시스템 프로그래밍6

S-개발자 4기 2026-03-12(목)

서울 송파구 동남로 130, 2층 제 4강의실

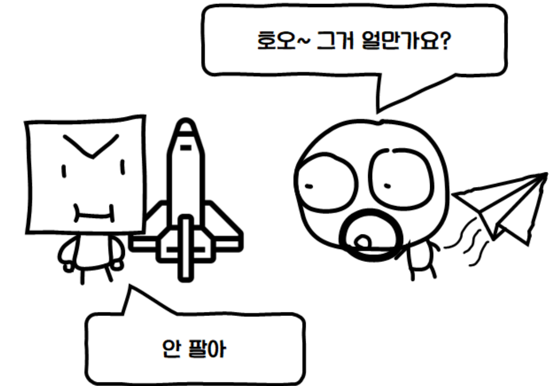
대표이사 전상현

들어가기 전에 수업 개요

과정명	시스템 프로그래밍	강사명	전상현
강의시간	24시간 (4일 x 6시간)		
강의목표	컴퓨터 시스템을 이해한다. 나아가 다양한 시스템을 제어하는 C++ 프로그래밍 기술을 익힌다.		
평가방식	시험 50%, 실습 50%		

강의내용

시간(H)	제목	내용
1H	오리엔테이션	강사 소개 후 간단한 퀴즈와 함께 실행파일의 구조를 이해한다.
2H	변수형과 유니코드	시스템에 존재하는 모든 종류의 변수형을 알아본다. 다국어를 표현하는 문자체계인 유니코드를 이해한다.
3H	개발환경 구축하기	윈도우/리눅스(wsl)에 빌드할 수 있는 환경을 구성한다. 기본적인 개발툴 사용법을 익힌다.
2H	문자열 정복하기	C++과 친해지기 위한 코드를 작성해본다. VisualStudio 디버깅 기술을 배운다.
2H	C++ 컴파일러/링커/디버거 정복하기	유니코드 상호 변환하는 함수를 이해한다. 문자열을 자유자재로 다루는 방법을 배운다.
2H	파일시스템 정복하기	플랫폼별 파일/디렉토리 접근 함수를 이해한다. 파일 시스템을 자유자재로 다루는 방법을 배운다.
2H	데이터 정복하기	STL 자료구조를 이해하고, 실무 응용 기술을 배운다. XML/JSON/INI 등을 자유자재로 다루는 방법을 배운다.
2H	메모리 정복하기	C++의 꽃, 스택 메모리의 장단점을 알아본다. 4가지 영역의 메모리를 자유자재로 활용한다.
2H	소켓 정복하기	UDP/TCP 통신, DATAGRAM/STREAM의 차이를 이해한다. 소켓 통신 시스템을 자유자재로 다루는 방법을 배운다.
6H	**최종실습**	**앞에서 배운 지식과 기술을 활용하여 미니 프로젝트를 진행한다.**



코딩 실력이 곧 우주선이다.
우주같이 광활한 컴퓨터 세계로 여행을 떠나자.

<참고서적>

크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북

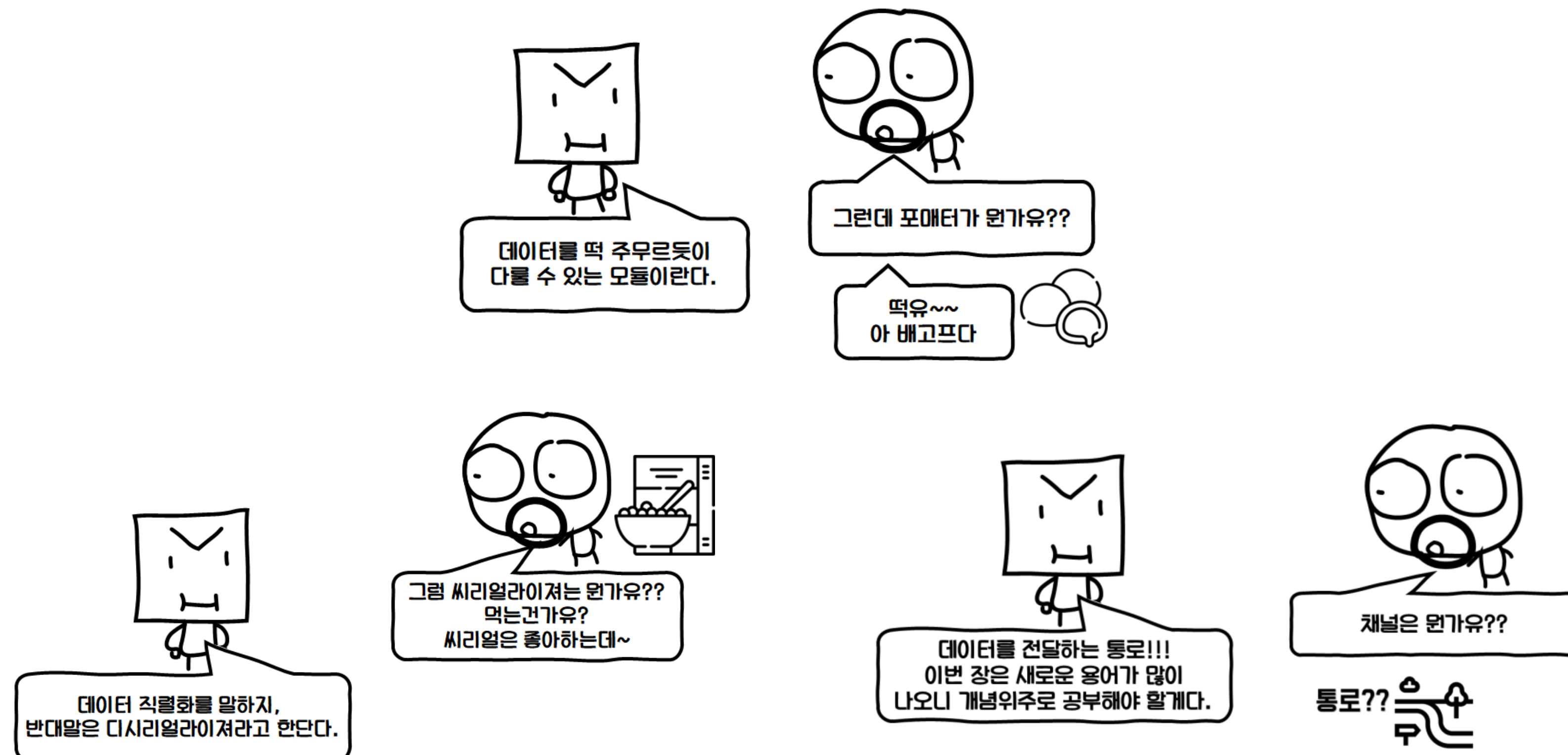
아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

데이터 정복하기

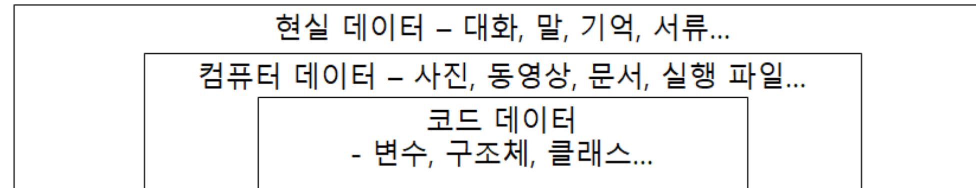
데이터 정복하기: 들어가기 전에

이번 장에서는 **데이터를 다루는 근본적인 기술**에 대해 이야기해봅니다. 사실상 프로그램의 중심은 데이터입니다. 정복할 수만 있다면 분야를 막론하고 코딩이 훨씬 쉬워질 겁니다.

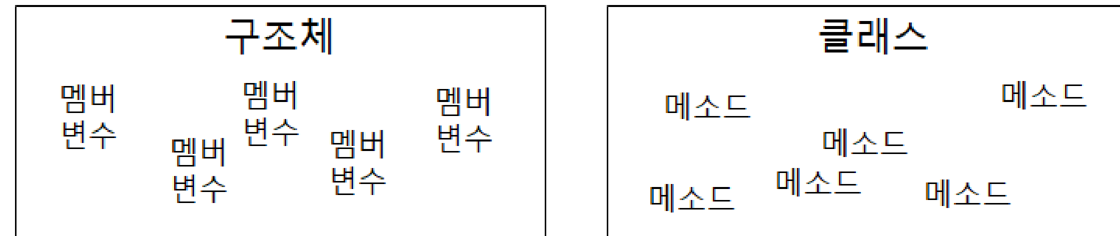
사실 데이터는 이미 정복된 영역입니다. 이번 장에 등장하는 **포매터**나 **채널** 등의 용어들은 통상적으로 사용하는 용어들을 차용한 것이죠. **잘 만들어진 기존의 라이브러리들을 사용하기 전에 어떤 원리와 구조로 만들어진 것인지 이해한다면 훨씬 쉽게 접근할 수 있을 것**이리라 생각합니다. 적어도 데이터를 아직 정복하지 못한 사람들이라면 말이죠.



