가 정확히 어떠한 형태의 `T`를 요구하는지 한 눈에 파악하기 힘들고 2) 같은 함수를 두 번 써야 한다는 점입니다.

C++ 표준 라이브러리의 `<type_traits>`에서는 여러가지 템플릿 함수들을 제공하는데, 이들 중 해당 타입이 포인터인지 아닌지 확인하는 함수도 있습니다. 이를 사용해서 한 번 구성해보겠습니다.

```
#include <iostream>
#include <type_traits>

template <typename T>
```

```

void show_value(T t) {
    if (std::is_pointer<T>::value) {
        std::cout << "포인터이다 : " << *t << std::endl;
    } else {
        std::cout << "포인터가 아니다 : " << t << std::endl;
    }
}

int main() {
    int x = 3;
    show_value(x);

    int* p = &x;
    show_value(p);
}

```

컴파일 하였다면

컴파일 오류

```

test2.cc: In instantiation of ‘void show_value(T) [with T =
→  int]’:
test2.cc:15:15:   required from here
test2.cc:6:43: error: invalid type argument of unary ‘*’ (have
→  ‘int’)
    std::cout << "포인터이다 : " << *t << std::endl;
                                         ^
~

```

와 같이 오류가 발생합니다. 일단 `is_pointer` 부터 살펴봅시다.

```

if (std::is_pointer<T>::value) {

```

`std::is_pointer` 는 전달한 인자 `T` 가 포인터라면 `value` 가 `true` 가 되고, 포인터가 아니면 `false` 가 되는 템플릿 메타 함수입니다. 따라서 만일 `T` 가 포인터라면 `*t` 를 출력하고 아니면 `t` 를 출력하겠지요.

하지만 문제는 템플릿이 인스턴스화 되면서 생성되는 코드에 컴파일이 불가능한 부분이 발생된다는 것입니다. `show_value(x)` 를 하게 된다면 생성되는 코드는

```

void show_value(int t) {
    if (std::is_pointer<int>::value) {
        std::cout << "포인터이다 : " << *t << std::endl;
    } else {
        std::cout << "포인터가 아니다 : " << t << std::endl;
    }
}

```

```
    }
}
```

가 되므로 `int` 타입인 `t`에 `*` 연산자가 붙게 됩니다. 따라서 위 `if` 문은 절대로 실행되지 않음에도 불구하고 컴파일 되지 않기 때문에 오류가 발생한 것이지요.

하지만 이 문제는 `if constexpr` 을 도입하면 깔끔히 해결됩니다.

```
#include <iostream>
#include <type_traits>

template <typename T>
void show_value(T t) {
    if constexpr (std::is_pointer<T>::value) {
        std::cout << "포인터이다 : " << *t << std::endl;
    } else {
        std::cout << "포인터가 아니다 : " << t << std::endl;
    }
}

int main() {
    int x = 3;
    show_value(x);

    int* p = &x;
    show_value(p);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
포인터가 아니다 : 3
포인터이다 : 3
```

와 같이 잘 나옵니다. `if constexpr` 은 조건이 반드시 `bool` 로 타입 변환될 수 있어야 하는 컴파일 타임 상수식이어야만 합니다. 그 대신, `if constexpr` 이 참이라면 `else` 에 해당하는 문장은 컴파일 되지 않고 (완전히 무시) 마찬가지로 `if constexpr` 이 거짓이라면 `else` 에 해당하는 부분만 컴파일 됩니다.

따라서 위 경우 `std::is_pointer<int>::value` 는 거짓 이므로 아예 `*t` 자체가 컴파일 되지 않습니다. 덕분에 컴파일 오류는 발생하지 않습니다.

참고로 `std::is_pointer<T>::value` 가 거추장스럽다면 그냥 `std::is_pointer_v<T>` 만 써도 됩니다. `std::is_pointer_v<T>` 는 아래와 같이 정의되어 있습니다.

```
template <class T>
inline constexpr bool is_pointer_v = is_pointer<T>::value;
```

(위 코드가 무엇을 뜻하는지 이해 하실 수 있겠죠?)

그렇다면 원래 함수는

```
template <typename T>
void show_value(T t) {
    if constexpr (std::is_pointer_t<T>) {
        std::cout << "포인터이다 : " << *t << std::endl;
    } else {
        std::cout << "포인터가 아니다 : " << t << std::endl;
    }
}
```

로 좀더 깔끔하게 바뀝니다.

C++ 20

C++ 20은 아직 나오지 않았지만, 추가될 기능들 중에 `constexpr vector (!!)` 와 `constexpr string (!!!)` 이 있습니다.¹⁾ 이를 위해서 `constexpr new` 와 `constexpr` 소멸자가 추가되었다고 하니 `constexpr` 은 좀 더 많은 곳에서 사용될 것 같습니다. 심지어 디폴트로 함수를 그냥 `constexpr` 로 만들어버리자는 이야기도 나오고 있고 말이지요.

C++ 는 정말 끊임없이 발전하고 있습니다. 덕분에 배워야 할 것들도 정말 끊임 없는 것 같네요..

그렇다면 이번 강좌는 여기에서 마치도록 하겠습니다. 다음 강좌에서는 C++ 의 여러가지 라이브러리에 대해 소개하는 시간을 갖도록 하겠습니다.

뭘 배웠지?

- `constexpr` 을 통해 컴파일 타임 상수인 객체를 선언할 수 있다.
- `const` 와 `constexpr` 은 다르다. `const` 는 컴파일 타임에 상수일 필요가 없다! (`const` 인 애들 중에서 `constexpr` 이 있다고 생각하면 된다)
- `constexpr` 로 정의된 함수는 인자로 리터럴을 전달하였을 때 컴파일 타임 상수를 리턴 한다.
- `constexpr` 생성자를 가진 클래스는 `constexpr` 객체를 생성할 수 있다.

1) 참고로 `string_view` 는 이미 `constexpr` 입니다.

decltype 와 std::decay

안녕하세요 여러분! 지난번 강좌에서 다룬 `constexpr` 는 잘 써먹고 계신가요? 그 동안 강좌에서 C++ 에서 사용되는 대부분의 키워드를 다루웠던 것 같은데 아직 하나 빠먹은 것이 있습니다. 바로 타입 관련 연산을 사용할 때 요긴하게 쓰이는 `decltype` 키워드입니다.

decltype

`decltype` 키워드는 C++ 11 에 추가된 키워드로, `decltype` 라는 이름의 함수 처럼 사용됩니다.

```
decltype(/* 타입을 알고자 하는 식*/)
```

이 때, `decltype` 는 함수와는 달리, 타입을 알고자 하는 식의 타입으로 치환되게 됩니다. 예를 들어서

```
#include <iostream>

struct A {
    double d;
};

int main() {
    int a = 3;
    decltype(a) b = 2; // int

    int& r_a = a;
    decltype(r_a) r_b = b; // int&

    int&& x = 3;
    decltype(x) y = 2; // int&&

    A* aa;
```

```
 decltype(aa->d) dd = 0.1; // double
}
```

위 코드의 경우 `decltype` 이 각각 `int`, `int&`, `int&&` 로 치환되어서 컴파일 되게 됩니다. 위와 같이 `decltype` 에 전달된 식이 팔호로 둘러쌓이지 않은 식별자 표현식(id-expression) 이라면 해당 식의 타입을 얻을 수 있습니다.

참고로 식별자 표현식이란 변수의 이름, 함수의 이름, `enum` 이름, 클래스 멤버 변수(`a.b` 나 `a->b` 같은 꼴) 등을 의미합니다. 엄밀한 정의는 [여기](#)에서 볼 수 있는데, 쉽게 생각하면 어떠한 연산을 하지 않고 단순히 객체 하나만을 가리키는 식이라고 보시면 됩니다.

그렇다면 만약에 `decltype` 에 식별자 표현식이 아닌 식을 전달하면 어떨까요? 그렇다면 해당 식의 값의 종류(value category)에 따라 달라집니다.

- 만일 식의 값 종류가 *xvalue* 라면 `decltype` 는 `T&&` 가 됩니다.
- 만일 식의 값 종류가 *lvalue* 라면 `decltype` 는 `T&` 가 됩니다.
- 만일 식의 값 종류가 *prvalue* 라면 `decltype` 는 `T` 가 됩니다.

잠깐! 지금 위 4 문장에서 너무나 많은 개념들이 나타났습니다. 값의 종류? *xvalue?* *lvalue?* *prvalue?* 애네들의 정체가 무엇인지 바로 알아보도록 하겠습니다.

Value Category

사람의 경우 이름과 나이라는 정보가 항상 따라다니듯이, 모든 C++ 식(expression)에는 두 가지 정보가 항상 따라다닙니다. 바로 식의 타입 과 값 카테고리(value category)입니다.

타입은 여태까지 공부하신 분들이라면 잘 알겠을 그 타입이 맞습니다. 하지만 값 카테고리가 뭔지는 약간 생소할 수도 있는데, 이는 이전에 이야기 하였던 좌측값/우측값을 일컫는 것입니다. 하지만 C++에서는 사실 총 5 가지의 값 카테고리가 존재합니다.

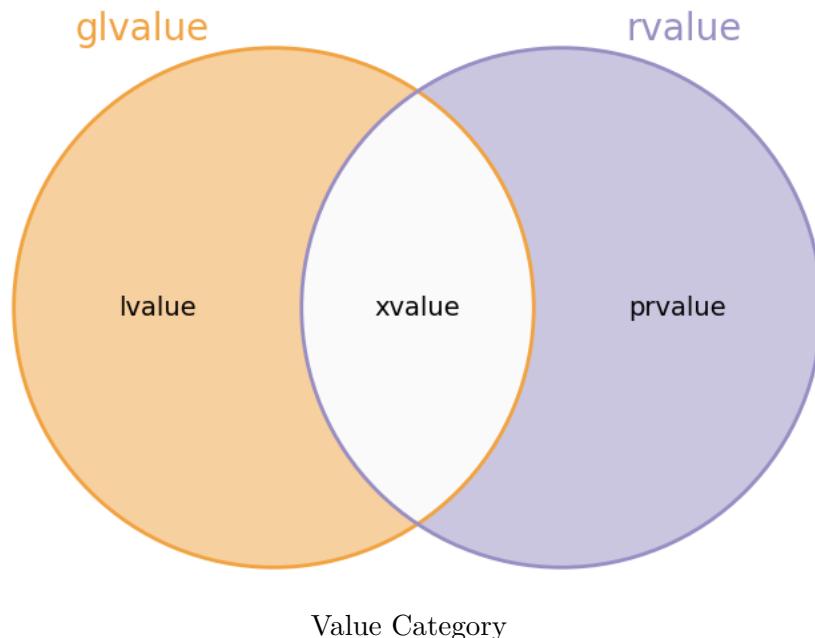
C++에서 어떠한 식의 값 카테고리를 따질 때 크게 두 가지 질문을 던질 수 있습니다.

- 정체를 알 수 있는가? 정체를 알 수 있다는 말은 해당 식이 어떤 다른 식과 같은 것인지 아닌지를 구분할 수 있다는 말입니다. 일반적인 변수라면 주소값을 취해서 구분할 수 있겠고, 함수의 경우라면 그냥 이름만 확인해보면 될 것입니다.
- 이동 시킬 수 있는가? 해당 식을 다른 곳으로 안전하게 이동할 수 있는지의 여부를 묻습니다. 즉 해당 식을 받는 이동 생성자, 이동 대입 연산자 등을 사용할 수 있어야만 합니다.

이를 바탕으로 값 카테고리를 구분해보자면 아래 표와 같습니다.

	이동 시킬 수 있다	이동 시킬 수 없다
정체를 알 수 있다	<i>xvalue</i>	<i>lvalue</i>
정체를 알 수 없다	<i>prvalue</i>	쓸모 없음!

덧붙여서 정체를 알 수 있는 모든 식들을 *glvalue* 라고 하며, 이동 시킬 수 있는 모든 식들을 *rvalue*라고 합니다. 그리고 C++ 에서 실체도 없으면서 이동도 시킬 수 없는 애들은 어차피 언어 상 아무런 의미를 갖지 않기 때문에 따로 부르는 명칭은 없습니다.¹⁾



위 그림을 보면 어떤 식으로 구분될 수 있는지 이해가 더 잘 될 것입니다.

lvalue

예를 들어서 평범한 *int* 타입 변수 *i* 를 생각해봅시다.

```
int i;
i;
```

그리고 우리가 *i* 라는 식을 썼을 때, 이 식의 정체를 알 수 있나요? 어떤 다른 식 *j* 라는 것이 있을 때 구분할 수 있을까요? 물론 이죠. *i* 라는 식의 주소값은 실제 변수 *i* 의 주소값이 될 것입니다. 그렇다면 *i* 는 이동 가능한가요? 아니죠. *int&& x = i;* 는 컴파일되지 않습니다. 따라서 *i* 는 *lvalue*입니다.

1) 각 값 카테고리를 단순하게 번역해보자면 *lvalue* 는 좌측값, *prvalue* (pure *rvalue*) 는 순수 우측값, *xvalue* (*eXpiring value*) 는 소멸하는 값, *glvalue* (*generalized lvalue*) 는 일반화된 좌측값, *rvalue* 는 우측값 이라 부를 수 있습니다. 근데 이렇게 한글 이름으로 부르는 것이 좀 더 헷갈리는 것 같아서 그냥 영문 이름으로 부르겠습니다.

이름을 가진 대부분의 객체들은 모두 *lvalue* 입니다. 왜냐하면 해당 객체의 주소값을 취할 수 있기 때문이죠. *lvalue* 카테고리 안에 들어가는 식들을 나열해보자면 ([자세한 내용은 여기를 참조!](#))

- 변수, 함수의 이름, 어떤 타입의 데이터 멤버 (예컨대 `std::endl`, `std::cin`) 등등
- 좌측값 레퍼런스를 리턴하는 함수의 호출식. `std::cout << 1`이나 `++it` 같은 것들
- `a = b`, `a += b`, `a *= b` 같이 복합 대입 연산자 식들
- `++a`, `--a` 같은 전위 증감 연산자 식들
- `a.m`, `p->m`과 같이 멤버를 참조할 때. 이 때 `m`은 `enum` 값이거나 `static` 이 아닌 멤버 함수인 경우 제외. (아래 설명 참조)

```
class A {
    int f();           // static 이 아닌 멤버 함수
    static int g();   // static 인 멤버 함수
}

A a;
a.g; // <-- lvalue
a.f; // <-- lvalue 아님 (아래 나올 prvalue)
```

- `a[n]`과 같은 배열 참조 식들
- 문자열 리터럴 "hi"

등등을 볼 수 있습니다. 특히 이 *lvalue* 들은 주소값 연산자(&)를 통해 해당 식의 주소값을 알아 낼 수 있습니다. 예를 들어서 `&++i`나 `&std::endl`은 모두 올바른 작업입니다. 또한 *lvalue* 들은 좌측값 레퍼런스를 초기화 하는데에 사용할 수 있습니다.

그렇다면 한 가지 퀴즈!

```
void f(int&& a) {
    a; // <-- ?
}

f(3);
```

위 코드에서 `a` 는 무슨 값 카테고리에 들어갈까요? `a` 는 우측값 레퍼런스기는 하지만, 식 `a` 의 경우는 *lvalue*입니다! 왜냐하면 이름이 있잖아요. 식 `a`의 타입은 우측값 레퍼런스가 맞지만, 식 `a`의 값 카테고리는 *lvalue*가 됩니다. 따라서 아래 같은 식들 모두 컴파일 됩니다.

```
#include <iostream>

void f(int&& a) { std::cout << &a; }
int main() { f(3); }
```

만약에 `a` 가 우측값 레퍼런스니까 `a` 는 우측값일꺼야 라고 생각했다면, 타입과 값 카테고리가 다른 개념이란 사실을 헷갈린 경우가 되겠습니다.

prvalue

```
int f() { return 10; }

f(); // <-- ?
```

그렇다면 위 코드의 `f()` 를 살펴봅시다. 위 식은 어떤 카테고리에 들어갈까요? 먼저 `f()` 는 실체가 있을까요? 쉽게 생각해서 `f()` 의 주소값을 취할 수 있을까요? 아닙니다. 하지만 `f()` 는 우측값 레퍼런스에 붙을 수 있습니다. 따라서 `f()` 는 *prvalue* 입니다.

prvalue 로 대표적인 것들은 아래와 같습니다.

- 문자열 리터럴을 제외 한 모든 리터럴들. 42, `true`, `nullptr` 같은 애들
- 레퍼런스가 아닌 것을 리턴하는 함수의 호출식. 예를 들어서 `str.substr(1, 2)`, `str1 + str2`
- 후위 증감 연산자 식. `a++`, `a--`
- 산술 연산자, 논리 연산자 식들. `a + b`, `a && b`, `a < b` 같은 것들을 말합니다. 물론, 이들은 연산자 오버로딩 된 경우들 말고 디폴트로 제공되는 것들을 말합니다.
- 주소값 연산자 식 `&a`
- `a.m`, `p->m` 과 같이 멤버를 참조할 때. 이 때 `m` 은 `enum` 값이거나 `static` 이 아닌 멤버 함수여야함.
- `this`
- `enum` 값
- 람다식 `[]() { return 0; };` 과 같은 애들.

등등... 여러가지가 있습니다.

이 *prvalue* 들은 정체를 알 수 없는 녀석들 이기 때문에 주소값을 취할 수 없습니다. 따라서 `&a++`이나 `&42` 와 같은 문장은 모두 오류입니다. 또한, *prvalue* 들은 식의 좌측에 올 수 없습니다. 하지만 *prvalue* 는 우측값 레퍼런스와 상수 좌측값 레퍼런스를 초기화하는데 사용할 수 있습니다. 예를 들어서;

```
const int& r = 42;
int&& rr = 42;
// int& rrr = 42; <-- 불가능
```

와 같이 됩니다.

xvalue

만일 값 카테고리가 *lvalue* 와 *prvalue* 두 개로만 구분된다면 문제가 있습니다. 만일 좌측값으로 분류되는 식을 이동 시킬 방법이 없기 때문입니다. 따라서 우리는 좌측값 처럼 정체가 있지만 이동도 시킬 수 있는 것들을 생각해봐야 합니다.

C++에서 이러한 형태의 값의 카테고리에 들어가는 식들로 가장 크게 우측값 레퍼런스를 리턴하는 함수의 호출식 을 들 수 있습니다. 대표적으로 `std::move(x)` 가 있지요. `std::move` 함수는 아래와 같이 생겼습니다.

```
template <class T>
constexpr typename std::remove_reference<T>::type&& move(T&& t) noexcept;
```

다른 복잡한 것들은 모두 건너 뛰더라도 `move` 의 리턴 타입 만큼은 우측값 레퍼런스 임을 알 수 있습니다. 따라서 `move` 를 호출한 식은 *lvalue* 처럼 좌측값 레퍼런스를 초기화하는데 사용할 수도 있고, *prvalue* 처럼 우측값 레퍼런스에 붙이거나 이동 생성자에 전달해서 이동 시킬 수 있습니다.

주의 사항

여기에서 다룬 값 카테고리에 대한 설명은 굉장히 가볍게 짚고 넘어간 것입니다. 좀 더 자세히 알고 싶은 분들은 언제나 그렇듯이 훌륭한 정보를 제공해주는 [cppreference](#) 와 C++ 표준 을 참조하시기를 바랍니다.

자, 그렇다면 이제 다시 `decltype` 에 대한 설명으로 돌아와보겠습니다. 앞서 말했듯이 `decltype` 에 식별자 표현식이 아닌 식이 전달된다면, 식의 타입이 `T` 라고 할 때 아래와 같은 방식으로 타입을 리턴한다고 하였습니다.

- 만일 식의 값 종류가 *xvalue* 라면 `decltype` 는 `T&&` 가 됩니다.
- 만일 식의 값 종류가 *lvalue* 라면 `decltype` 는 `T&` 가 됩니다.

- 만일 식의 값 종류가 *prvalue* 라면 decltype 는 T 가 됩니다.

그렇다면 예를 들어서 아래의 코드를 살펴봅시다.

```
int a, b;
decltype(a + b) c; // c 의 타입은?
```

위에서 본 바에 따르면 a + b 는 *prvalue* 이므로 a + b 식의 실제 타입인 int 로 추론됩니다. 따라서 위 식은 그냥 int c; 를 한 것과 똑같게 되겠지요.

그렇다면 아래와 같은 식은 어떨까요?

```
int a;
decltype((a)) b; // b 의 타입은?
```

일단 (a) 는 식별자 표현식이 아니기 때문에 어느 값 카테고리에 들어가는지 생각해봐야 합니다. 쉽게 생각하면 &(a) 와 같이 주소값 연산자를 적용할 수 있고, 당연히도 이동 불가능 이므로 *lvalue* 가 됩니다. 따라서 b 는 int 가 될 것이라는 예상과는 다르게 int& 로 추론됩니다!

이는 C++ 에서 괄호의 유무로 인해 무언가 결과가 달라지는 첫 번째 경우가 아닐까 싶네요.

decltype 의 쓰임새

그렇다면 decltype 는 도대체 왜 쓰이는 것일까요? 타입 추론이 필요한 부분에는 그냥 auto 로도 충분하지 않을까요? 예를 들어서

```
int i = 4;
auto j = i; // int j = i;
```

를 할 때나

```
int i = 4;
decltype(i) j = i; // int j = i;
```

는 같기 때문이지요. 하지만 auto 는 엄밀히 말하자면 정확한 타입을 표현하지 않습니다. 예를 들어서

```
const int i = 4;
auto j = i; // int j = i;
decltype(i) k = i; // const int k = i;
```

`auto` 의 경우 `const` 를 빼버리지만, `decltype` 의 경우 이를 그대로 보존합니다. 그 외에도 배열의 경우 `auto` 는 암시적으로 포인터로 변환하지만, `decltype` 의 경우 배열 타입 그대로를 전달할 수 있습니다. 예컨대

```
int arr[10];
auto arr2 = arr;      // int* arr2 = arr;
decltype(arr) arr3;  // int arr3[10];
```

이 되겠지요. 즉 `decltype` 를 이용하면 타입 그대로 정확하게 전달할 수 있습니다.

물론 이것 뿐만이 아닙니다. 템플릿 함수에서 어떤 객체의 타입이 템플릿 인자들에 의해서 결정되는 경우가 있습니다. 예를 들어서 아래와 같은 함수를 생각해봅시다.

```
template <typename T, typename U>
void add(T t, U u, /* 무슨 타입이 와야 할까요? */ result) {
    *result = t + u;
}
```

위 `add` 함수는 단순히 `t` 와 `u` 를 더해서 `result` 에 저장하는 함수입니다. 문제는 이 `result` 의 타입이 `t + u` 의 결과에 의해 결정된다는 사실입니다. 예를 들어서 `t` 가 `double` 이고 `u` 가 `int`라면 `result` 의 타입은 `double*` 이 되겠지요.

따라서 이런 경우에 `result` 에 타입이 올 자리에 `decltype` 를 사용해주면 됩니다.

```
template <typename T, typename U>
void add(T t, U u, decltype(t + u)* result) {
    *result = t + u;
}
```

와 같이 말이지요.

그렇다면 위 함수를 살짝 바꿔서 `result` 에 전달하는 대신에 그냥 더한 값을 리턴해버리는 함수를 만들어보면 어떨까요? 만약에 그냥

```
template <typename T, typename U>
decltype(t + u) add(T t, U u) {
    return t + u;
}
```

위와 같이 한다면 한 가지 문제가 있습니다. 위 식을 컴파일 한다면 아래와 같이 오류가 발생할 것입니다.

컴파일 오류

```
test2.cc:3:10: error: 't' was not declared in this scope
 decltype(t + u) add(T t, U, u) {
 ^
test2.cc:3:10: error: 't' was not declared in this scope
test2.cc:3:14: error: 'u' was not declared in this scope
 decltype(t + u) add(T t, U, u) {
```

컴파일러가 위 식을 컴파일 할 때 decltype 안의 t 와 u 를 보고 아니 이건 뭐지 한 것입니다. t 와 u 의 정의가 decltype 나중에 나오기 때문이지요. 이 경우 함수의 리턴값을 인자들 정의 부분 뒤에 써야 합니다. 이는 C++ 14 부터 추가된 아래와 같은 문법으로 구현 가능합니다.

```
template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

바로 리턴값 자리에는 그냥 auto 라고 써놓고, -> 뒤에 함수의 실제 리턴 타입을 지정해주는 것입니다. 이는 람다 함수 문법과 매우 유사합니다.

std::declval

다음으로 살펴볼 것은 decltype 처럼 C++ 11 에서 새로 추가된 std::declval 함수입니다. declval 은 decltype 과는 다르게 키워드가 아닌 <utility> 에 정의된 함수입니다.

예를 들어서 어떤 타입 T 의 f 라는 함수의 리턴 타입을 정의하고 싶다고 해봅시다. 그렇다면 decltype 를 이용하면 아래와 같은 코드를 작성할 수 있을 것입니다.

```
struct A {
    int f() { return 0; }
};

decltype(A().f()) ret_val; // int ret_val; 이 된다.
```

참고로 위 과정에서 실제로 A 의 객체가 생성되거나 함수 f 가 호출되거나 그러지는 않습니다. decltype 안에 들어가는 식은, 그냥 식의 형태로만 존재할 뿐 컴파일 시에, decltype() 전체 식이 타입으로 변환되기 때문에 decltype 안에 있는 식은 런타임 시에 실행되는 것이 아닙니다.

물론 그렇다고 해서 decltype 안에 문법상 틀린 식을 전달할 수 있는 것은 아닙니다. 예를 들어서 어떤 클래스에서 디폴트 생성자가 없다고 해봅시다.

```

struct B {
    B(int x) {}
    int f() { return 0; }
};

int main() {
    decltype(B().f()) ret_val; // B() 는 문법상 틀린 문장 :(
}

```

컴파일 한다면 아래와 같은 오류가 발생합니다.

컴파일 오류

```

test2.cc: In function ‘int main()’:
test2.cc:8:14: error: no matching function for call to ‘B::B()’
    decltype(B().f()) ret_val; // B() 는 문법상 틀린 문장 :(
                ^
test2.cc:3:3: note: candidate: B::B(int)
    B(int x) {}
                ^
test2.cc:3:3: note:    candidate expects 1 argument, 0 provided

```

그야 이유는 단순합니다.

B()

B()에 해당하는 생성자가 존재하지 않는다는 것이지요. 우리는 그냥 B의 멤버 함수 f의 타입 참조만 필요할 뿐인데, 실제 B 객체를 생성할 것도 아닌데도 B의 생성자 규칙에 맞는 코드를 작성해야합니다.

물론 우리는 그냥 B(1)과 같이 쓰면 된다는 것을 알고 있습니다. 그런데, 아래와 같은 상황을 생각해보세요.

```

template <typename T>
decltype(T().f()) call_f_and_return(T& t) {
    return t.f();
}

```

위 함수는 어떤 임의의 타입 T의 객체를 받아서 해당 객체의 멤버함수 f를 호출해주는 함수입니다. 이 함수를 이용하는 객체들에 멤버 함수 f가 정의되어 있다고 가정한다면, 모두 이용할 수 있습니다. 문제는 모든 타입 T들이 디폴트 생성자 T()를 정의하고 있지 않을 수도 있다는 말입니다.

```

template <typename T>
decltype(T().f()) call_f_and_return(T& t) {
    return t.f();
}

struct A {
    int f() { return 0; }
};

struct B {
    B(int x) {}
    int f() { return 0; }
};

int main() {
    A a;
    B b(1);

    call_f_and_return(a); // ok
    call_f_and_return(b); // BAD
}

```

컴파일 하였다면

컴파일 오류

```

test2.cc: In function ‘int main()’:
test2.cc:18:22: error: no matching function for call to
  ↳ ‘call_f_and_return(B&)’
      call_f_and_return(b); // BAD
          ^
test2.cc:2:19: note: candidate: template<class T> decltype
  ↳ (T().f()) call_f_and_return(T&)
  decltype(T().f()) call_f_and_return(T& t) {
          ~~~~~
test2.cc:2:19: note:   template argument deduction/substitution
  ↳ failed:
test2.cc: In substitution of ‘template<class T> decltype (T().f())’
  ↳ call_f_and_return(T&) [with T = B]:
test2.cc:18:22:   required from here
test2.cc:2:10: error: no matching function for call to ‘B::B()’
  decltype(T().f()) call_f_and_return(T& t) {
          ~~

```

와 같이 오류가 발생합니다.

위 경우 `call_f_and_return` 함수에 `a` 를 전달했을 때에는 `A` 에 디폴트 생성자가 있으므로 잘 컴파일 되지만, `b` 를 전달할 때에는 `B` 에 디폴트 생성자가 없으므로 오류가 발생하게 됩니다.

따라서 위처럼 직접 생성자를 사용하는 방식은 전달되는 타입들의 생성자가 모두 같은 풀이지 않을 경우 문제가 생깁니다.

이 문제는 `std::declval` 를 사용하면 깔끔하게 해결할 수 있습니다.

