

More Deeper C++ Programming Language

Based on the materials listed below

ISO/IEC 14882:2003 Programming languages — C++
AMD64 Architecture Programmer's Manual 1 : Application Programming
AMD64 Architecture Programmer's Manual 2 : System Programming
The C++ Programming Language - 4th Edition
Effective C++
Effective Modern C++
MicroSoft Developer Network(MSDN)
Structure and Interpretation of Computer Programs - 2nd Edition
Code Complete - 2nd Edition
Memory Performance Attacks : Denial of Memory Service in Multi-Core Systems - Paper
A Case for NUMA-aware Contention Management on Multicore Systems - Paper
Memory System Performance in a NUMA Multicore Multiprocessor - Paper
etc...

Notice

The purpose of this document is to explore and organize the content of C++ more deeply from various perspectives and share it with the public based on the grammar specified in the standard, based on various documents and personal experiences.

Reproduction of this document for non-profit purposes is therefore permitted.

Please note that some of the contents of the document may be partially incorrect or added to the author's opinion.

Contact

E-Mail : hyunsuyu6478@gmail.com

Contents

1. 값과 변수 그리고 표현식.....	11
1.1. 변수의 개념.....	11
1.1.1. 변수의 정의.....	11
1.1.2. 값과 변수 그리고 표현식의 관계.....	11
1.1.3. 변수의 생성.....	13
- 선언의 개념.....	13
- Simple declaration.....	13
- Attributes.....	14
- Specifiers.....	14
- Declarators.....	14
- Initializer.....	15
- 예제.....	16
# 1번 예제.....	16
# 2번 예제.....	16
1.2. C++에서의 Fundamental Types와 Built-In Operators.....	18
1.2.1. Fundamental Types.....	18
- Void Type.....	18
- Signed and Unsigned Integer Types.....	18
- Boolean Type.....	18
- Character Types.....	18
- Floating-Point Types.....	18
1.2.2. Range of Values.....	19
1.2.3. Built-In Operators.....	20
1.2.4. Placeholder Type Specifiers - auto Type.....	20
- auto Type Deduction.....	21
1.3. 값의 해석.....	23
1.3.1. C++ Data Model과 Type Size.....	23
1.3.2. Scope와 변수의 관계.....	24
- Scope와 변수의 생명 주기.....	24
- 중첩된 Scope와 동일한 변수의 Identifier.....	27
1.4. 값의 범주.....	29
1.4.1. 변수(Variable)와 리터럴(Literal).....	29
1.4.2. 값의 범주와 해석.....	30
1.4.2.1. Primary Categories.....	31
- lValue.....	31
- prValue.....	31

- xValue.....	31
1.4.2.2. Mixed Categories.....	32
- glValue.....	32
- rValue.....	32
1.4.3. 값의 범주의 활용.....	32
# 1번 예제.....	32
# 2번 예제.....	33
 1.5. 표현식의 범주.....	34
1.5.1. 표현식의 정의.....	34
1.5.2. 표현식의 범주와 해석.....	34
- Primary Expression.....	34
- Full-Expression.....	34
- Potentially-Evaluated Expressions.....	35
- Discarded-Value Expressions.....	35
 1.6. C/C++ 표준에서 제공하는 Type 관련 Library.....	36
1.6.1. [stdint.h] or [cstdint].....	36
 1.7. 알아둘만한 이슈들.....	42
1.7.1. 초기화 종류와 관련된 이슈.....	42
- Copy Initialization.....	42
- Direct Initialization.....	42
- List-initialization.....	42
1.7.2. 값의 표현 한계에 관련된 이슈.....	43
1.7.3. Overflow, Underflow 이슈.....	43
1.7.4. 중간 형태 컴파일 결과물의 Decompiler에 의한 코드 노출 보안 이슈.....	44
1.7.5. 변수의 결합 시점과 그에 따른 Trade-Off.....	44
 2. 메모리와 포인터 그리고 레퍼런스.....	47
2.1. 메모리와 주소의 개념.....	47
2.1.1. 프로그래머 관점에서의 메모리 형태.....	47
2.1.2. 주소의 개념.....	47
2.2. 포인터와 주소.....	48
2.2.1. 포인터의 개념.....	48
2.2.2. 참조의 개념.....	48
2.2.3. Built-in Address-of and Indirection Operator.....	49
- Address-of Operator.....	49
- Indirection Operator.....	51
2.2.4. 다중 포인터.....	52
- Multidimensional Pointer Declarator.....	53
- Multidimensional Built-in Indirection Operator.....	53

2.2.5. 포인터에 붙는 다양한 cv-Qualification 들에 의한 의미의 변형.....	54
2.2.6. void*의 특수성.....	55
 2.3. 레퍼런스.....	56
2.3.1. 레퍼런스의 개념.....	56
2.3.2. Built-in Reference Declarator.....	56
2.3.3. 레퍼런스와 포인터의 공통점과 차이점.....	57
 3. 할당.....	61
3.1. 할당의 개념.....	61
3.2. 정적 할당과 동적 할당 각각의 할당 메커니즘.....	61
3.2.1. 정적 할당 메커니즘.....	61
3.2.2. 동적 할당 메커니즘.....	62
3.2.2.1. Memory Allocator.....	62
3.2.2.2. Chunk.....	63
- In-use chunk.....	65
- Free chunk.....	66
- Top Chunk.....	66
- Last Remainder chunk.....	66
3.2.2.3. Bin.....	66
3.2.2.4. Arena.....	68
3.3. C 스타일 동적 할당.....	69
3.3.1. malloc.....	69
3.3.2. calloc.....	69
3.4. C++ 스타일 동적 할당.....	71
3.4.1. new, delete Operator.....	71
3.4.2. Smart Pointer.....	76
- std::unique_ptr.....	77
- std::shared_ptr.....	77
- std::weak_ptr.....	78
3.4.3. std::Allocator<T>.....	79
3.5. 동적 배열.....	80
3.6. 동적 할당에서 주의할 점.....	81
3.6.1. Dangling Pointer.....	81
3.6.2. Memory Leak.....	81
3.6.3. Pointer Double Release.....	82
3.7. RAI ^I 패턴.....	82

4. 흐름 제어	83
4.1. 소프트웨어의 성능 지표	83
4.2. 정보의 흐름 제어	84
4.2.1. 정보의 흐름을 제어한다는 것	84
4.2.2. 반복	84
4.2.2.1. 반복의 종류	84
- Linear Iteration	84
- Linear Recursion	85
- Tree-Recursion	86
4.2.2.2. 반복문의 종류 및 문법	86
- while	86
- for	87
4.2.2.3. 한 번쯤 생각해보면 좋을 것 같은 질문들	88
- 반복으로 처리할 수 있다고 반복을 쓰는 것이 무조건 좋은 걸까?	88
- 어떤 상황에서는 어떤 반복문을 사용하는 것이 좋을까?	92
- 반복문을 중첩한다면 어떤 순서로 어떻게 중첩하는 것이 좋을까?	92
- 다차원 배열을 다중 반복문으로 순회한다면 어떤 순서로 순회하는 것이 좋을까?	94
- 어떻게 더 반복문의 반복을 줄일 수 있을까? - break, return	100
- 추가적으로 직접 찾아보고 생각해보면 좋을만한 질문들	101
4.2.2.4. 반복 처리를 줄이기 위해 혹은 보다 효율적으로 처리하기 위해 고려 가능한 방안들	102
4.2.3. 조건	102
4.2.3.1. 조건의 종류	102
- 레이블링 조건(ex. Dictionary, Hash Table)	102
- 일반 조건	102
4.2.3.2. 조건문의 종류 및 문법	102
- if, else~if, else	102
- switch	103
4.2.3.3. 한 번쯤 생각해보면 좋을 것 같은 질문들	103
4.2.3.4. 조건 처리를 줄이기 위해 혹은 보다 효율적으로 처리하기 위해 고려 가능한 방안들	103
5. 함수	105
5.1. 함수의 종류	105
5.1.1. 종류	105
5.1.2. 함수의 동작에 변수를 주는 요소들	105
5.1.3. 매개변수 및 리턴값의 종류 및 성능 비교	106
- 클래스의 Call By Value에 의한 전달 비용	107
- 클래스의 Call By Pointer / Call By Reference에 의한 전달 비용	108
- 클래스의 보다 효율적인 전달 방식	109

- Built-In Type의 Call By Value에 의한 전달 비용.....	110
- Built-In Type의 Call By Pointer / Call By Reference에 의한 전달 비용.....	110
- Built-In Type의 보다 효율적인 전달 방식.....	110
5.1.4. 특수한 함수.....	110
- 순수 함수(수학 함수).....	110
5.2. 일반적인 함수.....	111
5.2.1. 일반적인 함수의 문법.....	111
5.2.1.1. 함수 선언.....	111
- 함수 선언문의 구성.....	111
- 함수 정의.....	111
- 값 반환.....	112
- inline 함수.....	115
- constexpr 함수.....	117
5.2.1.2. 인자 전달.....	117
- 리스트 인자.....	117
- 인자의 개수가 정해지지 않은 경우.....	118
- 기본 인자.....	119
5.2.1.3. 오버로딩 함수.....	119
- 자동 오버로딩 해결.....	120
- 오버로딩과 반환 타입.....	120
- 오버로딩과 유효 범위.....	120
5.2.1.4. 함수 포인터.....	121
5.2.2. 함수의 생애 주기에 따른 내부 동작.....	122
5.2.2.1. Compile Time.....	122
5.2.2.1.1. Compiler(MSVC, Visual C++).....	123
- inline 함수 선별 및 처리.....	123
- constexpr 함수 Compile Time 처리.....	126
- reinterpret_cast<T>에 따른 Type 변환 처리.....	127
- 함수 오버로딩 해결.....	127
5.2.2.1.2. Linker.....	129
- 함수 링킹.....	129
5.2.2.2. Run Time.....	131
5.2.2.2.1. 함수 호출.....	131
- 함수 매개변수 초기화.....	132
5.2.2.2.2. 함수 내 처리.....	132
- Exception Handling.....	132
5.2.2.2.3. 함수 탈출(종료).....	132
- 반환값 커뮤니케이션.....	132
5.3. 매크로 함수.....	133
6. 익명 함수(람다).....	135

6.1. 익명 함수(람다, 람다 표현식).....	135
6.1.1. 익명 함수의 문법.....	135
6.1.1.1. 구현 모델.....	136
6.1.1.2. 캡쳐.....	139
- 람다의 수명.....	139
- 네임스페이스 이름.....	140
- 람다와 this.....	140
6.1.1.3. 호출과 반환.....	141
- 람다를 함수 포인터처럼 사용.....	141
6.1.1.4. 람다의 타입.....	143
7. 클래스 - 기초.....	145
7.1. ADT의 개념.....	145
7.1.1. ADT가 필요한 예.....	145
7.1.2. ADT를 사용할 때 좋은 점.....	146
- 구현 세부 사항을 감출 수 있다.....	146
- 변경이 전체에 영향을 미치지 않는다.....	146
- 성능을 향상시키기 쉽다.....	146
- 프로그램이 명백해진다.....	146
- 프로그램의 가독성이 높아진다.....	146
- 전체 프로그램에 데이터를 넘길 필요가 없다.....	147
7.2. 클래스의 개념(ADT + 상속 + 다형성).....	148
7.3. 클래스, 인스턴스, 객체의 개념.....	148
7.4. 클래스의 문법.....	149
7.4.1. 클래스.....	149
7.4.1.1. 멤버 함수.....	149
7.4.1.2. 기본 복사.....	150
7.4.1.3. 접근 제어.....	151
7.4.1.4. class와 struct.....	154
7.4.1.5. 생성자.....	154
7.4.1.6. explicit 생성자.....	157
7.4.1.7. 가변성.....	158
- 상수 멤버 함수.....	158
- 불리적 및 논리적 상수성.....	158
- mutable.....	159
7.4.1.8. 자기 참조.....	160
7.4.1.9. 멤버 접근.....	161
7.4.1.10. static 멤버.....	162
8. 클래스의 보다 깊은 이해를 위해 - 이동 동작.....	165

8.1. 이동 의미론, 완벽 전달의 개념.....	165
8.2. 전달 참조(Forwarding Reference)와 오른값 참조.....	167
8.3. 참조 축약 규칙.....	171
8.4. std::move와 std::forward.....	173
- std::move.....	173
- std::forward.....	174
- 오른값 참조에는 std::move를, 전달 참조에는 std::forward를 사용하라.....	175
8.5. 완벽 전달이 실패하는 경우들.....	177
- 중괄호 초기치.....	177
- 널 포인터를 의미하는 0 혹은 NULL.....	180
- 선언만 된 정수 static const 및 constexpr.....	180
- 오버로딩된 함수 이름과 템플릿 이름.....	183
- 비트 필드.....	184
9. 클래스 - 상속.....	187
9.1. 파생 클래스.....	187
9.1.1. 파생 클래스.....	189
9.1.1.1. 멤버 함수.....	190
9.1.1.2. 생성자와 소멸자.....	191
9.1.2. 클래스 계층 구조.....	197
9.1.2.1. 클래스 계층 구조에 유연함을 부여하는 요소.....	198
9.1.2.1.1. 타입 필드.....	203
9.1.2.1.2. 가상 함수.....	203
- 가상 함수의 종류와 사용법.....	203
- 다형성의 동작 원리.....	205
9.1.2.1.3. 명시적 한정.....	214
9.1.2.1.4. 재정의 제어.....	215
- 다형성 override.....	215
- 이름 가리기 override.....	217
- override 키워드.....	218
- final 키워드.....	219
9.1.2.1.5. using 기반 클래스 멤버.....	219
- 생성자 상속.....	221
9.1.2.1.6. 반환 타입 완화.....	225
9.1.2.2. 클래스 계층 구조의 설계.....	226
9.1.2.2.1. 상속 접근 제어.....	241
- public 상속.....	242
- protected 상속.....	243
- private 상속.....	244

9.1.2.2. 구현 상속.....	245
9.1.2.3. 인터페이스 상속.....	246
9.1.2.3. 다중 상속.....	246
9.1.2.3.1. 다중 인터페이스.....	246
9.1.2.3.2. 다중 구현 클래스.....	246
9.1.2.3.3. 모호성 해결.....	247
9.1.2.3.4. 기반 클래스의 반복 사용.....	250
9.1.2.3.5. 가상 기반 클래스.....	251
9.1.3. 멤버를 가리키는 포인터.....	255

1. 값과 변수 그리고 표현식

1.1. 변수의 개념

1.1.1. 변수의 정의

프로그램의 동작이란 따지고 보면 결국 어떤 메모리 혹은 보조 저장장치의 값을 읽거나 바꾸는 것의 연속일 뿐이다

다만, 거기에 조금씩의 제약을 가함으로서 규칙을 정해 제어를 가할 뿐인 것이다

우리가 프로그램이라고 말하는 것들도 결국 OS에 의해 실행될 때에는 프로세스로서 활용할 수 있는 일정한 크기의 메모리 영역을 받고, 그 영역 내의 메모리의 값을 조작할 뿐인 것이다

다만, 이때 실질적으로 프로세스 수준에서 메모리를 조작할 때 활용할 수 있는 손발과 같은 도구는 몇 개 안되는 레지스터가 전부겠지만, 프로그래머가 모든 프로세서, 메모리 크기, 레지스터의 갯수와 크기 등등의 다양한 고려요소들을 파악해가며 어셈블리를 작성하기란 쉽지 않을 것이다

때문에 이러한 어셈블리를 다루는 일은 컴파일러에 맡겨두고 우리는 메모리를 조작할 때의 직접적인 손발인 레지스터를 다루기 보단 그보다 추상화된, 즉, 조금 더 상상하기 편하고 다루기 편한 변수란 개념을 도구로서 다루며 프로그램을 조직하는 것이다

즉, 정리하자면 우리가 다루는 변수라는 도구는 엄밀히 말해서 해당 프로세스에 예약된 메모리 청크의 일부이며, 이름인 식별자를 지정함으로써 프로그래머 수준에서는 메모리에 직접 참조하는 것이 아닌 식별자를 대신 참조할 수 있도록 해줌으로써 보다 사람이 문제를 사고하는 데 편리하도록 해주는 것이다

1.1.2. 값과 변수 그리고 표현식의 관계

값(Value)은 광범위한 개념으로서 매우 많은 것들을 다루게 되는데, 쉽게 말하자면 변수는 값의 부분집합에 불과하다고 생각하면 된다

그리고 표현식(Expression)이란 이러한 값을 피연산자로, 그리고 Built-In Operator를 비롯한 다양한 사용자 지정 연산자인 Overrided Operator를 연산자로 하여 계산을 하는 식(Statement)을 말하는 것이다

[1.4. 값의 범주]에서 다루겠지만 앞서 그 느낌만 조금 설명하자면 C++ 표준에서 정의하는 값과 변수 그리고 표현식의 관계를 잘 보다보면 변수 부분의 표준에 비해 값과 표현식 그 자체의 정의 부분은 조금 더 복잡하고 이해하기 어려운 감이 있다

예를 들어 우리가 변수를 다룰 때에는 굉장히 민감한 코드를 작성하는 것이 아니라면 일반적인 기술적 이슈들, 예컨대 타입, 식별자, 캐스팅, 알고리듬, 아키텍처 등등의 것들만 신경쓰면 되게 되어있다. 즉, 우리가 코드 그 자체에서 표면적으로 보이는 부분들의 개념은 매우 직관적이며, 해당 개념들 사이에서 일어나는 변환¹과 같은 것들도 조금의 설명만으로도 충분히 이해하고 납득할만한 규칙이 규정되어있다

¹ 대표적으로 Type Casting

그러나 곁으로 보이는 코드의 내부 동작을 담당하는 값과 표현식에 대한 세부 개념과 개념 사이의 변환이 직관적이라는 느낌이 들지는 않는다.

물론, 이러한 세부적인 값과 표현식에 대한 정확한 표준상의 정의는 컴파일러 제작사나 제작자를 위한 정보인 경향이 있지만, 이러한 세부 용어의 개념과 동작을 안다는 것은 디버깅, 최적화 등에 많은 도움이 될 수 있을 것이라 생각한다

예를 들면 다음과 같은 경우가 있다

```
int main()
{
    int arr[] = {, 0, 1, 2};

    return 0;
}
```

위와 같은 코드가 있다고 해보자. 이를 컴파일하고자 하면 컴파일러는 다음과 같은 에러 메시지를 반환한다

```
error: expected primary-expression before ',' token
11 |     int arr[] = {, 0, 1, 2};
   |           ^
```

문제를 해결하려면 물론 간단하게 구글링을 해볼 수도 있겠지만, 본인의 순수한 지식만으로 빠르게 문제를 해결하고자 하자면 에러 메시지에 있는 용어들부터 알아야할 것이다
즉, 이 상황에선 에러 메시지가 ‘,’ 토큰 전에 primary-expression이 있어야할 것 같다고 하는데 이 문장을 이해하려면 C++ 표준에서 정의하는 표현식의 범주 중 primary-expression이라는 용어를 알아야하는 것이다

또 다른 예도 존재한다

```
#include <iostream>

int main()
{
    int num = 0;
    int&& rf_1 = num++; // Fine
    //int&& rf_2 = ++num; // Error : rValue 참조를 lValue에 바인딩할
                           // 수 없습니다

    return 0;
}
```

위와 같은 코드가 있다고 하자. 아주 간단한 증감 연산자와 관련한 문제이다. '&&'이라는 Declarator가 나오는데, 이는 [2. 메모리와 포인터 그리고 레퍼런스]에서 말하겠지만 앞서서 말하자면 rValue를 참조하는 레퍼런스 변수를 만들 때 사용되는 Declarator이다
이러한 코드를 마주했을 때 제대로 이해하고, 또 이러한 코드를 작성하기 위해서는 C++ 표준에서 정의하는 값의 범주 중 prValue, glValue, rValue, xValue, lValue에 대해서 알고, 또한 각 값의 범주 간의 변환 규칙에 대해 알아야만 할 것이다

물론, C++에서는 간단한 변수, 즉, 객체를 다루며, 이러한 객체가 될 수 있는 것들로는 단순히 변수를 넘어 구조체, 열거체, 클래스, 함수 등의 많은 것들이 있다
그러나 이러한 것들도 결국에는 가장 기본적인 요소인 변수를 포함할 수 밖에 없기에 우선 변수와 관련 값과 표현식의 동작을 알고 넘어가고, 추후의 회차들에서 각각의 개념들과 관련된 값과 표현식의 동작을 알게 되면 C++에 대해 보다 깊은 이해를 가질 수 있을 것이다

1.1.3. 변수의 생성

- 선언의 개념

선언이란 프로그램에 변수, 더 나아가 객체의 이름을 알려주는 방법이다
그러나 주의해야 할 점이 있다. 프로그램에서 실질적으로 무언가를 사용하기 위해서는 선언 뿐만이 아니라 정의하는 것도 필요하다

[1. 값과 변수 그리고 표현식]에서 다루는 Fundamental Types에 대한 변수들의 경우에는 모든 선언이 곧 정의와 직결된다
그러나 Fundamental Types의 변수의 선언과는 다르게 예외인 경우들이 많이 존재한다.
이러한 예외들은 추후의 회차들에서 다루겠으니 여기선 프로그램에서 무언가를 사용하려면 선언을 하고 그 후에 정의를 해주어야만 한다는 순서만 기억해주길 바란다

이제 변수의 생성에 관련해서 표준에서 정의하는 구문들만을 쭉 설명한 다음 마지막에 간단한 예제 코드를 두고 하나하나 비교해가며 확인해보도록 하겠다

- Simple declaration

여기선 C++ 표준에서 정의하는 가장 단순한 선언 구문에 대해 이야기하겠다
먼저 표준에 있는 구문 정의를 그대로 가져오면 다음과 같다

decl-specifier-seq init-declarator-list(optional) ;	(1)
attr decl-specifier-seq init-declarator-list;	(2)

attr	(since C++11) sequence of any number of attributes
decl-specifier-seq	sequence of specifiers
init-declarator-list	comma-separated list of declarators with optional initializers. init-declarator-list is optional when declaring a named class/struct/union or a named enumeration

- Attributes

Attribute에 관해 표준에서 정의하는 구문 정의는 다음과 같다

[[attribute-list]]	(1)
[[using attribute-namespace : attribute-list]]	(2)

attribute-list 자리에는 다음의 것들이 들어갈 수 있다

identifier	<1>
attribute-namespace :: identifier	<2>
identifier (argument-list)	<3>
attribute-namespace :: identifier (argument-list)	<4>

- Specifiers

표준에서 정의하는 Specifier 중 간단한 type specifier를 제외하고 변수에 적용 가능한 대표적인 Specifier들은 다음과 같다

typedef specifier
inline specifier
constexpr specifier
constinit specifier
static specifier

- Declarators

Declarators에 관해 표준에서 정의하는 구문 정의는 다음과 같다

declarator initializer(optional)	(1)
declarator requires-clause	(2)

declarator	the declarator
initializer	optional initializer (except where required, such as when initializing references or const objects). See Initialization for details
requires-clause	a requires-clause, which adds a constraint to a function

	declaration
--	-------------

declarator 자리에는 다음의 것들이 올 수 있다

unqualified-id attr(optional)	<1>
qualified-id attr(optional)	<2>
... identifier attr(optional)	<3>
* attr(optional) cv(optional) declarator	<4>
nested-name-specifier * attr(optional) cv(optional) declarator	<5>
& attr(optional) declarator	<6>
&& attr(optional) declarator	<7>
noptr-declarator [constexpr(optional)] attr(optional)	<8>
noptr-declarator (parameter-list) cv(optional) ref(optional) except(optional) attr(optional)	<9>

- Initializer

Initializer에 관해 표준에서 정의하는 구문 정의는 다음과 같다

(expression-list)	(1)
= expression	(2)
{ initializer-list }	(3)
{ designated-initializer-list }	(4)

- 예제

1번 예제

```
const int num;
```

- *const int num;* // *decl-specifier-seq init-declarator-list(optional)* ;
 - *const int* // *cv-qualifier(specifier), simple type specifier*
 - *num* // *declarator initializer(optional)*
 - *num* // *unqualified-id attr(optional)*

이 예제에서 `const int num;`은 Declaration 구문 중 attr이 없음으로 decl-specifier-seq init-declarator-list로 사용되었음을 알 수 있다

decl-specifier-seq 부분에는 `int`라는 Type Specifier와 함께 `const`라는 cv-qualifier Specifier가 구성하고 있다

init-declarator-list 부분에는 Declarators 구문 중 declarator initializer(optional)를 사용하고 있으며, declarator 부분에 unqualified-id attr(optional) 구문을 사용하고 있다

2번 예제

```
#define SOME_ATTR  
[[SOME_ATTR]] constinit const static bool flag = false;
```

- *[[SOME_ATTR]] constinit const static bool flag = false;* // *attr decl-specifier-seq init-declarator-list*;
 - *[[SOME_ATTR]]* // *attr*
 - *SOME_ATTR* // *identifier*
 - *constinit const static bool* // *cv-qualifier(specifier), constinit specifier, static specifier, simple type specifier*
 - *flag = false* // *declarator initializer(optional)*
 - *flag* // *unqualified-id attr(optional)*
 - *= false* // *= expression*

이 예제에서 `[[SOME_ATTR]] constinit const static bool flag = false;`은 Declaration 구문 중 attr이 있음으로 attr decl-specifier-seq init-declarator-list로 사용되었음을 알 수 있다

attr 부분에는 사용자 지정 attribute가 사용되었음을 알 수 있다

decl-specifier-seq 부분에는 `bool`라는 Type Specifier와 함께 다양한 Specifier가 사용되었는데, 해당 변수가 정적 시간에 생성되기를 강제하는 `constinit`과 이를 지키기 위한 `static` 그리고 `cv-qualifier specifier`로서 `const`가 사용되고 있다

init-declarator-list 부분에는 Declarators 구문 중 declarator initializer(optional)를 사용하고 있으며, declarator 부분에 unqualified-id attr(optional) 구문을 사용하고 있다

그리고 initializer(optional) 부분에 해당하는 '= false'는 Initializer 구문 중 '= expression'를 사용하고 있다

initializer(optional) 부분은 선택 사항인데도 굳이 사용하게 된 것은 앞의 init-declarator-list에서 사용된 cv-qualifier specifier인 const의 특징이 객체가 선언됨과 동시에 초기화하기를 강제하기 때문이다

1.2. C++에서의 Fundamental Types와 Built-In Operators

1.2.1. Fundamental Types

- **Void Type**
 - void
 - std::nullptr_t
- **Signed and Unsigned Integer Types**

Type specifier	Equivalent type	Width in bits by data model									
		C++ standard	LP32	ILP32	LLP64	LP64					
<code>signed char</code>	<code>signed char</code>	at least 8	8	8	8	8					
<code>unsigned char</code>	<code>unsigned char</code>										
<code>short</code>	<code>short int</code>	at least 16	16	16	16	16					
<code>short int</code>											
<code>signed short</code>											
<code>signed short int</code>											
<code>unsigned short</code>	<code>unsigned short int</code>	at least 16	16	32	32	32					
<code>unsigned short int</code>											
<code>int</code>	<code>int</code>										
<code>signed</code>											
<code>signed int</code>											
<code>unsigned</code>	<code>unsigned int</code>	at least 32	32	32	64	64					
<code>unsigned int</code>											
<code>long</code>	<code>long int</code>										
<code>long int</code>											
<code>signed long</code>											
<code>signed long int</code>											
<code>unsigned long</code>	<code>unsigned long int</code>	at least 64	64	64	64	64					
<code>unsigned long int</code>											
<code>long long</code>	<code>long long int</code> (C++11)										
<code>long long int</code>											
<code>signed long long</code>											
<code>signed long long int</code>											
<code>unsigned long long</code>	<code>unsigned long long int</code> (C++11)										
<code>unsigned long long int</code>											

- **Boolean Type**
 - `bool`
- **Character Types**
 - `char`
 - `wchar_t`
 - `char8_t`
 - `char16_t`
 - `char32_t`
- **Floating-Point Types**
 - `float`
 - `double`
 - `long double`

1.2.2. Range of Values

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		-128 to 127
		unsigned		0 to 255
	16	UTF-16		0 to 65535
	32	UTF-32		0 to 1114111 (0x10ffff)
integer	16	signed	$\pm 3.27 \cdot 10^4$	-32768 to 32767
		unsigned	0 to $6.55 \cdot 10^4$	0 to 65535
	32	signed	$\pm 2.14 \cdot 10^9$	-2,147,483,648 to 2,147,483,647
		unsigned	0 to $4.29 \cdot 10^9$	0 to 4,294,967,295
	64	signed	$\pm 9.22 \cdot 10^{18}$	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to $1.84 \cdot 10^{19}$	0 to 18,446,744,073,709,551,615
binary floating point	32	IEEE-754 🔗	<ul style="list-style-type: none"> min subnormal: $\pm 1.401,298,4 \cdot 10^{-45}$ min normal: $\pm 1.175,494,3 \cdot 10^{-38}$ max: $\pm 3.402,823,4 \cdot 10^{38}$ 	<ul style="list-style-type: none"> min subnormal: $\pm 0x1p-149$ min normal: $\pm 0x1p-126$ max: $\pm 0x1.fffffepp+127$
			<ul style="list-style-type: none"> min subnormal: $\pm 4.940,656,458,412 \cdot 10^{-324}$ min normal: $\pm 2.225,073,858,507,201,4 \cdot 10^{-308}$ max: $\pm 1.797,693,134,862,315,7 \cdot 10^{308}$ 	<ul style="list-style-type: none"> min subnormal: $\pm 0x1p-1074$ min normal: $\pm 0x1p-1022$ max: $\pm 0x1.fffffffffffffp+1023$
	64	IEEE-754 🔗	<ul style="list-style-type: none"> min subnormal: $\pm 3.645,199,531,882,474,602,528 \cdot 10^{-4951}$ min normal: $\pm 3.362,103,143,112,093,506,263 \cdot 10^{-4932}$ max: $\pm 1.189,731,495,357,231,765,021 \cdot 10^{4932}$ 	<ul style="list-style-type: none"> min subnormal: $\pm 0x1p-16446$ min normal: $\pm 0x1p-16382$ max: $\pm 0x1.fffffffffffffp+16383$
			<ul style="list-style-type: none"> min subnormal: $\pm 6.475,175,119,438,025,110,924,438,958,227,646,552,5 \cdot 10^{-4966}$ min normal: $\pm 3.362,103,143,112,093,506,262,677,817,321,752,602,6 \cdot 10^{-4932}$ max: $\pm 1.189,731,495,357,231,765,085,759,326,628,007,016,2 \cdot 10^{4932}$ 	<ul style="list-style-type: none"> min subnormal: $\pm 0x1p-16494$ min normal: $\pm 0x1p-16382$ max: $\pm 0x1.ffffffffffffffffpp+16383$
	80 ^[note 1]	x86 🔗		
	128	IEEE-754 🔗		

1.2.3. Built-In Operators

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<pre>a = b a += b a -= b a *= b a /= b a %= b a &= b a = b a ^= b a <= b a >= b</pre>	<pre>++a --a a++ a--</pre>	<pre>+a -a a + b a - b a * b a / b a % b ~a a & b a b a ^ b a < b a > b a == b a != b a <= b a >= b a <= b a >= b</pre>	<pre>!a a && b a b</pre>		<pre>a[...] *a &a a->b a.b a->b a.*b</pre>	<pre>function call a(...)</pre>
						<pre>comma a, b</pre>
						<pre>conditional a ? b : c</pre>
Special operators						
<pre>static_cast converts one type to another related type dynamic_cast converts within inheritance hierarchies const_cast adds or removes cv-qualifiers reinterpret_cast converts type to unrelated type C-style cast converts one type to another by a mix of static_cast, const_cast, and reinterpret_cast new creates objects with dynamic storage duration delete destructs objects previously created by the new expression and releases obtained memory area sizeof queries the size of a type sizeof... queries the size of a parameter pack (since C++11) typeid queries the type information of a type noexcept checks if an expression can throw an exception (since C++11) alignof queries alignment requirements of a type (since C++11)</pre>						

1.2.4. Placeholder Type Specifiers - auto Type

auto는 변수의 생성 시에 선언의 initializer를 보고 자동으로 적절한 Type을 추론해준다

표준에서 정의하는 Placeholder Type Specifiers의 구문은 다음과 같다

type-constraint(optional) auto	(1)
type-constraint(optional) decltype (auto)	(2)

```
#include <iostream>

int main()
{
    auto a = 1 + 2;           // type of a is int
    auto b = 1 + 1.2;         // type of b is double
    static_assert(std::is_same_v<decltype(a), int>);
    static_assert(std::is_same_v<decltype(b), double>);

    auto c0 = a;              // type of c0 is int, holding a copy of a
```

```

 decltype(auto) c1 = a;      // type of c1 is int, holding a copy of a
 decltype(auto) c2 = (a);   // type of c2 is int&, an alias of a
 std::cout << "before modification through c2, a = " << a << '\n';
 ++c2;
 std::cout << " after modification through c2, a = " << a << '\n';

 //auto int x;           // valid C++98, error as of C++11
 //auto x;               // valid C, error in C++
}

```

- auto Type Deduction

Placeholder Type Specifiers의 표준에서 정의하는 구문 중 어느것을 사용하느냐에 따라 다른 Type Deduction 방식을 사용한다

만약 auto를 사용하게 된다면 Template Argument Deduction을 사용하게 된다

만약 decltype(auto)를 사용하게 된다면 decltype(expr)에 따라 Type Deduction을 계산하게 되며, 이때 expr은 Initializer가 된다

이때 규칙이 조금 복잡하게 느껴질 수 있는데 정리하자면 다음과 같다

expr0 entity일 경우	인수가 괄호로 묶이지 않은 id-expression일 경우 decltype은 entity의 유형을 생성한다 여기서 entity란 Values, Objects, References, Structured Bindings, Functions, Enumerators, Types, Class Members, Templates, Template Specializations, Parameter Packs, 그리고 Namespaces를 말한다
expr0 expression일 경우	a) if the value category of expression is xValue, then decltype yields T&&; b) if the value category of expression is lValue, then decltype yields T&; c) if the value category of expression is prValue, then decltype yields T.

이해를 돋기 위한 예제는 다음과 같다

```

// decltype ( entity )
auto val_0 = num;

// decltype ( expression )
auto val_1 = true ? a : a; // expression is xValue
auto val_2 = ++num;        // expression is lValue
auto val_3 = (num);        // expression is lValue
auto val_4 = num++;        // expression is prValue

```

```
int&& rf = num++; // This expression is identical to the  
// expression above, but no transformation  
// between the value categories occurred
```

1.3. 값의 해석

1.3.1. C++ Data Model과 Type Size

Data Model은 플랫폼 별로 그 목적에 따라 ‘우리 플랫폼에서 어떤 Type은 이러한 크기일 때 보다 효율적이다’라고 생각하는 바에 따라 적절하게 선택적으로 적용하는 것이다

Data Model의 Model 이름에는 대문자 알파벳들과 32 또는 64의 숫자가 나타나게 되는데, 그 의미는 다음과 같다

- S는 short Type을 의미한다
- I는 int Type을 의미한다
- L은 long Type을 의미한다
- LL은 long long type을 의미한다
- P은 Pointer Type을 의미한다

이때, 이 알파벳들의 나열과 숫자를 함께 해석하게 되는데, 숫자의 의미는 그 앞쪽에 나타나는 알파벳에 해당하는 Type들의 크기가 해당 Model에선 숫자만큼의 크기라는 것이다. 다만, 앞에 등장하지 않은 Type의 경우 C++ 표준에서 정의하는 최소 크기를 가진다

즉, 예를 들어 만약 ILP64라는 Model이 있을 경우에는 int, long, pointer의 크기는 64 bit이지만, 언급되지 않은 short와 long long은 각각 표준에서 정의하는 최소 크기인 16 bit와 64 bit의 크기를 가지게 된다

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
signed char	signed char	at least 8	8	8	8	8
unsigned char	unsigned char					
short						
short int						
signed short						
signed short int						
unsigned short						
unsigned short int	unsigned short int					
int						
signed						
signed int						
unsigned						
unsigned int	unsigned int					
long						
long int						
signed long						
signed long int						
unsigned long						
unsigned long int	unsigned long int					
long long						
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int					
unsigned long long int	(C++11)					

Data model	short (integer)	int	long (integer)	long long	pointers, size_t	Sample operating systems
ILP32	16	32	32	64	32	x32 and arm64i/p32 ABIs on Linux systems; MIPS N32 ABI.
LLP64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64) using Visual C++; and MinGW
LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g., Solaris, Linux, BSD, macOS. Windows when using Cygwin; z/OS
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to the SPARC64
SILP64	64	64	64	64	64	Classic UNICOS ^[46] ^[47] (versus UNICOS/mp, etc.)

1.3.2. Scope와 변수의 관계

Scope라는 것은 코드를 보면 한 눈에 파악하기 쉬우니 먼저 예제부터 보고 시작하겠다

```
int main()
{
    int i = 0;          // scope of outer i begins
    ++i;               // outer i is in scope

    {
        int i = 1;      // scope of inner i begins,
                        // scope of outer i pauses
        i = 42;         // inner i is in scope
    }                  // block ends, scope of inner i ends,
                        // scope of outer i resumes
}
//int j = i;          // error: i is not in scope
```

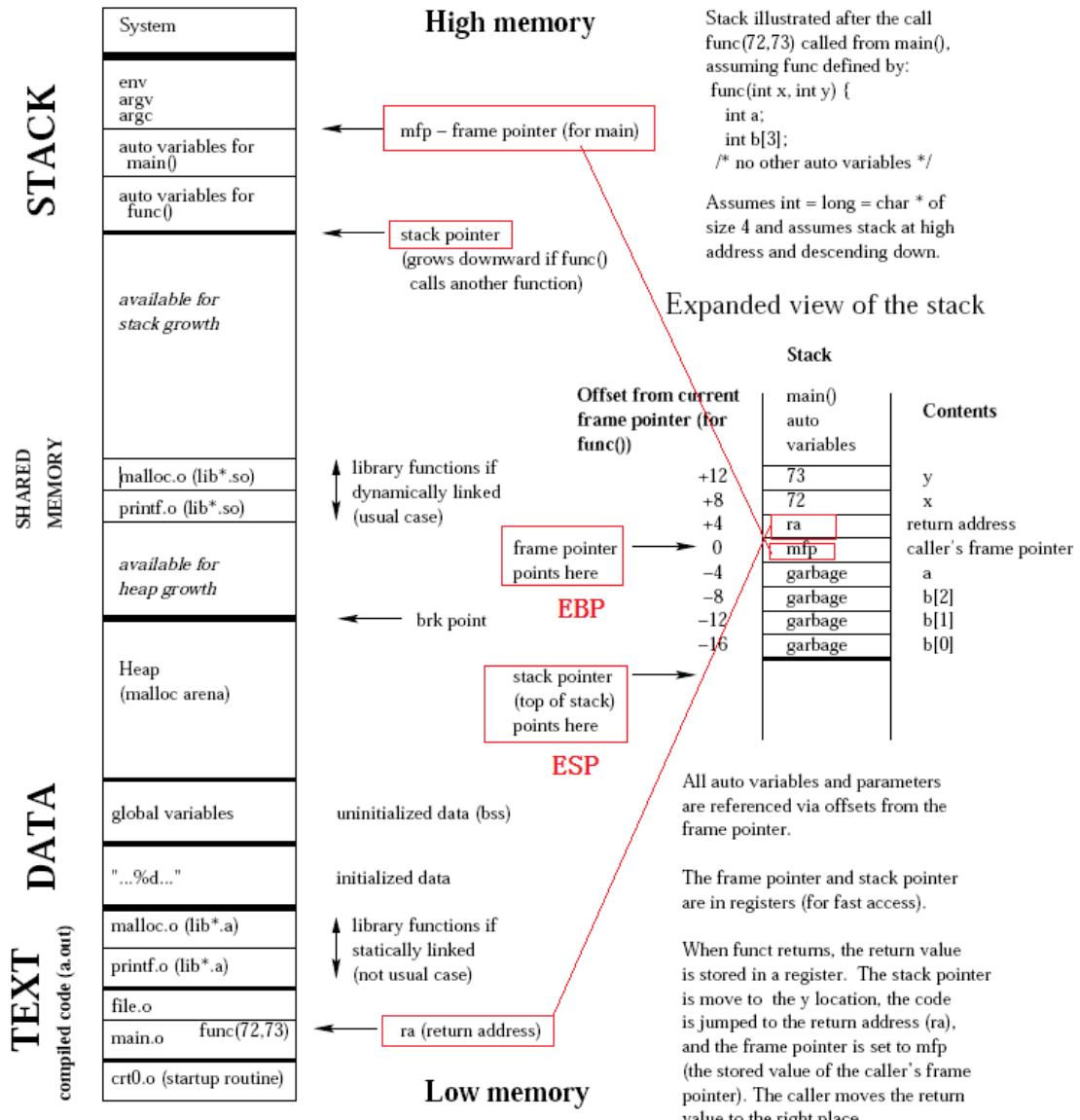
위의 예제에서 볼 수 있는 '{'로 시작하고 '}'로 끝나는 것이 바로 Scope하고 하는 것이며, 이들은 반드시 짹을 이뤄야만 한다

즉, 위의 예제 코드에서는 총 두 개의 Scope가 존재하는 것이며, Scope는 위에서 보듯이 중첩될 수 있다는 것을 알 수 있다

Scope라고 하는 것은 Block이라고도 한다

- Scope와 변수의 생명 주기

Scope는 변수의, 엄밀히는 Stack에 저장되는 변수들의 생명 주기를 결정짓는다. 그렇다면 Stack은 무엇이고 또 Stack에 저장되는 변수들의 종류들은 무엇이냐고 묻는다면 다음의 이미지가 이를 잘 설명해주고 있다

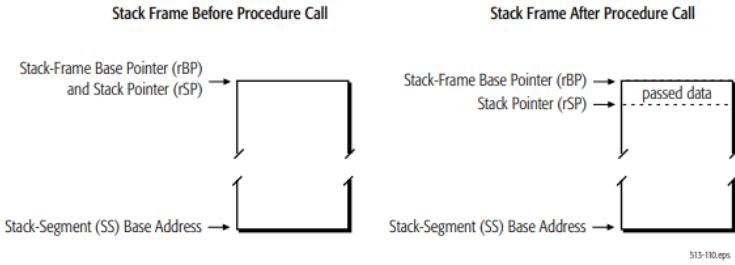


자세히 언급하기엔 문서의 범위를 넘어서기에 많은 부분을 언급할 수는 없지만, 개중 자세히 살펴봐야 할 부분은 이미지 왼쪽의 직사각형의 윗부분에 'STACK'이라고 적혀 있는 부분이다. 조금 더 보기 편하게 정리하면 다음과 같다.

env argv argc	<ul style="list-style-type: none"> a) env는 OS의 환경 변수들을 말하는 것이다 b) argv와 argc는 프로그램이 실행 시 주어지는 매개변수의 개수와 내용을 말하는 것이다 실제로 지금까지 예제 코드는 main 함수의 형태가 'int main () { body }'였지만, 'int main (int argc, char *argv[]) { body }'와 같은 형태도 C++ 표준에서는 정의하고 있다
auto variables for main()	main 함수의 Scope 안에서 동적 할당이나 정적 선언이 아닌 방식으로 선언된 변수를 말한다
auto variables for func()	main 함수는 아닌 다른 어떤 함수의 Scope 안에서 동적 할당이나 정적 선언이 아닌 방식으로 선언된 변수를 말한다

available for stack growth

Stack은 다음의 이미지와 같이 동작하게 된다



즉, Stack이라는 것은 다음의 총 세 개의 지표로 구성이 되게 된다

- Stack이 시작하는 점(낮은 주소)(Stack-Segment Base Address)
- Stack의 크기를 지정하는 점(높은 주소)(Stack-Frame Base Pointer)
- 높은 주소에서 점점 낮게 내려오면서 쌓여가는 데이터 중 가장 최근에 추가된 데이터를 가리키는 점(Stack Pointer)

여기서 말하는 available for stack growth라는 것은 이미지에서 보이는 Stack Pointer와 Stack-Segment Base Address 사이의 앞으로 데이터가 수납될 수 있는 예약된 공간을 의미하는 것이다

요약하자면 Scope에 의해 생명 주기가 결정되는 변수란 우리가 코드에서 작성하며 제어할 수 있는 한도 내에서는 어떠한 함수의 Scope 안에 선언된 동적 할당이나 정적 선언이 아닌 방식으로 선언된 변수라는 것이다

그렇다면 변수의 생명 주기가 Scope에 의해 결정된다는 것은 무슨 이야기인가 하면 먼저 변수의 내부 메커니즘 부터 알 필요가 있다

변수는 [1.1.1. 변수의 정의]에서 간략하게 설명하긴 했지만 결국은 프로그램이 OS에 의해 실행 가능한 형태인 프로세스로 전환되는 과정에서 사용할 수 있도록 할당받은 일정 크기의 메모리 덩어리, 그 중에서도 일부에 원한다면 어떤 값을 적어두는 것 뿐이다

즉, 우리가 프로그램에서 사용 가능한 메모리의 크기에는 한도가 있고 우리는 그걸 약간 약간 사용하는 것이다. 때문에 메모리는 이상적으로는 의미가 있는 값만이 적혀있도록 하고, 시간이 지나 더 이상 메모리에 적혀 있는 것이 의미가 없게 되면 해당 부분엔 다시 새로운 의미있는 값을 적어 재사용하는 것이 좋을 것이다

따라서 우리가 프로그램에서 사용 가능한 메모리의 크기가 한정되어있기에 변수는 필요할 때에만 존재하고, 더 이상 사용하지 않을 때에는 없애버려서 가용 공간을 남겨두어야만 할 것이다

동적 할당이나 정적 선언과 같은 방식으로 선언된 변수의 경우에는 다른 메커니즘의 생명 주기를 가지며, 사용에 주의가 필요한 부분이 많이 존재한다

그러나 [1. 값과 변수 그리고 표현식]에서 다루는 변수는 Stack에 저장되는 변수 뿐이며, 이 유형은 오로지 변수가 선언된 Scope가 코드에서 점거야는 영역만이 주요한 관심사로서 바라보면 될 뿐이기에 계산하기도 굉장히 쉽다. 그 방식은 다음과 같다

- Stack 변수는 선언된 순간부터 존재하고, 해당 변수가 선언된 Scope가 끝날 때 죽게 된다

예를 들어 다음의 예제 코드를 보자

```
int main()
{
    // Begin of main functions scope
    int num_0 = 0;          // (1)

    // lots of codes

    int num_1 = 0;          // (2)

    return 0;
} // End of main functions scope
```

예제 코드를 보면 (1)과 (2) 모두 main 함수의 Scope 안에 선언되었으며 다른 특별한 Specifier가 사용되지도 않았고 동적 할당을 하지도 않았다. 즉, num_0과 num_1은 모두 Stack에 존재하게 될 변수이다

num_0과 num_1은 모두 main 함수의 Scope가 끝나는 부분에서 (더 엄밀히 말하자면 함수가 Procedure Call Stack에서 Pop 되는 순간에) 죽게 된다. 즉, 메모리에서 지금까지 해당 변수가 사용하던 영역이 다시금 누구나 와서 다시 재사용 할 수 있는 공간으로 바뀌는 것이다
이때 앞서 말했듯 Stack 변수의 생명 주기는 변수가 선언된 순간부터 존재하게 된다. 따라서 num_0은 num_1보다는 조금 더 오래 생존하게 되는 것이다

- 중첩된 Scope와 동일한 변수의 Identifier

위쪽에 있던 예제 코드를 다시 살펴보자

```
int main()
{
    int i = 0;          // scope of outer i begins
    ++i;              // outer i is in scope

    {
        int i = 1;    // scope of inner i begins,
                      // scope of outer i pauses
        i = 42;       // inner i is in scope
    }                  // block ends, scope of inner i ends,
                      // scope of outer i resumes
}                  // block ends, scope of outer i ends
```

```
//int j = i;           // error: i is not in scope
```

이 예제에서 알 수 있는 점이 몇 개 있다

- a) 이미 함수의 Scope 안에서 새롭게 만들어진 Scope도 그대로 함수의 Scope로 취급한다. 즉, 바깥쪽 int i와 안쪽 int i는 둘 다 Stack 변수이다
- b) 중첩된 Scope 구조에서 동일한 이름의 변수를 사용할 때에는 보다 안쪽에 있는 Scope에서 사용하는 이름에 대응하는 변수는 안쪽에 있는 변수라는 것이다

1.4. 값의 범주

1.4.1. 변수(Variabile)와 리터럴(Literal)

리터럴(Literal), 엄밀히 말해 리터럴 변수(Literal Variable)는 쉽게 말해 `constexpr`한 변수의 유형이다. `constexpr`은 Compile Time에 결정되는 `const`라고 이해하면 되는데, 즉, 소스 코드에 내장된 상수로서 컴파일과 동시에 결정되는 값을 의미한다고 생각하면 된다. 이제 이 말을 하나씩 해석해보겠다.

먼저 `constexpr`이다.

`constexpr`은 `const`과 비슷한 개념으로서 결국 상수를 만들 때 사용하는 Specifier라는 점은 같다.

다만, 기존의 `const`는 상수 변수에 값을 초기화할 때 어느 순간에 초기화가가 일어나는지를 정확히는 정의하지 않지만 `constexpr`은 반드시 Compile Time에 초기화가 일어나도록 강제한다는 점에서 다르다.

그럼 Compile Time이 뭐고 그에 반대되는 Run Time이 뭐냐하면 다음과 같다

- Compile Time은 우리가 작성한 코드가 컴파일러에 의해 실행 가능한 형태인 바이너리 파일로 번역되는 과정의 시간을 Compile Time이라고 하며, 무언가가 Compile Time에 결정된다고 하면 Compile 과정 중에 그 무언가가 결정되어서 실제 프로그램이 프로세스로서 동작을 시작했을 때에는 추가적인 처리 없이 그냥 가져다 쓰기만 하면 된다는 것이다
- Run Time은 프로그램이 프로세스로서 실제 동작을 하는 시간을 말하는 것으로, 무언가가 Run Time에 결정된다고 하면 프로세스 동작 시간한 중에 무언가가 필요할 때마다 그때그때 필요한 처리들을 수행하게 되는 것이다

예를 들어 다음과 같은 예제 코드가 있다고 해보자

```
#include <iostream>

int GetNumber()
{
    std::cout << "Enter a number: ";
    int y = 0;
    std::cin >> y;

    return y;
}

int main()
{
    const int x{ 3 }; // x is a compile time constant
    constexpr int x_expr{ 3 }; // x_expr is a compile time
                             // constant
```

```

const int y{ GetNumber() };           // y is a runtime constant
//constexpr int y{ GetNumber() };     // Error : y_expr is a runtime
                                    // constant

const int z{ x + y };               // x + y is a runtime
                                    // expression
std::cout << z << '\n';          // this is also a runtime
                                    // expression

return 0;
}

```

즉, `constexpr`는 해당 상수의 초기화 값이 반드시 Compile하는 그 순간에 분명하게 계산될 수 있어야만 한다는 것이며, 따라서 Literal 변수가 `constexpr`하다는 것은 Literal 변수란 Compile Time에 정확히 연산되어 프로그램 이미지에 포함되어 있다가 프로세스로서 동작할 때에 OS에 의해 로드되는 값을 말한다는 것이다

다음의 종류가 흔히들 말하는 Literal Token들이다

- integer literals are decimal, octal, hexadecimal or binary numbers of integer type.
- character literals are individual characters of type
 - `char` or `wchar_t`
 - `char16_t` or `char32_t`
 - `char8_t`
- floating-point literals are values of type `float`, `double`, or `long double`
- string literals are sequences of characters of type
 - `const char[]` or `const wchar_t[]`
 - `const char16_t[]` or `const char32_t[]`
 - `const char8_t[]`
- boolean literals are values of type `bool`, that is `true` and `false`
- `nullptr` is the pointer literal which specifies a null pointer value
- user-defined literals are constant values of user-specified type

변수는 Literal을 포함하는 개념으로서 그 내용은 앞에서 지금껏 설명해왔던 바와 같다

1.4.2. 값의 범주와 해석

다른 개발자들과 이야기를 하거나 문서를 읽을 때 흔히 가장 많이들 접하는 값의 범주와 관련된 용어는 `LValue`와 `rValue`가 대부분일텐데 이것들은 다음과 같이 이해하면 편하다

- `LValue`는 단일 표현식 이후에도 사라지지 않고 참조할 수 있는 객체를 가리킨다. 대개 식의 왼편에 등장하는 부분을 말한다
- `RValue`는 단일 표현식 이후에는 더 이상 참조할 수 없는 임시 객체를 가리킨다. 대개 식의 오른편에 등장하는 부분을 말한다

여기서 임시 객체는 prValue가 구체화되어서 glValue로 사용되고자 할 때 발생된다
임시 객체는 해당 값(prValue)를 생성한 지점을 포함하는 Full-Expression이 완전히 평가되는
마지막 단계에 파괴되며, 이때 파괴되는 순서는 해당 표현식에서 임시 객체가 만들어진 반대
순서로 진행된다

1.4.2.1. Primary Categories

- lValue

lValue는 전통적으로 대입 표현식의 왼쪽에 나타나는 값을 묘사하기 위해 사용되던 단어기에
Left-Value의 줄임말이다

lValue의 정의는 xValue가 아닌 glValue라는 것이다

즉, 이 말의 의미는 식별자를 지닌 식별 할 수 있는 모든 객체는 lValue로서 취급하겠지만,
예외적으로 rValue 참조 같은 것을 사용하여 생명 주기를 늘린 값은 lValue로 취급하지
않는다는 것이다²

lValue는 glValue와 같이 다형성을 가진 값일 수 있으며, 클래스가 아닐 경우에는
cv-qualified일 수 있다

- prValue

prValue는 Pure-Value의 줄임말이다

prValue는 Built-In Operator의 연산 결과값으로서, 그 값에는 오로지 컴파일러만이 접근할 수
있으며 그 형태는 임시 객체의 형태로 나타나게 된다

prValue는 그 정의상 컴파일러만 접근할 수 있으며, 프로그래머의 입장에서는 주소값이
존재하지 않는 값이다. 따라서 &x++와 같이 prValue에 주소를 구하고자 할 경우 컴파일
에러가 발생하게 된다

prValue는 다형성을 가진 값일 수 없으며, cv-qualified일 수도 없다

- xValue

xValue는 Expiring-Value의 줄임말이다

xValue는 glValue 중 재사용할 수 있는 값을 말한다. 즉, rValue Reference로 참조할 수 있는
것들을 말한다

xValue는 rValue 참조에 바인딩될 수 있으며, glValue와 같이 다형성을 가진 값일 수 있으며,
클래스가 아닐 경우에는 cv-qualified일 수 있다

² rValue는 xValue를 포함하는 더 넓은 값의 범주기에 제한적으로 xValue로 치환해서 생각해도 된다

1.4.2.2. Mixed Categories

- **glValue**

glValue는 generalized-Value의 줄임말이다

glValue는 식별자를 지닌 식별 할 수 있는 있는 모든 객체는 glValue로서 평가하겠다는 것이다

glValue는 lValue와 rValue, 배열과 포인터, 함수와 포인터의 암시적 변환을 통해 암묵적으로 prValue로 평가될 수 있다

glValue는 다형성을 가진 값일 수 있으며, 클래스가 아닐 경우에는 cv-qualified일 수 있다

glValue는 Incomplete Type일 수 있으며, Incomplete Type의 종류는 다음과 같다. 아래의 Incomplete Type을 제외한 나머지 모든 Type은 Complete Type이다

- 선언은 되었으나 정의는 되지 않은 class(Forward Declaration)
- 범위를 모르는 array
- Incomplete Type의 array
- 정확한 내부 동작 Type이 지정되지 않은 enum

- **rValue**

rValue는 전통적으로 대입 표현식의 오른쪽에 나타나는 값을 묘사하기 위해 사용되던 단어기에 Right-Value의 줄임말이다

rValue는 prValue와 xValue를 포함하는 값의 범주이다

1.4.3. 값의 범주의 활용

1번 예제

```
int main()
{
    int num = 10;
    int temp = num++;
    int&& rf = num++;
}
```

만약 위과 같은 코드에서 num++까지만 작성했다면 이는 prValue로 평가된다

그리고 만약 int temp = num++;와 같이 처리한다면 num++라는 prValue가 int temp라는 lValue에 담겨야 하기에 prValue에서 rValue로 전환된다. 그리고 난 후 복사 초기화를 통해 temp를 초기화하게 되는 것이다

그러나 만약 `int&& rf = num++;`와 같이 처리한다면 '&&'라는 rValue 참조 연산자에 대응하기 위해 prValue는 rValue로서 평가되게 된다

2번 예제

```
// decltype ( entity )
auto val_0 = num;

// decltype ( expression )
auto val_1 = true ? a : a;    // expression is xValue
auto val_2 = ++num;           // expression is lValue
auto val_3 = (num);           // expression is lValue
auto val_4 = num++;           // expression is prValue
int&& rf = num++;           // This expression is identical to the
                             // expression above, but no transformation
                             // between the value categories occurred
```

이 예제 코드는 문서의 위쪽에 있던 어느 코드를 가져온 것이다

'`a ? b : c`' 형식을 통해 반환되는 표현식의 연산 결과값은 대표적인 xValue이기에 '`true ? a : a`' 표현식도 xValue로 평가된다

증감연산자 중 전위 증감연산자는 임시 객체를 생성하지 않는, 즉, prValue를 생성하지 않는 연산이기에 `++num`은 lValue로 평가된다

'`(num)`'의 경우에는 이러한 문맥에서는 괄호를 빼고 그냥 `num`을 사용하는 것과 차이가 없이 그냥 lValue로서 평가된다.