데이터는 프로그램의 핵심입니다. 알고리즘이 핵심인 것처럼 보일 때도 있지만 기본적으로 모든 것은 데이터에서 시작합니다. **입력, 처리, 출력으로 이어지는 프로세스는 무엇을 가공하고 전달하기 위한 것**인지 떠올려 보시면 금세 답을 찾으실 거라 믿습니다.



코딩에서의 데이터는 구조체입니다. 실행 중에 필요한 변수들을 메모리에 올려 둔 것이죠. 파일이나 DB, 네트워크 등 다양한 경로에서 얻어옵니다. 그럼 구조체만 있을까요? 사실 클래스도 포함됩니다. 둘은 거의 형제 같은 존재라서 둘을 통칭해 구조체라고 표현했습니다.



<그림. Public 범위에서 바라본 구조체와 클래스의 생김새>

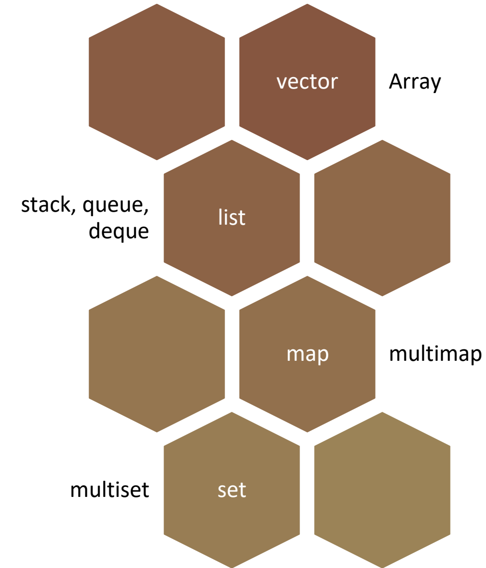
STL 자료구조

어느 정도 코드를 짤 수 있게 된다면 그 다음으로 신경 써야 할 부분이 바로 **자료 구조**입니다. 큐와 스택, 배열과 리스트, 트리로 대표되는 개념적인 자료형은 STL이라는 표준 라이브러리로 제공합니다. 표준이라는 것은 윈도우나 리눅스 맥OSX 등의 플랫폼에 상관없이 어디서든 쓸 수 있다는 뜻입니다. C++ 그 자체라는 것이죠.

논의하려는 것은 STL의 일반적인 사용법만은 아닙니다. 이미 STL에 관련된 책들은 시중에 많이 나와 있으니 같은 말을 되풀이할 필요는 없다고 생각합니다. 따라서 STL을 전혀 모르거나 생소하다면 미리 학습하기를 권장합니다. 그렇지 않더라도 STL이 무엇인지 맛보고 싶다면 가볍게 읽어보는 것도 좋겠습니다.

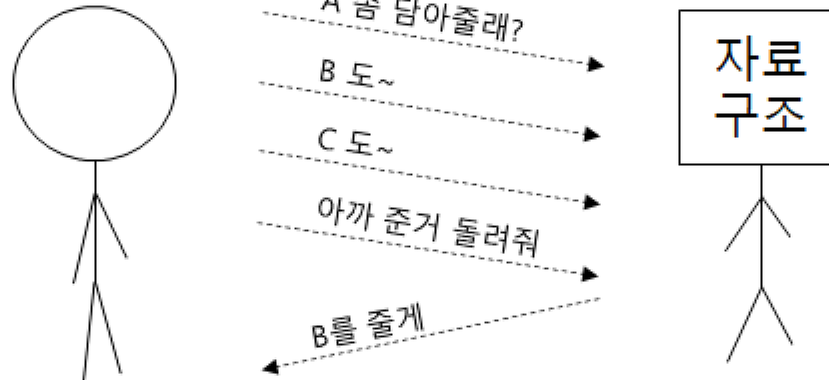
자료 구조는 데이터를 담는 상자, 그 곳에 어떻게 담는지를 알게 된다면.

개발자는 데이터 처리 도중 일시적으로 데이터를 모아두어야 하는 상황이 있습니다. 그리고 그것을 다시 필요할 때 꺼내 써야 하는데 넣는 순서와 꺼내 온 데이터의 순서적인 연관성을 다루는 것이 자료구조의 역할입니다. 믿음직한 도우미이죠.



잘 구성된 하나의 자료구조는 열 알고리즘이 부족하지 않습니다. 잘만 사용한다면 복잡한 문제도 의외로 쉽게 풀 수 있습니다. 그럼 어떤 종류들이 있는지 알아 둬야할 텐데요, 기본적으로 C++ STL에서 제공하는 자료형은 다음과 같이 크게 다섯 가지 종류입니다.

1) 벡터(vector)
2) 리스트(list)
3) 맵(map)
4) 셋(set)
5) 문자열(string)



벡터는 우리가 익히 알고 있는 배열을 모티브로 만든 것입니다. 가장 직관적이고 사용하기 쉽습니다.
바로 예제 코드를 보겠습니다.

```
#include <stdio.h>
#include <vector>

int main()
{
    std::vector<int> vecTest(10, 0);
    vecTest[1] = 1;
    vecTest[2] = 2;
    vecTest[8] = 8;
    vecTest[9] = 9;

    size_t i;
    for (i = 0; i < vecTest.size(); i++)
        printf("%d ", vecTest[i]);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int arrTest[10] = { 0, };
    arrTest[1] = 1;
    arrTest[2] = 2;
    arrTest[8] = 8;
    arrTest[9] = 9;

    size_t i;
    for (i = 0; i < sizeof(arrTest) / sizeof(arrTest[0]); i++)
        printf("%d ", arrTest[i]);
    return 0;
}
```

우리가 익히 알고 있는 배열과 사용법을 비교해봤는데 상당히 유사함을 알 수 있습니다.
차이점이라면 아무래도 벡터는 클래스 객체이다 보니 선언부와 배열의 개수를 얻어오는 부분정도 일겁니다.

클래스의 장점은 이렇게 가독성을 해칠 수 있는 코드를 읽기 좋게 만든다는 것입니다. 당연한 소리지만 size라는 메소드를 부르는 것이 sizeof 구문을 이용해 계산하는 것보다 읽기가 편하죠.

그 뿐만 아닙니다. 벡터만 가능한 일도 있습니다.

```
#include <stdio.h>
#include <vector>

int main()
{
    std::vector<int> vecTest;
    vecTest.push_back(1);
    vecTest.push_back(2);
    vecTest.push_back(3);
    vecTest.push_back(4);
    vecTest.push_back(5);

    size_t i;
    for (i = 0; i < vecTest.size(); i++)
        printf("%d ", vecTest[i]);
    return 0;
}
```

위 코드의 출력 결과는 아래와 같습니다.

```
1 2 3 4 5
```

엔트리를 고정해서만 사용할 수밖에 없는 배열에 비해 동적으로 확장하며 할당하는 것도 가능합니다. 물론 내부적으로 메모리를 파괴하고 다시 만드는 비효율적인 문제를 담고 있지만 중요한 것은 기능이 존재한다는 겁니다.

비슷하게 프로그램 실행 도중 배열의 크기를 한 번에 다시 정하는 방법도 있습니다.

```
#include <stdio.h>
#include <vector>

int main()
{
    std::vector<int> vecTest;
    vecTest.push_back(1);
    vecTest.push_back(2);

    vecTest.resize(10, -1);

    size_t i;
    for (i = 0; i < vecTest.size(); i++)
        printf("%d ", vecTest[i]);
    return 0;
}
```

이렇게 resize라는 메소드를 이용하면 기존의 배열 요소들은 보존하면서 정해진 크기로 새로 확장하는 것이 가능합니다. 위 코드의 실행 결과는 다음과 같습니다.

```
1 2 -1 -1 -1 -1 -1 -1 -1 -1
```

새로 할당된 공간을 -1로 초기화하여 총 10개의 요소를 가진 벡터가 된 것입니다. 벡터만이 가진 장점이죠.