```
#include <utility>

template <typename T>
decltype(std::declval<T>().f()) call_f_and_return(T& t) {
    return t.f();
}

struct A {
    int f() { return 0; }
};

struct B {
    B(int x) {}
    int f() { return 0; }
};

int main() {
    A a;
    B b(1);

    call_f_and_return(a); // ok
    call_f_and_return(b); // ok
}
```

위 코드는 잘 컴파일 됩니다.

`std::declval` 에 타입 `T` 를 전달하면, `T` 의 생성자를 직접 호출하지 않더라도 `T` 가 생성된 객체를 나타낼 수 있습니다. 즉,

```
std::declval<T>()
```

를 통해 심지어 `T` 에 생성자가 존재하지 않더라도 마치 `T()` 를 한 것과 같은 효과를 낼 수 있지요. 따라서 앞서 발생하였던 생성자의 형태가 모두 달라서 발생하는 오류를 막을 수 있습니다.

참고로 `declval` 함수를 타입 연산에서만 사용해야지, 실제로 런타임에 사용하면 오류가 발생합니다.

```
#include <utility>
```

```
struct B {
    B(int x) {}
    int f() { return 0; }
};

int main() { B b = std::declval<B>(); }
```

컴파일 하였다면

컴파일 오류

```
/usr/include/c++/7/type_traits: In instantiation of ‘typename
→ std::add_rvalue_reference< <template-parameter-1-1> >::type
→ std::declval() [with _Tp = B; typename
→ std::add_rvalue_reference< <template-parameter-1-1> >::type =
→ B&&]’:
test2.cc:9:25:   required from here
/usr/include/c++/7/type_traits:2256:7: error: static assertion
→ failed: declval() must not be used!
    static_assert(__declval_protector<_Tp>::__stop,
    ^~~~~~
```

와 같이 오류가 발생합니다.

참고로 C++ 14 부터는 함수의 리턴 타입을 컴파일러가 알아서 유추해주는 기능이 추가되었습니다.
이 경우 그냥 함수 리턴 타입을 auto 로 지정해주면 됩니다.

```
template <typename T>
auto call_f_and_return(T& t) {
    return t.f();
}
```

따라서 위처럼 간단하게 쓸 수 있습니다.

물론 그렇다고 해서 declval 의 쓰임새가 없어지느냐? 아닙니다. 다음 강좌에서 <type_traits> 라이브러리를 다루면서 decltype 과 std::declval 을 사용한 놀라운 템플릿 메타프로그래밍 기법들을 소개하고자 합니다.

그럼 이번 강좌는 여기에서 마치도록 하겠습니다.

뭘 배웠지?

- `decltype` 키워드를 통해서 우리가 원하는 식의 타입을 알 수 있습니다. 만일 해당 식이 단순한 식별자 표현식 (identifier expression) 이라면 그냥 그 식의 타입으로 치환됩니다. 그 이외의 경우라면 해당 식의 값 카테고리가 뭐냐에 따라서 `decltype` 의 타입이 정해집니다.
- C++ 의 모든 식에는 두 가지 꼬리표가 따라다니는데 하나는 타입이고, 다른 하나는 값 카테고리입니다.
- 값 카테고리는 크게 3 가지 종류로 *lvalue*, *prvalue*, *xvalue* 가 있습니다.
- `std::declval` 함수를 사용해서 원하는 타입의 생성자 호출을 우회해서 멤버 함수의 타입에 접근할 수 있습니다.

다양한 C++ 표준 라이브러리 소개

type_traits 라이브러리, SFINAE, enable_if

안녕하세요 여러분! 앞으로 총 5 강좌에 걸쳐서 C++ 11 이후에 추가된 여러가지 표준 라이브러리 들에 대해서 다루어볼 예정입니다.

이번 강좌에서는 C++ 에서 타입 관련 연산을 위한 템플릿 메타 함수 들을 제공해주는 `type_traits` 라이브러리에 대해서 알아보도록 하겠습니다.

공포의 템플릿

아무래도 여기 까지 강좌를 보신 분들이라면 조금 복잡한 C++ 코드를 여러 경로에서 접해 보셨을 것입니다. 그렇다면 아마 아래와 같은 혐오스러운 템플릿 코드도 보셨을 테지요.

```
template <class _CharT, class _Traits, class _Yp, class _Dp>
typename enable_if<
    is_same<void, typename __void_t<decltype(
        declval<basic_ostream<_CharT, _Traits>&>() << declval<
            typename unique_ptr<_Yp, _Dp>::pointer>())>::type>::value,
    basic_ostream<_CharT, _Traits>&>::type
operator<<(basic_ostream<_CharT, _Traits>& __os,
    unique_ptr<_Yp, _Dp> const& __p) {
    return __os << __p.get();
}
```

아마 위 코드를 보신 여러분들의 속마음은..



WTF

와 같겠죠. 아니 저게 도대체 뭐야!

위 코드는 libc++ 라이브러리에서 가져온 코드로, `unique_ptr`의 주소값을 출력해주는 `basic_ostream`의 `operator<<` 연산자를 구현한 것입니다. 도대체 왜 C++ 개발자들은 저런 험오스러운 코드를 작성하는 것일까요?¹⁾

사실 `type_traits` 라이브러리들의 템플릿 메타 함수 (`template meta function`)들을 잘 이해만 한다면 위와 같은 코드는 무리없이 해석할 수 있습니다. 이 강좌 끝에 도달하게 된다면 여러분들 역시 위 코드를 보고서도 크게 무리 없이 이해할 수 있을 것입니다.

템플릿 메타 함수

템플릿 메타 함수란, 사실 함수는 아니지만 마치 함수 처럼 동작하는 템플릿 클래스들을 이야기 합니다. 이들이 메타 함수인 이유는 보통의 함수들은 값에 대해 연산을 수행하지만, 메타 함수는 타입에 대해 연산을 수행한다는 점이 조금 다릅니다.

예를 들어서 어떤 수가 음수인지 아닌지 판별하는 함수 `is_negative` 가 있다고 해봅시다. 그렇다면 이 함수는 아래처럼 사용할겁니다.

```
if (is_negative(x)) {  
    // Do something...  
}
```

1) 여담이지만 대표적인 C++ 표준 라이브러리로 GCC에서 사용하는 libstdc++ 와 Clang에서 사용하는 libc++ 을 뽑을 수 있습니다.

템플릿 메타 함수도 매우 비슷합니다. 예를 들어서 어떤 타입이 `void` 인지 아닌지 판단하는 `is_void` 함수가 있다고 해봅시다. 그렇다면 이 함수는 아래와 같이 사용하게 됩니다.

```
// 어떤 타입 T 가 있어서
if (is_void<T>::value) {
    // Do something
}
```

아래 예제를 통해 실제로 코드를 실행해보세요.

```
#include <iostream>
#include <type_traits>

template <typename T>
void tell_type() {
    if (std::is_void<T>::value) {
        std::cout << "T 는 void ! \n";
    } else {
        std::cout << "T 는 void 가 아니다. \n";
    }
}

int main() {
    tell_type<int>(); // void 아님!
    tell_type<void>(); // void!
}
```

성공적으로 컴파일 하였다면

실행 결과

```
T 는 void 가 아니다.
T 는 void !
```

보시다시피 일반적인 함수와 몇 가지 차이점이 있습니다. 가장 중요한 점은 템플릿 메타 함수들은 실제론 함수가 아니라는 점입니다. 만일 함수였다면 () 를 통해서 호출을 했겠지요. 하지만 `is_void` 는 그렇지 않습니다. () 대신에 <> 를 통해서 함수 인자가 아닌 템플릿 인자를 전달하고 있습니다. 실제로 `is_void` 는 클래스로 구현되어 있습니다.

`is_void`

기존에 템플릿 메타프로그래밍 강좌를 숙지하신 분들은 알고 계시겠지만, 템플릿 메타프로그래밍에서 **if** 문은 템플릿 특수화를 통해서 구현된다고 하였습니다. `is_void` 의 경우도 마찬가지입니다.

```
#include <iostream>

template <typename T>
struct is_void {
    static constexpr bool value = false;
};

template <>
struct is_void<void> {
    static constexpr bool value = true;
};

template <typename T>
void tell_type() {
    if (is_void<T>::value) {
        std::cout << "T 는 void ! \n";
    } else {
        std::cout << "T 는 void 가 아니다. \n";
    }
}

int main() {
    tell_type<int>(); // void 아님!

    tell_type<void>(); // void!
}
```

성공적으로 컴파일 하였다면

실행 결과

```
T 는 void 가 아니다.
T 는 void !
```

와 같이 나옵니다.

```
template <typename T>
struct is_void {
    static constexpr bool value = false;
};

template <>
struct is_void<void> {
    static constexpr bool value = true;
};
```

위는 실제 `std::is_void`의 코드를 가져온 것은 아니지만 (매우 비슷합니다), `is_void` 가 어떠한 원리로 작동하는지 보는데 충분하다고 생각합니다. 템플릿 강좌를 잘 들으신 분이라면,

```
template <typename T>
struct is_void {
```

는 일반적인 모든 타입 `T`에 대해서 매칭이 되고,

```
template <>
struct is_void<void> {
```

는 `void`에 대해 특수화 된 클래스이죠.

따라서 `is_void<void>`를 하게 된다면, 바로 위 특수화 된 템플릿이 매칭이 되어서 `value`가 `true`가 되고, 그 외의 타입의 경우에는 맨 위의 일반적인 템플릿 클래스가 매칭이 되어서 `value`가 `false`가 될 것입니다. 따라서

```
if (is_void<T>::value) {
    std::cout << "T 는 void ! \n";
} else {
    std::cout << "T 는 void 가 아니다. \n";
}
```

위 부분에서 `is_void`의 `value`가 `true`일 때와 `false`일 때 적절히 나눠서 처리할 수 있습니다.

C++ 표준 라이브러리 중 하나인 `type_traits`에서는 `is_void`처럼 타입들에 대해서 여러 가지 연산을 수행할 수 있는 메타 함수들을 제공하고 있습니다. 한 가지 더 예를 들어보자면 정수 타입인지 확인해주는 `is_integral`이 있습니다.

```
#include <iostream>
#include <type_traits>

class A {};

// 정수 타입만 받는 함수
template <typename T>
void only_integer(const T& t) {
    static_assert(std::is_integral<T>::value);
    std::cout << "T is an integer \n";
}

int main() {
    int n = 3;
    only_integer(n);
```

```
A a;
only_integer(a);
}
```

컴파일 하였다면

컴파일 오류

```
test2.cc: In instantiation of ‘void only_integer(const T&) [with T
         = A]’:
test2.cc:17:17:   required from here
test2.cc:8:3: error: static assertion failed
  static_assert(std::is_integral<T>::value);
  ^~~~~~
```

와 같은 오류 메세지를 볼 수 있습니다.

```
static_assert(std::is_integral<T>::value);
```

static_assert 는 C++ 11 에 추가된 키워드로 (함수가 아닙니다.), 인자로 전달된 식이 참인지 아닌지를 컴파일 타임에 확인합니다. 다시 말해 **bool** 타입의 **constexpr** 만 **static_assert** 로 확인할 수 있고 그 외의 경우에는 컴파일 오류가 발생합니다.

만약에 **static_assert** 에 전달된 식이 참 이라면, 컴파일러에 의해 해당 식은 무시되고, 거짓 이라면 해당 문장에서 컴파일 오류를 발생시키게 됩니다.

따라서 **static_assert** 와 **std::is_integral** 을 잘 조합해서 **T** 가 반드시 정수 타입임을 강제할 수 있습니다. 위

```
int n = 3;
only_integer(n);

A a;
only_integer(a);
```

위와 같이 **only_integer** 에 **n** 을 전달한다면 **T** 가 **int** 로 추론되어 **is_integral** 의 **value** 가 참이 되겠지만, 그냥 일반 클래스 객체인 **a** 를 전달한다면 **false** 가 되어서 위처럼 *static assertion failed* 라는 컴파일 오류가 발생하겠지요.

이처럼 **static_assert** 와 **type_traits** 의 메타 함수들을 잘 사용한다면 특정 타입만 받는 함수를 간단하게 작성할 수 있습니다.

is_class

`type_traits`에 정의되어 있는 메타 함수들 중에서 흥미로운 함수로 `is_class`가 있습니다. 이 메타 함수는 인자로 전달된 타입이 클래스 인지 아닌지 확인하는 메타 함수입니다.

지금 잠시 눈을 감고 어떤 타입 `T`가 클래스 인지 아닌지 어떤식으로 확인할 것인지 생각해보세요. 그닥 방법이 떠오르지 않죠? 저도 그렇습니다. 실제로, `is_class`가 구현된 방법은 매우 기괴합니다. `cppreference`에서 가져온 코드를 살펴보자면;

```
namespace detail {
template <class T>
char test(int T::*);
struct two {
    char c[2];
};
template <class T>
two test(...);
} // namespace detail

template <class T>
struct is_class
: std::integral_constant<bool, sizeof(detail::test<T>(0)) == 1 &&
    !std::is_union<T>::value> {};
```

흠. 아무래도 위에서 쓰인 사진을 다시 가져와야겠습니다.



WTF

위 코드를 이해하기 위해서는 먼저 `std::integral_constant` 가 뭘 하는 녀석인지부터 알아야

합니다. `integral_constant` 는 `std::integral_constant<T, T v>` 로 정의되어 있는데, 그냥 `v` 를 `static` 인자로 가지는 클래스입니다. 쉽게 말해 그냥 어떠한 값을 `static` 객체로 가지고 있는 클래스를 만들어주는 템플릿이라고 생각하면 됩니다.

예를 들어서 `std::integral_constant<bool, false>` 는 그냥 `integral_constant<bool, false>::value` 가 `false` 인 클래스입니다. 따라서 만약에

```
sizeof(detail::test<T>(0)) == 1 && !std::is_union<T>::value
```

이 부분이 `false` 라면 `is_class` 는 그냥

```
template <class T>
struct is_class : std::integral_constant<bool, false> {};
```

로 정의되고, 따라서 `is_class::value` 는 `false` 가 될 것입니다. 반면에 해당 부분이 `true` 로 연산된다면 `is_class::value` 역시 `true` 가 되겠지요. 결과적으로

```
sizeof(detail::test<T>(0)) == 1 && !std::is_union<T>::value
```

위 코드는 `T` 가 클래스라면 참이고 클래스가 아니라면 거짓이 될 것입니다.

그렇다면 앞 부분인 `sizeof(detail::test<T>(0)) == 1` 은 왜 `T` 가 클래스 일 때만 1 이 될까요?

데이터 멤버를 가리키는 포인터 (Pointer to Data member)

```
template <class T>
char test(int T::*);
```

먼저 위 부분을 살펴봅시다. 아마도 `int T::*` 라는 문법이 매우 생소하실 것이라 생각합니다. 이는 `T` 의 `int` 멤버를 가리키는 포인터라는 의미입니다. 아무래도 말로 설명하는 것 보다 아래 예제 하나를 보는 것이 이해가 더 빠릅니다.

```
#include <iostream>
#include <type_traits>

class A {
public:
    int n;

    A(int n) : n(n) {}
}
```

```

};

int main() {
    int A::*p_n = &A::n;

    A a(3);
    std::cout << "a.n : " << a.n << std::endl;
    std::cout << "a.*p_n : " << a.*p_n << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

a.n : 3
a.*p_n : 3

```

와 같이 나옵니다.

```
int A::*p_n
```

위 `p_n` 은 `A` 의 `int` 멤버를 가리킬 수 있는 포인터를 의미합니다. 이 때 `p_n` 이 실제 존재하는 어떠한 객체의 `int` 멤버를 가리키는 것이 아닙니다!

```
int A::*p_n = &A::n;
```

위에서 정의한 방식을 보았듯이 이제 `p_n` 을 역참조 하게 된다면 이는 마치 `A` 의 `n` 을 참조하는 식으로 사용할 수 있습니다. 따라서

```

std::cout << "a.n : " << a.n << std::endl;
std::cout << "a.*p_n : " << a.*p_n << std::endl;

```

`a.n`이나 `a.*p_n`이나 같은 문장이 됩니다. 이와 같은 형태의 포인터를 데이터 멤버를 가리키는 포인터라고 합니다. 그리고 여기에 한 가지 제한점이 있습니다. 바로 이 문법은 클래스에만 사용할 수 있다는 것인지요.

```
template <class T>
char test(int T::*);
```

따라서 위 문장은 T 가 클래스가 아니라면 불가능한 문장입니다. 참고로 위 문장은 T 가 클래스라면 해당 클래스에 int 데이터 멤버가 없어도 유효한 문장입니다. 다만 아무 것도 가리킬 수 없겠지요. 하지만 어차피 여기선 필요 없습니다. T 가 클래스 인지 아닌지 판별하는데에만 사용하니까요!

두 번째 문장인

```
struct two {
    char c[2];
};

template <class T>
two test(...);
```

를 살펴봅시다. 이 test 함수의 경우 사실 T 가 무엇이냐에 관계없이 항상 인스턴스화 될 수 있습니다. test 함수 자체도 이전에 가변 길이 템플릿 함수에서 다른 것처럼 그냥 임의 개수의 인자를 받는 함수입니다.

자 그렇다면 T 가 클래스라고 해봅시다. `detail::test<T>(0)` 를 컴파일 할 때, 컴파일러는 1 번 후보인

```
template <class T>
char test(int T::*); // (1)
```

와 2 번 후보인

```
struct two {
    char c[2];
};

template <class T>
two test(...); // (2)
```

사이에서 어떤 것으로 오버로딩 할지 결정을 해야 합니다. 이 경우 1 번이 좀 더 구체적이므로 (인자가 명시되어 있음) 우선순위가 더 높기 때문에 1 번으로 오버로딩 됩니다. 따라서 `test<T>(0)` 의 리턴 타입은 `char` 이 되고 `sizeof(char)` 은 1 이므로 통과가 되겠네요.

반면에 T 가 클래스가 아니라면;

```
template <class T>
char test(int T::*); // (1)
```

위 문법은 불가능한 문법입니다. 이 경우 컴파일 오류가 발생되는 것이 아니라 오버로딩 후보군에서 제외됩니다. (이 부분에 대해서 아래에서 좀 더 자세히 설명하겠습니다.) 따라서, 2 번이 유일한 후보군 이므로, `detail::test<T>(0)` 의 리턴 타입은 `two` 가 되겠지요. 이 때 `two` 는 `char`

c[2] 이므로, `sizeof` 가 2가 됩니다. 덕분에 `is_class` 의 `value` 는 `false` 로 연산이 되겠네요.

```
sizeof(detail::test<T>(0)) == 1 && !std::is_union<T>::value
```

그렇다면 위 식의 앞부분은 T 가 클래스 일 때 참이 되고 클래스가 아니라면 거짓이 됨을 알 수 있었습니다. 참고로 C++ 에서 데이터 멤버를 가리키는 포인터가 허용되는 것은 클래스와 공용체(union) 딱 두 가지가 있습니다. 따라서 `sizeof(detail::test<T>(0)) == 1` 는 T 가 공용체 일 때도 성립하기 때문에 확실히 클래스임을 보이기 위해서는 추가적으로 `is_union` 을 통해 공용체가 아님을 확인해야 합니다.

참고로 `is_union` 이 어떻게 구현되어 있는지 궁금해 하시는 분들을 위해 안타까운 소식을 전하자면, C++ 에선 클래스와 공용체를 구별할 수 있는 방법이 없습니다. 따라서 `is_union` 은 **컴파일 러에 직접 의존한 방식으로 구현**되어 있기 때문에 자세한 내용은 생략하겠습니다.

아무튼 결과적으로 위 식은 T 가 클래스 일 때에만 참이 되고 나머지 경우에는 모두 거짓으로 연산됩니다.

치환 오류는 컴파일 오류가 아니다 (Substitution failure is not an error - SFINAE)

그렇다면 방금 전에 이야기했던 컴파일 오류 시에 오버로딩 후보군에서 제외된다 라는 말을 다시 짚고 넘어가고 싶습니다.

예를 들어서 아래와 같은 코드를 살펴보세요.

```
#include <iostream>

template <typename T>
void test(typename T::x a) {
    std::cout << "T::x \n";
}

template <typename T>
void test(typename T::y b) {
    std::cout << "T::y \n";
}

struct A {
    using x = int;
};

struct B {
```

```

using y = int;
};

int main() {
    test<A>(33);

    test<B>(22);
}

```

성공적으로 컴파일 하였다면

실행 결과

```

T::x
T::y

```

와 같이 나옵니다.

여러분들이 템플릿 함수를 사용 할 때, 컴파일러는 템플릿 인자의 타입들을 유추한 다음에, 템플릿 인자들을 해당 타입으로 치환하게 됩니다. 여기서 문제는 템플릿 인자들을 유추한 타입으로 치환을 할 때 문법적으로 말이 안되는 경우들이 있기 마련입니다.

예를 들어서;

```
test<A>(33);
```

위 문장의 경우 우리가 템플릿 인자로 A를 전달하였으므로,

```

template <typename T>
void test(typename T::x a) {
    std::cout << "T::x \n";
}

template <typename T>
void test(typename T::y b) {
    std::cout << "T::y \n";
}

```

위 두 함수들은 각각

```

void test(A::x a) { std::cout << "T::x \n"; }

void test(A::y b) { std::cout << "T::y \n"; }

```

로 치환되겠지요. 문제는 `A::y` 가 문법적으로 올바르지 않은 식이라는 점입니다. (클래스 `A`에 `y`라는 타입이 없습니다.) 그렇다면 컴파일러는 여기서 컴파일 오류를 발생시킬까요?

아닙니다! 바로 치환 오류는 컴파일 오류가 아니다 (**Substitution Failure Is Not An Error**) 흔히 줄여서 **SFINAE** 라는 원칙 때문에, 템플릿 인자 치환 후에 만들어진 식이 문법적으로 맞지 않는다면, 컴파일 오류를 발생 시키는 대신 단순히 함수의 오버로딩 후보군에서 제외만 시키게 됩니다.²⁾

따라서 위 경우 두 번째 `test` 함수의 경우 가능한 오버로딩 후보군에서 제외됩니다.

여기서 한 가지 중요한 점은, 컴파일러가 템플릿 인자 치환 시에 함수 내용 전체가 문법적으로 올바른지 확인하는 것이 아니라는 점입니다. 컴파일러는 단순히 함수의 인자들과 리턴 타입만이 문법적으로 올바른지를 확인합니다. 따라서, 함수 내부에서 문법적으로 올바르지 않은 내용이 있더라도 오버로딩 후보군에 남아 있게 됩니다.

```
#include <iostream>

template <typename T>
void test(typename T::x a) {
    typename T::y b;
}

template <typename T>
void test(typename T::y b) {
    std::cout << "T::y \n";
}

struct A {
    using x = int;
};

int main() { test<A>(11); }
```

컴파일 하였다면

컴파일 오류

```
test2.cc: In instantiation of ‘void test(typename T::x) [with T =
→ A; typename T::x = int]’:
test2.cc:22:13:   required from here
test2.cc:5:17: error: no type named ‘y’ in ‘struct A’
    typename T::y b;
           ^
```

2) 참고로 SFINAE 는 스피네라고 읽습니다

위와 같이 오류가 발생합니다.

만일 첫 번째 `test` 가 오버로딩 후보군에서 제외되었더라면, 템플릿 인자 `유추`가 실패하였다는 오류 메세지가 나와야 했을 것입니다. 하지만 위 경우, 템플릿 인자 `유추`는 성공 해서 첫 번째 `test`를 사용하였지만 해당 함수 내부에 `typename T::y b;`; 때문에 컴파일 할 수 없다는 의미입니다. 이렇게 SFINAE 를 활용하게 된다면 원하지 않는 타입들에 대해서 오버로딩 후보군에서 제외할 수 있습니다. `type_traits` 에는 해당 작업을 손쉽게 할 수 있는 메타 함수를 하나 제공하는데, 바로 `enable_if` 입니다.

enable_if

`enable_if` 는 SFINAE 를 통해서 조건에 맞지 않는 함수들을 오버로딩 후보군에서 쉽게 뺄 수 있게 도와주는 간단한 템플릿 메타 함수입니다. `enable_if` 는 다음과 같이 정의되어 있습니다.

```
template <bool B, class T = void>
struct enable_if;
```

이 때 `B` 부분에 우리가 확인하고픈 조건을 전달합니다. 만일 `B` 가 참으로 연산된다면 `enable_if::value` 의 타입이 `T` 가 되고, `B` 가 거짓이라면 `enable_if` 에 `value` 가 존재하지 않게 됩니다. 예를 들어서, 어떤 함수의 인자 `T` 가 정수 타입일 때만 오버로딩을 하고 싶다고 해봅시다. 그렇다면 해당 작업을 하는 `enable_if` 는 아래와 같이 쓸 수 있습니다.

```
std::enable_if<std::is_integral<T>::value>::type
```

위 처럼 `B` 자리에 원하는 조건인 `std::is_integral<T>::value` 를 전달한 것을 볼 수 있습니다. 실제 `enable_if` 를 사용한 함수는 아래와 같습니다.

```
#include <iostream>
#include <type_traits>

template <typename T,
          typename = typename std::enable_if<std::is_integral<T>::value>::type>
void test(const T& t) {
    std::cout << "t : " << t << std::endl;
}

int main() {
    test(1);      // int
    test(false);  // bool
    test('c');   // char
}
```

성공적으로 컴파일 하였다면

실행 결과

```
t : 1  
t : 0  
t : c
```

같이 나옵니다. 반면에 `test` 에 정수 타입이 아닌 객체를 전달할 경우

```
#include <iostream>  
#include <type_traits>  
  
template <typename T,  
          typename = typename std::enable_if<std::is_integral<T>::value>::type>  
void test(const T& t) {  
    std::cout << "t : " << t << std::endl;  
}  
  
struct A {};  
  
int main() { test(A{}); }
```

컴파일 하였을 경우 아래와 같은 오류가 발생합니다.

컴파일 오류

```
test2.cc: In function ‘int main()’:  
test2.cc:12:22: error: no matching function for call to ‘test(A)’  
  int main() { test(A{}); }  
                           ^  
  
test2.cc:6:6: note: candidate: template<class T, class> void  
  ↵  test(const T&)  
  void test(const T& t) {  
      ^~~~  
  
test2.cc:6:6: note:   template argument deduction/substitution  
  ↵  failed:  
test2.cc:5:11: error: no type named ‘type’ in ‘struct  
  ↵  std::enable_if<false, void>’  
          typename = typename  
          ↵  std::enable_if<std::is_integral<T>::value>::type>
```

위 처럼 `test(A{})` 가 가능한 오버로딩이 없다고 나오게 됩니다.

```
template <typename T,
          typename = typename std::enable_if<std::is_integral<T>::value>::type>
```

아마 이제는 잘 이해 하시겠지만, 위 코드가 어떻게 동작하는지 다시 한 번 설명을 해보자면, `std::integral<T>::value` 가 참 일 때에만 `std::enable_if` 에 `value` 가 정의되어서 위 코드가 컴파일 오류를 발생시키지 않습니다.

그리고 `typename =` 부분은 템플릿에 디폴트 인자를 전달하는 부분인데, 원래에는 `typename U =` 처럼 템플릿 인자를 받지만 우리의 경우 저 식 자체만 필요하기 때문에 굳이 인자를 정의할 필요가 없습니다.

그리고 `std::enable_if` 앞에 추가적으로 `typename` 이 또 붙는 이유는 `std::enable_if<>::type` 이 의존 타입 이기 때문입니다.

사실 위처럼 길게 쓰면 이해하기 힘든데, C++ 14 부터 `enable_if<>::value` 와 같이 자주 쓰이는 패턴에 대한 alias들을 활용할 수 있습니다. 그러면 아래처럼 조금 더 간단하게 표현됩니다.