물론, 괄호를 붙인 쪽은 괄호를 풀어 해석해야만 하기 때문에 표현식으로서 인식된 후 평가 결과 lValue가 되는 것이고, 그냥 `num`만 쓴 것은 곧바로 식별자로서 인식되기에 곧바로 lValue로 평가된다

증감연산자 중 후위 증감연산자는 본 객체에 1을 더한 임시 객체를 생성하는, 즉, prValue를 생성하는 연산이기에 `num++`은 prValue로 평가된다

`int&& rf = num++;`를 설명하자면 다음과 같은 순서를 거쳐 처리된다

- a) `num++`가 prValue로서 평가된다
- b) rValue Reference에 대응하기 위해 prValue는 rValue로 변환을 거친다

1.5. 표현식의 범주

1.5.1. 표현식의 정의

표현식이란 연산자(Operator)와 피연산자(Operand)의 나열이며, 어떤 연산을 구체화한다
표현식은 결과를 생성하며, 이는 결과 객체(Result Object)의 형태로 나타난다

C++에서 표현식이란 Type 범주와 Value 범주로 특징지어진다

각 요소들과 하위 표현식의 평가 순서는 중간 결과의 획득에 따라 특정된다.
쉽게 생각하면 연산자들의 우선 순위에 따라 하나의 표현식은 하위 표현식의 트리 형태로
묘사되며, 해당 하위 표현식 트리의 최하위 표현식부터 Recursive하게 하위 표현식들을
연산하여 중간 결과를 획득하고, 보다 상위의 표현식에 포함된 하위 표현식을 얻어낸 중간
결과값으로 치환하는 형태로 이루어진다

1.5.2. 표현식의 범주와 해석

- Primary Expression

Primary Expression의 피연산자로는 Primary Expression 혹은 다른 Expression가 될 수 있다

Primary Expression의 요소들은 다음과 같다

- this
- Literals
- id-expressions(identifier-expression)
 - unqualified identifiers
 - qualified identifiers
 - declarators
- lambda-expression
- fold-expression
- requires-expression

- Full-Expression

Full-Expression은 다른 표현식의 Subexpression이 아닌 표현식으로 정의된다. 다만,
평가되지 않은 피연산자와 같은 일부 컨텍스트에서 Subexpression은 Full-Expression으로
평가된다

Full-Expression의 요소들은 다음과 같다

- unevaluated operand
- constant expression
- immediate invocation

- a declarator of a simple declaration or a member initializer, including the constituent expressions of the initializer
- an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object whose lifetime has not been extended, or
- an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression

- **Potentially-Evaluated Expressions**

Potentially-Evaluated Expressions은 잠재적으로 계산될 표현식을 말한다.

즉, 실질적인 연산이 발생해야할 표현식을 Potentially-Evaluated Expressions이라고 말하며, sizeof나 typeid와 같은 연산이 필요하지 않은 Operand이거나 혹은 이러한 Operand를 포함하는 Subexpression이 존재한다면 Potentially-Evaluated Expressions이 아니다

- **Discarded-Value Expressions**

Discarded-Value Expressions은 식의 평가 중 부작용이 발생한 경우에만 사용되는 표현식이며, 이러한 식에서 계산된 값은 삭제된다

Discarded-Value Expressions의 대표적인 예로는 void Type으로의 Casting이 있다

1.6. C/C++ 표준에서 제공하는 Type 관련 Library

1.6.1. [stdint.h] or [cstdint]

```
// [cstdint]

// cstdint standard header (core)

// Copyright (c) Microsoft Corporation.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception

#pragma once
#ifndef _CSTDINT_
#define _CSTDINT_
#include <yvals_core.h>
#if __STL_COMPILER_PREPROCESSOR

#include <stdint.h>

#pragma pack(push, __CRT_PACKING)
#pragma warning(push, __STL_WARNING_LEVEL)
#pragma warning(disable : __STL_DISABLED_WARNINGS)
__STL_DISABLE_CLANG_WARNINGS
#pragma push_macro("new")
#undef new

__STD_BEGIN
using __CSTD int8_t;
using __CSTD int16_t;
using __CSTD int32_t;
using __CSTD int64_t;
using __CSTD uint8_t;
using __CSTD uint16_t;
using __CSTD uint32_t;
using __CSTD uint64_t;

using __CSTD int_least8_t;
using __CSTD int_least16_t;
using __CSTD int_least32_t;
using __CSTD int_least64_t;
using __CSTD uint_least8_t;
using __CSTD uint_least16_t;
using __CSTD uint_least32_t;
using __CSTD uint_least64_t;
```

```
using _CSTD int_fast8_t;
using _CSTD int_fast16_t;
using _CSTD int_fast32_t;
using _CSTD int_fast64_t;
using _CSTD uint_fast8_t;
using _CSTD uint_fast16_t;
using _CSTD uint_fast32_t;
using _CSTD uint_fast64_t;

using _CSTD intmax_t;
using _CSTD intptr_t;
using _CSTD uintmax_t;
using _CSTD uintptr_t;

#if _HAS_TR1_NAMESPACE
namespace _DEPRECATE_TR1_NAMESPACE tr1 {
    using _CSTD int8_t;
    using _CSTD int16_t;
    using _CSTD int32_t;
    using _CSTD int64_t;
    using _CSTD uint8_t;
    using _CSTD uint16_t;
    using _CSTD uint32_t;
    using _CSTD uint64_t;

    using _CSTD int_least8_t;
    using _CSTD int_least16_t;
    using _CSTD int_least32_t;
    using _CSTD int_least64_t;
    using _CSTD uint_least8_t;
    using _CSTD uint_least16_t;
    using _CSTD uint_least32_t;
    using _CSTD uint_least64_t;

    using _CSTD int_fast8_t;
    using _CSTD int_fast16_t;
    using _CSTD int_fast32_t;
    using _CSTD int_fast64_t;
    using _CSTD uint_fast8_t;
    using _CSTD uint_fast16_t;
    using _CSTD uint_fast32_t;
    using _CSTD uint_fast64_t;

    using _CSTD intmax_t;
    using _CSTD intptr_t;
    using _CSTD uintmax_t;
}
```

```

        using _CSTD uintptr_t;
    } // namespace tr1
#endif // _HAS_TR1_NAMESPACE
_STD_END

#pragma pop_macro("new")
_STL_RESTORE_CLANG_WARNINGS
#pragma warning(pop)
#pragma pack(pop)

#endif // _STL_COMPILER_PREPROCESSOR
#endif // _CSTDINT

```

```

// stdint.h
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// The C Standard Library <stdint.h> header.
//
#pragma once
#define _STDINT

#include <vcruntime.h>

#if _VCRT_COMPILER_PREPROCESSOR

#pragma warning(push)
#pragma warning(disable: _VCRUNTIME_DISABLED_WARNINGS)

typedef signed char          int8_t;
typedef short                 int16_t;
typedef int                  int32_t;
typedef long long             int64_t;
typedef unsigned char         uint8_t;
typedef unsigned short        uint16_t;
typedef unsigned int          uint32_t;
typedef unsigned long long    uint64_t;

typedef signed char          int_least8_t;
typedef short                int_least16_t;
typedef int                 int_least32_t;
typedef long long            int_least64_t;
typedef unsigned char         uint_least8_t;
typedef unsigned short        uint_least16_t;

```

```

typedef unsigned int      uint_least32_t;
typedef unsigned long long uint_least64_t;

typedef signed char      int_fast8_t;
typedef int               int_fast16_t;
typedef int               int_fast32_t;
typedef long long         int_fast64_t;
typedef unsigned char     uint_fast8_t;
typedef unsigned int      uint_fast16_t;
typedef unsigned int      uint_fast32_t;
typedef unsigned long long uint_fast64_t;

typedef long long         intmax_t;
typedef unsigned long long uintmax_t;

// These macros must exactly match those in the Windows SDK's intsafe.h.
#define INT8_MIN          (-127i8 - 1)
#define INT16_MIN          (-32767i16 - 1)
#define INT32_MIN          (-2147483647i32 - 1)
#define INT64_MIN          (-9223372036854775807i64 - 1)
#define INT8_MAX           127i8
#define INT16_MAX          32767i16
#define INT32_MAX          2147483647i32
#define INT64_MAX          9223372036854775807i64
#define UINT8_MAX          0xffui8
#define UINT16_MAX          0xffffui16
#define UINT32_MAX          0xffffffffui32
#define UINT64_MAX          0xffffffffffffffffui64

#define INT_LEAST8_MIN      INT8_MIN
#define INT_LEAST16_MIN     INT16_MIN
#define INT_LEAST32_MIN     INT32_MIN
#define INT_LEAST64_MIN     INT64_MIN
#define INT_FAST8_MIN        INT8_MIN
#define INT_FAST16_MIN       INT32_MIN
#define INT_FAST32_MIN       INT32_MIN
#define INT_FAST64_MIN       INT64_MIN

```

```

#define INT_FAST8_MAX    INT8_MAX
#define INT_FAST16_MAX   INT32_MAX
#define INT_FAST32_MAX   INT32_MAX
#define INT_FAST64_MAX   INT64_MAX
#define UINT_FAST8_MAX   UINT8_MAX
#define UINT_FAST16_MAX  UINT32_MAX
#define UINT_FAST32_MAX  UINT32_MAX
#define UINT_FAST64_MAX  UINT64_MAX

#ifndef _WIN64
    #define INTPTR_MIN    INT64_MIN
    #define INTPTR_MAX    INT64_MAX
    #define UINTPTR_MAX   UINT64_MAX
#else
    #define INTPTR_MIN    INT32_MIN
    #define INTPTR_MAX    INT32_MAX
    #define UINTPTR_MAX   UINT32_MAX
#endif

#define INTMAX_MIN        INT64_MIN
#define INTMAX_MAX        INT64_MAX
#define UINTMAX_MAX       UINT64_MAX

#define PTRDIFF_MIN       INTPTR_MIN
#define PTRDIFF_MAX       INTPTR_MAX

#ifndef SIZE_MAX
    // SIZE_MAX definition must match exactly with limits.h for modules
    // support.
    #ifdef _WIN64
        #define SIZE_MAX 0xffffffffffffffffui64
    #else
        #define SIZE_MAX 0xffffffffui32
    #endif
#endif

#define SIG_ATOMIC_MIN    INT32_MIN
#define SIG_ATOMIC_MAX    INT32_MAX

#define WCHAR_MIN          0x0000
#define WCHAR_MAX          0xffff

#define WINT_MIN           0x0000
#define WINT_MAX           0xffff

#define INT8_C(x)      (x)

```

```
#define INT16_C(x)    (x)
#define INT32_C(x)    (x)
#define INT64_C(x)    (x ## LL)

#define UINT8_C(x)    (x)
#define UINT16_C(x)   (x)
#define UINT32_C(x)   (x ## U)
#define UINT64_C(x)   (x ## ULL)

#define INTMAX_C(x)   INT64_C(x)
#define UINTMAX_C(x)  UINT64_C(x)

#pragma warning(pop) // _VCRUNTIME_DISABLED_WARNINGS

#endif // _VCRT_COMPILER_PREPROCESSOR
```

1.7. 알아둘만한 이슈들

1.7.1. 초기화 종류와 관련된 이슈

일반적으로 변수의 초기화에 사용되는 방식은 크게 세 가지이다
이 방식들에 내부적인 동작들이 다 다르기에 용도에 맞춰서 사용하는 것이 좋다

- **Copy Initialization**

```
int num = 0;
```

```
T object = other;
```

표준에서 정의하는 변수의 Copy Initialization 구문은 위와 같다

위의 예제는 다음과 같이 분석할 수 있다

- a) 0은 Literal이기에 prValue로 평가된다
- b) = 연산자의 각각 좌우에 lValue와 rValue를 필요로 하기에 prValue를 rValue로 변환한다
- c) =의 동작에 따라 0의 복사본을 만들어 num에 대입한다

- **Direct Initialization**

```
int num( 0 );
```

```
T object ( arg );
```

표준에서 정의하는 변수의 Direct Initialization 구문은 위와 같다

위의 예제는 다음과 같이 분석할 수 있다

- a) 0을 num에 암시적 변환을 통해 대입한다

- **List-initialization**

```
int num{ 0 };
```

```
T object { arg1, arg2, ... };
```

표준에서 정의하는 변수의 List-initialization 구문은 위와 같다

앞선 예제는 다음과 같이 분석할 수 있다

- a) 0을 num에 암시적 변환을 통해 대입한다. 다만, List-initialization에서 위와 같은 상황에서는 Narrowing conversions을 적용하여 암시적 변환을 시행한다

1.7.2. 값의 표현 한계에 관련된 이슈

당연하지만 어떤 변수가 표현할 수 있는 값의 범위는 해당 변수의 Type 크기에 따라 결정된다

그러나 Python과 같은 언어의 경우에는 만약 어떤 변수에 정수를 넣는다고 할 때, 그 이론상 값 표현의 한계가 존재하지 않는다

이는 Arbitrary Precision이라는 개념으로서, 쉽게 말하자면 필요하다면 사용 가능한 모든 메모리를 끌어모아서라도 값을 표현하고자 하는 것이다. 따라서 원리상으로는 값 표현의 한계가 없지만, 실제로는 물리적인 한계로 인해 한계가 존재하긴 한다

C/C++에도 이러한 타입의 구현을 시도한 여러 프로젝트들이 존재한다. 그러나 Arbitrary Precision과 같은 고상한 표현 보다는 주로 BigInteger과 같은 이름으로 부른다

1.7.3. Overflow, Underflow 이슈

변수란 앞서 말했듯 프로세스에 예약된 메모리 청크의 일부분을 다시금 변수로 사용하겠다고 선언한 것³이며, 메모리란 그저 이진수를 담을 뿐이기에 특정 메모리 영역을 어느 위치부터, 어느 정도의 크기로, 어떻게 해석하느냐에 따라 변수가 의미하는 것이 결정된다

그렇기에 크기를 벗어나는 Bit Operation으로 일어나는 Overflow나 Underflow는 치명적인 문제가 될 수 있다

쉽게 말하자면 Overflow란 bit값을 더하다 보니 올바른 최상위 bit 값을 소실하는 것이며, Underflow는 bit값을 빼다 보니 올바른 최하위 bit값을 소실하는 것이다
최상위 bit란 10000000란 bit의 나열이 있을 때 1이 있는 자리는 말하며, 최하위 bit란 00000001이란 bit의 나열이 있을 때 1이 있는 자리를 말한다

```
#include <cstdint>

int main()
{
    int16_t num{ INT16_MAX - 0b11 };
    for (int count{ 0 }; count < 100; ++count)
    {
        num += 0b1;
    }

    return 0;
}
```

³ 즉, 할당받은 것

위의 예제 코드는 Overflow를 발생시키는 코드이다

주의할 점은 Overflow, Underflow는 해당 단어가 사용된 문맥에 따라 같은 단어일지라도 전혀 다른 현상을 묘사할 때가 있다는 점이다. 따라서 위에서 설명한 현상은 객체의 값 표현에서 발생하는 현상을 묘사할 때의 Overflow와 Underflow라는 점을 명확히 구분해야만 한다

1.7.4. 중간 형태 컴파일 결과물의 Decompiler에 의한 코드 노출 보안 이슈

최종 소비자들이 가장 많이 접하는 프로그램의 형태인 바이너리 파일의 경우에는 우리가 작성한 식별자 등에 대한 정보가 일절 남아있지 않다 때문에 바이너리 코드를 뜯어본다고 하더라도 해당 바이너리 코드가 실행되는 타겟 플랫폼에 대한 정보를 바탕으로 어셈블리 코드로 변환하고, 이 어셈블리 코드를 통해 전체 프로그램의 구조를 이해하는 등의 많은 노력이 필요하다

그러나 dll과 같은 중간 형태의 프로그램 조각의 경우에는 다양한 상황에서의 활용을 위해 식별자 정보를 비롯한 다양한 Symbol들이 남아있을 수 있다

그렇기에 dll을 Decompiling해주는 도구를 사용하게 된다면 실제 우리가 코드를 작성할 때 썼던 코드와 거의 흡사한 고수준의 언어로 된 코드를 보고 분석할 수 있게 된다

이렇게 보안상으로 취약한 점을 보완하기 위한 기법으로서 난독화라는 것이 있다 말 그대로 코드를 보다 읽기 어렵게 하는 것인데, 가장 흔한 방식으로는 식별자들을 아무런 의미가 없는 a0, a1과 같은 것으로 변경해 코드의 역할이나 문맥을 읽기 어렵게 하는 것이다

그러나 우리가 코드를 작성할 때부터 식별자의 이름을 아무런 의미 없는 것으로 작성하면 코드 작성이 굉장히 어려워질 것이기에 시중에는 이미 잘 지어진 의미가 있는 식별자를 포함하는 코드를 넣으면 모든 식별자를 전혀 의미 없는 식별자로 바꾼 코드를 반환하는 도구들이 존재한다

1.7.5. 변수의 결합 시점과 그에 따른 Trade-Off

변수는 결국 값을 담는 컨테이너의 역할이다. 즉, 언젠가 변수는 값과 결합하게 된다는 것인데 그 타이밍은 다음과 같을 수 있다. 다만, 이 부분을 읽을 때 주의할 점은 여기서 말하는 결합 시점이라는 것은 내부적인 동작에서 실제로 변수에 값이 어떤 방식으로든 대입 되는 순간을 말하는 것이 아니라, 우리가 직관적으로 눈으로 코드를 봤을 때 이 변수에 어떤 값이 언제 들어가게 되는지를 말하는 것이다

- 코드 작성 시간(매직 넘버 사용)
- 컴파일 시간(이름 상수 사용)
- 로드 시간(.bss에 프로그램 이미지로서 저장된 static 혹은 전역 변수들이 시스템에 의해 로드)
- 객체 생성 시간(객체가 선언 및 정의될 때)
- 적시에(객체의 운용 도중에 객체가 묘사하는 값이 변할 경우)

일반적으로 결합 시점이 이를수록 유연성이 낮아지고 복잡성이 줄어든다

처음 두 가지 옵션의 경우 이름 상수를 사용하는 것이 매직 넘버를 사용하는 것보다 여러 가지 면에서 바람직하므로 좋은 프로그래밍 습관을 사용함으로써 이름 상수가 제공하는 유연성을 얻을 수 있다

그리고 요구되는 유연성이 높을수록 그러한 유연성을 지원하는 데 필요한 코드의 복잡성은 높아지고 오류를 유발할 가능성이 더 높아질 것이다

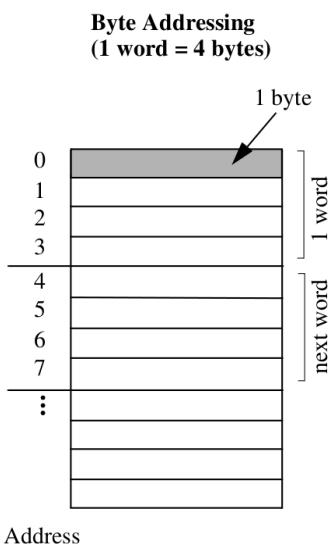
복잡하지 않을수록 더 나은 프로그래밍이기 때문에 노련한 개발자라면 소프트웨어의 요구사항을 충족시키는 데 필요한 만큼의 유연성을 추구하면서도 복잡성과 깊은 관련이 있는 유연성을 필요 이상으로 추구하지는 않을 것이다

CODE COMPLETE 2 p 269

2. 메모리와 포인터 그리고 레퍼런스

2.1. 메모리와 주소의 개념

2.1.1. 프로그래머 관점에서의 메모리 형태



프로그래머 관점에서 주로 고려해야만 할 메모리의 구조는 위의 이미지처럼 굉장히 단순하게 생겼다고 할 수 있다(정확히 말하자면 프로그래머 관점에서 직접적으로 조작하는 메모리의 주소는 Virtual Memory의 주소들이다)

보다 자세한 내용은 본 문서의 범위를 넘어서기에 담지 않았다

2.1.2. 주소의 개념

프로그래머 관점에서의 메모리, 더 정확히 Virtual Memory는 위의 그림처럼 연속된 Byte들의 나열로 볼 수 있으며, 여기서 주소란 많은 양의 Byte가 일렬로 나열해 있을 때 각 Byte마다 하나씩의 주소를 붙여 하나하나의 Byte들의 위치를 식별하고 다른 Byte들과 구분짓는 것이다 여기서 눈여겨봐야할 점은 메모리의 주소는 각 Byte마다 하나씩 배정된다는 것이다⁴

⁴ 물론, Physical Memory도 같은 형식의 주소 구조를 가지지만 프로그래머가 작성한 프로그램이 프로세스로서 실행될 때 대개 OS가 제공하는 Virtual Memory 기능 위에서 동작하기에 위와 같이 표현했다

2.2. 포인터와 주소

2.2.1. 포인터의 개념

포인터란 이름 그대로 무언가를 가리키는 용도를 가지는 변수를 말한다. 이때 가리키는 대상은 주소이다. 즉, 포인터란 주소값을 담는 변수의 특수한 형태인 것이다

2.2.2. 참조의 개념

프로그래밍에서 참조란 어떤 대상의 원하는 정보를 열람하거나 변경하는 행위라고 할 수 있다. 즉, 결국은 말미에는 뭘 하든간에 내가 원하는 어떤 대상에 접근하는 그 과정 자체를 참조라고 한다는 것이다

이때 참조는 크게 두 가지 종류로 나눌 수 있는데, 직접 참조와 간접 참조가 바로 그것들이다. 이 둘의 차이는 예제 코드를 통해 설명하겠다

```
int main()
{
    int n = 1;          // (1)
    int* pn = &n;      // (2)

    n += 2;            // (3)
    *pn += 2;          // (4)
}
```

먼저 (1)과 (2)를 살펴보자. (1)은 일반적인 정수형 변수를 선언 및 초기화 한 것이다. (2)는 참조의 개념을 설명하기 위해 이르게 등장하긴 했지만 n이라는 변수의 주소값을 저장하는 변수라고 이해하면 된다

주의깊게 바라봐야 할 곳은 (3)과 (4)의 부분들이다

(3)번의 경우 매우 익숙한 형태의 식으로서, 변수 그 자체에 값을 더하고 있다

(4)번의 경우에는 pn이라는 n의 주소값을 가지고 있는 변수를 이용해서, 즉, pn이라는 것을 한번 경유해서 간접적으로 n이라고 하는 변수에 값을 더하고 있다

직접 참조란 위의 예시에서 볼 수 있듯 직접 참조란 중간의 경유를 거치지 않고 내가 원하는 변수에 곧바로 찾아가는 것을 말한다

그에 비해 간접 참조란 내가 원하는 변수에 접근하기 위해 중간에 내가 원하는 변수로 향하는 단서를 쥐고 있는 변수들을 통해 따라따라 가면서 찾아가는 것을 말한다. 쉬운 예시로는 차례차례 단서가 등장하는 보물찾기와 같다고 생각하면 된다. 첫 번째 퍼즐을 풀면 두 번째 퍼즐이 숨겨져 있는 방으로 안내하는 종이가 들어있고, 이런 방식으로 계속 반복하다가 끝내 최종 보물이 숨겨져있는 곳을 알게 되는 것이다

이러한 간접 참조의 개념을 처음 들으면 간혹 이상하게 느껴질 수 있다. 작업의 처리 방식이 너무 비효율적인 것 같다는 생각이 들기 때문일 것이다

맞긴 맞다. 괜히 Redirection Cost라는 단어가 존재하는 것이 아니다. 간접 참조가 직접 참조에 비해 느린 것은 옳은 말이다. 그러나 간접 참조는 직접 참조는 따라올 수 없을 정도의 유연성을 프로그램에 부여할 수 있는 능력이 있다

예를 들어 다음과 같은 문제를 해결하는 프로그램을 만들어달라는 명세서가 왔다고 해보자

어떤 프랜차이즈에서는 이번에 가맹점들의 재고 요청이나 수수료 계산 등의 다양한 가맹점 맞춤 업무를 처리하기 위한 관리용 프로그램을 만들어 운용할 계획을 가지고 있다고 한다

그런데 요청 사항으로 모든 가맹점을 마치 엑셀처럼 같은 화면에 동시에 난잡한 숫자들을 파다하게 보여주지는 말고, 각 가맹점들별로 독립적인 관리 콘솔과 같이 구성해달라고 한다

이럴 경우 문제를 단순화하기 위해 여기선 프랜차이즈 본사의 가맹점 관리팀에서 볼 화면에 어떻게 가맹점들 각각의 관리 콘솔을 보여줄까의 접근법만 생각해보자

만약 직접 참조와 같은 방식으로 일을 처리한다면 가맹점 관리팀의 컴퓨터에는 각 가맹점들 개개의 관리를 위한 별도의 관리 프로그램들이 가맹점의 개수만큼 필요할 것이다

그러나 만약 간접 참조와 같은 방식으로 일을 처리한다면 가맹점 관리팀의 컴퓨터에 있는 관리 프로그램은 하나만 있으면 되고, 각 가맹점의 데이터들을 하나하나 바꿔끼워가면서, 즉, 바꿔 참조하면서 화면에 보여주면 될 것이다

이렇듯 간접 참조는 프로그램의 성능은 극히 조금 떨어질 수 있기는 하지만, 요즘과 같이 높은 컴퓨팅 파워를 가지는 CPU가 널리 보급된 환경에서 그 정도의 성능 저하는 체감도 되지 않을 정도이고, 그보다는 개발 도중의 개발 기간 감소 그리고 출시 후의 추가 업데이트에 소요되는 시간 감소 등 간접 참조가 가지는 유연성으로 인한 이익이 보다 크기에 적절한 간접 참조의 활용은 매우 유용하다고 할 수 있다

2.2.3. Built-in Address-of and Indirection Operator

- Address-of Operator

Operator Name	Syntax	Overloadable	Inside class Definition	Outside class Definition
Address-of	&a	Yes	R* T::operator&();	R* operator&(T a);

Address-of Operator의 Syntax를 보다 자세히 기술한다면 다음과 같다

& expr

이 Built-in Address-of의 expr 부분이 만약 변수라면 해당 Operator는 expr과 동등한 cv-qualification 자격을 갖춘 T* Type의 prValue를 생성하여 리턴한다
다음의 예제를 보면 이해하기 편할 것이다

```

#include <iostream>
#include <cstdint>
#include <typeinfo>

int main()
{
    int_fast32_t num{ 0 };
    const int_fast32_t num_c{ 0 };
    volatile int_fast32_t num_v{ 0 };
    const volatile int_fast32_t num_cv{ 0 };

    std::cout << typeid(&num).name() << std::endl;
    std::cout << typeid(&num_c).name() << std::endl;
    std::cout << typeid(&num_v).name() << std::endl;
    std::cout << typeid(&num_cv).name() << std::endl;

    return 0;
}

```

예제의 결과는 각 x64와 x86의 빌드 타겟에서 다음과 같이 나오게 된다

```

// x64
int * __ptr64
int const * __ptr64
int volatile * __ptr64
int const volatile * __ptr64

```

```

// x86
int *
int const *
int volatile *
int const volatile *

```

이 예제 코드를 보면 알 수 있듯 Built-in Address-of Operator는 expr에 들어가게 되는 변수와 동등한 자격의 cv-qualification를 가진 T*를 리턴해준다
또한, 리턴하는 T*가 prValue라는 점도 주목할만하다

- Indirection Operator

Operator Name	Syntax	Overloadable	Inside class Definition	Outside class Definition
Indirection	*a	Yes	R& T::operator*();	R& operator*(T a);

Address-of Operator의 Syntax를 보다 자세히 기술한다면 다음과 같다

* expr

expr에 들어갈 피연산자는 반드시 객체에 대한 포인터여야 하며, 결과적으로는 expr이 가리키는 객체의 Reference IValue를 리턴해준다

```
int main()
{
    int n = 1;
    int* ptr = &n; // pointer

    /*n;           // Incorrect Indirection
    *ptr;         // Correct Indirection
    *&n;         // Correct Indirection
}
```

즉, 위의 예제에서 볼 수 있듯 '*' Built-in Indirection Operator의 expr에 들어갈 식은 결과적으로 어떤 객체를 가리키는 포인터로 평가될 수 있어야만 하는 것이다. 이때, Built-in Indirection Operator가 리턴하는 것이 IValue인것으로 인해 Built-in Assignment Operator와 함께 다음의 예제와 같이 사용할 때 그냥 단순히 생각했을 때 문제가 될 수 있겠다고 생각할 수 있다

```
int main()
{
    int n = 1;
    int* ptr = &n;

    int m = *ptr; // indirection
}
```

Built-in Assignment Operator는 Operator의 좌측에는 IValue 그리고 우측에는 rValue를 필요로하는데, Built-in Indirection Operator는 IValue를 리턴하기 때문이다. 이 문제는 Uniform Assignment를 사용해도 좌측과 우측에 필요한 피연산자들의 값의 범주는 같기기에 같은 문제를 발생시킨다

정리하자면 `int m = *ptr`의 식에서 '='라는 Built-in Assignment Operator의 피연산자로 좌측에 `m`이라는 lValue가 있고, 우측에도 `*ptr`라는 lValue가 있게 되는 것이다. 따라서 기본적인 각 Operator들의 동작으로만 보자면 `int m = *ptr;`이라는 식은 말이 안되는 것이다

그러나 실제로는 이 식은 문제없이 동작하는데, 그 이유는 lValue-to-rValue Conversion이란게 있기 때문이다

이를 이해하기 위해선 먼저 값의 범주에서 각 범주의 동일시 관계를 파악하는 것이 도움이 될 것이다

lValue	-
prValue	-
xValue	-
glValue	lValue or xValue
rValue	prValue or xValue

즉, glValue와 rValue와 동일시되는 각 타입들은 평소에는 상호간의 치환이 불가능하지만 이번 Built-in Assignment Operator의 경우와 같이 특수한 경우에는 lValue-to-rValue Conversion을 통해 상호치환이 가능하다

다만, 이때 lValue-to-rValue Conversion를 통해 일어나는 변환의 시작 형태와 종료 형태는 C++ 표준에 의해 정의되어 있다

lValue to rValue	until C++ 11
glValue to prValue	since C++ 11

따라서 이 내용을 위의 예제에 적용시켜보자면 컴파일러가 적용중인 C++ 표준의 버전에 따라 다음과 같이 해석할 수 있다

- 만약 C++ 11 이전 버전이 적용된 컴파일러를 사용중이라면 `*ptr`은 lValue에서 rValue로 lValue-to-rValue Conversion이 발생하게 되고 문제는 해결된다
- 만약 C++ 11을 포함한 이후의 버전이 적용된 컴파일러를 사용중이라면 `*ptr`의 lValue는 먼저 glValue로 평가된 후, glValue가 prValue로 lValue-to-rValue Conversion를 통해 변환될 것이다
그런 후 prValue가 Built-in Assignment Operator의 문맥에서 rValue로 평가되면서 문제는 해결된다

2.2.4. 다중 포인터

포인터란 객체를 가리키는, 즉, 객체의 주소를 담는 변수이다

그렇다면 이런 포인터 객체를 가리키는 변수는 없을까라는 의문이 들 수 있는데 그 의문의 해답이 바로 다중 포인터이다

- Multidimensional Pointer Declarator

다중 포인터는 포인터를 가리키는 포인터이다. 이를 보다 쉽게 설명하기 위해 다음의 예제를 살펴보자

```
int main()
{
    int n{ 0 };
    int* ptrN{ &n };           // (1)
    int* ptrNBuffer{ &ptrN }; // (2)
    int** ptrPtrN{ &ptrN };   // (3)
}
```

(1)은 Fundamental Type인 변수를 가리키는 매우 일반적인 포인터이다

(2)는 얼핏보면 ptrN이라는 포인터를 가리키는 또 다른 포인터처럼 보일 수도 있다
그러나 잘 보면 ptrNBuffer의 초기화 표현식을 보면 결국 참조하는 것은 *ptrN의 주소임을 알 수 있다.

잘 기억해보면 Built-in Indirection Operator가 리턴하는 것은 결국 참조하는 대상의 IValue였기에 사실상 *ptrN은 n으로 치환해서 생각해도 상관없다. 따라서 ptrNBuffer가 가리키는 것은 결국에는 ptrN과 똑같이 n일 뿐이다

(3)을 보면 변수의 Declarator부터가 일반적인 포인터의 Declarator와는 다르게 생겼다
일반적인 포인터의 Declarator는 *가 하나만 붙는다. 그러나 다중 포인터, 그 중에서도 포인터를 가리키는 포인터, 이중 포인터의 경우에는 *가 두 개가 붙게 된다
잘 보면 ptrPtrN의 초기화 표현식에 들어간 것도 &ptrN으로서 포인터의 주소를 담고 있는 것을 알 수 있다

여기서 알아야 할 것은 n차원 포인터 객체를 가리키기 위해서는 n+1차원 포인터 객체가 필요하다는 것이다
즉, 0차원⁵를 가리키기 위해서는 1차원 포인터 변수가 필요하고, 1차원 포인터 변수를 가리키기 위해서는 2차원 포인터 변수가 필요하다는 것이다

- Multidimensional Built-in Indirection Operator

다중 포인터도 포인터인만큼 당연히 Built-in Indirection Operator가 존재한다
다음의 예제를 살펴보자

```
#include <iostream>
#include <typeinfo>

int main()
{
    int n{ 0 };
    int* ptrN{ &n };
    int** ptrPtrN{ &ptrN };
```

⁵ 즉, 일반적인 비 포인터 객체

```

    std::cout << typeid(ptrPtrN).name() << std::endl;
    std::cout << typeid(*ptrPtrN).name() << std::endl;
    std::cout << typeid(**ptrPtrN).name() << std::endl;
}

```

예제 코드의 실행 결과는 다음과 같다

```

// x64
int * __ptr64 * __ptr64
int * __ptr64
int

```

예제를 보면 다중 포인터의 경우, 이 경우에는 이중 포인터를 한 번 역참조하면 그냥 포인터가 나오고, 두 번 역참조하면 그냥 변수가 나오게 되는 것이다

2.2.5. 포인터에 붙는 다양한 cv-Qualification 들에 의한 의미의 변형

포인터는 식별자를 가지는 `IValue`이고, `IValue`는 `glValue`의 범주에 포함되기에 당연히 `cv-qualification`일 수 있다. 즉, `Specifier`로서 `const`, `volatile` 혹은 `const volatile`을 사용할 수 있다는 것이다

포인터는 근본적으로 가리키고자 하는 객체의 `Type`을 그대로 드러내는 포인터 `Type`으로 선언되어야만 한다
예컨대 다음과 같은 것이다

```

T val;
T* ptr = &val;

```

이러한 포인터의 특성에 Built-in Address-of Operator는 피연산자의 `Type T`에 대하여 `T*`를 내놓기에 완전히 딱 들어맞는다고 할 수 있다

특히나 포인터의 Type Matching은 Specifier 수준에서까지 걸맞아야만 한다. 그런데 Built-in Address-of Operator의 동작의 정의를 보면 피연산자의 `cv-qualification` 그대로 `T*`를 리턴한다고 정의하고 있기에 더욱 잘 맞는 것이다

그렇다면 결론은 포인터가 담는 주소값도 `cv-qualification`일 수 있다는 것이다. 그런데 포인터 변수도 결국은 변수이기에 포인터 변수가 담는 주소값의 `cv-qualification`이 아닌 포인터 변수 그 자체에도 `cv-qualification`가 붙을 수 있지 않을까?

당연히 포인터 변수도 `IValue`이기에 `cv-qualification`일 수 있다
예를 들어 다음의 코드를 살펴보자

```

int main()
{
    const int num{ 0 };

```

```
const int* ptr_0{ &num };
const int* const ptr_1{ &num };
}
```

ptr_0은 얼핏보면 const인 포인터 변수처럼 보일 수 있지만 아니다. const int인 객체를 가리키기 위해 const int*인 것 뿐이며, 따라서 ptr_0에는 이후에 얼마든지 다른 변수의 주소값을 대입할 수 있다

그에 비해 ptr_1은 const한 포인터 변수라고 할 수 있다. 즉, 이후에 다른 변수의 주소값을 재대입할 수 없다는 뜻이다

그렇다면 ptr_1를 가리키는 포인터를 const하게 선언하려면 어떻게 해야만 할까

```
const int* const* const ptr_last{ &ptr_1 };
```

여기선 const int* const*까지가 가리키기 위한 객체를 묘사하기 위한 부분, 즉, T*에 해당하는 부분이고 마지막의 const가 실질적으로 ptr_last가 상수 변수임을 알리는 부분이라고 할 수 있다

2.2.6. void*의 특수성

void*는 다른 포인터들에 비해 조금 특수한 경우에 속한다
다음의 예제를 보자

```
int n = 1;
int* p1 = &n;
void* pv = p1;
int* p2 = static_cast<int*>(pv);
std::cout << *p2 << '\n';           // prints 1
```

즉, void*는 어떤 Type이라도 Implicit Conversions을 통해 가리킬 수 있다. 물론, cv-qualification은 지켜줘야 한다
그러나 반대로 void*를 어떤 Type T의 T*로 변환해줘야 할 때에는 static_cast 혹은 Explicit Cast가 필요하다

만약 void* 그대로 포인터를 운용하다가 역참조하게 된다면 char*와 같이 취급하게 된다

2.3. 레퍼런스

2.3.1. 레퍼런스의 개념

레퍼런스란 포인터과 비슷한 개념으로서 포인터와 같이 다른 객체를 간접참조할 때 활용하는 용도로 사용된다는 점에서 같다

2.3.2. Built-in Reference Declarator

Built-in Reference Declarator 및 선언에 대해 C++ 표준에서 정의하는 바는 다음과 같다

& attr(optional) declarator (1)

&& attr(optional) declarator (2) (since C++ 11)

여기서 (1)은 lValue Reference이고, (2)는 rValue Reference이다
예제를 보면 곧바로 이해가 될 것이다

```
int main()
{
    int num{ 0 };
    int& rf_l{ num };
    int&& rf_r{ num + 1 };
}
```

lValue Reference는 lValue만을 참조할 수 있다. rf_l의 경우에는 num이라는 lValue를 참조하고 있기에 식에 문제가 없다

rValue Reference는 rValue만을 참조할 수 있다. rf_r의 경우에는 'num + 1'이라는 표현식의 평가 결과값인 prValue를 rValue로서 참조하고 있다

rValue Reference에서 xValue를 참조하는 경우의 식은 조금 억지를 부리자면 다음과 같아 만들 수 있다

```
int main()
{
    int num{ 0 };
    int&& rf{ (true ? num : 0) }; // (1)
    //int&& rf{ (true ? num : num) }; // (2)
```

```

        return 0;
}

```

이 경우 (1)은 옳은 식이고 (2)는 잘못된 식이다

우선, 'a ? b : c'는 b나 c에 평가가 필요한 경우에 xValue로서 결과를 리턴하는 대표적인 표현식이다

(1)의 경우에는 'true ? num : 0'이 핵심인데, 우선 평가 결과는 언제나 num으로 귀결되겠지만, 'b'와 'c' 자리에 들어간 표현식들의 값의 범주가 다르기에, 정확히 말하자면 'b'나 'c' 중에 rValue Reference에 의해 참조될 수 있는 표현식이 들어가 있기는 하기에 평가가 필요한 올바른 표현식이라고 컴파일러를 속이게 된다.

따라서 본래대로라면 num은 lValue로서 평가되어 rValue Reference에는 담을 수 없지만, 컴파일러를 속임으로서 강제로 xValue로 평가받아 rValue Reference에 담을 수 있게 된 것이다

이 경우에는 내부적으로 num을 rf에 강제로 담기 위해 lValue-to-rValue Conversion이 일어났을 것으로 추측된다

(2)의 경우에는 컴파일러를 속일 것도 없이 "b"와 "c" 어느 곳도 모두 lValue이기에 평가할 필요도 없이 int&& rf{ (true ? num : num) };라는 전체 식은 틀린 식이 되는 것이다

2.3.3. 레퍼런스와 포인터의 공통점과 차이점

다음은 포인터와 레퍼런스를 사용하여 동일한 동작을 하는 C++ 코드와 해당 코드의 어셈블리이다

```

int main()
{
    int num{ 0 };

    int* ptr{ &num };
    int& rf{ num };

    ++*ptr;
    ++rf;

    return 0;
}

```

```

int main()
{
00007F753E11F50  push      rbp
00007F753E11F52  push      rdi
00007F753E11F53  sub       rsp,148h
00007F753E11F5A  lea       rbp,[rsp+20h]
00007F753E11F5F  lea       rdi,[rsp+20h]

```

```

00007F753E11F64  mov      ecx, 1Ah
00007F753E11F69  mov      eax, 0CCCCCCCCCh
00007F753E11F6E  rep stos  dword ptr [rdi]
00007F753E11F70  mov      rax,qword ptr [_security_cookie (07FF753E1E008h)]
00007F753E11F77  xor      rax,rbp
00007F753E11F7A  mov      qword ptr [rbp+118h],rax
00007F753E11F81  lea      rcx,[__139BE4AA_main@cpp (07FF753E24067h)]
00007F753E11F88  call    __CheckForDebuggerJustMyCode (07FF753E11406h)
00007F753E11F8D  mov      dword ptr [num],0

        int* ptr{ &num };
00007F753E11F94  lea      rax,[num]
00007F753E11F98  mov      qword ptr [ptr],rax
        int& rf{ num };
00007F753E11F9C  lea      rax,[num]
00007F753E11FA0  mov      qword ptr [rf],rax

        ++ptr;
00007F753E11FA4  mov      rax,qword ptr [ptr]
00007F753E11FA8  mov      eax,dword ptr [rax]
00007F753E11FAA  inc      eax
00007F753E11FAC  mov      rcx,qword ptr [ptr]
00007F753E11FB0  mov      dword ptr [rcx],eax
        ++rf;
00007F753E11FB2  mov      rax,qword ptr [rf]
00007F753E11FB6  mov      eax,dword ptr [rax]
00007F753E11FB8  inc      eax
00007F753E11FB4  mov      rcx,qword ptr [rf]
00007F753E11FBE  mov      dword ptr [rcx],eax

        return 0;
00007F753E11FC0  xor      eax, eax
}

```

잘 보면 어셈블리 수준까지 가면 포인터와 레퍼런스의 근본적인 동작은 완전히 같은 것을 알 수 있다

두 개념 사이의 차이가 있다면 C++ 수준에서의 사소한 정의 차이가 있을 뿐이다. 그 차이는 다음과 같다

- 포인터는 가리키는 객체가 없을 수 있으며, 선언과 동시에 초기화될 필요는 없다
- 레퍼런스는 반드시 객체를 가리키고 있어야만 하며, 선언과 동시에 초기화되어야만 한다

이 차이는 개발을 하다보면 극히 사소한 차이인듯 하면서도 어쩔 때는 굉장히 큰 차이를 유발하기도 한다

사실 main 함수 안에서만 모든 코드를 작성한다면 두 개념 사이의 실질적인 차이를 실감하기 매우 어려울지도 모른다. 1차원 포인터까지만 사용하는 코드라는 가정 하에 둘 사이의 사용감의 차이는 사실상 미미하기 때문이다

그러나 추후에 설명할 함수와 예외 처리 그리고 해당 C++ 스터디에서는 다루지는 않지만 팀 프로그램에서의 개발에 빠질 수 없는 요소인 개발 기간 관리 및 세부적인 시스템 아키텍처 설계 등에서 두 개념에 정해져있는 각각의 한계와 제한사항들은 매우 유용한 도구로서 활용될 수 있다

3. 할당

3.1. 할당의 개념

할당은 Process가 부여받은 Memory page 중 일부에 특정한 목적을 위해 값을 저장하고자 할 때 해당 값이 적힐 영역을 다른 목적으로는 침해하여 덮어쓰는 등의 처리를 추가로 가하지 못하도록 점거하는 것이라고 할 수 있다

즉, 요약해서 말하자면 메모리의 이 부분은 내꺼니까 다른 애들이 사용하지 못하도록 표시해 주는 것이라고 생각하면 된다

3.2. 정적 할당과 동적 할당 각각의 할당 메커니즘

C++ 스터디 1, 2회차에서 다루었던 변수와 같은 것들이 메모리에 할당되는 형태를 정적 할당이라고 한다

정적 할당과는 다른 형태의 할당으로는 동적 할당이 있다

정적 할당과 동적 할당의 차이를 쉽게 구분하자면 '정적 할당은 알아서 해제를 해주고, 동적 할당은 (특수한 기법을 사용하지 않으면) 알아서 해제해주지 않는다'라는 것이다

3.2.1. 정적 할당 메커니즘

정적 할당 변수는 C 스타일의 동적 할당⁶이나 C++ 스타일의 동적 할당⁷을 활용하지 않은 변수들은 어지간하면 정적 할당된 변수라고 생각하면 된다

정적 할당 메커니즘은 함수 단위로 이루어지게 된다

때문에 [5. 함수]에서 다룰 내용이긴 하지만 대강 함수가 어떻게 생겼는지만 보이자면 다음과 같다

```
void func()
{
    // ...
}
int main()
{
    func();    // (1)

    return 0;
}
```

위의 코드에서 보면 알 수 있듯 main() 자체도 하나의 함수이며, 개발자가 원한다면 main 함수와 비슷한 형태로 다른 함수들도 얼마든지 만들 수 있다

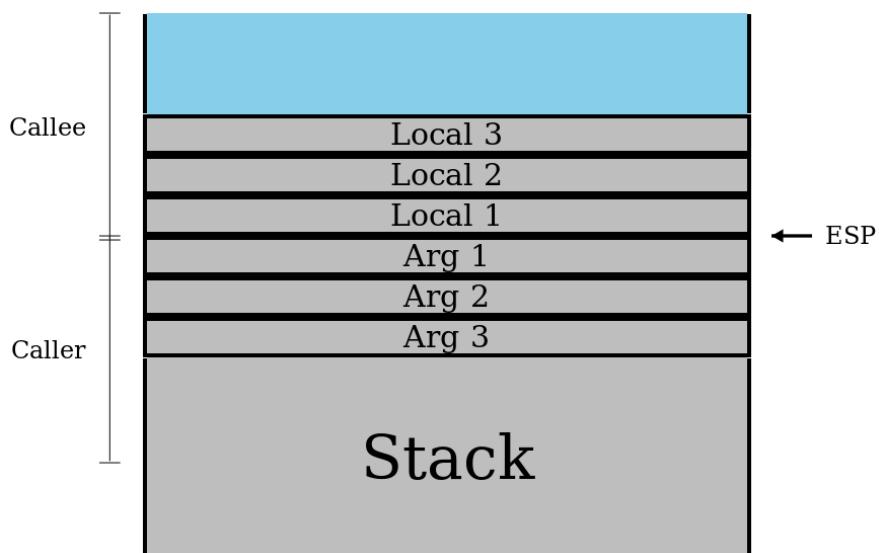
여기서 중요한 점은 함수의 호출이 일어날 때인데, (1) 부분이 바로 그곳이다

⁶ malloc(...), calloc(...) 등의 [stdlib.h] 혹은 [cstdlib]에서 제공하는 할당용 함수

⁷ new Keyword, Smart Pointer

전 회차의 스터디에서 스택 메모리의 동작 방식을 알려준 것을 기억할 것이다. 그 스택 메모리 부분에는 함수가 들어가게 된다

즉, 스택 메모리의 rBP와 가장 가까운 곳에는 main 함수를 구성하는데 필요한 기본적인 정보⁸와 main 함수에서 선언하여 사용한 변수들이 차례차례 쌓여간다
그러다가 func() 함수가 호출되게 되면 기존의 스택 메모리⁹에 func()를 구성하는 기본 데이터가 쌓이고 func()에서 선언하여 사용한 변수들이 쌓여가게 되는 것이다
그리고 만약 func()가 끝나게 되면 현재 스택 메모리에서 func의 시작 부분까지 스택 메모리에 쌓여있던 것들¹⁰을 해제하고, func() 호출 시에 쌓아뒀던 리턴 포인터를 이용해서 자기 자신을 호출했던 곳, 즉, 이 상황에선 main()의 (1) 부분을 가리키는 포인터를 통해 찾아가서 다시 main()의 나머지 처리를 이어가게 되는 것이다



따라서 이를 통해 보자면 정적 할당이라는 개념을 통해 할당받는 변수들은 변수 그 자체만 오롯이 메모리에 존재할 수는 없다. 정적 함수는 함수가 스택 메모리에 쌓이는 과정 중에 존재하게 되며, 함수의 호출과 리턴 개념과 아주 밀접하게 연관되어 있다는 것을 알 수 있다

3.2.2. 동적 할당 메커니즘

3.2.2.1. Memory Allocator

정적 할당의 경우에는 기본적으로 일정 영역이 Process 내에 Stack 영역으로 묶여있어서 해당 영역 내에 Push와 Pop을 반복하며 데이터를 관리하게 된다

그러나 Heap의 경우에는 Process를 막 시작했을 때에는 Stack과는 달리 Heap만을 위해 묶여있는 일정 크기의 영역이 존재하지 않는다. 대신, 실제로 동적 할당이 발생했을 때, Heap을 위한 영역을 OS로부터 부여받게 된다

⁸ 대표적으로 함수 포인터, 리턴 포인터

⁹ main()과 관련 데이터가 이미 쌓여있는 상태

¹⁰ func()에서 선언된 정적 변수들

```

pwndbg> info proc map
process 16508
Mapped address spaces:

Start Addr      End Addr      Size      Offset objfile
0x400000        0x401000      0x1000    0x0 /home/elasjay/4th/lazenga/in-use
0x600000        0x601000      0x1000    0x0 /home/elasjay/4th/lazenga/in-use
0x601000        0x602000      0x1000    0x1000 /home/elasjay/4th/lazenga/in-use
0x7ffff7a0d000  0x7ffff7bcd000 0x1c0000 0x0 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7bcd000  0x7ffff7cd000 0x200000 0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dc0d000 0x7ffff7dd1000 0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dd1000  0x7ffff7dd3000 0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dd3000  0x7ffff7dd7000 0x4000   0x0
0x7ffff7dd7000  0x7ffff7fd000 0x26000  0x0 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7fd0000  0x7ffff7fd000 0x3000   0x0
0x7ffff7fa0000  0x7ffff7fda000 0x3000   0x0 [vvar]
0x7ffff7fffa000 0x7ffff7ffc000 0x2000   0x0 [vds]
0x7ffff7ffc000  0x7ffff7ffd000 0x1000   0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7ffd000  0x7ffff7fe000 0x1000   0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7fe000  0x7ffff7fff000 0x1000   0x0
0x7ffff7ffe000  0x7ffff7fff000 0x21000  0x0 [stack]
0xffffffffffff600000 0xffffffffffff601000 0x1000   0x0 [vsyscall]

```

```

pwndbg> info proc map
process 16508
Mapped address spaces:

Start Addr      End Addr      Size      Offset objfile
0x400000        0x401000      0x1000    0x0 /home/elasjay/4th/lazenga/in-use
0x600000        0x601000      0x1000    0x0 /home/elasjay/4th/lazenga/in-use
0x601000        0x602000      0x1000    0x1000 /home/elasjay/4th/lazenga/in-use
0x602000        0x623000      0x21000  0x0 [heap]
0x7ffff7a0d000  0x7ffff7bcd000 0x1c0000 0x0 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7bcd000  0x7ffff7cd000 0x200000 0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dc0d000 0x7ffff7dd1000 0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dd1000  0x7ffff7dd3000 0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
0x7ffff7dd3000  0x7ffff7dd7000 0x4000   0x0
0x7ffff7dd7000  0x7ffff7fd000 0x26000  0x0 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7fd0000  0x7ffff7fd000 0x3000   0x0
0x7ffff7fa0000  0x7ffff7fda000 0x3000   0x0 [vvar]
0x7ffff7fffa000 0x7ffff7ffc000 0x2000   0x0 [vds]
0x7ffff7ffc000  0x7ffff7ffd000 0x1000   0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7ffd000  0x7ffff7fe000 0x1000   0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
0x7ffff7fe000  0x7ffff7fff000 0x1000   0x0
0x7ffff7ffe000  0x7ffff7fff000 0x21000  0x0 [stack]
0xffffffffffff600000 0xffffffffffff601000 0x1000   0x0 [vsyscall]

pwndbg> i r rax
rax          0x602010 6299664
pwndbg> x/32gx 0x602010 - 0x10
0x602000: 0x0000000000000000 0x0000000000000091
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000000
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000000
0x602090: 0x0000000000000000 0x00000000000029f71
0x6020a0: 0x0000000000000000 0x0000000000000000
0x6020b0: 0x0000000000000000 0x0000000000000000
0x6020c0: 0x0000000000000000 0x0000000000000000
0x6020d0: 0x0000000000000000 0x0000000000000000
0x6020e0: 0x0000000000000000 0x0000000000000000
0x6020f0: 0x0000000000000000 0x0000000000000000