마지막으로 가장 중요한 점이 있습니다. 과연 배열이 가진 장점도 벡터가 그대로 가지고 있을까?

다음과 같은 경우입니다. **양쪽의 두 코드는 완전히 동일하게 동작합니다.**

```
#include <Windows.h>

int main()
{
	const size_t tBufferSize = 1024;
	BYTE* pBuffer = new BYTE[tBufferSize];
	memset(pBuffer, 0, tBufferSize);
	...
	delete [] pBuffer;
	return 0;
}
```

```
#include <Windows.h>
#include <vector>

int main()
{
	std::vector<BYTE> buffer;
	buffer.resize(1024);
	memset(&buffer[0], 0, buffer.size());
	return 0;
}
```

벡터도 내부에 배열처럼 연속된 메모리를 할당해서 사용하고 있거든요. 그 점을 잘 이해하는 개발자라면 이미 위와 같은 코드를 작성해서 사용하고 있었을 거라 믿습니다.

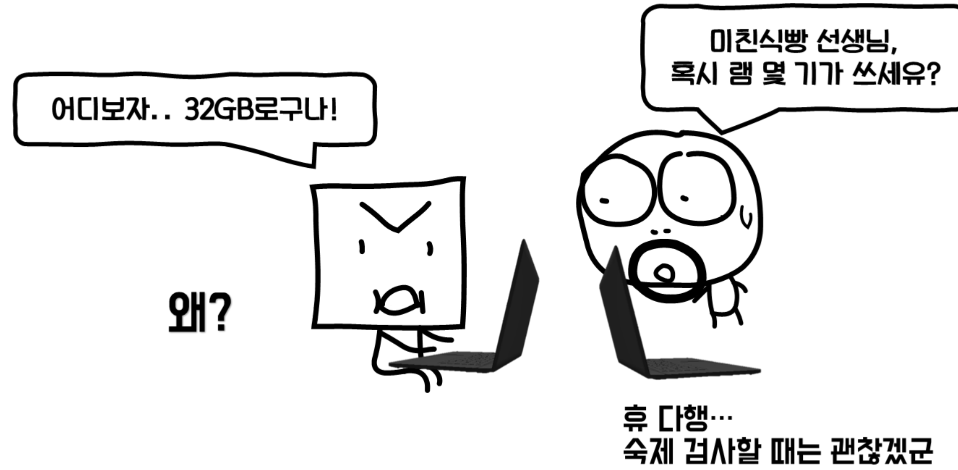
벡터를 이용한 동적 버퍼 할당은 다음과 같은 장점을 갖고 있습니다.

- 버퍼를 임의의 크기로 생성하고 언제든 그 크기를 size로 조회할 수 있다.
- 버퍼를 파괴할 필요가 없다.

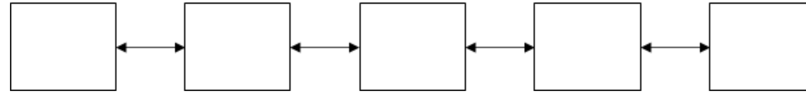
첫번째 장점은 **코드를 좀더 보기 좋게 만든다**는 정도네요. 하지만 두번째는 정말 중요한 부분입니다. C++의 최대 장점이자 단점은 메모리를 직접 관리할 수 있다는 점입니다. 따라서 어찌보면 new와 delete를 많이 사용하는 것이 C++을 잘 쓰는 것이라 여겨질 수 있지만 자칫 delete를 놓치는 실수가 발생하고는 합니다. 겪어본 사람들은 알 텐데 한 번 발생한 메모리 누수는 교정하기가 생각보다 쉽지 않습니다. 잘못하면 이미 파괴한 것을 한 번 더 파괴해서 메모리 접근 오류라는 치명적인 에러를 유발할 수도 있으니까요.

그럼 **버퍼를 파괴할 필요가 없다**는 것이 얼마나 중요한 장점인지 아시겠죠? 메모리를 관리할 수 있다는 C++ 고유의 장점을 수용하면서 메모리 릭이나 접근 오류가 발생하는 치명적인 단점을 제거할 수 있다는 것입니다. 가급적 앞으로는 new와 delete 대신 벡터로 대체하는 방안을 찾아보길 바랍니다.

STL 벡터(vector)를 동적 메모리처럼 활용하기



리스트형 자료구조에서 가장 대표적인 것은 링크드 리스트(linked list)입니다. 그 외 큐(queue)와 스택(stack), 데큐(deque)는 일종의 변형이죠. 메모리 상의 특징은 연속되지 않은 노드를 일렬로 연결짓는다는 점입니다.



그 덕분에 항목을 추가, 제거하는 비용이 적습니다. 먼저 대표격인 list 자료구조의 사용방법을 코드로 보겠습니다. 더불어 별반 차이가 없는 deque도 같이 보여드립니다.

```
#include <list>

int main()
{
    std::list<int> listTest;
    listTest.push_back(1);
    listTest.push_back(2);
    listTest.push_back(3);
    listTest.pop_back();
    listTest.push_front(4);

    for (int nValue : listTest)
        printf("%d ", nValue);

return 0;
}
```

```
#include <deque>

int main()
{
    std::deque<int> dequeTest;
    dequeTest.push_back(1);
    dequeTest.push_back(2);
    dequeTest.push_back(3);
    dequeTest.pop_back();
    dequeTest.push_front(4);

    for (int nValue : dequeTest)
        printf("%d ", nValue);

    return 0;
}
```

메소드가 꽤 직관적입니다. 출력 결과는 다음과 같습니다.

비슷하게 사용되는 stack과 queue도 보시지요.

```
#include <stack>

int main()
{
    std::stack<int> stackTest;
    stackTest.push(1);
    stackTest.push(2);
    stackTest.push(3);

    while (!stackTest.empty())
    {
        printf("%d ", stackTest.top());
        stackTest.pop();
    }

    return 0;
}
```

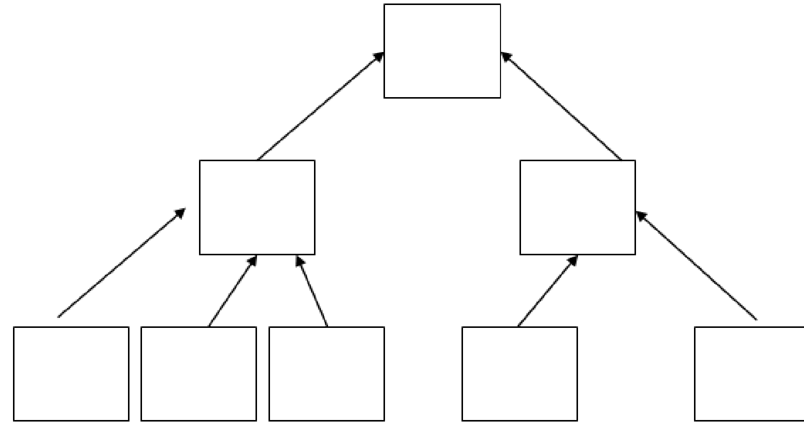
```
#include <queue>

int main()
{
    std::queue<int> queueTest;
    queueTest.push(1);
    queueTest.push(2);
    queueTest.push(3);

    while (!queueTest.empty())
    {
        printf("%d ", queueTest.front());
        queueTest.pop();
    }

    return 0;
}
```

둘의 차이는 최상단의 항목을 조회하는 메소드 뿐입니다. 스택은 top()이고, 큐는 front()로군요. 자료구조 개념에 부합하는 이름이라고 볼 수 있습니다. 가만보면 리스트에서는 push_back, push_front 두 가지 입력 메소드가 있었던 반면, 스택이나 큐는 단순 push밖에 없습니다. 그 이유는 입력할 수 있는 부분이 한쪽 뿐이기 때문입니다.

출력도 마찬가지인데 리스트는 전체 항목을 순회할 수 있는 장치가 있던 반면, 스택과 큐는 오로지 top과 front뿐입니다. 따라서 **전체를 보기 위해서는 데이터가 소진할 때까지 하나씩 pop해가며 얻어오는 방법 밖에 없습니다.**

앞에서 미리 보여 드린 것처럼 데큐는 리스트와 거의 동일합니다. 결론은 같은 리스트형 구조는 한쪽에서만 입출력할 수 있는 것과 양쪽에서 가능한 두 타입으로 나뉜다는 겁니다.