```
template <typename T, typename = std::enable_if_t<std::is_integral_v<T>>>
void test(const T& t) {
    std::cout << "t : " << t << std::endl;
}
```

참고로

```
template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type; // C++ 14 부터 사용 가능

template <class T>
inline constexpr bool is_integral_v =
    is_integral<T>::value; // C++ 17 부터 사용 가능.
```

위 처럼 정의되어 있습니다.

enable_if 구현 방식

아마도 여기 까지 강의를 잘 따라 오신 분들이라면 `enable_if` 가 어떤 식으로 작성되어 있을 지 짐작 하실 수 있을 것이라 생각합니다. 실제 구현 방식을 보면 매우 간단합니다.

```
template <bool B, class T = void>
struct enable_if {};
```

```
template <class T>
struct enable_if<true, T> {
    typedef T type;
};
```

위 처럼 B 가 true 일 때 특수화 된 버전에서 type 이 정의되어 있고, 나머지 경우에는 그냥 enable_if 내부에 아무 것도 정의되어 있지 않습니다. 따라서 B 가 참 일 때만 std::enable_if::value 가 올바른 문장이 되어서 오버로딩 후보군에서 제외되지 않고 남을 수 있게 된 것입니다.

enable_if 또 다른 예시

여러분이 `vector` 클래스의 제작자라고 해봅시다. 그렇다면 `vector` 의 생성자로 아래 두 가지 형태를 제공할 것입니다.

```
template <typename T>
class vector {
public:
    // element 가 num 개 들어있는 vector 를 만든다.
    vector(size_t num, const T& element);

    // 반복자 start 부터 end 까지로 벡터 생성
    template <typename Iterator>
    vector(Iterator start, Iterator end);
};
```

첫 번째 생성자는 단순하게 원소가 num 개 들어있는 `vector` 를 만드는 생성자이고, 두 번째 생성자는 반복자 시작과 끝을 받는 생성자입니다. 참고로 반복자의 경우, 딱히 클래스가 따로 정해져 있는 것이 아니라 그냥 `start`, `end`, `++` 등등의 함수만 들어있는 클래스라면 반복자처럼 사용할 수 있습니다.

그렇다면 만약에 `vector` 클래스의 사용자가 아래와 같은 코드를 썼다면 어떤 식으로 해석되어 할까요?

```
vector<int> v(10, 3);
```

당연히도 사용자는 첫 번째 오버로드인 3 이 10 개 들어있는 벡터를 생성하기를 원했을 것입니다. 그런데 말이죠, 실제로 컴파일 해보면 아래와 같이 나옵니다.

```
#include <iostream>

template <typename T>
```

```

class vector {
    public:
        vector(size_t num, const T& element) {
            std::cout << element << " 를 " << num << " 개 만들기" << std::endl;
        }

        template <typename Iterator>
        vector(Iterator start, Iterator end) {
            std::cout << "반복자를 이용한 생성자 호출" << std::endl;
        }
};

int main() { vector<int> v(10, 3); }

```

성공적으로 컴파일 하였다면

실행 결과

반복자를 이용한 생성자 호출

와 같이 나옵니다. 의외지요? 사실 이렇게 나온 이유는 간단합니다. 우리가 원했던 버전의 오버로딩은

```
vector(size_t num, const T& element) {
```

와 같이 생겼습니다. 여기서 주목할 점은 num의 타입이 size_t라는 점입니다. size_t는 부호가 없는 정수 타입이지요. 문제는 v(10, 3)을 했을 때 10은 부호가 있는 정수라는 점입니다. 물론, C++ 컴파일러는 똑똑하기 때문에 이정도는 알아서 캐스팅 해줘서 넘어갈 수도 있었습니다. 다만 더 나은 후보가 없다는 가정 하에 말이죠.

```

template <typename Iterator>
vector(Iterator start, Iterator end) {
```

문제는 이 친구가 Iterator를 int로 오버로딩 한다면 v(10, 3)를 완벽하게 매칭 시킬 수 있다는 점입니다. 따라서 결과적으로 우리의 예상과는 다르게 반복자를 이용한 생성자 호출이 선택됩니다.

따라서 이 경우 Iterator가 실제 반복자임을 강제할 필요성이 있습니다. 그렇다면 만약에 is_iterator라는 메타 함수가 있다고 가정한다면, 위 코드를 아래와 같이 쓸 수 있습니다.³⁾

3) is_iterator는 표준에 정의되어 있는 메타 함수가 아닙니다.

```
template <typename Iterator,
          typename = std::enable_if_t<is_iterator<Iterator>::value>>
vector(Iterator start, Iterator end) {
    std::cout << "반복자를 이용한 생성자 호출" << std::endl;
}
```

이 경우 `Iterator` 가 실제로 반복자 일 경우에만 해당 `vector` 생성자가 오버로딩 후보군에 들어가겠지요.

특정 멤버 함수가 존재하는 타입 만을 받는 함수

여태 까지 `enable_if` 와 여러가지 메타 함수로 할 수 있었던 것들은 이러한 조건을 만족하는 타입을 인자로 받는 함수를 만들고 싶다 였습니다.

하지만 만약에 이러한 멤버 함수가 있는 타입을 인자로 받는 함수를 만들고 싶다는 어떨까요? 예를 들어서 멤버 함수로 `func` 이라는 것이 있는 클래스만 받고 싶다고 해봅시다.

그렇다면 아래와 같은 코드를 쓸 수 있을 것입니다.

```
#include <iostream>
#include <type_traits>

template <typename T, typename = decltype(std::declval<T>().func())>
void test(const T& t) {
    std::cout << "t.func() : " << t.func() << std::endl;
}

struct A {
    int func() const { return 1; }
};

int main() { test(A{}); }
```

성공적으로 컴파일 하였다면

실행 결과

```
t.func() : 1
```

와 같이 잘 나옵니다. 만약에 `func` 가 정의되어 있지 않은 클래스의 객체를 전달한다면

```
#include <iostream>
#include <type_traits>
```

```

template <typename T, typename = decltype(std::declval<T>().func())>
void test(const T& t) {
    std::cout << "t.func() : " << t.func() << std::endl;
}

struct A {
    int func() const { return 1; }
};

struct B {};

int main() { test(B{}); }

```

컴파일 시 아래와 같은 오류가 발생합니다.

컴파일 오류

```

test2.cc: In function ‘int main()’:
test2.cc:16:11: error: no matching function for call to ‘test(B)’
    test(B{});
               ^
test2.cc:5:6: note: candidate: template<class T, class> void
→  test(const T&)
void test(const T& t) {
    ^~~~
test2.cc:5:6: note:    template argument deduction/substitution
→  failed:
test2.cc:4:61: error: ‘struct B’ has no member named ‘func’
template <typename T, typename =
→  decltype(std::declval<T>().func())

```

보시다시피, `test(B{})` 를 오버로딩 하는 함수가 없다고 나와 있습니다. 왜냐하면 `decltype(std::declval<T>().func())` 이 올바르지 않은 문장이기 때문에 오버로딩 후보군에서 제외되었기 때문이지요.

만약에 `func()` 의 리턴 타입 까지 강제하고 싶다면 아래와 같이 `enable_if` 를 활용하면 됩니다.

```

#include <iostream>
#include <type_traits>

// T 는 반드시 정수 타입을 리턴하는 멤버 함수 func 을 가지고 있어야 한다.
template <typename T, typename = std::enable_if_t<
            std::is_integral_v<decltype(std::declval<T>().func())>>>

```

```

void test(const T& t) {
    std::cout << "t.func() : " << t.func() << std::endl;
}

struct A {
    int func() const { return 1; }
};

struct B {
    char func() const { return 'a'; }
};

int main() {
    test(A{});
    test(B{});
}

```

성공적으로 컴파일 하였다면

실행 결과

```
t.func() : 1
t.func() : a
```

와 같이 잘 나옵니다. 반면에 함수의 리턴 타입이 정수 타입이 아닌 경우

```

#include <iostream>
#include <type_traits>

template <typename T, typename = std::enable_if_t<
            std::is_integral_v<decltype(std::declval<T>().func())>>>
void test(const T& t) {
    std::cout << "t.func() : " << t.func() << std::endl;
}

struct A {
    int func() const { return 1; }
};

struct C {
    A func() const { return A{}; }
};

int main() { test(C{}); }

```

컴파일 하였다면

컴파일 오류

```
test2.cc: In function ‘int main()’:
test2.cc:24:11: error: no matching function for call to ‘test(C)’
    test(C{});  
          ^
test2.cc:6:6: note: candidate: template<class T, class> void
→  test(const T&)
void test(const T& t) {
    ^~~~
test2.cc:6:6: note:   template argument deduction/substitution
→  failed:
```

역시나 위처럼 `test(C{})` 의 가능한 오버로딩이 없다고 나오게 됩니다.

그렇다면 만약에 `func2` 말고도 여러 개의 함수를 확인하고 싶다면 어떨까요? 예를 들어서 컨테이너의 모든 원소들을 출력하는 `print` 함수를 작성하고 싶다고 해봅시다. 물론 주어진 타입 `T` 가 컨테이너 인지 아닌지 쉽게 알 수 있는 방법은 없지만 적어도 원소들을 출력하기 위해선 `begin`과 `end` 가 정의되어 있다는 사실은 알고 있지요. 따라서, 우리의 `print` 함수는 최소한 `T` 에 `begin`과 `end` 가 정의되어 있는지 확인해야 할 것입니다.

```
#include <iostream>
#include <set>
#include <type_traits>
#include <vector>

template <typename Cont, typename = decltype(std::declval<Cont>().begin()),
           typename = decltype(std::declval<Cont>().end())>
void print(const Cont& container) {
    std::cout << "[ ";
    for (auto it = container.begin(); it != container.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "] \n";
}

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    print(v);

    std::set<char> s = {'a', 'b', 'f', 'i'};
    print(s);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
[ 1 2 3 4 5 ]
[ a b f i ]
```

와 같이 잘 나옵니다. 반면에 `begin` 과 `end` 둘 다 정의되어 있지 않은 클래스의 경우

```
#include <iostream>
#include <type_traits>

template <typename Cont, typename = decltype(std::declval<Cont>().begin()),
           typename = decltype(std::declval<Cont>().end())>
void print(const Cont& container) {
    std::cout << "[ ";
    for (auto it = container.begin(); it != container.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "] \n";
}

struct Bad {
    void begin();
};

int main() { print(Bad{}); }
```

컴파일 하였다면

컴파일 오류

```
test.cc:21:3: error: no matching function for call to 'print'
    print(Bad{});
    ^~~~~~
test.cc:7:6: note: candidate template ignored: substitution
      failure [with Cont = Bad, $1 = void]: no member named
          'end' in 'Bad'
void print(const Cont& container) {
    ^
```

위 처럼 `print(Bad{})` 를 오버로딩 하는 함수가 없다는 오류가 발생하게 됩니다. 우리가 예상한 대로 다 잘 작동하는 것 처럼 보이지만 그래도 한 가지 개선할 여지가 있습니다.

```
template <typename Cont, typename = decltype(std::declval<Cont>().begin()),  
         typename = decltype(std::declval<Cont>().end())>
```

바로 디폴트 템플릿 인자 `typename =` 이 너무 많아진다는 점입니다. 위 템플릿을 그냥 제 3 자 입장에서 보았을 때 `Print` 함수가 정확히 어떠한 템플릿 인자를 받는지 쉽게 알아보기 힘듭니다. 또한 디폴트 템플릿 인자가 1 개였다면 그래도 그냥저냥 넘어갈 만 했지만, 2 개 이상 부터는 가독성이 너무 떨어집니다.

그래서 C++ 17 부터 `void_t` 라는 신기한 메타 함수가 추가되었습니다.

void_t

`void_t` 의 정의를 보면 놀랄 만큼 단순합니다.

```
template <class...>  
using void_t = void;
```

즉 가변길이 템플릿을 이용해서 `void_t` 에 템플릿 인자로 임의의 개수의 타입들을 전달할 수 있고, 어찌 되었든 `void_t` 는 결국 `void` 와 동일합니다.

```
void_t<A, B, C, D> // --> 결국 void
```

그런데 `void_t` 에 전달된 템플릿 인자들 중 문법적으로 올바르지 못한 템플릿 인자가 있다면 해당 `void_t` 를 사용한 템플릿 함수의 경우 `void` 가 되는 대신에 SFINAE 에 의해서 오버로딩 목록에서 제외가 되겠지요. 따라서

```
template <typename Cont, typename = decltype(std::declval<Cont>().begin()),  
         typename = decltype(std::declval<Cont>().end())>
```

위 식은 아래와 같이 좀 더 깔끔하게 다시 쓸 수 있습니다.

```
template <typename Cont,  
         typename = std::void_t<decltype(std::declval<Cont>().begin()),  
                           decltype(std::declval<Cont>().end())>>
```

즉 `void_t` 에 전달된 `decltype(std::declval<Cont>().begin())` 이나 `decltype(std::declval<Cont>().end())` 중 하나라도 문법적으로 올바르지 않다면 SFINAE 에 의해서 해당 `print`

함수는 오버로딩 후보군에서 제외됩니다. 반면에 `vector`처럼 두 코드가 문법적으로 성립하는 경우에는 `print` 가 잘 오버로딩 되겠네요.

물론 아직도 위 코드가 완벽한 것이 아닙니다. 만일 사용자가 실수로 템플릿 인자에 컨테이너 말고 인자를 한 개 더 전달했다고 해봅시다.

```
#include <iostream>
#include <type_traits>

template <typename Cont,
          typename = std::void_t<decltype(std::declval<Cont>().begin()),
                               decltype(std::declval<Cont>().end())>>
void print(const Cont& container) {
    std::cout << "[ ";
    for (auto it = container.begin(); it != container.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "]\\n";
}

struct Bad {};

int main() {
    // 위 print 는 오버로딩 후보군에서 제외되지 않음!
    print<Bad, void>(Bad{});
}
```

컴파일 하였다면

컴파일 오류

```
test2.cc: In instantiation of ‘void print(const Cont&) [with Cont
→   = Bad; <template-parameter-1-2> = void]’:
test2.cc:18:36:   required from here
test2.cc:10:28: error: ‘const struct Bad’ has no member named
→   ‘begin’
    for (auto it = container.begin(); it != container.end(); ++it)
    →   {
        ~~~~~^~~~~~
test2.cc:10:53: error: ‘const struct Bad’ has no member named
→   ‘end’
    for (auto it = container.begin(); it != container.end(); ++it)
    →   {
        ~~~~~^~~~
```

위와 같이 `print` 가 오버로딩 후보군에서 제외되지 않았음을 볼 수 있습니다. 왜냐하면 사용자가 실수로 `print` 의 템플릿 인자로 `Cont` 의 타입을 체크하는 자리에 `void`라는 인자를 전달하였기 때문에 디폴트 인자가 사용되지 않았습니다. 이 때문에 타입 체크를 생략하게 됩니다.

만약에 위 `print` 함수가 표준 라이브러리 함수들처럼 여러 사용자들을 고려해야 하는 상황이라면, 위와 같이 사용자가 실수 했을 때에도 정상적으로 작동할 수 있도록 설계해야 할 것입니다. 이를 위해선 타입 체크하는 부분을 다른 곳으로 빼야 합니다.

```
template <typename Cont>
std::void_t<decltype(std::declval<Cont>().begin()),  
           decltype(std::declval<Cont>().end())>  
print(const Cont& container)
```

따라서 완성된 코드가 위와 같습니다. 타입을 체크하는 부분을 템플릿의 디폴트 인자에서 함수의 리턴 타입으로 옮겼습니다. 이전에도 이야기 하였지만, 함수의 리턴 타입 역시 SFINAE 가 적용되는 부분이므로 동일한 효과를 낼 수 있습니다. 뿐만 아니라 템플릿 정의 부분에 불필요한 디폴트 인자가 들어가 있지 않으므로 사용자의 실수로부터 안전해졌습니다.

공포의 템플릿 다시 살펴보기

자 그럼 이제 맨 위에서 보았던 공포의 템플릿을 이해할 수 있는 능력치를 쌓은 것 같습니다.

```
template <class _CharT, class _Traits, class _Yp, class _Dp>  
typename enable_if<  
    is_same<void, typename __void_t<decltype(  
        (declval<basic_ostream<_CharT, _Traits>&() << declval<  
            typename unique_ptr<_Yp, _Dp>::pointer>()))>::type>::value,  
    basic_ostream<_CharT, _Traits>&::type  
operator<<(basic_ostream<_CharT, _Traits>& __os,  
            unique_ptr<_Yp, _Dp> const& __p) {  
    return __os << __p.get();  
}
```

위 코드는 표준 라이브러리에 들어가 있는 만큼 최대한 안전하게 설계되어야 합니다. 따라서 템플릿 디폴트 인자로 타입을 체크하는 대신 방금 우리가 소개한 방식처럼 함수 리턴 타입을 통해서 타입을 체크하고 있습니다.

```
__void_t<decltype((declval<basic_ostream<_CharT, _Traits>&()  
                  << declval<typename unique_ptr<_Yp, _Dp>::pointer>()))>::type
```

자 그렇다면 위 부분은 무슨 일을 하고 있는 것일까요? (참고로 `__void_t` 와 `std::void_t` 는 같은 함수입니다.) 바로

```
declval<basic_ostream<_CharT, _Traits>&>()
    << declval<typename unique_ptr<_Yp, _Dp>::pointer>()
```

가 문법 상 올바른 문장인지 확인하고 있는 것입니다. 다시 말해 `basic_ostream` 의 `operator <<` 가 `unique_ptr` 의 `pointer` 타입 객체를 출력할 수 있는지 확인하고 있는 것이지요. 만일 해당 타입 객체를 출력할 수 있다면 위 `_void_t` 는 `void` 로 연산될 것이고, 해당 문장이 문법 상 불가능 하다면 위 `operator <<` 는 오버로딩 목록에서 제외될 것입니다.

만약에 `basic_ostream` 이 `unique_ptr` 의 `pointer` 타입을 출력할 수 있다고 해봅시다. 그렇다면

```
typename enable_if<
    is_same<void, typename __void_t<decltype(
        declval<basic_ostream<_CharT, _Traits>&>() << declval<
            typename unique_ptr<_Yp, _Dp>::pointer>())>::type>::value,
    basic_ostream<_CharT, _Traits>&>::type
```

위 코드는

```
typename enable_if<is_same<void, void>::value,
    basic_ostream<_CharT, _Traits>&>::type
```

로 바뀔 것입니다. 참고로 `is_same` 은 `type_traits` 에 정의되어 있는 메타 함수로 인자로 전달된 두 타입이 같으면 `value` 가 `true` 아니면 `value` 가 `false` 가 되는 메타 함수입니다. 위 경우 두 타입이 `void` 로 같기 때문에 `is_same<void, void>::value` 는 `true` 가 됩니다.

따라서 위 식은

```
typename enable_if<true, basic_ostream<_CharT, _Traits>&>::type operator<<(
    basic_ostream<_CharT, _Traits>& __os, unique_ptr<_Yp, _Dp> const& __p) {
    return __os << __p.get();
}
```

가 되서 결과적으로 `enable_if` 에 의해

```
basic_ostream<_CharT, _Traits>& operator<<(basic_ostream<_CharT, _Traits>& __os,
                                                unique_ptr<_Yp, _Dp> const& __p) {
    return __os << __p.get();
}
```

가 되어서 우리가 원하는 함수가 됩니다.

상당히 복잡해 보이는 코드였지만, 알고보면 위에서 컨테이너를 사용한 예제와 큰 차이가 없습니다. 다만 그 예제의 경우 리턴값이 `void` 였던 대신에 `operator<<`는 `basic_ostream<_CharT, _Traits>&`를 리턴해야 하므로 `is_same`과 `enable_if`를 활용해서 리턴 타입을 바꿔준 것이라 볼 수 있습니다.

자 그렇다면 이번 강좌를 여기서 마치도록 하겠습니다. 이번 강좌를 통해서 `type_traits` 라이브러리와 복잡한 템플릿 정의 코드를 이해할 수 있는 능력을 키울 수 있었으면 좋겠습니다.

다음 강좌에서는 C++ 의 표준 라이브러리를 중 하나인 정규 표현식 라이브러리 (`<regex>`) 에 대해 살펴보도록 하겠습니다.

뭘 배웠지?

- `type_traits` 에 정의되어 있는 메타 함수들이 무엇인지 이해하였습니다.
- C++ 에서 템플릿 인자 치환 시 문법적으로 올바르지 않은 코드가 생성될 경우 컴파일 오류를 출력하는 대신 해당 함수를 오버로딩 후보군에서 제외합니다. 이 때, 컴파일러가 모든 코드를 치환 하는 것이 아니라 함수의 타입, 인자 정의 템플릿 인자 정의 부분만 살펴봅니다. 이와 같은 규칙을 SFINAE 라고 합니다.
- `enable_if` 를 통해서 원하는 타입만 받는 함수를 작성할 수 있습니다.
- `void_t` 를 통해서 원하는 타입만 받는 함수를 작성할 수 있습니다.

정규 표현식(<regex>) 라이브러리 소개

안녕하세요 여러분! 이번 강좌에서는 C++ 11 부터 표준에 포함된 정규 표현식(regular expression) 라이브러리에 대해서 간단하게 알아보는 시간을 가지도록 하겠습니다. 아마 프로그래밍을 오래 하신 분들이라면 정규 표현식 자체는 아마 여러 상황에서 사용해보셨으리라 생각합니다.

정규 표현식은 문자열에서 패턴을 찾는데 사용하는데, 이를 통해

- 주어진 문자열이 주어진 규칙에 맞는지 확인할 때
- 주어진 문자열에서 원하는 패턴의 문자열을 검색할 때
- 주어진 문자열에서 원하는 패턴의 문자열로 치환할 때

와 같은 경우에 매우 유용하게 사용됩니다.

그렇다면 각각의 상황에 맞게 C++에서 제공하는 정규 표현식 라이브러리를 어떻게 사용할지 살펴봅시다.

전체 문자열 매칭하기

만약에 여러분이 어떠한 서버를 관리하고 있는데, 해당 서버에서 매 시간마다 로그 파일을 생성한다고 해봅시다. 이 때 로그 파일은 db-(시간)-log.txt 와 같은 형태로 생성된다고 합니다.

그러다면 여러 가지 파일들이 섞여 있는 폴더에서 우리가 원하는 로그 파일들만 어떻게 간단히 읽어 들 수 있을까요? 일단 정규 표현식을 사용할 줄 아는 분들이라면 위 패턴을 보고 아래와 같은 정규 표현식을 떠올리실 것입니다.

```
db-\d*-log\.txt
```

여기서 \d* 는 임의의 개수의 숫자를 의미하는 것이고, . 앞에 \ 을 붙여준 이유는 . 을 그냥 썼을 때 임의의 문자로 해석되기 때문입니다.

그렇다면 해당 정규표현식을 바탕으로 C++을 통해 파일 이름이 주어진 패턴을 만족하는지 확인하는 프로그램을 작성해보도록 하겠습니다.

```
#include <iostream>
#include <regex>
#include <vector>

int main() {
    // 주어진 파일 이름들.
```

```

std::vector<std::string> file_names = {"db-123-log.txt", "db-124-log.txt",
                                         "not-db-log.txt", "db-12-log.txt",
                                         "db-12-log.jpg"};
std::regex re("db-\\d*-log\\.txt");
for (const auto &file_name : file_names) {
    // std::boolalpha 는 bool 을 0 과 1 대신에 false, true 로 표현하게 해줍니다.
    std::cout << file_name << ":" << std::boolalpha
        << std::regex_match(file_name, re) << '\n';
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

db-123-log.txt: true
db-124-log.txt: true
not-db-log.txt: false
db-12-log.txt: true
db-12-log.jpg: false

```

와 같이 주어진 정규 표현식에 맞는 파일 이름들만 true 로 나옴을 볼 수 있습니다.

```
std::regex re("db-\\d*-log\\.txt");
```

C++에서 정규 표현식을 사용하기 위해서는 먼저 위와 같이 정규 표현식 객체를 정의해야 합니다. 참고로 정규 표현식 문법의 종류와, 정규 표현식을 처리하는 엔진 역시 여러가지 종류가 있는데, 위 생성자에 추가적인 인자로 전달할 수 있습니다. 예를 들어서 grep 의 정규 표현식 문법을 사용하고 싶다면

```
std::regex re("db-\\d*-log\\.txt", std::regex::grep);
```

과 같이 전달하면 됩니다. 만약에 인자를 지정하지 않았다면 디폴트로 std::regex::ECMAScript 가 들어가게 됩니다. 어떤 문법을 사용할지 이외에도 몇 가지 특성을 더 추가할 수 있는데, 예를 들어서 std::regex::icase 를 전달한다면 대소 문자를 구분하지 않게 됩니다. 이 때 특성을 추가하는 방법은 | 로 연결하면 됩니다. 예를 들어서

```
std::regex re("db-\\d*-log\\.txt", std::regex::grep | std::regex::icase);
```

와 같이 말이지요.

참고로 정규 표현식의 성능이 중요할 경우에는 `std::regex::optimize` 를 추가적으로 전달할 수 있습니다. 이 경우 정규 표현식 객체를 생성하는데에는 시간이 좀 더 걸리지만 정규 표현식 객체를 사용하는 작업은 좀 더 빠르게 수행됩니다.

```
std::cout << file_name << ":" << std::boolalpha  
<< std::regex_match(file_name, re) << '\n';
```

자 그렇다면 만들어진 정규식 객체를 사용하는 부분을 살펴봅시다. `std::regex_match` 는 첫 번째 인자로 전달된 문자열 (위 경우 `file_name`) 이 두 번째 인자로 전달된 정규 표현식 객체 (위 경우 `re`) 와 완전히 매칭이 된다면 `true` 를 리턴합니다. 완전히 매칭 된다는 말은 문자열 전체가 정규 표현식의 패턴에 부합해야 한다는 것이지요.

부분 매칭 뽑아내기

앞서 `regex_match` 를 사용해서 전체 문자열이 주어진 정규 표현식 패턴을 만족하는지 알아 낼 수 있다고 하였습니다. 그렇다면 해당 조건을 만족하는 문자열에서 패턴 일부분 을 뽑아내고 싶다면 어떨까요?

예를 들어서 사용자로 부터 전화번호를 받는 정규 표현식을 생각해봅시다. 전화번호는 간단히 생각 해서 다음과 같은 규칙을 만족한다고 생각합니다.

- (숫자)-(숫자)-(숫자) 꼴로 있어야 합니다.
- 맨 앞자리는 반드시 3 자리이며 0 과 1 로만 이루어져 있어야 합니다.
- 가운데 자리는 3 자리 혹은 4 자리 여야 합니다.
- 마지막 자리는 4 자리 여야 합니다.

예를 들어 아래와 같은 번호들이 조건을 만족하겠죠.

```
010-1234-5678 010-123-4567 011-1234-5678
```

그렇다면 다음과 같이 정규 표현식을 작성할 수 있겠습니다.

```
[01]{3}-\d{3,4}-\d{4}
```

맨 앞에 `[01]` 의 뜻은 0 혹은 1 이라는 의미이고, 뒤에 `{3}` 은 해당 종류의 문자가 3 번 나타날 수 있다는 의미입니다.

```
#include <iostream>
#include <regex>
#include <vector>

int main() {
    std::vector<std::string> phone_numbers = {"010-1234-5678", "010-123-4567",
                                                "011-1234-5567", "010-12345-6789",
                                                "123-4567-8901", "010-1234-567"};
    std::regex re("[01]{3}-\\d{3,4}-\\d{4}");
    for (const auto &number : phone_numbers) {
        std::cout << number << ":" << std::boolalpha
            << std::regex_match(number, re) << '\n';
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
010-1234-5678: true
010-123-4567: true
011-1234-5567: true
010-12345-6789: false
123-4567-8901: false
010-1234-567: false
```

예상한 대로 잘 작동함을 알 수 있습니다.

만약에 조건에 만족하는 전화번호 중에서 가운데 번호를 추출하고 싶다면 어떨까요? 정규 표현식을 배운 분들은 아시겠지만, 캡쳐 그룹 (capture group) 을 사용하면 된다는 것을 알고 계실 것입니다. C++ 의 경우도 마찬가지입니다.

```
std::regex re("[01]{3}-(\\d{3,4})-\\d{4}");
```

위와 같이 () 로 원하는 부분을 감싸게 된다면 해당 부분에 매칭된 문자열을 얻을 수 있게 됩니다. 그렇다면 매칭된 부분을 어떻게 얻을 수 있는지 살펴보도록 합시다.