```

이렇게 Heap을 OS로부터 부여받을 때 범용적으로 사용되는 Memory Allocator에는 dlmalloc, ptmalloc2, jemalloc, tcmalloc, libumem 등 다양한 종류가 존재한다

앞으로 설명할 것들은 ptmalloc2라는 GNU C Library에서 주로 사용되는 Memory Allocator 기준으로 진행된다

3.2.2.2. Chunk

Heap은 앞서 Process 내에서 미리 잡혀있는 공간이 없고, 매 동적 할당 때마다 OS로부터 Memory Allocator를 통해 부여받는다고 했다. 해당 Chunk들은 자신에 대한 다양한 단서를 가지고 있게 되는데, 이는 다음과 같은 malloc_chunk 구조체를 통해 관리된다

```

struct malloc_chunk;
typedef struct malloc_chunk* mchunkptr;

struct malloc_chunk {
    INTERNAL_SIZE_T prev_size; /*Size of previous chunk (if free) */
}

```

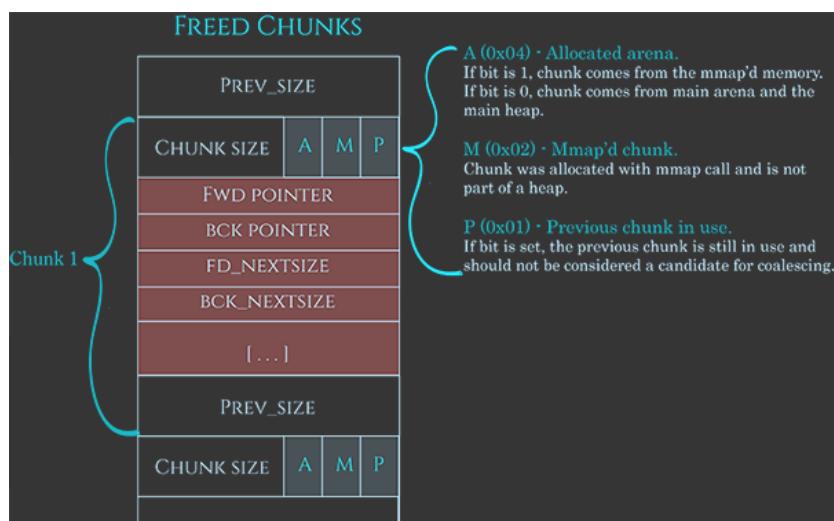
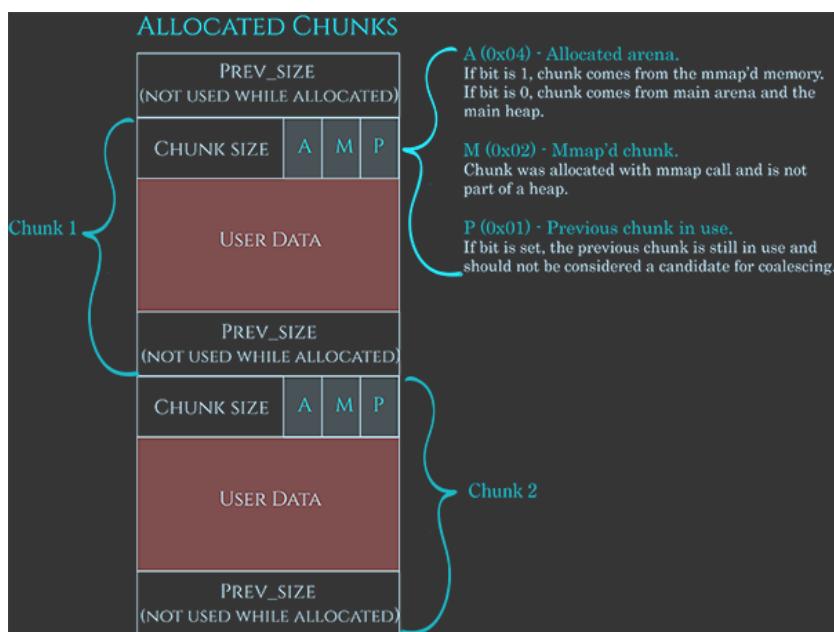
```

INTERNAL_SIZE_T size;           /*Size in bytes, including overhead*/
struct malloc_chunk* fd;       /*double links -- used only if free*/
struct malloc_chunk* bk;

/* Only used for large blocks: pointer to next larger size*/
/* double links -- used only if free*/
struct malloc_chunk* fd_nextsize;
struct malloc_chunk* bk_nextsize;
};

```

이를 그림으로 표현하면 다음과 같다



이러한 구조체를 활용하여 각 Chunk들은 또 내부적으로 In-use Chunk, Free Chunk, Top Chunk, Last Reminder Chunk로 분류된다

이 사실을 알면 구조체 각 멤버에 대해 다음과 같이 설명할 수 있게 된다

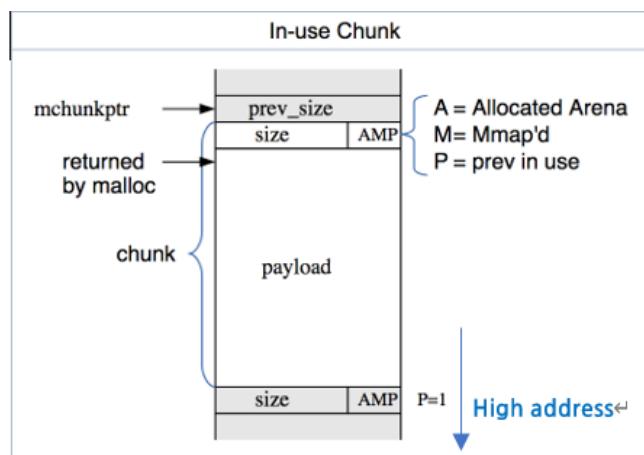
prev_size	이전 heap chunk가 free chunk가 되면 해제된 heap chunk의 크기를 저장 해제 되기 전에는 이전 힙 청크의 데이터 영역으로 사용
size	In-use chunk의 크기
flags (3bit)	필드의 맨 끝 3bit <ul style="list-style-type: none"> - PREV_INUSE (P): 이전 heap chunk가 해제된 경우 설정 <ul style="list-style-type: none"> - 1 = 이전 청크 해제 X - 0 = 이전 청크 해제 O - IS_MAPPED (M): 현재 청크가 mmap 시스템 콜을 사용해 할당된 경우 설정 - NON_MAIN_ARENA (A): 현재 Chunk가 main_arena에서 관리하지 않을 경우 설정
FD (Forward pointer)	다음 free chunk의 포인터
BK (Backward pointer)	이전 free chunk의 포인터
fd_nextsize	large bin에서 사용하는 포인터, 현재 heap chunk의 크기보다 작은 heap chunk의 주소를 가리킴
bk_nextsize	large bin에서 사용하는 포인터, 현재 heap chunk의 크기보다 큰 heap chunk의 주소를 가리킴

* fd, bk, fd_nextsize, bk_nextsize 는 현재 chunk가 free chunk일 경우에만 사용

* chunk의 크기 = MALLOC_ALIGNMENT(2*sizeof(size_t))의 배수;
 32bit의 경우 size_t = 4byte >> chunk의 크기는 8의 배수
 64bit의 경우 size_t = 8byte >> chunk의 크기는 16의 배수

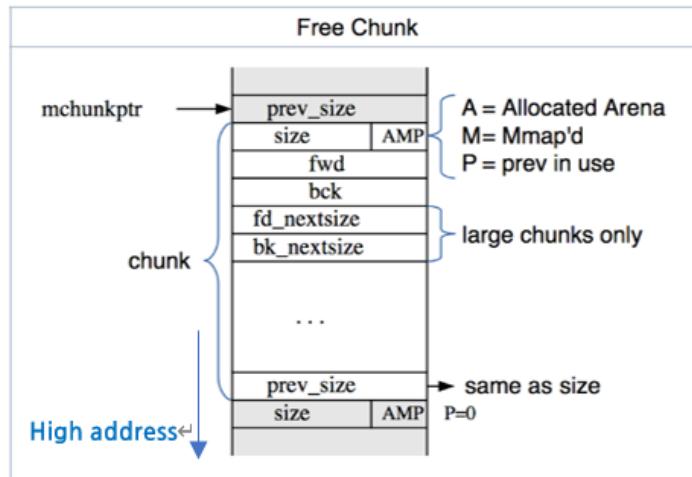
- In-use chunk

할당자로부터 메모리를 할당받아 사용중인 메모리 청크



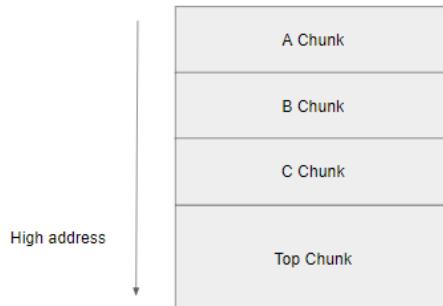
- Free chunk

할당자에게 반환된 chunk



- Top Chunk

heap memory의 마지막에 위치한 chunk



- Last Remainder chunk

작은 사이즈의 할당 요청이 들어왔을 때, Free chunk가 쪼개지고 남은 chunk* 연속된 작은 사이즈의 할당 요청이 들어왔을 때 비슷한 주소에 heap chunk가 할당되는 할당의 지역성을 유지하기 위해 사용된다

3.2.2.3. Bin

Free Chunk, 즉, 한 번 해제를 겪은 Chunk들은 그 크기나 특징에 따라 각각의 bin이라는 freelist 구조체¹¹를 통해 관리된다

따라서 최초의 Heap 할당은 애초에 Bin이 비어있기에 OS로부터 받게 되지만, 그 다음부터는 상황에 따라 다르게 처리되게 된다

¹¹ 내부적으로는 Linked-List와 비슷한 구조를 가진다고 생각하면 된다

새로운 Heap 할당 요청의 크기를 충당하는 Free Chunk가 있을 경우	freelist에서 Heap 요청의 크기에 걸맞는 Chunk를 찾아내 해당 영역을 사용하고, freelist에서는 제거한다
새로운 Heap 할당 요청의 크기를 충당하는 Free Chunk가 없을 경우	OS로부터 새로운 Heap Chunk를 부여받는다

반대로 해제의 경우에는 In-use Chunk에서 Free Chunk로 전환됨에 따라 먼저 Unsorted Bin이라는 곳에 추가되었다가 다양한 조건들에 따라 Fast Bin, Small Bin과 Large Bin이라는 곳으로 분류되게 된다

이를 정리하면 다음과 같다

Fast Bin	<p>작은 크기의 heap chunk를 할당하고 해제할 때 사용하는 bin chunk의 크기: 16~64 byte (32 bit), 32~128byte (64 bit)</p> <ul style="list-style-type: none"> • LIFO (Last-In First-Out) 방식 사용; 마지막으로 해제된 chunk가 가장 먼저 재할당 • 최대 10개의 bin 관리 • fastbin의 상한 값보다 작은 chunk들을 관리 - 64bit: 32, 48, 64, 80, 96, 112 byte의 chunk • single-linked list로 구성; ptmalloc2의 bin 중 할당 및 해제 속도 가장 빠름 • 같은 bin에 포함된 chunk들끼리 서로 인접해도 하나의 chunk로 병합X
Small Bin	<p>512byte(32bit)/1024byte(64bit) 미만의 사이즈의 chunk가 해제되었을 때 unsorted bin에 리스트가 추가된 후 저장되는 bin smallbin 크기의 heap chunk 해제 → unsortedbin → smallbin에 할당</p> <ul style="list-style-type: none"> • small bin이 포함하는 chunk: MIN_LARGE_SIZE보다 작은 chunk들; 32bit-MIN_LARGE_SIZE=512 / 64bit-MIN_LARGE_SIZE=1024 • FIFO (First-In, First-Out) 방식 사용; 먼저 해제된 chunk가 먼저 재할당 • 64개의 bin들을 관리 • doubly-linked list로 구성; 같은 크기의 chunk들끼리 하나의 bin으로 연결 • small bin chunk들은 서로 인접하게 배치 될 수 없음 (인접해 있을 경우 하나의 chunk로 병합)
Large Bin	<p>해제된 chunk의 크기가 512byte(32bit)/1024byte(64bit)이상</p> <ul style="list-style-type: none"> • large bin이 포함하는 chunk의 크기: MIN_LARGE_SIZE와 크거나 같은 chunk들; 32bit - 512이상, 64 - 1024 이상 • FIFO (First-In, First-Out) 방식 사용; 먼저 해제된 chunk가 먼저 재할당 • 63개의 bin들을 관리 • doubly-linked list로 구성; 하나의 bin에 다양한 크기의 chunk들을

	<p>크기별로 정렬해 보관 / fd_nextsize와 bk_nextsize를 사용해 서로 다른 크기의 chunk들을 리스트로 연결</p> <ul style="list-style-type: none"> large bin chunk들은 서로 인접하게 배치 될 수 없음 (인접해 있을 경우 하나의 chunk로 병합)
Unsorted Bin	<p>small bin과 large bin 크기의 heap chunk가 해제되면 이후 재할당을 위해 사용되는 bin</p> <ul style="list-style-type: none"> unsorted bin은 크기의 제한 X, 다양한 크기의 heap chunk를 저장 / fast bin에 해당하는 chunk는 배치되지 않음 FIFO (First-In, First-Out) 방식 사용; 먼저 해제된 chunk가 먼저 재할당 1개의 bin double-linked list로 구성; 해제된 chunk를 재사용하기 위해 해제된 chunk의 크기보다 작거나 동일한 크기의 chunk가 할당되어야 함 large bin chunk들은 서로 인접하게 배치 될 수 없음 (인접해 있을 경우 하나의 chunk로 병합)

3.2.2.4. Arena

ptmalloc2에서 각 스레드가 간섭하지 않고 서로 다른 메모리 영역에 액세스 할 수 있는 메모리 영역

- **main arena**

malloc() 함수에서 main arena를 가리키는 정적변수가 있고 각 arena에는 추가 arena를 연결하는 포인터들이 있다

- **각 arena는 하나 이상의 heap memory를 갖는다**

main arena: 프로그램 초기 heap 사용(.bss 영역 등의 직후)

추가 arena: mmap을 통해 heap에 메모리를 할당하고 이전 heap이 소모되면 더 많은 heap을 heap 목록에 추가

- **heap 메모리에서 할당된 chunk들을 관리한다**

arena에서 관리되는 chunk: 응용 프로그램에서 사용중이거나 사용이 가능한 chunk

사용중인 chunk는 arena에서 추적 불가능

free chunk를 크기와 히스토리에 따라 분류해 arena에 저장

할당자는 arena에서 할당 요청을 충족하는 chunk를 신속하게 찾을 수 있음

3.3. C 스타일 동적 할당

3.3.1. malloc

malloc의 선언은 다음과 같이 생겼다

```
void* malloc( size_t size );
```

예시 코드는 다음과 같다

```
#include <stdlib.h>

int main()
{
    int* num = static_cast<int*>(malloc(sizeof(int)));

    free(num);

    return 0;
}
```

보면 malloc의 파라미터로 sizeof(int)가 들어간 것을 볼 수 있는데, 이는 int 하나 들어갈 만큼의 크기를 동적 할당받기를 원하기에 그런 것이다
또한, malloc이 리턴하는 것이 void*이기에 이를 T*인 포인터에 대입하려고 할 때에는 반드시 static_cast<T> 혹은 명시적 캐스팅을 해주어야만 한다

malloc의 특징은 할당받은 영역에 값을 따로 초기화해주는 처리같은 것 없이 그저 할당받은 영역을 전달해주는 역할만 한다는 것이다

3.3.2. calloc

calloc의 선언은 다음과 같이 생겼다

```
void* calloc( size_t num, size_t size );
```

예시 코드는 다음과 같다

```
#include <stdlib.h>

int main()
{
    int* num = static_cast<int*>(calloc(1, sizeof(int)));
```

```
    free(num);  
  
    return 0;  
}
```

보면 malloc과 다 비슷하지만 파라미터로 size만큼을 몇 개 할당받을 것인지를 파라미터로 받는다는 것을 알 수 있다

또한, malloc과 마찬가지로로 리턴하는 것이 void*이기에 이를 T*인 포인터에 대입하려고 할 때에는 반드시 static_cast<T> 혹은 명시적 캐스팅을 해주어야만 한다

calloc의 특징은 malloc과는 달리 할당받은 메모리 영역을 만약 Value Type이라면 0으로, Pointer Type이라면 nullptr로 초기화해준다는 차이점이 있다

3.4. C++ 스타일 동적 할당

3.4.1. new, delete Operator

C++에서 동적 할당의 가장 근본적인 방법론이 이 new Keyword라고 할 수 있다
일반적으로 널리 쓰이는 new Operator의 표준 선언 형태는 다음과 같다

```
void* operator new (std::size_t) throw(std::bad_alloc);
void* operator new (std::size_t, void*) throw();
void* operator new (std::size_t, const std::nothrow_t&) throw();
```

```
void operator delete (void*) throw();
void operator delete (void*, void*) throw();
void operator delete (void, const std::nothrow_t&) throw();
```

보면 malloc과 비슷하게 생긴 걸 알 수 있다
이를 활용하는 예제 코드는 다음과 같다

```
int main()
{
    int* num = new int;

    delete num;

    return 0;
}
```

new Operator도 malloc과 같이 값을 따로 초기화해주지는 않는다
다만 다음과 같이 사용하게 될 경우 초기화를 해준다

```
int main()
{
    int* num = new int();

    delete num;

    return 0;
}
```

즉, new Operator 뒤쪽에 '('만 붙여주면 되는 것이다. 이때, '(' 안에 초기화하고 싶은 값을
쓰면 된다. 만약 비어있다면 0으로 초기화해준다
이 방식은 'ISO C++03 5.3.4[expr.new]/15'에 명시되어 있으며, 그 내용은 요약하자면 다음과
같다

ISO C++03 5.3.4[expr.new]/15

A `new`-expression that creates an object of type `T` initializes that object as follows:

...

If the `new`-initializer is of the form `()`, the item is value-initialized (8.5);

추가로 참고로 말하자면, `new Operator`와 `delete Operator`의 선언과 실 사용 형태가 다른 것을 알았을 것이다

실제 선언대로 하나하나 다 작성한다면 코드는 다음과 같이 되어야만 할 것이다

```
int main()
{
    int* num = static_cast<int*>(::operator new(sizeof(int)));
    *num = 0;

    std::cout << *num;

    ::operator delete(num);

    return 0;
}
```

그러나 명시적으로 작성하지 않아도 그에 대응하는 적절한 `new, delete Operator`를 찾아 매개변수들을 대입해 호출하도록 컴파일러가 내부적으로 처리를 해주고 있기에 우리는 이런식으로 작성하지 않아도 되는 것이다

사실 `new operator`와 `delete Operator`도 결국은 함수일 뿐이기에 우리가 직접 만들 수도 있다 굉장히 간단한 버전으로 만들자면 다음과 같이 만들 수 있다¹²

```
void* operator new(std::size_t size) throw(std::bad_alloc)
{
    if (size == 0)
    {
        size = 1;
    }

    while (true)
    {
        void* pMem = malloc(size);
        if (pMem)
        {
            return pMem;
        }
    }
}
```

¹² 다만, 해당 코드는 `new, delete Operator`를 작성할 때 지켜야 할 관례를 지키기는 하지만 몇몇 이유에서 허용하다고는 할 수 없으니 그대로 사용하는 일은 없길 바란다

```

    }

    std::new_handler globalhandler = std::set_new_handler(NULL);
    std::set_new_handler(globalhandler);

    if (globalhandler)
    {
        (*globalhandler)();
    }
    else
    {
        throw std::bad_alloc();
    }
}

```

```

void operator delete(void* pMem) throw()
{
    if (pMem)
    {
        return;
    }

    free(pMem);
}

```

new Operator를 작성할 때 지켜야 할 관례는 다음과 같다

1. 반환 값이 제대로 되어있어야만 한다
2. 가용 메모리가 부족할 경우에는 new 처리자 함수(std::new_handler)를 호출해야한다
3. 크기가 없는(0Byte) 메모리 요청에 대한 대비책을 갖춰야한다
4. 다음의 조건 중 하나가 충족되기까지 계속해서 메모리 할당을 시도한다
 - 메모리 할당이 성공한다
 - new 처리가 함수가 다음 중 하나 혹은 다수의 처리를 한다
 - 다음 루프에서는 메모리 할당에 성공할 수 있을 가능성에 걸고 사용할 수 있는 메모리를 더 많이 확보한다
 - 다른 new 처리자 함수를 설치한다
 - new 처리자 함수의 설치를 제거한다(할당에 실패하고 std::bad_alloc을 던진다)
 - 예외를 던진다(std::bad_alloc 혹은 std::bad_alloc을 상속한 Type의 예외를 던진다)
 - 복귀하지 않는다(주로 exit() 혹은 abort()를 호출한다. 즉, 프로그램의 강제 종료)
5. 기본 형태의 new가 가려지지 않도록 한다
6. 만약 상속 관계에서 자식 클래스가 부모 클래스의 new를 통해 할당을 시도하는 경우 부모 클래스와 자식 클래스의 크기를 비교하여 같지 않을 경우의 방안을

마련한다(까다로운 아키텍처의 경우 Byte Alignment가 맞지 않을 경우 하드웨어 에러를 발생시킬 수 있기도 하며, 유연한 아키텍처의 경우에도 눈에 띄게 성능이 하락할 수 있다)

앞의 코드와 관례를 함께 두고 보면 쉽게 알 수 있듯 new에서는 할당에 성공하거나 new 처리자가 무언가 해주지 않으면 무한 루프에 빠져버릴 수 있다

new 처리자 함수¹³는 new Operator를 통해 메모리 할당을 시도했으나 실패했을 경우 이에 대한 대처를 하기 위한, 즉, 에러에 대처하기 위한한 내용이 담기는 함수를 말한다

더욱 자세한 설명을 하려면 3회차의 벗어나기에 샘플 코드만 남기고 추가적인 설명은 않도록 하겠다. 만약 해당 코드의 동작이 궁금하다면 이후에 따로 물어보길 바란다
일단 다음의 함수는 특정 클래스(Test)가 기본 할당 처리 동작과는 달리 자기만의 특수한 처리를 원한다고 가정했을 때의 코드이다¹⁴

```
// newHandlerSample.h
#include <new>

template<typename T>
class NewHandlerSupport
{
public:
    static std::new_handler set_new_handler(std::new_handler p)
throw();
    static void* operator new(std::size_t size) throw(std::bad_alloc);

private:
    static std::new_handler curHandler;
};

class Test_Way_1 : public NewHandlerSupport<NewHandlerSample>
{

};

class Test_Way_0
{
public:
    static std::new_handler set_new_handler(std::new_handler p)
throw();
    static void* operator new(std::size_t size) throw(std::bad_alloc);

private:
    static std::new_handler curHandler;
```

¹³ std::new_handler

¹⁴ 다만, 해당 코드는 각 클래스에서 new를 정의하는데 비해 delete를 정의하고있지는 않기에 아주 바람직한 코드라고는 할 수 없다는 것은 알아주었으면 좋겠다

```

};

class NewHandlerHolder
{
public:
    explicit NewHandlerHolder(std::new_handler nh) : handler(nh) {}
    ~NewHandlerHolder() { std::set_new_handler(handler); }

private:
    std::new_handler handler;

    NewHandlerHolder(const NewHandlerHolder&);
    NewHandlerHolder& operator =(const NewHandlerHolder&);

};

```

```

// newHandlerSample.cpp
#include "newHandlerSample.h"

template<typename T>
std::new_handler NewHandlerSupport<T>::curHandler = 0;
template<typename T>
std::new_handler NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
    std::new_handler oldHandler = curHandler;
    curHandler = p;
    return oldHandler;
}
template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
throw(std::bad_alloc)
{
    NewHandlerHolder h(std::set_new_handler(curHandler));

    return ::operator new(size);
}

std::new_handler Test_Way_0::curHandler = 0;
std::new_handler Test_Way_0::set_new_handler(std::new_handler p) throw()
{
    std::new_handler oldHandler = curHandler;
    curHandler = p;
    return oldHandler;
}

```

```
}

void* Test_Way_0::operator new(std::size_t size) throw(std::bad_alloc)
{
    NewHandlerHolder h(std::set_new_handler(curHandler));

    return ::operator new(size);
}
```

```
// main.cpp
#include "newHandlerSample.h"

int main()
{
    Test_Way_0::set_new_handler(FailMemAlloc);
    Test_Way_0* node_0 = new Test_Way_0;

    delete node_0;

    Test_Way_1::set_new_handler(FailMemAlloc);
    Test_Way_1* node_1 = new Test_Way_1;

    delete node_1;

    return 0;
}
```

delete Operator를 작성할 때 지켜야 할 관례는 다음과 같다

1. Null 포인터에 대한 delete가 안전하도록 보장한다

다만, Visual C++의 경우에는 이러한 경우에 대해 보장하지는 않는다

3.4.2. Smart Pointer

지금까지 앞에서 소개했던 동적 할당 방식들은 모두 개발자가 직접 할당과 해제 코드를 작성해줘야만 했으나, Smart Pointer의 경우에는 해제 부분의 코드를 따로 작성해주지 않아도 동작하도록 해준다¹⁵

그 원리는 생각보다 단순하다

앞서 Stack에 할당되는 것들은 Function Call Stack과 관계가 깊다고 했던 것을 기억할 것이다. 정확히 말하자면 어떤 함수 A가 호출되어 Call Stack에 Push되고, A에서 사용된 여러

¹⁵ 해당 Smart Pointer가 가리키는 대상이 적절한 방식으로 구성되어있다는 가정하에

변수들이 있다고 해보자. 이런 상황에서 만약 A가 리턴되면 A에서 사용되던 여러 변수들은 해당 변수 Type T에 해당하는 각각의 소멸자¹⁶를 호출하는 방식으로 동작하게 된다. 물론, 세부적으로 들어가자면 함수 단위 뿐만 아니라 함수 내부의 Scope 단위로도 Stack 내에서 Function Call Stack과 동일한 흐름으로 진행되기에 염밀히 말하자면 Scope 단위로 Smart Pointer는 자원을 해제해준다고 생각하는 것이 좋을 것이다.

Smart Pointer는 이 부분을 파고 드는 것이다. Smart Pointer 자체는 Stack에 할당되도록 하고, 해당 Smart Pointer 내부에 진짜 가리키고 싶은 동적 할당된 인스턴스가 존재하는 것이다.

그러다가 만약 Smart Pointer가 선언된 함수가 리턴되게 되면 자연스레 해당 Smart Pointer의 소멸자가 호출되게 될 것인데, 해당 소멸자에서 타고타고 가다보면 명시적으로 delete를 호출해주는 것을 알 수 있다.

Smart Pointer의 종류에는 std::unique_ptr, std::shared_ptr, std::weak_ptr의 세 가지가 존재한다. std::auto_ptr도 존재했으나 C++ 11부터 비권장 동작으로 분류되었다.

- std::unique_ptr

std::unique_ptr의 특징은 std::unique_ptr로 동적 할당을 받은 메모리의 경우에는 한 번에 하나의 인스턴스만이 가리킬 수 있다는 것이다. 즉, 동시에 두 개 이상의 변수가 같은 데이터를 가리킬 수 없다.

```
#include <memory>
#include <cassert>

int main()
{
    // Create a (uniquely owned) resource
    std::unique_ptr<int> p = std::make_unique<int>();

    // Transfer ownership to `pass_through` ,
    // transfers ownership back via the return value
    // std::unique_ptr<int> q = p;           // Error
    std::unique_ptr<int> q = std::move(p);

    // p is now in a moved-from 'empty' state, equal to nullptr
    assert(!p);

    return 0;
}
```

- std::shared_ptr

std::shared_ptr의 특징은 같은 데이터에 대한 참조 포인트는 여러 개 존재할 수 있다. 대신, 참조 포인트의 개수를 센다.

¹⁶ T의 인스턴스가 사용한 값을 정리해주는 역할

즉, 참조 포인트가 늘어나면 참조 개수가 1 증가하고, 참조 포인트가 줄어들면 1 감소한다
그러다가 참조 개수가 0이 되면 할당을 해제한다

```
#include <memory>
#include <cassert>
#include <iostream>

int main()
{
    std::shared_ptr<int> num = std::make_shared<int>(0);
    std::cout << num.use_count() << std::endl;

    std::shared_ptr<int> numBuffer = num;
    std::cout << num.use_count() << "\t" << numBuffer.use_count() <<
std::endl;

    numBuffer.reset();
    std::cout << num.use_count() << "\t" << numBuffer.use_count() <<
std::endl;

    num.reset();
    std::cout << num.use_count() << "\t" << numBuffer.use_count() <<
std::endl;

    return 0;
}
```

```
// Output
1
2      2
1      0
0      0
```

- std::weak_ptr

std::weak_ptr은 그 자체만 사용할 수 있는 것은 아니고 std::shared_ptr과 함께 사용하는 방식이다

std::shared_ptr의 경우 참조 개수를 세며 운영된다고 앞서 말했다. 즉, 참조 포인트를 늘릴 수록 이를 관리해나가기 어려워진다는 것이다. 이럴 때 std::weak_ptr는 참조 포인트는 늘리지만 참조 개수로는 세지 않게 하는 것이 가능하다

```
#include <iostream>
#include <memory>
```

```

std::weak_ptr<int> gw;

void observe()
{
    std::cout << "gw.use_count() == " << gw.use_count() << ";" ;
    // we have to make a copy of shared pointer before usage:
    if (std::shared_ptr<int> spt = gw.lock())
    {
        std::cout << "*spt == " << *spt << '\n';
    }
    else
    {
        std::cout << "gw is expired\n";
    }
}

int main()
{
{
    auto sp = std::make_shared<int>(42);
    gw = sp;

    observe();
}

    observe();
}

```

```

// Output
gw.use_count() == 1; *spt == 42
gw.use_count() == 0; gw is expired

```

3.4.3. std::Allocator<T>

std::Allocator<T>는 std namespace에서 제공하는 기능들, 즉, 표준 컨테이너나 다양한 유ти리티 함수 등의 내부 메모리 동적 할당을 위해 존재하는 클래스이다
이것까지 알 필요는 없다고 생각은 하지만 굳이 넣은 이유는 std::Allocator<T>도 결국은 내부적으로 new Operator를 사용한다는 것을 보여주려고 넣은 것이다

따라서 다음의 코드를 내부적으로 따라따라 가며 보이겠다

```

#include <vector>

int main()

```

```

{
    std::vector<int> numList;
    numList.push_back(0);

    return 0;
}

```

3.5. 동적 배열

```

#include <stdlib.h>
#include <memory>

int main()
{
    size_t arrSize{ 10 };

    // C Style
    {
        int* numList = static_cast<int*>(calloc(arrSize,
sizeof(int)));
    }

    // C++ Style
    {
        int* numList = new int[arrSize]();
        /*std::shared_ptr<int> numList(new int[10], [](){}); */
        delete[] numList;
    }
}

return 0;
}

```

동적 배열은 물리적으로 연속된 메모리 Chunk들의 나열을 동적 할당받고, 메모리 Chunk 나열의 가장 앞쪽 부분, 즉, 가장 작은 주소값 부분의 주소를 적절한 포인터가 가리키게 하는 것이다

위의 코드에서는 생략했지만, 동적 배열 또한 동적 할당을 통해 구성되는 것이기에 당연하게도 적절한 해제 과정을 거쳐주는 것이 바람직하다

3.6. 동적 할당에서 주의할 점

3.6.1. Dangling Pointer

Dangling Pointer란 가리키는 대상을 잃은 포인터를 말한다
예를 들어 다음과 같은 코드가 있다고 해보자

```
#include <iostream>

int main()
{
    int* num = new int(3);

    delete num;

    //++*num;      // Error

    std::cout << "test";

    return 0;
}
```

즉, 이미 할당이 끝나서 해제된 메모리 영역에 접근하려고 하는 것이 Dangling Pointer라는 것이다

이러한 동작이 발생할 경우 옳지 않은 메모리로의 접근이 발생했기에 예외가 발생하게 되고, 따로 예외 처리를 해주지 않으면 프로그램에 뺏어버리며 비정상적으로 종료될 것이다

3.6.2. Memory Leak

Memory Leak은 사용하지도 않는는데도 해제되지 않고 관리되지 않는 동적 할당된 메모리를 말한다

Memory Leak이 발생할 수 있는 상황들은 다음과 같다

```
int* p = new int(7);    // dynamically allocated int with value 7
p = nullptr;           // memory leak
```

```
void f()
{
    int* p = new int(7);
}                      // memory leak
```

```

void f()
{
    int* p = new int(7);
    g();           // may throw
    delete p;      // okay if no exception
}               // memory leak if g() throws

```

3.6.3. Pointer Double Release

Pointer Double Release는 동적 할당받은 하나의 변수를 두 번 해제하려고 시도하는 것을 말한다

당연히 이미 한 번 해제되었던 메모리에 다시 한 번 접근하여 해제하려고 하는 것이기에 옮지 않은 메모리로의 접근이며 따라서 예외가 발생하게 되고, 따로 예외 처리를 해주지 않으면 Core Dump와 함께 프로세스가 뺏어버리며 비정상적으로 종료될 것이다

```

#include <iostream>

int main()
{
    int* num = new int(3);

    delete num;
    //delete num;    // Error

    std::cout << "test";

    return 0;
}

```

3.7. RAII 패턴

Resource Acquisition Is Initialization라고 해서 RAII라고 하며, Bjarne Stroustrup이 제안한 디자인 패턴이다

직역하자면 자원의 획득, 할당은 초기화라는 것인데 말하자면 앞서 Smart Pointer의 원리가 그렇듯 Stack의 원리를 활용하여 자동으로 메모리가 해제되도록 하는 것이 RAII의 목표이다 사실상 Smart Pointer 자체가 이 RAII 패턴으로부터 시작해서 표준에 추가되었다고 할 수 있다

4. 흐름 제어

4.1. 소프트웨어의 성능 지표

소프트웨어를 평가할 수 있는 지표는 다양할 것이다. 특히 유저의 제품 사용 경험(UX)과 비즈니스로서의 소프트웨어 설계 방법으로서 DDD(Domain Driven Design)같은 다양한 다양한 설계 방법론이 대두되고 있는 요즘에는 더더욱 그럴 것이다

그러나 소프트웨어를 단순히 공학적 산출물로서만 바라본다면 평가 지표는 **시간 복잡도**, **공간 복잡도** 단 두 가지만 존재하면 된다

두 개념은 너무 쉬워서 다들 알 것이다

시간 복잡도는 ‘어떤 문제를 해결할 때 얼마나 시간이 걸리느냐’, 공간 복잡도는 ‘어떤 문제를 해결하는데 얼마나 메모리를 사용하느냐’의 문제인 것이다

즉, 단순히 소프트웨어를 공학적 산출물로서 순수하게 성능으로만 바라볼 때에는 경제적으로 가장 적게 써서 가장 많은 일을 처리할 수 있으면 그게 가장 이상적이라는 것이다

그러나 이와는 다르지만 전통적으로 많이들 말하는 지표도 있다

바로 **가독성**이다

코드도 결국엔 글이다. 그런 의미에서는 서점에서 팔리는 책들과 같다

때문에 글을 작성할 때 활용할 언어도 있고, 언어의 문법도 있는 것이다

그리고 글을 읽는 독자도 동료 개발자와 컴파일러의 형태로 분명히 존재한다

다행히 컴파일러는 글이 아무리 복잡하고 난잡해도 문법만 맞추고 말만 된다면 불만을 표하지는 않는다. 그러나 글을 읽는 사람들은 불만을 표하지 않을까?

코드를 읽는 더 중요한 독자가 누구인가를 잘 생각해봐야 할 것이다

컴파일러인가? 개발자들인가?

대개 성능과 가독성은 Trade-Off 관계라고들 한다

그렇다면 둘 중 어느쪽에 보다 무게를 둘 것인가의 문제는 앞선 질문에 어떤 결론을 내느냐에 따라 달라질 것이다

4.2. 정보의 흐름 제어

4.2.1. 정보의 흐름을 제어한다는 것

코드를 작성하다보면 반복문과 조건문을 자주 활용하게 된다

사실 이 두 형태의 구문을 활용하지 않고서 프로그래밍을 하기란 어지간이 간단한 프로그램이 아닌 이상은 불가능할 것이다

때문에 이 두 가지를 일상적으로 사용하게 된다

때로는 각각, 혹은 때로는 두 가지를 막 뒤섞어가며 고도로 중첩해 사용할 때도 있을 것이다

그런데 생각해보자

반복문과 조건문을 사용하는 것은 좋다

막 어지럽게 중첩해가며 코드를 짜서 문제 해결은 된다? 그것도 좋다

그런데 그게 가장 효율적인 방법인가?

정보의 흐름을 제어한다는 것에서 생각해볼만한 큰 주제는 [4.1. 소프트웨어의 성능 지표]에서 말했던 두 가지의 평가 지표로 추려볼 수 있을 것 같다

첫째, 과연 정보의 흐름을 제어하는 것이 혹은 그 방식이 성능의 측면에서 효율적인가?

둘째, 과연 정보의 흐름을 제어하는 것이 가독성의 측면에서 효율적인가? 코드를 읽는 주 독자는 누구인가?

4.2.2. 반복

4.2.2.1. 반복의 종류

반복의 종류를 구분하고자 한다면 그 분류법에는 다양한 방법론이 있을 수 있을거라고 생각한다

그러나 여기서 사용하고자 하는 구분 방법은 프로시저의 콜스택이 자라나는지 여부를 바라보는 것이다

- Linear Iteration

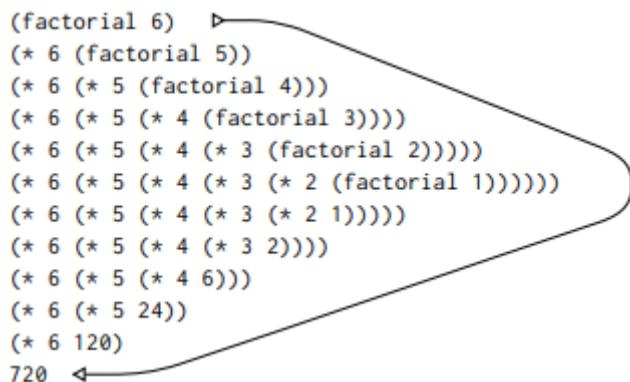
Linear Iteration의 예제는 다음과 같다고 할 수 있다

```
(factorial 6) ▶
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720 ◀
```

```
// Lisp
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

위의 코드는 팩토리얼을 계산하기 코드이며, 그 위의 이미지는 해당 프로세저를 실행했을 때 프로세저가 진행되는 과정을 펼쳐서 보여주는 것이다
이미지를 보면 프로세저를 실행하는데 프로세저의 매개변수 자리의 값만 바꿔가며 같은 형태의 프로세저가 말 그대로 반복 되는 것을 볼 수 있다

- Linear Recursion

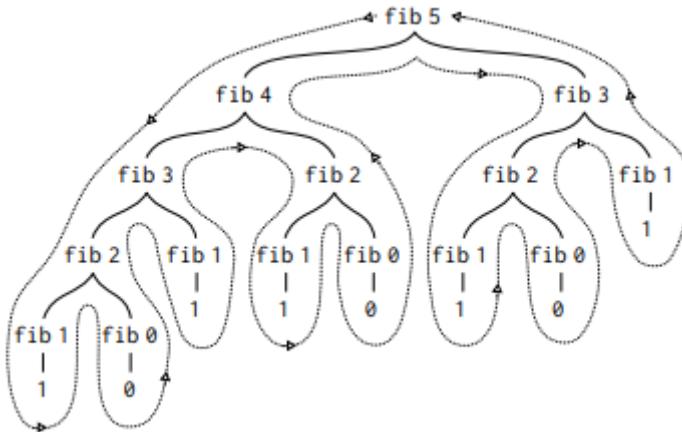


```
// Lisp
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

위의 코드 또한 팩토리얼을 계산하기 위한 코드이며, 그 위의 이미지는 해당 프로세저를 실행했을 때 진행되는 과정을 펼쳐서 보여주는 것이다
이미지를 보았을 때 Linear Iteration과는 확연한 차이를 확인할 수 있다. Linear Iteration과는 달리 직접적인 연산을 되는데까지 뒤로 미뤘다가 한계에 이르렀을 때부터 한 단계씩 줄어들며 연산을 점점 회수해온다

이렇게 했을 때의 장점이라 한다면 코드가 문제의 형태를 그대로 모델링해오면 되기에 코드가 직관적이며 읽기 쉽다는 장점이 있다
그러나 Linear Iteration보다는 많은 메모리가 필요하다는 단점이 존재한다

- Tree-Recursion



$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise.} \end{cases}$$

```
// Lisp
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

위의 코드는 피보나치 수열을 계산하기 위한 코드이며, 중앙의 이미지는 피보나치 수열의 각 항을 계산하기 위한 함수, 최상단의 이미지는 해당 프로시저를 실행했을 때 진행되는 과정을 펼쳐서 보여주는 것이다

보면 각 항을 계산하기 위해서는 그 항을 계산하기 위해 그 전 항과 전전 항을 계산해야 하기에 프로시저가 마치 Tree의 형태로 자라나는 것을 볼 수 있다

Tree-Recursion도 Linear Recursion와 같은 장단점을 공유한다

그러나 Tree-Recursion의 경우에는 Linear Recursion보다 더욱 많은 메모리를 요구한다는 점에서 성능에 미치는 영향이 보다 심각하다고 할 수 있다

4.2.2.2. 반복문의 종류 및 문법

- while

while Loop의 구문은 다음과 같다

```
attr(optional) while ( condition ) statement
```

간단한 예제는 다음과 같다

```
while (--x >= 0)
{
    int i;
} // i goes out of scope
```

while을 풀어서 생각하자면 다음과 같이 생각할 수도 있다

```
label:
{
    T t = x;           // start of condition scope
    if (t)
    {
        statement
        goto label;   // calls the destructor of t
    }
}
```

- for

for Loop의 구문은 다음과 같다

```
formal syntax:
attr (optional) for ( init-statement condition (optional);
iteration-expression (optional) ) statement

informal syntax:
attr (optional) for ( declaration-or-expression (optional);
condition (optional); expression (optional) ) statement
```

간단한 예제는 다음과 같다

```
for (int i = 0;;)
{
    long i = 1;    // valid C, invalid C++
    // ...
}
```

for를 풀어서 생각하자면 다음과 같이 생각할 수도 있다

```
{
    init-statement
    while ( condition )
```

```

{
    statement
    iteration-expression;
}
}

```

4.2.2.3. 한 번쯤 생각해보면 좋은 것 같은 질문들

- 반복으로 처리할 수 있다고 반복을 쓰는 것이 무조건 좋은 걸까?

반복을 사용할 수 있는 상황이라고 반복을 사용하는 것이 반드시 좋은 걸까?
 이를 확실하게 확인하기 위해서는 같은 처리를 하는 코드를 반복을 사용하는 버전과
 사용하지 않는 버전 두 가지를 준비한 후 그 어셈블리를 확인하는 것이 가장 직관적이고 빠를
 것이다

```

; Code 1

#include <iostream>

int main()
{
00007FF601EB1002  sub      rsp,20h
    int temp{ 0 };
00007FF601EB1006  xor      ebx,ebx
00007FF601EB1008  nop      dword ptr [rax+rax]
    while (temp < 10)
    {
        temp += 1;
        std::cout << temp;
00007FF601EB1010  mov      rcx,qword ptr [_imp_std::cout
(07FF601EB2088h)]
00007FF601EB1017  inc      ebx
00007FF601EB1019  mov      edx,ebx
00007FF601EB101B  call     qword ptr
[_imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
(07FF601EB2080h)]
00007FF601EB1021  cmp      ebx,0Ah
00007FF601EB1024  jl      main+10h (07FF601EB1010h)
    }

    return 0;
00007FF601EB1026  xor      eax,eax
}
00007FF601EB1028  add      rsp,20h
00007FF601EB102C  pop      rbx

```

```
00007FF601EB102D  ret
```

```
; Code 2

#include <iostream>

int main()
{
00007FF7595E1000  sub        rsp,28h

    int temp{ 0 };

    temp += 1;
    std::cout << temp;
00007FF7595E1004  mov        rcx,qword ptr [__imp_std::cout
(07FF7595E2088h)]
00007FF7595E100B  mov        edx,1
00007FF7595E1010  call       qword ptr
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
(07FF7595E2080h)]

    temp += 1;
    std::cout << temp;
00007FF7595E1016  mov        rcx,qword ptr [__imp_std::cout
(07FF7595E2088h)]
00007FF7595E101D  mov        edx,2
00007FF7595E1022  call       qword ptr
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
(07FF7595E2080h)]

    temp += 1;
    std::cout << temp;
00007FF7595E1028  mov        rcx,qword ptr [__imp_std::cout
(07FF7595E2088h)]
00007FF7595E102F  mov        edx,3
00007FF7595E1034  call       qword ptr
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
(07FF7595E2080h)]

    temp += 1;
    std::cout << temp;
00007FF7595E103A  mov        rcx,qword ptr [__imp_std::cout
(07FF7595E2088h)]
00007FF7595E1041  mov        edx,4
00007FF7595E1046  call       qword ptr
```

```
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<  
 (07FF7595E2080h)]  
  
    temp += 1;  
    std::cout << temp;  
00007FF7595E104C  mov        rcx,qword ptr [__imp_std::cout  
 (07FF7595E2088h)]  
00007FF7595E1053  mov        edx,5  
00007FF7595E1058  call       qword ptr  
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<  
 (07FF7595E2080h)]  
  
    temp += 1;  
    std::cout << temp;  
00007FF7595E105E  mov        rcx,qword ptr [__imp_std::cout  
 (07FF7595E2088h)]  
00007FF7595E1065  mov        edx,6  
00007FF7595E106A  call       qword ptr  
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<  
 (07FF7595E2080h)]  
  
    temp += 1;  
    std::cout << temp;  
00007FF7595E1070  mov        rcx,qword ptr [__imp_std::cout  
 (07FF7595E2088h)]  
00007FF7595E1077  mov        edx,7  
00007FF7595E107C  call       qword ptr  
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<  
 (07FF7595E2080h)]  
  
    temp += 1;  
    std::cout << temp;  
00007FF7595E1082  mov        rcx,qword ptr [__imp_std::cout  
 (07FF7595E2088h)]  
00007FF7595E1089  mov        edx,8  
00007FF7595E108E  call       qword ptr  
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<  
 (07FF7595E2080h)]  
  
    temp += 1;  
    std::cout << temp;  
00007FF7595E1094  mov        rcx,qword ptr [__imp_std::cout  
 (07FF7595E2088h)]  
00007FF7595E109B  mov        edx,9  
00007FF7595E10A0  call       qword ptr  
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
```

```

(07FF7595E2080h)]

    temp += 1;
    std::cout << temp;
00007FF7595E10A6  mov        rcx,qword ptr [__imp_std::cout
(07FF7595E2088h)]
00007FF7595E10AD  mov        edx,0Ah
00007FF7595E10B2  call       qword ptr
[__imp_std::basic_ostream<char, std::char_traits<char> >::operator<<
(07FF7595E2080h)]


    return 0;
00007FF7595E10B8  xor        eax,eax
}
00007FF7595E10BA  add        rsp,28h
00007FF7595E10BE  ret

```

위쪽의 코드가 반복문을 사용하는 코드, 아래가 반복문을 사용하지 않는 코드이다
반복을 사용하는 코드의 경우에는 반복 처리를 위해 매 반복마다 6개의 Instruction이
처리되며, 총 10회 반복되기에 반복 처리를 위해 총 60개의 Instruction이 사용되었다는 것을
알 수 있다

그에 비해 반복을 사용하지 않는 코드의 경우에는 반복 처리에 해당하는 부분의 매 부분마다
3개의 Instruction이 처리되며, 총 10회 반복되기에 반복 처리를 위해 총 30개의 Instruction이
사용되었다는 것을 알 수 있다

이 결과를 통해 매 반복마다 다음과 같은 2개의 Instruction이 필요했다는 것을 알 수 있다

```

00007FF601EB1021  cmp        ebx,0Ah
00007FF601EB1024  j1        main+10h (07FF601EB1010h)

```

즉, 반복이라는 흐름 제어를 사용한다는 것은 가독성을 의해 성능을 다소 희생한다는
의미라는 것을 알 수 있다
그렇다면 성능을 위해서는 가독성을 모조리 희생하고 다 풀어서 작성해야만 할까? 반복
횟수가 프로그램의 런타임에만 알 수 있다면 미리 풀어 쓸 수 없을텐데 이런 경우는 어떻게
해야할까?

먼저, 성능을 위해 가독성을 모조리 희생하고 다 풀어서 작성해야만 할까라는 질문의
답으로는 반드시 그럴 필요는 없다
Loop-Unroll이라는 기법이 존재하며, 이를 잘만 활용한다면 우리가 작성하는 코드는 반복문을
사용하는 코드 그대로지만 컴파일러 수준에서 컴파일할 때 반복문을 풀어서 작성해준다
즉, 반복 처리를 위해 필요한 쓸데없는 Instruction들을 생략할 수 있는 것이다
이때, 반복 횟수는 당연하게도 컴파일 타임에 결정되어야한다는 조건이 붙는다

다음으로, 반복 횟수가 프로그램의 런타임에만 알 수 있다면 미리 풀어 쓸 수 없을텐데 이런
경우는 어떻게 해야할까라는 질문의 답으로는 그 반복 횟수가 런타임에만 알 수 있다면

Loop-Unroll과 같은 기법을 사용할 수 없기에 어쩔 수 없이 반복을 위한 Instruction들이 필요하다¹⁷

- 어떤 상황에서는 어떤 반복문을 사용하는 것이 좋을까?

이 질문의 답은 사람에 따라 주관적일 수 있다

나는 개인적으로 반복의 횟수를 알 수 없을 때에는 while, 반복의 횟수를 알 수 있을 때에는 for를 많이 사용한다

왜냐하면 while의 경우에는 반복문의 시작 부분에 표시되는 정보가 많지 않고, 반복과 관련된 처리가 어쩔 수 없이 while에 종속되는 Statement의 이곳저곳에 흩어져서 속하게 된다
따라서 코드를 읽는 독자의 입장에서 while의 반복 횟수를 파악하기 위해서는 while의 Statement를 모두 읽어야만 한다

따라서 그럴 바에는 차라리 while은 무한 반복이 기본이고, 무한 반복을 끝내기 위한 조건문만을 한 곳에 모아두는 것이 차라리 읽기 편하다

그러나 for의 경우에는 반복문의 시작 부분에 표시되는 정보나 많으며, 반복의 횟수를 파악할 수 있는 모든 정보가 들어있다

때문에 for는 반복 횟수가 정해져있을 때 사용하는 것이 보다 읽기 편하다

이때, for에 종속되는 Statement에는 반복 조건과 관련된 변수는 웬만하면 조작하지 않는다는 조건이 필요하다. 반복 조건과 관련된 변수를 조작하는 순간 반복의 정확한 횟수를 파악하기 위해서 Statement 부분을 읽어야만 하기 때문이다

- 반복문을 중첩한다면 어떤 순서로 어떻게 중첩하는 것이 좋을까?

반복문을 중첩한다는 것에는 다양한 상황이 있을 수 있으니 다음과 같이 크게 두 가지로 추릴 수 있을 것 같다

- 배열, 버퍼 등의 직렬 데이터를 순회하는 경우
- 반복되는 현상을 캡쳐하여 처리하기 위해 혹은 기타 처리를 위해 반복하는 경우

먼저 배열, 버퍼 등의 직렬 데이터를 순회하는 경우이다

이 부분은 개인마다의 스타일에 따라 다를 수 있겠지만 개인적으로 이런 경우에는 배열 혹은 버퍼에 접근하는 순서와 반복문 중첩의 순서를 맞추는 것이 보다 코드를 읽기가 편하다고 생각한다

예를 들어 다음과 같은 것이다

```
#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE][ARR_SIZE] = { 0, };

    for (int coord_y{ 0 }; coord_y < ARR_SIZE; ++coord_y)
```

¹⁷ 물론, 이와 같은 결과는 사용된 컴파일러 그리고 같은 컴파일러에서도 사용된 최적화 플래그 설정의 경우의 수에 따라 조금씩 다른 결과가 나올 수 있다는 점을 감안하기 바란다