맵은 트리 구조로 구성됩니다. **키와 값의 쌍으로 이뤄진다는 이유로 사전식 자료구조**라고도 불립니다.



정렬의 기준은 키이고 각 노드는 키에 대응하는 값이 들어가게 됩니다. 트리 특성상 노드를 순회하거나 추가, 삭제하는 것도 가능합니다. 예제코드를 보겠습니다.

```cpp
#include <map>

int main()
{
    std::map<int, const char*> mapTest;
    mapTest.insert(std::make_pair(0, "zero"));
    mapTest.insert(std::make_pair(0, "zero2"));
    mapTest.insert(std::make_pair(1, "one"));
    for (auto iter = mapTest.begin(); iter != mapTest.end(); iter++)
        printf("%d-%s ", iter->first, iter->second);

    mapTest.erase(0);
    for (auto iter = mapTest.begin(); iter != mapTest.end(); iter++)
        printf("%d-%s ", iter->first, iter->second);

    return 0;
}
```

```cpp
#include <map>

int main()
{
    std::multimap<int, const char*> mapTest;
    mapTest.insert(std::make_pair(0, "zero"));
    mapTest.insert(std::make_pair(0, "zero2"));
    mapTest.insert(std::make_pair(1, "one"));
    for (auto iter = mapTest.begin(); iter != mapTest.end(); iter++)
        printf("%d-%s ", iter->first, iter->second);

    mapTest.erase(0);
    for (auto iter = mapTest.begin(); iter != mapTest.end(); iter++)
        printf("%d-%s ", iter->first, iter->second);

return 0;
}
```

맵(map)을 정렬 알고리즘처럼 사용하기

맵이 갖고 있는 특별함이 있습니다. 바로 정렬이라는 개념입니다. 트리를 구성하기 위해서는 키를 정렬해야 하므로 새로운 노드를 insert 할 때마다 기존 노드들 사이에 정렬된 위치에 들어가게 됩니다. 그래서 삽입의 복잡도를 O(log(n))이라고 표현하죠.

그럼 맵에 모인 데이터들은 내부에 정렬이 돼있다는 가정을 할 수 있습니다. 그것을 실험하기 위해 다음과 같은 코드를 작성해봤습니다.

```
#include <map>

int main()
{
	std::map<int, const char*> mapTest;
	mapTest.insert(std::make_pair(5, "five"));
	mapTest.insert(std::make_pair(2, "two"));
	mapTest.insert(std::make_pair(3, "three"));
	for (auto iter = mapTest.begin(); iter != mapTest.end(); iter++)
		printf("%d-%s ", iter->first, iter->second);

	return 0;
}
```

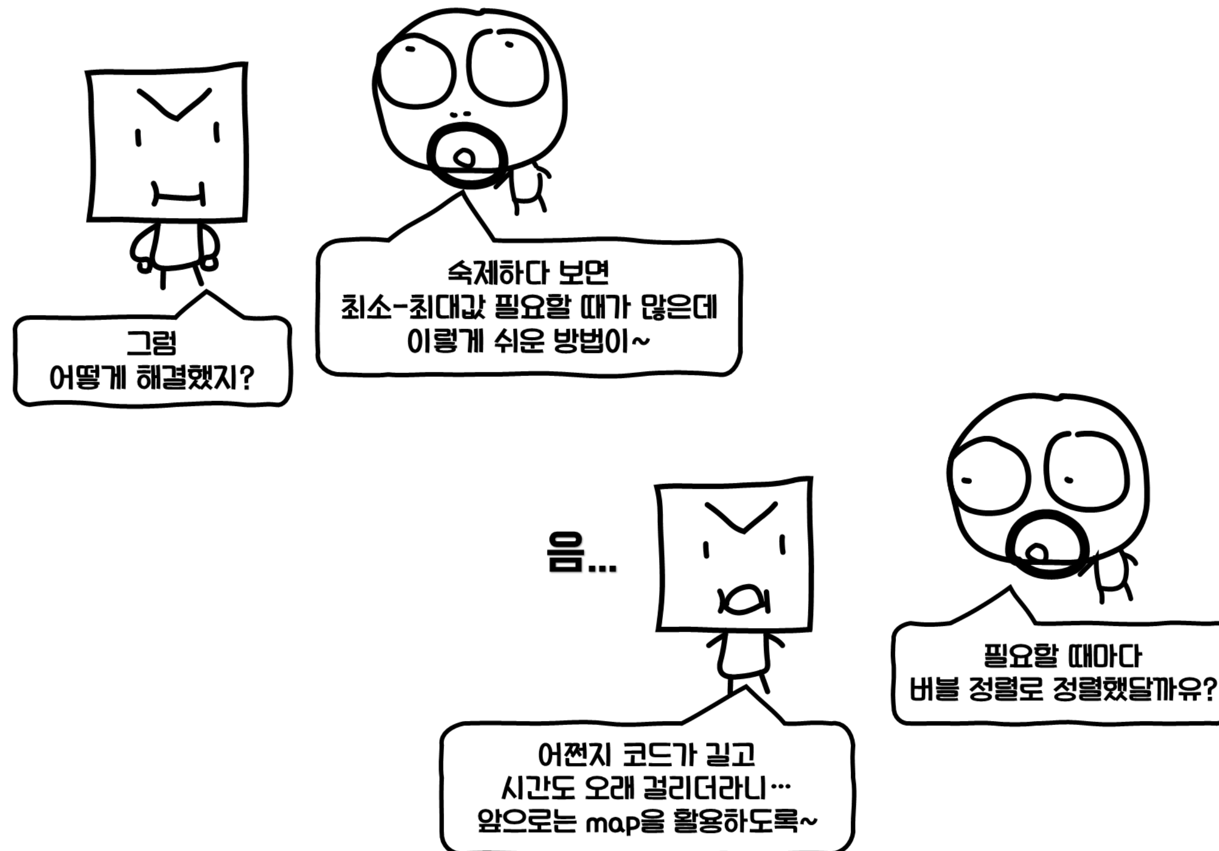
실행 결과는 다음과 같습니다.

맵(map)을 정렬 알고리즘처럼 사용하기

반대로 **가장 큰 키의 값을 가져오는 것**은 end 메소드를 쓰면 될까요? 그건 안됩니다. 메모리 접근 오류가 발생합니다. 맵에 접근하는 반복자의 끝은 널이기 때문입니다.

대신 다음과 같이 **rbegin 메소드를 이용**하면 됩니다. rend와 사용되는 순환자로 반대 방향으로 접근하기 위해 쓰입니다.

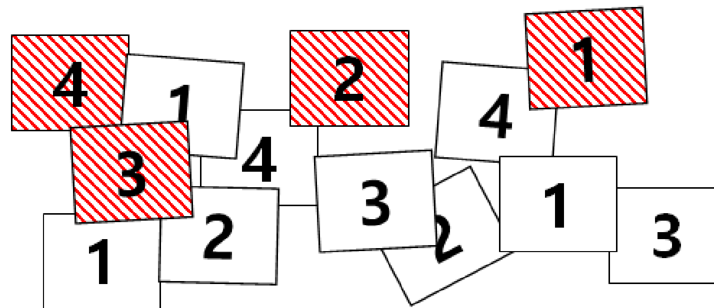
```
printf("%d-%s", mapTest.rbegin()->first, mapTest.rbegin()->second);
```



셋은 맵의 변형입니다. 맵은 키와 값의 쌍이었다면 셋 값만 존재합니다. 값이 곧 키로 쓰이는 것이죠.
구성은 마찬가지로 트리입니다. 그러다 보니 정렬된 값으로 보관된다는 점도 동일합니다. 역시 최소 최대 값을 얻어내는 것도 begin과 rbegin 메소드를 이용하면 됩니다.

```
#include <set>

int main()
{
    std::set<int> setTest;
    setTest.insert(4);
    setTest.insert(1);
    setTest.insert(3);
    setTest.insert(4);
    setTest.insert(2);
    for (auto iter = setTest.begin(); iter != setTest.end(); iter++)
        printf("%d\n", *iter);
    return 0;
}
```

키를 따로 관리하는 map보다는 간결합니다. 다만 값 자체로 정렬하여 사용하는 경우는 매우 한정적이니 필요할 때 잊지 않고 사용할 수 있어야 합니다. 맵은 안되지만 셋이니까 가능한 활용법이 있습니다. 중복이 허용되는 상황에서 **유일한 것만 남겨두는 것이 필요한 경우입니다**.