```
#include <iostream>
#include <regex>
#include <vector>

int main() {
    std::vector<std::string> phone_numbers = {"010-1234-5678", "010-123-4567",
```

```

    "011-1234-5567", "010-12345-6789",
    "123-4567-8901", "010-1234-567"};

std::regex re("[01]{3}-(\\d{3,4})-\\d{4}");
std::smatch match; // 매칭된 결과를 string 으로 보관
for (const auto &number : phone_numbers) {
    if (std::regex_match(number, match, re)) {
        for (size_t i = 0; i < match.size(); i++) {
            std::cout << "Match : " << match[i].str() << std::endl;
        }
        std::cout << "-----\n";
    }
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Match : 010-1234-5678
Match : 1234
-----
Match : 010-123-4567
Match : 123
-----
Match : 011-1234-5567
Match : 1234
-----
```

와 같이 나옵니다.

먼저 매칭된 문자열을 보관하는 객체를 정의한 부분부터 살펴봅시다.

```
std::smatch match;
```

위 smatch 의 경우 매칭된 문자열을 std::string 의 형태로 돌려줍니다. 그 외에도 const char* 로 돌려주는 cmatch 가 있습니다.

```
if (std::regex_match(number, match, re)) {
```

다음에 regex_match 에 매칭된 결과를 보관할 match 와 정규 표현식 re 를 모두 전달합니다. 만일 number 가 re 의 패턴에 부합하다면 match 에 매칭된 결과가 들어 있을 것입니다.

```
for (size_t i = 0; i < match.size(); i++) {
    std::cout << "Match : " << match[i].str() << std::endl;
}
```

자 그럼 이제 `match`에서 매칭된 문자열들을 `match[i].str()`을 통해 접근할 수 있습니다. 참고로 우리의 `match`가 `sregex`이므로 `match[i].str()`은 `std::string`이 됩니다. 반면에 `match`가 `cmatch`였다면 `match[i].str()`은 `const char*`이 되겠지요.

참고로 `regex_match`의 경우 전체 문자열이 매칭이 된 것이기 때문에 첫 번째 결과에 전체 문자열이 나타나게 됩니다. 그 다음으로 () 안에 들어있던 문자열이 나타나게 되죠. 만약에 정규 표현식 안에 () 가 여러개 있다면 마찬가지로 `for` 문을 통해 순차적으로 접근할 수 있습니다.

원하는 패턴 검색하기

앞서 `regex_match`를 통해 문자열 전체가 패턴에 부합하는지 확인하는 작업을 하였습니다. 이번에는 전체 말고 패턴을 만족하는 문자열 일부를 검색하는 작업을 수행해보도록 하겠습니다.

우리가 하고 싶은 일은 HTML 문서에서 아래와 같은 형태의 태그만 읽어들이는 것입니다.

```
<div class="sk...">...</div>
```

그렇다면 해당 조건을 만족하는 정규 표현식은 아래와 같이 작성할 수 있습니다.

```
<div class="sk[\w-]*">\w*</div>
```

자 그렇다면 해당 정규 표현식을 사용해서 문자열에서 원하는 패턴을 어떻게 검색하는지 살펴보겠습니다.

```
#include <iostream>
#include <regex>

int main() {
    std::string html = R"(

        <div class="social-login">
            <div class="small-comment">다음으로 로그인 </div>
            <div>
                <i class="xi-facebook-official fb-login"></i>
                <i class="xi-google-plus goog-login"></i>
            </div>
        </div>
        <div class="manual">
            <div class="small-comment">
                또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
            </div>
        </div>
    )";
}
```

```

        </div>
        <input name="name" id="name" placeholder="이름">
        <input name="password" id="password" type="password"
        ↵ placeholder="비밀번호">
        </div>
        <div id="adding-comment" class="sk-fading-circle">
            <div class="sk-circle1 sk-circle">a</div>
            <div class="sk-circle2 sk-circle">b</div>
            <div class="sk-circle3 sk-circle">asd</div>
            <div class="sk-circle4 sk-circle">asdfasf</div>
            <div class="sk-circle5 sk-circle">123</div>
            <div class="sk-circle6 sk-circle">aax</div>
            <div class="sk-circle7 sk-circle">sxz</div>
        </div>
    )";
}

std::regex re(R"(<div class="sk[\w -]*>\w*</div>)");
std::smatch match;
while (std::regex_search(html, match, re)) {
    std::cout << match.str() << '\n';
    html = match.suffix();
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

<div class="sk-circle1 sk-circle">a</div>
<div class="sk-circle2 sk-circle">b</div>
<div class="sk-circle3 sk-circle">asd</div>
<div class="sk-circle4 sk-circle">asdfasf</div>
<div class="sk-circle5 sk-circle">123</div>
<div class="sk-circle6 sk-circle">aax</div>
<div class="sk-circle7 sk-circle">sxz</div>

```

와 같이 잘 나옵니다.

```

while (std::regex_search(html, match, re)) {

```

문자열에서 원하는 패턴을 검색하는 일은 `regex_search`를 사용하면 됩니다. `regex_match`처럼, 첫 번째에 검색을 하고픈 문자열을, 두 번째에 일치된 패턴을 보관할 `match` 객체를, 마지막 인자로 실제 정규 표현식 객체를 전달하면 됩니다. 만일 `html`에서 정규 표현식과 매칭이 되는 패턴이 존재한다면 `regex_search`가 `true`를 리턴하게 되지요.

```
std::cout << match.str() << '\n';
```

그리고 매칭된 패턴은 위와 같이 `match.str()` 을 통해서 접근할 수 있습니다. 우리의 `match` 가 `smatch` 의 객체 이므로 만들어진 `match.str()` 은 `string` 이 됩니다.

문제는 만일 그냥 `std::regex_search(html, match, re)` 를 다시 실행하게 된다면 그냥 이전에 찾았던 패턴을 다시 뱉을 것입니다. 따라서 우리는 `html` 을 업데이트 해서 검색된 패턴 바로 뒤 부터 다시 검색할 수 있도록 바꿔야합니다.

```
html = match.suffix();
```

`match.suffix()` 를 하면 `std::sub_match` 객체를 리턴합니다. `sub_match` 는 단순히 어떠한 문자열의 시작과 끝을 가리키는 반복자 두 개를 가지고 있다고 보시면 됩니다. 이 때 `suffix` 의 경우, 원 문자열에서 검색된 패턴 바로 뒤부터, 이전 문자열의 끝 까지에 해당하는 `sub_match` 객체를 리턴합니다.

예를 들어서 정규 표현식이 "is" 라면

match.str()

My name **is** Jaebum Lee

match.prefix() match.suffix()

이 때 `sub_match` 클래스에는 `string` 으로 변환할 수 있는 캐스팅 연산자가 들어 있습니다. 따라서 위와 같이 `html` 에 그냥 대입하게 되면 알아서 문자열로 변환되어서 들어가게 됩니다. 덕분에 이미 찾은 패턴 뒤부터 다시 새로운 검색을 시작할 수 있겠죠.

std::regex_iterator

`regex_iterator` 를 사용하면 좀 더 편리하게 검색을 수행할 수 있습니다. 예를 들어서;

```
#include <iostream>
#include <regex>

int main() {
    std::string html = R"(

다음으로 로그인 </div>


```

```

        <i class="xi-facebook-official fb-login"></i>
        <i class="xi-google-plus goog-login"></i>
    </div>
</div>
<div class="manual">
    <div class="small-comment">
        또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
    </div>
    <input name="name" id="name" placeholder="이름">
    <input name="password" id="password" type="password"
    ↳ placeholder="비밀번호">
    </div>
    <div id="adding-comment" class="sk-fading-circle">
        <div class="sk-circle1 sk-circle">a</div>
        <div class="sk-circle2 sk-circle">b</div>
        <div class="sk-circle3 sk-circle">asd</div>
        <div class="sk-circle4 sk-circle">asdfasf</div>
        <div class="sk-circle5 sk-circle">123</div>
        <div class="sk-circle6 sk-circle">aax</div>
        <div class="sk-circle7 sk-circle">sxz</div>
    </div>
)>;

```

```

std::regex re(R"(<div class="sk[\w -]*>\w*</div>)");
std::smatch match;

auto start = std::sregex_iterator(html.begin(), html.end(), re);
auto end = std::sregex_iterator();

while (start != end) {
    std::cout << start->str() << std::endl;
    ++start;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

<div class="sk-circle1 sk-circle">a</div>
<div class="sk-circle2 sk-circle">b</div>
<div class="sk-circle3 sk-circle">asd</div>
<div class="sk-circle4 sk-circle">asdfasf</div>
<div class="sk-circle5 sk-circle">123</div>
<div class="sk-circle6 sk-circle">aax</div>
<div class="sk-circle7 sk-circle">sxz</div>

```

와 같이 잘 나옵니다.

`std::regex_iterator` 는 주어진 문자열에서 정규 표현식으로 매칭된 문자열들을 차례로 뽑아낼 수 있는 반복자입니다.

```
auto start = std::sregex_iterator(html.begin(), html.end(), re);
```

`regex_iterator` 의 경우 위와 같이 생성자를 호출함으로써 생성할 수 있습니다. 첫 번째와 두 번째 인자로 검색을 수행할 문자열의 시작과 끝을 전달하고, 마지막 인자로 사용하고픈 정규 표현식 객체를 전달하면 됩니다. 참고로 `sregex_iterator` 는 `regex_iterator` 중에서 `string` 을 사용하는 반복자입니다.

`regex_iterator` 의 경우 처음 생성될 때와, `++` 연산자로 증감 될 때마다 `regex_search` 를 통해 초기에 주어진 문자열 범위 내에서 패턴에 맞는 문자열을 검색합니다. 또한 `*` 연산자를 통해서 역참조 하게 된다면 현재 검색된 패턴에 대한 `match_results` 객체를 얻어낼 수 있습니다.

따라서 아래와 같이 간단하게 패턴과 매칭되는 문자열들을 뽑아낼 수 있게 됩니다.

```
while (start != end) {
    std::cout << start->str() << std::endl;
    ++start;
}
```

원하는 패턴 치환하기

마지막으로 살펴볼 기능은 정규 표현식을 통해서 원하는 패턴의 문자열을 다른 문자열로 치환(replace)하는 작업입니다. 해당 작업은 `std::regex_replace` 를 통해서 구현할 수 있습니다.

예를 들어서 `html` 에서 `sk-circle1` 과 같은 문자열을 `1-sk-circle` 로 바꿔보는 정규 표현식을 생각해봅시다. 이를 위해서 먼저 `sk-circle1` 과 같은 형태를 어떤 `regex` 로 매칭 시킬지 생각해야 하는데 이는 간단히

```
sk-circle\d
```

로 할 수 있습니다. 그 다음에, 이를 어떠한 형태로 치환할지를 생각해야 합니다. 간단히 생각해서

```
숫자-sk-circle
```

로 해야 하는데, 문제는 위 숫자에 해당하는 부분이 매칭된 패턴에 `\d`에 해당하는 부분이라는 점입니다. 하지만 다행이도, 캡쳐 그룹을 이용하면 이를 간단히 해결할 수 있습니다. 먼저

```
sk-circle(\d)
```

를 통해서 숫자에 해당하는 부분을 첫 번째 캡쳐그룹으로 만듭니다. 그 다음에, 치환을 할 때 첫 번째 캡쳐 그룹을 표현하기 위해 \$1 라고 명시할 수 있습니다. (이와 같은 요소를 *back reference*라고 부릅니다.) 따라서

```
$1-sk-circle
```

과 같이 표현할 수 있습니다.

이를 바탕으로 C++ 코드를 작성해봅시다.

```
#include <iostream>
#include <regex>

int main() {
    std::string html = R"
        <div class="social-login">
            <div class="small-comment">다음으로 로그인 </div>
            <div>
                <i class="xi-facebook-official fb-login"></i>
                <i class="xi-google-plus goog-login"></i>
            </div>
        </div>
        <div class="manual">
            <div class="small-comment">
                또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
            </div>
            <input name="name" id="name" placeholder="이름">
            <input name="password" id="password" type="password"
        ↵   placeholder="비밀번호">
            </div>
            <div id="adding-comment" class="sk-fading-circle">
                <div class="sk-circle1 sk-circle">a</div>
                <div class="sk-circle2 sk-circle">b</div>
                <div class="sk-circle3 sk-circle">asd</div>
                <div class="sk-circle4 sk-circle">asdfasf</div>
                <div class="sk-circle5 sk-circle">123</div>
                <div class="sk-circle6 sk-circle">aax</div>
                <div class="sk-circle7 sk-circle">sxz</div>
            </div>
        )";
    std::regex re(R"r(sk-circle(\d))r");
    std::smatch match;

    std::string modified_html = std::regex_replace(html, re, "$1-sk-circle");
    std::cout << modified_html;
```

{}

성공적으로 컴파일 하였다면

실행 결과

```
<div class="social-login">
  <div class="small-comment">다음으로 로그인 </div>
  <div>
    <i class="xi-facebook-official fb-login"></i>
    <i class="xi-google-plus goog-login"></i>
  </div>
</div>
<div class="manual">
  <div class="small-comment">
    또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
  </div>
  <input name="name" id="name" placeholder="이름">
  <input name="password" id="password" type="password"
    ↴ placeholder="비밀번호">
</div>
<div id="adding-comment" class="sk-fading-circle">
  <div class="1-sk-circle sk-circle">a</div>
  <div class="2-sk-circle sk-circle">b</div>
  <div class="3-sk-circle sk-circle">asd</div>
  <div class="4-sk-circle sk-circle">asdfasf</div>
  <div class="5-sk-circle sk-circle">123</div>
  <div class="6-sk-circle sk-circle">aax</div>
  <div class="7-sk-circle sk-circle">sxz</div>
</div>
```

와 같이 잘 나옵니다.

```
std::regex_replace(html, re, "$1-sk-circle");
```

`regex_replace`로 문자열을 치환하고 싶다면 첫 번째 인자로 치환하고자 하는 문자열을, 두 번째 인자로 정규 표현식 객체를, 마지막으로 치환 시에 어떠한 패턴으로 바꿀 지 적어주면 됩니다. `regex_replace`의 오버로딩 형태는 여러가지가 있는데 위 형태의 경우 치환된 문자열을 생성해서 돌려줍니다.

만약에 치환된 문자열을 생성하지 않고 그냥 `stdout`에 출력하고 싶다면;

```
std::regex_replace(std::ostreambuf_iterator<char>(std::cout), html.begin(),
                   html.end(), re, "$1-sk-circle");
```

와 같이 쓰면 됩니다. 이 경우 `std::regex_replace`의 첫 번째 인자로 출력할 위치의 시작점을 가리키는 반복자를 넣어주면 됩니다.

중첩된 캡쳐 그룹

만약에 더 나아가서 다음과 같이 치환을 하고 싶다 해봅시다.

```
<div class="sk-circle1 sk-circle">a</div>
```

에서

```
<div class="1-sk-circle">a</div>
```

로 말이지요. 즉 뒷 부분의 `sk-circle` 을 완전히 날려버리는 것입니다. 이를 위해서 두 개의 캡쳐그룹이 필요합니다. 일단, "" 안에 전체 문자열을 건드려야 하기 때문에

```
(sk-circle\d sk-circle)
```

과 같이 전체 패턴에 해당하는 캡쳐 그룹이 필요하지요. 그 다음에, 숫자 부분만 다시 뽑아내야 하므로

```
(sk-circle(\d) sk-circle)
```

와 같이 해야 됩니다. 문제는 위와 같이 캡쳐 그룹이 중첩되었을 때 어느 것이 \$1이고 \$2인지 알아야 하는데, 괄호가 열리는 순서대로 \$1, \$2, ... 로 진행한다고 생각하면 됩니다. 즉 해당 패턴 전체 캡쳐 그룹이 \$1이고 \d에 해당하는 캡쳐 그룹이 \$2가 되죠.

따라서 치환될 패턴은

```
$2-sk-circle
```

이 됨을 알 수 있겠네요. 실제로 코드를 작성해보면

```
#include <iostream>
#include <regex>
```

```
int main() {
    std::string html = R"
        <div class='social-login'>
            <div class='small-comment'>다음으로 로그인 </div>
            <div>
                <i class='xi-facebook-official fb-login'></i>
                <i class='xi-google-plus goog-login'></i>
            </div>
        </div>
        <div class='manual'>
            <div class='small-comment'>
                또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
            </div>
            <input name='name' id='name' placeholder='이름'>
            <input name='password' id='password' type='password'
        ↪   placeholder='비밀번호'>
        </div>
        <div id='adding-comment' class='sk-fading-circle'>
            <div class='sk-circle1 sk-circle'>a</div>
            <div class='sk-circle2 sk-circle'>b</div>
            <div class='sk-circle3 sk-circle'>asd</div>
            <div class='sk-circle4 sk-circle'>asdfasf</div>
            <div class='sk-circle5 sk-circle'>123</div>
            <div class='sk-circle6 sk-circle'>aax</div>
            <div class='sk-circle7 sk-circle'>sxz</div>
        </div>
    )";
    std::regex re(R"r((sk-circle(\d) sk-circle))r");
    std::smatch match;

    std::string modified_html = std::regex_replace(html, re, "$2-sk-circle");
    std::cout << modified_html;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
<div class='social-login'>
    <div class='small-comment'>다음으로 로그인 </div>
    <div>
        <i class='xi-facebook-official fb-login'></i>
        <i class='xi-google-plus goog-login'></i>
    </div>
</div>
```

```
<div class="manual">
  <div class="small-comment">
    또는 직접 입력하세요 (댓글 수정시 비밀번호가 필요합니다)
  </div>
  <input name="name" id="name" placeholder="이름">
  <input name="password" id="password" type="password"
    ↳ placeholder="비밀번호">
</div>
<div id="adding-comment" class="sk-fading-circle">
  <div class="1-sk-circle">a</div>
  <div class="2-sk-circle">b</div>
  <div class="3-sk-circle">asd</div>
  <div class="4-sk-circle">asdfasf</div>
  <div class="5-sk-circle">123</div>
  <div class="6-sk-circle">aax</div>
  <div class="7-sk-circle">sxz</div>
</div>
```

와 같이 잘 바뀌었음을 알 수 있습니다.

자 그럼 이것으로 C++ 정규 표현식 라이브러리를 어떻게 사용하는지 간단하게 알아보았습니다. 다음 강좌에서는 C++ 11에서 새롭게 추가된 난수 생성 라이브러리 (`<random>`) 과 시간 관련 라이브러리 (`<chrono>`) 를 살펴보도록 하겠습니다.

뭘 배웠지?

- `std::regex` 를 통해서 정규 표현식 객체를 생성할 수 있습니다.
- `std::regex_match` 로 전체 문자열이 패턴을 만족하는지 확인할 수 있습니다.
- `std::regex_search` 를 통해 패턴에 맞는 문자열을 검색할 수 있습니다.
- `std::regex_replace` 를 통해 원하는 패턴의 문자열을 다른 문자열로 치환할 수 있습니다.

난수 생성(<random>)과 시간 관련 라이브러리(<chrono>) 소개

안녕하세요 여러분! 이번 강좌에서는 C++ 11 에 추가된 난수(Random number)를 쉽게 생성할 수 있도록 도와주는 <random> 라이브러리와 시간 관련 데이터를 다룰 수 있게 도와주는 <chrono> 라이브러리를 살펴보도록 하겠습니다.

아무래도 C 언어를 먼저 접한 분들은 C++ 에서도 난수 생성이나 날짜 관련 계산을 위해 C 라이브러리 (`time.h`이나 `stdlib.h`) 를 사용하는 경우가 종종 있는데 이번 기회에 왜 C++ 라이브러리를 사용해야만 하는지 짚고 넘어가도록 할 것입니다.

C 스타일의 난수 생성의 문제점

아래는 C 스타일로 0 부터 99 까지의 난수를 생성하는 코드입니다.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));

    for (int i = 0; i < 5; i++) {
        printf("난수 : %d \n", rand() % 100);
    }
    return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
난수 : 75
난수 : 95
난수 : 20
난수 : 47
난수 : 41
```

와 같이 나옵니다. 참고로 위 코드는 엄밀히 말하자면 진짜 난수를 생성하는 것이 아니라 마치 난수처럼 보이는 의사 난수 (pseudo random number) 을 생성하는 코드입니다. 컴퓨터 상에서 완전한 무작위로 난수를 생성하는 것은 생각보다 어렵습니다. 그 대신에, 첫 번째 수 만 무작위로 정하고, 나머지 수들은 그 수를 기반으로 여러가지 수학적 방법을 통해서 마치 난수처럼 보이지만 실제로는 무작위로 생성된 것이 아닌 수열들을 만들어내게 됩니다.

무작위로 정해진 첫 번째 수를 시드(seed) 라고 부르는데, C 의 경우 `srand` 를 통해 `seed` 를 설정할 수 있습니다. 우리의 경우 `time(NULL)` 을 통해 프로그램을 실행했던 초를 시드값으로 지정하였습니다. 그리고 `rand()` 는 호출 할 때마다 시드값을 기반으로 무작위처럼 보이는 수열을 생성하게 되죠.

하지만 위 코드는 여러가지 문제점들이 있습니다.

시드값이 너무 천천히 변한다.

보시다시피 시드값으로 현재의 초를 지정하였습니다. 이 말은 즉슨 같은 시간대에 시작된 프로그램의 경우 모두 같은 의사 난수 수열을 생성한다는 점입니다. 만일 여러가지 프로그램들이 같이 돌아가는 시스템에서 위 코드를 사용하였다면 아마 같은 난수열을 생성하는 프로그램이 생기게 될 것입니다.

0부터 99 까지 균등하게 난수를 생성하지 않는다.

위 코드에서 우리가 0 부터 99 까지 난수를 생성하기 위해서

```
printf("난수 : %d \n", rand() % 100);
```

와 같이 하였습니다. 문제는 `rand()` 가 리턴하는 값이 0 부터 `RAND_MAX` 라는 점입니다. 물론 `rand()` 가 0 부터 `RAND_MAX` 까지의 모든 값을 같은 확률로 난수를 생성하지만, 100 으로 나눈 나머지는 꼭 그러라는 법이 없습니다. 예를 들어서 `RAND_MAX` 가 128 이라고 합시다. 그렇다면 1 의 경우 `rand()` 가 리턴한 값이 1 이거나 101 일 때 생성되지만 50 의 경우 `rand()` 가 리턴한 값이 50 일 때만 생성됩니다. 따라서 1 이 뽑힐 확률이 50 이 뽑힐 확률 보다 2 배나 높게 됩니다.

`rand()` 자체도 별로 뛰어나지 않다.

무엇보다도 C 의 `rand()` 함수는 **선형 합동 생성기** (Linear congruential generator) 이라는 알고리즘을 바탕으로 구현되어 있는데 이 알고리즘은 썩 좋은 품질의 난수열을 생성하지 못합니다. 더 깊게 설명하자면 생성되는 난수열들의 상관 관계가 높아서 일부 시뮬레이션에 사용하기에 적합하지 않습니다.

결론적으로 말하자면

주의 사항

C++ 에서는 C 의 srand 와 rand 는 갖다 버리자!

<random>

먼저 위 코드 처럼 0 부터 99 까지의 난수를 생성하는 코드를 C++ 의 <random> 라이브러리를 사용해서 어떻게 작성하는지 살펴보도록 하겠습니다.

```
#include <iostream>
#include <random>

int main() {
    // 시드값을 얻기 위한 random_device 생성.
    std::random_device rd;

    // random_device 를 통해 난수 생성 엔진을 초기화 한다.
    std::mt19937 gen(rd());

    // 0 부터 99 까지 균등하게 나타나는 난수열을 생성하기 위해 균등 분포 정의.
    std::uniform_int_distribution<int> dis(0, 99);

    for (int i = 0; i < 5; i++) {
        std::cout << "난수 : " << dis(gen) << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
난수 : 77
난수 : 11
난수 : 45
난수 : 72
난수 : 3
```

자 그렇다면 위 코드를 하나 하나씩 살펴보도록 하겠습니다.

```
// 시드값을 얻기 위한 random_device 생성.
std::random_device rd;
```

앞서 C의 경우 `time(NULL)`을 통해서 시드값을 지정하였지만 이는 여러가지 문제점이 있었습니다. C++에서는 좀 더 양질의 시드값을 얻기 위해 `random_device`라는 것을 제공합니다.

대부분의 운영체제에는 진짜 난수값들을 얻어낼 수 있는 여러가지 방식들을 제공하고 있습니다. 예를 들어서 리눅스의 경우 `/dev/random`나 `/dev/urandom`을 통해서 난수값을 얻을 수 있습니다. 이 난수값은, 이전에 우리가 이야기 하였던 무슨 수학적 알고리즘을 통해 생성되는 가짜 난수가 아니라 정말로 컴퓨터가 실행하면서 마주치는 무작위적인 요소들 (예를 들어 장치 드라이버들의 noise)을 기반으로한 진정한 난수를 제공합니다.

`random_device`를 이용하면 운영체제 단에서 제공하는 진짜 난수를 사용할 수 있습니다. 다만 진짜 난수의 경우 컴퓨터가 주변의 환경과 무작위적으로 상호작용하면서 만들어지는 것이기 때문에 의사 난수보다 난수를 생성하는 속도가 매우 느립니다. 따라서 시드값처럼 난수 엔진을 초기화하는데 사용하고, 그 이후의 난수열은 난수 엔진으로 생성하는 것이 적합합니다.

```
// random_device 를 통해 난수 생성 엔진을 초기화 한다.
std::mt19937 gen(rd());
```

위와 같이 생성한 `random_device` 객체를 이용해서 난수 생성 엔진 객체를 정의할 수 있습니다. 만약에 `random_device` 대신에 그냥 여러분이 원하는 값을 시드값으로 넣어주고 싶다면 그냥

```
std::mt19937 gen(1234);
```

와 같이 해도 됩니다.

`std::mt19937`은 C++ <`random`> 라이브러리에서 제공하는 난수 생성 엔진 중 하나로, **메르센 트위스터**라는 알고리즘을 사용합니다. 이 알고리즘은 기존에 `rand` 가 사용하였던 선형 합동 방식보다 좀 더 양질의 난수열을 생성한다고 알려져있습니다. 무엇보다도 생성되는 난수들 간의 상관관계가 매우 작기 때문에 여러 시뮬레이션에서 사용할 수 있습니다.

참고적으로 <`random`> 라이브러리에는 위 메르센 트위스터 기반 엔진 말고도 기존의 `rand` 와 같이 선형 합동 알고리즘을 사용한 `minstd_rand` 외 여러가지 엔진들이 정의되어 있습니다. 물론 `mt19937`이 훌륭한 난수를 생성하기에는 적합하지만 생각보다 객체 크기가 커서 (2KB 이상) 메모리가 부족한 시스템에서는 오히려 `minstd_rand` 가 적합할 수 있습니다.⁴⁾

이처럼 난수 생성 엔진을 만들었지만 아직 바로 난수를 생성할 수 있는 것은 아닙니다. C++의 경우 어디에서 수들을 뽑아낼지 알려주는 **분포(distribution)**을 정의해야 합니다.

앞서 우리의 경우 0부터 99까지 균등한 확률로 정수를 뽑아내고 싶다고 하였습니다. 따라서 이를 위해선 아래와 같이 균등 분포 (Uniform distribution) 객체를 정의해야 합니다.

4) `mt19937`를 생성한 이후에 난수를 생성하는 작업은 매우 빠릅니다.

```
// 0 부터 99 까지 균등하게 나타나는 난수열을 생성하기 위해 균등 분포 정의.
std::uniform_int_distribution<int> dis(0, 99);
```

위와 같이 `uniform_int_distribution<int>` 의 생성자에 원하는 범위를 써넣으면 됩니다.

```
for (int i = 0; i < 5; i++) {
    std::cout << "난수 : " << dis(gen) << std::endl;
}
```

그리고 마지막으로 균등 분포에 사용할 난수 엔진을 전달함으로써 균등 분포에서 무작위로 샘플을 뽑아낼 수 있습니다.