```

    {
        for (int coord_x{ 0 }; coord_x < ARR_SIZE; ++coord_x)
        {
            std::cout << arr[coord_y][coord_x];
        }
        std::cout << std::endl;
    }

    return 0;
}

```

```

#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE][ARR_SIZE] = { 0, };

    for (int coord_x{ 0 }; coord_x < ARR_SIZE; ++coord_x)
    {
        for (int coord_y{ 0 }; coord_y < ARR_SIZE; ++coord_y)
        {
            std::cout << arr[coord_y][coord_x];
        }
        std::cout << std::endl;
    }

    return 0;
}

```

두 코드의 차이점은 coord_y와 coord_x를 각각 반복 변수로서 사용하는 반복문의 중첩 순서가 바뀐 것 뿐이다

다만, 배열 혹은 버퍼에 접근하는 순서와 반복문 중첩의 순서를 맞추던, 그 반대로 하던간에 확실한 것은 순서를 마구잡이로 섞는 것은 확실하게 가독성을 망치고 성능의 측정도 어렵게 한다는 것이다

또한 배열 혹은 버퍼를 순회하는 용도로 반복문을 사용한다면 중요하게 신경써야 할 점은 반복문에서 사용하는 변수의 이름이다

배열 혹은 버퍼를 순회한다는 것은 자주 발생하는 매우 일반화된 상황이다. 그런데 매번 전혀 다른 이름의 변수를 순회에 사용한다면 코드의 독자는 해당 반복문이 단순한 순회를 위한 반복문이 아니라 어떤 복잡한 처리를 하는 중요한 반복문일 가능성을 생각하느라 심력을 낭비할 수 있다

때문에 자주 발생하고 일반화된 이러한 순회를 위한 반복문에는 마찬가지로 극히 일반화된 변수 이름 규칙을 적용하여 코드의 독자의 쉬운 이해를 위해야 한다

개인적으로는 그 용도에 따라 다음과 같은 이름 규칙들을 사용한다

- 1차원 배열의 인덱스로 활용할 경우 => index_[목적]
- 다차원 배열의 인덱스로 활용할 경우 => coord_[축]
- Iterator를 활용할 경우 => itr_[목적]

다음으로 반복되는 현상을 캡쳐하여 처리하기 위해 혹은 기타 처리를 위해 반복하는 경우이다

이러한 경우 중첩의 순서를 정하는 데 개인적으로 중요하게 생각하는 점은 다음과 같다

- 개념적 혹은 분류상 보다 넓은 범위에 속하는 것을 묘사하는 반복이 보다 바깥쪽에 중첩되게 배치
- 중복되는 코드 조각이 최대한 적게 발생하는 순서로 배치

여기서 보다 넓은 범위에 속하는 것을 묘사하는 반복이 보다 바깥쪽에 중첩되게 배치한다는 것은 가독성을 위한 조치이다

코드의 흐름을 따라갈 때 개념적으로 위계를 가지는 객체를 묘사할 때 난잡하게 별 생각없이 순서를 마구잡이로 중첩하여 접근하는 것은 코드의 흐름을 따라가기 어렵다. 그러나 가장 추상적인 넓은 개념부터 차근차근 전문화된 객체를 소거법처럼 좁혀가며 접근하는 것이 사람이 생각하는 방식과 비슷하다고 생각하기에 보다 읽기 쉬울 것 같기 때문이다

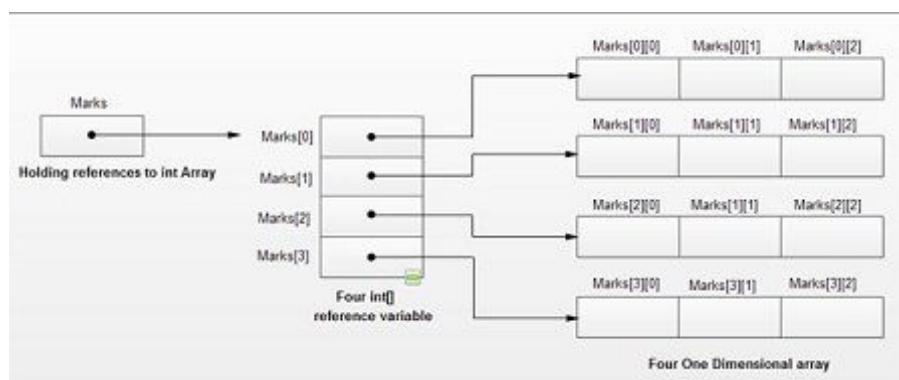
중복되는 코드 조각이 최대한 적게 발생하는 순서로 배치한다는 것은 가독성과 유지보수 둘 다를 생각했을 때의 조치이다

우선 코드의 독자가 읽고 생각해야 할 부분을 최대한 줄여서 부담을 줄여준다는 의미가 있으며, 또한 코드에 수정 사항이 생겼을 때 비슷한 부분이라 둘 다 수정을 해줘야하는데 까먹고 한 부분만 수정하는 등의 문제를 예방하기 위해서이다

또한, 만약 최대한 중첩되는 코드 조각을 줄이려 했지만 어쩔 수 없이 반복되는 코드 조각이 존재한다면 그 부분은 그대로 둘 것이 아니라 해당 코드 조각은 따로 함수로 분리를 해야하는 부분이 아닌지 진지하게 고려를 해야만 할 것이다

- **다차원 배열을 다중 반복문으로 순회한다면 어떤 순서로 순회하는 것이 좋을까?**

이 질문의 답을 구하기 위해서는 우선 다차원 배열이 어떤 방식으로 조직되는지를 먼저 알아야 할 것이다



이를 보면 알 수 있듯 코드에서 배열을 사용하는 감각으로는 만약 $N * M$ 의 다차원 배열이라면 딱 $N * M$ 만큼의 크기만 메모리상에 존재할 것 같지만 실제로는 포인터가 1차원 배열을 둘고,

1차원 배열의 각 Node가 또 각각의 1차원 배열을 둘고 이런 방식으로 2차원 배열을 조직한다는 것이다¹⁸

이 사실을 알고 다음의 코드를 보자

```
// [예시 1]
#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE][ARR_SIZE] = {{0},};

    for (int coord_y{ 0 }; coord_y < ARR_SIZE; ++coord_y)
    {
        for (int coord_x{ 0 }; coord_x < ARR_SIZE; ++coord_x)
        {
            std::cout << arr[coord_y][coord_x];
        }
    }

    return 0;
}
```

```
// [예시 2]
#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE][ARR_SIZE] = {{0},};

    for (int coord_x{ 0 }; coord_x < ARR_SIZE; ++coord_x)
    {
        for (int coord_y{ 0 }; coord_y < ARR_SIZE; ++coord_y)
        {
            std::cout << arr[coord_y][coord_x];
        }
    }

    return 0;
}
```

¹⁸ 이 설명은 동적 다차원 배열 기준이며, 정적 다차원 배열은 조직 방식이 약간 다르다

위 두 코드의 차이점은 순회를 위한 인덱스 변수의 중첩 순서 뿐이다
위와 같은 코드들은 실제로 동작했을 때 어떤 차이점을 보일까?

다음의 영상을 먼저 시청하고 다음의 추가적인 설명을 확인하길 바란다. 다음의 영상은 MIT OpenCourseWare에서 제공하는 'MIT 6.172 Performance Engineering of Software Systems, Fall 2018' 코드의 1강 내용 중 일부이다

[https://www.youtube.com/watch?v=o7h_sYMK_oc&list=PLUI4u3cNGP63VIBQVWguXxZZi0566y7Wf&index=1&t=314s]

[29:59 - 34:28]

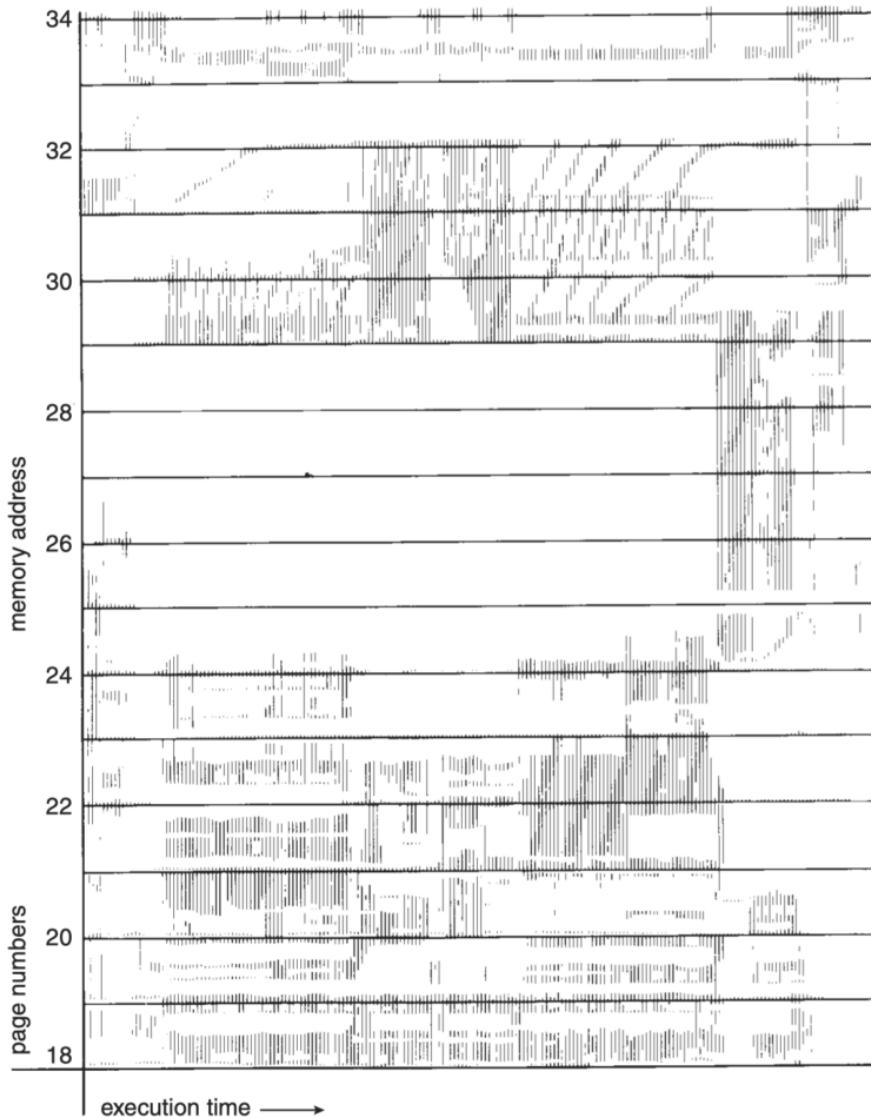
즉, 이 문제는 Memory Indirection Cost를 얼마나 줄일 수 있느냐에 따라 보다 효율적인 중첩 순서가 결정되는 것이다

Memory Indirection Cost란 말 그대로 메모리 상에서 다른 곳에 접근하고자 할 때 발생하는 비용의 총합을 말하는 것이다. 이때 고려해야 할 변수는 상당히 다양한데 간단하게만 살펴봐도 다음과 같이 여러개가 존재할 것이다

- 접근하고자 하는 객체가 메모리상에서 조직된 구조
- L1, L2 등의 Cache Memory에서의 Cache Hit 여부
- Memory Controller의 Scheduling Method와 메모리 접근을 두고 경쟁하는 다른 Process들의 상태
- Memory Bank 수준에서의 Row Buffer Hit 여부

이 중에서도 이번 질문과 특히나 깊게 연관된 것은 Cache Memory와 Memory Bank에서의 Row Buffer Hit 여부이며, 그 중에서도 Cache Memory가 더욱 큰 영향이 있다

Cache Memory는 지역성(Locality)라는 개념과 연관이 깊은데, 이 지역성도 시간 지역성(Temporal Locality)과 공간 지역성(Spatial Locality)으로 구분할 수 있다
이는 다음의 이미지를 보면 쉽게 이해할 수 있다



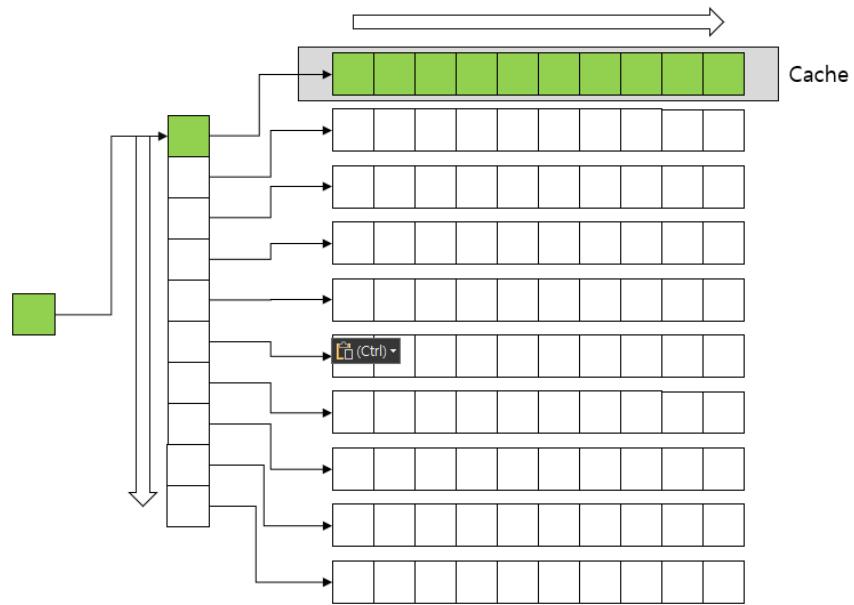
위의 이미지는 프로세스에서 메모리로의 접근을 접근한 시간과 접근한 주소를 기준으로 시각화한 그래프이다. 이 그래프에서 세로로 길게 이어져 있는 부분이 공간 지역성을 보인 부분이며, 가로로 길게 이어져 있는 부분이 시간 지역성을 보인 부분이라고 할 수 있다

즉, 같은 메모리 주소에 다시 접근할 때 시간 지역성이 있다고 하며, 비슷한 메모리 주소에 다시 접근할 때 공간 지역성이 있다고 한다

그리고 Cache Memory는 메모리 접근할 때 접근하는 그 지점을 포함한 커다란 덩어리를 통째로 Cache Memory에 가져온 후, 만약 해당 데이터 덩어리 중에서 Cache Hit가 발생되면 보다 높은 레벨(예컨데 L3에서 L2, L2에서 L1으로)로 그 부분을 이동시키는 방식으로 동작하며, Cache Memory를 둘러보는 순서는 L1부터 L3까지 순서대로 진행된다. 당연히 레벨이 높은 Cache일수록 물리적으로 가장 Core에 가깝기 때문에 접근 시간이 빠르다

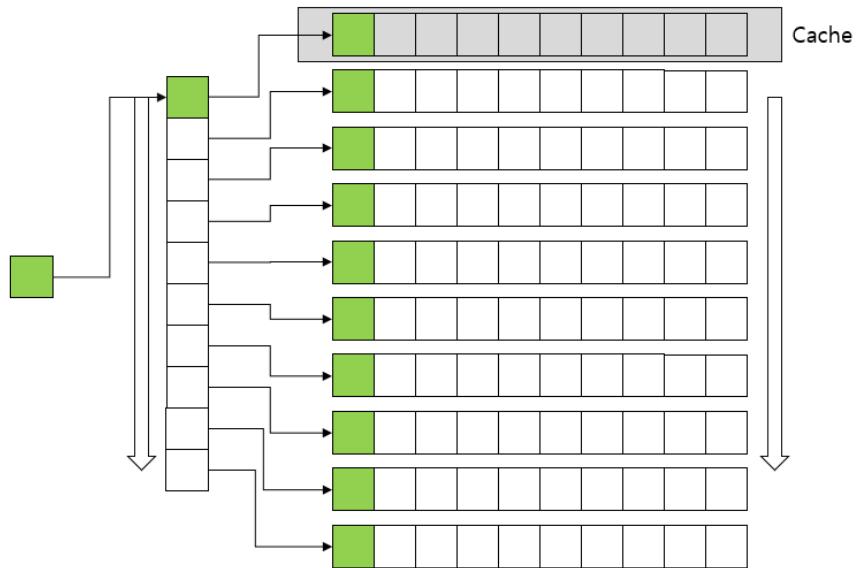
이러한 정보를 알고서 이제 위의 코드들을 분석해보자

우선 [예시 1]의 코드에서의 배열로의 접근 과정을 도식화해보면 다음과 같다



이 그림에서 보면 알 수 있듯 낮은 차원을 담당하는 부분은 Fix하고 보다 높은 차원을 담당하는 부분을 Iterate하면 Cache에 담긴 부분에 지속적으로 접근할 수 있게 되고(Cache Hit) 이는 결과적으로 Indirection Cost의 감소로 이어져 성능 향상을 불러오게 된다

이에 비해 [예제 2]의 코드에서의 배열로의 접근 과정을 도식화하면 다음과 같다



보다시피 매번 Cache Memory에 담긴 부분은 피해가며 접근하기에 Cache Hit는 발생할 수 없게 되고 이는 Indirection Cost의 증가로 이어지게 된다

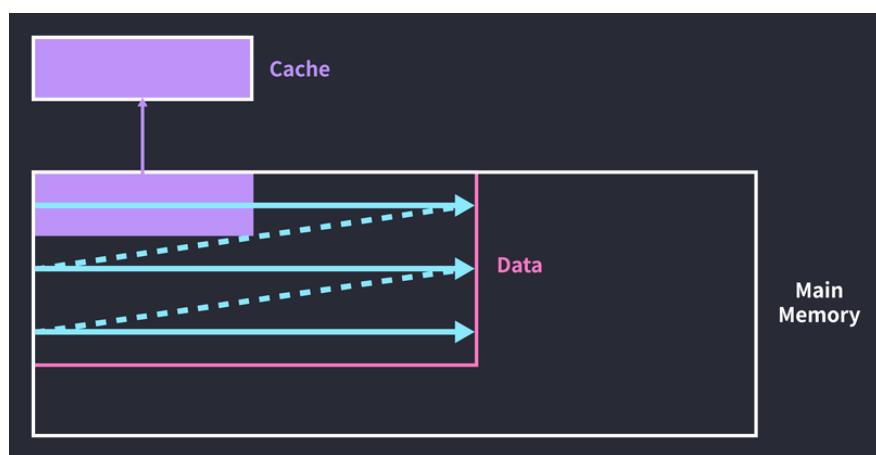
위의 분석은 공간 지역성에 기반한 분석 및 최적화라고 할 수 있다
그런데 시간 지역성에 기반한 최적화는 할 수 없을까? 다음과 같은 예시를 보자

```

for (int count{0}; count < repeatCount; ++count)
{
    for (int index{0}; index < len(arr); ++index)
    {
        arr[index] = pow(arr[index]);
    }
}

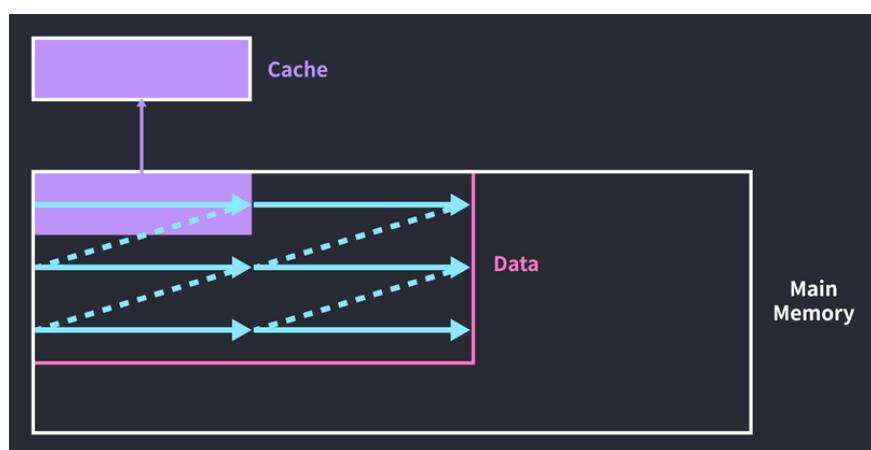
```

이 코드는 얼핏 보기에는 공간 지역성을 충족하며 따라서 어느 정도의 Cache Hit를 보장할 것이다. 그러나 완전하지는 않은데 이는 다음의 이미지를 보면 한 눈에 알 수 있다



즉, arr의 앞 부분과 뒷 부분을 Cache Memory의 크기만큼 잘라서 번갈아가면서 Cache Memory에 갈아끼면서 접근하고 있는 것이다
때문에 이러한 상황에서는 Cache Memory에 arr의 일부분을 갈아끼우는 그 횟수만큼 Cache Conflict가 발생할 것이다

이 문제를 해결하기 위해서는 다음과 같이 접근하면 된다



즉, arr의 일정 부분을 Cache Memory에서 쓸만큼 다 사용한 후, arr의 그 다음 부분을 Cache Memory에 넣고를 반복하는 것이다

이렇게 할 경우 Cache Conflict가 발생하는 하는 횟수는 단순히 생각해도 공간 지역성만 고려하여 코드를 작성할 때보다 크게 감소할 것이다

시간 지역성을 고려한 코드는 다음과 같다¹⁹

```
for (int index_CacheChunk{0}; index_CacheChunk < len(arr); i += CACHE_SIZE)
{
    for (int count{0}; count < repeatCount; ++count)
    {
        for (int index{0}; index < CACHE_SIZE; ++index)
        {
            arr[index_CacheChunk + index] = pow(arr[
index_CacheChunk + index]);
        }
    }
}
```

Cache Hit와 관련하여 재밌는 실험을 할 수 있는 사이트 링크를 남겨두니 보다 깊은 이해를 위해 참고하면 좋을 듯 하다

[<https://www3.ntu.edu.sg/home/smitha/ParaCache/Paracache/start.html>]

- 어떻게 더 반복문의 반복을 줄일 수 있을까? - break, return

반복문에서 활용할 수 있는 제어문으로는 break와 continue가 있으며, 추가적으로 본래는 반복의 흐름 제어를 위해 있는 것은 아니지만 return 또한 반복문에서의 흐름 제어에 활용될 수 있다

다음과 같은 예시를 생각해보자

```
#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE]{ {0, } };

    int findTarget = 5;
    bool targetExist = false;
    for (int index{ 0 }; index < ARR_SIZE; ++index)
    {
        if (arr[index] == findTarget)
```

¹⁹ 아래의 코드는 단순화를 위해 $[(\text{len}(\text{arr}) * \text{sizeof}(\text{decltype}(\text{**arr}))) \% \text{CACHE_SIZE}]$ 가 0이라는 가정 하에 작성된 코드이며, 실제 제품에서 사용하기 위해서는 다소의 조정이 필요할 것이다

```

        {
            targetExist = true;
        }
    }

    return 0;
}

```

이 코드는 arr에서 목표로 하는 값인 findTarget이 존재하는지를 확인하여 만약 존재한다면 targetExist에 true를 담는 처리를 한다

그런데 잘 생각해보면 만약 arr[0]에서 findTarget와 일치한다는 것을 확인했다고 가정해보자. 그러면 굳이 추가적으로 arr을 더 순회할 필요가 있을까? 이미 arr에 findTarget이 존재하는지 여부는 true로서 판명이 났기에 추가적인 순회는 낭비임이 분명하다
때문에 이러한 경우에는 다음과 같이 처리하여 불필요한 반복은 차단해줌이 바람직하다

```

#include <iostream>

#define ARR_SIZE 10

int main()
{
    int arr[ARR_SIZE]{ {0}, } ;

    int findTarget = 5, resultIndex;
    for (int index{ 0 }; index < ARR_SIZE; ++index)
    {
        if (arr[index] == findTarget)
        {
            resultIndex = index;
            break;
            // or return;
        }
    }

    return 0;
}

```

- 추가적으로 직접 찾아보고 생각해보면 좋을만한 질문들

- 반복문 내의 지역 변수의 생성 및 소멸은 각각 어디서 어떻게 하는 것이 좋을까? 반복문의 중첩 깊이를 줄일 혹은 줄은 것처럼 보이게 하는 방법은 없을까?
 - 현재의 반복문과 조건문의 조건을 봤을 때, Branch Prediction이 너무 심하게 깨지지 않을까?

- 고도로 중첩된 반복문에서 내가 원하는 수준으로 콕 찍어 벗어날 방법은 없을까?
- 어떻게 반복을 작성해야 읽기 편할까?

4.2.2.4. 반복 처리를 줄이기 위해 혹은 보다 효율적으로 처리하기 위해 고려 가능한 방안들

- break, return
- goto(주의)
- SIMD
 - [<https://blog.naver.com/lloie6478/222389369471>]
- GPU Processing
 - [<https://docs.unity3d.com/kr/2018.4/Manual/class-ComputeShader.html>]
 - [https://www.khronos.org/opengl/wiki/Compute_Shader]
- Multi-Threading
 - [<https://youtu.be/M1e9nmmD3II>]
- Loop Unroll
 - [<https://blog.popekim.com/ko/2012/01/11/compile-time-hash-string-generation.html>]
- Memory Access Request 독점에 따른 배열 순회 접근 성능 향상
 - [https://users.ece.cmu.edu/~omutlu/pub/mph_usenix_security07.pdf]

4.2.3. 조건

4.2.3.1. 조건의 종류

- 레이블링 조건(ex. Dictionary, Hash Table)

분류, 종류, 플래그 등의 특정 집합을 구성하여 제어 가능한 조건

- 일반 조건

레이블링 가능한 조건을 제외한 나머지 모든 조건

4.2.3.2. 조건문의 종류 및 문법

- if, else~if, else

```
attr (optional) if constexpr(optional) ( init-statement (optional)
condition ) statement-true

attr (optional) if constexpr(optional) ( init-statement (optional)
condition ) statement-true else statement-false
```

간단한 예시는 다음과 같다

```
template<typename T>
```

```

auto get_value(T t)
{
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t; // deduces return type to int for T = int
}

```

- **switch**

```
attr (optional) switch ( init-statement (optional) condition ) statement
```

간단한 예시는 다음과 같다

```

switch (1)
{
    case 1:
        int x = 0; // initialization
        std::cout << x << '\n';
        break;
    default:
        // compilation error: jump to default:
        // would enter the scope of 'x' without initializing it
        std::cout << "default\n";
        break;
}

```

4.2.3.3. 한 번쯤 생각해보면 좋을 것 같은 질문들

- 다수의 조건식이 논리 연산자로 엮여있을 때 어떤식으로 처리될까?
- 다수의 조건식이 논리 연산자로 엮여있을 때 조건식들을 어떤 순서로 배치해야 효율적일까?
- switch는 어떨 때 사용해야 유리할까?
- switch를 사용할 바에는 LUT를 사용하는 게 더 좋지 않을까?
- else나 default는 어떤식으로 활용할 수 있을까?
- 고도로 중첩된 조건문을 어떻게 읽기 편하게 만들 수 있을까?
- 조건문이 반복문 안에 있을 때와 밖에 있을 때, 어떨 때 더 빠르고 느릴까?
- 조건으로서의 goto의 활용은 왜 금기시 되는 걸까?

4.2.3.4. 조건 처리를 줄이기 위해 혹은 보다 효율적으로 처리하기 위해 고려 가능한 방안들

- LUT
- Branch Prediction

5. 함수

5.1. 함수의 종류

5.1.1. 종류

함수는 종류를 불문하고 입력값, 처리 과정, 결과값의 세 부분이 공통적으로 존재한다
그러나 그 이외로는 큰 갈래의 종류에 따라 세세한 차이가 존재할 수 있다
그 종류는 다음과 같다

일반적인 함수
매크로 함수
익명 함수
클래스 내 함수

5.1.2. 함수의 동작에 변수를 주는 요소들

함수의 동작은 매우 추상화하여 간단하게 생각하면 입력값을 중간 처리 과정을 거쳐 결과값을 돌려주는 것만이 동작이라고 받아들이기 쉽다
그러나 이것은 지극히 추상적인 개념일 뿐이며, 함수의 동작에 영향을 줄 수 있는 요소는 굉장히 다양하다
그 종류는 다음과 같다

함수의 이름	필수
인자 리스트	()와 같이 비어있을 수 있다
반환 타입	void일 수 있으며, 전위형이거나 후위형일 수 있다
inline	함수 호출이 함수 본체를 인라인으로 넣어서 구현되게 하겠다는 의사를 보이는 것이다
constexpr	상수 표현식이 인자로 주어진 경우 컴파일 타임에 함수를 평가하는 것이 가능해야 한다는 점을 나타낸다
throw() (C++ 98) noexcept (C++ 11)	함수가 예외를 던지지 않을 것이라는 것을 나타낸다
[[noreturn]] : Attribute	함수가 일반적인 호출/반환 메커니즘을 이용해서 반환하지 않을 것이라는 점을 나타낸다
virtual	파생 클래스에서 재정의될 수 있다는 점을 나타낸다
override	기반 클래스의 가상 함수를 재정의할 것이라는 점을 나타낸다

final	파생 클래스에서 재정의할 수 없다는 점을 나타낸다
static	특정 객체와 연관되지 않는다는 점을 나타낸다
const	그것의 객체를 변경하지 않는다는 점을 나타낸다
volatile	volatile 객체에 적용될 수 있다는 점을 나타낸다
template	주어진 임의의 타입들에 대해 대응되는 유형의 복제본을 만들어낸다는 점을 나타낸다
explicit	인자 리스트의 인자들에 대하여 암시적 변환이 불가함을 나타낸다
mutable	해당 키워드가 선언된 멤버 변수는 상수 함수일지라도 값의 변경이 가능함을 나타낸다
&, && : Reference Specifier	함수가 접근하는 this*가 L-Value이거나 R-Value이거나에 따라 암시적으로 호출되는 중의적 오버로딩 함수를 정의함을 나타낸다

다만, 위의 목록은 함수 단독으로 있을 때 뿐만 아니라 클래스의 매서드도 상정한 목록임을 인지하길 바란다

5.1.3. 매개변수 및 리턴값의 종류 및 성능 비교

C++은 세 가지 종류의 인자 리스트 및 반환값의 전달 메커니즘을 제공한다

Call By Value Return By Value	<p>전달되고자 하는 객체는 해당 객체의 복사 생성자를 통해 초기화된 후 전달된다</p> <p>따라서 실제 전달되는 객체는 원본 객체와 같은 자료 구조를 가졌지만 메모리 상에서 다른 위치에 존재하는 다른 객체인 것이다</p>
Call By Pointer Return By Pointer	<p>전달되고자 하는 객체는 해당 객체가 메모리 상에 존재하는 주소로 포인터를 초기화하여 해당 포인터를 전달한다</p> <p>따라서 실제 전달되는 객체는 원본 객체가 아닌 원본 객체를 가리키는 포인터 일 뿐이며, 원본 객체로의 접근은 포인터를 통한 간접 참조를 통해 이루어진다</p> <p>반환값에 대해서는 정적 할당 지역 변수에 대한 Return By Pointer는 상식적으로 말도 안 되는 일이기에 삼가해야만 한다</p>
Call By Reference Return By Reference	<p>Call By Pointer / Return By pointer와 기본적인 동작 방식은 같다 그러나 Reference의 특성에 따라 해당 매개변수는 nullptr일 수 없다</p> <p>반환값에 대해서는 Return By Pointer에서의 경우와 같이 정적 할당 지역 변수에 대한 Return By Reference는 말도 안 되는 일이기에 삼가해야만 한다</p>

매개변수 및 리턴값의 종류에 대해 이야기할 때 곧잘 잇달아 나오는 주제가 전달 방식들의 성능과 관련된 이슘이다
질문은 상당히 단순한데 ‘어떤 상황에서 어떤 인자 전달 방식을 사용하는 것이 효율적인가?’이다

- 클래스의 Call By Value에 의한 전달 비용

먼저, 클래스와 같이 내부에 규모가 있는 자료 구조를 지니는 타입의 Call By Value 전달에 의한 비용을 살펴보자

```
class Person
{
private:
    std::string m_name;
    std::string m_address;

public:
    Person();
    virtual ~Person();
};

class Student : public Person
{
private:
    std::string m_schoolName;
    std::string m_schoolAddress;

public:
    Student();
    ~Student();
};

bool CheckValidStudent(Student student);
```

클래스와 같이 내부에 규모가 있는 자료 구조가 포함될 수 있는 타입의 경우 Call By Value 전달의 경우 이를 복사 생성자를 통해 처리하며, 어떤 클래스가 Call By Value 전달을 통해 전달될 때 얼마만큼의 비용이 요구되는지 보려면 복사 생성자가 어떻게 동작하는지를 살펴보면 일목요연하다

위의 예제에서 만약 Student 객체를 하나 만든 후 해당 객체를 CheckValidStudent(...)에 넣는다고 가정하고 일어날 일들을 분석해보면 다음과 같다

컴파일 타임

- CheckValidStudent(...)가 호출되는 사실 + 인자 전달이 Call By Value로 이루어진다는 점으로부터 Student의 복사 생성자의 호출이 필요하다는 사실 유추 그러나 Student의 선언부에는 복사 생성자가 존재하지 않기에 컴파일 타임에 컴파일러 버전의 복사 생성자 추가

련타임

1. CheckValidStudent(...) 호출
 - 1.1. 인자(Student) 초기화
 - 1.1.1. 부모 클래스인 Person의 기본 생성자 호출
 - 1.1.1.1. 멤버 변수 Person::name 초기화를 위한 생성자 호출
 - 1.1.1.2. 멤버 변수 Person::address 초기화를 위한 생성자 호출
 - 1.1.2. 자식 클래스인 Student의 컴파일러 버전 복사 생성자 호출
 - 1.1.2.1. 멤버 변수 Student::schoolName 초기화를 위한 생성자 호출
 - 1.1.2.2. 멤버 변수 Student::schoolAddress 초기화를 위한 생성자 호출
 2. 인자값 접근
 - 2.1. 원본 객체의 사본인 인자에 대한 접근
 3. CheckValidStudent(...) 반환
 - 3.1. 인자 소멸
 - 3.1.1. 자식 클래스인 Student의 소멸자 호출
 - 3.1.1.1. 멤버 변수 Student::schoolName 소멸을 위한 소멸자 호출
 - 3.1.1.2. 멤버 변수 Student::schoolAddress 소멸을 위한 소멸자 호출
 - 3.1.2. 부모 클래스인 Person의 소멸자 호출
 - 3.1.2.1. 멤버 변수 Person::name 소멸을 위한 소멸자 호출
 - 3.1.2.2. 멤버 변수 Person::address 소멸을 위한 소멸자 호출

위의 표를 보면 알 수 있듯 매우 간단한 자료 구조와 상속 구조를 가지는 클래스의 복사 생성자의 경우에도 함수 호출 당시의 인자 초기화를 위해 내부적으로 총 6번의 생성자가 호출되고, 함수의 모든 처리가 끝나 결과값을 반환하기 전 내부 자료를 해제할 당시에도 내부적으로 총 6번의 소멸자가 호출된다

심지어 만약 호출되는 생성자와 소멸자가 내부적으로 대입 연산을 취할 경우에는 대입 연산이 발생한 만큼 복사 대입 생성자가 추가적으로 호출될 것이며, 이를 모두 뒤로 하고 난 후에도 생성자와 소멸자 자체의 코드의 품질에 의한 성능도 전적으로 가능한 최고의 최적화를 염두에 둔 것이라고 생각하기는 어렵다

- 클래스의 Call By Pointer / Call By Reference에 의한 전달 비용

```
class Person
{
private:
    std::string m_name;
    std::string m_address;

public:
    Person();
    virtual ~Person();
};

class Student : public Person
{
private:
```

```

    std::string m_schoolName;
    std::string m_schoolAddress;

public:
    Student();
    ~Student();
};

bool CheckValidStudent(const Student& student);

```

위의 코드는 앞선 [클래스의 Call By Value에 의한 전달 비용]에서 보였던 예제 코드에서 `CheckValidStudent(...)`의 인자 리스트 부분이 `Student student`에서 `const Student& student`로 바뀐 것 말고는 차이가 없는 코드이다

이제 여기선 무슨 일이 발생하는지 분석해보면 다음과 같다

컴파일 타임

런타임

1. `CheckValidStudent(...)` 호출
 - 1.1. 인자 초기화
 - 1.1.1. Pointer / Reference 생성자 호출
2. 인자값 접근
 - 2.1. Pointer / Reference에 의한 간접 참조
3. `CheckValidStudent(...)` 반환
 - 3.1. Pointer / Reference 소멸자 호출

위의 표에서 볼 수 있듯 Call By Value에 비해 단순하다

포인터는 그 크기가 플랫폼 의존적이지만 동일한 플랫폼에선 항상 크기가 일정하고, 복사 생성자를 통한 복사 방식이 비트 수준에서의 복사를 통해 이루어진다는 점에서 Built-In Type과 같다고 봐도 무방하다

그렇기에 `CheckValidStudent(...)`의 호출과 반환에서 발생하는 인자의 생성과 소멸에서 발생하는 비용은 심지어 어셈블리 수준에서 비트 수준 연산이 몇 번 발생하느냐까지 분석할 수 있을 정도로 간단한 편이다

다만, 인자값에 대한 접근도 인자에 대한 직접 참조가 아닌 간접 참조에 의한 피할 수 없는 비용이 있기에 인자값 접근에 대한 비용 자체는 Call By Value에 비해 보다 많은 비용이 요구된다고 볼 수 있다

- 클래스의 보다 효율적인 전달 방식

앞선 각 전달 방식에 대한 분석에 따라 클래스의 보다 효율적인 전달 방식을 본다면 Call By Pointer / Call By Reference를 사용하는 편이 꽤 많이 효율적이며, 안전하기까지 하다²⁰

²⁰ 예를 들어 복사손실 문제로부터

- Built-In Type의 Call By Value에 의한 전달 비용

```
bool CheckValudNum(int num);
```

위의 코드의 경우와 같이 Built-In Type이 Call By Value에 의해 인자로 전달될 때에는 많은 경우 복사 생성자와 같은 개념으로 사본의 초기화가 일어나게 되는데, 이때의 초기화는 비트 수준에서의 복사가 일어나게 된다

인자값으로의 접근은 일반적인 감각의 변수와 다를 바가 없기에 굉장히 빠른 편의 접근 속도를 보여준다

함수가 반환됨과 함께 인자가 소멸될 때에는 그저 Stack에 쌓인 바이트 덩어리에 대한 참조만 날려버리면 되기에 빠른 편이라고 할 수 있다

- Built-In Type의 Call By Pointer / Call By Reference에 의한 전달 비용

```
bool CheckValudNum(const int& num);
```

위의 코드의 경우와 같이 Built-In Type이 Call By Pointer / Call By Reference에 의해 인자로 전달될 때에는 Pointer 혹은 Reference의 초기화 방식 자체는 일반적인 Built-In Type과 다르지 않기에 방식 그 자체에 의한 성능 차이는 존재하지 않을 수 있다

그나마 차이가 존재할 수 있는 부분은 가리키고자 하는 Built-In Type의 크기와 플랫폼 의존적인 Pointer 혹은 Reference의 크기에 따른 초기화할 비트의 수 차이밖에 없을 것이다

인자값으로의 접근은 간접 참조에 의한 접근 비용의 증가로 인해 Call By Value보다 반드시 비쌀 수 밖에 없다

함수가 반환됨과 함께 인자가 소멸될 때에는 Call By Value 방식과 같이 그저 Stack에 쌓인 바이트 덩어리의 참조만 날려버리면 되기에 빠른 편이다

- Built-In Type의 보다 효율적인 전달 방식

Built-In Type의 경우 전달 방식에 따라 보다 효율적인 전달 방식은 매우 명확한데, Built-In Type은 Call By Value 전달이 Call By Pointer / Call By Reference 전달보다 빠르다

5.1.4. 특수한 함수

- 순수 함수(수학 함수)

순수 함수란 수학 함수란 이름으로도 불린다

순수 함수는 이름 그대로 마치 수학에서의 함수가 같은 입력값에 대해 항상 같은 결과값을 내놓는 것과 같이 입력값에 대해 항상 같은 결과값을 내놓는 함수를 순수 함수라고 한다
다만, 이때의 순수 함수란 단어를 다형성 구현을 위한 순수 가상 함수라는 단어와 혼갈리지 않도록 주의하길 바란다

5.2. 일반적인 함수

5.2.1. 일반적인 함수의 문법

5.2.1.1. 함수 선언

- 함수 선언문의 구성

8.4 Function definitions

[**dcl.fct.def**]

- 1 Function definitions have the form

function-definition:
decl-specifier-seq_{opt} declarator ctor-initializer_{opt} function-body
decl-specifier-seq_{opt} declarator function-try-block

function-body:
compound-statement

The *declarator* in a *function-definition* shall have the form

D1 (parameter-declaration-clause) cv-qualifier-seq_{opt} exception-specification_{opt}

as described in 8.3.5. A function shall be defined only in namespace or class scope.

- 2 [Example:] a simple example of a complete function definition is

```
int max(int a, int b, int c)
{
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here *int* is the *decl-specifier-seq*; *max (int a, int b, int c)* is the *declarator*; { /* ... */ } is the *function-body*.]

- 3 A *ctor-initializer* is used only in a constructor; see 12.1 and 12.6.

- 4 A *cv-qualifier-seq* can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.3.2. It is part of the function type.

- 5 [Note:] unused parameters need not be named. For example,

```
void print(int a, int)
{
    printf("a = %d\n", a);
}
```

—end note]

- 함수 정의

함수의 절대적인 첫 번째 규칙은 호출되는 모든 함수는 반드시 어딘가에 정의되어 있어야만 한다는 것이다

함수는 선언부와 정의부가 구분될 수 있으며 예제 코드로 보자면 다음과 같다

```
void Swap(int*, int*);
void Swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
```

```
*b = temp;  
}
```

위의 예제 코드에서 추가적으로 볼 수 있는 점은 함수의 선언부에는 인자 리스트에 식별자가 반드시 포함될 필요는 없다

또한, 선언부와 정의부의 시그니처의 반환값과 인자 리스트의 타입은 둘이 동일해야하며, C와의 호환성을 위해 `const`는 함수 오버로딩 규칙에서 무시된다. 즉, `const int`와 `int`는 함수 오버로딩 측면에서 같은 타입으로 취급한다

- 값 반환

함수 선언에는 반드시 반환값의 타입에 대한 세부 사항이 포함되어 있다
전통적으로 C/C++에서는 반환값 타입이 함수 선언의 맨 앞부분에 존재하지만, 오히려 함수 선언의 맨 뒷부분에 위치시키는 것 또한 가능하다

```
int GetNum();  
auto GetNum() -> int;
```

다만, 후위형 반환 타입의 경우 람다식의 표기법과 문법적 유사성으로 인해 헷갈릴 수 있으니 주의

함수는 선언에 포함된 반환값의 타입에 대한 세부 사항에 따라 값을 반환하게 되어있으며, 이때 `return`을 통해 값을 반환하게 된다
그러나 하나의 함수에는 다수의 `return`이 존재할 수 있다

그리고 함수는 흔히 알려진 `return`을 통해 빠져나가는 방법을 포함하여 총 다섯 개의 방법이 존재한다

- return문의 실행

- 함수의 끝에서 떨어져 나간다

즉, 말 그대로 함수의 진정한 끝인 함수의 스코프 마지막까지 도달하는 것이다
흔히하는 착각으로 함수의 가장 마지막 부분에 존재하는 `return`에 도달하면 함수의 끝에 도달한 것으로 생각할 수 있지만, `void` 함수나 `main` 함수와 같은 경우에만 함수의 진정한 끝에 도달할 수 있다

- 지역적으로 잡히지 않는 예외를 던지는 경우

- 예외가 던져지고 `noexcept` 함수에서 지역적으로 잡히지 못하는 것으로 인한 종료

`noexcept`는 `operator, specifier`의 두 가지 형태로 C++ 11부터 추가되었다

이번 강의록의 범위는 벗어나지만 `std::operator noexcept()`의 경우 컴파일 타임에 `operator`의 인자로 주어진 표현식이 예외를 던지는 표현식인지를 체크하여 다음의 경우 중 하나라도 포함되면 `false`를, 그렇지 않으면 `true`를 반환한다

- 상수 표현식이 아닌 함수가 noexcept 키워드를 가지지 않을 경우
- 런타임 체크가 필요한 dynamic_cast 등의 RTTI가 포함된 경우
- typeid 표현식에 포함된 타입이 상속 관계에 있는 클래스나 구조체일 경우

그리고 이번 강의록의 범위인 noexcept specifier의 경우 다음과 같이 쓰일 수 있다

```
int GetNum() noexcept;
auto GetNum() noexcept -> int;
```

만약 noexcept specifier가 선언에 포함된 함수에서 만약 예외가 발생할 경우 std::terminate가 호출된다

std::terminate가 호출되면 잇달아 std::terminate_handler를 호출하게 되는데 std::terminate_handler가 기본적으로 잡고 있는 기본 동작 함수가 std::abort이다

즉, 기본 동작 그대로 사용한다면 ‘예외를 던지지 않기로 한 함수에서 던져진 예외를 적절히 관리하지 못하고 함수 밖으로 터져나가려 한다면 프로그램 자체를 종료시킨다’는 것이다 물론, 이러한 방식의 함수 탈출 방식은 std::terminate_handler가 잡고 있는 함수를 변경할 경우 항상 유효하지 않을 수 있다

여담으로 다음의 함수들은 기본적으로 noexcept를 가진다

- 암시적으로 생성되는 기본 생성자, 복사 생성자, 복사 대입 생성자, 이동 생성자(C++ 11), 이동 연산자, 소멸자
- 유저가 명시적으로 noexcept(false)로 선언했거나 혹은 부모의 소멸자가 noexcept(false)로 선언된 경우를 제외한 모든 사용자 정의 소멸자
- operator delete 함수들

C++ 11 이후 Modern C++에서의 noexcept는 최적화 측면에서도 중요하다

C++ 11 이후의 STL은 move semantics 개념이 도입되어 잘만 활용하면 성능상의 이득을 꾀할 수 있게 되었다

객체에 대해 이동 처리를 할 때 해당 객체가 noexcept, 즉, 예외에 대해 완전히 안전하다는 보장을 해주지 못한다면, 즉, strong exception guarantee를 보장하지 못한다면 move semantics가 아닌 copy semantics로 이동 처리를 하게 되고 따라서 성능상의 불익이 생기게 된다는 것이다

예외 안전성 보장의 종류는 다음과 같다

Basic Exception Guarantee	<p>함수 동작 중에 예외가 발생하면 실행 중인 프로그램에 관련된 모든 것들을 유효한 상태로 유지하겠다는 보장이다</p> <p>어떤 객체나 자료 구조도 더럽혀지지 않으며, 모든 객체의 상태는 내부적으로 일관성을 유지한다 하지만 프로그램의 상태가 정확히 어떠한지는 예측이 안 될 수도 있다 즉, 프로그램의 상태가 유효하기만 하면 예외 발생 후에 정확히 어떤 상태로 돌아갈지에 대해 호출하는 입장에서는 전혀 예상할 수 없다</p>
Strong Exception Guarantee	함수 동작 중에 예외가 발생하면 프로그램의 상태를

	<p>절대로 변경하지 않겠다는 보장이다 이런 함수를 호출하는 것은 원자적인(atomic) 동작이라고도 할 수 있다</p> <p>호출이 성공하면 마무리까지 완벽하게 성공하고, 호출이 실패하면 함수 호출이 없었던 것처럼 프로그램의 상태가 호출 이전 상태로 돌아간다는 것이다</p> <p>즉, 어떤 함수가 Strong Exception Guarantee를 보장한다면 해당 함수의 호출 후에 가능한 상태는 '성공적으로 실행을 마친 후의 상태'와 '실패했을 때 함수가 호출될 때의 상태'의 두 가지 상태만이 존재한다는 것이다</p>
Nothrow Exception Guarantee	<p>예외를 절대 던지지 않겠다는 보장이다 Nothrow Exception Guarantee은 'throw()'나 'noexcept'를 통해 구현된다</p> <p>그러나 Nothrow Exception Guarantee를 보장한다고 해서 해당 함수에서 예외가 발생하지 않는다는 말은 아니다. Nothrow Exception Guarantee는 그저 예외를 던지지 않겠다는 보장일 뿐인 것이다</p> <p>이는 말장난처럼 느껴질 수 있겠지만 예외가 발생하는 것과 던져지는 것은 다른 개념이다 단순히 발생 한 것은 말 그대로 예외가 발생하기 만한 것이고, 던져진다는 것은 함수의 호출 스택을 거슬러 올라가며 예외가 터져나간다는 것을 말하는 것이다</p> <p>만약 Nothrow Exception Guarantee를 보장하는 함수에서 예외가 던져질 가능성이 존재한다면 약속된 동작을 수행하게 되며, 일반적으로 기본 동작으로서 프로그램 자체를 종료시킨다</p> <p>즉, 사실 Nothrow Exception Guarantee를 보장한다고 해서 반드시 예외 안전성을 보장한다는 것은 아니라 아예 보장하지 않을 수도 있다는 것이다</p>
즉, 정리하자면 성능상의 이득을 꾀하고자 한다면 noexcept를 통해 strong exception guarantee를 보장하는 것이 중요하다	
- 반환하지 않는 시스템 함수(exit() 등)을 직접적 혹은 간접적으로 호출하는 경우	

- **inline** 함수

inline specifier는 해당 함수가 함수 코드를 한 번 저장한 다음, 통상적인 함수 호출 메커니즘을 통해 호출하는 것이 아니라 인라인 호출 코드를 생성하려고 시도해봐야 한다는 점을 컴파일러에게 알려주는 단어이다

이때, 어떤 함수가 **inline** 함수인가 하는 여부는 암시적 방식과 명시적 방식이 모두 존재한다

암시적 inline	<p>암시적 inline 함수의 경우에는 클래스의 선언부에 함수의 완전한 정의까지 포함할 경우 해당 함수는 inline 함수 후보로 암시적으로 지정된다</p>
	<pre>class Person { private: int m_age; public: int GetAge() const { return m_age; } };</pre> <p>위의 예제 코드의 경우 <code>Person::GetAge()</code>의 경우 inline specifier가 명시적으로 선언되어 있지 않지만 <code>Person</code> 클래스의 선언부에 매서드의 선언과 함께 완전한 정의가 포함되어 있기에 inline 함수의 후보로 지정된다</p> <p>주의할 점은 어디까지나 inline 함수의 ‘후보’가 되었을 뿐 inline 될 수 있는 조건을 추가적으로 모두 만족해야만 실제로 inline 될 수 있다</p>
명시적 inline	<p>명시적 inline 함수의 경우에는 모든 함수에 inline specifier가 선언될 경우 inline 함수의 후보로서 지정된다</p> <pre>inline int GetUID(const std::string& name);</pre> <p>위의 예제 코드와 같이 명시적으로 inline specifier를 선언할 경우 해당 함수는 inline 함수의 후보로 지정된다</p> <p>주의할 점은 암시적 inline에서 언급한 것과 같이 어디까지나 inline 함수의 ‘후보’가 되었을 뿐 inline 될 수 있는 조건을 추가적으로 모두 만족해야만 실제로 inline 될 수 있다</p>

어떤 함수가 **inline** 함수의 후보로서 지정되는 간단한 규칙은 위와 같다

그러나 **inline** 함수의 후보로 지정된 함수가 실제로 **inline** 되는데에는 해당 함수가 몇몇 추가적인 조건을 만족해야만 한다
해당 조건을 간단하게 요약하자면 ‘조금이라도 복잡한 함수는 절대 **inline** 되지 않는다’이다

- 함수의 내용에 루프가 들어가지 않는다

- 재귀 함수의 형태를 취하지 않는다
- 가상 함수 호출 형태(다형성)를 취하지 않는다
- 함수 정의 본문은 하나만 존재해야 한다
- 모든 컴파일 단위에서 동일한 방식으로 선언되어야만 한다

어찌보면 굉장히 단순한 조건들이다. 물론, 이 조건들은 컴파일러 의존적이기에 항상 적용되는 목록은 아닐 수 있다²¹

그러나 어떤 함수가 `inline` 후보로 지정되어 있고 위의 조건들도 만족하여 이건 무조건 `inline` 되어야만 한다고 확신하는 경우에도 실제로는 되지 않는 경우가 존재할 수 있다

```
inline int GetUID(const std::string& name);

int (*GetUIDPtr)(const std::string& name) = GetUID;

void Sample()
{
    std::string name{"John"};

    GetUID(name);

    GetUIDPtr(name);
}
```

위의 예제 코드에서 `inline int GetUID(const std::string& name)`는 확실히 `inline` 되는 함수라고 가정해보자

그러할 경우 `GetUID(name);`의 경우 확실히 `inline`된다. 그러나 `GetUIDPtr(name)`은 확실히 `inline`되지 않는다

차이는 단순한데, 바로 `inline` 함수의 주소를 취한 함수 포인터를 통해 호출되는 함수는 `inline`되지 않는다

이유도 조금만 생각해보면 매우 단순한데, `inline` 함수란 함수의 본문을 함수가 호출된 부분에 붙여넣는 것이다. 즉, 함수라는 형태가 없어지고 본문만 남는 것이다. 그런데 본문만 있는데 함수 포인터는 어떻게 함수의 포인터를 가져올 수 있겠는가

이 결과를 보고 그냥 본인만 `inline` 함수의 함수 포인터를 통해 호출하는 일이 없도록 하면 될 것이라고 생각한다면 순진한 생각일 수 있다

C++의 방대한 메커니즘과 라이브러리 중에는 사용자 정의 함수가 `inline` 함수이든 아니든 가리지 않고 함수 포인터로 가리키며 처리하는 것들이 생각보다 많기 때문이다

또한, `inline` 할 때 주의할 점은 디버깅이 힘들어질 수 있다는 것이다

분명 나는 함수의 호출 부분에 중단점을 걸고 디버깅을 하고 싶은데, 해당 함수의 호출이 `inline` 되었다면 결국엔 존재하지도 않는 함수에 도대체 어떻게 중단점을 건다는 것인가

²¹ 이 소챕터에서 언급하는 `inline`에 대한 사항들은 Visual C++ 기준이다

- constexpr 함수

일반적으로 함수는 컴파일 타임에 평가될 수 없기 때문에 상수 표현식으로 호출될 수 없다 하지만 함수에 constexpr로 선언하면 상수 표현식이 인자로 주어지는 경우 해당 함수가 상수 표현식 안에서 사용되게 만들 수 있다

constexpr 함수의 동작을 정리하자면 다음과 같다

- 컴파일 타입 상수를 요구하는 문맥에 constexpr 함수를 사용할 수 있다 그런 문맥에서, 만일 constexpr 함수에 넘겨주는 인자의 값이 컴파일 시점에서 알려진다면 함수의 결과는 계산된다 인자의 값이 컴파일 시점에서 알려지지 않는다면 코드의 컴파일이 거부된다
- 컴파일 시점에서 알려지지 않는 하나 이상의 값으로 constexpr 함수를 호출하면 함수는 보통의 함수처럼 동작한다. 즉, 그 결과는 실행 시점에서 계산된다 이는 같은 연산을 수행하는 함수를 두 버전, 즉, 컴파일 시점 상수를 위한 버전과 다른 모든 값을 위한 버전으로 나누어서 구현할 필요가 없음을 뜻한다. 그냥 하나의 constexpr 함수를 두 가지 용도로 사용하면 된다

```
constexpr int GetFac(const int n)
{
    return (n > 1) ? n * GetFac(n - 1) : 1;
}

void Test(const int n)
{
    int f5 = GetFac(5);                      // 컴파일 타임에 평가
    int fn = GetFac(n);                      // 런타임에 평가

    constexpr int cf5 = GetFac(5);           // OK - 컴파일 타임에 평가
    //constexpr int cfn = GetFac(n);          // Error

    char arr_0[GetFac(5)];                  // OK - 배열 크기로 컴파일 타임
                                            // 상수 사용 가능
    //char arr_1[GetFac(n)];                // Error
}
```

5.2.1.2. 인자 전달

- 리스트 인자

{ }로 둘러싸인 리스트는 다음의 매개변수에 대한 인자로 사용될 수 있다

- std::initializer_list<T> Type, 리스트의 값이 T로 암시적으로 평가될 수 있다
- 리스트에서 제공된 값으로 초기화될 수 있는 Type
- T로 이뤄진 배열의 참조자, 리스트의 값이 T로 암시적으로 평가될 수 있다

```

#include <initializer_list>
#include <string>

template <typename _T>
void f1(std::initializer_list<_T>);

struct S
{
    int m_age;
    std::string m_name;
};

void f2(S);

void f3(int);

void Test()
{
    f1({0, 1, 2, 3, 4, 5});
    f2({1, "john"});
    f3({1});
}