걱정 말고 쓰십시오. 단, 파이썬과 같은 언어에서는 통하지 않을 수 있습니다.

C++에서만 확인된 것입니다.

어찌보면 가장 흔히 쓰고 친숙해야 할 자료형이지만 가장 마지막에 배치한 이유가 있습니다. 사용법이 너무 자료구조스럽기 때문입니다. 아마도 문자열은 자료구조와는 다르게 더하기 연산자나 substr 및 compare, find 등 문자열 고유 함수들이 있어 별개라고 생각했을지도 모르겠네요.

문자열은 하나 이상의 문자를 보관하고 있는 자료구조입니다. 차이점이라면 그 문자가 char냐 wchar_t냐의 차이일 뿐입니다.

문자 형식	자료형	유사 자료형
char	std::string	std::vector<char>
wchar_t	std::wstring	std::vector<wchar_t>

문자열은 개별 문자들을 연속된 메모리 공간에 두는 것입니다. 그런 의미에서 vector와 유사합니다.
차이점이라면 **문장의 끝을 나타내기 위한 마지막 널(\0) 문자의 존재 여부**입니다. 이를 테면 다음과 같은 명령으로 할당되는 메모리 공간은 1바이트 차이가 납니다.

자료형	명령어	할당된 메모리
std::string	strTest.resize(10)	11 바이트(널 포함)
std::vector<char>	vecTest.resize(10)	10 바이트

다른 고급언어에 비해 STL의 문자열은 다음과 같은 단점이 있습니다.

- Trim, Format, Lower, Upper 등 흔히 쓰이는 문자열 함수가 없다.
- 문자열 포인터를 꺼내려면 c_str 함수를 이용해야 한다.

첫번째 단점은 사실 C++로서는 치명적입니다. C언어나 C++이 다른 언어보다 어렵다고 느껴지는 이유가 바로 이 문자열 함수 미비입니다. 생각보다 프로그램에서는 문자열을 많이 다룹니다. 문자열을 다룰 때마다 다른 언어에서 손쉽게 사용하던 함수들이 없다면 답답할 수밖에 없죠. 아마 대부분의 C++ 숙련자들은 본인들만의 STL 문자열 함수들을 만들어 사용하고 있을 겁니다.

두번째 단점도 문제입니다. 단편적인 상황을 보여드리겠습니다.

```
#include <string>

int main()
{
    std::string strTest = "Hello world!";
    printf("%s\n", strTest);
    return 0;
}
```

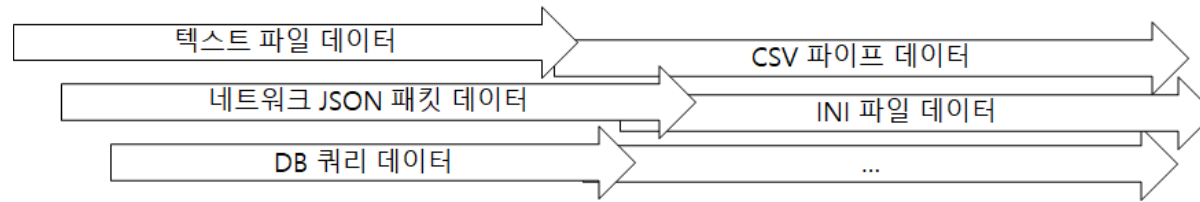
```
#include <string>

int main()
{
    std::string strTest = "Hello world!";
    printf("%s\n", strTest.c_str());
    return 0;
}
```

왼쪽과 같은 코드는 strTest의 객체 주소를 출력하는 겁니다. 오른쪽과 같이 **STL 문자열의 포인터를 얻어와서 화면에 찍는 것이 올바른 사용법**입니다.

STL 문자열에 익숙한 사람들은 c_str()을 잊지 않고 잘 사용하지만 처음에는 생소할 수 있습니다. 이 또한 C++의 진입 장벽이 될 법한 사항입니다.

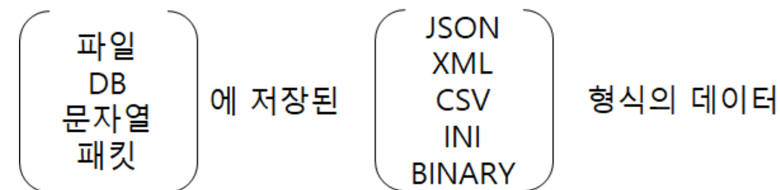
포매터 이해하기

최초의 데이터는 어디에서 생성될까요? **사람으로부터 만들어집니다.** 현실에서 사람에 의해 만들어진 데이터는 **컴퓨터를 통해 코드의 영역까지 흘러 들어옵니다.**

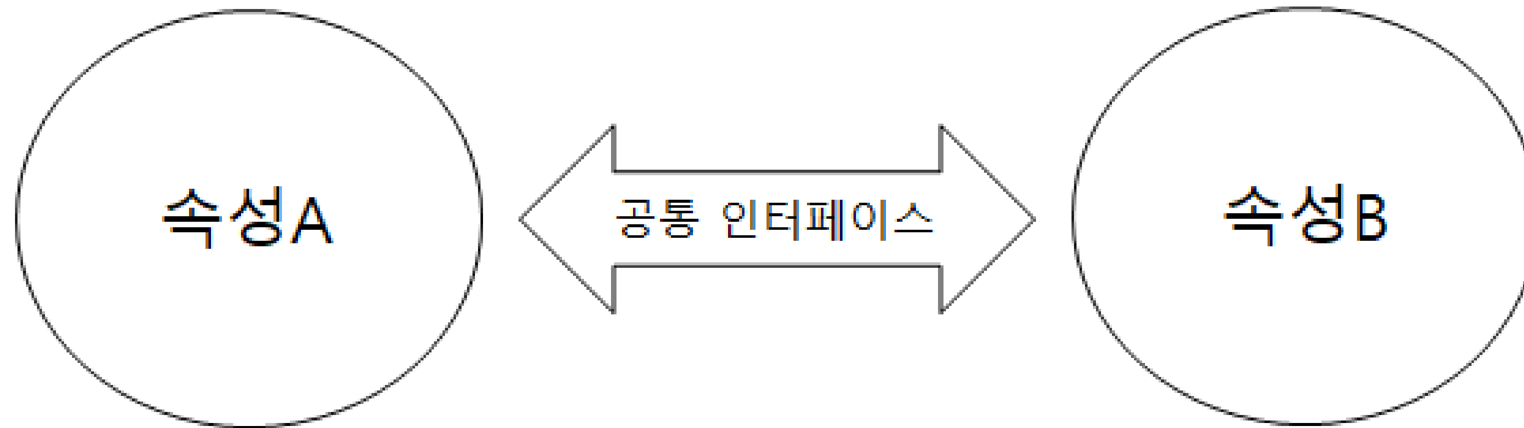


대체 얼마나 많은 형식의 데이터가 있는지 잠시 생각해봅시다. 당장 떠올려봐도 **스무 가지**나 되네요

파일에 저장된 JSON 데이터	DB에 저장된 JSON 데이터
파일에 저장된 XML 데이터	DB에 저장된 XML 데이터
파일에 저장된 CSV 데이터	DB에 저장된 CSV 데이터
파일에 저장된 INI 데이터	DB에 저장된 INI 데이터
파일에 저장된 BINARY	DB에 저장된 BINARY
문자열에 저장된 JSON 데이터	패킷에 저장된 JSON 데이터
문자열에 저장된 XML 데이터	패킷에 저장된 XML 데이터
문자열에 저장된 CSV 데이터	패킷에 저장된 CSV 데이터
문자열에 저장된 INI 데이터	패킷에 저장된 INI 데이터
문자열에 저장된 BINARY	패킷에 저장된 BINARY



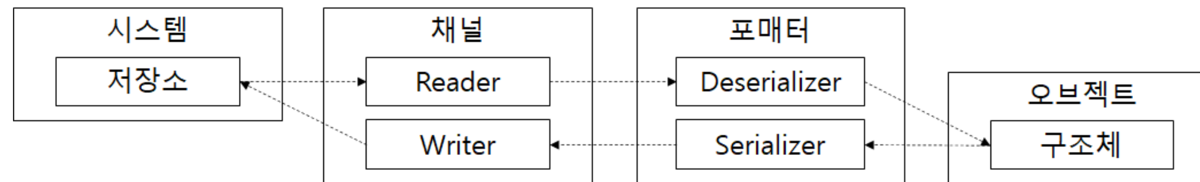
"브릿지 패턴"이라는 것이 있습니다. 서로 다른 둘 이상의 속성에 마치 다리를 놓듯이 공통된 인터페이스를 두고 조합하여 사용하는 구조입니다. 레고 블록을 조립하는 것처럼 말이죠.