`<random>` 라이브러리에서는 균등 분포 말고도 여러가지 분포들을 제공하고 있습니다. 여기서는 다 일일히 소개하기 어렵지만 그 중 가장 많이 쓰이는 정규 분포 (Normal distribution) 만 간단히 살펴보겠습니다. (전체 목록은 [여기](#)서 보세요.)

```
#include <iomanip>
#include <iostream>
#include <map>
#include <random>

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<double> dist(/* 평균 = */ 0, /* 표준 편차 = */ 1);

    std::map<int, int> hist{};
    for (int n = 0; n < 10000; ++n) {
        ++hist[std::round(dist(gen))];
    }
    for (auto p : hist) {
        std::cout << std::setw(2) << p.first << ' '
              << std::string(p.second / 100, '*') << " " << p.second << '\n';
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
-4   1
-3   38
-2 ***** 638
```

```
-1 **** 2407
0 ***** 3821
1 ***** 2429
2 *** 595
3 70
4 1
```

코드를 보면 간단합니다.

```
std::normal_distribution<double> dist(/* 평균 = */ 0, /* 표준 편차 = */ 1);
```

이번에는 평균 0이고 표준 편차가 1인 정규 분포를 정의하였고,

```
for (int n = 0; n < 10000; ++n) {
    ++hist[std::round(dist(gen))];
}
```

이를 바탕으로 위와 같이 정규 분포에서 10000 개의 샘플을 무작위로 뽑아내게 됩니다. 실제로 위 그림처럼 아름다운 정규 분포 곡선이 나옴을 확인할 수 있습니다.

자 그럼 이것으로 `random` 라이브러리 사용법을 간단히 알아보았습니다.

이번에는 C++ 11 에 같이 추가된 시간 관련 데이터를 쉽게 계산할 수 있도록 도와주는 `chrono` 라이브러리를 살펴보도록 하겠습니다.

chrono 소개

`<chrono>` 는 크게 아래와 같이 3 가지 요소들로 구성되어 있습니다.

- 현재 시간을 알려주는 시계 - 예를 들어서 `system_clock`
- 특정 시간을 나타내는 `time_stamp`
- 시간의 간격을 나타내는 `duration`

로 말이지요.

chrono에서 지원하는 clock들

<chrono>에서는 여러가지 종류의 시계들을 지원하고 있습니다. 예를 들어서 일반적 상황에서 현재 컴퓨터 상 시간을 얻어 오기 위해서는 `std::system_clock` 을 사용하면 되고, 좀 더 정밀한 시간 계산이 필요한 경우 (예를 들어 프로그램 성능을 측정하고 싶을 때) `std::high_resolution_clock` 을 사용하시면 됩니다.

이들 객체의 이름이 시계이기는 하지만 실제 시계 처럼 지금 딱 몇 시 이렇게 이야기 해주는 것이 아닙니다. 그 대신에, 지정된 시점으로 부터 몇 번의 틱(tick)이 발생 하였는지 알려주는 `time_stamp` 객체를 리턴합니다. 예를 들어서 `std::system_clock` 의 경우 1970년 1월 1일 부터 현재 까지 발생한 틱의 횟수를 리턴한다고 보시면 됩니다.⁵⁾ 쉽게 말해 `time_stamp` 객체는 `clock` 의 시작점과 현재 시간의 `duration` 을 보관하는 객체입니다.

여기서 틱이라고 하면 시계의 초침이 한 번 똑딱 거리는 것이라 생각하면 됩니다. 컴퓨터의 경우도 내부에 시계가 있어서 특정 진동수로 똑딱 거리게 됩니다.⁶⁾

각 시계마다 정밀도가 다르기 때문에 각 `clock` 에서 얻어지는 `tick` 값 자체는 조금씩 다릅니다. 예를 들어서 `system_clock` 이 1초에 1 tick 이라면, `high_resolution_clock` 의 경우 0.00000001초마다 1 tick 움직일 수 있습니다.

자 그렇다면 실제로 chrono 라이브러리를 사용하는 코드를 살펴보도록 하겠습니다. 아래 코드는 난수를 생성 속도를 측정합니다.

```
#include <chrono>
#include <iomanip>
#include <iostream>
#include <random>
#include <vector>

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<> dist(0, 1000);

    for (int total = 1; total <= 1000000; total *= 10) {
        std::vector<int> random_numbers;
        random_numbers.reserve(total);

        std::chrono::time_point<std::chrono::high_resolution_clock> start =
            std::chrono::high_resolution_clock::now();

        for (int i = 0; i < total; i++) {
            random_numbers.push_back(dist(gen));
        }
    }
}
```

5) 이를 흔히 유닉스 타임(Unix time) 이라 부릅니다.

6) 우리나라는 시계가 똑딱 거린다고 하지만 미국에서는 tick-tock 이라 하죠

```

    }

    std::chrono::time_point<std::chrono::high_resolution_clock> end =
        std::chrono::high_resolution_clock::now();

    // C++ 17 이전
    auto diff = end - start;

    // C++ 17 이후
    // std::chrono::duration diff = end - start;

    std::cout << std::setw(7) << total
        << "개 난수 생성 시 틱 횟수 : " << diff.count() << std::endl;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

1개 난수 생성 시 틱 횟수 : 535
10개 난수 생성 시 틱 횟수 : 1370
100개 난수 생성 시 틱 횟수 : 11354
1000개 난수 생성 시 틱 횟수 : 110219
10000개 난수 생성 시 틱 횟수 : 1145811
100000개 난수 생성 시 틱 횟수 : 11040923
1000000개 난수 생성 시 틱 횟수 : 99170277

```

와 같이 나옵니다.

```

std::chrono::time_point<std::chrono::high_resolution_clock> start =
    std::chrono::high_resolution_clock::now();

```

먼저 `high_resolution_clock` 으로 부터 현재의 `time_point`를 얻어오는 코드부터 살펴봅시다. `chrono` 라이브러리의 경우 다른 표준 라이브러리와는 다르게 객체들이 `std::chrono` 이름 공간 안에 정의되어 있습니다. 따라서 `high_resolution_clock`를 쓰기 위해서는 `std::high_resolution_clock` 가 아니라 `std::chrono::high_resolution_clock` 와 같이 적어야 합니다.

만약에 매번 `std::chrono` 를 쓰기에 번거롭다면 그냥

```

namespace ch = std::chrono;

```

와 같이 `ch`라는 별명을 지어주고 `ch`로 대체하면 됩니다.

이들 `clock`에는 현재의 `time_point`를 리턴하는 `static` 함수인 `now`가 정의되어 있습니다. 이 `now()`를 호출하면 위와 같이 해당 `clock`에 맞는 `time_point` 객체를 리턴합니다. 우리의 경우 `high_resolution_clock::now()`를 호출하였으므로, `std::chrono::time_point<ch::high_resolution_clock>`를 리턴합니다.

`time_point`가 `clock`을 왜 템플릿 인자로 가지는지는 앞서도 설명하였듯이 `clock`마다 1초에 발생하는 틱 횟수가 모두 다르기 때문에 나중에 실제 시간으로 변환 시에 어떤 `clock`을 사용했는지에 대한 정보가 필요하기 때문입니다.

```
auto diff = end - start;
```

자 이제 난수 생성이 끝나면 `end`에 끝나는 시간을 또 받아서 그 차이를 계산해야 합니다. 위와 같이 두 `time_stamp`를 빼게 된다면 `duration` 객체를 리턴합니다.

주의 사항

참고로 C++ 17 이전에서는 `end - start` 가 리턴하는 `duration` 객체의 템플릿 인자를 전달해야 합니다. 따라서 굳이 `duration`의 템플릿 인자들을 지정하기 보다는 속시원하게 그냥 `auto diff = end - start`로 하는게 낫습니다.

```
std::cout << std::setw(7) << total
    << "개 난수 생성 시 틱 횟수 : " << diff.count() << std::endl;
```

`duration`에는 `count`라는 멤버 함수가 정의되어 있는데 이는 해당 시간 차이 동안 몇 번의 틱이 발생하였는지를 알려줍니다. 하지만 우리에게 좀 더 의미 있는 정보는 틱이 아니라 실제 시간으로 얼마나 걸렸는지 알아내는 것이지요. 이를 위해선 `duration_cast`를 사용해야 합니다.

```
#include <chrono>
#include <iomanip>
#include <iostream>
#include <random>
#include <vector>

namespace ch = std::chrono;

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<> dist(0, 1000);

    for (int total = 1; total <= 1000000; total *= 10) {
```

```

std::vector<int> random_numbers;
random_numbers.reserve(total);

ch::time_point<ch::high_resolution_clock> start =
    ch::high_resolution_clock::now();

for (int i = 0; i < total; i++) {
    random_numbers.push_back(dist(gen));
}

ch::time_point<ch::high_resolution_clock> end =
    ch::high_resolution_clock::now();

auto diff = end - start;
std::cout << std::setw(7) << total << "개 난수 생성 시 걸리는 시간: "
    << ch::duration_cast<ch::microseconds>(diff).count() << "us"
    << std::endl;
}
}

```

성공적으로 컴파일 하였다면

실행 결과

```

1개 난수 생성 시 걸리는 시간: 0us
10개 난수 생성 시 걸리는 시간: 1us
100개 난수 생성 시 걸리는 시간: 10us
1000개 난수 생성 시 걸리는 시간: 101us
10000개 난수 생성 시 걸리는 시간: 1033us
100000개 난수 생성 시 걸리는 시간: 10702us
1000000개 난수 생성 시 걸리는 시간: 98950us

```

와 같이 나옵니다.

```
ch::duration_cast<ch::microseconds>(diff).count()
```

`duration_cast` 는 임의의 `duration` 객체를 받아서 우리가 원하는 `duration` 으로 캐스팅 할 수 있습니다. `std::chrono::microseconds` 는 `<chrono>` 에 미리 정의되어 있는 `duration` 객체 중 하나로, 1 초에 10^6 번 톱을 하게 됩니다. 따라서 `microseconds` 로 캐스팅 한뒤에 리턴하는 `count` 값은 해당 `duration` 이 몇 마이크로초 인지를 나타내는 것인지요.

우리의 경우 1000000 개의 난수를 생성하는데 불과 98950 마이크로초, 대량 98 밀리초 정도 걸린다고 나왔습니다. `<chrono>` 에는 `std::chrono::microseconds` 외에도 `nanoseconds`,

`milliseconds`, `seconds`, `minutes`, `hours` 가 정의되어 있기 때문에 상황에 맞게 사용하시면 됩니다.

현재 시간을 날짜로

안타깝게도 C++ 17 까지에서는 `chrono` 라이브러리 상에서 날짜를 간단하게 다를 수 있도록 도와주는 클래스가 없습니다. 예를 들어서 현재 시간을 출력하고 싶다면 C 의 함수들에 의존해야 합니다.

예를 들어서 현재 시간을 출력하는 코드를 살펴봅시다.

```
#include <chrono>
#include <ctime>
#include <iomanip>
#include <iostream>

int main() {
    auto now = std::chrono::system_clock::now();
    std::time_t t = std::chrono::system_clock::to_time_t(now);
    std::cout << "현재 시간은 : " << std::put_time(std::localtime(&t), "%F %T %z")
          << '\n';
}
```

성공적으로 컴파일 하였다면

실행 결과

현재 시간은 : 2020-01-06 00:28:08 +0900

먼저 `system_clock` 에서 현재의 `time_point` 를 얻어온 후에, 날짜를 출력하기 위해서 `time_t` 객체로 변환해야 합니다.

```
std::time_t t = std::chrono::system_clock::to_time_t(now);
```

이를 위해 위와 같이 각 `clock` 이 제공하는 `static` 함수인 `to_time_t` 를 사용하면 됩니다.

```
std::cout << "현재 시간은 : " << std::put_time(std::localtime(&t), "%F %T %z")
          << '\n';
```

그 다음에 현재 시간을 `std::tm` 객체로 변환하기 위해서 `std::localtime` 에 `t` 를 전달하였고, 마지막으로 `std::put_time` 을 사용해서 우리가 원하는 형태의 문자열로 구성할 수 있게 됩니다. 참고로 `put_time` 에 전달된 인자인 "%F %T %z" 는 `strftime` 에서 사용되는 인자와 동일합

니다. 따라서 %F 와 같은 것들이 무엇을 수행하는지 알고 싶다면 해당 함수 레퍼런스를 참고하시기 바랍니다.

안타깝게도 C++ 17 현재 C 의 함수들을 이용하지 않고서 날짜를 다룰 수 있는 방법은 없습니다. 하지만 C++ 20 부터 <chrono> 에 C 라이브러리 필요 없이 날짜를 다룰 수 있는 클래스와 함수들이 추가된다고 하니 조금만 기다려주시기 바랍니다!

그렇다면 이번 강좌는 여기에서 마치도록 하겠습니다. 다음 강좌에서는 C++ 17 에서 등장한 <filesystem> 라이브러리를 살펴볼 것입니다.

뭘 배웠지?

- 난수를 생성하기 위해서 C 의 `srand` 와 `rand` 를 사용하지 말자.
- 대부분의 상황에서는 `std::mt19937` 로 충분히 양질의 난수를 뽑아낼 수 있다. 특히 `<random>` 라이브러리를 사용할 경우 원하는 확률 분포에서 샘플링할 수 있다.
- 현재 시간을 알기 위해서는 `system_clock` 을 사용하면 되고, 좀 더 정밀한 측정이 필요할 경우 `high_resolution_clock` 을 사용하면 된다.
- `duration_cast` 를 이용해서 원하는 시간 스케일로 변환할 수 있다.

C++ 파일 시스템(<filesystem>) 라이브러리 소개

안녕하세요 여러분! 이번 강좌에서는 C++ 17 에 비로소 도입된 파일 시스템 라이브러리 (<filesystem>)를 간단히 소개하는 시간을 가지도록 하겠습니다.

파일 시스템 라이브러리는 파일 데이터의 입출력을 담당하는 파일 입출력 라이브러리 (<fstream>)과는 다릅니다. <fstream> 의 경우, 파일 하나가 주어지면 해당 파일의 데이터를 읽어내는 역할을 합니다만 그 외에 파일에 관한 정보 (파일 이름, 위치, 등등)에 관한 데이터를 수정할 수는 없습니다. 반면에 파일 시스템 라이브러리의 경우, 파일에 관한 정보 (파일 메타데이터)에 대한 접근을 도와주는 역할을 수행하며, 파일 자체를 읽는 일은 수행하지 않습니다.

쉽게 말해서 하드 디스크 어딘가에 있는 `a.txt`라는 파일을 찾고 싶다면 `filesystem` 라이브러리를 사용하게 되고, 해당 파일을 찾은 이후에 `a.txt`를 읽고 싶다면 `fstream` 라이브러리를 사용하면 되겠습니다.

뿐만 아니라 파일 시스템 라이브러리를 통해서 원하는 경로에 폴더를 추가한다던지, 파일을 삭제한다던지, 아니면 파일의 정보 - 예를 들어서 파일의 생성 시간이라던지, 권한이라던지와 같은 것들을 보는 데에도 사용할 수 있습니다.

자 그렇다면 파일 시스템 라이브러리를 어떻게 사용하는지 차근 차근 살펴보도록 하겠습니다.

파일을 찾아보자

먼저 파일 시스템 라이브러리를 어떻게 사용하는지 간단한 예제로 살펴보도록 하겠습니다.

```
#include <filesystem>
#include <iostream>

int main() {
    std::filesystem::path p("./some_file");

    std::cout << "Does " << p << " exist? [" << std::boolalpha
        << std::filesystem::exists(p) << "]"
        << std::endl;
    std::cout << "Is " << p << " file? [" << std::filesystem::is_regular_file(p)
        << "]"
        << std::endl;
    std::cout << "Is " << p << " directory? [" << std::filesystem::is_directory(p)
        << "]"
        << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Does "./some_file" exist? [true]
Is "./some_file" file? [true]
Is "./some_file" directory? [false]
```

와 같이 나옵니다. 참고로 g++ 로 컴파일 하시는 분들은 꼭 8 버전 이상의 컴파일러가 설치되어 있어야 <filesystem> 을 사용하실 수 있습니다. 그 이하 버전의 경우 <experimental/filesystem> 을 사용하셔야 합니다 (없을 수도 있음) 특히 컴파일 시에 반드시 아래와 같이 컴파일 옵션을 줘야 합니다.

```
g++-9 test.cc -o test --std=c++17
```

또한 필요에 따라서 -lstdc++fs 를 추가해야 할 수도 있습니다.

파일 시스템 라이브러리의 경우 모든 클래스와 함수들이 std::filesystem 이름 공간 안에 정의되어 있습니다. 예를 들어서 파일 시스템의 path 클래스를 사용하기 위해서는 위와 같이

```
std::filesystem::path
```

와 같이 써야 합니다. 이는 기존의 chrono 라이브러리에서 std::chrono 안에 정의되어 있는 것과 일맥 상통합니다. std::filesystem 를 매번 일일히 쓰는 것이 번거롭기 때문에 편의상 그냥

```
namespace fs = std::filesystem;
```

와 같이 정의해놓고, fs::path 와 같이 간단하게 쓰는 것이 보통입니다.

경로 (path)

자 위 코드를 다시 살펴봅시다. 먼저 path 클래스의 객체를 선언하는 부분부터 봅시다.

```
std::filesystem::path p("./some_file");
```

컴퓨터 상의 모든 파일에는 해당 파일의 위치를 나타내는 고유의 주소가 있는데 이를 경로(path)라고 합니다. 왜 이를 주소가 아니라 경로라고 부르냐면, 컴퓨터에서 해당 파일을 참조할 때 가장

맨 첫 번째 디렉토리⁷⁾ 부터 순차적으로 찾아가기 때문입니다. 예를 들어서 /a/b/c 라는 경로를 따라가기 위해서는 맨 처음에 /a 디렉토리를 찾고, 그 디렉토리 안에 b 라는 디렉토리를 찾고 맨 마지막으로 b 안에 c 라는 파일을 찾기 때문이지요.

이 때 경로를 지정하는 방식에는 두 가지가 있는데, 바로 절대 경로 (absolute path) 와 상대 경로 (relative path) 가 있습니다.

- 절대 경로는 가장 최상위 디렉토리 (이를 보통 root 디렉토리라고 합니다) 에서 내가 원하는 파일까지의 경로를 의미하는 말입니다. 윈도우의 경우 root 디렉토리는 C:\나 D:\와 같은 것들이 되겠고, 리눅스의 경우 간단히 / 가 될 것입니다. 즉, 경로의 맨 앞에 / 거나 C:\이면 절대 경로라 생각하시면 됩니다.
- 상대 경로의 경우 반대로 현재 프로그램이 실행되고 있는 위치에서 해당 파일을 찾아가는 경로입니다. 예를 들어서 경로를 그냥 a/b 라고 했다면 이는 현재 프로그램의 실행 위치에서 a라는 디렉토리를 찾고 그 안에 b라는 파일을 찾는 식이지요.
따라서 만약에 현재 프로그램의 실행 절대 경로가 /foo/bar라면 b의 절대 경로는 /foo/bar/a/b 가 될 것입니다.

그렇다면 우리가 전달한 ./some_file 의 경우는 어떨까요? 맨 앞이 / 가 아니므로 이는 상대 경로입니다. 참고로 . 은 현재 디렉토리를 의미하는 문자 이므로 결과적으로 위 경로는 현재 프로그램이 실행되고 있는 위치에 존재하는 some_file 를 나타내겠지요.

`filesystem` 라이브러리에서 파일이나 디렉토리를 다루는 모든 함수들은 파일을 나타내기 위해서 `path` 객체를 인자로 받습니다. 따라서 보통

1. 원하는 경로에 있는 파일/디렉토리의 `path` 를 정의
2. 해당 `path` 로 파일/디렉토리 정보 수집

의 순서로 작업을 하게 됩니다. 한 가지 중요한 점은 `path` 객체만으로는 실제 해당 경로에 파일이 존재하는지 아닌지 알 수 없습니다. `path` 클래스는 그냥 경로를 나타낼 뿐 실제 파일을 지칭하는 것은 아닙니다.

만약에 해당 경로에 파일이 실제로 존재하는지 아닌지 보려면 아래와 같이 `exists` 함수를 사용해야 합니다.

```
std::cout << "Does " << p << " exist? [" << std::boolalpha
    << std::filesystem::exists(p) << "] " << std::endl;
```

위 경우 p에 파일이 존재한다면 `true` 라고 표시 됩니다.

7) 흔히 우리가 폴더라고 부르는 것이 바로 디렉토리(directory) 입니다.

```
std::cout << "Is " << p << " file? [" << std::filesystem::is_regular_file(p)
      << "]"
std::cout << "Is " << p << " directory? [" << std::filesystem::is_directory(p)
      << "]"
```

비슷하게 해당 위치에 있는 것이 파일인지 아니면 디렉토리인지 `is_regular_file` 과 `is_directory` 함수로 확인할 수 있습니다.

참고로 왜 그냥 `is_file` 이 아니라 굳이 *regular* 파일인지 궁금하실 수 있는데 이는 리눅스 상에서 주변 장치(device) 나 소켓(socket) 들도 다 파일로 취급하기 때문입니다. 추후에 시간이 되면 "*Everything is a File*" 이라는 글을 한 번 읽어보세요!

여러 경로 관련 함수들

파일시스템 라이브러리에서는 경로를 가지고 여러가지 작업을 할 수 있는 함수들을 지원합니다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./some_file.txt");

    std::cout << "내 현재 경로 : " << fs::current_path() << std::endl;
    std::cout << "상대 경로 : " << p.relative_path() << std::endl;
    std::cout << "절대 경로 : " << fs::absolute(p) << std::endl;
    std::cout << "공식적인 절대 경로 : " << fs::canonical(p) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
내 현재 경로 : "/Users/jblee/Test"
상대 경로 : "./some_file.txt"
절대 경로 : "/Users/jblee/Test./some_file.txt"
공식적인 절대 경로 : "/Users/jblee/Test/some_file.txt"
```

와 같이 나옵니다.

먼저 `current_path()` 함수의 경우 프로그램이 실행되는 경로를 리턴하게 됩니다. 모든 상대 경로는 이 프로그램의 현재 실행 경로를 기반으로 해서 구해집니다. 예를 들어서 우리의 프로그램의 경로가 `/Users/jblee/Test` 였으므로 `./some_file.txt`의 절대 경로는 `/Users/jblee/Test/some_file.txt` 가 되겠죠.

물론 주어진 경로를 바로 절대 경로로 바꾸고 싶다면 `absolute` 함수를 사용하면 됩니다. 하지만 `absolute` 의 단점이라 하면 주어진 경로를 절대 경로로 바꿔주기는 하지만 .. 이나 .. 와 같은 불필요한 요소들을 포함할 수 있습니다.⁸⁾

따라서 좀 더 깔끔하게 표현하고자 한다면 `canonical` 함수를 사용하면 됩니다. `canonical` 의 경우 해당 파일의 경로를 가장 짧게 나타낼 수 있는 공식적인 절대 경로를 제공합니다.

위 모든 함수들의 경우 입력 받는 경로에 파일이 존재하지 않는다면 모두 예외를 `throw` 합니다. 따라서 위 함수들을 호출하기 전에 반드시 `exist` 를 통해서 파일이 존재하는지 확인해야 합니다.

만약에 예외를 처리하고 싶지 않다면 마지막 인자로 발생한 오류를 받는 `std::error_code` 객체를 전달하면 됩니다. 이 경우 예외를 던질 상황이 생기면 예외를 던지는 대신에 `error_code` 객체에 발생한 오류를 설정합니다. 참고로 `filesystem` 에서 예외를 던지는 함수들의 경우 이처럼 마지막 인자로 `error_code` 를 받는 오버로딩이 제공됩니다.

디렉토리 관련 작업들

파일 시스템 라이브러리를 통해서 디렉토리에 여러가지 작업들을 할 수 있습니다. 예를 들어서

- 해당 디렉토리 안에 있는 파일/폴더들 살펴보기
- 해당 디렉토리 안에 폴더 생성하기 (파일 생성은 `ofstream` 으로 할 수 있죠!)
- 해당 디렉토리 안에 파일/폴더 복사하기
- 해당 디렉토리 안에 파일/폴더 삭제하기

등등을 말입니다.

디렉토리 안에 모든 파일들 순회하기

가장 먼저 디렉토리 안에 있는 파일들을 접근하는 방법을 살펴봅시다. 디렉토리는 그냥 쉽게 생각하면 책의 목차라고 보시면 됩니다. 즉, 디렉토리는 해당 디렉토리에 어떠한 파일들이 정의되어

8) .. 의 경우 상위 폴더를 나타내는 경로입니다. 예를 들어서 `/a/..` 는 그냥 `a` 입니다.

있는지 쭈르륵 써져 있는 파일이라 볼 수 있습니다. 이를 쉽게 접근하기 위해서 `filesystem` 라이브러리에서는 `directory_iterator`라는 반복자를 제공합니다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::directory_iterator itr(fs::current_path() / "a");
    while (itr != fs::end(itr)) {
        const fs::directory_entry& entry = *itr;
        std::cout << entry.path() << std::endl;
        itr++;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
"/Users/jblee/Test/a/3.txt"
"/Users/jblee/Test/a/2.txt"
"/Users/jblee/Test/a/1.txt"
"/Users/jblee/Test/a/b"
```

와 같이 나옵니다. 먼저 `directory_iterator`를 정의하는 부분부터 살펴봅시다.

```
fs::directory_iterator itr(fs::current_path() / "a");
```

기존에 `vector` 와 같은 컨테이너를 생각했을 때 반복자를 정의하기 위해서는 `v.begin()`처럼 컨테이너 자체의 함수를 통해서 정의하였지 직접 반복자를 정의하지 않았는데요, `directory_iterator`는 특이하게 반복자 자체를 스스로 정의해야 하고, 반복자의 생성자에 우리가 탐색할 경로를 전달해줘야 합니다.

참고로 `path`에는 `operator/` 가 정의되어 있어서

```
fs::current_path() / "a"
```

위와 같이 현재 경로에 `/a`를 편리하게 추가할 수 있습니다. (매우 직관적이죠!) 즉 우리 프로그램의 실행 경로가 `/Users/jblee/Test` 이므로 위 반복자는 `/Users/jblee/Test/a` 안에 있는 파일들을 탐색하게 됩니다.

자 그럼 이 반복자의 끝은 어떻게 나타낼까요?

```
while (itr != fs::end(itr)) {
```

바로 `filesystem` 에 정의되어 있는 `end` 함수에 현재 반복자를 전달하면 해당 반복자의 끝을 얻어낼 수 있습니다.

```
const fs::directory_entry& entry = *itr;
```

그리고 각각의 반복자들은 디렉토리에 정의되어 있는 개개의 파일을 나타내는 `directory_entry` 를 가리키고 있습니다. `directory_entry` 에는 여러가지 정보들이 저장되어 있는데 파일의 이름이나, 크기 등등을 알 수 있습니다.

```
std::cout << entry.path() << std::endl;
```

그리고 마지막으로 해당 파일의 경로를 위와 같이 출력하였습니다.

위 코드는 C++ 11 에 도입된 `range for` 문을 사용하면 더욱 간단히 작성할 수 있습니다.

```
#include <iostream>

namespace fs = std::filesystem;

int main() {
    for (const fs::directory_entry& entry :
        fs::directory_iterator(fs::current_path() / "a")) {
        std::cout << entry.path() << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
"/Users/jblee/Test/a/3.txt"
"/Users/jblee/Test/a/2.txt"
"/Users/jblee/Test/a/1.txt"
"/Users/jblee/Test/a/b"
```

와 같이 동일하게 나옵니다. 매우 깔끔하죠?

`directory_iterator` 의 한 가지 단점은 해당 디렉토리 안에 다른 디렉토리가 있을 경우 그 안까지는 살펴보지 않는다는 점입니다. 예를 들어서 위 경우 `b` 안에 여러 파일들이 있는데 이들은 순회 대상에서 제외되었습니다.