```

만약 애매한 상황이 발생하면 `std::initializer_list<T>` 매개변수가 우선순위를 가진다

- 인자의 개수가 정해지지 않은 경우

어떤 함수에 대해서는 호출 내에 있어야 될 모든 인자의 개수와 타입을 지정할 수 없는 경우가 있다

그런 인터페이스를 구현하는 데는 3가지 방법이 있다

- 가변 인자 템플릿을 사용한다
이 방법을 사용하면 임의의 타입으로 이뤄진 임의의 개수의 인자를 타입 안전적인 방식으로 처리할 수 있는데, 템플릿 메타 프로그래밍을 통해 인자 리스트의 의미를 해석하고 적절한 동작을 취하게 하는 것이다
- `std::initializer_list<T>`를 인자 타입으로 사용한다
이렇게 하면 단일 타입으로 이뤄진 임의의 개수의 인자를 타입 안전적인 방식으로 처리할 수 있다
많은 경우 이러한 균질적인 리스트가 가장 흔하기도 하고 가장 중요한 경우다
- 인자 리스트를 생략 부호(...)로 끝내는 것이다
이는 ‘몇 개의 인자가 더 있을 수 있다’란 뜻이다
이렇게 하면 `<cstdarg>`에 있는 일부 매크로를 이용해서 거의 의의의 타입으로 이뤄진 임의의 개수의 인자를 처리할 수 있다
이 해결책은 본질적으로 타입 안전적이지 않으며, 복잡한 사용자 정의 타입과 함께 쓰이기 어려울 수 있다
하지만 이 메커니즘은 C의 초창기 시절부터 사용되어 왔다

여기서 설명할 방법은 세 번째 방법인 생략 부호(...)를 사용하는 방법이다

이 방법의 대표적인 예는 `<cstdio>`의 `printf(...)`이다

```
extern "C" inline int __cdecl printf(const char* _Format, ...);
```

이 방식을 사용하는 함수를 작성하는 예시로서 다음의 코드를 살펴보자

```
#include <cstdarg>
#include <iostream>

void error(int severity, ...) {
    va_list ap;
    va_start(ap, severity);
    std::cout << "Error: ";
    switch (severity) {
        case 0:
            break;
        case 1:
            std::cout << "Warning: ";
            break;
        case 2:
            std::cout << "Error: ";
            break;
        default:
            std::cout << "Unknown severity: ";
            break;
    }
    std::cout << va_arg(ap, const char*) << std::endl;
    va_end(ap);
}
```

- 기본 인자

```
void Test(int = 3, char c = 0);
```

5.2.1.3. 오버로딩 함수

함수 오버로딩은 함수의 이름 말고는 아무런 공통점 없이 덮어 쓰는 것을 말한다

이 개념은 함수 오버라이딩과 혷갈릴 수 있는데, 함수 오버라이딩은 파생 클래스가 기반 클래스의 가상 함수를 재정의하는 것을 말하는 것이다

- 자동 오버로딩 해결

함수가 호출되었을 때 오버로딩 된 함수 중 하나인 것이 판명된다면 다음의 과정을 통해 모든 오버로딩 함수 중 어느 것이 가장 적절한 것인지를 판단해줘야만 한다

1. 정확한 일치, 즉, 전혀 변환이 일어나지 않거나 사소한 변환²²만 사용하는 일치
2. 타입 승격을 사용한 일치
 - 통합 정수 타입 승격²³
 - float에서 double로
3. 표준 변환을 사용한 일치
 - int에서 double로, double에서 int로, double에서 long double로, Derived*에서 Base*로, T*에서 void*로, int에서 unsigned int로
4. 사용자 정의 타입 변환을 사용한 일치
 - 예를 들어 double에서 complex<double>로
5. 하수 선언에서 생략 부호 ...을 사용한 일치

일치가 발견된 가장 높은 수준에서 두 개 이상의 일치가 발견되면 해당 호출은 모호하다고 판단되어 컴파일 되지 않는다

- 오버로딩과 반환 타입

함수의 반환 타입은 오버로딩 해결에서 고려되지 않는다

- 오버로딩과 유효 범위

오버로딩은 오버로딩 집합의 멤버 사이에서 일어난다

이는 단일 유효 범위의 함수, 즉, 네임스페이스가 아닌 다른 유효 범위에서 선언된 함수는 오버로딩되지 않는다는 것이다

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

namespace Sample
{
    void f(double)
    {
        std::cout << "f(double)" << std::endl;
    }
    void Test()
    {
        f(1);          // calls f(double)
    }
}
```

즉, 위의 예제와 같은 경우에는 함수 오버로딩의 개념이 적용되는 것이 아니다

²² 예를 들어 배열 이름을 포인터로, 함수 이름을 함수를 가리키는 포인터로, T를 const T로

²³ bool에서 int로, char에서 int로, short에서 int로, signed 정수에서 unsigned 정수로

5.2.1.4. 함수 포인터

객체와 마찬가지로 함수 본체를 위해 생성된 코드는 메모리 어딘가에 위치하므로 그에 따른 주소를 가진다. 엄밀히는 프로세스에 할당된 메모리 덩이리 중 코드 영역에 존재한다. 객체를 가리키는 포인터가 존재할 수 있는 것처럼 함수를 가리키는 함수도 존재할 수 있다. 그러나 함수 포인터의 변경은 허용되지 않는다

```
void error(std::string msg);
void (*error_handler)(std::string) = error;

int main()
{
    error_handler("Hello, world!");
}
```

여기서 주목할 점은 함수 포인터에 함수의 주소를 대입할 때 함수에 주소 연산자를 붙이는 것은 선택 사항이며, 함수 포인터를 통해 함수를 호출할 때 역참조하는 것 또한 선택 사항이다

`reinterpret_cast<T>`을 이용하여 함수를 가리키는 포인터를 다른 함수를 가리키는 포인터로 변환할 수 있지만 그 결과는 예측 불능일 수 있다. 즉, Undefined Behavior이다

```
using P1 = int(*)(int*);
using P2 = void(*)(void);

void Test(P1 pf)
{
    P2 pf2 = reinterpret_cast<P2>(pf);
    pf2();                                // 심각한 문제가 발생할 가능성
                                            // 있음(Compiler
                                            // Implementation Defined)

    P1 pf1 = reinterpret_cast<P1>(pf2);
    int n{ 0 };
    int result{ pf1(&n) };                // OK
}
```

7) Any pointer to function can be converted to a pointer to a different function type. Calling the function through a pointer to a different function type is undefined, but converting such pointer back to pointer to the original function type yields the pointer to the original function.

`reinterpret_cast<T>`는 아무렇게나 사용하다가는 Undefined Behavior가 지뢰처럼 깔려있어 프로그램의 정확한 동작을 파악하기가 불가능해질 수 있다

함수 포인터는 그 자체로 호출하는 것 뿐만이 아니라 이를 함수의 매개변수로 넘기는 것 또한 가능하다

```

using CMP = int(const void*, const void*);

void Sort(CMP cmp);
int cmp1(const void* a, const void* b);

void Test()
{
    Sort(cmp1);

    CMP* fptr = cmp1;
    Sort(fptr);
    Sort(*fptr);
}

```

위의 예제 코드에서 ‘CMP’는 함수 포인터는 아니며, 함수 시그니처 그 자체를 가리키는 것이다

따라서 CMP를 통해 함수를 가리키게 하고자 한다면 CMP의 포인터를 통해 취해야만 한다 그러나 CMP의 포인터를 통해 함수 인자인 CMP에 넣을 때에는 역참조를 취하는 것은 선택 사항이다

5.2.2. 함수의 생애 주기에 따른 내부 동작

함수는 생애 주기를 거치며 주어진 명세와 사용법에 따라 어떨 땐 심지어 Undefined Behavior에 속하는 동작이 발생할 수도 있는 등 매우 다양한 내부 동작을 보인다 심지어 이러한 문제는 겉으로 보이는 동작만 표준에 따르는 것처럼한체, 실제 내부적으로는 컴파일러 제작사에서 의도한대로 표준과는 전혀 다르게 동작할 수도 있다는 점에서 더욱 이러한 복잡성을 가중시킨다

따라서 여기서는 가능한 표준에 근거한 함수의 동작을 다루며 컴파일러 의존적 및 플랫폼 의존적인 사항은 최대한 피하고자 노력하겠다

5.2.2.1. Compile Time

보통 컴파일이라고 하면 빌드 과정과 동의어로 생각하곤 하지만, 엄격한 기준을 적용한다면 컴파일은 빌드 과정의 일부에 불과하다

정석적인 빌드 과정을 축약해서 나타내자면 다음과 같다

1. Preprocessor
2. Compiler
3. Assembler
4. Linker

위의 과정은 정석적인 간략한 버전의 빌드 과정일 뿐, 실제로는 Optimizer를 포함한 다양한 추가 과정이 추가될 여지가 다분하다

함수는 Compile Time에 각 과정을 거치며 이 시간에 타입 추론, 함수 오버로딩 해결, 링킹, 최적화 등 다양한 처리를 겪게 된다

5.2.2.1.1. Compiler(MSVC, Visual C++)

- **inline** 함수 선별 및 처리
 - **inline** 함수 선별

MSVC의 경우 다음의 과정을 거쳐 inline 확장될 함수를 선별하게 된다

- a. 빌드 옵션 확인
- b. inline 확장이 요청된 함수 탐색
- c. 비용-편의 분석 수행

a. 빌드 옵션 확인

- 인라인 함수 확장 옵션
 - **inline** 확장 금지 요청
 - /Ob0
 - **inline** 확장 요청
 - /Ob{1/2/3}
- 최적화 옵션
 - **inline** 확장 요청
 - /O{1/2/x} : /Ob2 적용
 - **inline** 확장 금지 요청
 - /Od

b, inline 확장이 요청된 함수 탐색

- Specifier
 - **inline** 확장 요청
 - `inline`, `_inline`, `__inline`
 - `forceinline`, `__forceinline`
 - **inline** 확장 금지 요청
 - `__declspec(noinline)`
- Attribute
 - **inline** 확장 요청
 - `[[msvc::forceinline]]`
 - `[[msvc::forceinline_calls]]`
 - **inline** 확장 금지 요청
 - `[[msvc::noinline]]`
 - `[[msvc::noinline_calls]]`
- Pragma
 - **inline** 확장 요청
 - `#pragma auto_inline(on)`
 - `#pragma inline_recursion(on)`
 - **inline** 확장 금지 요청
 - `#pragma auto_inline(off)`
 - `#pragma inline_recursion(off)`

컴파일러는 다음과 같은 경우 함수에 inline 확장을 적용할 수 없다

- 함수 혹은 함수 호출자가 /Ob0으로 컴파일 되는 경우
- 함수 혹은 함수 호출자가 서로 다른 유형의 예외 처리를 사용하는 경우
- 함수가 Variable Argument Lists를 포함하는 경우
- /Ox, /O1 또는 /O2로 컴파일되지 않은 경우
- 함수가 재귀적임과 동시에 #pragma inline_recursion(on)이 설정되어 있지 않은 경우
- 함수가 가상 함수이며, 다형성이 적용될 경우
- 함수를 가리키는 함수 포인터를 통해 호출이 이루어질 경우
- naked, __declspec(noinline) Specifier가 함께 선언된 경우

컴파일러가 forceinline으로 선언된 함수를 inline 확장할 수 없으면 다음과 같은 경우를 제외하고 레벨 1 경고를 생성한다

- /Od 또는 /Ob0로 컴파일 되는 경우
- 라이브러리 혹은 다른 형태의 외부 소스에 정의되어 있는 경우

재귀 함수는 #pragma inline_depth에서 지정한 깊이²⁴까지만 inline 확장을 적용할 수 있으며, 그 이상으로는 재귀 함수 호출이 함수 인스턴스에 대한 호출로 처리된다
재귀 함수에 inline 확장을 적용하는지 그 자체의 여부는 #pragma inline_recursion을 통해 제어한다

여담으로 생성자는 암시적으로 inline 확장의 대상이다

c. 비용-편익 분석(Cost-Benefit Analysis) 수행

일반적으로 inline 확장은 다음의 성능 향상을 야기할 수 있다고 기대된다

- 함수 호출(스택에 객체의 주소를 배치하고 전달하는 매개 변수 포함)
- 호출자의 스택 프레임 보존
- 새 스택 프레임 설정
- 반환값 커뮤니케이션
- 이전 스택 프레임 복원
- Return
- Object File 크기 감소
- 가상 메모리 페이징 횟수 감소
- Instruction 감소
- Instruction Cache Hit Rate 증가

그러나 inline 확장은 남용할 경우 다음의 성능 하락을 야기할 수도 있다고 기대된다

- Object File 크기 증가
- 가상 메모리 페이징 횟수 증가
- Instruction Cache Hit Rate 감소

이러한 성능 향상/하락은 대개 어셈블리 수준에서 함수 본문에 대해 만들어지는 코드가 함수 호출문에 대해 만들어지는 코드보다 짧아지는 때를 기점으로 어느 한 쪽으로 기울게

²⁴ 최대 255

된다

비용-편익 분석은 정적 분석(Static Analysis)를 통해 위의 요소들을 종합적으로 고려하여 해당 `inline` 확장이 성능 향상을 야기할 것으로 기대될 때에만 해당 함수에 대해 `inline` 확장을 적용한다

`inline` 및 `_inline` Specifier는 컴파일러에게 함수가 호출되는 각 위치에 함수 본문의 복사본을 삽입하도록 요청한다. 여기서 핵심은 이는 그저 요청일 뿐이기에 컴파일러는 얼마든지 해당 요청을 무시할 수 있으며, `inline` 되는 함수는 함수 그 자체가 완전히 분해되는 게 아니라 함수 본문의 복사본을 사용하기에 함수 원본 그 자체는 그대로 유지된다는 것이다

MSVC의 경우 `inline` 확장은 비용-편익 분석(Cost-Benefit Analysis) 과정을 거쳐 해당 `inline` 확장 요청이 비용적 이익이 된다고 판단될 때에만 `inline` 요청을 받아들인다

물론, MSVC의 이러한 비용-편익 분석이 마음에 들지 않을 경우 `_forceinline` Specifier를 통해 비용-편익 분석 과정을 무시할 수 있다
그러나 비용-편익 분석 과정을 건너 뛴다고 하더라도 반드시 `inline` 확장이 적용된다고 보장할 수는 없다

그리고 MSVC에서 /clr 컴파일 옵션을 적용할 경우 그 어떤 함수도 인라인 될 수 없다
또한, 이전 버전과의 호환성을 위해 컴파일러 옵션 /Za(언어 확장 사용 안 함)를 지정하지 않는 한 `_inline` 및 `_forceinline`은 각각 `_inline` 및 `_forceinline`에 대한 동의어이다

- `inline` 확장 적용

`inline` 확장 적용이 확정되면 해당 함수가 재귀 함수인지 여부에 따라 조금 다르게 `inline` 확장이 적용된다

만약 `inline` 확장 적용이 확정된 함수가 재귀 함수가 아닐 경우 해당 함수의 본문의 복사본을 함수의 호출 위치에 삽입하게 된다

그러나 만약 `inline` 확장 적용이 확정된 함수가 재귀 함수일 경우에는 `#pragma inline_depth([n])`에 따라 특정 깊이까지만 `inline` 확장이 적용될 수 있으며, 그 이상으로는 일반적인 함수 호출이 적용된다

`#pragma inline_depth([n])`의 기본값은 16이며, n은 0 ~ 255 범위 내에서 설정될 수 있다
즉, 만약 `inline_depth`를 넘어서는 재귀 호출이 있을 경우 다음과 같이 `inline` 확장이 적용될 수 있다

```
inline int GetFac(const int n)
{
    return (n > 2) ? n * GetFac(n - 1) : n;
```

위의 코드와 같이 `GetFac(int)`가 있고, `GetFac(6)`을 호출했다고 할 경우 다음과 같이 처리될 가능성이 모두 존재한다

- 함수 내 재귀 깊이가 `inline_depth` 보다 작거나 같을 경우

이 경우에는 GetFac(6)의 함수 호출 대신 Compile Time에 계산된 720이 삽입될 가능성이 존재한다

- 함수 내 재귀 깊이가 inline_depth 보다 클 경우

이 경우에는 GetFac(6)의 함수 호출 대신 6 * GetFac(5)와 같이 Compile Time에 계산되는 결과가 어중간하게 끊긴 결과가 삽입될 가능성이 존재한다

- *constexpr* 함수 Compile Time 처리

MSVC의 경우 Visual Studio 2015 혹은 그 이상에서 *constexpr*을 지원한다
constexpr 함수는 다음의 규칙을 따른다

- Visual Studio 2015 이하

- *constexpr* 함수 혹은 생성자는 암시적으로 *inline*이다
- *constexpr* 함수는 인자와 반환값으로 Literal Type만을 허용한다
- *constexpr* 함수는 재귀 함수일 수 있다
- C++ 17 이하일 경우 *constexpr* 함수는 가상 함수일 수 없다. 그러나 C++ 20 이상부터는 가상 함수일 수 있다
- *constexpr* 함수의 본문은 '= default' 혹은 '= delete' 일 수 있다
- *constexpr* 함수의 본문은 goto 혹은 try를 포함할 수 있다
- non-*constexpr* Template의 명시적 특수화 인스턴스는 *constexpr*로서 선언될 수 있다
- *constexpr* Template의 명시적 특수화 인스턴스는 다시 *constexpr*로서 선언되지 않아도 *constexpr*이다

- Visual Studio 2017 이상

- *constexpr* 함수는 if 구문과 switch 구문이 포함될 수 있다
- *constexpr* 함수는 for, range-based for, while, 그리고 do-while 구문이 포함될 수 있다
- *constexpr* 함수는 지역 변수 선언을 포함할 수 있다. 그러나 지역 변수는 Literal Type이어야 하고, static 이거나 스레드 독립적일 수 없으며, 반드시 초기화되어야 한다. 지역 변수는 상수일 필요가 없으며 변경될 수 있다
- non-static *constexpr* 멤버 함수는 암시적으로 const일 필요는 없다

constexpr 함수가 위의 조건을 모두 만족할 경우 해당 *constexpr* 함수의 호출은 컴파일 타임에 계산될 수 있다

constexpr 함수는 암시적으로 *inline*이다. 만약 *inline* 되길 원치 않는다면 임의의 Specifier, Attribute, Pragma를 명시적으로 표기해줘야만 한다

다만, 만약 *constexpr* 함수가 재귀 함수의 형태를 띠고 있을 경우 상수 표현식 평가에 너무 많은 시간을 소비하지 않도록 평가 단계, 재귀 깊이, 역추적 깊이를 컴파일 옵션을 통해 제어할 수 있다

- /constexpr:depthN

depthN는 *constexpr* 함수의 재귀 깊이 한계를 결정한다
기본값은 512이다

- /constexpr:backtraceN

`backtraceN`은 상수 표현식 평가가 동시에 최대 N개 가능함을 결정한다
기본값은 10이다

- `/constexpr:stepsN`
`stepsN`은 N번째 평가 과정 후에 `constexpr` 평가를 종료한다
기본값은 100,000이다

- `reinterpret_cast<T>`에 따른 Type 변환 처리

`reinterpret_cast`에 의한 Pointer Type 간 변환은 반드시 Compile Time에 이루어지며 이는 표준에서 보장한다

Unlike `static_cast`, but like `const_cast`, the `reinterpret_cast` expression does not compile to any CPU instructions (except when converting between integers and pointers or on obscure architectures where pointer representation depends on its type). It is purely a compile-time directive which instructs the compiler to treat `expression` as if it had the type `new-type`.

즉, 함수 포인터간의 변환 또한 반드시 Compile Time에 이루어지는 처리이다

- 함수 오버로딩 해결

- **Argument Matching**

컴파일러는 함수 호출에 제공된 인자와 현재 범위의 함수 선언 중 가장 잘 일치하는 함수를 기준으로 오버로드된 함수를 선택한다

여기서 적합한 함수란 다음의 목록을 의미한다

1. 정확히 일치한다
2. 사소한 변환의 수행으로 일치한다
3. 통합 프로모션(Integral promotions)의 수행으로 일치한다
4. 표준 변환(Standard Conversions)의 수행으로 일치한다
5. 사용자 정의 변환(변환 연산자 혹은 생성자)의 수행으로 일치한다
6. 생략 부호(...)로 표시되는 인자가 존재한다

컴파일러는 범위에 따라 그리고 각 인자에 따라 후보 함수 집합을 구성하는데, 후보 함수는 해당 위치의 실제 인자 리스트를 인자 유형으로 변환할 수 있는 함수이다

이때, 함수의 조회는 인자 종속적인 이름 조회를 사용하여 정규화되지 않은 함수 호출의 정의를 찾아내게 된다. 이러한 인자 의존적인 이름 조회 방식을 Koenig 조회라고 한다
함수 호출의 모든 인자 유형은 네임스페이스, 클래스, 구조체, 유니온 혹은 Template의 계층 내에 정의된다

위에 기술된 인자 매칭 목록은 Koenig 조회를 통해 같은 범위 내에서 찾아진 후보 함수 집합에 적용되는 것이며, 1순위 적합 함수더라도 만약 범위가 함수 호출 시점의 범위 밖의 범위라면 미뤄질 가능성이 존재한다

만약 인자 적합을 시도한 결과 같은 순위 접합이 둘 이상 존재할 경우 해당 함수 호출은 모호한 함수 호출로 판단되어 컴파일 에러가 발생하게 된다

```
Fraction& Add(Fraction& f, long l); // Variant 1
```

```

Fraction& Add(long l, Fraction& f);           // Variant 2
Fraction& Add(Fraction& f1, Fraction& f2);    // Variant 3

void Test()
{
    Fraction f1, f2, f3;

    f1 = Add(f1, 5);    // Variant 1
    f2 = Add(5, f2);    // Error : Variant 1, 2 모호한 호출
    f3 = Add(f1, f2);    // Variant 3
}

```

위의 예제 코드와 같은 경우 모호한 호출에도 만약 생략 부호(...)를 포함하는 함수가 존재할 경우 와일드카드로서 동작하여 어떤 Type에도 적합하게 모호한 호출을 해결해줄 가능성이 존재한다

그러나 주의할 점은 생략 부호의 남용은 오히려 모호한 호출의 다발을 유발할 수 있다

- Argument Matching and Conversions

컴파일러가 함수 선언의 인자와 함수 호출의 실제 인자를 인치시키려 할 때 정확한 일치를 찾을 수 없는 경우 올바른 유형을 얻기 위해 표준 혹은 사용자 정의 변환을 적용할 수 있다
변환 적용에는 다음 규칙이 적용된다

- 둘 이상의 사용자 정의 변환을 포함하는 변환 시퀀스는 고려되지 않는다
- 중간 변환을 제거하여 단축할 수 있는 변환 시퀀스는 고려되지 않는다

만약 원하는 목표를 달성하는 변환 시퀀스가 존재하더라도 더욱 짧은 시퀀스가 존재할 경우 최적 일치는 아니다. 이러한 경우 컴파일러는 사소한 변환(Trivial Conversions)을 적용하여 가장 적합한 시퀀스를 선택하는 제 적용한다

Argument type	Converted type
<i>type-name</i>	<i>type-name&</i>
<i>type-name&</i>	<i>type-name</i>
<i>type-name[]</i>	<i>type-name*</i>
<i>type-name(argument-list)</i>	<i>(*type-name)(argument-list)</i>
<i>type-name</i>	<i>const type-name</i>
<i>type-name</i>	<i>volatile type-name</i>
<i>type-name*</i>	<i>const type-name*</i>
<i>type-name*</i>	<i>volatile type-name*</i>

변환이 시도되는 순서는 다음과 같다

1. 정확한 혹은 사소한 변환을 통한 일치

함수가 호출되는 유형과 함수 시그니처 유형간의 정확한 일치가 가장 좋은 경우이다

사소한 변환을 통해 일치하는 것 또한 정확한 일치로 분류되나, 사소한 변화를 수행하지 않는 것이 사소한 변화를 수행하는 것보다는 더욱 좋은 것으로 간주된다

다음은 사소한 변환의 예시이다

- *type-name** to *const type-name**
- *type-name** to *volatile type-name**
- *type-name&* to *const type-name&*
- *type-name&* to *volatile type&*

2. 프로모션의 수행을 통한 일치

float에서 double로의 변환과 통합 프로모션만 포함하는, 정확한 일치로 분류되지 않는 일치는 프로모션의 수행을 통한 일치로 분류된다

정확한 일치만큼 좋은 일치는 아니지만, 프로모션을 사용한 일치가 표준 변환의 수행을 통한 일치보다는 좋은 것으로 간주된다

3. 표준 변환의 수행을 통한 일치

표준 변환과 사소한 변환만 포함하는, 정확한 일치와 프로모션의 수행을 통한 일치로 분류되지 않은 일치는 표준 변환의 수행을 통한 일치로 분류된다

표준 변환의 수행을 통한 일치는 기반 클래스 포인터와 파생 클래스 포인터간의 변환 관계도 기술하지만 본 강의록의 범위를 넘어서기에 정리하지는 않겠다
추가적인 조사를 원한다면 다음의 링크를 참고하길 바란다

[https://learn.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170#arguments-matching-and-conversions-:text=using%20standard%20conversions.-_Match%20using%20standard%20conversions,-.%20Any%20sequence%20not] - MSDN / Match using standard conversions

5.2.2.1.2. Linker

- 함수 링킹

C++ 프로그램에서 Symbol(변수 혹은 함수 이름 등)은 범위 내에서 여러 번 선언될 수 있다.
그러나 정의는 단 하나만 존재할 수 있다. 이러한 규칙은 단일 이름 규칙(ODR, One Definition Rule)이라고 한다

선언은 이름을 나중에 정의와 연결할 수 있는 충분한 정보와 함께 프로그램에 삽입된다
정의는 이름을 소개하고 이름을 만드는 데 필요한 모든 정보를 제공한다

프로그램은 하나 이상의 번역 단위로 구성된다

번역 단위는 구현 파일과 직간접적으로 포함된 모든 헤더로 구성된다

일반적으로 구현 파일의 확장자는 '.cpp'와 '.cxx'이다

일반적으로 헤더 파일의 확장자는 '.h'와 '.hpp'이다

만약 함수를 선언하고 정의하는 데 있어 ODR 규칙을 위반할 경우 일반적으로 링커 오류가 발생하게 된다

또한, 같은 이름이 둘 이상의 번역 단위에 정의된 경우에도 링커 오류가 발생하게 된다

지유 함수(Free Function)은 전역 혹은 네임스페이스 범위에 정의된 함수이며, 쉽게 말하자면 클래스, 구조체 등에 속한 멤버 함수가 아닌 함수를 말한다

일반적으로 non-const 전역 자유 함수는 외부 링크를 가지고 있으며 프로그램의 모든 번역 단위에서 볼 수 있으며, 본래 ODR 규칙에 따라 정의가 하나만 존재한다면 선언은 여러 개 존재할 수 있지만, 이 경우에는 다른 전역 객체는 같은 이름을 가질 수 없다

내부 연결이 있거나 연결이 없는 기호는 선언된 번역 단위 내에서만 볼 수 있다. 이 경우에는 ODR 규칙에 따라 동일한 이름이 다른 번역 단위에 존재할 수 있다

전역 이름을 정적으로 명시적으로 선언하여 내부 연결을 강제로 설정할 수도 있으며, 이러한 경우 해당 이름은 정적으로 명시적으로 선언된 번역 단위로 표시가 제한된다

다음의 오브젝트들은 기본적으로 내부 연결을 가진다

- const 객체
- constexpr 객체
- typedef 객체
- 네임스페이스 안의 static 객체

그러나 만약 내부 연결 객체에 외부 연결로 설정하고 싶다면 **extern Specifier**를 사용할 수 있다

```
extern "C"
{
#include <stdio.h>
}

extern "C"
{
    char ShowChar(char c);
    char GetChar(void);
}

extern "C" char ShowChar(char c)
{
    putchar(c);
    return c;
}

extern "C" char GetChar(void)
{
    char c;
    c = getchar();
    return c;
}
```

```
}
```

```
extern "C" int errno;
```

5.2.2.2. Run Time

5.2.2.2.1. 함수 호출

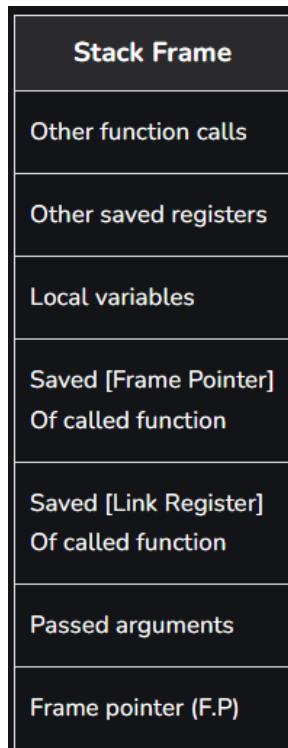
constexpr 함수, inline 함수, Macro 함수가 상수 표현식으로서 Compile Time에 처리되는 것을 제외하면 일반적으로 함수는 Run Time에 처리된다

이때의 일반적인 함수란 내부 연결과 외부 연결, 자유 함수와 비자유 함수, static과 non-static 등을 불문하고 Run Time에 실행되며, 이 경우 스택 프레임(Stack Frame)이라는 단위로서 Stack에 적재되게 된다

각 Stack Frame은 스택 포인터(SP)와 프레임 포인터(FP)를 유지하며, SP는 스택의 맨 위를 가리키고 FP는 프레임의 맨 위를 가리킨다. 또한 실행할 다음 Instruction을 가리키는 PC도 유지한다

함수 호출이 이루어질 때마다 스택 프레임은 Stack Segment에 생성된다²⁵

함수 호출에 의해 주어진 인자는 호출 함수의 스택 프레임 내에 메모리를 얻고 호출 함수의 스택 프레임 안으로 푸시된다²⁶



²⁵ CPU Architecture의 Memory Model이 Segmented Memory Model을 채택했을 경우

²⁶ 스택 프레임도 결국 Stack이니까

- 함수 매개변수 초기화

잘 알려지지 않은 사실이지만 함수의 매개변수가 둘 이상 있을 때 해당 매개변수들이 어떤 순서대로 초기화되는지는 표준에서 정의하지 않는다. 즉, 컴파일러 제작사에 따라 사양이 달라진다

두 개의 인자를 받는 함수가 있고 두 번째 인자가 첫 번째 인자에 의존적인 상황이 있다고 해보자

해당 함수의 호출은 컴파일 환경에 따라 어떤 환경에서는 정상적으로 동작하지만, 다른 환경에서는 의도한 생성자와 다른 생성자가 호출되거나 심각할 경우 예외가 터져나올 수도 있다

5.2.2.2.2. 함수 내 처리

- *Exception Handling*

Exception Handling이라는 주제는 함수의 설계 및 구현 그리고 실행 중의 관리에 있어서도 많은 관심을 기울여야 마땅할 주제이다

그러나 이번 강의록에서 함께 다루기에는 그 내용이 지나치게 방대하고, 표준에서 제공하는 정의와 거기에 더한 컴파일러 제작사들마다의 커스텀 버전이 다양하기에 추후에 별도의 회차에 통합적으로 다뤄보도록 하겠다

5.2.2.2.3. 함수 탈출(종료)

함수의 실행이 완료되어 함수를 탈출하게 되면 스택 프레임 안으로 푸시되었던 객체들(인자 밑 지역 변수)은 스택 프레임에서 팝업되며, 실행 스레드는 해당 함수를 호출했던 범위의 이전 실행 지점으로 돌아가 계속된다

- 반환값 커뮤니케이션

반환하고자 하는 값은 먼저 함수의 Return Type에 맞게 필요할 경우 적절한 변환 과정을 거쳐야만 한다

반환값을 받는 측, 즉, 함수를 호출했던 입장에서 보자면 반환값은 임시 객체의 R-Value이기에 이를 감안하고 적절한 처리를 해주어야만 한다

5.3. 매크로 함수

매크로란 Preprocessor 중에서도 '#define'을 통해 정의되며, #define 뒤에 오는 이름이 그 뒤에 오는 키워드로 대체된다

#define identifier replacement-list(optional)	(1)
#define identifier(parameters) replacement-list(optional)	(2)
#define identifier(parameters, ...) replacement-list(optional)	(3)
#define identifier(...) replacement-list(optional)	(4)
#undef identifier	(5)

```
#define NUM 10      // Preprocessor에 의해 10으로 치환된다
```

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

void Test()
{
    int num_0{ 10 }, num_1{ 20 };

    std::cout << MAX(num_0, num_1) << std::endl;
}
```

매크로 함수는 Preprocessor(#으로 시작하는 명령어)의 일종이므로 당연히 매크로 함수의 처리는 컴파일 타임, 그 중에서도 특히 Preprocessor(코드 번역기의 일종)에서 처리된다

#define의 이름은 관례적으로 모두 대문자에 띄어쓰기는 '_'로 표기하게 된다
즉, 만약 'Play Time'을 매크로로 정의하고자 할 때 관례를 따른다면 'PLAY_TIME'으로
표기하는 것이 정석이다
그러나 물론 이는 관례에 불과하기에 지키지 않아도 전혀 문제는 없다. 심지어 ISO/IEC에서
발행하는 C++ 표준 문서에서도 매크로 정의에 소문자를 사용하고 있다

매크로 함수는 성능상의 이득을 위해 사용하는 것이 아니다. 매크로 함수는 함수와 비슷하게
생겼을 뿐 실질적인 연산은 전혀 이루어지지 않기 때문이다
심지어 매크로 함수는 실수 혹은 굉장히 위험한 동작을 유발시킬 가능성을 내포한다

```
#define MUL(a, b) a * b

void Test()
{
    int num_0{ 10 }, num_1{ 20 };
    MUL(num_0 + 30, num_1 + 40);
```

```
    std::string str_0{ "Hello" }, str_1{ "World" };
    MUL(str_0, str_1);
}
```

위 예제에서 등장하는 `MUL` 사용 예시의 두 경우 모두 프로그래머가 생각하던 동작과는 굉장히 다르게 동작하게 된다
특히나 두 번째 사용 예제인 `MUL(str_0, str_1);`의 경우에는 컴파일 도중에 링커 에러가 발생할 수 있다²⁷. 좋은 컴파일러의 경우 컴파일 전 경고 혹은 에러를 띄워줄 수 있지만 그렇지 않은 컴파일러도 존재할 수 있다

²⁷ 'std::string::operator *(std::string)'가 존재하지 않을 경우

6. 익명 함수(람다)

6.1. 익명 함수(람다, 람다 표현식)

익명 함수, 람다 표현식(Lambda Expression)는 람다(Lambda)라는 이름으로 보다 자주 불리며, 특징으로는 함수와 비슷하게 동작하긴 하지만 이름이 존재하지 않는다

6.1.1. 익명 함수의 문법

람다는 operator()로 이름을 가진 클래스를 정의하고, 나중에 해당 클래스의 객체를 만들고, 최종적으로 그것을 호출하는 대신에 단축 방법을 사용하는 것이다

람다의 문법을 다루기 전에 간단한 예제를 통해 느낌을 보자면 다음과 같다

```
void PrintModule(const std::vector<int>& v, std::ostream& os, const int m)
{
    std::for_each(std::begin(v), std::end(v), [&os, &m](int x)
    {
        if (x % m == 0)
        {
            os << x << " ";
        }
    });
}

class ModulePrint
{
private:
    std::ostream& os;
    const int m;

public:
    ModulePrint(std::ostream& os, const int m) : os(os), m(m) {}

    void operator()(int x) const
    {
        if (x % m == 0)
        {
            os << x << " ";
        }
    }
};
```

따라서 위의 두 덩어리의 코드는 사실상 동일한 코드이다

이런 방식은 특히 연산을 인자로 알고리듬에 전달하고자 할 때 특히 유용하다. GUI의 맥락에서 이런 처리는 종종 콜백(Call Back)이라고 불리기도 한다

6.1.1.1. 구현 모델

[captures] (params) specs requires(optional) { body }	(1)	C++ 11
[captures] attr (params) specs requires(optional) { body }	(1)	C++ 23
[captures] { body }	(2)	C++ 11
[captures] attr specs { body }	(2)	C++ 23
[captures] <tparams> requires(optional) (params) specs requires(optional) { body }	(3)	C++ 20
[captures] <tparams> requires(optional) attr (params) specs requires(optional) { body }	(3)	C++ 23
[captures] <tparams> requires(optional) { body }	(4)	C++ 20
[captures] <tparams> requires(optional) attr specs { body }	(4)	C++ 23

- 1) Full form.
- 2) Omitted parameter list: function takes no arguments, as if the parameter list were `()`.
- 3) Same as (1), but specifies a generic lambda and explicitly provides a list of template parameters.
- 4) Same as (2), but specifies a generic lambda and explicitly provides a list of template parameters.

<i>captures</i>	- a comma-separated list of zero or more <i>captures</i> , optionally beginning with a <i>capture-default</i> . See below for the detailed description of captures.
<i>tparams</i>	- a non-empty comma-separated list of <i>template parameters</i> , used to provide names to the template parameters of a generic lambda (see <i>ClosureType::operator()</i> below).
<i>params</i>	- The list of parameters, as in <i>named functions</i> .
<i>specs</i>	- consists of <i>specifiers</i> , <i>exception</i> , <i>attr</i> and <i>trailing-return-type</i> in that order; each of these components is optional
<i>specifiers</i>	- Optional sequence of <i>specifiers</i> . If not provided, the objects captured by copy are <i>const</i> in the lambda body. The following specifiers are allowed at most once in each sequence: <ul style="list-style-type: none"> • mutable: allows <i>body</i> to modify the objects captured by copy, and to call their <i>non-const</i> member functions • constexpr: explicitly specifies that the function call operator or any given operator template specialization is a <i>constexpr</i> function. When this specifier is not present, the function call operator or any given operator template specialization will be <i>constexpr</i> anyway, if it happens to satisfy all <i>constexpr</i> function requirements (since C++17) • constexpr: specifies that the function call operator or any given operator template specialization is an <i>immediate function</i>. constexpr and constexpr cannot be used at the same time. (since C++20) • static: specifies that the function call operator or any given operator template specialization is a <i>static member function</i>. mutable and static cannot be used at the same time, and <i>captures</i> must be empty (since C++23) if static is present.
<i>exception</i>	- provides the <i>dynamic exception specification</i> or (until C++20) the <i>noexcept specifier</i> for <i>operator()</i> of the closure type
<i>attr</i>	- optional <i>attribute specifier sequence</i> . <p>An attribute specifier sequence applies to the type of the function call operator or operator template of the closure type. Any attribute so specified does not appertain to the function call operator or operator template itself, but its type. (For example, the <code>[[noreturn]]</code> attribute cannot be used.) (until C++23)</p> <p>If an attribute specifier sequence appears before the parameter list, lambda specifiers, or noexcept specifier (there must be one of them), it applies to the function call operator or operator template of the closure type (and thus (since C++23) the <code>[[noreturn]]</code> attribute can be used). Otherwise, it applies to the type of the function call operator or operator template.</p>
<i>trailing-return-type</i>	- <code>> ret</code> , where <i>ret</i> specifies the return type. If <i>trailing-return-type</i> is not present, the return type of the closure's <i>operator()</i> is deduced from return statements as if for a function whose return type is declared <i>auto</i> .
<i>requires</i>	- (since C++20) adds constraints to <i>operator()</i> of the closure type
<i>body</i>	- Function body

인자 전달, 반환 결과 및 본체 지정에 대한 세부 사항은 함수와 유사하다
 지역 객체를 ‘캡처’한다는 개념은 함수에서는 등장하지 않기에 다소 어색할 수 있는 개념이다.
 이는 실제로는 해당 지역 객체가 람다 범위 밖에 있더라도 마치 람다 범위 안에 선언된 지역 객체처럼 사용하겠다는 것을 의미한다

본격적인 람다의 구성 부분을 살펴보자면 다음과 같다

캡처 리스트는 []로 구분되며, 정의 환경에 있는 어떤 이름이 람다 표현식에 사용될 수 있는지를 지정한다. 쉽게 람다 클로저 객체에 캡처되는 객체들을 초기화하는 생성자라고 생각하면 된다
이때, 이름들이 복사되는지, 참조자에 의해 접근되는지까지 지정할 수 있다

캡처된 이름들은 마치 람다 내부에 이미 존재하던 지역 객체처럼 사용할 수 있게 된다

매개변수 리스트는 ()로 구성되고 void일 수 있으며, 함수의 매개변수 리스트와 같이 람다에 어떤 매개변수가 필요한지를 지정한다

mutable Specifier는 클로저 객체에 저장된 객체의 값에 변경을 해야만 할 때 선택적으로 지정할 수 있다

람다의 대안인 operator()로 말하자면 본래 mutable이 붙지 않은 상태로는 operator() const인 상태이다. 그러나 mutable이 붙게 되면 operator()인 상태가 되는 것이다

런타임에 람다로부터 생성되는 클래스는 객체는 클로저 객체로서 불리는데, 이는 메모리에 존재하게 되며 캡처한 값은 클로저 객체 내부에 저장되게 된다. 이때 캡처한 값은 마치 클래스의 멤버 변수와 같은 입장이고 람다 본체는 메서드와 같은 입장을 취하게 된다고 생각하면 되는데, mutable을 사용하는 순간 이 멤버 변수의 값을 변경할 수 있는 권한을 얻게 되는 것이다

```
void FillVector(const std::vector<int>& vector)
{
    int count{ vector.size() };
    std::generate(vector.begin(), vector.end(), [count]() mutable
    {
        return --count;
    });
}
```

선택적인 noexcept Specifier

'->'를 통한 선택적인 명시적인 반환 타입 지정

기본적으로 람다는 어떤 타입의 값을 반환할지 알 수 없다. 다만 이는 ->를 통해 명시적으로 지정해줄 수 있다

```
void FillVector(const std::vector<int>& vector)
{
    int count{ vector.size() };
    std::generate(vector.begin(), vector.end(), [count]() mutable ->
int
    {
        return --count;
    });
}
```

본체는 {}를 통해 실행될 코드를 지정한다

6.1.1.2. 캡처

캡처의 종류는 다음과 같다

□	빈 캡처 리스트 스택 프레임의 지역 객체가 람다 본체에서 사용될 수 없다는 의미이다
[&]	참조에 의한 암시적 캡처 스택 프레임의 모든 지역 객체에 대한 참조자에 의한 접근이 허용된다
[=]	값에의한 암시적 캡처 람다의 호출 시점에 스택 프레임에 존재한 지역 객체들의 복사본에 대한 참조를 허용한다
[캡처 리스트]	명시적인 캡처 캡처 리스트는 참조 혹은 값에 의해 캡처될 지역 객체의 이름으로 이뤄진 리스트이다 캡처 리스트는 this와 ...를 포함할 수 있다
[&, 캡처 리스트]	배타적인 참조에 의한 암시적 캡처 리스트에 언급되지 않은 스택 프레임의 모든 지역 객체에 대한 참조에 의한 접근이 허용된다 캡처 리스트는 this를 포함할 수 있으며, &가 붙을 수 없다 캡처 리스트에 명시된 이름들은 값에 의해 참조된다
[=, 캡처 리스트]	배타적인 값에 의한 암시적 캡처 리스트에 언급되지 않은 람다의 호출 시점에 스택 프레임에 존재한 지역 객체들의 복사본에 대한 참조를 허용한다 캡처 리스트는 this를 포함할 수 없으며, 리스트에 포함되는 이름들은 &가 붙어야만 한다 캡처 리스트에 명시된 이름들은 참조자에 의한 접근이 허용된다

- 람다의 수명

람다는 호출자보다 오래 살아남을 수 있으며, 이는 람다를 다른 스레드에 전달하거나
호출자가 람다를 차후에 사용하기 위해 저장해둘 경우 이런 현상이 발생할 수 있다

```
void Setup(Menu& m)
{
    Point p1, p2, p3;
    m.Add([&] { m.Draw(p1, p2, p3); });
}
```

위는 Menu의 내부적인 자료구조나 매커니즘은 모르겠지만 Menu 인스턴스 참조자에 추후에
사용될 m.Draw(p1, p2, p3)를 담아놓는 과정을 기술하고 있다

여기서 주목할 점은 p1, p2, p3가 Setup의 스택 프레임에 종속되는 지역 객체라는 점인데, 만약 Setup 스택 프레임이 회수된 후 Menu 인스턴스에 저장된 m.Draw(p1, p2, p3)가 호출될 경우 이미 사라진 지역 객체들에 대한 접근으로 인해 System.NullReferenceException과 같은 예외가 터져나올 가능성이 높다

이렇듯 람다가 호출자보다 오래 살아남을 가능성이 있는 경우에는 캡처할 지역 정보가 있을 경우 복사로서 클로저 객체 내부에 저장되고, 값들은 return 메커니즘 혹은 적절한 인자를 통해 반환하도록 하는 것이 발생할 수 있는 예외를 예방하는 하나의 방법이 될 수 있다

이 방법을 통해 위의 문제가 발생할 가능성이 있는 코드를 수정하는 것은 어렵지 않다

```
void Setup(Menu& m)
{
    Point p1, p2, p3;
    m.Add([=] { m.Draw(p1, p2, p3); });
}
```

- 네임스페이스 이름

```
template <typename _U, typename _V>
std::ostream& operator<<(std::ostream& os, const std::pair<_U, _V>& p)
{
    return os << "(" << p.first << ", " << p.second << ")";
}

void PrintAll(const std::map<std::string, int> m)
{
    std::for_each(m.begin(), m.end(), [](const std::pair<std::string, int>& p)
    {
        std::cout << p << std::endl;
    });
}
```

네임스페이스 변수, 즉, 해당 람다가 정의된 범위에 혹은 그 바깥에 존재하는 것들에 대해서는 항상 접근 가능하기에 그것들은 캡처할 필요 없다

위의 예제의 경우 cout와 pair에 대한 연산자에 대해 캡처하지 않아도 된다

- 람다와 this

클래스의 멤버 함수에 쓰인 람다에서 멤버에 접근하기 위해서는 캡처 리스트에 this를 사용하면 된다

이때 this는 복사에 의한 캡처만을 지원하는데, 그렇다고 클래스 인스턴스의 복사본을 가지는 것은 아니다. this는 인스턴스 그 자체가 아닌 인스턴스 포인터라는 점을 상기하면 this를 캡처함으로서 접근하는 멤버는 실제 원본이라는 것을 알 수 있다

6.1.1.3. 호출과 반환

람다에 인자를 전달하는 규칙은 함수에 대한 것과 동일하며, 결과 반환 규칙도 마찬가지다. 사실상 캡처 규칙을 제외하고는 람다에 대한 대부분의 규칙은 함수와 클래스로부터 빌려온 것이다. 하지만 두 가지 변칙적인 사항에 유의해야 한다

람다 표현식이 아무 인자도 받아들이지 않는 경우에는 인자 리스트가 생략될 수 있다
따라서 가장 짧은 람다 표현식은 '`[]`'이다

람다 표현식의 반환 타입은 그것의 본체로부터 추론될 수 있다

람다 본체에 `return`이 포함되어 있지 않다면 람다의 반환 타입은 `void`이다

람다 본체가 단 하나의 `return`으로 구성되어 있다면 람다의 반환 타입은 `return` 표현식의 타입이 된다

그 어느 쪽에도 해당되지 않는다면 명시적으로 반환 타입을 지정해야만 한다

- 람다를 함수 포인터처럼 사용

람다를 함수 포인터처럼 사용하고 싶다는 것은 흔하게 두 가지 의미일 것이다

람다에 이름을 붙여 보관하고 싶은 목적

람다를 함수의 매개변수로 넘기고 싶은 목적

```
class Sample
{
public:
    Sample() {}
    void operator()()
    {
        std::cout << "Sample" << std::endl;
    }
};
```

Sample 클래스가 위와 같이 생겼다고 가정하고 아래의 예시들을 보길 바란다

람다에 이름을 붙인다는 것은 생각보다 쉬운데 다음과 같이 `auto`를 통해 람다를 붙잡아놓으면 된다

```
void Test()
{
    Sample sample;

    auto fn = [&sample]() mutable {sample(); };
    fn();
}
```

그렇다면 람다를 함수의 매개변수로 넘길 때는 어떻게 하면 좋을지 알아보자면 다음과 같다
먼저 람다를 매개변수로 받는 측에서는 다음과 같이 하면 된다

```
template <typename _Fn>
void Func(_Fn fn)
{
    fn();
}
```

람다를 매개변수로 보내는 측은 다음의 여러 방식들 중 하나를 선택하면 된다

```
void Test()
{
    Sample sample;

    Func([&sample]() mutable {sample(); });

    auto fn = [&sample]() mutable {sample(); };
    Func(fn);

    Func(Sample{});
}
```

람다를 받는 측에서 굳이 템플릿을 쓰지 않고 그냥 람다의 시그니처 그대로를 쓰면 되지
않느냐할 수 있다. 그러나 위의 경우 람다의 시그니처를 굳이 기술해보자면 *lambda []()*
*mutable ->void*인데, C++은 lambda라는 키워드를 지원하지도 않기에 애초에 람다의 완벽한
시그니처를 기술할 수 없으며, lambda라는 키워드를 제외한 시그니처를 기입할 경우
완전하지 않은 형식으로서 컴파일되지 않는다
따라서 람다를 함수의 매개변수로서 받고 싶다면 템플릿을 사용하는 것이 좋으며, 이는 Visual
C++ STL에서도 다음의 예제와 같이 사용하는 방법이다

```
_EXPORT_STD template <class _InIt, class _Fn>
_CONSTEXPR20 _Fn for_each(_InIt _First, _InIt _Last, _Fn _Func) {
// perform function for each element [_First, _Last)
    _Adl_verify_range(_First, _Last);
    auto _UFirst      = _Get_unwrapped(_First);
    const auto _ULast = _Get_unwrapped(_Last);
    for (; _UFirst != _ULast; ++_UFirst) {
        _Func(*_UFirst);
    }

    return _Func;
}
```

6.1.1.4. 람다의 타입

람다는 그때그때 상황마다 최적화된 버전을 제공하기 위해 정확한 타입을 정의하지 않는다. 이러한 타입을 Closure Type이라고 하는데, 이는 모든 람다마다 고유한 것으로 어떤 두 람다가 동일한 타입을 가지는 것은 불가능하다. 두 개 이상의 람다가 동일한 타입을 가진다면 템플릿 인스턴스화 메커니즘이 혼동을 일으킬 수 있다

람다는 생성자와 const 멤버 함수 operator() ()를 가진 지역 클래스 타입이다. 람다를 인자로 활용하는 것 이외에도 auto나 std::function<R(AL)>로 선언된 변수를 초기화할 수 있는데, 여기서 R이 람다의 반환 타입이고 AL은 인자 리스트의 타입들을 나타낸다

auto와 std::function<R(AL)>을 사용하는 타이밍의 차이점은 람다의 타입 추론이 언제 일어나는가에 따라 달라지게 되는데, 쉽게 말해 재귀가 람다에 포함되어있느냐 그렇지 않느냐의 차이다

```
void Test()
{
    auto rev = [&rev](char* b, char* e)
    {
        if (1 < e - b)
        {
            std::swap(*b, *--e);
            rev(++b, e);
        }
    };
}
```

즉, 위의 경우에 람다를 이용해 auto rev를 초기화하려는 것은 불가능하다. 왜냐하면 람다는 그 특성상 그때마다의 상황에 따라 타입을 유동적으로 추론하려 하는데 람다가 내부적으로 재귀의 형태를 띄고 있어 정확한 추론을 이끌어낼 수 없기 때문이다(예를 들어 자기 자신 재귀 뿐만 아니라 여러 함수가 환형으로 재귀를 도는 등 재귀는 너무나 다양한 상황이 있을 수 있다는 점을 보면 명확하다)

따라서 위와 같이 람다가 재귀를 품고 있을 경우에는 std::function<R(AL)>을 통해 잡아두는 것이 합리적이다

```
void Test()
{
    std::function<void(char* b, char* e)> rev = [&rev](char* b, char*
e)
    {
        if (1 < e - b)
        {
            std::swap(*b, *--e);
            rev(++b, e);
        }
    };
}
```

```
    };
}
```

이렇게 하면 `std::function<R(AL)>`에 의해 람다의 모든 타입이 정의되었기에 재귀를 포함하더라도 잡아둘 수 있다

그러나 만약 람다가 아무것도 캡처하지 않는다면 적절한 타입의 함수 포인터를 통해 잡아둘 수 있다

```
void Test()
{
    double (*fptr_0)(double) = [](double num)
    {
        return std::sqrt(num);
    };

    // Error
    /*double (*fptr_1)(double) = [&](double num)
    {
        return std::sqrt(num);
    };*/
}

// Error
/*double (*fptr_2)(int) = [](double num)
{
    return std::sqrt(num);
};*/
}
```

7. 클래스 - 기초

7.1. ADT의 개념

추상 데이터형(Abstract Data Type)은 데이터와 데이터를 처리하는 연산의 집합이다
연산은 프로그램의 나머지 부분에 데이터가 무엇인가를 설명해주는 역할과 나머지
프로그램에서 그 데이터를 변경할 수 있게 해주는 역할을 한다

'추상 데이터형'에서 '데이터'는 다양한 의미로 사용된다. 추상 데이터형은 사용되는 모든
연산이 포함된 그래픽 윈도우일 수도 있고, 파일과 파일 연산일 수도 있다

만약 ADT를 제대로 이해하지 못한채로 작성하는 클래스는 그저 실제로는 연관성이 높지
않은 데이터와 루틴을 편의를 위해 보관하는 상자와 다름없이 작성되고 다뤄질 것이기
때문이다

7.1.1. ADT가 필요한 예

우선 ADT가 유용하게 쓰이는 예를 살펴보면 다음과 같다

여러 가지 글꼴과 크기, 폰트 특성(굵은 글씨나 기울임꼴 등)을 사용해 화면에 출력할
텍스트를 제어하는 프로그램을 작성한다고 가정하자

프로그램의 일부는 텍스트의 폰트를 처리한다

위와 같은 상황 가정에서 만약 ADT를 사용하지 않는다면 폰트를 제어하기 위해
주먹구구식의 방식을 취해야만 할 것이다. 예를 들어 폰트 크기를 12포인트(16 픽셀 크기)로
바꿔야 한다면 다음과 같은 코드를 작성해야만 할 것이다

```
curFont.m_size = 16;
```

라이브러리 루틴을 미리 구축해 놓았다면 위의 코드는 아마 다음과 같이 조금 더 읽기 편한
코드로 작성할 수도 있다

```
curFont.m_size = PointToPixels(16);
```

또한 만약 폰트를 굵게 설정할 때는 논리 연산자와 16진수 상수 0x02를 사용하는 다음과 같은
코드를 작성할 수 있다

```
curFont.m_attr |= 0x02;
```

위의 코드를 조금 더 깔끔하게 작성할 수는 있겠지만, 그나마 가장 깔끔한 코드가 아래
정도이다

```
curFont.m_attr |= FontAttr.Bold;
```

```
// or  
  
curFont.m_bold = true;
```

위와 같은 방식으로 데이터를 제어할 경우 유사한 코드가 프로그램의 여러 곳에 뿌려지게 되어 사소한 변경사항에도 큰 영향을 받게되기 마련이다

7.1.2. ADT를 사용할 때 좋은 점

- 구현 세부 사항을 감출 수 있다

폰트 데이터 유형에 대한 구체적인 정보를 숨기면 데이터 유형이 바뀌어도 프로그램 전체에 아무런 영향을 미치지 않고 혹은 적은 영향을 미치며 어느 한 곳에서 변경할 수 있다

예를 들어 ADT를 통해 구현 세부 사항을 감추지 않았다면 폰트의 굵기를 변경할 때 한 곳이 아니라 이전 값으로 설정된 모든 곳에서 코드를 변경해야만 한다. 또한 세부 구현을 감추면 해당 정보를 메모리가 아닌 외부 저장소에 저장할 때나 폰트 처리 루틴을 다른 프로그래밍 언어로 구현하고자 할 때도 다른 부분을 변경할 필요가 없다

- 변경이 전체에 영향을 미치지 않는다

폰트를 추가하고 더 많은 속성(작은 대문자, 취소선 등)을 지원하고자 할 때도 한 곳에서 프로그램을 변경할 수 있다

당연히 프로그램의 나머지 부분에는 영향을 미치지 않을 것이다

- 성능을 향상시키기 쉽다

폰트의 성능을 향상시켜야만 할 때 전체 프로그램을 건드리기보다는 잘 정의된 루틴 몇몇만 다시 작성하면 된다

- 프로그램이 명백해진다

curFont.m_attribute |= FontAttribute.Bold와 같은 코드로 직접 데이터를 다루기보단 curFont.SetBold(bool)와 같은 함수를 호출하는 편이 보다 안전할 수 있다

프로그램 전체에서 데이터를 직접 다루도록 둔다는 것은 잘못된 클래스 혹은 구조체 식별자나 필드 이름, 잘못된 연산(or 대신 and), 잘못된 속성값(0x02 대신 0x20) 등 다양한 실수가 있을 수 있다

그러나 curFont.SetBold(bool)에서 틀릴 수 있는 부분은 변수 이름이나 호출할 루틴의 이름을 틀리는 경우 정도 뿐이며, 이 정도는 여러 개발 도구를 통해 쉽게 바로잡을 수 있는 부분이다

- 프로그램의 가독성이 높아진다

curFont.m_attribute |= FontAttribute.Bold보단 curFont.SetBold(bool)가 읽기도 쉽고 무슨 처리가 일어나는지 파악하기도 쉽다

- 전체 프로그램에 데이터를 넘길 필요가 없다

curFont.m_size = 16이나 curFont.m_attribute |= FontAttribute.Bold의 경우 해당 연산을 수행하는 모든 루틴에 해당 값이 전달되어 읽기/쓰기 권한을 모두 부여하는 것이라고 볼 수 있으며, 개발 과정에서 실수가 언젠가는 반드시 발생할 수 있다는 점을 고려할 때 매우 좋은 선택이라고 볼 수는 없다

ADT는 이러한 점에서 ADT를 설계하는 측과 ADT를 사용하는 측 모두에게 일종을 필터처럼 작동하여 이점을 줄 수 있다

ADT를 설계한다는 것은 이제와서 새삼스러울 수도 있지만 하나의 새로운 타입을 고안하는 것이다. 그러나 이 사실을 명확히 인지하는 것이 매우 중요한데, 설계되어 구현된 ADT 자체는 완전한 하나의 타입으로서 사용하는 측에게 제공되어 사용하는 측에서는 해당 ADT를 통해 설계자가 딱히 막지 않은 것이라면 그 무엇을 해도 아무도 말리지 못하는 자유를 가지게 되기 때문이다. 그러나 만약 그렇게 방지한다면 거의 틀림없이 의도와 달리 동작할 것이다 때문에 설계자는 타입의 대부분을 사용자가 볼 수 조차 없게 설정해두고 일부의 필수적이고 안전한 동작을 제공하며, 심지어는 제공한 동작에 사용자가 넣은 매개변수 조차 내부적인 필터링 처리를 통해 적법한 범위에 있는 값만을 선별적으로 받고 그렇지 않으면 Assert나 예외를 던지게 해둠으로서 사용자측에 강력한 제약과 경고를 날릴 수 있다

사용자가 사용하는 타입이 만약 내부의 대부분이 외부에 공개된 설계를 가졌다면 어떨까 사용자 입장에서는 위와 같은 타입만큼 쓰기 까다로운 타입도 그리 없을 것이다. 만약 `FontInfo.m_fontSize`와 `FontInfo.SetFontSize(int)`가 동시에 사용자에게 보일 때 둘 중 어떤 것을 사용하는 것이 옳은가? `FontInfo.m_fontSize`에 곧바로 값을 대입해도 타입을 올바른 동작을 보일 것인가? 아니면 `FontInfo.SetFontSize(int)` 안에서 추가적인 타입 내의 자료 구조 등을 변경하는 처리가 연쇄적으로 일어날 수도 있기에 `FontInfo.SetFontSize(int)`를 사용해야만 하는가?