우리가 앞에서 알아본 20개의 데이터는 4*5의 조합으로 유래된 것인데 이것을 독립적으로 나눌 수만 있다면 **4+5로 9개만 구현해도 20개의 효과를 볼 수 있어요**. 이 패턴은 장기적으로 더 빛을 발휘하게 됩니다. 개수가 많아질수록 조합할 수 있는 방법의 수가 많아지니까요.

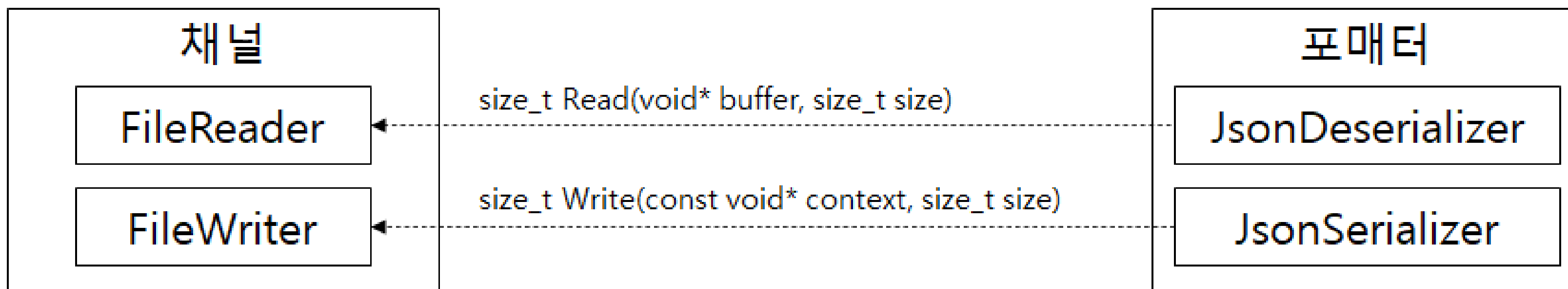
이 패턴의 관건은 **공통 인터페이스**에 있습니다. 어떤 방법으로 다리를 놓아줄지 잘 정해 둬야 합니다. 이 부분을 논하기 전에 간단히 몇 가지 용어를 정의해 볼게요. 앞으로 자주 사용될 예정입니다.

용어	설명
Object	최종 결과물인 구조체
Formatter	Object를 정해진 형식으로 내보내거나 가져오는 기능
Channel	저장소에 접근하는 통로 및 접근자
Serializer	메모리에 저장된 값을 특정 형식의 데이터로 만드는 것
Deserializer	특정 형식의 데이터를 파싱하여 값을 추출하는 것



데이터 정복하기: 채널과 포매터의 공통 인터페이스

공통 인터페이스는 **다양한 채널과 다양한 포매터에 모두 적용할 수 있을 함수를 말합니다**.
한 번에 떠올리기가 막연하다면 구체적으로 어떤 채널과 포매터를 구현할지 생각해봅시다.



위 그림은 파일 접근 채널과 Json 포매터를 도식화한 것입니다. 여기에 쓰인 **화살표는 종속성을 뜻하는 것으로 A가 B를 알고 있다 혹은 A가 B를 사용한다**라고 생각하면 됩니다. 상황에 따른 화살표의 쓰임이 다르다는 것을 알고 계시면 됩니다.

Read 함수는 FileReader에서 제공하는 것으로 JsonDeserializer가 호출합니다. 텍스트 파일에 기록된 Json 문자열을 읽어 오기 위한 것입니다. Write함수는 반대로 Json 구문을 텍스트 파일로 저장하기 위한 것입니다.

두 함수는 고맙게도 형태가 같습니다. 공통 인터페이스를 만들기 위한 중요한 단서죠. 다만 함수 이름은 바꿔야 겠네요. Read와 Write를 포함하는 표현인 Access가 적합할 것 같습니다.

```
size_t Access(void* pData, size_t tDataSize);
```

위 함수는 채널의 멤버 함수로 구현해두면 좋을 것 같습니다.

예를 들어 다음과 같은 구조체가 있다고 가정해 봅시다. 그리고 JsonDeserializer가 메모리에 항목별 키와 값을 갖고 있다고 가정합시다. 그럼 구조체의 각 변수에 어떻게 그 값을 채울 수 있을까요?

```
struct ST_USER_INFO
{
    std::string strName;
    int nAge;
};
```

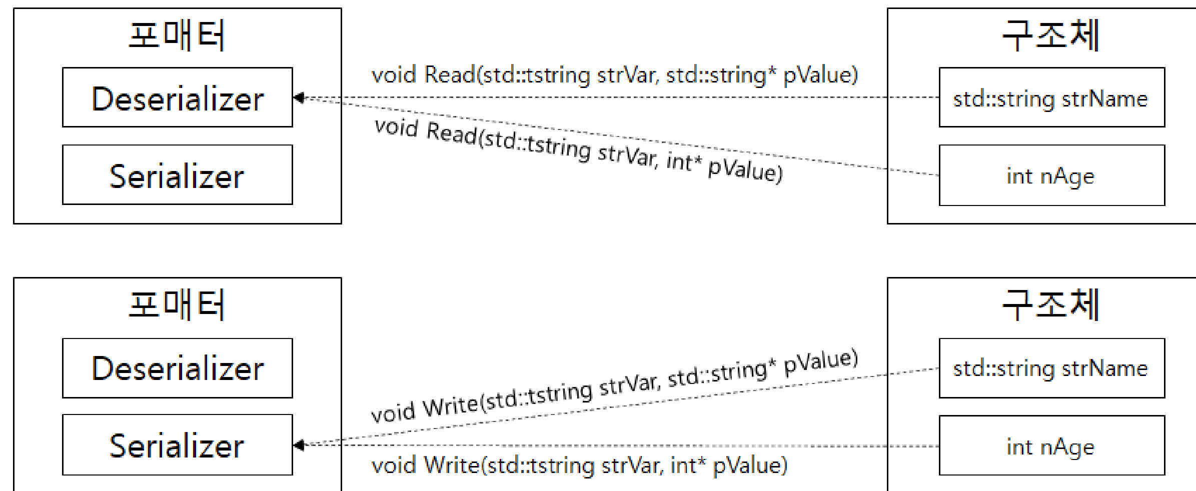
```
int main(void)
{
    ST_USER_INFO info;
    info.strName = "전상현";
    info.nAge = 41;
    return 0;
}
```

위 코드처럼 구조체 멤버 변수에 임의로 값을 입력하기 위해 필요한 정보는 다음과 같습니다.

1) 멤버 변수의 이름
2) 멤버 변수의 타입
3) 멤버 변수 포인터

우리가 유일하게 아는 것은 구조체를 가리키는 포인터입니다. 헌데 그것 만으로는 실행 중에 이 정보들을 얻을 수 없습니다. 언어적인 지원을 받아야만 이 문제를 풀 수 있습니다. 예를 들어 C#은 리플렉션이라는 메타 데이터의 특성에 접근할 수 있는 방법을 제공합니다. 그 덕에 일찍부터 데이터를 정복할 수 있던 겁니다.

우리의 C++ 개발자들은 이대로 포기해야 할까요? 당연히 아닙니다. 여기서 좌절하지 맙시다. 없으면 만들면 되죠. 구조체가 스스로 포매터에 알려주는 것은 어떨가요?



구조체에는 다양한 형식의 변수가 존재할 수 있습니다. 예제에 있는 string과 int뿐 아니라 list나 map, set과 같은 집합 자료형까지도요. 더불어 구조체 자체가 포함되는 것도 감안해야 합니다. 이와 같이 다양한 형식의 변수를 모두 준비해야 합니다. 변수 이름은 함수를 호출할 때 제공합니다.

남은 것은 함수의 네이밍인데 이번에는 **Sync**라는 용어를 써보려고 합니다. 채널처럼 접근한다는 개념이 아니라 **특정 변수의 값을 저장소와 동기화한다**는 게 더 적합하기 때문이죠.