만약에 여러분이 디렉토리 안에 서브 디렉토리까지 모두 순회할 수 있는 반복자를 사용하고 싶다면 `recursive_directory_iterator` 를 사용하면 됩니다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    for (const fs::directory_entry& entry :
        fs::recursive_directory_iterator(fs::current_path() / "a")) {
        std::cout << entry.path() << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
"/Users/jblee/Test/a/3.txt"
"/Users/jblee/Test/a/2.txt"
"/Users/jblee/Test/a/1.txt"
"/Users/jblee/Test/a/b"
"/Users/jblee/Test/a/b/5.txt"
"/Users/jblee/Test/a/b/4.txt"
"/Users/jblee/Test/a/b/6.txt"
```

와 같이 잘 나옵니다.

디렉토리 생성하기

이전에 `ofstream` 라이브러리를 사용했으면 파일을 간단하게 생성할 수 있었습니다. 간단히 아래 처럼

```
std::ofstream out("a.txt");
out << "hi";
```

를 하면 만약에 `a.txt`라는 파일이 존재하지 않을 시에 파일이 생성 되며 내용이 작성됩니다.

하지만 `ofstream` 라이브러리를 통해서는 디렉토리를 생성할 수 없습니다. 예를 들어서

```
std::ofstream out("./b/a.txt");
out << "hi";
```

만약에 `b`라는 폴더가 없다면 `a.txt`도 생성되지 않고 위 `out <<`은 실패하게 됩니다.

따라서 디렉토리를 생성하고 싶다면 `filesystem`에서 제공하는 `create_directory` 함수를 사용해야 합니다. 아래 예제를 보시죠.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./a/c");
    std::cout << "Does " << p << " exist? [" << std::boolalpha << fs::exists(p)
        << "]"
        << std::endl;

    fs::create_directory(p);

    std::cout << "Does " << p << " exist? [" << fs::exists(p) << "]"
        << std::endl;
    std::cout << "Is " << p << " directory? [" << fs::is_directory(p) << "]"
        << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Does "./a/c" exist? [false]
Does "./a/c" exist? [true]
Is "./a/c" directory? [true]
```

와 같이 잘 나옵니다.

```
fs::create_directory("./a/c");
```

`create_directory` 함수는 주어진 경로를 인자로 받아서 디렉토리를 생성합니다. 다만 한 가지 주의할 점은 생성하는 디렉토리의 부모 디렉토리는 반드시 존재하고 있어야 합니다. 위 경우 `./a`라는 디렉토리가 이미 존재하고 있기 때문에 `create_directory` 가 성공적으로 수행되었습니다.

예를 들어서

```
fs::path p("./c/d/e/f"); // ./c 는 존재하고 있지 않는 디렉토리
fs::create_directory(p);
```

위를 실행한다면

실행 결과

```
terminate called after throwing an instance of
  'std::filesystem::__cxx11::filesystem_error'
    what():  filesystem error: cannot create directory: No such file
              or directory [./c/d/e/f]
[1] 15954 abort (core dumped) ./test
```

와 같이 예외를 던지게 됩니다.

그렇다면 만약에 부모 디렉토리들까지 한꺼번에 만들고 싶다면 어떨까요? 이를 위해선 `create_directories` 함수를 사용하면 됩니다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./c/d/e/f");
    std::cout << "Does " << p << " exist? [" << std::boolalpha << fs::exists(p)
                  << "]"
                  << std::endl;

    fs::create_directories(p);

    std::cout << "Does " << p << " exist? [" << fs::exists(p) << "]"
                  << std::endl;
    std::cout << "Is " << p << " directory? [" << fs::is_directory(p) << "]"
                  << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Does "./c/d/e/f" exist? [false]
```

```
Does "./c/d/e/f" exist? [true]
Is "./c/d/e/f" directory? [true]
```

위와 같이 c, d, e, f 디렉토리 전부가 잘 생성된 것을 볼 수 있습니다. 실제로 `tree` 명령어를 통해 디렉토리를 살펴보면

실행 결과

```
$ tree
└── c
    └── d
        └── e
            └── f
```

이쁘게 잘 나옵니다!

파일과 폴더 복사/삭제하기

마지막으로 파일 시스템 라이브러리에서 살펴볼 기능으로 복사와 삭제 기능입니다. 물론 기존의 `ofstream` 을 통해서 파일 간의 복사는 가능했습니다. (파일 내용 전체를 읽어들인 뒤에 다른 파일로 출력하는 방식으로). 하지만 디렉토리를 복사하거나, 디렉토리 안의 모든 파일들을 복사하는 등의 작업들은 불가능하였는데, `filesystem` 의 `copy` 를 사용하면 간단히 수행할 수 있습니다.

실행 결과

```
$ tree
└── a
    ├── a.txt
    └── b
        └── c.txt
└── c
```

예를 들어서 현재 파일들 상황이 위와 같다고 해봅시다. a 디렉토리 안에 여러 파일과 폴더들이 있고 c 는 비어있는 디렉토리입니다. 만약에 a 안의 모든 파일들을 c 에 복사하고 싶다면 어떨까요? 아주 간단합니다.

```
#include <filesystem>
#include <iostream>
```

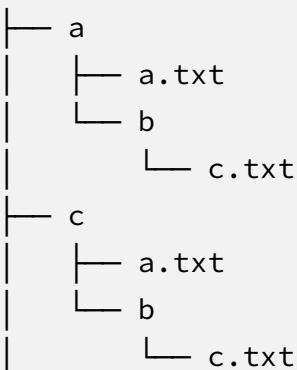
```
namespace fs = std::filesystem;

int main() {
    fs::path from("./a");
    fs::path to("./c");

    fs::copy(from, to, fs::copy_options::recursive);
}
```

성공적으로 컴파일 하였다면 아래와 같이 파일과 폴더들이 잘 복사된 것을 볼 수 있습니다.

실행 결과



`copy` 함수에 복사할 대상과, 복사할 위치를 차례대로 인자로 전달하면 됩니다. 그리고 마지막 인자로 어떠한 방식으로 복사할지 지정해줘야 합니다.

실행 결과

```
fs::copy(from, to, fs::copy_options::recursive);
```

우리의 경우 `recursive` 옵션을 주었는데 이는 복사할 대상에 존재하는 모든 디렉토리와 파일들을 복사하게 됩니다. 만일 `recursive` 옵션을 주지 않는다면 `a`에 존재하는 파일들, 즉 `a.txt` 딱 하나만 복사됩니다.

만약에 복사할 대상이 이미 존재하고 있다면 예외를 던지게 됩니다. 예를 들어서 위 상태에서 `a/a.txt`를 `c`에 복사한다고 해봅시다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path from("./a/a.txt");
```

```

fs::path to("./c");

fs::copy(from, to);
}

```

컴파일 하였다면

실행 결과

```

terminate called after throwing an instance of
  'std::filesystem::__cxx11::filesystem_error'
  what():  filesystem error: cannot copy: File exists [./a/a.txt]
            [./c]
[1] 21859 abort (core dumped) ./test

```

위와 같이 c에 이미 a.txt가 존재한다고 예외가 던져집니다.

다행으로 filesystem 라이브러리에는 이 경우 여러분이 택할 수 있는 선택지를 3개나 제공하고 있습니다.

- `skip_existing` : 이미 존재하는 파일은 무시 (예외 안던지고)
- `overwrite_existing` : 이미 존재하는 파일은 덮어 씌운다.
- `update_existing` : 이미 존재하는 파일이 더 오래되었을 경우 덮어 씌운다.

예를 들어서

```
fs::copy(from, to, fs::copy_options::overwrite_existing);
```

와 같이 한다면 a.txt를 잘 덮어씌우겠지요.

파일 / 디렉토리 삭제하기

자 그러면 마지막으로 살펴볼 기능은 파일이나 폴더를 삭제하는 작업입니다. 파일 삭제는 `remove` 함수를 통해서 간단히 경로만 전달하면 수행할 수 있습니다.

```

#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

```

```

int main() {
    fs::path p("./a/b.txt");
    std::cout << "Does " << p << " exist? [" << std::boolalpha
        << std::filesystem::exists(p) << "]"
        << std::endl;
    fs::remove(p);
    std::cout << "Does " << p << " exist? [" << std::boolalpha
        << std::filesystem::exists(p) << "]"
        << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Does "./a/b.txt" exist? [true]
Does "./a/b.txt" exist? [false]

```

와 같이 잘 나옵니다.

`remove` 함수를 통해서 디렉토리 역시 지울 수 있습니다. 단, 해당 디렉토리는 반드시 빈 디렉토리여야 합니다. 만일 비어있지 않은 디렉토리를 삭제하고 싶다면 `remove_all` 함수를 사용하면 됩니다.

```

#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./c/b");

    std::error_code err;
    fs::remove(p, err); // 실패
    std::cout << err.message() << std::endl;

    fs::remove_all(p); // 성공!
}

```

성공적으로 컴파일 하였다면

실행 결과

```

Directory not empty

```

와 같이 나옵니다. 이 메세지는 `fs::remove`에서 발생한 메세지 이죠.

하지만 `remove_all`을 통해서 제대로 c/b 가 지워진 것을 확인할 수 있습니다.

실행 결과

```
$ tree
└── c
    ├── a.txt
    └── b
        └── c.txt
```

위 상태에서

실행 결과

```
$ tree
└── c
    └── a.txt
```

로 제대로 b 가 지워졌습니다.

directory_iterator 사용시 주의할 점

예를 들어서 해당 디렉토리 안에 확장자가 txt 인 파일을 모두 삭제하는 프로그램을 만들고 싶다고 해봅시다. 이 경우 아마 아래와 같이 코드를 작성할 것입니다.

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./a");
    for (const auto& entry : fs::directory_iterator("./a")) {
        const std::string ext = entry.path().extension();
        if (ext == ".txt") {
            fs::remove(entry.path());
        }
    }
}
```

하지만 사실 이 코드는 한 가지 문제점이 있습니다.

```
fs::remove(entry.path());
```

./a 디렉토리에서 파일을 하나 삭제할 때마다 해당 디렉토리의 구조가 바뀌게 됩니다. 그런데 `directory_iterator`는 디렉토리의 구조가 바뀔 때마다 무효화 됩니다! 따라서 `fs::remove` 후에 `entry`는 사용할 수 없는 반복자가 됩니다. 따라서 `++entry` 가 다음 디렉토리를 가리키는 것을 보장할 수 없게 되죠. 따라서 이 경우 어쩔 수 없이;

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    fs::path p("./a");
    while (true) {
        bool is_modified = false;
        for (const auto& entry : fs::directory_iterator("./a")) {
            const std::string ext = entry.path().extension();
            if (ext == ".txt") {
                fs::remove(entry.path());
                is_modified = true;
                break;
            }
        }
        if (!is_modified) {
            break;
        }
    }
}
```

위와 같이 파일을 삭제할 때마다 반복자를 초기화 해줘야만 합니다.

자 그럼 이것으로 `filesystem` 라이브러리를 살펴보았습니다. 파일 시스템 라이브러리를 통해서 기존에 파일 입출력 라이브러리로는 할 수 없었던 디렉토리들을 다루는 것과 파일 삭제, 복사 등등을 간단히 수행할 수 있게 되었습니다.

이제 정말 C++ 강좌도 끝을 향해서 달려갑니다. (마지막이 될 수도 있는) 다음 강좌에서는 C++의 `utility` 라이브러리에 정의된 여러가지 도구들을 살펴볼 것입니다.

뭘 배웠지?

- `filesystem` 라이브러리를 통해서 파일과 디렉토리를 다룰 수 있다.
- `path` 객체를 통해 파일과 폴더의 경로에 대한 작업을 손쉽게 할 수 있다.
- `directory_iterator` 을 통해 디렉토리 안에 존재하는 파일들을 살펴볼 수 있고, `recursive_directory_iterator` 을 사용해서 디렉토리 안에 있는 모든 파일/폴더들을 탐색할 수 있다.
- `create_directory` 를 통해 디렉토리를 생성하고 `copy`, `remove` 함수를 통해 파일/폴더를 복사/삭제 할 수 있다.

C++ 유틸리티 라이브러리 소개

안녕하세요 여러분! 이번 강좌에서는 기존 강좌에서 채 다루지 못한 C++ 표준 라이브러리에서 제공하는 여러가지 유용한 도구들에 대해 간단하게 설명하도록 하겠습니다.

참고로 아래에서 설명할 도구들에 대한 설명은 C++ 17 기준으로 작성되었으며, 그 이하 버전의 C++ 을 사용하는 경우 원하는 기능을 사용하지 못할 수도 있습니다. 각각의 요소들에 대해서 최소 몇 이상의 C++ 을 사용해야 하는지 명시하였습니다.

std::optional (C++ 17 이상 - <optional>)

예를 들어서 어떠한 map 에서 주어진 키에 대응하는 값이 있는지 확인하는 함수를 만들고 싶다고 해봅시다. 그렇다면 어떤 식으로 함수를 작성하면 될까요?

만약에 단순하게 짠다면 아래와 같이 작성할 수 있습니다.

```
#include <iostream>
#include <map>
#include <string>

std::string GetValueFromMap(const std::map<int, std::string>& m, int key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        return itr->second;
    }

    return std::string();
}

int main() {
    std::map<int, std::string> data = {{1, "hi"}, {2, "hello"}, {3, "hiroo"}};
    std::cout << "맵에서 2 에 대응되는 값은? " << GetValueFromMap(data, 2)
              << std::endl;
    std::cout << "맵에서 4 에 대응되는 값은? " << GetValueFromMap(data, 4)
              << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

맵에서 2 에 대응되는 값은? hello

맵에서 4 에 대응되는 값은?

와 같이 나옵니다.

잘 돌아가는 것 처럼 보입니다. 하지만 위 방식은 한 가지 문제점이 있습니다. 바로 맵에 주어진 키가 들어가 있지 않을 때 인데요;

```
return std::string();
```

위 경우 빈 string 객체를 리턴하지만 만약에 진짜로 어떤 키에 대응하는 값이 빈 문자열이면 어떨까요? 따라서 위와 같은 방식은 맵에 키가 존재하지 않는 경우 와 키가 존재하는데 대응하는 값이 빈 문자열인 경우 를 제대로 구분하지 못하게 됩니다.

그렇다면 아래와 같은 방식은 어떨까요?

```
#include <iostream>
#include <map>
#include <string>

std::pair<std::string, bool> GetValueFromMap(
    const std::map<int, std::string>& m, int key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        return std::make_pair(itr->second, true);
    }

    return std::make_pair(std::string(), false);
}

int main() {
    std::map<int, std::string> data = {{1, "hi"}, {2, "hello"}, {3, "hiroo"}};
    std::cout << "맵에서 2 에 대응되는 값은? " << GetValueFromMap(data, 2).first
          << std::endl;
    std::cout << "맵에 4 는 존재하나요 " << std::boolalpha
          << GetValueFromMap(data, 4).second << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

맵에서 2 에 대응되는 값은? hello

맵에 4 는 존재하나요 false

와 같이 나옵니다. 이번에는 아예 `std::pair` 를 이용해서 대응하는 값과 함께 실제 맵에 존재하는지의 유무를 같이 전달하도록 하였습니다. 이 방식도 꽤 팬찮아 보이지만 한 가지 문제점이 있습니다.

바로 맵에 키가 존재하지 않을 때 디폴트 객체를 리턴해야 한다는 점입니다. 이는 몇 가지 문제점이 있는데;

1. 객체의 디폴트 생성자가 정의되어 있지 않을 수도 있고
2. 객체를 디폴트 생성하는 것이 매우 오래 걸릴 수도 있다

와 같기 때문입니다. 이와 같은 문제를 해결하기 위해서는 원하는 값을 보관할 수도, 안할 수도 있는 클래스를 도입하는 것입니다.⁹⁾ 이를 가능하게 한 것이 바로 `std::optional` 입니다.

`std::optional` 를 어떻게 사용하는지 아래 예제를 통해 간단히 살펴봅시다.

```
#include <iostream>
#include <map>
#include <string>
#include <utility>

std::optional<std::string> GetValueFromMap(const std::map<int, std::string>& m,
                                             int key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        return itr->second;
    }

    // nullopt 는 <utility> 에 정의된 객체로 비어있는 optional 을 의미한다.
    return std::nullopt;
}

int main() {
    std::map<int, std::string> data = {{1, "hi"}, {2, "hello"}, {3, "hiroo"}};
    std::cout << "맵에서 2 에 대응되는 값은? " << GetValueFromMap(data, 2).value()
          << std::endl;
    std::cout << "맵에 4 는 존재하나요? " << std::boolalpha
          << GetValueFromMap(data, 4).has_value() << std::endl;
}
```

성공적으로 컴파일 하였다면

9) 물론 다른 방식으로는 `GetValueFromMap` 에 `bool&` 를 받아서 객체의 존재 유무를 따로 뽑아낼 수도 있습니다. 하지만 함수의 디자인 측면에서 그리 깔끔한 방식은 아니지요.

실행 결과

```
맵에서 2 에 대응되는 값은? hello
맵에 4 는 존재하나요 false
```

와 같이 잘 나옵니다.

먼저 `std::optional` 의 정의부터 살펴봅시다.

```
std::optional<std::string>
```

위와 같이 템플릿 인자로 `optional` 이 보관하고자 하는 객체의 타입을 써주시면 됩니다. 해당 `optional` 객체는 `std::string` 을 보관하던지, 아니면 안하던지 둘 중 하나의 상태만을 가지게 됩니다.

```
auto itr = m.find(key);
if (itr != m.end()) {
    return itr->second;
}
```

그리고 `GetValueFromMap` 함수 안에서 키에 대응하는 값이 존재한다면 그냥 해당 값을 리턴하였습니다. `std::optional` 에는 보관하고자 하는 타입을 받는 생성자가 정의되어 있기 때문에 위와 같이 그냥 리턴하더라도 `optional` 객체로 알아서 만들어져서 리턴됩니다.

이 때 `optional` 의 가장 큰 장점으로, 객체를 보관하는 과정에서 동적 할당이 발생하지 않는다는 점입니다. 따라서 불필요한 오버헤드가 없습니다. 쉽게 생각해서, `optional` 자체에 객체가 포함되어 있다고 보시면 됩니다.

```
// nullopt 는 <utility> 에 정의된 객체로 비어있는 optional 을 의미한다.
return std::nullopt;
```

만약에 아무런 객체도 가지고 있지 않은 빈 `optional` 객체를 리턴하고 싶다면, 그냥 `nullopt` 객체를 리턴하면 됩니다. `std::nullopt` 는 미리 정의되어 있는 빈 `optional` 객체를 나타냅니다.

```
GetValueFromMap(data, 2).value()
```

만일 `optional` 객체가 가지고 있는 객체를 접근하고 싶다면 `value()` 함수를 호출하면 됩니다. 주의해야할 점은 만일 `optional` 이 가지고 있는 객체가 없다면 `std::bad_optional_access` 예외를 던지게 됩니다. 따라서 반드시 `optional` 가 들고 있는 객체에 접근하기 전에 실제로 값을 가지고 있는지 확인해야 하는데, 이는

```
GetValueFromMap(data, 4).has_value()
```

처럼 `has_value` 함수로 사용하면 됩니다. 한 가지 유용한 텁으로 `optional` 객체 자체에 `bool`로 변환하는 캐스팅 연산자가 포함되어 있으므로 그냥

```
if (GetValueFromMap(data, 4))
```

나

```
if (GetValueFromMap(data, 4).has_value())
```

는 동일한 의미의 문장이 되겠습니다. 마찬가지로 `value()` 함수 대신에 역참조 연산자를(*) 이용하셔도 됩니다. 즉 `GetValueFromMap(data, 2).value()` 와 `*GetValueFromMap(data, 2)` 는 동일한 문장입니다.

이 `std::optional<T>` 가 `std::pair<bool, T>` 와 가장 큰 차이점이 바로 `pair` 와는 달리 아무 것도 들고 있지 않는 상태에서 디폴트 객체를 가질 필요가 없다라는 점이었습니다. 해당 사실이 정말로 맞는지 아래 코드를 통해 확인해보겠습니다.

```
#include <iostream>
#include <utility>

class A {
public:
    A() { std::cout << "디폴트 생성" << std::endl; }

    A(const A& a) { std::cout << "복사 생성" << std::endl; }
};

int main() {
    A a;

    std::cout << "Optional 객체 만들 ---- " << std::endl;
    std::optional<A> maybe_a;

    std::cout << "maybe_a 는 A 객체를 포함하고 있지 않기 때문에 디폴트 생성할 "
           "필요가 없다."
           << std::endl;
    maybe_a = a;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
디폴트 생성
Optional 객체 만들 ----
maybe_a 는 A 객체를 포함하고 있지 않기 때문에 디폴트 생성할 필요가 없다.
복사 생성
```

와 같이 나옵니다. 보시다시피 `optional` 객체에 `a` 객체를 전달하기 직전까지 디폴트 생성되었다는 메세지가 뜨지 않습니다 (처음에 `a` 만들 때 빼고). 정말로 `optional`은 빈 객체 상태에서는 해당 객체를 가지고 있지 않는다는 사실을 알 수 있습니다.

이와 같이 `std::optional` 을 이용해서 어떠한 객체를 보관하거나 말거나 라는 의미를 쉽게 전달할 수 있습니다.

레퍼런스를 가지는 `std::optional`

`std::optional` 의 한 가지 단점으로는 일반적인 방법으로는 레퍼런스를 포함할 수 없다는 점입니다. 예를 들어서 아래와 같이 레퍼런스에 대한 `optional` 객체를 정의하고 한다면

```
#include <iostream>
#include <map>
#include <string>
#include <utility>

class A {
public:
    A() { std::cout << "디폴트 생성" << std::endl; }

    A(const A& a) { std::cout << "복사 생성" << std::endl; }
};

int main() {
    A a;

    std::optional<A&> maybe_a = a;
}
```

컴파일 하였을 경우

컴파일 오류

```
/usr/include/c++/9/optional: In instantiation of ‘union
  std::_Optional_payload_base<A&>::_Storage<A&, true>’:
```

```

/usr/include/c++/9/optional:239:30: required from ‘struct
→ std::_Optional_payload_base<A&>’
/usr/include/c++/9/optional:295:12: required from ‘struct
→ std::_Optional_payload<A&, true, true, true>’
/usr/include/c++/9/optional:628:30: required from ‘struct
→ std::_Optional_base<A&, true, true>’
/usr/include/c++/9/optional:656:11: required from ‘class
→ std::optional<A&>’

test.cc:16:21: required from here
/usr/include/c++/9/optional:212:15: error: non-static data member
→ ‘std::_Optional_payload_base<A&>::_Storage<A&,
→ true>::_M_value’ in a union may not have reference type ‘A&’
212 |         _Up _M_value;
|           ~~~~~~
/usr/include/c++/9/optional: In instantiation of ‘class
→ std::optional<A&>’:
test.cc:16:21: required from here
/usr/include/c++/9/optional:672:21: error: static assertion failed
672 |     static_assert(!is_reference_v<_Tp>);
|           ~~~~~~
/usr/include/c++/9/optional:888:7: error: forming pointer to
→ reference type ‘A&’
888 |     operator->() const
|       ~~~~~~
/usr/include/c++/9/optional:893:7: error: forming pointer to
→ reference type ‘A&’
893 |     operator->()
|       ~~~~~~

```

와 같은 끔찍한 컴파일 오류를 보실 수 있습니다.

물론, 그렇다고 해서 레퍼런스를 `optional`이 포함할 수 없는 것은 아닙니다. 바로 `std::reference_wrapper` 를 사용해서 레퍼런스 처럼 동작하는 `wrapper` 객체를 정의하면 됩니다.

```

#include <functional>
#include <iostream>
#include <optional>
#include <utility>

class A {
public:

```

```

int data;
};

int main() {
    A a;
    a.data = 5;

    // maybe_a 는 a 의 복사복이 아닌 a 객체 자체의 레퍼런스를 보관하게 된다.
    std::optional<std::reference_wrapper<A>> maybe_a = std::ref(a);

    maybe_a->get().data = 3;

    // 실제로 a 객체의 data 가 바뀐 것을 알 수 있다.
    std::cout << "a.data : " << a.data << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```
a.data : 3
```

와 같이 잘 실행되었음을 알 수 있습니다.

```
std::optional<std::reference_wrapper<A>> maybe_a = std::ref(a);
```

`std::reference_wrapper` 는 레퍼런스가 아니라 일반적인 객체이기 때문에 `optional`에 전달할 수 있습니다. `reference_wrapper` 를 `get()` 함수를 통해서 레퍼런스 하고 있는 객체를 얻어오게 됩니다. 대신 `reference_wrapper` 객체를 생성하기 위해서는 `std::ref` 함수를 사용해야 합니다.

```

maybe_a->get().data = 3;

// 실제로 a 객체의 data 가 바뀐 것을 알 수 있다.
std::cout << "a.data : " << a.data << std::endl;

```

앞서 `optional`에 역참조 연산자가 정의되어 있어서 가지고 있는 값을 `*`를 통해 간단하게 얻어낼 수 있다고 하였는데요, 이와 같은 선상에서 `->` 연산자 역시 정의되어 있어서 가지고 있는 값에 함수를 호출할 수 있습니다.

따라서 위와 같이 `reference_wrapper` 가 가지고 있는 `a`에 대한 레퍼런스를 `get` 함수를 통해 얻어낼 수 있습니다. 그리고 해당 레퍼런스의 `data` 를 바꾸면 실제로 `a`의 `data` 가 바뀐 것을 확인할 수 있죠!

이와 같이 `std::optional` 은 여러 모로 아주 쓸모가 많은 녀석입니다. 혼업에서 여러 모로 자주 활용할 수 있을 것이라 생각합니다.

`std::variant (C++ 17 이상 - <variant>)`

`std::variant` 는 *one-of* 를 구현한 클래스라고 보시면 됩니다. 즉 컴파일 타임에 정해진 여러가지 타입들 중에 한 가지 타입의 객체를 보관할 수 있는 클래스입니다.

물론 공용체(union) 을 이용해서 해결할 수도 있겠지만, 공용체가 현재 어떤 타입의 객체를 보관하고 있는지 알 수 없기 때문에 실제로 사용하기에는 매우 위험합니다.

예를 들어서 아래 간단한 예제를 살펴봅시다.

```
// v 는 이제 int
std::variant<int, std::string, double> v = 1;

// v 는 이제 std::string
v = "abc";

// v는 이제 double
v = 3.14;
```

먼저 `variant` 를 정의할 때 포함하고자 하는 타입들을 명시해줘야 합니다. 우리의 경우 정의한 `variant` 는 `int`, `std::string`, `double` 이 세 중 하나의 타입을 가질 수 있습니다.

`variant` 의 가장 큰 특징으로는 반드시 값을 들고 있어야 한다는 점입니다. 만약에 그냥

```
std::variant<int, std::string, double> v;
```

을 정의한다면 `v` 에는 첫 번째 타입 인자 (`int`) 의 디폴트 생성자가 호출되게 됩니다. 즉 위 경우 `v`에는 0 이 들어가겠지요. 즉 비어 있는 `variant` 는 불가능한 상태라고 보시면 됩니다.

`variant` 는 `optional` 과 비슷하게 객체의 대입 시에 어떠한 동적 할당도 발생하지 않습니다. 따라서 굉장히 작은 오버헤드로 객체들을 보관할 수 있습니다. 다만 `variant` 객체 자체의 크기는 나열된 가능한 타입들 중 가장 큰 타입의 크기를 따라갑니다.

`variant` 는 이리이러한 타입들 중 하나(*one-of*) 를 표현하기에 매우 적합한 도구입니다. 예를 들어서 어떤 데이터 베이스에 검색을 해서 결과를 돌려주는 함수를 생각해봅시다. 이 결과는 조건에 따라 클래스 A 객체나 클래스 B 객체가 될 수 있습니다.

```
class A {};
class B {};
```

```
/* ?? */ GetDataFromDB(bool is_a) {
    if (is_a) {
        return A();
    }
    return B();
}
```

여러분이라면 위 함수를 어떻게 만들 수 있을까요? 상황에서 따라서 A 나 B 객체를 리턴할 수 있는 함수를요.

한 가지 방법이라면 C++ 의 다형성(polymorphism)을 이용하는 것입니다. 이를 위해서는 A 와 B 클래스의 공통 부모가 정의되어 있어야 합니다.

```
class Data {};
class A : public Data {};
class B : public Data {};

std::unique_ptr<Data> GetDataFromDB(bool is_a) {
    if (is_a) {
        return std::make_unique<A>();
    }
    return std::make_unique<B>();
}
```

따라서 위와 같이 A 혹은 B 객체를 리턴할 수 있습니다. 그리고 해당 함수를 호출하는 곳에서 리턴하는 Data 의 실제 객체가 무엇인지 간단하게 알아낼 수 있겠고요.