위와 같은 이지선다는 그냥 대강 루틴을 선택하는 편이 보다 안전해보인다는 점에서 그나마 쉬운 선택일 수 있지만 만약 해당 타입이 프레임워크와 같이 반드시 정해진 순서의 루틴이 연속적으로 호출되어야만 하는 설계라면 어떨 것인가? 이를 사용자 입장에서 코드를 작성하는 그 순간에 손쉽게 알아챌 수 있는가?

사용자 입장에서도 타입은 설계자가 타입의 대부분을 감춰둔채 적절한 제약과 경고를 주는 편이 사용하기 편이 사용하기 편하다

7.2. 클래스의 개념(ADT + 상속 + 다형성)

ADT는 다시 말하자면 연관성 있는 데이터를 모아두고, 해당 데이터에 접근하는 방식을 정의하고, 데이터에 관련된 연산을 모아두는 것이다
C++에서의 클래스는 ADT의 개념 위에 상속과 다형성 개념을 더한 것이라 보아도 될 것이다

7.3. 클래스, 인스턴스, 객체의 개념

- 클래스, 인스턴스, 객체

위의 세 단어는 비슷한 문맥에서 곧잘 함께 사용되어 혼동하기 쉬운 개념들이지만 조금만 보면 쉽게 구분할 수 있다

먼저 클래스는 비유하자면 설계도와 같다고 할 수 있다

클래스 그 자체는 코드로서 런타임에는 바이너리 코드의 형태로 메모리의 코드 영역에 존재하며, 클래스 본인만으로서는 그 무엇도 할 수 없다. 대신 클래스는 앞으로 만들어질 실체가 있어 데이터를 주고 여러 동작을 취할 수 있는 인스턴스가 어떻게 생겼고 무엇을 할 수 있는지를 기술하는 것이다

이는 마치 설계도가 존재는 하지만 그 자체로서 물건이나 건물은 되지 못하지만, 완성되면 어떤 생김새에 어떤 용도로 사용될 수 있는지를 기술하는 것과 같다

인스턴스는 설계도를 바탕으로 실제 만들어져 세상에 나온 제품과 같다고 할 수 있다

인스턴스는 클래스로 정의된 타입의 설계가 메모리에 올라가게 되어 실제로 데이터를 주고 이런저런 동작을 취하며 프로그램 내에서 영향력을 미칠 수 있는 실체를 가지게 된 것이라고 할 수 있다

객체는 제품 전체를 싸잡아 부르는 것이라 말할 수 있다

즉, 객체는 일종의 인스턴스의 복수형과 같은 것이라 생각하면 편하다
그러나 때로는 그냥 객체라고 하는 말이 인스턴스를 가리키기도 하기에 문맥에 따라 잘 해석해야만 한다

7.4. 클래스의 문법

클래스는 위낙 방대한 개념이기도 하고 버전마다 컴파일러 제조사마다 지원하는 범위가 다르기도 하고, 파고들고자 하기만 한다면 미묘한 부분이 지나치게 많기에 이 문서에선 클래스로 일반적인 개발을 할 때 신경써야만 할 점도만 다루겠다

7.4.1. 클래스

C++에서의 클래스를 아주 간단히 요약하면 다음과 같다

- 클래스는 사용자 정의 타입이다
- 클래스는 멤버의 집합으로 구성된다. 가장 흔한 종류의 멤버는 데이터 멤버와 멤버 함수다
- 멤버 함수는 초기화(생성), 복사, 이동 및 마무리의 의미를 정의할 수 있다
- 객체에 대해서는 .(점)과 포인터에 대해서는 -(화살표)를 이용하여 멤버에 접근할 수 있다
- +, !, [] 등의 연산자는 어떤 클래스에 대해 정의될 수 있다
- 클래스는 멤버가 포함된 네임스페이스다
- public 멤버는 클래스의 인터페이스를 제공하고, private 멤버는 구현 세부 사항을 제공한다
- struct는 public 멤버가 기본인 클래스다

7.4.1.1. 멤버 함수

멤버 함수를 설명하기 위해 Date 표시와 이 타입의 변수를 조작하기 위한 함수를 먼저 C 스타일로 작성하면 다음과 같다

```
struct Date
{
    int m_d, m_m, m_y;
};

void initDate(Date& d, int, int, int);
void AddDay(Date& d, int);
void AddMonth(Date& d, int);
void AddYear(Date& d, int);
```

위의 코드는 데이터 타입 Date와 함수들 사이에 어떤 명시적 연결 관계도 없다. 이들이 관계가 있다는 점은 주의깊게 살펴보아야만 의식적으로 알 수 있을 뿐이다. 그러나 만약 다음의 코드와 같이 이들을 멤버로서 선언하면 손쉽게 명시적 연결 관계를 구축하여 보일 수 있게 된다

```
struct Date
{
    int m_d, m_m, m_y;
```

```

    void initDate(int, int, int);
    void AddDay(int);
    void AddMonth(int);
    void AddYear(int);
};


```

클래스²⁸ 정의 안에서 선언된 함수를 멤버 함수라고 하는데, 이런 함수는 구조체 멤버 접근을 위한 표준 문법²⁹을 이용해서 호출할 수 있다

```

void Test()
{
    Date today;
    today.initDate(2023, 10, 10);
    today.AddDay(7);
}

```

다른 구조체도 동일한 이름을 가진 멤버 함수를 가질 수 있기에 멤버 함수를 정의할 때는 구조체 이름을 지정해야 한다

```

void Date::initDate(int year, int month, int day)
{
    m_y = year;
    m_m = month;
    m_d = day;
}

```

멤버 함수 내에서는 객체를 명시적으로 가리키지 않고 멤버 이름을 사용할 수 있다. 이런 경우 이름은 함수가 호출된 대상 객체의 해당 멤버를 가리킨다
예를 들어 Date::InitDate(...)가 today에 대해 호출되면 m_y = y는 today.m_y에 대해 대입된다

즉, 클래스 멤버 함수는 자신이 어떤 객체에 대해 호출되었는지 알고 있다. 하지만 static 멤버의 경우 특정 객체에 독립적으로 존재하기에 차이에 주의해야만 한다

7.4.1.2. 기본 복사

기본적으로 객체는 복사될 수 있다. 특히 클래스 객체는 그것의 클래스 객체의 사본으로 초기화될 수 있다

```

void Test()
{
    Date today;
    Date d0 = today;
    Date d1{ today };
}

```

²⁸ struct도 class의 일종이다

²⁹ .(점) or ->(화살표)

```
}
```

클래스의 복사는 기본적으로 클래스 객체 멤버들의 사본을 생성하고, 이를 복사하는 것이다. 만약 이러한 기본 동작이 원하는 동작이 아니라면 복사 생성자, 복사 대입 연산자를 재정의하여 동작을 바꿀 수 있다

7.4.1.3. 접근 제어

앞선 예제에서는 구조체를 사용했기에 모든 것이 외부에 공개되며, 외부에서의 접근을 제한할 수 있는 도구가 없다. 만약 외부로의 공개 정도를 제어하고 싶다면 클래스를 사용해야만 한다

```
class Date
{
    int m_d, m_m, m_y;

public:
    void initDate(int, int, int);
    void AddDay(int);
    void AddMonth(int);
    void AddYear(int);
};
```

위의 예제에서 public 레이블은 클래스 본체를 두 부분으로 분리해준다

첫 번째 부분인 private인 부분³⁰에 선언된 이름들은 멤버에 의해서만 사용될 수 있다

두 번째 부분인 public 부분에 선언된 이름들은 클래스의 인터페이스를 구성하며, 외부에서 볼 수 있는 부분을 정의한다

struct는 멤버가 기본적으로 public이며 접근 제어가 따로 불가능한 class의 일종이다. 따라서 멤버 함수는 앞에서와 똑같이 정의될 수 있다
하지만 멤버가 아닌 함수는 비공개 멤버의 사용이 금지된다

```
void DoSomething(Date& date)
{
    //date.m_m = 10; // Error
}
```

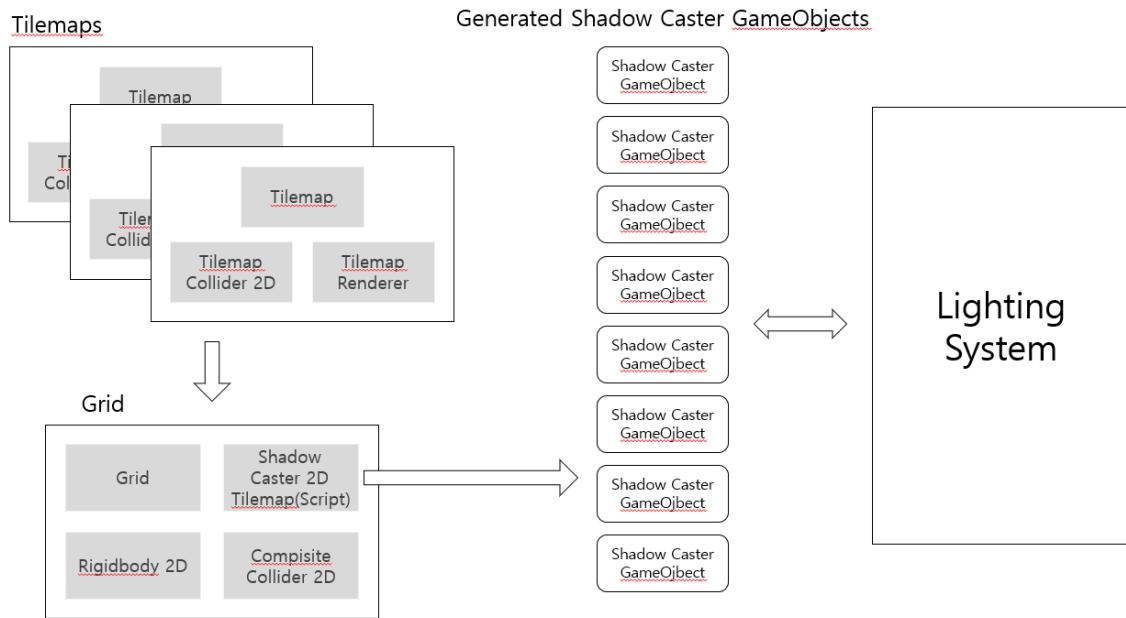
이러한 접근 제어는 앞서 ADT에서 설명한 다양한 이익을 얻기 위한 수단으로서 매우 중요한 개념이다

다만, 이러한 접근 제어를 설정하더라도 주소 조작, 명시적 타입 변환, 리플렉션(C#의 경우), 상속 등의 방식을 통해 이를 우회하여 얹지도 private 멤버에 접근할 수는 있다. 그러나 대부분의 경우 이는 꼼수이며, 일반적으로는 여전히 유효하다

³⁰ class의 모든 멤버는 기본적으로 private이다

물론, 이러한 접근 제어를 우회하여 억지로 멤버에 접근해야만 원하는 처리에 달성할 수 있을 때도 있기에 방법을 알아두는 것도 유용하다
예를 들어 다음의 경우는 Unity/C#에서의 예이지만 예로 들기에 유용할 것 같아 첨부한다

이 예시는 2D 게임 환경에서 실시간으로 Tilemap에 Tile이 추가됨에 따라 2D Light에 따른 Shadow Casting을 해주기 위한 코드이다. 아래 코드에서는 본래 접근이 불가능한 ShadowCaster2D의 private 멤버를 리플렉션을 활용하여 강제로 접근하여 조작하고 있다



[영상]

[<https://drive.google.com/file/d/1o1hgjNN63bQUJM6BAjkMJZLmDlyR-EwX/view?usp=sharing>]

```

using System;
using System.Collections.Generic;
using System.Reflection;
using UnityEngine;
using UnityEngine.Rendering.Universal;

public class ShadowCaster2DTilemap : MonoBehaviour
{
    public CompositeCollider2D[] sourceColliders = { };

    private static BindingFlags accessFlagsPrivate = BindingFlags.NonPublic | BindingFlags.Instance;

    private static FieldInfo shapePathField = typeof(ShadowCaster2D).GetField("m_ShapePath", accessFlagsPrivate);

    private static FieldInfo meshHashField = typeof(ShadowCaster2D).GetField("m_ShapePathHash", accessFlagsPrivate);

    void OnEnable()
    {
        if (Application.isEditor) RefreshWorldShadows();
    }
    private void FixedUpdate()
    {
        RefreshWorldShadows();
    }

    private void RefreshWorldShadows()
    {
        ClearChildShadows();
        foreach (var source in sourceColliders)
        {
            CreateGroupsForCollider(source);
        }
    }
    private void ClearChildShadows()
    {
        var doomed = new List<Transform>();
        for (int i = 0; i < transform.childCount; i++)
        {
            var child = transform.GetChild(i);
            if (child.GetComponent<ShadowCaster2D>() == null || !child.name.StartsWith("_caster_")) continue;
            doomed.Add(child);
        }

        foreach (var child in doomed)
        {
            DestroyImmediate(child.gameObject);
        }
    }
    private void CreateGroupsForCollider(CompositeCollider2D source)
    {
        for (int i = 0; i < source.pathCount; i++)
        {
            // get the path data
            Vector2[] pathVertices = new Vector2[source.GetPathPointCount(i)];
            source.GetPath(i, pathVertices);
            Vector3[] finalVerts = Array.ConvertAll<Vector2, Vector3>(pathVertices, input => input);

            // make a new child
            var shadowCaster = new GameObject("_caster_" + i + "_" + source.transform.name);
            shadowCaster.transform.parent = transform;

            // create & prime the shadow caster
            var shadow = shadowCaster.AddComponent<ShadowCaster2D>();
            shadow.selfShadows = true;
            shapePathField.SetValue(shadow, finalVerts);
            // invalidate the hash so it re-generates the shadow mesh on the next Update()
            meshHashField.SetValue(shadow, -1);
        }
    }
}

```

7.4.1.4. class와 struct

다음 구문 요소는 클래스 정의라고 불린다

```
class X { ... };
```

이러한 클래스 정의는 다른 말로 클래스 선언이라고도 하며, 다른 선언과 마찬가지로 #include를 통해 다른 소스 파일에서 반복되어도 단일 정의 규칙을 위배하지 않는다

정의에 의하면 구조체는 소속된 멤버가 기본적으로 공개 사양인 클래스의 일종이다. 즉,

```
struct X { ... };
```

이 코드는 다음의 코드를 줄여 쓴 것이나 마찬가지다

```
class X { public: ... };
```

7.4.1.5. 생성자

*Init()*과 같은 함수를 통해 클래스 객체를 초기화하는 방식은 깔끔하지 않고 오류에 취약하다. 객체를 반드시 초기화해야한다고 강제할 수 없기에 클래스의 사용자는 초기화를 까먹거나 이중 초기화를 할 수도 있다 때문에 보다 바람직한 접근법은 객체 초기화하는 명시적 목적을 가지는 함수를 선언하는 것인데, 이러한 함수는 주어진 타입의 값을 만들어내기에 생성자라고 부른다. 생성자는 클래스 자체와 똑같은 이름을 가진다

```
class Date
{
    int m_d, m_m, m_y;

public:
    Date(const int y, const int m, const int d);
};
```

클래스가 생성자를 가지고 있으면 해당 클래스의 모든 객체는 반드시 생성자 호출에 의해 초기화된다. 생성자가 인자를 요구하면 그런 인자들이 제공되어야만 한다 만약 클래스가 위의 예제와 같은 구조를 띄고 있을 경우 제공되는 생성자 목록은 다음과 같다

Date::Date(Date&);
Date::Date(Date&&);
Date::Date(const int, const int, const int);

즉, 만약 본인이 일반적인 형태의 생성자를 선언한다면 L-Value Reference 복사 생성자와 R-Value Reference 복사 생성자가 선언될 가능성이 존재한다³¹

그러나 만약 다음과 같이 R-Value Reference 복사 생성자를 명시적으로 선언해주는 순간 그에 반대되는 L-Value Reference의 자동 생성된 버전의 복사 생성자는 삭제된 함수로 취급받기에 L-Value Reference 복사 생성자의 사용이 불가해진다. 반대의 경우로 마찬가지다

```
class Date
{
    int m_d, m_m, m_y;

public:
    Date(const int y, const int m, const int d);
    Date(Date&&); // Error
};

void Test()
{
    Date today{ 2020, 10, 14 };
    //Date dday{ today }; // Error
}
```

클래스 생성자는 클래스의 초기화를 정의하는 것으로 당연히 위의 예제에서와 같이 {}를 사용할 수도 있다

클래스 설계자는 또한 여러 개의 생성자를 제공함으로써 어떤 타입의 객체를 초기화하는 다양한 방식을 제공할 수 있다. 이때, 생성자는 통상적인 함수가 준수하는 똑같은 오버로딩 규칙을 따른다.

클래스 설계자는 클래스를 사용하는 누군가에게는 필요할 것만 같다는 이유로 항상 기능을 추가하려는 유혹을 느끼기 쉽다. 그럴 때 좋은 방법 중 하나는 기본 인자를 사용하는 것이다

```
class Date
{
    int m_d, m_m, m_y;

public:
    Date(const int y = 2024, const int m = 1, const int d = 1);
};
```

추가적으로, 클래스 선언에 생성자를 작성하지 않더라도 객체 생성 자체는 문제없이 이루어지는 것을 경험해 본적이 있을 것이다

³¹ 실제로 이 복사 생성자들이 생기는 것은 각 타입에 해당하는 복사 생성자 호출이 발생하는 순간이다

```

class Date
{
    int m_d, m_m, m_y;

public:
    void InitDate(int, int, int);
    void AddDay(int);
    void AddMonth(int);
    void AddYear(int);
};

void Test()
{
    Date today;
}

```

위의 예제에서 `today`는 `Date::Date()`가 정의되어있지 않은데도 불구하고 문제없이 생성자를 호출하여 객체를 초기화하고 있다

이는 기본 생성자³²가 호출되었는데 만약 클래스 내부에 기본 생성자 정의가 포함되어있지 않을 경우 컴파일러 버전의 기본 생성자가 클래스 내부 정의에 자동으로 포함되게 되어있기 때문이다

이러한 컴파일러 버전의 기본 생성자의 동작은 컴파일러마다 조금씩 다를 수 있겠지만, 대개 멤버 변수들에 대해 기본 생성자를 호출해주는 방식으로 동작한다. 즉, 위의 예제 코드와 같은 경우 컴파일러가 만들어주는 기본 생성자는 아마도 다음과 같이 생겼을 것이란 말이다

```

Date::Date()
{
    m_y = int();
    m_m = int();
    m_d = int();
}

```

³² 인자가 없는 생성자

7.4.1.6. explicit 생성자

생성자를 기본 세팅으로만 사용한다면 단일 인자에 의해 호출된 생성자는 인자의 타입에서 생성자 타입으로 암시적으로 변환할 수 있다

```
void Test()
{
    Date today = { 2020, 10, 14 };
}
```

위의 코드를 보면 {10, 1, 2}가 Date 타입으로 암시적으로 변환되고 있는 것을 볼 수 있다.
엄밀히 말하자면 위의 코드는 다음의 코드와 같다고 할 수 있다

```
void Test()
{
    Date today = Date(2020, 10, 14);
}
```

즉, 초기화 리스트 부분을 통해 암시적으로 생성자를 추론하여 임시 객체를 생성하고, 이를 복사 대입 연산자를 통해 본 객체를 초기화하는 것이다. 이때, 복사 대입 연산자를 호출하는 부분은 임시 객체 개념과 이동 연산에 의한 개념으로 최적화될 수 있다

만약 위와 같은 어찌보면 모호할 수 있는 생성자 호출을 제한하고 명확한 형식의 명시적인 생성자 호출만을 허가하고 싶다면 `explicit` 키워드를 생성자 앞에 추가하면 된다

```
class Date
{
    int m_d, m_m, m_y;

public:
    explicit Date(int, int, int);
};

void Test()
{
    //Date today = { 2022, 10, 14 };    // Error
}
```

7.4.1.7. 가변성

- 상수 멤버 함수

클래스 내의 멤버 함수가 상수성을 가진다 함은 멤버 변수의 값에 대해 읽기 연산만을 수행하며, 쓰기 연산은 수행하지 않음을 나타낸다 만약 상수 멤버 함수 내부에서 멤버 변수 값을 변화시키려는 시도를 할 경우 예러를 뱉어낼 것이다

상수 멤버 함수에 붙는 `const`는 그 자체로 함수 시그니처의 일부이다

다시 말해 `explicit` 생성자의 경우 클래스 외부에 정의를 제공할 경우 키워드를 함께 제공하면 예러를 뱉지만, 상수 멤버 함수는 오히려 `const`를 붙이지 않을 경우 다른 함수를 지칭하는 것이나 다름없이 된다

- 물리적 및 논리적 상수성

경우에 따라 멤버 함수가 논리적으로는 `const`이지만, 여전히 멤버의 값을 변경할 필요가 있을 때가 있다. 즉, 사용자에게는 함수가 객체의 상태를 변경하지 않는 것처럼 보이지만, 사용자가 직접적으로 인지할 수 없는 어떤 세부 사항이 간신될 수 있다. 이를 가리켜 논리적 상수성이라고 한다. 물리적 상수성이란 논리적 상수성에 포함하여 실제로도 값이 상수성을 띠는 온전한 상수성 그 자체를 말한다

예를 들어 문자열 하나를 미리 Caching해두고 이를 공개 상수 멤버 함수를 통해 사용자가 읽어갈 수 있도록 한다고 해보자

```
class Data
{
private:
    std::string m_data;

public:
    std::string GetData() const;
};
```

모종의 방법으로 `Data::m_data`를 잘 초기화해준다고 가정한다면 위 코드는 잘 작동할 것이다 그런데 만약 `Data` 내부적인 로직상 Caching된 문자열을 한 번 읽고 나면 비워야 할 필요가 생겼다고 해보자

```
class Data
{
private:
    std::string m_data = "Hello World";

public:
    std::string GetData() const;

private:
    void ClearChcheStr() const;
```

```
};
```

그러면 클래스는 위와 같이 생겼을 것이며, 함수들은 아래와 같이 정의될 것이다

```
std::string Data::GetData() const
{
    std::string data = m_data;
    ClearChcheStr();
    return data;
}
void Data::ClearChcheStr() const
{
    //m_data = "";      // Error
}
```

그런데 위의 코드로는 Data::ClearCacheStr() const에서 에러가 발생한다.

Data::ClearCacheStr() const는 Data::GetData() const에서 호출되기에 상수 멤버 함수일 수밖에 없어 상수 멤버 함수로 정의한 것인데, 그렇게 하고 보니 상수 멤버 함수의 정의에 따라 멤버 변수인 Data::m_data의 값을 변경할 수 없게 된 것이다

- **mutable**

위와 같은 경우에 필요한 것이 mutable 키워드이다

```
class Data
{
private:
    mutable std::string m_data = "Hello World";

public:
    std::string GetData() const;

private:
    void ClearChcheStr() const;
};

std::string Data::GetData() const
{
    std::string data = m_data;
    ClearChcheStr();
    return data;
}
void Data::ClearChcheStr() const
{
    m_data = "";
}
```

위와 같이 Data::m_data를 mutable로 선언하면, 사용자 입장에서의 Data::GetData() const의 논리적 상수성은 지키면서 내부적인 처리는 문제없이 돌아가게 된다

7.4.1.8. 자기 참조

상태 갱신 함수인 AddYear(int), AddMonth(int), AddDay(int)는 값을 반환하지 않도록 정의되어 있다. 그러나 많은 경우 이러한 갱신 함수의 경우 갱신된 객체에 대한 참조하는 반환하여 연산들이 연쇄적으로 처리될 수 있도록 하는 것이 유용할 수도 있다. 예를 들어 다음과 같이 사용하고 싶을 수 있다

```
int main()
{
    Date date;
    date.AddYear(2024).AddMonth(1).AddDay(1);
}
```

이를 위해서는 다음과 같이 각 함수가 Date에 대한 참조자를 반환하게 선언해야 한다

```
class Date
{
private:
    int m_y, m_m, m_d;

public:
    Date& AddYear(int);
    Date& AddMonth(int);
    Date& AddDay(int);
};
```

각각의 멤버 함수는 어떤 객체에 대해 호출되었는지를 알고 있으며, 명시적으로 해당 객체를 참조할 수 있다. 다음의 예를 보라

```
Date& Date::AddYear(int year)
{
    m_y = year;
    return *this;
}
```

static이 아닌 멤버 함수에서 this 키워드는 함수의 호출 대상이 되는 객체를 가리키는 포인터이다. 클래스 X에 소속된 const가 아닌 멤버 함수에서는 this 타입이 X*가 된다 하지만, this는 우변값으로 간주되므로 this의 주소를 얻는다던지 this에 대입하던지 하는 것은 불가능하다. 클래스 X의 const 멤버 함수에서는 this 타입이 const X*로서 객체 자체의 수정을 방지해준다

대부분 `this`의 사용은 암시적이다. 특히 클래스 내부에서 `static`이 아닌 멤버 함수 내부에서의 멤버 변수에 대한 참조자는 `this`의 암시적 사용에 의존한다. 즉, 위의 코드는 다음과 같이 바꿔도 똑같다

```
Date& Date::AddYear(int year)
{
    this->m_y = year;
    return *this;
}
```

7.4.1.9. 멤버 접근

클래스 X의 객체에 대해서는 `.(점)` 연산자를 적용하거나 클래스 X의 객체를 가리키는 포인터에 대해서는 `->(화살표)`를 적용하여 클래스 X의 멤버에 접근할 수 있다

```
struct Data
{
    void Func();
    int m_m;
};

void Test(Data data, Data* dataPtr)
{
    //m_m = 1;           // Error
    data.m_m = 1;
    //data->m_m = 1;    // Error
    //dataPtr.m_m = 1;    // Error
    dataPtr->m_m = 1;
}
```

특정 객체의 멤버가 아니라 일반적으로 어떤 멤버를 참조하고 싶다면 `::`에 이어지는 클래스 이름으로 한정할 수 있다

```
struct Data
{
    void Func();
    int m_m;
};

void Data::Func()
{
    // ...
}

void Test()
```

```
{  
    std::cout << typeid(Data::m_m).name();  
    void(Data::* fptr)() { &Data::Func };  
}
```

7.4.1.10. static 멤버

클래스의 일부이면서도 해당 클래스의 객체가 아니며, 따라서 일종의 전역 변수와 같은 감각으로 사용할 수 있는 변수가 static 멤버이다

아래한 static 멤버는 통상적인 non-static 멤버처럼 객체당 하나씩 사본이 있는 것이 아니라, static 멤버는 해당 클래스에 정확히 하나의 사본이 존재한다. 마찬가지로 클래스의 멤버에 접근을 필요로 하는 함수지만, 특정 객체에 대해 호출될 필요가 없는 함수를 static 멤버 함수라고 한다

멀티스레드 코드에서 일반적으로 static 데이터 멤버는 Race-Condition을 피하기 위해 일종의 잠금이나 접근 순서 제어가 필요한 등의 사용에 주의가 필요하다

이러한 static 멤버에 대한 유명한 예 중 하나로 Single-Ton 패턴이 있다

```
class DataChiefManager  
{  
private:  
    static DataChiefManager* m_instance;  
  
    // ...  
  
public:  
    static DataChiefManager* GetInstance()  
    {  
        if (m_instance == nullptr)  
        {  
            m_instance = new DataChiefManager();  
        }  
        return m_instance;  
    }  
    static bool Dispose()  
    {  
        if (m_instance == nullptr)  
        {  
            return false;  
        }  
        delete m_instance;  
        return true;  
    }  
}
```

```
// ...  
};
```

알아두어야 할 점은 위의 코드는 그대로 사용하기엔 문제가 있으며, 그저 예시를 보이기 위한 코드임을 인지해야만 한다. 또한, Single-Ton을 구현하는 방법도 위의 방법 뿐만 아니라 Lazy-Initialization 개념을 동반하여 구현하는 방법 등 다양한 방법이 존재한다는 것도 인지해야만 할 것이다

8. 클래스의 보다 깊은 이해를 위해 - 이동 동작

이동(move)란 C++ 11부터 추가되었으며 복사와 비슷하지만 다른 기능이다

8.1. 이동 의미론, 완벽 전달의 개념

C++ 11부터는 이동 연산을 통해 이동 의미론과 완벽 전달이라는 개념을 지원하며, 이 개념들은 다음과 같다

이동 의미론은 소유권의 이전이란 개념이라 생각해도 되며, 보다 쉽게 이해하기 위해서는 `std::unique_ptr`을 보면 대략적인 느낌을 알 수 있다

즉, 값이 이동된다는 것은 원본 객체의 값을 말 그대로 온전히 목표 객체가 전달받고 원본 객체는 비어있는 상태가 된다는 것이다. 이는 목표 객체를 원본 객체의 사본으로 채우고 그 결과 원본과 사본 모두가 활용 가능한 상태로 존재하는 것과는 전혀 다른 동작이다

의미 이동론이 유효한 문맥에서 컴파일러는 비싼 복사 연산을 덜 비싼 이동 연산으로 대체할 수 있다

객체를 복사한다는 것의 구체적인 의미를 그 객체에 대한 복사 생성자와 복사 대입 연산자를 통해서 명시적으로 지정할 수 있듯이, 객체를 이동한다는 것의 의미를 그 객체에 대한 이동 생성자와 이동 대입 연산자를 통해 지정할 수 있다

이동 의미론은 `std::unique_ptr`이나 `std::futre`, `std::thread`와 같은 이동 전용 형식의 작성을 가능하게 한다

완벽 전달에서 전달(Forwarding)이란 말 그대로 한 함수가 자신의 인자를 다른 함수에 전달하는 행위를 말한다. 이때 목표는 피호출자 함수가 받은 매개변수가 호출자 함수가 받은 매개변수와 완전히 동일한 객체임을 목표로 한다

완벽 전달이란 단순히 객체들을 전달하는 것만이 아니라, 그 객체들의 주요 특징들도 포함하여 온전한 상태 그대로 전달하는 것을 의미한다. 이때 보존되는 특징에는 다음과 같은 것들이 존재한다

- 형식
- 원값 혹은 오른값 여부
- `const` 혹은 `volatile` 여부

이러한 정보들을 보존하며 전달하는 방법으로는 크게 포인터를 활용하는 방법과 참조를 활용하는 방법이 있을 수 있다. 그러나 호출자에게 포인터를 전달하도록 강제하는 것은 그것이 올바른 때(포인터 연산이 필요한 경우)도 있겠지만 일반적이고 범용적인 함수들에서는 참조를 사용하는 것이 호출자 입장에서도 우선 일차적으로 위험한 포인터 연산을 사용하지 않을 것이라는 가정을 깔고가는 것이고 호출 문장을 보더라도 일반적인 값 복사 전달과 차이를 보이지 않는다는 점에서 보다 바람직하다고 할 수 있을 것 같다

따라서 완벽 전달을 위해서는 참조 매개변수가 필요하며, 더 구체적으로는 전달 참조를 통해야만 한다

C++에서 이 두 개념이 구현될 수 있도록 해주는 것은 오른쪽 참조라는 언어적 기능 덕분이다

이동의 개념이 사용되는 코드를 보면 주의를 기울여 봐야만 할 것이다. 왜냐하면 이동은 미묘한 구석이 많은 개념이고 그렇기에 실수하기 쉽기 때문이다

예를 들어 (1) 복사 대신 이동을 사용함으로써 최적화를 시도한 것 같지만, 이동이 항상 복사보다 저렴한 것은 아닐 수 있고, (2) std::move가 모든 것을 이동시키지는 않으며, (3) 완벽 전달은 모든 것을 완벽하게 전달해주지 못한다. (4) 또한 이동이 유효한 문맥에서 항상 이동 연산이 호출되는 것도 아니며, (5) 'T&&' 형식이 항상 오른값 참조를 나타내는 것도 아니다

이 챕터를 읽기위해 명심해야할 개념이 있으니 함수 매개 변수의 참조 형식과 매개 변수 그 자체의 값 형식은 별개라는 것이다

```
template <typename _T>
void Func(_T&& value);
```

예를 들어 위의 T&& value를 분석해보자

먼저 이 값의 타입만 보자면 템플릿 인자 T에 대한 오른값 참조임이 자명하다. 그러나 T&& value 그 자체가 담고 있는 값은 왼값인지 오른값인지 분명치 않으며 둘 중 어느쪽 값을 쥐고 있는지는 템플릿 인자 T에 부호화되어 담겨있다. 또한 중요한 점 나머지 하나는 value는 T&&로서 오른값 참조가 타입이지만 변수 자체로서는 왼값임을 언제나 명심해야만 한다

이동 연산은 이러한 왼값, 오른값과 같은 값의 범주와 위에서 보인 하나의 오른값 참조에 공존할 수 있는 다양한 해석에 의존하기에 이동 연산을 사용하는 코드를 작성할 때에는 언제나 주의해야만 한다

8.2. 전달 참조(Forwarding Reference)와 오른값 참조

전달 참조란 이동 연산이 필요한 참조의 하나로서 대강만 보자면 오른값 참조와 구분이 쉽지 않을 수 있으며, 심지어 전달 참조의 형태를 국지적으로 완벽하게 갖추더라도 전체의 맥락을 봤을 때 조건을 맞추지 못한다면 전달 참조가 아닌 오른값 참조로서 동작하게 된다

전달 참조란 유례없는 유연성을 C++에 부여해줄 수 있는 참조의 일종으로서 그 특징으로서 전달 참조는 문맥에 따라 왼값 참조일 수도 있고, 오른값 참조일 수도 있다
예를 들어 다음과 같이 사용될 수 있다는 것이다

```
template <typename _T>
void Func(_T&& value);

void Test()
{
    Date today;

    Func(today);           // 전달 참조는 왼값 참조로서 동작한다
    Func(std::move(today)); // 전달 참조는 오른값 참조로서 동작한다
}
```

전달 참조의 조건은 다음의 두 가지이며 얼핏 보이겐 크게 어렵게 보이지 않을 수 있다

- 전달 참조는 반드시 T&&의 형태여야만 한다
- 형식 연역이 일어나야만 한다

위의 조건을 만족하며 전달 참조가 나타날 수 있는 맥락은 다음의 두 가지가 존재한다

```
template <typename _T>
void Func(_T&& value);           // value는 전달 참조

void Test()
{
    auto&& val = tempVal; // val은 전달 참조
}
```

void func(T&& value)의 value와 auto&& val의 val은 전달 참조이다

그렇다면 비슷하게 생겼지만 전달 참조가 아닌 예를 살펴보자

```
void Func(Date&& value);           // value는 오른값 참조

void Test()
{
```

```
        Date&& val = tempVal;    //  val은 오른값 참조
    }
```

위의 예에서 보이는 value와 val은 둘 다 오른값 참조이다

위의 두 예시와 전달 참조의 조건을 함께 보자면 전달 참조가 기능하는 맥락을 보자면 다음과 같다

- 템플릿 함수의 경우

템플릿 함수가 템플릿 인자 T에 대해 형식이 연역, 즉, 형식이 확정되어 인스턴스화되는 단위가 함수 그 자체로서 딱 떨어지기 때문이다

- auto 변수의 경우

auto 변수에 초기화 혹은 대입하는 R-Value에 대해 auto 변수의 형식이 연역되기 때문이다

auto&&의 경우 헷갈릴 여지가 적다. 그러나 템플릿 함수가 어떤 클래스의 멤버 함수로 존재할 경우 맥락에 따라 전달 참조가 아닌 단순한 오른값 참조로서만 기능하게 된다

```
// In vector
_EXPORT_STD template <class _Ty, class _Alloc = allocator<_Ty>>
class vector { // varying size array of values
    // ...

public:
    _CONSTEXPR20 void push_back(_Ty&& _Val) {
        // insert by moving into element at end, provide strong
guarantee
        _Emplace_one_at_back(_STD move(_Val));
    }

    // ...

private:
    template <class... _Valty>
    _CONSTEXPR20 _Ty& _Emplace_one_at_back(_Valty&&... _Val) {
        // insert by perfectly forwarding into element at end, provide
strong guarantee
        auto& _My_data = _Mypair._Myval2;
        pointer& _Mylast = _My_data._Mylast;

        if (_Mylast != _My_data._Myend) {
            return _Emplace_back_with_unused_capacity(_STD
forward<_Valty>(_Val)...);
        }
    }
}
```

```

        return *_Emplace_reallocate(_Mylast, _STD
forward<_Valty>(_Val)...);
    }

    // ...
};

// In main.cpp
void Test()
{
    std::vector<int> v;
    v.push_back(1);
}

```

위의 코드는 Visual C++의 `std::vector` 코드의 일부이다
보면 `std::vector`의 경우 명확히는 `std::vector<_Ty, _Alloc>`으로 선언된다는 것을 알 수 있다

중요한 `std::vector<_Ty, _Alloc>::push_back(_Ty&& _Val)`을 살펴보자. 이때의 `_Val`은 전달 참조인가? 아니면 오른값 참조인가? 답은 너무나 자명하게도 오른값 참조이다
왜냐하면 `std::vector<_Ty, _Alloc>::push_back(_Ty&& _Val)`에서 `_Ty&& _Val`의 `_Ty`는 `vector`의 인스턴스를 선언할 때부터 연역되며, 따라서 `std::vector<_Ty, _Alloc>::push_back(_Ty&& _Val)`는 클래스 단위로 형식 연역이 일어나는 것이지 함수 단위로 연역되는 것이 아니기 때문이다
즉, 위의 코드에서 `push_back`이 호출되는 시점에서 이미 `push_back`은 다음과 같이 알려져 있으며, 따라서 형식 연역이 발생할 여지가 전혀 없다는 것이다

```

// In vector
_EXPORT_STD template <class _Ty, class _Alloc = allocator<_Ty>>
class vector { // varying size array of values
    // ...

public:
    _CONSTEXPR20 void push_back(_Ty&& _Val) {
        // insert by moving into element at end, provide strong
guarantee
        _Emplace_one_at_back(_STD move(_Val));
    }
    // ...
};

// In main.cpp
void Test()
{
    std::vector<int> v;
    v.push_back(1);
}

```

}

그렇다고 Visual C++의 `std::vector`가 이동 연산에 따른 최적화를 포기했느냐 하면 그렇지 않은데, 그 흔적을 `_Emplace_one_at_back(_Valty&&... _Val)`을 통해 알 수 있다
`_Emplace_one_at_back(_Valty&&... _Val)`을 보면 `std::vector<_Ty, _Alloc>`와 독립된 템플릿 인자인 `<class... _Valty>`을 쓰고 있는 것을 알 수 있으며, 따라서
`_Emplace_one_at_back(_Valty&&... _Val)`의 `_Valty`는 `_Emplace_one_at_back`가 호출됨에 따라 함수 단위로 형식 연역이 일어나며 인스턴스화된다. 따라서
`_Emplace_one_at_back(_Valty&&... _Val)`에서의 `_Val`은 전달 참조이다
즉, 위의 코드에서 `_Emplace_one_at_back`은 `push_back` 안에서 호출되는 시점에서야
`_Valty`의 형식을 연역할 수 있기에 전달 참조인 것이다

8.3. 참조 축약 규칙

```
template <typename _T>
void Func(_T&& value);
```

문서의 앞 부분에서 언급했듯 다음과 같은 함수에서 인자가 템플릿에 전달되었을 때 템플릿 매개변수에 대해 연역된 형식에는 그 인자가 원값인지 아니면 오른값인지에 대한 정보가 부호화되어 내제되어 있다

그런데 중요한 점은 이러한 부호화는 템플릿 인자가 전달 참조 매개변수를 초기화하는 데 쓰일 때에만 일어난다

부호화가 원값인 경우와 오른값인 경우에 대해 어떤 메커니즘으로 일어나는지는 간단하다

전달된 인자	부호화된 인자
T(원값인 경우)	T&
T(오른값인 경우)	T

이를 다음의 예제를 다시 활용하여 함수의 선언부가 어떻게 연역되는지를 보자면 다음과 같다

```
template <typename _T>
void Func(_T&& value);

void Test()
{
    Date today;

    Func(today);           // 전달 참조는 원값 참조로서 동작한다
    Func(std::move(today)); // 전달 참조는 오른값 참조로서 동작한다
}
```

func(today)	void Func(Date& && value);
func(std::move(today))	void Func(Date&& value);

위의 연역된 코드를 보다보면 이상한 점을 눈치챌 수 있을텐데, Date & &&라는 말도 안되는 타입이 적혀있기 때문이다. C++에서 허용되는 참조의 종류는 T&와 T&&밖에 없다. 그런데

위의 코드는 문제없이 동작한다. 그 이유는 특수한 몇몇 문맥에서는 위와 같은 말도 안되는 참조 형식을 본래 C++에서 허용되는 참조 종류의 하나로 참조를 축약해주기 때문이다

그 규칙은 다음과 같다

만일 두 참조 중 하나라도 원값 참조면 결과는 원값 참조이다. 그렇지 않으면 결과는 오른값 참조이다

즉, 참조의 조합에 따른 참조 축약 결과는 다음과 같이 된다

참조 축약 전 타입	참조 축약 후 타입
Date& &	Date&
Date& &&	Date&
Date&& &	Date&
Date&& &&	Date&&

따라서 다음의 경우

```
void Func(Date& && value);
```

다음과 같이 참조 축약된다

```
void Func(Date& value);
```

8.4. std::move와 std::forward

- std::move

오른값 참조는 이동할 수 있는 객체에만 묶인다. 어떤 매개변수가 오른값 참조라면 그 참조에 묶인 객체를 이동할 수 있음이 확실하다

```
class Date
{
public:
    Date(Date&& date);
};
```

그러한 객체를 다른 함수에 넘겨주되, 오른값이 성질을 활용할 수 있도록 넘겨줘야 하는 경우도 생긴다. 그럴 때에는 그런 객체에 묶이는 객체를 오른값으로 캐스팅해야만 한다³³. std::move가 하는 일이 바로 이것이다

이를 확실히 보기 위해 Visual C++의 std::move의 구현을 보자면 다음과 같다

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr remove_reference_t<_Ty>&&
move(_Ty&& _Arg) noexcept {
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);
}
```

여기서 핵심은 remove_reference_t이며, 이것이 하는 일은 이름 그대로 해당 타입의 참조 속성을 없애준다. 그러면 std::move의 동작은 매우 명확해진다 예를 들어 Date& 타입의 왼값으로 std::move를 사용한다고 해보자. 그럴 경우 참조 축약은 다음과 같이 차례대로 전개된다

```
_NODISCARD _MSVC_INTRINSIC constexpr remove_reference_t<Date&>&&
move(Date& && _Arg) noexcept {
    return static_cast<remove_reference_t<Date&>&&>(_Arg);
}
```

```
_NODISCARD _MSVC_INTRINSIC constexpr Date&& move(Date& _Arg) noexcept {
    return static_cast<Date&&>(_Arg);
}
```

³³ 오른값을 묶고 있는 매개변수 그 자체는 원값이기에

- std::forward

반면 전달 참조는 이동에 적합한 객체에 묶일 수도 있고 아닐 수도 있다. 전달 참조는 오른값으로 초기화된 경우에만 오른값으로 캐스팅해야만 한다
이를 확실히 보기 위해 Visual C++의 std::forward의 구현을 보자면 다음과 같다

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&&
forward(remove_reference_t<_Ty>& _Arg) noexcept {
    return static_cast<_Ty&&>(_Arg);
}

_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&&
forward(remove_reference_t<_Ty>&& _Arg) noexcept {
    static_assert(!is_lvalue_reference_v<_Ty>, "bad forward call");
    return static_cast<_Ty&&>(_Arg);
}
```

std::move 때와 같이 Date& 타입의 왼값으로 std::forward를 사용한다고 해보자

```
_NODISCARD _MSVC_INTRINSIC constexpr Date& &&
forward(remove_reference_t<Date&&>& _Arg) noexcept {
    return static_cast<Date& &&>(_Arg);
}
```

```
_NODISCARD _MSVC_INTRINSIC constexpr Date& forward(Date& _Arg) noexcept
{
    return static_cast<Date&>(_Arg);
}
```

그에 비해 Date&& 타입의 왼값으로 std::forward를 사용한다고 해보자

```
_NODISCARD _MSVC_INTRINSIC constexpr Date&& &&
forward(remove_reference_t<Date&&>&& _Arg) noexcept {
    static_assert(!is_lvalue_reference_v<Date&&>, "bad forward call");
    return static_cast<Date&& &&>(_Arg);
}
```

```
_NODISCARD _MSVC_INTRINSIC constexpr Date&& forward(Date&& _Arg)
noexcept {
    static_assert(!is_lvalue_reference_v<Date&&>, "bad forward call");
    return static_cast<Date&>(_Arg);
}
```

- 오른값 참조에는 `std::move`를, 전달 참조에는 `std::forward`를 사용하라

결론부터 말하자면 `std::move`는 오른값 참조를 다른 함수로 전달할 때에만 사용되어야만 하며, `std::forward`는 전달 참조를 다른 함수로 전달할 때에만 사용되어야만 한다

물론, 오른값 참조에 `std::forward`를 적용해도 원하는 행동이 일어나게 하는 것은 가능하다. 그러나 사용에 있어 실수의 여지가 존재한다는 점에서 그러한 용태는 지양하는 것이 바람직할 것이다. 따라서 오른값 참조에는 `std::forward`를 사용하는 것은 피해야 할 것이다

직접 참조에 `std::move`를 적용하는 것은 더 나쁜 결과를 불러올 수 있는데 원본 객체가 의도치 않게 활용이 불가능한 전혀 쓸모없는 상태로 전환될 가능성성이 존재하기 때문이다

```
class Widget
{
private:
    std::string m_name;

public:
    template <typename _T>
    void SetName(_T&& name)
    {
        m_name = std::forward<_T>(name);
    }
};

std::string GetWidgetName();

int main()
{
    Widget w;
    auto targetName = GetWidgetName();

    // 여기까지의 코드에서 targetName의 값은 명확하다

    w.SetName(targetName);

    // 여기까지의 코드에서 targetName의 값은 명확하지 않다
}
```

위의 코드에서 핵심은 `main` 함수 내의 `name`의 상태이다. 분명 `Widget` 클래스의 설계자가 함수 설계의 관례를 알고 있다면 `Widget::SetName`의 매개변수인 `name`은 읽기 전용으로 쓰이길 바라고, 매개변수를 전달 참조로 사용한 것은 이동 연산으로서 복사의 비용을 아끼고

싶었던 것일 것이다. 즉, 클래스 설계자는 매개변수의 원본에 영향을 미치는 일은 일반적으로 상정하지 않았을 것이다

그러나 이동 연산의 특징상 이동 연산 후의 원본 객체의 값은 쓸모없는 거의 빈 것이나 다름없는 상태로 만드는 것이 권장 사항이며, 따라서 이동 연산이 있은 후 main 함수 내 targetName은 쓸모 없는 상태가 될 것이다

이를 해결하기 위한 방법의 일환으로서 Widget::SetName을 원값 참조를 받는 버전과 오른값 참조를 받는 버전을 각각 하나씩 오버로딩하는 방법을 사용할 수도 있을 것이다

그러나 생각해보면 전달 참조 매개변수가 언제나 하나라는 법은 없다. 만약 전달 참조 매개변수가 n개인 경우를 생각해보자면 해당 함수의 오버로딩 버전은 2^n 개 필요할 것이다. 심지어 ...을 활용하여 잠재적으로 무제한으로 전달 참조를 받는 함수가 존재할 수도 있다

이러한 상황은 잠재적으로 소스코드의 크기와 최종 바이너리의 크기를 모두 기하급수적으로 늘릴 수 밖에 없으나, 이를 간단하게 해결하는 방법이 바로 std::forward를 사용하는 것이다. std::forward를 사용하면 기하급수적으로 늘어날 수 있는 오버로딩 함수를 선언할 필요 없이 원값과 오른값을 적절히 처리할 수 있다

예를 들어 다음은 Visual C++의 std::make_shared 코드이다. 코드를 자세히 보면 위에서 언급한 이슈를 피하기 위해 std::forward를 사용하고 있는 것을 볼 수 있다

```
_EXPORT_STD template <class _Ty, class... _Types>
_NODISCARD_SMART_PTR_ALLOC shared_ptr<_Ty> make_shared(_Types&&...
_Args) {
    const auto _Rx = new _Ref_count_obj2<_Ty>(_STD
forward<_Types>(_Args)...);
    shared_ptr<_Ty> _Ret;
    _Ret._Set_ptr_rep_and_enable_shared(_STD
addressof(_Rx->_Storage._Value), _Rx);
    return _Ret;
}
```

8.5. 완벽 전달이 실패하는 경우들

이동 연산은 완벽 전달을 가정하고 사용하는 처리이다. 그러나 때로 이동 연산은 완벽 전달을 달성하지 못할 때가 있으며, 그러한 조건도 평소 자주 접할 수 있는 형태들이기에 알아두는 것이 좋다

완벽 전달이 성공한건지 실패한건지를 확인하고자 한다면 다음의 조건을 만족하면 된다

- 보편 참조가 아닌 매개변수를 받는 평범한 함수 func와 그런 func를 전달하는 다음과 같이 생긴 fwd가 존재할 때, 두 가지 방법에서 최종적으로 func가 받는 매개변수가 완벽히 일치한다면 완벽 전달이 성공한 것이고, 조금이라도 다른 점이 있다면 완벽 전달이 실패한 것이다

```
template <typename _T>
void fwd(_T&& param)
{
    func(std::forward<_T>(param));
}

// or

template <typename... _T>
void fwd(_T&&... param)
{
    func(std::forward<_T>(param)...);
}
```

- 중괄호 초기치

- func의 선언이 **void func(const std::vector<int>& numList)**라고 가정하자
- 이러한 상황에서 다음과 같은 코드를 실행한다고 하자

```
func({ 0, 1, 2, 3 }); // (1)
fwd({ 0, 1, 2, 3 }); // (2)
```

각각의 경우에 어떤 일이 일어나는지를 살펴보자면 다음과 같이 정리할 수 있다

- (1)의 경우

이 경우 func의 매개변수의 형식인 `std::vector<_Ty, _Alloc>`의 생성자 오버로딩 중에서 { 0, 1, 2, 3 }를 암시적 변환하여 오버로딩 해결될 수 있는 하나의 생성자가 있는지를 살펴봐야만 한다

그런데 `std::vector<_Ty, _Alloc>::vector(initializer_list<_Ty> _llist, const _Alloc& _Al = _Alloc())`가 존재한다. 따라서 오버로딩은 해결되고 { 0, 1, 2, 3 }을 통해 `std::initializer_list<_Ty>`를 생성하고자 시도한다

`std::initializer_list<_Ty>`의 경우 다음의 생성자를 통해 초기화되게 된다

```
_STD_BEGIN
_EXPORT_STD template <class _Elem>
class initializer_list {
public:
    constexpr initializer_list(const _Elem* _First_arg, const _Elem* _Last_arg) noexcept
        : _First(_First_arg), _Last(_Last_arg) {}

    // ...

private:
    const _Elem* _First;
    const _Elem* _Last;
};
```

그 후 `std::initializer_list<_Ty>`를 통해 `std::vector<_Ty, _Alloc>`를 생성한다

```
_EXPORT_STD template <class _Ty, class _Alloc = allocator<_Ty>>
class vector { // varying size array of values
private:
    using _Alty = _Rebind_alloc_t<_Alloc, _Ty>;
    using _Alty_traits = allocator_traits<_Alty>;

public:
    using value_type = _Ty;
    using allocator_type = _Alloc;
    using pointer = typename _Alty_traits::pointer;
    using const_pointer = typename _Alty_traits::const_pointer;
    using reference = _Ty&;
    using const_reference = const _Ty&;
    using size_type = typename _Alty_traits::size_type;
    using difference_type = typename _Alty_traits::difference_type;

private:
    using _Scary_val =
    _Vector_val<conditional_t<_Is_simple_alloc_v<_Alty>,
    _Simple_types<_Ty>,
    _Vec_iter_types<_Ty, size_type, difference_type, pointer,
    const_pointer>>;
```

```

public:
    _CONSTEXPR20 vector(initializer_list<_Ty> _Ilist, const _Alloc& _Al
= _Alloc())
        : _Mypair(_One_then_variadic_args_t{}, _Al) {
            _Construct_n(_Convert_size<size_type>(_Ilist.size()),
_Ilist.begin(), _Ilist.end());
    }

private:
    _Compressed_pair<_Alty, _Scary_val> _Mypair;
};

```

Visual C++에서는 `std::vector<_Ty, _Alloc>::vector(initializer_list<_Ty> _Ilist, const _Alloc& _Al = _Alloc())`가 호출된 후로도 추가적인 처리들이 존재하지만 단순화하자면 이 정도에서 `func`의 매개변수 준비가 끝난다

- (2)의 경우

이러한 경우에는 `func`의 매개변수의 형식인 `std::vector<_Ty, _Alloc>`와 `fwd`의 인자로서 넘겨진 `{ 0, 1, 2, 3 }`을 곧바로 비교하여 `std::vector<_Ty, _Alloc>` 생성자의 오버로딩 해결을 할 수 없다

이럴 때 컴파일러가 할 수 있는 일은 `fwd`의 형식인 `_T`로 `{ 0, 1, 2, 3 }`을 연역하고, 이렇게 연역된 형식을 `std::vector<_Ty, _Alloc>`과 비교하여 적절한 생성자를 오버로딩 해결을 통해 찾아내 매개변수를 초기화할 수 있는지를 시도한다

그 결과로서는 컴파일 자체가 불가능하거나 혹은 컴파일 된다 하더라도 그냥 곧바로 `func`를 호출할 때와는 결과가 달라지게 된다

(2)의 경우에는 특히 `{...}` 형식을 통해 템플릿 인자를 연역하고자 시도하는데, 이러한 경우를 표준에서는 비연역 문맥(Non-Deduced Context)라고 하며 따라서 `{...}`를 통해 템플릿 인자 형식을 연역하는 것이 금지되어 있다는 것이다

즉, 애초에 (2)의 경우는 컴파일조차 되지 않는다는 것이다

- 해결책

`auto` 변수는 `{...}`에서 `std::initializer_list<_Ty>`로의 연역이 가능한 경우 중 하나이다. 따라서 다음과 같이 사용해 경우 (2)의 문제를 해결하고 완벽 전달이 가능하게 된다

```

auto numList = { 0, 1, 2, 3 };
fwd(numList);

```

- 널 포인터를 의미하는 0 혹은 NULL

- func의 선언이 **void func(const int* ptr)**이라고 가정하자
- 이러한 상황에서 다음과 같은 코드를 실행한다고 하자

```
// (1)
func(0);           // Fine
func(NULL);        // Fine
func(nullptr);     // Fine

// (2)
//fwd(0);          // Error
//fwd(NULL);        // Error
fwd(nullptr);      // Fine
```

각각의 경우에 어떤 일이 일어나는지를 살펴보자면 다음과 같이 정리할 수 있다

- (1)의 경우

이 경우에는 0과 NULL은 0x0000...0000으로서 int*로서 연역될 수 있다. 또한, nullptr은 본래 정의상 어떤 포인터 형식으로도 연역될 수 있다. 따라서 (1)의 경우에는 0와 NULL을 이런 방식으로 사용하는 것이 바람직한지는 차치하더라도 컴파일과 실행이 문제없이 된다

- (2)의 경우

이 경우에는 0과 NULL의 경우 fwd의 인자 param의 경우에는 _T가 int로서 연역된다. 그 후 func의 ptr의 형식인 int*로서 연역해야만 하는데, int에서 int*로의 연역은 불가능하다

물론 nullptr의 경우에는 아무런 문제없이 동작한다

- 해결책

즉, 빈 포인터를 명시할 때에는 0과 NULL을 사용하기보다는 nullptr을 사용하는 것이 의도치 않은 에러 혹은 동작을 미연에 방지하는 대책이 될 수 있다

- 선언만 된 정수 static const 및 constexpr

- static const와 constexpr 멤버 변수는 정의를 제공할 필요 없이 클래스 내 선언만 존재하면 어떤 함수에서 해당 변수를 사용하는데 대개의 경우에는 큰 문제 없이 돌아간다. 이러한 개념은 상수 전파(const propagation)이라고 하며, 이러한 경우 해당 멤버를 위한 메모리를 할당하지 않고 해당 멤버 변수가 등장한 곳에 해당 멤버 변수의 값을 맞바꿈으로서 동작하기 때문이다
- func의 선언이 **void func(const std::size_t size)**라고 가정하자
- 이러한 상황에서 다음과 같은 코드를 실행한다고 하자

```

class Widget
{
public:
    static const std::size_t m_val0 = 10;
    static constexpr std::size_t m_val1 = 20;
};

int main()
{
    // (1)
    func(Widget::m_val0);
    func(Widget::m_val1);

    // (2)
    fwd(Widget::m_val0);
    fwd(Widget::m_val1);
}

```

각각의 경우에 어떤 일이 일어나는지를 살펴보자면 다음과 같이 정리할 수 있다

- (1)의 경우

이 경우에는 상수 전파를 통해 Widget::m_val0과 Widget::m_val1이 각각의 값으로 교체되며 문제 없이 동작하게 된다

- (2)의 경우

이 경우는 컴파일러 구현사항에 따라 달라질 수 있다. 실제로 테스트한 결과 Visual C++ v143 기준으로는 문제없이 동작하지만, g++ 11.3.0 -std=c++11 기준으로는 다음의 에러 메시지와 함께 컴파일 실패³⁴

```

/usr/bin/ld: /tmp/cczKXL4a.o: warning: relocation against
`_ZN6Widget6m_val1E' in read-only section `text'
/usr/bin/ld: /tmp/cczKXL4a.o: in function `main':
main.cpp:(.text+0x2e): undefined reference to `Widget::m_val0'
/usr/bin/ld: main.cpp:(.text+0x3d): undefined reference to
`Widget::m_val1'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
collect2: error: ld returned 1 exit status

```

즉, 이 경우의 결과는 컴파일러 구현사항에 따라 달라진다.