정리하자면 다음과 같은 함수가 포매터에 준비되면 됩니다.

```
void Sync(std::tstring strVar, int* pValue);
void Sync(std::tstring strVar, std::string* pValue);
...
template <typename T> void Sync(std::tstring strVar, std::list<T>* pValue);
template <typename T> void Sync(std::tstring strVar, std::vector<T>* pValue);
template <typename K, typename T> void Sync(std::tstring strVar, std::map<K,T>* pValue);
```

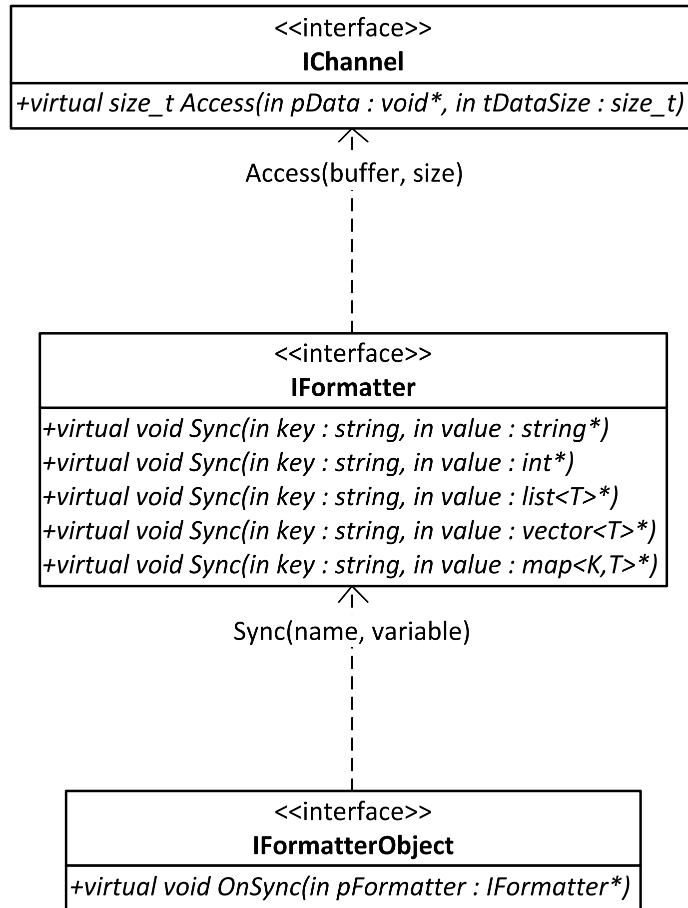
여기서 잠깐, 왜 C++에는 메타 데이터 특성에 접근하는 기능을 제공하지 않았을까?

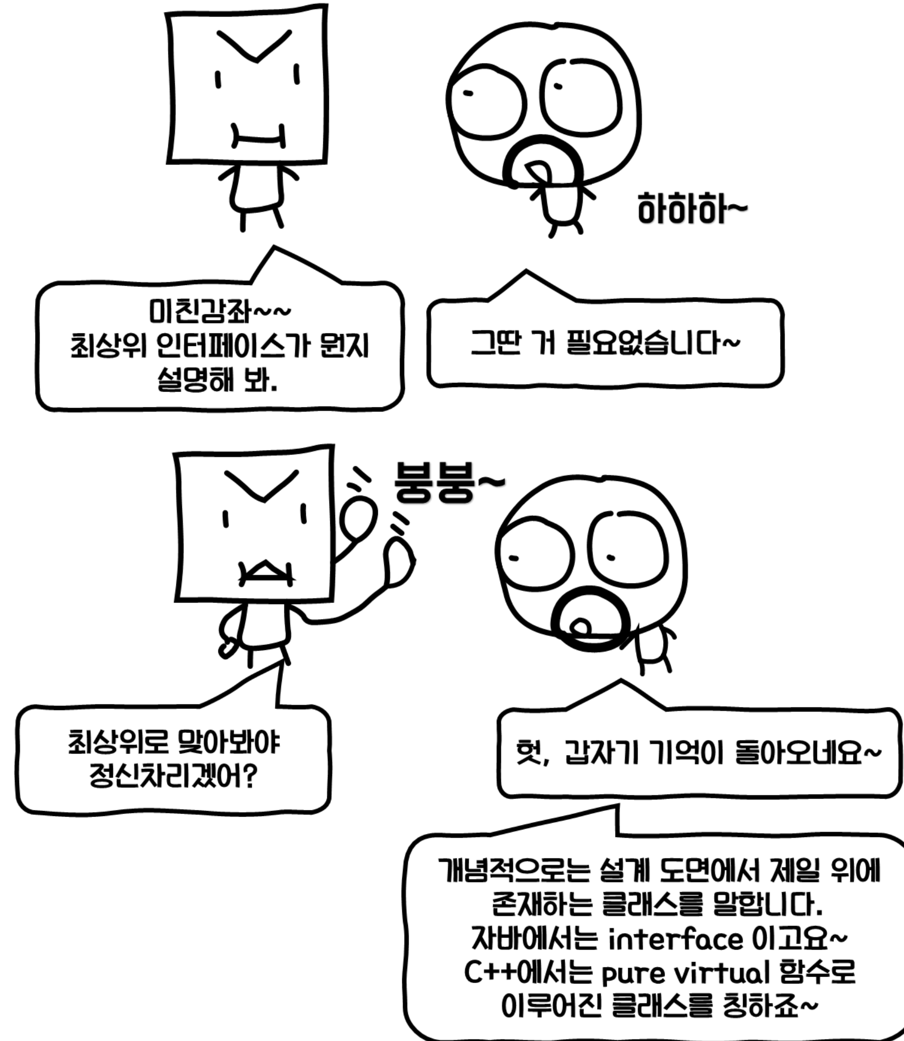
생각보다 프로그램에 개입하는 구조체나 클래스 같은 메타 데이터의 수가 많습니다. 그들의 특성 정보를 런타임에 제공하는 것은 실행 성능에 생각보다 많은 부하를 줄 수 있습니다. 어찌 보면 이 부분이 중급언어와 고급언어의 경계점이 될 수 있겠는데요, 약간의 불편함을 감수하고 성능을 보존하는 것을 선택한 것이 중급 언어인 C++이라고 생각할 수 있겠습니다.

대신 이번에 구현한대로 개발자가 독자적인 인터페이스를 만들어 해결은 할 수 있다는 점을 우리는 기억해야 합니다.

데이터 정복하기: 최상위 인터페이스 설계

지금까지 이야기한 것들을 다이어그램으로 나타내 보면 다음과 같습니다.



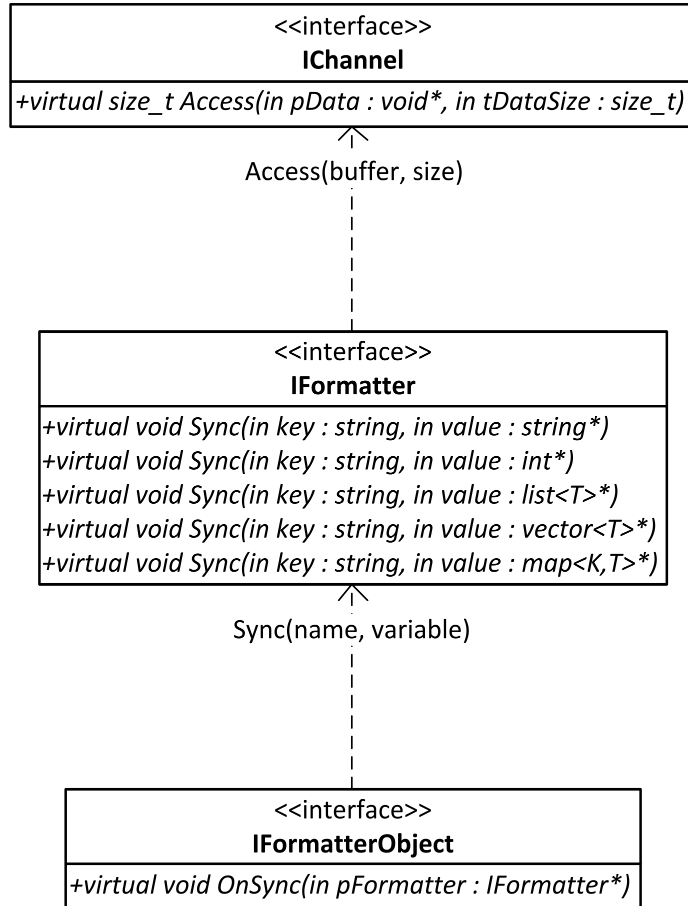


데이터 정복하기: 최하위 인터페이스

최상위가 있다는 것은 어딘가에는 최하위가 있다는 뜻입니다. 여기에 사용된 **최하위의 의미는** 더 이상 관련 구조에 연관된 것이 없다는 것으로 이해하면 됩니다. 다른 표현으로 **가장 구체화된 코드**라고 볼 수도 있어요. 그런 의미에서 아래의 코드는 상당히 구체적이고 명확합니다.

```
bool WriteJsonToFile(IFormatterObject* pObject, std::string strFilename);

bool ReadJsonFromFile(IFormatterObject* pObject, std::string strFilename);
```