하지만 위 문제는 리턴하고자 하는 클래스들의 부모 클래스가 공통으로 정의되어 있어야 하고, `std::string` 이나 `int` 와 같은 표준 클래스의 객체들에는 적용할 수 없다는 문제가 있습니다. 하지만 `std::variant` 를 이용하면 매우 간단하게 해결할 수 있습니다.

```
#include <iostream>
#include <memory>
#include <variant>

class A {
public:
    void a() { std::cout << "I am A" << std::endl; }
};

class B {
public:
    void b() { std::cout << "I am B" << std::endl; }
};

std::variant<A, B> GetDataFromDB(bool is_a) {
    if (is_a) {
```

```

    return A();
}
return B();
}

int main() {
auto v = GetDataFromDB(true);

std::cout << v.index() << std::endl;
std::get<A>(v).a(); // 혹은 std::get<0>(v).a()
}

```

성공적으로 컴파일 하였다면

실행 결과

```

0
I am A

```

와 같이 나옵니다.

```

std::variant<A, B> GetDataFromDB(bool is_a) {
if (is_a) {
    return A();
}
return B();
}

```

`variant` 역시 `optional` 과 마찬가지로 각각의 타입의 객체를 받는 생성자가 정의되어 있기 때문에 그냥 `A` 를 리턴하면 `A` 를 가지는 `variant` 가, `B` 를 리턴하면 `B` 를 가지는 `variant` 가 생성됩니다.

자 그렇다면 이렇게 `variant` 에서 원하는 값을 어떻게 뽑을 수 있는지 살펴보도록 하겠습니다.

```

std::cout << v.index() << std::endl;
std::get<A>(v).a(); // 혹은 std::get<0>(v).a()

```

먼저 현재 `variant` 에 몇 번째 타입이 들어있는지 알고 싶다면 `index()` 함수를 사용하면 됩니다. 우리의 경우 `A` 타입의 객체가 들어 있는데 `A` 는 `variant` 에서 첫 번째 타입이므로 0 을 리턴하게 되겠죠.

그 다음으로 실제로 원하는 값을 뽑아내고 싶다면 외부에 정의되어 있는 함수인 `std::get<T>` 를 이용하시면 됩니다. 이 때 이 `T` 자리에 우리가 뽑아내고자 하는 타입을 써주던지, 아니면 해당 타입의 `index` 를 넣어주시면 됩니다.

따라서 A를 뽑고 싶다면 `std::get<A>(v)` 나 `std::get<0>(v)`를 하면 되고, B를 뽑고 싶다면 `std::get(v)` 나 `std::get<1>(v)`를 하면 됩니다.

여기서 한 가지 알 수 있는 점은 `variant` 가 보관하는 객체들은 타입으로 구분된다는 점입니다. 따라서 `variant` 를 정의할 때 같은 타입을 여러 번 써주면 안됩니다. 예를 들어서

```
std::variant<std::string, std::string> v;
```

는 컴파일 시 오류가 발생하게 됩니다.

`std::monostate`

만약에 굳이 `variant` 에 아무 것도 들고 있지 않은 상태를 표현하고자 싶다면 해당 타입으로 `std::monostate` 를 사용하면 됩니다. 이를 통해서 마치 `std::optional` 과 같은 효과를 낼 수 있습니다.

예를 들어서

```
std::variant<std::monostate, A, B> v;
```

와 같이 `variant` 를 정의한다면 v에는 아무것도 안들어 있거나 A 혹은 B 가 들어가 있을 수 있습니다. 또한 `variant` 안에 정의된 타입들 중에 디폴트 생성자가 있는 타입이 하나도 없는 경우 역시 `std::monostate` 를 활용하면 됩니다. 예를 들어서

```
class A {
public:
    A(int i) {}
};

class B {
public:
    B(int i) {}
};
```

위와 같이 디폴트 생성자가 없는 클래스가 있다고 하였을 때;

```
std::variant<A, B> v;
```

를 뎅그러니 정의하게 되면 컴파일 오류가 발생하게 됩니다. 왜냐하면 앞서 이야기 하였듯이 `variant` 는 반드시 객체를 들고 있어야 하는데, 이를 지정하지 않을 경우 자동으로 첫 번째 타

입의 디폴트 생성된 객체를 갖고 있으려고 하기 때문이죠. 하지만 위의 경우 A 의 디폴트 생성자가 없기 때문에 컴파일 오류가 발생합니다.

이 경우 그냥 첫 번째 타입으로 `std::monostate` 를 지정해주면 깔끔하게 해결됩니다.

```
std::variant<std::monostate, A, B> v;
```

와 같이 하게 되면 디폴트로 `std::monostate` 가 v 에 들어가게 되어서 문제 없습니다.

std::tuple (C++ 11 이상 - <tuple>)

마지막으로 여러 서로 다른 타입들의 묶음을 간단하게 다룰 수 있도록 제공하는 `std::tuple` 에 대해 살펴보겠습니다. C++ 에서 같은 타입 객체들을 여러개 다루기 위해서는 `std::vector` 나 배열을 사용하였습니다.

하지만 다른 타입의 객체들을 여러 개 다루는 방법은 꽤나 골치 아픕니다. 보통은 아래와 같이

```
struct Collection {
    int a;
    std::string s;
    double d;
};
```

간단히 구조체를 정의해서 전달하곤 합니다. 하지만 매번 이렇게 의미 없는 구조체를 생성하게 된다면 코드를 읽는 사람 입장에서 상당히 골치아픕니다. 파이썬과 같은 언어에서는 (1, 'abc', 3.14) 처럼 간단히 tuple 을 생성할 수 있는데 말이죠.

다행이도 C++ 11 부터 `std::tuple` 라이브러리가 추가되어서 간단히 서로 다른 타입들의 집합을 생성할 수 있습니다.

```
#include <iostream>
#include <string>
#include <tuple>

int main() {
    std::tuple<int, double, std::string> tp;
    tp = std::make_tuple(1, 3.14, "hi");

    std::cout << std::get<0>(tp) << ", " << std::get<1>(tp) << ", "
          << std::get<2>(tp) << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1, 3.14, hi
```

와 같이 잘 나옵니다.

```
std::tuple<int, double, std::string> tp;
```

`tuple` 을 정의하는 방법은 간단합니다. `tuple` 이 보관하고자 하는 타입들을 쭈르륵 나열해주면 됩니다. 위 `tp` 의 경우 `int`, `double`, `std::string` 이 세 개 타입의 객체를 보관하는 컨테이너라 생각하시면 됩니다.

참고로 `variant` 와는 다르게 `tuple` 에는 같은 타입들이 들어 있어도 전혀 문제가 될 것이 없습니다.

```
tp = std::make_tuple(1, 3.14, "hi");
```

`tuple` 객체를 생성하기 위해서는 `make_tuple` 함수를 사용하면 됩니다.

```
std::cout << std::get<0>(tp) << ", " << std::get<1>(tp) << ", "
<< std::get<2>(tp) << std::endl;
```

그리고 마지막으로 `tuple` 의 각각의 원소에 접근하기 위해서는 이전의 `variant` 처럼 `std::get` 을 이용하시면 됩니다. 이 때 `get` 에 템플릿 인자로 몇 번째 원소에 접근할지 지정해주면 됩니다.

참고로 원하는 타입의 원소를 뽑아내고 싶다면 타입을 전달해도 되는데, 예를 들어서 `std::get<std::string>` 을 하게 되면 `tuple` 에 정의된 문자열 객체가 뽑혀져 나오게 됩니다. 다만, `tuple` 에 `std::string` 이 없거나, 2 개 이상 존재한다면 예외가 발생하게 됩니다.

Structured binding (C++ 17 이상)

C++ 17 에서는 structured binding 이라는 테크닉이 추가되어서 `tuple` 을 좀 더 편리하게 다룰 수 있게 되었습니다. 예를 들어서 아래와 같은 상황을 생각해봅시다.

```
#include <iostream>
#include <string>
#include <tuple>

std::tuple<int, std::string, bool> GetStudent(int id) {
    if (id == 0) {
        return std::make_tuple(30, "철수", true);
```

```

} else {
    return std::make_tuple(28, "영희", false);
}
}

int main() {
    auto student = GetStudent(1);

    int age = std::get<0>(student);
    std::string name = std::get<1>(student);
    bool is_male = std::get<2>(student);

    std::cout << "이름 : " << name << std::endl;
    std::cout << "나이 : " << age << std::endl;
    std::cout << "남자 ? " << std::boolalpha << is_male << std::endl;
}

```

성공적으로 컴파일 하였다면

실행 결과

```

이름 : 영희
나이 : 28
남자 ? false

```

와 같이 잘 나옵니다. tuple에서 각각의 원소들을 뽑아내기 위해서는 아래와 같이 해야 합니다.

```

int age = std::get<0>(student);
std::string name = std::get<1>(student);
bool is_male = std::get<2>(student);

```

상당히 귀찮죠. 하지만 C++ 17 부터는 *structured binding*이라는 방식을 통해 아주 간단하게 표현할 수 있습니다.

```

#include <iostream>
#include <string>
#include <tuple>

std::tuple<int, std::string, bool> GetStudent(int id) {
    if (id == 0) {
        return std::make_tuple(30, "철수", true);
    } else {
        return std::make_tuple(28, "영희", false);
    }
}

```

```
int main() {
    auto student = GetStudent(1);

    auto [age, name, is_male] = student;

    std::cout << "이름 : " << name << std::endl;
    std::cout << "나이 : " << age << std::endl;
    std::cout << "남자 ? " << std::boolalpha << is_male << std::endl;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
이름 : 영희
나이 : 28
남자 ? false
```

와 같이 잘 나옵니다. structured binding 이 적용된 부분은 아래와 같습니다.

```
auto [age, name, is_male] = student;
```

마치 파이썬을 생각나게 하는 문법입니다. structured binding 을 사용하기 위해선

```
auto /* & 혹은 && 도 가능 */ /* tuple 안에 원소들을 받기 위한 객체 */ = tp;
```

와 같이 쓰면 됩니다. 자세한 내용은 [여기](#)에서 볼 수 있습니다.

예를 들어서 만약에 tuple 안에 객체들을 복사하지 않고 그냥 레퍼런스만 취하고 싶다면

```
auto& [age, name, is_male] = student;
```

와 같이 하면 됩니다.

한 가지 중요한 점은 tuple 의 모든 원소들을 반드시 받아야 한다는 점입니다. 안타깝게도 structured binding 을 사용해선 중간의 원소 하나만 빼고 받기와 같은 것은 할 수 없습니다.

그래도 structured binding 은 여러가지 쓰임새들이 매우 많습니다. 꼭 tuple 말고도, 데이터 멤버들이 정의되어 있는 구조체의 데이터 필드들을 받는데에도 사용할 수 있습니다. 예를 들어서

```
struct Data {
    int i;
    std::string s;
    bool b;
};

Data d;
auto [i, s, b] = d;
```

와 같이 각각의 데이터 필드를 받아낼 수 있습니다.

덕분에 `pair` 와 같은 클래스들 역시 structured binding 을 사용할 수 있습니다.

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<int, std::string> m = {{3, "hi"}, {5, "hello"}};
    for (auto& [key, val] : m) {
        std::cout << "Key : " << key << " value : " << val << std::endl;
    }
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Key : 3 value : hi
Key : 5 value : hello
```

와 같이 나옵니다. 기존에는 iterator 로 받아서 first 와 second 로 키와 대응되는 값을 나타내야 하지만 structured binding 을 사용해서 훨씬 깔끔하게 나타낼 수 있습니다.

자 그렇다면 이것으로 이번 강좌를 마치도록 하겠습니다. C++ 에서 제공하는 유용한 도구들을 사용해서 코드를 훨씬 깔끔하게 나타내보세요!

마무리

아무래도 이번 강좌가 제 마지막 C++ 강좌가 될 것 같습니다. 다음 글에서는 C++ 강의를 긴 시간 동안 작성해오면서 느낀 짧은 소회를 적어보자 합니다. (물론 그렇다고 해서 아예 끝난 것은

아닙니다. C++ 은 정말 빠르게 발전하고 있기 때문에 C++ 20 이 본격적으로 도입이 된다면 또 이야기할 주제들이 생각나겠죠!)

감사합니다.

뭘 배웠지?

- `optional` 을 통해 원하는 데이터를 가지거나 가지지 않는 객체를 만들 수 있습니다.
- `variant` 를 통해 여러 타입 들 중 하나를 나타내는 객체를 만들 수 있습니다.
- `tuple` 을 통해 여러 서로다른 타입들의 모음 을 나타내는 객체를 만들 수 있습니다.

강의를 마무리하면서

안녕하세요 여러분! 여기까지 오시느라 수고 많으셨습니다.

(C++ 전체 강의를 보시려면 [여기로](#) 가시면 됩니다)

2011년에 시작한 강좌를 드디어 2020년에 와서야 끝마치게 되었습니다. 긴 기간동안 중간에 공백도 있었고 (군대갔다오느라 흥...) 답글들도 자주 못달아 드렸는데, 그래도 제 글들을 읽어주시는 독자들의 성원 덕분에 어찌어찌해서 완결을 할 수 있었습니다. 강의를 완결하는데 오래걸린 만큼 틀린 내용도 많이 수정하고 내용도 보완해서 좀 더 탄탄한 강좌로 완성할 수 있었던 것 같습니다. 소중한 답글 달아주신 분들에 감사의 말을 표하고 싶습니다.

돌이켜보면 9년이라는 세월이 어떻게 보면 길면서도 또 어떻게 보면 또 금방 지나갔던 것 같습니다. 이 세월 동안 저 역시 학생에서 지금은 C++ 덕분에 밥먹고 사는 소프트웨어 엔지니어가 되었습니다.

C++ 역시 정말 많이 변했습니다. 2011년에 C++ 11이 출시되었지만, 아무래도 당시에는 C++ 98을 주로 쓰는 추세였죠. 하지만 세월이 흘러 이제는 C++ 20의 표준안이 완성되어서 일부 컴파일러에서는 사용할 수 있게 되었습니다.

제 강좌도 2011년 당시에 처음 시작할 때에는 주로 C++ 98/03의 내용만을 다루다가 강좌 후반에 C++ 11의 내용을 살짝 다루는 식으로 계획했었는데, 이제 C++ 11 혹은 그 이상인 C++ 14와 17이 대세가 된 이상 해당 내용도 모두 포함하는 식으로 계획을 바꾸게 되었습니다.

물론 C++ 20이 얼마 전에 공개되었지만 아직 해당 표준안을 전부 지원하는 컴파일러는 없기 때문에 실무에서 C++ 20을 사용하기에는 조금 더 시간이 걸릴 것으로 예상 됩니다.¹⁾

물론 제 C++ 강좌가 여기서 끝나는 것은 아닙니다! 일단 여태까지의 강좌를 통해서 C++의 기초적인 개념들을 다루었더라면, 후에는 좀 더 C++을 실제로 사용하는데 도움이 되는 내용들을 다루고 싶습니다. 제가 생각하고 있는 주제들로는

- [Makefile](#) 과 CMake 사용법

1) 참고로 구글에서 C++ 17이 표준으로 된 지는 채 1년이 지나지 않았습니다. 다시말해 새로운 표준안이 공개되어도 회사에서 이를 적용하는데에는 여러가지 호환성 문제와 겹침 문제 때문에 시간이 많이 걸립니다.

- C++ 툴들 (clang-format, clang-tidy, ASAN, TSAN 등등) 활용법.
- C++ 유닛 테스트 프레임워크 사용하기 (googletest 와 catch)
- C++ 디자인 패턴 (Singleton, Factory, CRTP, Dependency Injection 등등)
- C++ 20 에 등장할 `concept` 과 `module`
- 할당자 (allocator) 만들기

들이 있는데 추후에 다루어 보겠습니다.

그럼 이제 뭘 해야 되나?

여러분에 손에는 이제 C++ 이라는 강력한 무기가 주어졌습니다. 이제 이 무기를 휘두루기만 하면 되죠! 첫 번째 강의에서 이야기 하였지만 수 많은 프로그램과 라이브러리들이 C++ 로 만들어져 있는 만큼 여러분들도 이제 만들려가시면 됩니다 ㅎㅎ

만약에 C++ 을 좀 더 공부하고 싶다면 제가 추천하는 책과 영상들이 있습니다.

읽어볼만한 책들

- 일반적인 프로그래밍 : *The C++ Programming Language, 4th Edition*, *Effective Modern C++*
- C++ 템플릿을 좀 더 알고 싶다면 : *C++ Templates: The Complete Guide*
- C++ 동시성 프로그래밍 대해 좀 더 알고 싶다면 : *C++ Concurrency in Action, Second Edition*
- C++ 디자인 패턴을 좀 더 알고 싶다면 : *Modern C++ Design* 과 *Hands-On Design Patterns with C++*

참고로 Scott Meyers 의 *Effective Modern C++* 은 C++ 을 실전에서 사용하기 전에 한 번쯤 꼭 읽어봐야할 명서라고 생각합니다.

볼만한 영상들

CppCon 은 1 년에 한 번씩 열리는 C++ 컨퍼런스 인데, 해당 컨퍼런스에서 발표된 영상들이 모두 유튜브에 올라오니 관심 있는 주제로 영상을 보는 것을 추천드립니다. 특히 최신 C++ 트렌드를 따라가는데 도움이 많이 됩니다.

아래 영상들은 제가 인상적으로 보았던 영상들입니다.

- CppCon 2019: Bjarne Stroustrup "C++20: C++ at 40"
- CppCon 2019: Andrei Alexandrescu "Speed Is Found In The Minds of People"
- CppCon 2017: Fedor Pikus "C++ atomics, from basic to advanced. What do they really do?"
- CppCon 2014: Walter E. Brown "Modern Template Metaprogramming: A Compendium, Part I"
- CppCon 2017: Jason Turner "Practical C++17"
- CppCon 2017: Louis Brandy "Curiously Recurring C++ Bugs at Facebook"
- CppCon 2014: Chandler Carruth "Efficiency with Algorithms, Performance with Data Structures"
- CppCon 2019: Stephan T. Lavavej "Floating-Point <charconv>: Making Your Code 10x Faster With C++17's Final Boss"

사실 C++ 과 직접 관련은 없지만 매우 흥미로운 주제여서 추가하였습니다.

- CppCon 2018: Chandler Carruth "Spectre: Secrets, Side-Channels, Sandboxes, and Security"

그리고 Stephan Lavevej 의 표준 라이브러리에 관한 강의도 도움이 많이 되었습니다.

PDF 파일

현재 전체 강의 내용을 PDF 로 만들어서 책처럼 제작하고 있습니다. 사실 LaTeX 를 사용해서 거의 완성이 됬는데 몇몇 그림들이 짤리는 경우가 있어서 수정 중에 있습니다. 조만간 완성해서 올릴 예정입니다.

그럼 그동안 제 강좌를 봐주셔서 감사합니다!

C++ 개발자가 알면 좋을 것들

Make 사용 가이드 (Makefile 만들기)

안녕하세요. 이번 강좌부터는 씹어먹는 C++ 시리즈의 부록 과 같은 개념으로 C++ 언어 자체와는 직접 관련은 없지만 실제로 C++ 을 프로그래밍 하기 위해서 필요한 지식들과 툴들에 대해서 이야기하고자 합니다. 그 첫 번째 타자로 *Makefile* 만들기가 되겠습니다.

아무래도 리눅스 환경에서 프로그래밍을 하신 분들은 아시겠지만, 보통 프로그램을 컴파일 할 때 *make* 라는 프로그램을 사용합니다. 윈도우에서 비주얼 스튜디오를 사용하신다면 컴파일 버튼을 누르면 알아서 컴파일 되는 것과는 달리, 셸 상에서 컴파일을 하려면 어떤 파일들을 컴파일 하고, 어떠한 방식으로 컴파일 할지 직접 컴파일러에게 알려줘야 합니다.

물론 매번 명령어를 치면 문제가 없겠지만 프로젝트의 크기가 커지고 파일들이 많아진다면 매번 그렇게 친다는 것이 불가능에 가까워집니다. 이 문제를 해결하기 위해서 리눅스에서는 *make* 라는 프로그램을 제공하는데, 이 프로그램은 *Makefile* 라는 파일을 읽어서 주어진 방식대로 명령어를 처리하게 합니다. 덕분에 많은 수의 파일들을 명령어 한 번으로 컴파일 할 수 있습니다.

이번 글에서는 *make* 프로그램이 어떠한 방식으로 작동되고, 프로그램들을 우리가 원하는 방식으로 컴파일 하기 위해서는 어떠한 방식으로 *Makefile* 을 작성해야 하는지 살펴보도록 하겠습니다.

소스 코드에서 완성된 프로그램까지

먼저 *make* 가 어떠한 방식으로 작동하는지 알기 전에 컴파일러를 통해 프로그램을 빌드하게 되면 어떠한 방식으로 소스 코드에서 하나의 프로그램이 완성되는지 살펴보도록 합시다.

컴파일 (Compile)

가장 먼저 여러분의 머리 속으로 떠오르는 과정은 바로 컴파일(Compile) 일 것입니다. 이 컴파일이라는 과정은 여러분의 소스 코드를 컴퓨터가 이해할 수 있는 어셈블리어로 변환하는 과정입니다.

예를 들어서 아래와 같이 `foo.h`에 정의된 `foo` 함수와 `bar.h`에 정의된 `bar` 함수를 `main` 함수에서 호출하는 간단한 프로그램을 살펴봅시다.

- `foo.h`

```
int foo();
```

- `foo.cc`

```
#include <iostream>

#include "foo.h"

int foo() {
    std::cout << "Foo!" << std::endl;
    return 0;
}
```

- `bar.h`

```
int bar();
```

- `bar.cc`

```
#include <iostream>

#include "bar.h"

int bar() {
    std::cout << "Bar!" << std::endl;
    return 0;
}
```

- `main.cc`

```
#include "bar.h"
#include "foo.h"
```

```
int main() {
    foo();
    bar();
}
```

예를 들어서 `main.cc` 를 컴파일 하기 위해서는 아래와 같이 하면 됩니다.¹⁾

```
$ g++ -c main.cc
```

`g++` 에 전달하는 `-c` 명령어는 인자 다음에 주어지는 파일을 컴파일 해서 목적 파일(object file)을 생성하라는 의미입니다.

실제로, 성공적으로 컴파일 하였다면 아래와 같이 `main.o` 라는 파일이 생성된 것을 알 수 있습니다.

```
$ ls main.o
main.o
```

이 `main.o` 는 `main.cc` 를 컴파일 한 어셈블리 코드가 담겨있는 파일입니다. 실제로 `main.o` 를 뜯어보면

```
$ objdump -S main.o
main.o:      file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <main>:
 0:   f3 0f 1e fa          endbr64
 4:   55                   push    %rbp
 5:   48 89 e5             mov     %rsp,%rbp
 8:   e8 00 00 00 00        callq   d <main+0xd>
 d:   e8 00 00 00 00        callq   12 <main+0x12>
12:   b8 00 00 00 00        mov     $0x0,%eax
17:   5d                   pop    %rbp
18:   c3                   retq
```

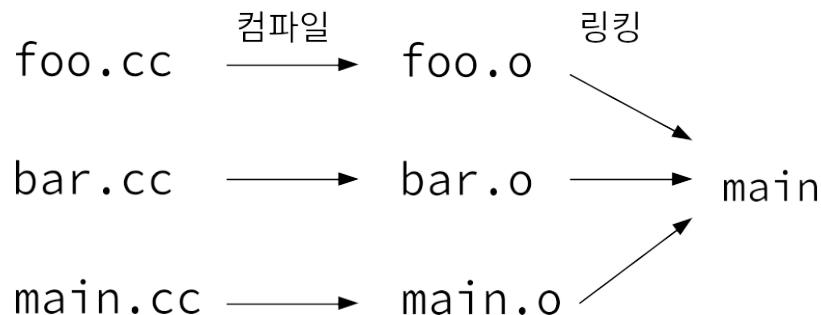
1) 만일 `clang` 을 사용하고 싶다면 `g++` 자리에 `clang` 을 넣으시면 됩니다. `clang` 과 `g++` 은 사용 명령어가 거의 똑같습니다.

와 같이 `main.cc` 의 어셈블리 코드가 잘 들어있음을 알 수 있습니다.²⁾

한 가지 재미있는 점은 컴파일러가 `main.cc` 를 컴파일 할 때 다른 파일들을 참조하지 않았다는 점입니다. 실제로 `main.o` 의 어셈블리를 슬쩍 보면 `foo` 나 `bar` 에 관한 내용이 하나도 없음을 알 수 있습니다.

따라서 `main.o` 자체로는 우리의 프로그램을 만들 수 없습니다. 왜냐하면 `main.o`에는 `foo` 라는 함수를 호출해라! 라는 내용만 있지, `foo` 는 어디에 있고 이러한 방식으로 동작한다에 관한 이야기는 없기 때문이죠.³⁾ 실제로 작동하는 `foo`에 관한 정보를 얻기 위해서는 `foo.cc` 를 컴파일 해서 만들어진 `foo.o` 가 필요하고, 마찬가지로 `bar`에 관한 정보를 얻기 위해서는 `bar.cc` 를 컴파일 해서 만든 `bar.o` 가 필요합니다.

따라서 실제 프로그램을 제작하기 위해서는 이 각각의 `main.o`, `foo.o`, `bar.o` 를 하나로 합치는 과정이 필요하겠죠. 이와 같은 과정을 아래 그림과 같이 링킹(Linking) 이라고 합니다.



링킹 (Linking)

링킹이 이름 그대로 링크 하는 작업인 이유는 실제로 서로 다른 파일에 흩어져 있던 함수나 클래스들을 한 데 묶어서 링크해주는 작업이기 때문이죠. 이 과정에서 `main` 함수 안에 `foo` 함수가 어디에 정의되어 있는지 위치를 찾고 제대로 된 함수를 호출할 수 있게 됩니다. 사실 이 링킹에 대해서 이야기만 해도 한참 할 수 있는데, 일단 이 글의 목표는 빌드 파일 만들기 이므로 여기서 줄이겠습니다.

아무튼 이 링킹 과정은 아래와 같이 컴파일러에 목적파일을 전달함으로써 수행됩니다.

```
$ g++ main.o foo.o bar.o -o main
```

여기서 `-o` 옵션은 그 뒤에 링킹 후에 생성할 실행 파일 이름을 이야기 합니다. 위 경우 `main` 이라는 실행 파일을 만들었고 만약에 이를 따로 지정하지 않는다면 그냥 `./a.out` 이라는 파일을 디폴트로 생성하게 됩니다.

2) 위 어셈블리가 뭔지 이해가 전혀 안가도 Make 가 뭔지 이해하는데 전혀 상관 없습니다. 그냥 '목적파일에는 소스 코드를 컴파일 한 어셈블리가 들어 있구나' 정도로 이해하시면 됩니다.

3) 눈썰미가 좋으신 분들은 위 어셈블리의 `callq` 자리에 함수 이름 대신에 이상한 값이 들어있음을 알 수 있습니다.

실제로 `main` 파일을 실행하게 된다면

실행 결과

Foo!

Bar!

와 같이 잘 나옵니다.

그렇다면 왜 `make` 를 쓰는지?

일단 위와 같이 `main` 실행 파일을 생성하기 위해서 입력한 쉘 명령어를 쭉 나열해보자면;

```
$ g++ -c main.cc  
$ g++ -c foo.cc  
$ g++ -c bar.cc  
$ g++ main.o foo.o bar.o -o main
```

와 같습니다. 물론 파일 개수가 작다면 매 번 입력하는 것이 크게 문제가 되지는 않습니다만, 프로젝트 크기가 커지면 쳐야할 명령어가 더 많아지게 되서 복잡하겠지요 (특히 파일들이 다른 디렉토리에 있는다면 말이죠.)

물론 쉘 스크립트를 조금 아시는 분이라면

그냥 위 명령어들을 순차적으로 실행하는 쉘 스크립트를 짜면 되는거 아닌가?

라고 물으실 수 있습니다. 그런데 이 방식으로 한다면 한 가지 문제점이 있는데, 예를 들어서 여러분이 `main.cc` 를 수정하였다고 해봅시다.

그렇다면 실제로는

```
$ g++ -c main.cc  
$ g++ main.o foo.o bar.o -o main
```

딱 이 두 명령어만 실행하면 됩니다. 왜냐하면 `main` 파일을 바꾼다고 해서 `foo` 나 `bar` 의 컴파일 된 내용이 바뀌지 않기 때문이죠. 하지만 위처럼 단순하게 쉘 스크립트를 짜게 된다면 파일 하나만 바꿔도 전체 모든 파일들을 컴파일 하게 되므로 컴파일 시간이 매우 길어지게 됩니다. 하지만 `make` 를 사용하면 일부 파일을 수정할 경우 필요한 명령만 빠르게 컴파일을 할 수 있도록 도와줍니다.

자 그렇다면 거두절미하고 `make` 의 사용법과 `make` 를 위한 `Makefile` 을 어떻게 작성하는지 알아보겠습니다.

make

make 는 간단히 말하자면 주어진 쉘 명령어들을 조건에 맞게 실행하는 프로그램이라고 볼 수 있습니다. 이 때 어떠한 조건으로 명령어를 실행할지 담은 파일을 *Makefile*이라고 부르며, *make* 를 터미널 상에서 실행하게 된다면 해당 위치에 있는 *Makefile* 을 찾아서 읽어들이게 됩니다.

그렇다면 *Makefile* 에는 어떠한 방식으로 조건을 기술할까요?