문제의 원인을 보자면 자명하다. 앞서 상수 전파는 마치 매크로와 같이 멤버 변수를 위해 메모리를 할당하는 대신 그저 해당 멤버 변수가 언급된 곳에 멤버 변수의 값을 복사&붙여넣기해줄 뿐이기 때문이다³⁵

³⁴ 정확히는 링킹 실패

³⁵ 물론, 프리프로세서에 의해 동작하는 것이 아니기에 형식 연역에 의한 최소한의 타입 검증은 일어난다

즉, 원칙적으로 말하자면 상수 전파가 일어나는 멤버 변수는 애초에 실체가 있는 객체조차 아니기에 당연히 이를 가리키는 주소값이 존재할 수 없다

그러나 fwd의 경우 이를 전달 참조를 통해 받아오고자 하고 있다. 이는 명백한 모순이다. 참조는 그 종류와 상관없이 충분히 저수준으로 내려간다면 포인터와 구분이 되지 않는다. 즉, 참조는 적용되는 형태와 규칙이 조금 다른 포인터라고도 말할 수 있는 것이다. 그런데 포인터에 상수 전파 멤버 변수를 가리키게하는 것은 불가능하기에 당연히 링킹 과정에서 주소값을 찾을 수 없어 컴파일 에러를 일으키는 것이다

Visual C++의 경우 컴파일러 차원에서 이러한 문제가 발생할 때 프로그래머가 추가적인 조치를 해주지 않더라도 임의로 상수 전파 멤버 변수의 정의를 추가하여 포인터가 가리킬 주소값을 마련하기에 문제없이 컴파일되고 동작하는 것이다

- 해결책

본인이 Visual C++과 같이 상수 전파의 문제를 알아서 해결해주는 컴파일러가 동작하는 플랫폼에서만 제품을 작성할 것이라면 사실 문제점은 애초에 존재하지 않는 것이나 진배없다. 그러나 그렇지 않다면 원칙적인 방법에 따라 상수 전파에 의한 최적화는 포기하되 다음과 같이 static const 혹은 static constexpr 멤버 변수의 정의를 제공하는 것으로 문제를 해결할 수 있다

```
const std::size_t Widget::m_val0;
constexpr std::size_t Widget::m_val1;
```

이를 적용한 코드는 다음과 같다

```
class Widget
{
public:
    static const std::size_t m_val0 = 10;
    static constexpr std::size_t m_val1 = 20;
};

const std::size_t Widget::m_val0;
constexpr std::size_t Widget::m_val1;

int main()
{
    // (1)
    func(Widget::m_val0);
    func(Widget::m_val1);

    // (2)
    fwd(Widget::m_val0);
    fwd(Widget::m_val1);
}
```

- 오버로딩된 함수 이름과 템플릿 이름

- func의 선언이 **void func(int (*fn)(int))** 혹은 **void func(int fn(int))**라고 가정하자
- 또한 int TargetFunc(int)와 int TargetFunc(int, int)가 존재한다고 하자
- 이러한 상황에서 다음과 같은 코드를 실행한다고 하자

```
int TargetFunc(int);
int TargetFunc(int, int);

int main()
{
    // (1)
    func(TargetFunc);

    // (2)
    fwd(TargetFunc);
}
```

각각의 경우에 어떤 일이 일어나는지를 살펴보자면 다음과 같이 정리할 수 있다

- (1)의 경우

이 경우의 결과부터 말하자면 가능한 사용법이며 문제없이 동작할 것이다

얼핏 보면 당연하다고 생각할 수 있지만 함수가 오버로딩된 이상 함수의 이름 그 자체는 특정 하나의 함수를 가리키지 못하며 단순한 특정한 함수 오버로딩 집합을 지칭하는 것 이상의 의미를 지니지 못한다. 그럼에도 문제없이 동작하는 것은 컴파일러가 함수 오버로딩 집합에 존재하는 함수들의 시그니처와 func의 매개변수의 함수 포인터 형식을 비교하여 적절한 함수를 찾아 그 주소를 넘겨주기 때문이다

- (2)의 경우

이 경우에도 (1)에서의 경우와 같이 컴파일러가 알아서 잘 연결해줄 것이라 기대할 수도 있겠지만 그렇지 않다

분명 위와 같이 코드를 작성한 프로그래머는 fwd의 템플릿 인자 _T가 적절한 함수 포인터 형식으로 연역되길 기대하겠지만, 앞서 말했듯 함수가 오버로딩된 순간 그 이름은 특정한 함수 하나를 지칭하지 않고 함수 오버로딩 집합을 가리킬 뿐이다. 따라서 _T는 함수 오버로딩 집합 내에 존재하는 많은 함수 중 어떤 함수의 형식으로 연역되어야만 하는지 판단할 수 없다. 즉, 다시 말해 형식 연역이 일어나지 않는다는 것이다

완벽 전달은 형식 연역이 일어나야만 한다는 것이 전제 조건이기에 (2)는 완벽 전달이 실패하며, 컴파일조차 실패할 것이다

- 해결책

앞서 중괄호 초기치의 예와 같이 따로 함수 포인터 지역 변수를 정의한 뒤, 해당 지역 변수를 넘기는 방식으로 진행할 경우 문제없이 동작한다

- 비트 필드

- func의 선언이 void func(const std::size_t size)라고 가정하자
- 또한 IPv6용 헤더를 나타내는 구조체가 존재한다고 하자
- 이러한 상황에서 다음과 같은 코드를 실행한다고 하자

```
struct ipv6hdr {  
#if defined(__LITTLE_ENDIAN_BITFIELD)  
    __u8    priority : 4,  
            version : 4;  
#elif defined(__BIG_ENDIAN_BITFIELD)  
    __u8    version : 4,  
            priority : 4;  
#else  
#error      "Please fix <asm/bytorder.h>"  

```

각각의 경우에 어떤 일이 일어나는지를 살펴보자면 다음과 같이 정리할 수 있다

- (1)의 경우
이 경우 문제없이 동작한다

- (2)의 경우
이 경우에는 애초에 표준에서 '비 const 참조는 비트필드에 묶이지 않는다'라고 명시하고

있다. 또한 애초에 참조는 바이트 단위의 주소값을 가리키는데 비트필드는 말 그대로 비트 단위로 동작하기에 비트의 어디서부터 어디까지를 지칭할 방법 자체가 존재하지 않는다

- **해결책**

앞서 중괄호 초기치의 예와 같이 따로 함수 포인터 지역변수를 정의한 뒤, 해당 지역 변수를 넘기는 방식으로 진행할 경우 문제없이 동작한다

9. 클래스 - 상속

9.1. 파생 클래스

클래스 혹은 구조체를 사용한다는 것은 근본적으로 어떠한 현상 또는 사상을 묘사하고자 한다는 것이다. 개념은 여러 속성을 통해 묘사되기도 하지만, 개념간의 관계를 통해 묘사되기도 한다

[5.5. 파생 클래스]의 하위 요소로 들어가는 설명 중 상속에 관한 설명은 직접적으로 언급하는 경우를 제외하고는 기본적으로 public 상속을 가정하고 있음을 알린다

개념간의 관계에는 크게 두 개의 형태가 있을 수 있다

- 포함 관계(has-a or implemented-in-terms-of)
- 상속 관계(is-a)

포함 관계(has-a or implemented-in-terms-of)

포함 관계는 상위 개념 A가 하위 개념 B를 포함한다는 것이다

포함 관계에는 두 가지 갈래로 나눠지는데, has-a 관계와 implemented-in-terms-of 관계가 그것들이다

has-a

이는 쉽게 말해 프론트엔드쪽에서 발생한 포함 관계를 말한다. 다만, 여기서 말하는 프론트란 외부와 접하는 부분을 말하는 것이다. 즉, 모니터 출력 장치에 보이는 부분 뿐만 아니라 네트워크 연결, DB 쿼리 등등을 일컫는 것이다

개발을 하다보면 프론트엔드에 가까운 코드일수록 눈으로 직관적으로 보이는 대상이나 개념 그 자체를 다루게되는 경우가 많다. 대표적인 예로 UI의 구성요소나 게임 레벨, DB 연결 매니저 등이 있을 수 있다

implemented-in-terms-of

이는 쉽게 말해 백엔드쪽에서 발생한 포함 관계를 말한다. 여기서 말하는 백엔드란 외부와 접하지 않기에 인터페이스 측면에서 쉽게 보이지 않는, 혹은 일부로 감춰놓은 부분들을 말하는 것이다

백엔드에 가까운 코드일수록 겉으로 보이지 않는 보다 추상적인 처리들에 중점을 두게 된다. 대표적인 예로 OS 커널, 메모리 할당자 등등이 있을 수 있다

위의 두 형태는 Effective C++에서 명시된 것이지만 이를 구분하는 기준이 상황에 따라 명확히 구분하기 어려울 수도 있고, 또한 둘이 실질적으로 의미하는 대상이 크게 다른 것도 아니다. 심지어 외부에 노출되어있느냐는 본인의 관점을 어디에 두느냐에 따라서 달라질 수 있는 모호한 잣대이기에 느슨하게 혼용해도 문제되진 않을 것 같다

C++에서 이러한 관계들을 구현하는 방법 또한 두 가지 갈래로 나뉜다

합성

합성(Composition)과 같은 의미로 쓰이는 단어들로는 레이어링(Layering), 포함(Containment), 통합(Aggregation), 내장(Embedding)이 있다

보통 포함 혹은 합성이라 칭하면 `private` 상속이 아닌 여기서 설명할 방법을 말하는 것이다

```
class A {};  
  
class B  
{  
private:  
    A m_a;  
};
```

합성을 통해 포함 관계를 구현하는 방법은 위의 코드와 같이 다른 클래스의 인스턴스를 멤버 변수로서 말 그대로 포함하게 하는 것이다

이러한 방식은 지극히 자연스러운 생각의 흐름이기에 이해하기 매우 쉽다

`private` 상속

보통 상속이라 함은 `is-a` 관계로서 상위 객체의 내용을 하위 객체가 물려받는 것을 말한다. 그러나 C++의 상속 종류 중에서도 `private` 상속은 특이하게도 그 특성상 객체간의 포함 관계를 묘사하게 된다

```
class A {};  
  
class B : private A {};
```

`private` 상속을 간단히 말하자면 상속을 하긴 하되, 상위 객체의 모든 내용을 `private`으로서 물려받게 된다

`private` 상속을 통해 포함 관계를 구현할 때 일반적인 합성을 통한 방식과의 차이점은 다음과 같다

합성과 `private` 상속 모두 하위 객체가 상위 객체의 `public` 영역과 `protected` 영역의 요소들에 접근할 수 있음은 같다. 그러나 차이점이라고 한다면 합성은 단순히 상위 객체의 인스턴스를 멤버로서 포함하는 것 뿐이기에 상위 객체의 가상 함수를 재정의할 수 없다. 그러나 `private` 상속의 경우 상속의 특성상 상위 객체의 가상 함수를 재정의할 수 있다

`private` 상속이 포함 관계를 구현하는데 쓰이는 방법인 중요한 이유는 `private` 상속이 이뤄지면 상위 객체의 내용을 물려받을 때 이 내용을 하위 객체의 `private` 영역 내에 선언된 것으로 취급한다는 것이다. 즉, 만약 B가 A를 `private` 상속을 통해 물려받고, B를 다시 C가 물려받는다고 해도 C는 A의 내용을 단 하나도 접근하지 못한다는 것이다

만약 상속에 있어 특별히 상속 제어를 하지 않는 경우 기본으로 `private` 상속을 한다는 것은 주목할만한 점이다

상속 관계(`is-a`)

`private` 상속을 제외한 `public` 상속과 `protected` 상속은 인터페이스 상속이라고도 한다.
반대로 `private` 상속은 구현 상속이라고도 한다
상속은 일반적으로 상위 객체의 새로운 일종(一種)을 만들어내는 것이 목적이다. 즉, A 객체를 상속받은 B라는 객체는 A의 새로운 아종으로서 동작하게 되는 것이다

```
class A {};  
  
class B : public A {};
```

9.1.1. 파생 클래스

클래스를 상속한다는 것은 상위 객체의 아종을 만든다는 것이고, 따라서 하위 객체는 상위 객체의 일종이 되는 것이다. 즉, 하위 객체는 상위 객체이기도 한 것이다
즉, 이를 코드로서 확인하자면 다음과 같다

```
class A {};  
  
class B : public A {};  
  
int main()  
{  
    A* object{ nullptr }; // (1) 정적 바인딩 : 정적 타입은 A*  
    object = new B();      // (2) 동적 바인딩 : 동적 타입은 B*  
}
```

먼저 클래스간의 관계를 보자면 너무나도 단순명확하다. A는 기반 클래스로서 존재하고 B는 파생 클래스로서 A를 상속받게 되며, 따라서 B는 A의 일종이 된다

다음으로 `main`의 내용을 분석하기 위해서는 먼저 정적 타입과 정적 바인딩 그리고 동적 타입과 동적 바인딩이란 개념이 선제되어야만 한다

정적 타입과 정적 바인딩

정적 타입(Static Type)은 겉으로 보이는 객체의 타입을 말한다. 즉, 만약 `A*`라는 타입이 보인다는 보이는 그대로 이의 정적 타입은 `A*`라는 것이다

정적 바인딩(Static Binding)은 공식적으로는 선행 바인딩(Early Binding)이라고도 하는데, 정적 타입에 정적 시간동안 객체가 묶이는 것을 말하는 것이다. 즉, 쉽게 말해 컴파일 타임에 정적 타입에 코드에 명시되어있는 객체를 묶는 것을 말하는 것이다

쉽게 말해 만약 `A* ptr`이라고 된 코드가 존재한다면 `A*`는 정적 타입이고, `ptr`이 `A*`라는 정적 타입에 정적 바인딩된다는 것이다

동적 타입과 동적 바인딩

동적 타입(Dynamic Type)은 해당 객체가 실제로 무엇이냐에 따라 결정되는 타입을 말한다. 즉, 정적 타입이 A*라고 해도 실제로는 A*와는 다른 무언가를 의미할 수 있고, 실제로 존재하는 객체의 타입이 바로 동적 타입으로서 인식되는 것이다

중요한 점은 동적 타입은 해당 객체가 실제로는 무엇이냐에 따라 결정되기에 실질적인 내용이 존재하지 않는다면 이는 동적 타입이 없는 것으로 간주된다. 대표적으로는 다음과 같은 상황이 있을 수 있다

포인터 정적 타입이 nullptr을 가리키는 경우

```
A* object{ nullptr };
```

이 경우 ptr의 동적 타입은 존재하지 않는다

동적 바인딩(Dynamic Binding)은 공식적으로는 지연 바인딩(Late Binding)이라고도 하는데, 정적 타입에 동적 시간동안 묶이는 것을 말하는 것이다. 즉, 쉽게 말해 런타임에 실질적인 객체가 기존에는 정적 타입이었던 것에 묶이고³⁶, 묶인 객체의 실제 타입이 동적 타입으로서 인식되도록 만드는 것이다

이제 실제로 앞선 코드를 분석해보자면 다음과 같다

```
# A* object{ nullptr }; 분석
```

정적 타입 - A*
동적 타입 - 존재하지 않음

```
# object = new B(); 분석
```

정적 타입 - A*
동적 타입 - B*

9.1.1.1. 멤버 함수

만약 상속 관계에 있는 객체들이 단순히 멤버 변수들만을 쥐고있다면 이는 그다지 유용하지 않을 것이다. 상속의 유용성인 유연한 확장성이 드러나는 것은 상속 관계에 있는 객체들이 멤버 함수를 포함할 때이다

상속 관계에서 기반 클래스의 private이 아닌 멤버 함수는 자동으로 파생 클래스에서도 보이게 된다. 여기서 중요한 점은 파생 클래스 입장에서 기반 클래스의 private 요소들은 보이지 않으며 따라서 접근조차 불가능하다는 것이다. 즉, 다음과 같은 상황이 성립하는 것이다

```
class Base  
{
```

³⁶ 동적 바인딩되고

```

private:
    std::string m_name;
    void Func();
};

class Derived : public Base
{
public:
    void Test()
    {
        //m_name = "Hello";           // Error
        //Func();                   // Error
    }
};

```

9.1.1.2. 생성자와 소멸자

C++에서 상속 관계에서의 생성자와 소멸자와 관련하여 명심해야 할 점들은 다음과 같다

- 객체는 상향식으로 생성되며 하향식으로 소멸된다
- 각 클래스는 자신의 멤버와 기반 클래스를 초기화할 수 있다
- 대개 계층 구조 내의 소멸자는 virtual이어야 한다
- 계층 구조 내에 있는 클래스의 복사 생성자는 복사 손실을 피하기 위해 신중하게 사용되어야 한다
- 가상 함수 호출, dynamic_cast의 해결, 생성자 또는 소멸자 내의 typeid()는 아직 완료되지 않은 객체의 타입보다는 생성과 소멸의 단계를 반영한다

위의 각 요소들은 상속 관계 관점에서 클래스의 동작을 이해하는데도 굉장히 유용하고 또 중요하기에 면밀히 살펴보겠다

객체는 상향식으로 생성되며 하향식으로 소멸된다

예를 들어 다음과 같은 코드가 있다고 하자

```

class Base
{
private:
    std::string m_name;

public:
    virtual ~Base();

    void Func();
};

```

```

class Derived : public Base
{
private:
    int m_num;

public:
    ~Derived() override;

    void Test();
};

int main()
{
    Derived d{};
}

```

먼저 일반적으로 바람직한 생성되고 소멸되는 과정은 다음과 같다

- 생성
 - 가장 상위 클래스의 생성자 호출
 - 가장 상위 클래스의 멤버 변수들의 생성자 호출
 - 다음 상위 클래스의 생성자 호출
 - 다음 상위 클래스의 멤버 변수들의 생성자 호출
 - ...
 - 가장 하위 클래스의 생성자 호출
 - 가장 하위 클래스의 멤버 변수들의 생성자 호출
- 소멸
 - 가장 하위 클래스의 소멸자 호출
 - 가장 하위 클래스의 멤버 변수들의 소멸자 호출
 - 다음 하위 클래스의 소멸자 호출
 - 다음 하위 클래스의 멤버 변수들의 소멸자 호출
 - ...
 - 가장 상위 클래스의 소멸자 호출
 - 가장 상위 클래스의 멤버 변수들의 소멸자 호출

C++의 상속 메커니즘은 위와 같은 과정을 통해 아무리 깊고 복잡한 계층 구조를 가지더라도 빠짐없이 생성하고 소멸할 수 있도록 해준다

여기서 주목해야 할 점은 메모리의 관점에서 파생 클래스는 기반 클래스의 멤버 변수들만큼의 공간만을 더 준비하면 될 뿐이며, 다형성을 사용하지 않는다고 하면 추가적인 메모리 오버헤드가 전혀 발생하지 않는다는 것이다

```

#include <cstdint>

class Base
{
public:

```

```

Base() = default;
virtual ~Base() = default;

void Func() {}
};

class Derived : public Base
{
private:
    int32_t m_num;

public:
    ~Derived() override = default;

    void Test() {};
};

int main()
{
    Base data;
    Derived temp;

    std::cout << sizeof(data) << std::endl; // x64 : 8 || x86 : 4
    std::cout << sizeof(temp) << std::endl; // x64 : 16 || x86 : 8
}

```

또한, 파생 객체의 생성이 상위 클래스부터 차례대로 내려오며, 따라서 각 단계에서 존재하는 것은 각 단계와 그보다 상위 클래스의 요소들 뿐이라는 것이다

이 점은 ‘[가상 함수 호출, dynamic_cast의 해결, 생성자 또는 소멸자 내의 typeid\(\)는 아직 완료되지 않은 객체의 타입보다는 생성과 소멸의 단계를 반영한다](#)’에서 핵심 요소로서 동작하게 된다

각 클래스는 자신의 멤버와 기반 클래스를 초기화할 수 있다

```

class Base
{
private:
    std::string m_name;

public:
    Base(const std::string& name) : m_name(name) {}
    virtual ~Base() = default;
};

class Derived : public Base

```

```

{
public:
    Derived(const std::string& name) : Base(name) {}
    ~Derived() override = default;
};

```

파생 클래스의 생성자 쪽에서는 위와 같이 원하는 기반 클래스의 생성자를 명시적으로 지정해줄 수 있다. 만약 그러지 않을 경우에는 기반 클래스의 기본 생성자가 호출되며, 이는 의도치 않은 코드 크기 증가를 야기할 수도 있다

대개 계층 구조 내의 소멸자는 virtual이어야 한다

```

class Base
{
private:
    std::string m_name;

public:
    Base(const std::string& name) : m_name(name) {}
    virtual ~Base() = default;
};

class Derived : public Base
{
private:
    int m_value;

public:
    Derived(const std::string& name, const int value) : Base(name),
m_value(value) {}
    ~Derived() override = default;
};

int main()
{
    Base* temp = new Derived("temp", 5);
    delete temp;
}

```

위의 코드는 정상적으로 기반 클래스 부분과 파생 클래스 부분의 메모리 전체가 깔끔하게 정리되어 소멸 과정이 이루어지는 코드이다. 이를 확인하기 위해 main 함수 내부를 살펴보자면 다음과 같다

- 소멸자가 virtual일 경우

먼저, 파생 클래스 임시객체를 부모 클래스 포인터에 암시적 캐스팅을 통해 대입하고 있음을 알 수 있다. 즉, 현재 정적 타입은 Base*이고, 동적 타입은 Derived*이다

`delete temp;`를 보자면 이는 결과적으로 `temp`에 의미하는 타입의 소멸자를 호출한다.
중요한 점은 어떤 소멸자를 호출하는가이다

명심할 점은 어떤 클래스 객체의 멤버 함수를 호출할 때, 해당 객체의 정적 타입에 빗대서 해당 함수가 만약 가상 함수가 아니라면 정적 타입에 해당하는 멤버 함수를 호출하고, 만약 해당 함수가 가상 함수라면 동적 타입에 근거하여 V-Table에 검색하여 동적 타입에 알맞은 멤버 함수를 찾아 호출해준다

이러한 점을 놓고 봤을 때, 현재 `temp`의 정적 타입은 `Base*`이다. 즉, 먼저 `Base::~Base()`를 살펴보게 된다. 이때 현재 `Base::~Base()`는 가상 함수이기에 `temp`의 동적 타입에 근거하여 V-Table을 살펴보고 적절한 소멸자를 찾아 호출해준다. `temp`의 동적 타입은 `Derived*`이기에 `Derived::~Derived()`를 찾아 적절히 호출해준다.
그렇게 파생 클래스 쪽이 소멸된 후, 기반 클래스 또한 적절히 소멸되게 된다

- 소멸자가 `virtual`이 아닐 경우

이 경우 `temp`의 정적 타입과 동적 타입은 앞선 경우와 같다. 또한, `temp`의 정적 타입에 해당하는 `Base::~Base()`가 가상 함수인지 여부를 살펴본다. 이 경우에는 해당 소멸자가 가상 함수가 아닌 경우를 가정하기에 결과적으로 `Base::~Base()`를 호출하게 된다.
소멸 과정은 그냥 여기서 끝나게 되며, 파생 클래스 부분에 해당하는 메모리는 전혀 정리되지 못한다

계층 구조 내에 있는 클래스의 복사 생성자는 복사 손실을 피하기 위해 신중하게 사용되어야 한다

파생 클래스의 객체는 기반 클래스 포인터에 의해 참조될 수 있다. 이는 파생 클래스는 기반 클래스의 일종이기 때문이다. 그러나 이 과정에서 복사 손실이 발생할 가능성이 존재한다

```
class Base
{
private:
    std::string m_name;

public:
    Base();
    Base(const Base& target);
    virtual ~Base();
};

class Derived : public Base
{
private:
    int m_value;

public:
    Derived();
    Derived(const Derived& target);
    ~Derived() override;
};
```

```

void Func(const Base* base)
{
    Base b = *base;
}

int main()
{
    Derived derived{};

    Func(&derived);
    Base targetBase{ derived };
}

```

위의 코드에서 main 함수를 살펴보자면 먼저 Derived의 객체가 생성된다
그 후 Func를 호출하는데, Func의 매개변수의 타입은 Base*이며 이때는 복사 손실이
발생하지 않는다. 매개변수가 가리키는 대상은 여전히 main 함수에 존재하는 derived의
메모리 주소이기 때문이다. 그러나 Func 내부에서 d에 대하여 복사 대입 연산자를
호출하면서 복사 손실이 발생하게 된다
다시 main으로 돌아와서 targetBase를 생성할 때 복사 생성자를 호출할 때에도 복사
손실이 발생한다

이와 같은 손실은 의도된 것일 수도 있지만, 만약 그렇지 않을 경우 미묘한 문제로서
남게된다. 만약 이러한 손실이 의도치 않은 것이라면 다음의 방법 중 하나를 구현 함으로써
예방할 수 있다

- 기반 클래스의 복사 생성자와 복사 대입 연산자를 delete한다
- 파생 클래스를 가리키는 포인터에서 기반 클래스를 가리키는 포인터로의 변환을
금지한다. 즉, public 상속이 아닌 protected 상속 또는 private 상속을 구현한다

**# 가상 함수 호출, dynamic_cast의 해결, 생성자 또는 소멸자 내의 typeid()는 아직
완료되지 않은 객체의 타입보다는 생성과 소멸의 단계를 반영한다**

이 내용은 생성과 소멸 과정에서의 파생 계층에 따른 단계적 생성과 소멸 그리고 그 단계적
처리들에서 동적 타입의 취급에 의해 발생되는 이슘이다

```

class Base
{
private:
    std::string m_name;

public:
    Base()
    {
        Func();
    }
    virtual ~Base();

    virtual void Func();
}

```

```

};

class Derived : public Base
{
public:
    ~Derived() override;

    void Func() override;
};

int main()
{
    Derived derived{};
}

```

위의 코드를 작성한 프로그래머의 생각을 유추해보자면 Derived 객체를 생성하며 가변 클래스인 Base의 생성자에 명시된대로 Derived::Func를 호출하고 싶었던 것일 것이다. 그러나 해당 코드는 전혀 그렇게 동작하지 않고 Base::Func를 호출하게 된다

어째서 그렇게 되느냐하면 클래스의 계층 과정에서 차근차근 생성을 해나갈 때, 아직 생성이 되지 않은 부분은 존재조차 하지 않는 것으로 간주하기 때문이다. 엄밀히 말하자면 초기화되지 않은 것은 정의되지 않은 것으로 간주한다

따라서 위의 코드의 경우 Base::Base가 호출될 때에는 아직 Derived::Derived가 호출되지 않았기에 Derived 부분은 정의조차 되지 않은 것으로 간주하고, 해당 시점에서 동적 타입은 Base가 된다. 그렇기에 Func를 호출하고자 하면 Base::Func를 호출할 수 밖에 없던 것이다

이와 같은 문제는 소멸 과정중에도 발생할 수 있는데, 소멸의 경우 생성과 반대로 파생 클래스의 소멸자부터 호출할 때 소멸자가 호출된 부분이 정의조차 되지 않은 것으로 간주되게 된다

이러한 사실 dynamic_cast와 typeid()에도 똑같이 적용되어서 생성 또는 소멸의 과정 중에 이것들이 사용될 경우 해당 순간에 처리중인 생성자 혹은 소멸자에 해당하는 타입으로 해당 객체의 동적 타입이 간주되어 그에 따라 처리된다

9.1.2. 클래스 계층 구조

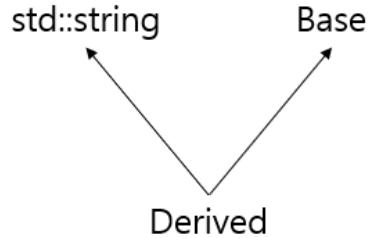
어떤 클래스가 상속 관계를 가질 때 해당 클래스와 상속 관계로서 연결된 전체 클래스의 집합을 클래스 계층 구조라고 한다. 이러한 계층 구조는 다음과 같이 화살표에 의한 다이어그램으로 표현될 수 있다

```

class Base {};

class Derived : public Base, private std::string {};

```



9.1.2.1. 클래스 계층 구조에 유연함을 부여하는 요소

상속을 단순한 단축 표현 그 이상으로 활용하고 싶을 수도 있다. 이러한 기능을 구현하는 방법에는 몇 가지가 있을 수 있다

1. 단일 타입의 객체만을 가리켜지게 한다

이 방법은 템플릿을 활용하는 것이다

```

template <typename _T>
class BaseData
{
private:
    _T m_data;

public:
    virtual ~BaseData() = default;
};

class Data : public BaseData<int>
{
public:
    ~Data() override = default;
};

```

위와 같은 코드에서 Data가 상속받는 BaseData<_T>의 경우 템플릿 인자 타입에 따라 _T를 활용하는 부분이 자동으로 인스턴스화된다
보통 이러한 경우 기반 클래스는 데이터 컨테이너나 그에 준하는 설계일 텐데 넘겨주는 템플릿 인자 타입에 따라 코드 사용성을 극대화하면서도 안전하게 동작을 추구할 가능성이 높다

물론 템플릿 클래스를 상속받는 것은 신중해야만 한다. 특히 주의깊게 봐야할 점은 기반 클래스로서 사용하고 싶은 클래스의 설계가 상속 구조에 적합하게 설계되었는지 여부이다. 간단하게 살펴볼 체크리스트는 다음과 같다

- 기반 클래스의 소멸자는 virtual과 함께 선언되어야만 한다
- 기반 클래스의 특수 멤버 함수 집합은 신중히 작성되어야만 한다
- 기반 클래스 그 자체가 상속 구조를 가진다면 해당 계층 구조는 건전하게

- 구축되어 있어야만 한다
- 기반 클래스는 적절한 가상 함수 집합을 포함하는 것이 바람직하다

2. 파생 계층 내의 함수 내에서 활용할 타입 필드를 기반 클래스에 둔다

```

class Base
{
public:
    enum Type{BaseT, DerivedT};

    Type m_type;
    int m_num;

public:
    Base() : m_type(Type::BaseT) {}
    virtual ~Base() = default;
};

class Derived : public Base
{
public:
    std::string m_name;

public:
    Derived() { m_type = Base::Type::DerivedT; }
    ~Derived() override = default;
};

void PrintInfo(const Base* data)
{
    switch (data->m_type)
    {
        case Base::Type::BaseT:
            std::cout << data->m_num << std::endl;
            break;

        case Base::Type::DerivedT:
            std::cout << data->m_num << std::endl;

            const Derived* derivedData{dynamic_cast<const
Derived*>(data)};
            std::cout << derivedData->m_num << derivedData->m_name <<
std::endl;
            break;
    }
}

```

```
    }  
}
```

위와 같이 해당 클래스가 계층 구조에 따라 혹은 계층 구조와는 다른 의미 체계에 따라 어느 위치에 해당하는지를 의미하는 타입 필드를 멤버 변수로서 포함함으로써 이를 활용하는 측에 유연성을 공급할 수 있다

이 방법은 극단적인 장단점이 존재할 수 있다

장점

- 가상 함수와 같이 언어 차원에서 지원하는 기능을 활용하면 계층 구조에 따른 유연성을 구현할 수 있다. 그러나 타입 필드를 활용할 경우 계층 구조를 초월하여 임의의 의미 체계를 적용하여 그에 따른 유연성을 구현할 가능성이 있다
- 여담이지만 C와 같이 언어 차원에서 객체 지향을 지원하지 않는 언어에서 클래스를 구현할 때 이 방법으로 다음과 같이 다형성을 구현할 수 있다

```
#include <stdio.h>  
  
typedef enum Type { BaseT, DerivedT } Type;  
  
typedef struct Base  
{  
    Type m_type;  
    int m_num;  
}Base;  
  
typedef struct Derived  
{  
    Base* m_basePtr;  
  
    Type m_type;  
    char m_char;  
}Derived;  
  
void ExecutePolymorphism(Type type, void* objectPtr)  
{  
    switch (type)  
    {  
        case BaseT:  
        {  
            Base* basePtr = (Base*)objectPtr;  
            printf("%d\n", basePtr->m_num);  
            break;  
        }  
    }  
}
```

```

case DerivedT:
{
    Derived* derivedPtr = (Derived*)objectPtr;
    printf("%d %c\n", derivedPtr->m_basePtr->m_num,
derivedPtr->m_char);
    break;
}
}

int main()
{
    Base base;
    base.m_type = BaseT;
    base.m_num = 1;

    Derived derived;
    derived.m_type = DerivedT;
    derived.m_basePtr = &base;
    derived.m_char = 'a';

    ExcutePolymorphism(base.m_type, &base);           // 1
    ExcutePolymorphism(derived.m_type, &derived);     // 1 a

    return 0;
}

```

단점

- 이 방법은 다시 말하자면 언어 차원에서 계층 구조에 따라 지원해주던 다형성을 개발자가 직접 구현하는 것이다. 즉, 계층 구조에 변경이 가해질 경우 모든 것이 개발자에 의해 갱신되어야만 한다는 것이며, 이러한 임의의 다형성을 구현하는 함수 집합이 확장될수록 코드의 유지보수가 비례하도록 어려워질 것이다

3. dynamic_cast를 활용한다

클래스의 파생 관계에서 자식 클래스는 부모 클래스의 일종이다. 따라서 자식 클래스 포인터는 기반 클래스 포인터가 가리킬 수 있으며, 이는 매우 자연스러운 동작이다

그러나 역으로 자식 클래스 포인터가 부모 클래스 포인터를 가리키는 것은 보통 자식 클래스의 내부 자료구조가 기반 클래스에 비해 확장되기에 실수로 자식 클래스에만 존재하는 멤버에 접근하고자 할 경우 읽기 액세스 위반으로 예외가 발생하게 된다

```

class Base
{

```

```

public:
    virtual ~Base() = default;
};

class Derived : public Base
{
public:
    std::string m_name;

public:
    ~Derived() override = default;
};

int main()
{
    Derived data;

    Base* ptr = dynamic_cast<Base*>(&data);
    //std::cout << ptr->m_name;           // Error
}

```

4. 가상 함수를 활용한다

가상 함수는 클래스 계층 구조에 따라 다형성을 구현하기 위해 언어 차원에서 지원하는 기본적인 방법이다

```

class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func();
};

class Data : public BaseData
{
public:
    ~Data() override = default;

    void Func() override;
};

```

9.1.2.1.1. 타입 필드

타입 필드는 앞서 언급했듯 원시적으로 클래스 계층 구조에 다형성이란 유연성을 공급하는 구현 방법 중 하나이다. 따라서 심지어 타입 필드는 언어 차원에서 객체 지향을 지원하지 않는 C에서도 다형성을 구현할 수 있도록 해준다

타입 필드의 보다 자세한 사항은 앞선 내용을 참조하길 바란다

9.1.2.1.2. 가상 함수

가상 함수는 기반 클래스에 선언된 가상 혹은 순수 가상 함수의 선언부가 해당 클래스로부터 파생된 클래스에 존재하는 동일한 이름의 함수에 대한 인터페이스가 되도록 할 수 있다

- **가상 함수의 종류와 사용법**

가상 함수에는 일반적인 가상 함수와 순수 가상 함수가 존재한다

일반적인 가상 함수

일반적인 가상 함수는 클래스 내의 멤버 함수에 `virtual` 키워드를 추가함으로써 선언될 수 있다

```
class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func();
};

class Data : public BaseData
{
public:
    ~Data() = default;

    void Func();
};

int main()
{
    BaseData data{Data{}};
    data.Func();
}
```

위와 같은 경우 `main` 함수 내의 `data`는 정적 타입은 `BaseData`이지만 동적 타입이 `Data`이기에 `data.Func()`에서 `Func`는 `BaseData::Func`가 아닌 `Data::Func`가 선택되어 호출된다. 다시 말해 다형성이 발휘된다

그러나 주의할 점은 `virtual`이라는 키워드의 실질적인 의미는 문맥에 따라 그 유무가 결정될 수 있다는 것이다

```

class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func();
};

class Data : public BaseData
{
public:
    ~Data() = default;

    virtual void Func();
};

```

기존의 코드와 달라진 점은 Data::Func에 virtual이 추가된 것 뿐이다. 그렇다면 BaseData::Func와 Data::Func는 이름은 같지만 각각 다른 함수를 가상 함수로서 선언하는 것인가 착각할 수 있다

이 경우 Data::Func에 붙은 virtual가 어떻게 해석되는지는 Data가 포함된 계층 구조에 속한 클래스들의 설계에 따라 의미가 크게 달라질 수 있다

만약 상위 계층의 클래스에 Data::Func와 같은 시그니처의 가상 함수가 존재 O

이 경우에는 Data::Func에 붙은 virtual은 아무런 의미도 지니지 않으며, 따라서 virtual이 붙지 않은 경우와 완벽히 동일하다

만약 상위 계층의 클래스에 Data::Func와 같은 시그니처의 가상 함수가 존재 X

이 경우에는 Data::Func는 새로운 가상 함수를 선언한 것이며, 따라서 virtual이 붙지 않은 경우와 달리 클래스 내부 자료구조와 다형성 구현 사항에 변화를 주는 것을 실질적으로 결정하는 역할을 한다

virtual 키워드는 프로그램 전체에 일관적인 규칙을 통해 사용된다면 매번 virtual을 사용하던, 필요할 때만 사용하던 큰 오해를 불러 일으키지 않을 수 있으며, 이 정도는 개인 취향의 문제이다. 그러나 그때그때 아무렇게나 일관성 없이 사용한다면 코드 독해에 불편함을 줄 수 있기에 주의할 필요가 있다

순수 가상 함수

순수 가상 함수는 일반적인 가상 함수와 같이 다형성을 발휘할 수 있도록 해준다. 그러나 그 과정에서 파생 관계에 있는 클래스들에 몇몇 제약을 가한다는 점에서 그런 제약이 존재하지 않고 자유로운 일반적인 가상 함수와 구분된다

다음의 목록이 순수 가상 함수에 가해지는 제약들이다

- 기반 클래스에 존재하는 순수 가상 함수는 선언만이 존재하고 본문이 존재할 수 없다. 즉, 정의될 수 없다

- 파생 클래스는 기반 클래스의 순수 가상 함수를 반드시 재정의해야만 한다
- 순수 가상 함수가 선언된 클래스는 그 자체로서 인스턴스화될 수 없다

```

class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func() = 0;
};

class Data : public BaseData
{
public:
    ~Data() = default;

    void Func();
};

int main()
{
    //BaseData baseData;      // Error
    Data data;

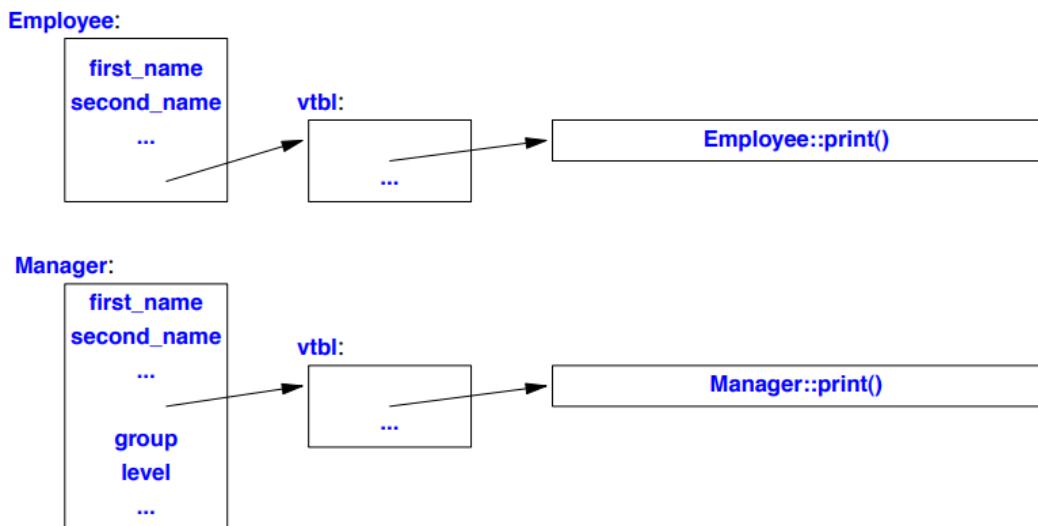
    BaseData* baseDataPtr = &data;
}

```

위의 코드를 보면 `BaseData::Func`의 선언 끝에 `= 0`이 붙은 것을 볼 수 있다. 이는 해당 함수에 정의가 존재할 수 없다는 것을 보여주며, 따라서 `BaseData::Func`를 순수 가상 함수로서 선언하는 것이다
 주목할 것은 `main`에 있다. `BaseData`와 관련된 코드를 보면 `BaseData`를 그 자체로서 인스턴스화하는 것은 불가능하다. 그러나 `BaseData*`로서 인스턴스화되어 사용되는 것은 가능하다

- 다형성의 동작 원리

만약 어떤 클래스가 다형성을 구현하게 된다는 말은 해당 클래스와 다형성으로 관련된 모든 클래스를 포함한 가상 함수 테이블(Virtual Function Table, V-Table, vtbl)이 보이지 않는 뒷면에 생성하여 다형성 해결 처리를 수행한다는 것과 같은 말이다



위의 그림은 보통 다형성을 구현하는 클래스를 묘사할 때의 전형적인 그림이다. 이를 요소요소별로 차근차근 설명하자면 다음과 같다

먼저 클래스 그 자체로서 보자면 클래스에 다형성 설계가 개입되는 순간 겉으로 보이진 않지만 클래스의 내부 자료구조에 포인터 하나가 추가된다. 이는 가상 함수가 포함되었을 때와 그렇지 않은 때의 크기를 비교해보면 명확히 드러난다

```

class Sample1
{
private:
    int m_num;

public:
    void FUnc();
};

class Sample2
{
private:
    int m_num;

public:
    virtual void Func();
};

int main()
{
    std::cout << sizeof(Sample1) << std::endl; // x64 : 4 || x86 : 4
    std::cout << sizeof(Sample2) << std::endl; // x64 : 16 || x86 : 8
}

```

위의 Sample1과 Sample2는 각각의 함수에 포함된 Func 멤버 함수에 virtual이 여부만이 다르다. 그럼에도 클래스의 크기를 보자면 차이를 보이는 것을 알 수 있다

이렇게 추가된 포인터는 가상 함수 테이블을 가리킨다

Visual C++

아래의 예는 Visual C++ 143 /Od로 빌드했다

```
#include <iostream>
#include <string>

class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func() { std::cout << "base func"; }
    virtual void Test() { std::cout << "base test"; }
};

class Data : public BaseData
{
public:
    ~Data() override = default;

    void Func() override { std::cout << "der func"; }
    void Test() override { std::cout << "der test"; }
};

int main()
{
    BaseData baseData;
    Data data;

    baseData.Func();
    data.Func();

    baseData.Test();
    data.Test();
}
```

```
--- ...\\main.cpp -----
```

```

        virtual void Func() { std::cout << "base func"; }
00007FF6E3CD1000  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1005  sub      rsp,28h
00007FF6E3CD1009  lea      rdx,[string "base func" (07FF6E3CD32E0h)]
00007FF6E3CD1010  mov      rcx,qword ptr [_imp_std::cout (07FF6E3CD30E8h)]
00007FF6E3CD1017  call     std::operator<<<std::char_traits<char> >
(07FF6E3CD12D0h)
00007FF6E3CD101C  add      rsp,28h
00007FF6E3CD1020  ret
--- 소스 파일이 없습니다.

00007FF6E3CD1021  int     3
00007FF6E3CD1022  int     3
00007FF6E3CD1023  int     3
00007FF6E3CD1024  int     3
00007FF6E3CD1025  int     3
00007FF6E3CD1026  int     3
00007FF6E3CD1027  int     3
00007FF6E3CD1028  int     3
00007FF6E3CD1029  int     3
00007FF6E3CD102A  int     3
00007FF6E3CD102B  int     3
00007FF6E3CD102C  int     3
00007FF6E3CD102D  int     3
00007FF6E3CD102E  int     3
00007FF6E3CD102F  int     3
--- ...\\main.cpp -----
        virtual void Test() { std::cout << "base test"; }
00007FF6E3CD1030  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1035  sub      rsp,28h
00007FF6E3CD1039  lea      rdx,[string "base test" (07FF6E3CD32F0h)]
00007FF6E3CD1040  mov      rcx,qword ptr [_imp_std::cout (07FF6E3CD30E8h)]
00007FF6E3CD1047  call     std::operator<<<std::char_traits<char> >
(07FF6E3CD12D0h)
00007FF6E3CD104C  add      rsp,28h
00007FF6E3CD1050  ret
--- 소스 파일이 없습니다.

00007FF6E3CD1051  int     3
00007FF6E3CD1052  int     3
00007FF6E3CD1053  int     3
00007FF6E3CD1054  int     3
00007FF6E3CD1055  int     3
00007FF6E3CD1056  int     3
00007FF6E3CD1057  int     3
00007FF6E3CD1058  int     3
00007FF6E3CD1059  int     3
00007FF6E3CD105A  int     3
00007FF6E3CD105B  int     3
00007FF6E3CD105C  int     3
00007FF6E3CD105D  int     3
00007FF6E3CD105E  int     3
00007FF6E3CD105F  int     3
--- ...\\main.cpp -----

```

```

void Func() override { std::cout << "der func"; }
00007FF6E3CD1060  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1065  sub      rsp,28h
00007FF6E3CD1069  lea      rdx,[string "der func" (07FF6E3CD3300h)]
00007FF6E3CD1070  mov      rcx,qword ptr [_imp_std::cout (07FF6E3CD30E8h)]
00007FF6E3CD1077  call     std::operator<<<std::char_traits<char> >
(07FF6E3CD12D0h)
00007FF6E3CD107C  add      rsp,28h
00007FF6E3CD1080  ret
--- 소스 파일이 없습니다.

00007FF6E3CD1081  int     3
00007FF6E3CD1082  int     3
00007FF6E3CD1083  int     3
00007FF6E3CD1084  int     3
00007FF6E3CD1085  int     3
00007FF6E3CD1086  int     3
00007FF6E3CD1087  int     3
00007FF6E3CD1088  int     3
00007FF6E3CD1089  int     3
00007FF6E3CD108A  int     3
00007FF6E3CD108B  int     3
00007FF6E3CD108C  int     3
00007FF6E3CD108D  int     3
00007FF6E3CD108E  int     3
00007FF6E3CD108F  int     3
--- ...\\main.cpp -----
void Test() override { std::cout << "der test"; }
00007FF6E3CD1090  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1095  sub      rsp,28h
00007FF6E3CD1099  lea      rdx,[string "der test" (07FF6E3CD3310h)]
00007FF6E3CD10A0  mov      rcx,qword ptr [_imp_std::cout (07FF6E3CD30E8h)]
00007FF6E3CD10A7  call     std::operator<<<std::char_traits<char> >
(07FF6E3CD12D0h)
00007FF6E3CD10AC  add      rsp,28h
00007FF6E3CD10B0  ret
--- 소스 파일이 없습니다.

00007FF6E3CD10B1  int     3
00007FF6E3CD10B2  int     3
00007FF6E3CD10B3  int     3
00007FF6E3CD10B4  int     3
00007FF6E3CD10B5  int     3
00007FF6E3CD10B6  int     3
00007FF6E3CD10B7  int     3
00007FF6E3CD10B8  int     3
00007FF6E3CD10B9  int     3
00007FF6E3CD10BA  int     3
00007FF6E3CD10BB  int     3
00007FF6E3CD10BC  int     3
00007FF6E3CD10BD  int     3
00007FF6E3CD10BE  int     3
00007FF6E3CD10BF  int     3
--- ...\\main.cpp -----
// ...