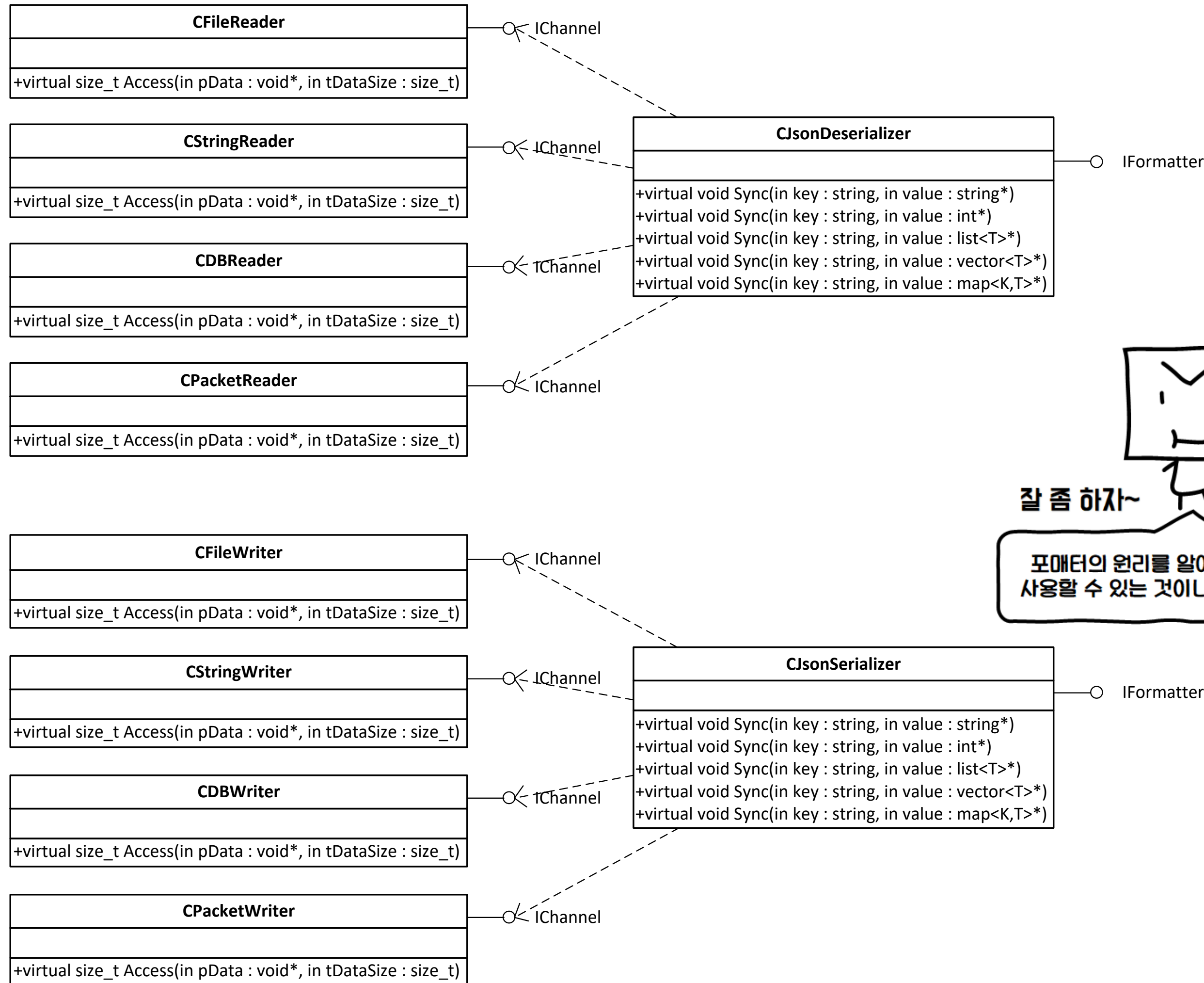
데이터 정복하기: 최상위 인터페이스 구현




```cpp
struct IChannel
{
    virtual ~IChannel() {}
    virtual bool CheckValidity(std::tstring& refStrErrMsg) = 0;
    virtual size_t Access(void* pData, size_t tDataSize) = 0;
};

struct IFormatter
{
    virtual ~IFormatter() {}
    virtual void Sync(std::tstring strVar, int* pValue) = 0;
    virtual void Sync(std::tstring strVar, float* pValue) = 0;
    virtual void Sync(std::tstring strVar, double* pValue) = 0;
    virtual void Sync(std::tstring strVar, std::string* pValue) = 0;
    template <typename T>
    virtual void Sync(std::tstring strVar, std::list<T>* pValue) = 0;
    template <typename T>
    virtual void Sync(std::tstring strVar, std::vector<T>* pValue) = 0;
    template <typename K, typename T>
    virtual void Sync(std::tstring strVar, std::map<K, T>* pValue) = 0;
};

struct IFormatterObject
{
    virtual ~IFormatterObject() {}
    virtual void OnSync(IFormatter& formatter) = 0;
};
```

데이터 정복하기: 포매터 구현하기 (설계도 관점)

채널과 포매터를 연결한 최종 구현체는 다음과 같습니다.



너무 복잡합니다요 선생님~

잘 좀 하자~

포매터의 원리를 알아야
사용할 수 있는 것이니라~

데이터 정복하기: 포매터 구현하기 (코드 관점)

코드 수준에서 보여주면 되려 간단합니다. 우리는 코드쟁이잖아요~

```
bool WriteJsonToFile(const IFormatterObject* pObject, std::string strFilename);
bool ReadJsonFromFile(IFormatterObject* pObject, std::string strFilename);
bool WriteJsonToString(const IFormatterObject* pObject, std::string& strOutput);
bool ReadJsonFromString(IFormatterObject* pObject, std::string strInput);
bool WriteJsonToDB(const IFormatterObject* pObject, HANDLE hDB, std::string strTable);
bool ReadJsonFromDB(IFormatterObject* pObject, HANDLE hDB, std::string strTable);
bool WriteBinToPacket(const IFormatterObjectW* pObject, std::vector<BYTE>& vecPacket);
bool ReadBinFromPacket(IFormatterObjectA* pObject, const std::vector<BYTE>& vecPacket);
```

최하위 함수 구현부도 상당히 깔끔합니다. 마치 레고 블록 쌓듯이 클래스를 선언하고 연결하면 됩니다. 나머지는 클래스 내부에서 처리하는 것이죠.

```
bool WriteJsonToFile(IFormatterObject* pObject, std::tstring strFilename)
{
    CFileWritter channel(strFilename.c_str());
    CJSONSerializer formatter(channel);
    pObject->OnSync(&formatter);
    return formatter.CheckValidity();
}
```

데이터 정복하기: 포매터 사용하기

이제 남은 것은 구조체를 선언하고 포매터를 이용하는 것입니다. 이미 최하위 함수들을 봤으므로 사용법이 예상될 겁니다.

구조체의 멤버 변수를 포매터에 알려야 하므로 **IFormatterObject를 상속받아 OnSync 함수를 이용하는 겁니다.** 약간은 귀찮아질 수 있는데 이게 어딥니까. 드디어 C++에서도 데이터를 정복할 수 있게 되었는데요.

```
struct ST_USER_INFO : public core::IFormatterObject
{
    std::tstring strName;
    int nAge;

    void OnSync(core::IFormatter& formatter)
    {
        formatter.Sync(TEXT("Name"), strName);
        formatter.Sync(TEXT("Age"), nAge);
    }
};
```

고생 많으셨습니다!