```
target ... : prerequisites ...
(탭) recipe
...
...
```

Makefile 은 기본적으로 위와 같이 3 가지 요소로 구성되어 있습니다.

target

make 를 실행할 때 *make abc* 과 같이 어떠한 것을 *make* 할 지 전달하게 되는데 이를 타겟(target)이라고 부릅니다. 만일 *make abc* 를 하였을 경우 타겟 중에 abc 를 찾아서 이에 대응되는 명령을 실행해줍니다.

실행할 명령어(recipes)

주어진 타겟을 *make* 할 때 실행할 명령어들의 나열입니다. 한 가지 중요한 점은 recipe 자리에 명령어를 쓸 때 반드시 텁 한 번으로 들여쓰기를 해줘야만 합니다. 보통 요즘의 편집기의 경우 (예를 들어 VSCode), 자동으로 텁을 스페이스로 바꿔주는 옵션이 활성화 되어 있을 텐데 *make* 가 *Makefile* 을 제대로 읽어들이기 위해서는 반드시 텁을 사용해야만 합니다!

필요 조건들(prerequisites)

주어진 타겟을 *make* 할 때 사용될 파일들의 목록입니다. 다른 말로 의존 파일(dependency) 이라고도 합니다. 왜냐하면 해당 타겟을 처리하기 위해 건드려야 할 파일들을 써놓은 것이기 때문입니다.

만일 주어진 파일들의 수정 시간 보다 타겟이 더 나중에 수정되었다면 해당 타겟의 명령어를 실행하지 않습니다. 왜냐하면 이미 이전에 타겟이 만들어져있다고 간주하기 때문이죠.

예를 들어서 타겟이 *main.o* 이고, 명령어가 *g++ -c main.cc* 라면, 필요 조건들은 *main.cc*, *foo.h*, *bar.h* 가 되겠죠. 왜냐하면 이들 파일들 중 하나라도 바뀐다면 *main* 을 새로 컴파일 해야하기 때문이죠. 반면에 *main.o* 의 생성 시간이 *main.cc*, *foo.h*, *bar.h* 들의 마지막 수정 시간보다 나중이라면, 굳이 *main.o* 를 다시 컴파일 할 필요가 없습니다.

그렇다면 위 내용을 바탕으로 *Makefile* 을 간단하게 구성해보겠습니다.

```
foo.o : foo.h foo.cc  
g++ -c foo.cc  
  
bar.o : bar.h bar.cc  
g++ -c bar.cc  
  
main.o : main.cc foo.h bar.h  
g++ -c main.cc  
  
main : foo.o bar.o main.o  
g++ foo.o bar.o main.o -o main
```

주의 사항

위 *Makefile* 을 그대로 복사해서 실행한다면 아마 실행이 안될 것입니다. 왜냐하면 템이 스페이스로 바뀌어서 그런데요, 제대로 *Makefile* 을 만드시기 위해서는 `g++` 앞에 오는 스페이스 두 번을 템으로 바꾸어서 저장해보세요.

우리의 목표는 실행 파일인 `main` 을 만드는 것이기 때문에 `make main` 을 실행해보면

```
$ make main  
g++ -c foo.cc  
g++ -c bar.cc  
g++ -c main.cc  
g++ foo.o bar.o main.o -o main
```

그리고 실제로 `main` 이 잘 만들어짐을 알 수 있습니다.

여러분이 한 번도 컴파일 하지 않은 상태에서 `make main` 을 실행하면 `make` 파일은 다음과 같은 순서로 명령어를 처리합니다.

1. `make main` 이니까 *Makefile* 에서 타겟이 `main` 인 녀석을 찾아보자.
2. 오. `main` 의 필요한 파일들이 `foo.o bar.o main.o` 이네? 이들 파일을 어떻게 만드는지 각각의 파일 이름으로된 타겟들을 찾아보자.
3. `foo.o` 의 경우 필요한 파일이 `foo.cc` 네? 아직 `foo.o` 가 없으니까 주어진 명령어를 실행 해야 하겠군!

4. 마찬가지로 `bar.o`, `main.o` 도 컴파일
5. 자 이제 마지막으로 `g++ foo.o bar.o main.o -o main` 를 실행해야지

만약에 여러분이 `make main` 을 한 번 실행한 상태에서 `foo.cc` 만 수정하였다고 해봅시다. 그렇다면 아래와 같이 필요한 부분만 재컴파일 됩니다.

1. `make main` 이니까 *Makefile* 에서 타겟이 `main` 인 녀석을 찾아보자.
2. 오. `main` 의 필요한 파일들이 `foo.o` `bar.o` `main.o` 이네? 이들 파일을 어떻게 만드는지 각각의 파일 이름으로된 타겟들을 찾아보자.
3. `foo.o` 의 경우 필요한 파일이 `foo.cc` 네? 그런데 `foo.o` 의 생성 시간 보다 `foo.cc` 의 수정 시간이 더 나중이군! 주어진 명령어를 다시 실행해야겠어
4. `bar.o` 의 경우 필요한 파일이 `bar.cc` 인데, `bar.o` 의 생성 시간이 `bar.cc` 의 수정 시간 보다 나중이네. 굳이 명령어들을 실행하지 않아도 되겠어
5. `main.o` 도 마찬가지로 다시 안바꿔도 되겠군!
6. `main` 의 필요한 파일들 중 `foo.o` 가 바뀌었으니 내 명령어들도 다시 실행해야겠어.

따라서 딱 필요한 `g++ -c foo.cc` 와 `g++ foo.o bar.o main.o -o main` 만이 실행됩니다.

변수

재미있게도 *Makefile* 내에서 변수를 정의할 수 있습니다.

```
CC = g++
```

위 경우 `CC` 라는 변수를 정의하였는데, 이제 *Makefile* 내에서 `CC` 를 사용하게 된다면 해당 변수의 문자열인 `g++` 로 치환됩니다. 이 때 변수를 사용하기 위해서는 `$(CC)` 와 같이 `$()` 안에 사용하고자 하는 변수의 이름을 지정하면 됩니다. 예를 들어서

```
CC = g++
```

```
foo.o : foo.h foo.cc  
$(CC) -c foo.cc
```

는 사실

```
CC = g++
foo.o : foo.h foo.cc
g++ -c foo.cc
```

과 정확히 같은 명령이 됩니다.

참고로 정의하지 않는 변수를 참조하게 된다면 그냥 빈 문자열로 치환됩니다.

변수를 정의하는 두 가지 방법

주의 사항

이 부분은 TMI 이므로 바쁘신 분들은 그냥 넘어가셔도 됩니다.

Makefile 상에서 변수를 정의하는 방법으로 `=` 를 사용해서 정의하는 방법과 `:=` 를 사용해서 정의하는 방법이 있습니다. 이 둘은 살짝 다릅니다.

`=` 를 사용해서 변수를 정의하였을 때, 정의에 다른 변수가 포함되어 있다면 해당 변수가 정의되기 될 때 까지 변수의 값이 정해지지 않습니다. 예를 들어서

```
B = $(A)
C = $(B)
A = a
```

C 의 경우 B 의 값을 참조하고 있고, B 의 경우 A 의 값을 참조하고 있습니다. 하지만 `B = $(A)` 를 실행한 시점에서 A 가 정의되지 않았으므로 B 는 그냥 빈 문자열이 되어야 하지만 `=`로 정의하였기 때문에 A 가 실제로 정의될 때 까지 B 와 C 가 결정되지 않습니다. 결국 마지막에 `A = a` 를 통해 A 가 a 로 대응되어야, C 가 a 로 결정됩니다.

```
B := $(A)
A = a
```

반면에 `:=` 로 변수를 정의할 경우, 해당 시점에의 변수의 값만 확인 합니다. 따라서 위 경우 B 는 그냥 빈 문자열이 되겠지요.

대부분의 상황에서는 `=` 나 `:=` 중 아무거나 사용해도 상관 없습니다. 하지만

- 만일 변수들의 정의 순서에 크게 구애받고 싶지 않다면 `=` 를 사용하는 것이 편합니다.

- `A = $(A)` `b` 와 같이 자기 자신을 수정하고 싶다면 `:=` 를 사용해야지 무한 루프를 피할 수 있습니다.

만 명심하면 됩니다.

그렇다면 이들 변수를 사용해서 *Makefile* 을 조금 더 깔끔하게 바꾸어보겠습니다.

```
CC = g++
CXXFLAGS = -Wall -O2
OBJS = foo.o bar.o main.o

foo.o : foo.h foo.cc
$(CC) $(CXXFLAGS) -c foo.cc

bar.o : bar.h bar.cc
$(CC) $(CXXFLAGS) -c bar.cc

main.o : main.cc foo.h bar.h
$(CC) $(CXXFLAGS) -c main.cc

main : $(OBJS)
$(CC) $(CXXFLAGS) $(OBJS) -o main
```

make 를 실행해보면

```
$ make main
g++ -Wall -O2 -c foo.cc
g++ -Wall -O2 -c bar.cc
g++ -Wall -O2 -c main.cc
g++ -Wall -O2 foo.o bar.o main.o -o main
```

와 같이 잘 나옵니다.

`CC` 와 `CXXFLAGS` 는 *Makefile* 에서 자주 사용되는 변수로 보통 `CC` 에는 사용하는 컴파일러 이름을, `CXXFLAGS` 에는 컴파일러 옵션을 주는 것이 일반적입니다. 참고로 이는 C++ 의 경우이고, C 의 경우 `CFLAGS` 에 옵션을 줍니다.

우리의 경우 `g++` 컴파일러를 사용하며, 옵션으로는 `Wall` (모든 컴파일 경고를 표시) 과 `O2` (최적화 레벨 2) 를 주었습니다.⁴⁾

4) 사실 `-Wall` 은 이름과는 다르게 모든 경고를 표시하지 않습니다.

PHONY

*Makefile*에 흔히 추가하는 기능으로 빌드 관련된 파일들 (.o 파일들)을 모두 제거하는 명령을 넣습니다.

```
clean:
    rm -f $(OBJS) main
```

실제로 `make clean`을 실행해보면 생성된 모든 목적 파일과 `main`을 지워버림을 알 수 있습니다.

그런데, 만약에 실제로 `clean`이라는 파일이 디렉토리에 생성된다면 어떨까요? 우리가 `make clean`을 하게 되면, `make`는 `clean`의 필요 파일들이 없는데, `clean` 파일이 있으니까 `clean` 파일은 항상 최신이네? *recipe*를 실행 안해도 되겠네! 하면서 그냥 `make clean` 명령을 무시해버리게 됩니다.

```
$ ls clean
clean
$ make clean
make: 'clean' is up to date.
```

실제로 디렉토리에 `clean`이라는 파일을 만들어놓고 실행해보면 위와 같이 이미 `clean`은 최신이라며 *recipe* 실행을 거부합니다.

이와 같은 상황을 막기 위해서는 `clean`을 PHONY라고 등록하면 됩니다.⁵⁾ 아래와 같이 말이지요.

```
.PHONY: clean
clean:
    rm -f $(OBJS) main
```

이제 `make clean`을 하게 되면 `clean` 파일의 유무와 상관 없이 언제나 해당 타겟의 명령을 실행하게 됩니다.

패턴 사용하기

우리의 경우 파일이 3개 밖에 없어서 다행이였지만 실제 프로젝트에는 수십~수백 개의 파일들을 다루게 될 것입니다. 그런데, 각각의 파일들에 대해서 모두 빌드 방식을 명시해준다면 *Makefile*의

5) Phony는 '가짜의, 허위의'라는 뜻입니다.

크기가 엄청 커지겠지요.

다행이도 *Makefile* 에서는 패턴 매칭을 통해서 특정 조건에 부합하는 파일들에 대해서 간단하게 *recipe* 를 작성할 수 있게 해줍니다.

```
foo.o : foo.h foo.cc
$(CC) $(CXXFLAGS) -c foo.cc

bar.o : bar.h bar.cc
$(CC) $(CXXFLAGS) -c bar.cc
```

일단 먼저 비슷하게 생긴 위 두 명령들을 어떻게 하면 하나로 간단하게 나타낼 수 있는지 보겠습니다.

```
%.o: %.cc %.h
$(CC) $(CXXFLAGS) -c $<
```

먼저 %.o 는 와일드카드로 따지면 마치 *.o 와 같다고 볼 수 있습니다. 즉, .o 로 끝나는 파일 이름들이 타겟이 될 수 있겠지요. 예를 들어서 foo.o 가 타겟이라면 % 에는 foo 가 들어갈 것이고 bar.o 의 경우 % 에는 bar 가 들어갈 것입니다.

따라서 예를 들어 foo.o 가 타겟일 경우

```
foo.o: foo.cc foo.h
$(CC) $(CXXFLAGS) -c $<
```

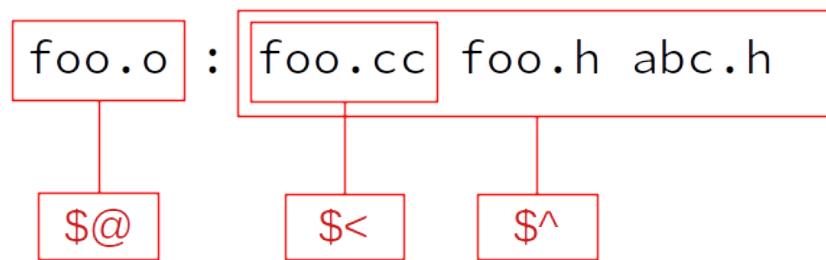
가 됩니다. 참고로 패턴은 타겟과 *prerequisite* 부분에만 사용할 수 있습니다. *recipe* 부분에서는 패턴을 사용할 수 없습니다. 따라서 컴파일러에 foo.cc 를 전달하기 위해서는 *Makefile* 의 자동 변수를 사용해야 합니다.

\$< 의 경우 *prerequisite* 에서 첫 번째 파일의 이름에 대응되어 있는 변수입니다. 위 경우 foo.cc 가 되겠지요. 따라서 위 명령어는 결과적으로

```
foo.o: foo.cc foo.h
$(CC) $(CXXFLAGS) -c foo.cc
```

가 되어서 이전의 명령어와 동일하게 만들어냅니다.

Makefile 에서 제공하는 자동 변수로는 그 외에도 아래 그림과 같이 \$@, \$<, \$* 등등이 있습니다.



- \$@ : 타겟 이름에 대응됩니다.
- \$< : 의존 파일 목록에 첫 번째 파일에 대응됩니다.
- \$: 의존 파일 목록 전체에 대응됩니다.
- \$? : 타겟 보다 최신인 의존 파일들에 대응됩니다.
- \$+ : \$ 와 비슷하지만, 중복된 파일 이름들 까지 모두 포함합니다.

하지만 애석하게도 위 패턴으로는

```

main.o : main.cc foo.h bar.h
$(CC) $(CXXFLAGS) -c main.cc
  
```

를 표현하기에는 부족합니다. 왜냐하면 의존 파일 목록에 main.h 가 없고 foo.h 와 bar.h 가 있기 때문이죠. 사실 곰곰히 생각해보면 이 의존파일 목록에는 해당 소스 파일이 어떠한 헤더파일을 포함하느냐에 결정되어 있습니다. main.cc 가 foo.h 와 bar.h 를 include 하고 있기 때문에 main.o 의 prerequisite 로 main.cc 외에도 foo.h 와 bar.h 가 들어가 있는 것입니다.

물론 매번 이렇게 일일히 추가할 수 있겠지만, 소스 파일에 헤더 파일을 추가할 때마다 Makefile 을 바꿀 수는 없는 노릇이니까요. 하지만 다행이도 컴파일러의 도움을 받아서 의존파일 목록 부분을 작성할 수 있습니다.

자동으로 prerequisite 만들기

컴파일 시에 -MD 옵션을 추가해서 컴파일 해봅시다.

```
$ g++ -c -MD main.cc
```

그렇다면 main.d 라는 파일을 생성합니다. 파일 내용을 살펴보면;

```
$ cat main.d
main.o: main.cc /usr/include/stdc-predef.h foo.h bar.h
```

놀랍게도 마치 *Makefile*의 `target: prerequisite` 인것 같은 부분을 생성하였습니다. 그렇습니다. 컴파일 시에 `-MD` 옵션을 추가해주면, 목적 파일 말고도 컴파일 한 소스파일을 타겟으로 하는 의존파일 목록을 담은 파일을 생성해줍니다.

참고로 `main.cc`, `foo.h`, `bar.h` 까지는 이해가 가는데 왜 생뚱맞은 `/usr/include/stdc-predef.h` 이 들어가 있느냐고 물을 수 있는데, 이 파일은 컴파일러가 컴파일 할 때 암묵적으로 참조하는 헤더 파일이라고 보시면 됩니다. 아무튼 이 때문에 컴파일러가 생성한 의존 파일 목록에는 포함되었습니다.

문제는 이렇게 생성된 `main.d` 를 어떻게 우리의 *Makefile*에 포함할 수 있느냐입니다. 이는 생각보다 간단합니다.

```
CC = g++
CXXFLAGS = -Wall -O2
OBJS = foo.o bar.o main.o

%.o: %.cc %.h
$(CC) $(CXXFLAGS) -c $<

main : $(OBJS)
$(CC) $(CXXFLAGS) $(OBJS) -o main

.PHONY: clean
clean:
rm -f $(OBJS) main

include main.d
```

위 `include main.d` 는 `main.d`라는 파일의 내용을 *Makefile*에 포함하라는 의미입니다.

그렇다면 아싸리

```
%.o: %.cc %.h
$(CC) $(CXXFLAGS) -c $<
```

부분을 아예 컴파일러가 생성한 `.d` 파일로 대체할 수는 없을까요? 물론 있습니다!

```
CC = g++
CXXFLAGS = -Wall -O2
```

```

OBJS = foo.o bar.o main.o

%.o: %.cc
    $(CC) $(CXXFLAGS) -c $<

main : $(OBJS)
    $(CC) $(CXXFLAGS) $(OBJS) -o main

.PHONY: clean
clean:
    rm -f $(OBJS) main

#include $(OBJS:.o=.d)

```

`$(OBJS:.o=.d)` 부분은 `OBJS`에서 `.o`로 끝나는 부분을 `.d`로 모두 대체하라는 의미입니다. 즉, 해당 부분은 `-include foo.d bar.d main.d`가 되겠죠. 참고로 `foo.d`나 `bar.d`가 `include` 될 때 이미 있는 `%.o: %.cc`는 어떻게 되냐고 물을 수 있는데 같은 타겟에 대해서 여러 의존 파일 목록들이 정해져 있다면 이는 `make`에 의해 모두 하나로 합쳐집니다. 따라서 크게 걱정 하실 필요는 없습니다.

덧붙여 `include`에서 `-include`로 바꾸었는데, `-include`의 경우 포함하고자 하는 파일이 존재하지 않아도 `make` 메세지를 출력하지 않습니다.

맨 처음에 `make`를 할 때에는 `.d` 파일들이 제대로 생성되지 않은 상태이기 때문에 `include` 가 아무런 `.d` 파일들을 포함하지 않습니다. 물론 크게 문제 없는 것이 어차피 `.o` 파일들도 `make` 가 `%.o: %.cc` 부분의 명령어들을 실행하면서 컴파일을 하기 때문에 다음에 `make`를 하게 될 때에는 제대로 `.d` 파일들을 로드할 수 있겠죠.

최종 정리

아래와 같이 간단한 프로젝트 구조를 생각해봅시다.

```

$ tree
.
├── Makefile
└── obj
    └── src
        ├── bar.cc
        └── bar.h

```

```
├── foo.cc  
├── foo.h  
└── main.cc
```

모든 소스 파일은 `src`에 들어가고 빌드 파일들은 `obj`에 들어갑니다. 종종 헤더 파일들을 따로 `include`에 빼는 경우가 있는데 저는 굳이 라이브러리를 만드는 경우가 아니라면 별로 선호하지 않습니다. (굳이 다른 디렉토리에 넣을 이유가 뭔지 모르겠습니다.)

아무튼 이와 같은 구조에서 항상 사용할 수 있는 만능 `Makefile`은 아래와 같습니다.

주의 사항

복사한 후에 `$(CC)` 와 `rm` 앞에 스페이스 두 개를 꼭 TAB으로 치환해주세요! 안 그러면 `make` 가 읽지 못합니다.

```
CC = g++  
  
# C++ 컴파일러 옵션  
CXXFLAGS = -Wall -O2  
  
# 링커 옵션  
LDFLAGS =  
  
# 소스 파일 디렉토리  
SRC_DIR = ./src  
  
# 오브젝트 파일 디렉토리  
OBJ_DIR = ./obj  
  
# 생성하고자 하는 실행 파일 이름  
TARGET = main  
  
# Make 할 소스 파일들  
# wildcard로 SRC_DIR에서 *.cc로 된 파일들 목록을 뽑아낸 뒤에  
# notdir로 파일 이름만 뽑아낸다.  
# (e.g SRCS는 foo.cc bar.cc main.cc가 된다.)  
SRCS = $(notdir $(wildcard $(SRC_DIR)/*.cc))  
  
OBJS = $(SRCS:.cc=.o)
```

```

# OBJS 안의 object 파일들 이름 앞에 $(OBJ_DIR) / 을 붙인다.
OBJECTS = $(patsubst %.o,$(OBJ_DIR)/%.o,$(OBJS))
DEPS = $(OBJECTS:.o=.d)

all: main

$(OBJ_DIR)/%.o : $(SRC_DIR)/%.cc
    $(CC) $(CXXFLAGS) -c $< -o $@ -MD $(LDFLAGS)

$(TARGET) : $(OBJECTS)
    $(CC) $(CXXFLAGS) $(OBJECTS) -o $(TARGET) $(LDFLAGS)

.PHONY: clean all
clean:
    rm -f $(OBJECTS) $(DEPS) $(TARGET)

-include $(DEPS)

```

추가된 부분만 간단히 부연 설명을 하자면

```

# Make 할 소스 파일들
# wildcard 로 SRC_DIR 에서 *.cc 로 된 파일들 목록을 뽑아낸 뒤에
# notdir 로 파일 이름만 뽑아낸다.
# (e.g SRCS 는 foo.cc bar.cc main.cc 가 된다.)
SRCS = $(notdir $(wildcard $(SRC_DIR)/*.cc))

```

먼저 SRC_DIR 안에 있는 모든 파일들을 SRCS로 읽어들이려 하고 있습니다. wildcard는 함수로 해당 조건에 맞는 파일들을 뽑아내게 되는데, 예를 들어서 foo.cc, bar.cc, main.cc 가 있을 경우 \$(wildcard \$(SRC_DIR)/*.cc) 의 실행 결과는 ./src/foo.cc ./src/bar.cc ./src/main.cc 가 될 것입니다.

여기서 우리는 foo.cc bar.cc main.cc 로 깔끔하게 경로를 제외한 파일 이름만 뽑아내기 위해 nodir 함수를 사용합니다. nodir은 앞에 오는 경로를 날려버리고 파일 이름만 깔끔하게 추출해 줍니다.

```
OBJS = $(SRCS:.cc=.o)
```

따라서 이 부분에서 OBJS 는 foo.o bar.o main.o 가 될 것입니다.

이제 이 OBJS 를 바탕으로 실제 .o 파일들의 경로를 만들어내고 싶습니다. 이를 위해서는 이들 파일 이름 앞에 \$(OBJ_DIR) / 을 붙여줘야 겠지요. 이를 위해선 patsubst 함수를 사용하면 됩니다.

```
# OBJS 안의 object 파일들 이름 앞에 $(OBJ_DIR) / 을 붙인다.
OBJECTS = $(patsubst %.o,$(OBJ_DIR)/%.o,$(OBJS))
```

patsubst 함수는 \$(patsubst 패턴, 치환 후 형태, 변수) 의 같은 꼴로 사용합니다.

따라서 위 경우 **\$(OBJS)** 안에 있는 모든 **%.o** 패턴을 **\$(OBJ_DIR)/%.o** 로 치환해라 라는 의미가 될 것입니다. 아무튼 덕분에 OBJECTS 에는 이제 ./obj/foo.o ./obj/bar.o ./obj/main.o 가 들어가게 됩니다.

그 뒤에 내용은 앞의 글을 잘 따라 오신 분들이라면 잘 이해 하실 수 있으리라 믿습니다.

헤더 파일들을 따로 뽑는 경우

만약에 헤더 파일들만 따로 뽑는다면 아래와 같은 파일 구조를 가지겠죠.

```
$ tree
.
├── include
│   ├── bar.h
│   └── foo.h
├── Makefile
└── obj
    └── src
        ├── bar.cc
        ├── foo.cc
        └── main.cc
```

이 경우 *Makefile* 을 아래와 같이 간단히 수정하면 됩니다.

```
CC = g++

# C++ 컴파일러 옵션
CXXFLAGS = -Wall -O2

# 링커 옵션
```

```
LDFLAGS =  
  
# 헤더파일 경로  
INCLUDE = -Iinclude/  
  
# 소스 파일 디렉토리  
SRC_DIR = ./src  
  
# 오브젝트 파일 디렉토리  
OBJ_DIR = ./obj  
  
# 생성하고자 하는 실행 파일 이름  
TARGET = main  
  
# Make 할 소스 파일들  
# wildcard 로 SRC_DIR 에서 *.cc 로 된 파일들 목록을 뽑아낸 뒤에  
# notdir 로 파일 이름만 뽑아낸다.  
# (e.g SRCS 는 foo.cc bar.cc main.cc 가 된다.)  
SRCS = $(notdir $(wildcard $(SRC_DIR)/*.cc))  
  
OBJS = $(SRCS:.cc=.o)  
DEPS = $(SRCS:.cc=.d)  
  
# OBJS 안의 object 파일들 이름 앞에 $(OBJ_DIR) / 을 붙인다.  
OBJECTS = $(patsubst %.o,$(OBJ_DIR)/%.o,$(OBJS))  
DEPS = $(OBJECTS:.o=.d)  
  
all: main  
  
$(OBJ_DIR)/%.o : $(SRC_DIR)/*.cc  
    $(CC) $(CXXFLAGS) $(INCLUDE) -c $< -o $@ -MD $(LDFLAGS)  
  
$(TARGET) : $(OBJECTS)  
    $(CC) $(CXXFLAGS) $(OBJECTS) -o $(TARGET) $(LDFLAGS)  
  
.PHONY: clean all  
clean:
```

```
rm -f $(OBJECTS) $(DEPS) $(TARGET)  
-include $(DEPS)
```

사실 기존과 별 차이 없고 그냥 컴파일러 옵션에 `-Iinclude/` 를 추가해주면 됩니다. 여기서 `include` 는 헤더파일 경로입니다.

멀티 코어를 활용해서 Make 속도를 올리자

한 가지 팁으로 그냥 `make` 를 실행하게 되면 1 개의 쓰레드만 실행되어서 속도가 꽤나 느립니다. 특히 GCC 나 커널을 컴파일 할 경우 한 두 시간은 그냥 걸리지요. 만일 여러분의 컴퓨터가 멀티 코어 CPU 를 사용한다면 (아마 대부분 그럴 것이라 생각합니다) `make` 를 여러 개의 쓰레드에서 돌릴 수 있습니다. 이를 위해서는 인자로 `-j` 뒤에 몇 개의 쓰레드를 사용할 지 숫자를 적어서 전달하면 됩니다.

예를 들어서

```
$ make -j8
```

을 하면 `make` 가 8 개의 쓰레드에 나뉘어서 실행됩니다. 아마 `make` 속도가 월등하게 향상되는 것을 보실 수 있을 것입니다. 통상적으로 코어 개수 + 1 만큼의 쓰레드를 생성해서 돌리는 것이 가장 속도가 빠릅니다.