```

```

};

int main()
{
00007FF6E3CD10C0  sub      rsp,48h
00007FF6E3CD10C4  mov      rax,qword ptr [__security_cookie (07FF6E3CD5008h)]
00007FF6E3CD10CB  xor      rax,rs
00007FF6E3CD10CE  mov      qword ptr [rsp+30h],rax
    BaseData baseData;
00007FF6E3CD10D3  lea      rcx,[baseData]
00007FF6E3CD10D8  call     BaseData::BaseData (07FF6E3CD1140h)
00007FF6E3CD10DD  nop
    Data data;
00007FF6E3CD10DE  lea      rcx,[data]
00007FF6E3CD10E3  call     Data::Data (07FF6E3CD1170h)
00007FF6E3CD10E8  nop

    baseData.Func();
00007FF6E3CD10E9  lea      rcx,[baseData]
00007FF6E3CD10EE  call     BaseData::Func (07FF6E3CD1000h)
    data.Func();
00007FF6E3CD10F3  lea      rcx,[data]
00007FF6E3CD10F8  call     Data::Func (07FF6E3CD1060h)

    baseData.Test();
00007FF6E3CD10FD  lea      rcx,[baseData]
00007FF6E3CD1102  call     BaseData::Test (07FF6E3CD1030h)
    data.Test();
00007FF6E3CD1107  lea      rcx,[data]
00007FF6E3CD110C  call     Data::Test (07FF6E3CD1090h)
00007FF6E3CD1111  nop
}
00007FF6E3CD1112  lea      rcx,[data]
00007FF6E3CD1117  call     Data::~Data (07FF6E3CD11A0h)
00007FF6E3CD111C  nop
00007FF6E3CD111D  lea      rcx,[baseData]
00007FF6E3CD1122  call     BaseData::~BaseData (07FF6E3CD1160h)
00007FF6E3CD1127  xor      eax,eax
00007FF6E3CD1129  mov      rcx,qword ptr [rsp+30h]
00007FF6E3CD112E  xor      rcx,rs
00007FF6E3CD1131  call     __security_check_cookie (07FF6E3CD18A0h)
00007FF6E3CD1136  add      rsp,48h
00007FF6E3CD113A  ret

--- 소스 파일이 없습니다.

-----
00007FF6E3CD113B  int     3
00007FF6E3CD113C  int     3
00007FF6E3CD113D  int     3
00007FF6E3CD113E  int     3
00007FF6E3CD113F  int     3
Sandbox.exe!BaseData::BaseData(void):
00007FF6E3CD1140  mov      qword ptr [this],rcx
00007FF6E3CD1145  mov      rax,qword ptr [this]
00007FF6E3CD114A  lea      rcx,[BaseData::'vftable' (07FF6E3CD3328h)]
00007FF6E3CD1151  mov      qword ptr [rax],rcx

```

```

00007FF6E3CD1154  mov      rax,qword ptr [this]
00007FF6E3CD1159  ret
00007FF6E3CD115A  int     3
00007FF6E3CD115B  int     3
00007FF6E3CD115C  int     3
00007FF6E3CD115D  int     3
00007FF6E3CD115E  int     3
00007FF6E3CD115F  int     3
--- ...\main.cpp -----
#include <iostream>
#include <string>

class BaseData
{
public:
    virtual ~BaseData() = default;
00007FF6E3CD1160  mov      qword ptr [this],rcx
00007FF6E3CD1165  ret
--- 소스 파일이 없습니다.

-----
00007FF6E3CD1166  int     3
00007FF6E3CD1167  int     3
00007FF6E3CD1168  int     3
00007FF6E3CD1169  int     3
00007FF6E3CD116A  int     3
00007FF6E3CD116B  int     3
00007FF6E3CD116C  int     3
00007FF6E3CD116D  int     3
00007FF6E3CD116E  int     3
00007FF6E3CD116F  int     3
Sandbox.exe!Data::Data(void):
00007FF6E3CD1170  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1175  sub      rsp,28h
00007FF6E3CD1179  mov      rcx,qword ptr [this]
00007FF6E3CD117E  call    BaseData::BaseData (07FF6E3CD1140h)
00007FF6E3CD1183  mov      rax,qword ptr [this]
00007FF6E3CD1188  lea      rcx,[Data::'vftable' (07FF6E3CD3488h)]
00007FF6E3CD118F  mov      qword ptr [rax],rcx
00007FF6E3CD1192  mov      rax,qword ptr [this]
00007FF6E3CD1197  add      rsp,28h
00007FF6E3CD119B  ret
00007FF6E3CD119C  int     3
00007FF6E3CD119D  int     3
00007FF6E3CD119E  int     3
00007FF6E3CD119F  int     3
--- ...\main.cpp -----
    // ...
};

class Data : public BaseData
{
public:
    ~Data() = default;
00007FF6E3CD11A0  mov      qword ptr [rsp+8],rcx
00007FF6E3CD11A5  sub      rsp,28h

```

```

00007FF6E3CD11A9  mov      rcx,qword ptr [this]
00007FF6E3CD11AE  call     BaseData::~BaseData (07FF6E3CD1160h)
00007FF6E3CD11B3  add      rsp,28h
00007FF6E3CD11B7  ret
--- 소스 파일이 없습니다.

-----
00007FF6E3CD11B8  int      3
00007FF6E3CD11B9  int      3
00007FF6E3CD11BA  int      3
00007FF6E3CD11BB  int      3
00007FF6E3CD11BC  int      3
00007FF6E3CD11BD  int      3
00007FF6E3CD11BE  int      3
00007FF6E3CD11BF  int      3
Sandbox.exe!BaseData::'scalar deleting destructor'(unsigned int):
00007FF6E3CD11C0  mov      dword ptr [rsp+10h],edx
00007FF6E3CD11C4  mov      qword ptr [rsp+8],rcx
00007FF6E3CD11C9  sub      rsp,28h
00007FF6E3CD11CD  mov      rcx,qword ptr [this]
00007FF6E3CD11D2  call    BaseData::~BaseData (07FF6E3CD1160h)
00007FF6E3CD11D7  mov      eax,dword ptr [rsp+38h]
00007FF6E3CD11DB  and      eax,1
00007FF6E3CD11DE  test    eax,eax
00007FF6E3CD11E0  je      BaseData::'scalar deleting destructor'+31h
(07FF6E3CD11F1h)
00007FF6E3CD11E2  mov      edx,8
00007FF6E3CD11E7  mov      rcx,qword ptr [this]
00007FF6E3CD11EC  call    operator delete (07FF6E3CD18C0h)
00007FF6E3CD11F1  mov      rax,qword ptr [this]
00007FF6E3CD11F6  add      rsp,28h
00007FF6E3CD11FA  ret
00007FF6E3CD11FB  int      3
00007FF6E3CD11FC  int      3
00007FF6E3CD11FD  int      3
00007FF6E3CD11FE  int      3
00007FF6E3CD11FF  int      3
Sandbox.exe!Data::'scalar deleting destructor'(unsigned int):
00007FF6E3CD1200  mov      dword ptr [rsp+10h],edx
00007FF6E3CD1204  mov      qword ptr [rsp+8],rcx
00007FF6E3CD1209  sub      rsp,28h
00007FF6E3CD120D  mov      rcx,qword ptr [this]
00007FF6E3CD1212  call    Data::~Data (07FF6E3CD11A0h)
00007FF6E3CD1217  mov      eax,dword ptr [rsp+38h]
00007FF6E3CD121B  and      eax,1
00007FF6E3CD121E  test    eax,eax
00007FF6E3CD1220  je      Data::'scalar deleting destructor'+31h (07FF6E3CD1231h)
00007FF6E3CD1222  mov      edx,8
00007FF6E3CD1227  mov      rcx,qword ptr [this]
00007FF6E3CD122C  call    operator delete (07FF6E3CD18C0h)
00007FF6E3CD1231  mov      rax,qword ptr [this]
00007FF6E3CD1236  add      rsp,28h
00007FF6E3CD123A  ret
00007FF6E3CD123B  int      3
00007FF6E3CD123C  int      3
00007FF6E3CD123D  int      3

```

```
00007FF6E3CD123E int      3
00007FF6E3CD123F int      3
```

이름	값	형식
baseData	{...}	BaseData
_vptr	0x00007ff6e3cd3328 {Sandbox.exe!void(*`BaseData::`vtable`[4])() {0x00007ff6e3cd11c0 {Sandbox.exe!BaseData::`scalar deleting destructor`(`unsigned int`); ...}}	void**
[0]	0x00007ff6e3cd11c0 {Sandbox.exe!BaseData::`scalar deleting destructor`(`unsigned int`); ...}	void*
[1]	0x00007ff6e3cd1000 {Sandbox.exe!BaseData::Func(void)}	void*
[2]	0x00007ff6e3cd1030 {Sandbox.exe!BaseData::Test(void)}	void*

GCC

아래의 예는 g++ -m32 -fno-rtti -fno-exceptions -O1 file.cpp로 빌드했다

```
#include <iostream>
#include <cstdlib>

struct Mammal
{
    Mammal() { std::cout << "Mammal::Mammal\n"; }
    virtual ~Mammal() { std::cout << "Mammal::~Mammal\n"; }
    virtual void run() = 0;
    virtual void walk() = 0;
    virtual void move() { walk(); }
};

struct Cat : Mammal
{
    Cat() { std::cout << "Cat::Cat\n"; }
    ~Cat() override { std::cout << "Cat::~Cat\n"; }
    void run() override { std::cout << "Cat::run\n"; }
    void walk() override { std::cout << "Cat::walk\n"; }
};

struct Dog : Mammal
{
    Dog() { std::cout << "Dog::Dog\n"; }
    ~Dog() override { std::cout << "Dog::~Dog\n"; }
    void run() override { std::cout << "Dog::run\n"; }
    void walk() override { std::cout << "Dog::walk\n"; }
};
```

```

idata:080489EB aCatCat_0          db 'Cat::~Cat',0Ah,0      ; DATA XREF: Cat_destruc
idata:080489F6 aDogDog_0          db 'Dog::~Dog',0Ah,0      ; DATA XREF: Dog_destruc
idata:08048A01 aCatCat           db 'Cat::Cat',0Ah,0       ; DATA XREF: Cat_Cat+171
idata:08048A0B aDogDog           db 'Dog::Dog',0Ah,0       ; DATA XREF: Dog_Dog+171
idata:08048A15
idata:08048A20 Mammal_vtable    align 10h
idata:08048A20                  dd 0                         ; DATA XREF: Mammal_Mamm
idata:08048A20                  ; sub_80487F6+7^o
idata:08048A24
idata:08048A28
idata:08048A2C
idata:08048A30
idata:08048A34
idata:08048A35
idata:08048A36
idata:08048A37
idata:08048A38
idata:08048A39
idata:08048A3A
idata:08048A3B
idata:08048A3C Cat_vtable        dd 0
idata:08048A3C                  dd offset __cxa_pure_virtual
idata:08048A40
idata:08048A44
idata:08048A48
idata:08048A4C
idata:08048A50
idata:08048A51
idata:08048A52
idata:08048A53
idata:08048A54
idata:08048A55
idata:08048A56
idata:08048A57
idata:08048A58 Dog_vtable        dd 0
idata:08048A58                  dd offset __cxa_pure_virtual
idata:08048A5C
idata:08048A60
idata:08048A64
idata:08048A68
idata:08048A68 _rodata         dd offset __cxa_pure_virtual
idata:08048A68                  dd offset Mammal_move
idata:08048A68                  ends

```

왼쪽에 잘린 영어는 .rodata이며, 이는 .text 영역의 하위 집합으로서 주로 포인터 관련된 코드들이 위치하게 된다

위의 예에서 알 수 있듯 V-Table을 구현하는 방법은 표준에서 정의하지 않기에 사용하는 컴파일러마다 상이함을 알 수 있다

Visual C++과 GCC의 V-Table 구현 방법을 비교해보자면 확연한 차이를 보임을 알 수 있다

- Visual C++의 경우 다형성을 구현하는 클래스의 인스턴스는 각각 _vfptr의 2차원 포인터를 내부적으로 가진다. _vfptr의 각 원소는 실제 함수의 시작 Instruction 주소를 담고 있다
- GCC의 경우 .rodata 영역에 V-Table을 기록하며, 각 가상 함수들에 대한 오프셋을 지정하는 방식으로 구현한다

9.1.2.1.3. 명시적 한정

```

class BaseData
{
public:

```

```

    virtual ~BaseData() = default;

    virtual void Func();
};

class Data : public BaseData
{
public:
    ~Data() override = default;

    void Func() override
    {
        BaseData::Func();
    }
    void Test()
    {
        BaseData::Func();
    }
};

int main()
{
    Data data;
    data.Func();
    data.Test();
}

```

위에서와 같이 어떤 멤버 함수 내에서 특정 가상 함수를 네임스페이스까지 포함하여 명시적으로 한정하여 호출하면 가상 함수의 메커니즘에서 벗어나 일반적인 함수 호출과 같이 작동한다. 즉, main 함수 내의 data.Func와 data.Test 모두 실질적으로 BaseData::Func를 호출한다
만약 이러한 명시적 한정이 동작하지 않았다면 Data::Func는 재귀로서 무한루프에 갇히게 될 것이다

9.1.2.1.4. 재정의 제어

재정의, 즉, 오버라이딩은 오버로딩과 비슷한 느낌으로 인해 헷갈리기 쉽다. 그러나 오버로딩의 경우 함수의 시그니처가 완전히 같다면 오버로딩 해결이 불가하기에 컴파일이 불가능하지만, 오버라이딩의 경우에는 함수의 시그니처가 완전히 같고 오버라이딩이 실패하더라도 이름을 가려버릴 뿐이지 컴파일과 실행은 정상적으로 된다. 그저 개발자가 의도하던 것과 전혀 다르게 동작할 뿐이다

- **다형성 override**

정상적으로 오버라이딩이 되기 위해서는 다음의 조건들을 만족해야만 한다

- | |
|---|
| 1) 파생 클래스의 어떤 상위 클래스에 가상 혹은 순수 가상 함수 Func의 정의가 존재해야만 한다 |
|---|

- 2) 파생 클래스의 오버라이딩 함수는 어떤 상위 클래스에 선언된 가상 혹은 순수 가상 함수와 시그니처가 동일해야만 한다

즉, 다음과 같은 코드가 성립하게 되는 것이다

```
class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func() {}

};

class A : public BaseData {};
class B : public A {};
class C : public B {};
class D : public C {};
class E : public D {};

class Data : public E
{
public:
    ~Data() override = default;

    virtual void Func() {}
    virtual void Test() {}

};
```

위의 코드에서 Data에 존재하는 두 함수인 Data::Func와 Data::Test를 분석해보면 파생 클래스의 어떤 함수가 오버라이딩 함수인지를 검증하는 것이 상당히 귀찮은 작업임을 알 수 있다

먼저, Data::Func부터 보자면 Data의 파생 계층을 거슬러 올라가 가장 위쪽에 위치하는 BaseData에 함수 시그니처가 일치하는 가상 함수인 Func가 존재한다. 따라서 Data::Func는 오버라이딩 함수임이 분명하다. 이때 붙은 virtual은 생략되어도 무관하다

다음으로 Data::Test를 보면 파생 계층에서 상위의 클래스 중 내부 설계가 보이는 BaseData에는 함수 시그니처가 일치하는 가상 함수인 Test가 존재하지 않는다. 그러나 내부 설계를 확인할 수 없는 A, B, C, D, E 중 어딘가에 오버라이딩 조건을 만족하는 Test가 존재할 가능성이 전혀 없다고 단언할 수는 없는 일이다. 따라서 만약 A~E 중 어딘가에 적절한 Test가 존재한다면 Data::Test는 오버라이딩 함수가 될 것이다. 그러나 A~E 중 어디에도 적절한 가상 함수가 존재하지 않는다면 Data::Test에 붙은 virtual 키워드는 실질적인 역할을 가지고 새로운 가상 함수 선언으로서 Data로부터 파생되는 클래스에 인터페이스로서 의미를 가지게 될 것이다

- 이름 가리기 *override*

만약 오버라이딩 조건을 어중간하게 달성할 경우 파생 클래스의 해당 함수는 오버로딩 함수가 되거나 단순히 기반 클래스의 함수를 가려버리게 된다

이름을 가린다는 것은 오버로딩과는 아주 다르다. 오버로딩의 경우 동일한 이름의 오버로딩 집합을 구성하여 같은 이름으로 함수가 동시에 여럿 존재하는 것인데, 이름을 가리게 되면 기반 클래스의 함수를 잡아먹어버려 파생 클래스의 입장에선 파생 클래스에 선언된 해당 함수만이 존재하게 된다

오버라이딩의 2번 조건만을 만족한다면 파생 클래스의 해당 함수는 이름 가리기가 될 가능성이 다분하다

오버라이딩의 1번 조건만을 만족한다면 파생 클래스의 해당 함수는 오버로딩이 될 가능성이 다분하다

```
class BaseData
{
public:
    virtual ~BaseData() = default;

    virtual void Func(int) {}
    void Test() {}

class A : public BaseData {};
class B : public A {};
class C : public B {};
class D : public C {};
class E : public D {};

class Data : public E
{
public:
    ~Data() override = default;

    void Func(double) {}
    void Test() {}

};
```

먼저 Data::Func부터 살펴보면 이 함수는 오버라이딩 함수 혹은 오버로딩 함수로서 존재하게 된다

만약 A ~ E까지의 클래스 중 어느 하나에 가상 함수 Data::Func의 시그니처와 일치하는 멤버 함수의 선언이 존재한다면 Data::Func는 오버라이딩 함수일 것이다. 그러나 만약 A ~ E 중 하나에 적절한 가상 함수가 존재하지 않는다면 Data::Func는 BaseData::Func의 오버로딩 함수가 된다

다음으로 Data::Test의 경우는 조금 복잡하다. 이 경우에는 이름 가리기와 오버라이딩이 동시에 발생하거나, 혹은 단순히 이름 가리기만 발생하게 된다
만약 A ~ E까지의 클래스 중 어느 하나에 가상 함수 Data::Test의 시그니처와 일치하는 멤버 함수의 선언이 존재한다면 Data::Test는 해당 함수의 오버라이딩 함수가 되고 동시에 BaseData::Test는 가리게 된다. 그러나 만약 A ~ E 중 하나에 적절한 가상 함수가 존재하지 않는다면 Data::Test는 단순히 BaseData::Test를 가리게 될 뿐이다

- override 키워드

앞선 다형성과 관련된 다양한 문제, 예컨대 의도치 않은 오버로딩과 이름 가리기 그리고 컴파일 에러의 문제점은 단순하게 보자면 파생 클래스에 선언된 함수가 과연 오버라이딩을 위해 만들어진 것인지 아닌지 하는 의도를 컴파일러가 알 수 없기 때문이다

파생 클래스의 어떤 함수를 오버라이딩할 것이라는 개발자의 암시적인 의도를 명시적으로 드러내 컴파일러가 알 수 있도록 해주는 것이 override 키워드이다

```
class BaseData
{
public:
    virtual ~BaseData() = default;

    void Func() {}
    void Test(int) {}

};

class Data : public BaseData
{
public:
    ~Data() override = default;

    //void Func() override {}           // Error
    //void Test(double) override {}     // Error
    //void Clear() override {}         // Error
};
```

위의 코드와 같은 경우 Data의 소멸자를 제외한 모든 멤버 함수가 비정상적인 선언으로 인해 컴파일 에러를 발생시킨다
만약 override 키워드가 추가되지 않았다면 컴파일과 실행이 되긴 했을 것이다. 다만 실제로는 개발자가 의도한 바와 같이 정상적으로 오버라이드가 되진 않았을 것이다. 그러나 멤버 함수들에 override가 추가되었기에 이제 컴파일러는 해당 멤버 함수들이 기반 클래스의 어떤 가상 함수를 오버라이드하려고 시도하는 중이라는 것을 알고 있다. 그러나 기반 클래스인 BaseData에 적절한 가상 함수가 선언되어있지 않기에 Data의 오버라이드 시도는 위의 코드 상황으로서는 부적절한 것으로 취급되어 컴파일 에러가 발생하는 것이다

물론 override를 사용하지 않더라도 오버라이딩 자체는 문제없이 동작할 수 있고, 심지어는 override가 추가되지 않았을 때의 미묘한 동작이 오히려 개발자가 원하던 동작일 수도 있다.

그러나 일반적인 상황인 안정적인 오버라이딩을 원하는 상황이라면 `override` 키워드를 사용함으로써 개발자의 눈으로 일일히 실제 오버라이딩 성공 여부를 확인하거나 더 최악의 상황으로 코드 실행 중의 의도치 않은 동작으로 오버라이딩 실패를 눈치채는 경우는 피할 수 있다

- `final` 키워드

클래스 계층 구조를 설계하다보면 해당 클래스 혹은 해당 클래스의 오버라이딩 함수가 더이상 하위 계층의 클래스에서 상속받지 못하도록 혹은 오버라이딩 하지 못하도록 막고 싶을 때가 있다. 이런 상황에서 사용되는 `final` 키워드는 C#의 `sealed`과 비슷하면서도 다른 키워드라고 할 수 있다

```
class BaseData : SomeBase
{
public:
    virtual ~BaseData() = default;

    virtual void Func() {}
    void Test() override final {}

};

class Data final : public BaseData
{
public:
    ~Data() override = default;

    void Func() override {}
    //void Test() override {};           // Error
};

//class Temp : public Data {};        // Error
```

`final`의 동작은 매우 직관적인데 앞서 말했듯 더이상의 상속이나 오버라이딩을 금지하는 것이다

위의 코드를 보자면 `Data`에 `final`이 추가되어있다. 그런데 `Temp`에서 `Data`를 상속받고자 하는 것은 `final` 위반이다

또한 `BaseData::Test`는 `SomeBase` 혹은 그보다 더 위에 선언된 가상함수 `Test`를 오버라이드함과 동시에 `final`을 지정한다. 그렇기에 `Data`에서 `Test`를 오버라이드하고자 하는 것도 `final` 위반이다

9.1.2.1.5. `using` 기반 클래스 멤버

함수를 여러 유효 범위에 걸쳐 오버로딩되지 않는다. 이는 오버로딩 해결 우선순위로 인해 발생한 현상이다

```

class BaseData
{
public:
    virtual ~BaseData() = default;

    void Func(int num) { std::cout << "base" << std::endl; }
};

class Data : public BaseData
{
public:
    ~Data() override = default;

    void Func(double num) { std::cout << "derived" << std::endl; }
};

int main()
{
    BaseData baseData{Data{}};
    baseData.Func(1); // BaseData::Func(int)

    Data data;
    data.Func(1); // Data::Func(double)

    // Output
    // base
    // derived
}

```

위의 코드의 결과는 이해하기 어려울 수도 있고, 대강 감은 오더라도 헷갈릴 수도 있다. 아마 의문이 드는 점은 main 함수 내에서 baseData.Func를 했을 때 BaseData::Func를 호출한다는 점일 것이다. 즉, BaseData 타입이 Data를 가리키는데 오버로딩 해결의 결과로 Data::Func가 호출되지 않는 것이 이상한 것이다

이러한 현상에 적용되는 규칙은 이미 앞서 언급한 바 있다. 클래스는 그 자체로 하나의 네임스페이스이기도 하다는 것이다. 또한, 호출하고자 하는 멤버 함수가 가상 함수라면 동적 타입에 해당하는 클래스에서 해당 함수를 검색하여 호출하고, 호출하고자 하는 멤버 함수가 가상 함수가 아니라면 정적 타입에 해당하는 클래스에서 해당 함수를 검색하여 호출하고자 시도한다는 것이다

따라서 baseData.Func(1)은 Func가 비가상 함수이기에 baseData의 정적 타입인 BaseData::Func에서 오버로딩 해결을 시도한다. 그 결과 BaseData::Func(int)가 최종 선택되어 호출된다

```
class BaseData
```

```

{
public:
    virtual ~BaseData() = default;

    void Func(int num) { std::cout << "base" << std::endl; }
};

class Data : public BaseData
{
public:
    ~Data() override = default;

    using BaseData::Func;
    void Func(double num) { std::cout << "derived" << std::endl; }
};

int main()
{
    BaseData baseData{Data{}};
    baseData.Func(1); // BaseData::Func(int)

    Data data;
    data.Func(1); // BaseData::Func(int)

    // Output
    // base
    // base
}

```

이번 코드에서 달라진 점은 Data에 `using BaseData::Func`가 추가된 것 뿐이다. 이것의 의미는 매우 단순한데, `using` 뒤쪽에 나오는 것을 `using`이 사용된 유효 범위에 보이게 하는 것이다

- 생성자 상속

생성자는 기본적으로는 상속을 한다고 해도 아래쪽으로 내려오지 않으며, 이는 소멸자도 마찬가지이다. 생성자와 소멸자는 겉으로 보여지기에 특정 클래스의 이름과 동일하게 설명되기에 특정한 식별자에 묶이는 것 처럼 보이지만 실상은 생성자와 소멸자라는 그 역할 자체에 묶이기 때문이다

12.1 Constructors

[class.ctor]

Constructors do not have names. A special declarator syntax using an optional sequence of *function-specifiers* (7.1.2) followed by the constructor's class name followed by a parameter list is used to declare or define the constructor. In such a declaration, optional parentheses around the constructor class name are ignored. [Example:

```
class C {  
public:  
    C(); // declares the constructor  
};  
  
C::C() {} // defines the constructor  
—end example]
```

즉, 겉으로 보기에 부모 클래스의 생성자와 자식 클래스의 생성자는 식별자가 다르기에 다른 함수로 취급될 것 같지만 실상은 생성자는 이름 자체가 존재하지 않고 그 역할로서 구분되기에 자식 클래스의 생성자가 부모 클래스의 생성자를 가려버리는 것과 같이 동작한다는 것이다. 그렇기에 부모 클래스를 자식 클래스가 상속받는다고 하더라도 일반적으로는 자식 클래스 입장에서 부모 클래스의 생성자는 볼 수 없다

그렇다면 부모 클래스의 생성자에 `virtual`로서 선언하면 될 것 같지만 이는 허용되지 않는 문법이다. 일반적으로 생성자를 가상 함수로서 선언한다는 것은 자칫하면 부모 클래스의 `private`으로 선언된 멤버 변수들을 초기화하지 못할 수 있다는 심각한 문제를 야기할 수 있기 때문이다

소멸자가 가상 함수로서 선언될 수 있는 것은 특수한 경우라고 할 수 있다

그럼에도 불구하고 자식 클래스에서 부모 클래스의 생성자를 볼 수 있게 하고 싶은 경우가 존재한다. 단순히 결과적으로 부모 클래스의 생성자를 호출하기만 하면 된다면 다음과 같은 방법이 있을 수 있다

```
class Base  
{  
public:  
    Base(int n) : m_age(n) {}  
    virtual ~Base() = default;  
};  
  
class Derived : public Base  
{  
private:  
    int m_age;  
    std::string m_name;  
  
public:  
    Derived(int n) : Base(n) {}  
    ~Derived() override = default;  
};
```

```

int main()
{
    Base* data = new Derived(10);
    delete data;

    return 0;
}

```

그러나 위의 방법은 자식 클래스의 생성자를 경유하여 부모 클래스의 생성자를 호출하는 방법이기에 자식 클래스 그 자체에서 부모 클래스의 생성자를 호출할 수는 없다. 만약 그렇게 하고 싶다면 다음과 같이 코드를 작성하면 된다

```

class Base
{
public:
    Base(int n) : m_age(n) {}
    virtual ~Base() = default;
};

class Derived : public Base
{
private:
    int m_age;
    std::string m_name;

public:
    using Base::Base;
    ~Derived() override = default;
};

int main()
{
    Base* data = new Derived(10);
    delete data;

    return 0;
}

```

위의 두 경우를 비교해보자면 사용하는 입장에서의 코드는 전혀 차이가 없음을 알 수 있다. 만약 위의 코드가 라이브러리라고 한다면 코드를 실행해보고 코드를 뜯어보지 않는 이상 내부적으로 어떤 방식을 사용했는지 알 수 없을 것이다

그러나 이렇듯 내부를 쉽게 알 수 없다는 불편함이 있다고 한들 문제없이 동작만 한다면야 문제가 크게 없을 것이다. 그러나 두 방법 다 원칙적으로는 문제가 발생할 수 있다

발생할 수 있는 문제라고 한다면 어찌보면 당연한 문제인데, 부모 클래스의 생성자를 자식 클래스로 끌고온다면 부모 클래스의 생성자는 자식 클래스의 멤버 변수를 모르는 게 당연하다. 즉, 만약 자식 클래스에 멤버 변수가 존재하고 부모 클래스의 생성자를 끌어와서 호출한다면 자식 클래스의 멤버 변수가 초기화되지 않을 가능성이 존재한다는 것이다

위의 코드를 통해 결론부터 말하면 원칙적으로는 Derived::m_name은 초기화될 것이고, Derived::m_age는 초기화되지 않을 것이다. 이는 명시적으로 생성자에서 초기화하지 않더라도 암시적으로 초기화되는 멤버 변수의 조건이 존재하기 때문인데, 만약 멤버 변수가 기본 생성자의 정의를 가지고 있다면 암시적으로 초기화해준다
즉, Derived::m_age의 타입은 int이고 Built-In 타입은 생성자처럼 보이는 걸 사용할 수 있을 뿐이지 실제 생성자가 존재하지는 않기에 원칙적으로는 초기화되지 않는다

만약 생성자에서 멤버 변수가 초기화되지 않을 때 발생할 수 있는 문제를 생각해보자면 명확하다. 할당받지 않은 메모리로의 접근에 따른 문제이다

그러나 최근의 컴파일러의 경우에는 이러한 문제를 볼 수 없다. 즉, 위에서 예시를 든 코드들을 실행하더라도 문제가 발생하지 않는다는 것이다. 다만 원칙적으로 이러한 문제를 해결하고자 한다면 다음과 같이 멤버 변수의 선언 자체에 초기화식을 제공하는 방법이 있다

```
class Base
{
public:
    Base(int n) : m_age(n) {}
    virtual ~Base() = default;
};

class Derived : public Base
{
private:
    int m_age{ 0 };
    std::string m_name;

public:
    Derived(int n) : Base(n) {}
    ~Derived() override = default;
};

int main()
{
    Base* data = new Derived(10);
    delete data;

    return 0;
}
```

9.1.2.1.6. 반환 타입 완화

앞서 오버라이딩의 경우 다음의 두 조건을 만족해야만 한다고 한 바 있다

- 1) 파생 클래스의 어떤 상위 클래스에 가상 혹은 순수 가상 함수 Func의 정의가 존재해야만 한다
- 2) 파생 클래스의 오버라이딩 함수는 어떤 상위 클래스에 선언된 가상 혹은 순수 가상 함수와 시그니처가 동일해야만 한다

여기서 2번 조건을 보면 상위 클래스의 가상 혹은 순수 가상 함수와 파생 클래스의 오버라이딩 함수의 시그니처가 동일해야한다고 언급한다. 즉, 함수의 반환 타입과 식별자 그리고 매개변수 리스트가 모두 동일해야만 한다는 것이다. 그러나 이와 같은 조건에서 반환 타입에 관한 조건을 약간 완화해주는 특수한 경우가 존재한다

이러한 반환 타입 완화는 다음과 같은 형태로 동작한다

- 1) Derived가 Base를 상속받는다
- 2) Base* virtual Base::Func()와 같은 함수가 존재한다

위와 같은 조건이 만족될 때 Derived에서 Func를 오버라이딩할 때 다음과 같이 재정의할 수 있다

- Derived* Derived::Func() override

이러한 반환 타입 완화는 다른 말로는 공변 반환(Covariant Return) 규칙이라고도 불리는데, 이를 그대로 클래스 상속의 그 특성에 따른 동작을 이용하기에 가능한 것이다. 그렇기에 이러한 반환 타입 완화가 동작하려면 다음의 조건을 만족해야만 한다

- 반환 타입 완화가 동작하려면 가상 함수의 반환 타입이 포인터 혹은 참조여야만 한다

즉, Base*는 Derived*로 완화될 수 있고, Base&는 Derived&로 완화될 수 있다

```
class Base
{
public:
    virtual Base* Func();
};

class Derived : public Base
{
public:
    Derived* Func() override;
};
```

9.1.2.2. 클래스 계층 구조의 설계

하나의 클래스를 설계한다는 것은 새로운 타입을 설계하는 것과 같다

우리는 보통 Built-In 타입 혹은 표준 타입들을 사용하면서 다음과 같은 합리적인 기준을 만족하는 기본 동작을 기대하게 된다. 만약 이상적인 상황이라면 우리가 개발하는 코드 또한 다음의 기준들을 만족하는 것이 좋긴 할 것이다

- 적절한 생성, 소멸, 대입, 이동 메커니즘
- 적절한 멤버 함수 집합의 구성
- 적절한 예외 제어와 테스트로 안전성이 검증되어 코드
- 합리적이고 엄격한 접근 제어

클래스간의 관계를 설계한다는 것은 다음과 같은 다양한 의도로 활용될 수 있다

- 사용하는 측이 코드에 접근할 방법의 수와 방식을 제한하는 인터페이스 역할
- 이미 검증이 되어 안전한 코드를 재활용함으로써 코드의 작성 및 테스트 비용 절감
- 코드를 레고 블럭처럼 조립식으로 설계 가능하게 하고 코드를 한 곳에서 관리하도록 함으로서 설계 및 변경이 용이하게
- 코드간의 결합의 정도 조절

위의 목록의 요소들은 그대로 읽기만 하는 것으로는 다소 추상적이기에 이해하기 힘들 수 있다. 그렇기에 조금 자세하게 예시들과 함께 항목별로 묘사해보겠다. 또한 클래스간의 관계를 다루는 이론 혹은 기법으로 유명한 것이 디자인 패턴이다. 따라서 설명 도중에 관련된 패턴 혹은 내가 경험한 사례가 있다면 함께 소개하겠다

- 접근 방식을 제한하는 인터페이스 역할

인터페이스란 쉽게 말해 외부에 노출된 출입구와 같은 역할인 것이다. 예를 들어 성벽에 둘러싸인 도시가 있다고 할 때, 도시 내부에 다양한 기능을 수행하는 갖가지 지형과 건물 그리고 사람들이 존재하겠지만 도시에 출입할 수 있는 입구는 몇몇 정해진 곳만이 존재할 것이다. 원칙적으로는 도시 내부에 접근을 원하는 사람은 반드시 도시의 입구를 통해서만 할 것이며, 이러한 입구에서는 도시 내에 접근하는 사람의 검문 그리고 도시에서 나가는 사람이 반출하는 것들에 대한 제한 등을 할 수 있을 것이다

클래스 내부의 모든 것을 도시의 내부로, 도시의 입구를 public으로서 선언된 멤버 변수와 함수로 보면 클래스의 인터페이스가 된다. 인터페이스의 목적은 단순하다. 설계자와 사용자 모두에게 해당 클래스로의 접근을 필요할 때 필요한 정도만 제공하여 예상할 수 있게 하는 것이다

이러한 목적을 달성하는 데에는 다양한 방법이 존재할 수 있을 텐데, 가장 단순한 방법은 클래스 내부의 접근 제어를 활용하는 방법일 것이다

```
class Circle
{
public:
    void Draw();
```

```
        Guid m_guid;  
};
```

그런데 위의 코드를 보면 큰 문제가 발생할 가능성이 존재하는데, 멤버 변수인 `m_guid`가 인터페이스로서 날 것 그대로 공개되었기에 읽기/쓰기 동작을 통해 `Circle`이 암시적으로 가정하고 있는 `m_guid`의 유효한 값 범위를 벗어나는 옳지 않은 값이 섞여들어올 가능성이 존재한다는 것이다.

이러한 문제를 해결하는 간단한 방법은 멤버 변수 그 자체를 노출시키지 말고 읽기 및 쓰기를 수행하면서 적절한 필터링 처리를 가미해주는 전용 함수를 추가하는 것이다. 이 방식은 `Getter/Setter Pattern`으로도 알려져 있는데, C#의 경우 해당 패턴을 아예 언어 자체적인 문법 차원에서 지원하면서 `Getter/Setter` 역할을 하는 함수를 특별히 `프로퍼티`라고 부르기도 한다

```
class Circle  
{  
private:  
    Guid m_guid;  
  
public:  
    void Draw();  
  
    Guid GetGuid();  
    bool SetGuid(Guid guid);  
};
```

그런데 위의 코드도 어떤 관점에서는 여전히 문제를 가지고 있다. `Circle`이라는 클래스가 분명 필요한 최소한의 데이터와 함수를 포함하고 있지만 조금 더 작은 여러 파트로도 충분히 나눌 수 있는 다른 관점의 데이터와 함수를 포함하고 있으며, 때로는 특정 한 관점에서의 데이터와 함수의 인터페이스만이 필요하고 다른 관점의 데이터와 함수로의 인터페이스는 불필요하게 과도한 접근 권한으로서 오히려 실수의 가능성을 높이기만 하는 요소일 수 있다는 것이다. 즉, 다음과 같은 상황이 있을 수 있다

```
class Circle  
{  
private:  
    Guid m_guid;  
  
public:  
    void Draw();  
  
    Guid GetGuid();  
    bool SetGuid(Guid guid);  
};  
  
class ObjectGuidManager  
{
```

```

// ...

public:
    bool AddObject( ... );
};

```

위 상황에서의 문제점이자 한계점을 인터페이스 관점에서 보자면 크게 두 가지이다. 첫째는 앞서 말했듯 ObjectGuidManager 관점에서는 Circle의 Guid만이 궁금할 뿐인데 쓸데없이 Circle.Draw()의 인터페이스까지 접근 가능하기에 실수가 나올 수 있다. 둘째로는 위의 코드 그 자체로만 보자면 ObjectGuidManager가 Circle의 Guid를 관리하게 코드를 작성한다면 다른 Guid를 포함하는 클래스 객체는 함께 관리할 수 없다는 문제점이 존재한다. ObjectGuidManager를 관리하고자 하는 클래스 객체마다 여러 번 작성하는 불편함을 극복하기 위해 템플릿을 사용할 수는 있겠지만, 여전히 메모리상에는 관리하는 클래스 객체별로 ObjectGuidManager가 별도로 존재할 것이다

위의 두 가지 문제점을 한 번에 극복할 수 있는 방법은 단순한데, Circle의 내부에 존재하는 다른 관점으로 모아 분리하여 볼 수 있는 정보들을 분할하여 따로 선언하는 것이다

```

__interface IShape
{
    virtual void Draw() = 0;
};

__interface IObjectWithGuid
{
    virtual Guid GetGuid() = 0;
    virtual bool SetGuid(Guid guid) = 0;
};

class Circle : public IShape, public IObjectWithGuid
{
private:
    Guid m_guid;

public:
    void Draw() override {}

    Guid GetGuid() override {}
    bool SetGuid(Guid guid) override {}
};

class ObjectGuidManager
{
    // ...

public:

```

```

        bool AddObject(const IObjectWithGuid* targetObject);
};

int main()
{
    IObjectWithGuid* circle = new Circle();
    circle->GetGuid();
    //circle->Draw();           // Error

    ObjectGuidManager guidManager;
    guidManager.AddObject(circle);
}

```

위의 코드를 보면 Circle의 내용 중 도형의 모양과 관련된 부분과 Guid를 가진 객체와 관련된 부분의 관점들을 분리하여 각각 IShape와 IObjectWithGuid로서 선언하고, ObjectGuidManager는 IObjectWithGuid만을 매개변수로서 다루게 된다

이 방식의 장점은 명확하다. 첫째, ObjectGuidManager는 이제 템플릿을 사용할 필요도 없이 IObjectWithGuid를 상속받는 클래스 객체라면 그 어떤것도 다룰 수 있게 된다. 심지어 Guid 데이터로의 접근은 IObjectWithGuid내에 선언된 인터페이스만을 통해 이루어지기에 IObjectWithGuid를 상속받는 클래스 객체의 다른 인터페이스로의 접근 가능성은 애초에 원천 차단되게 된다. 둘째, 이렇게 관점별로 분리하여 선언해두고 이를 상속받게 하는 순간 이를 상속받는 클래스들이 필요한 연산을 구현한다는 것을 보장할 수 있으며, 이러한 클래스들의 가짓수를 늘리며 프로그램을 확장하는 것도 충분히 관리 가능할만한 일이 된다는 것이다

이 방법을 사용할 때 고려해야할 점들은 다음과 같은 것들이 있을 수 있다

클래스 내 데이터와 함수를 관점에 따라 분리하고자 할 때 어느 정도 수준까지 분해할 것인가 하는 문제가 고려되어야만 한다. 프로젝트의 성격에 따라 적절한 수준의 분해는 프로그램의 원활한 확장과 최소한의 구현을 보장해줄 수 있지만 지나치게 분해하면 오히려 상속 관계와 코드 자체가 난잡해지고 관리가 어려워질 것이다

관점을 분리할 때 분리하여 선언된 부분을 인터페이스와 클래스 중 어떤 걸로 할지 등의 다양한 선택 사항이 있을 수 있다. 이를 간단히 도식화한 것은 아래에 있다

이러한 결정을 논의하기 전에 먼저 생각해야만 하는게, 이러한 설계는 어느정도 관념적인 부분이 있다는 것이다. 즉, 언어 차원에서 제공하는 모든 기능을 되는데로 전부 사용하는 것이 항상 옳은 것은 아니고, 때로는 일부로 제한하는 것이 유리할 수도 있다는 것이다. 기능을 모두 사용하는 것 보다 제한할 때 유리한 점은 여럿이 있을 수 있다

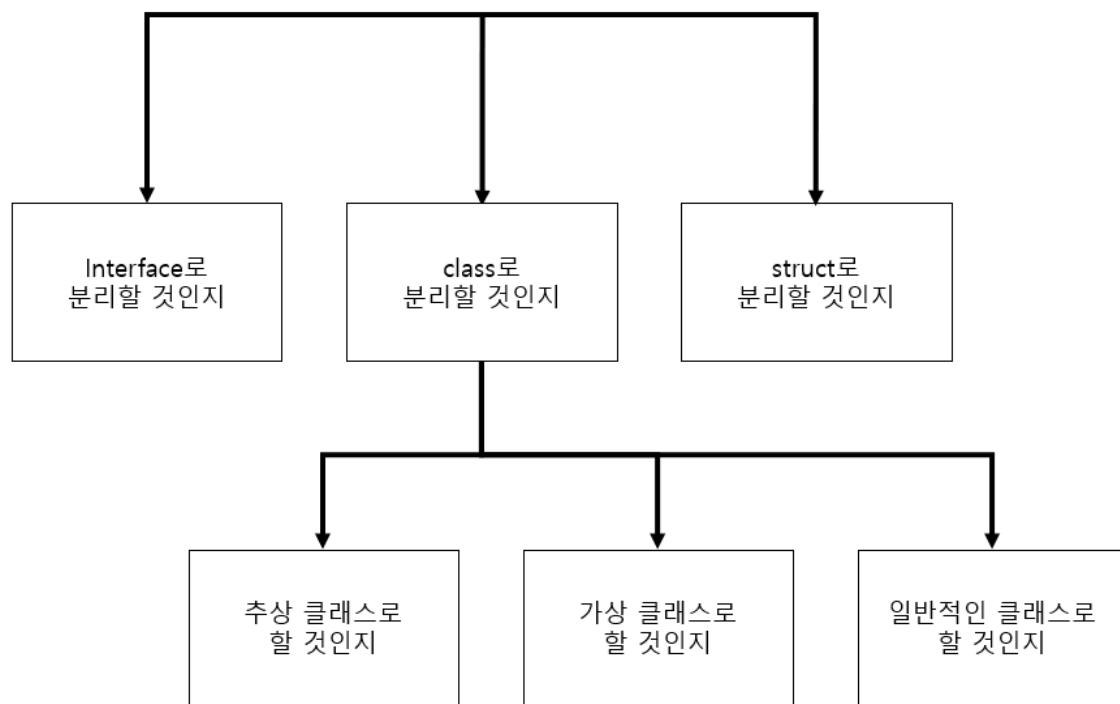
예를 들어 class와 struct를 볼 때 class는 어떤 때에만 쓰고 struct를 어떤 때에만 쓴다고 제한을 하면 코드 관리가 용이해질 수 있다
또한 고려해야만 하는 것이 프로젝트에서 하나의 언어만 사용하는 경우는 잘 없다는 것이다. 예를 들어 Unity 프로젝트에서 C#으로 게임 로직 스크립트를 작성하고 성능에 크리티컬한 부분은 C++로 작성한다고 생각해보자. 이렇게 되면 C#에서 C++의 다양한 구조체와 클래스, 함수 등을 호출해야만 할 것인데 당연하게도 C#과 C++의 문법이 다르고, 겉으로 보기엔 같아 보이는 키워드도 여러 동작이나 제한이 다른 경우가 존재한다. 이런 상황에서는 프로젝트의 성격에 맞춰 C++ 측면에서 사용할 기능과 그 형태에 엄격한 제한을 두는 것이 전체적으로 프로젝트에 들어가는 수고를 줄이는 기능을

할 수 있다

따라서 아래에서 전개되는 논의는 C++ 문법에서 가능한 모든 기능을 논의하지 않고 C#의 문법을 참고하여 진행하게 될 것이다
그 이유라고 한다면 C++의 특징이라 한다면 가능한 최대한 많은 부분에서 개발자에게 자유를 준다고 한다면, C#의 경우 언어 차원에서 다양한 문제로 인해 민감하거나 위험할 수 있는 부분은 기능을 제공하지 않거나 특별히 수고스러운 방식을 통해만 사용할 수 있도록 제한을 한다

이는 언어 철학의 차이라고 할 수 있는데, C#의 언어 철학이 안전을 추구하면서도 유연함을 제공하는 것이기에 C++ 코드를 작성함에 있어서도 분명히 참고할만한 점이 존재하기도 하고, 또한 Unity 개발을 하다보면 만약 회사가 Unity와 협약을 맺는다면 Unity Engine의 보통은 외부에 공개되지 않는 부분인 C++로 작성된 부분을 만지게 될 일이 있을 수도 있다. 그렇기에 위와 같은 이유로 다음의 논의는 C#의 부분을 어느정도 참고하여 전개될 것이다

때에 따라 예시에 C# 코드가 등장할 수도 있다는 점은 양해 바란다



먼저 첫 번째 갈래부터 살펴보자면 먼저 분리할 때의 수단으로서 interface를 사용할 것인지 class를 사용할 것인지를 결정해야만 한다. 우선 여기서 사용되는 용어들을 다음과 같이 정리하겠다

interface	<ul style="list-style-type: none">- interface는 멤버 변수를 가질 수 없다- interface에 선언된 멤버 함수는 순수 가상 함수와 같이 정의를 가질 수 없다- interface는 그 자체로 인스턴스화될 수 없다- interface는 용어 그대로 해당 interface를 상속받는 어떤 것은 이런이런 동작을 가지고 있다라는 것만을 의미한다
------------------	---

class	<ul style="list-style-type: none"> - class는 멤버 변수를 가질 수 있다 - class에 선언된 멤버 함수는 정의를 가질 수 있다 - class는 그 자체로 인스턴스화될 수 있다 - class는 동작에 주안점을 둔 객체를 구현할 때 사용한다 <ul style="list-style-type: none"> - class로부터의 복사 생성, 대입 등은 기본적으로 참조를 통한다 - class는 interface만이 아닌 다른 class도 상속할 수 있다
struct	<ul style="list-style-type: none"> - struct는 멤버 변수를 가질 수 있다 - struct에 선언된 멤버 함수는 정의를 가질 수 있다 - struct는 그 자체로 인스턴스화될 수 있다 - struct는 데이터에 주안점을 둔 객체를 구현할 때 사용한다 <ul style="list-style-type: none"> - struct로부터의 복사 생성, 대입 등은 기본적으로 복사를 통한다 - struct는 interface만을 상속할 수 있다
abstract class	<ul style="list-style-type: none"> - abstract class는 멤버 변수를 가질 수 있다 - abstract class에 선언된 멤버 함수는 정의를 가질 수 있기도 하고 없기도 하다 - abstract class는 순수 가상 함수를 포함할 수 있다 - abstract class는 가상 함수를 포함할 수 있다 - abstract class는 그 자체로 인스턴스화될 수 없다 - abstract class는 기본적으로 interface와 비슷하지만, 다음의 측면들에서 차이를 보인다 <ul style="list-style-type: none"> - interface는 데이터를 가질 수 없지만, abstract class는 데이터를 가질 수 있다 - interface는 내부에 함수 정의를 포함할 수 없지만, abstract class는 내부에 함수 정의를 가질 수 있다 - 종합적으로 봤을 때 interface와 abstract class는 비슷해 보이지만 개념적으로 봤을 때 interface가 abstract class에 비해 보다 엄격하게 인터페이스의 역할에 집중한다
virtual class	<ul style="list-style-type: none"> - virtual class는 멤버 변수를 가질 수 있다 - virtual class에 선언된 멤버 함수는 정의를 가질 수 있다 - virtual class는 순수 가상 함수를 포함할 수 없다 - virtual class는 가상 함수를 포함할 수 있다 - virtual class는 그 자체로 인스턴스화될 수 있다 - virtual class는 그 자체로 부모 클래스로서 동작할 가능성을 염두에 둔 설계일 때 의미있게 사용될 수 있다
normal class	<ul style="list-style-type: none"> - normal class는 멤버 변수를 가질 수 있다 - normal class에 선언된 멤버 함수는 정의를 가질 수 있다

- | | |
|--|---|
| | <ul style="list-style-type: none"> - normal class는 순수 가상 함수를 포함할 수 없다 - normal class는 가상 함수를 포함할 수 없다 - normal class는 그 자체로 인스턴스화될 수 있다
 - normal class는 보통 계층 구조의 말단에 존재하면서 부모 클래스로서 이용되기보단 그 자체의 인스턴스로서 사용된다 |
|--|---|

본론으로 돌아와 첫 번째 갈래에서 **우선 interface를 사용**하여 관점을 분리하고자 한다는 것은 사용자 측에서 뭔가에 접근할 때 interface라는 도구를 활용하여 접근은 가능하게 하되, interface를 통해 인스턴스를 따로 구축하여 활용하는 것은 막고 싶을 때 사용된다. 즉, 말 그대로 사용자는 아무것도 못하고 그냥 말 그대로 interface가 제공하는 정해진 동작들만을 수행할 수 있도록 아주 강력하게 제한할 때 사용하면 유용하다

interface의 예는 앞서 나왔던 코드를 재사용하여 아래에 있다

```
_interface IShape
{
    virtual void Draw() = 0;
};

_interface IOBJECTWithGUID
{
    virtual Guid GetGUID() = 0;
    virtual bool SetGUID(Guid guid) = 0;
};

class Circle : public IShape, public IOBJECTWithGUID
{
private:
    Guid m_guid;

public:
    void Draw() override {}

    Guid GetGUID() override {}
    bool SetGUID(Guid guid) override {}
};

class ObjectGUIDManager
{
    // ...

public:
    bool AddObject(const IOBJECTWithGUID* targetObject);
};
```

```

int main()
{
    IObjectWithGuid* circle = new Circle();
    circle->GetGuid();
    //circle->Draw();           // Error

    ObjectGuidManager guidManager;
    guidManager.AddObject(circle);
}

```

다음으로 만약 class를 활용한다면 그 아래로 뻗어나온 세 가지 옵션이 존재하게 된다. 먼저 **abstract class**부터 보자면 사용자에게 주는 제약은 interface와 같다. 사용자에게 abstract class가 제공하는 정해진 동작들만을 허용하고 나머지는 모두 제한할 때 유용하다. 다만, abstract class는 interface와 달리 멤버 변수와 멤버 함수 정의를 포함할 수 있기에 interface보다 조금 더 넓은 개념이다. 즉, interface가 아닌 abstract class를 사용한다는 것은 여전히 이를 일종의 인터페이스로서 활용하긴 할것이지만 클래스 계층 구조에서 구현에도 어느정도 기여하도록 만들겠다는 의도가 반영된 것이라고 할 수 있다

따라서 abstract class에서 쓸 수 있는 유용한 방법이 abstract class가 선언한 동작에 필요한 데이터를 스스로 포함하여서 이 데이터가 존재한다는 것을 자식 클래스에게서 숨기는 것이다

interface의 경우 동작의 종류만을 선언하고 데이터를 포함할 수 없기에 interface를 상속받는 객체는 적절한 멤버 변수를 직접 선언해야만 한다. 그러나 abstract class의 경우 데이터를 포함할 수 있기에 위와 같은 동작이 가능한 것이다

위와 같은 활용법에는 입장일단이 있을 수 있다. 필요한 동작과 필요한 데이터를 적당히 내부에 포함하기에 abstract class를 상속받는 입장에서는 편할 수 있다. 그러나 상속 받는 입장에서 abstract class가 포함하고 있는 데이터의 유형이 언제나 유효하거나 유용한 형태는 아닐 수 있다. 그런 경우에는 오히려 interface를 사용하는 것이 좋을 것이다

abstract class를 사용하는 예는 다음과 같다

```

__interface IShape
{
    virtual void Draw()
    {
        std::cout << "IShape::Draw()" << std::endl;
    }
};

class IObjectWithGuid
{
private:
    Guid m_guid;

```

```

public:
    virtual Guid GetGuid() = 0;
    virtual bool SetGuid(Guid guid) = 0;
};

class Circle : public IShape, public IObjectWithGuid
{
public:
    void Draw() override {}

    Guid GetGuid() override {}
    bool SetGuid(Guid guid) override {}
};

```

다음 class의 옵션으로 virtual class는 워낙 활용도가 높아 여러 의도로 활용되겠지만 그 중 몇가지만 언급하자면 우선 interface와 abstract를 한 데 모아주는 깔때기와 같은 역할로 사용되기도 한다

왜냐하면 관점에 따라 동작을 이곳저곳으로 분리하고 이를 다시 모아 객체를 구성하려고 하다 보면 그때그때 상속받아야만 하는 것이 지나치게 많아지게된다. 그러나 개발하다보면 분명 동일한 일련의 목록을 상속받는 객체를 여럿 작성해야만 하는 일이 생기곤 한다. 이런 때의 문제는 실수로 일련의 목록의 일부를 깜빡할 수 있다는 것이며, 이러한 문제는 목록의 길이가 길어질수록 커질 것이다.
 그렇기에 동일한 일련의 목록을 상속받는 객체가 여러개 존재한다면 해당 목록을 상속받는 깔때기 역할의 객체를 하나 만들고, 본래 목록을 상속받아야만 할 객체들은 깔때기 객체를 하나 상속받는 형태로 만드는 것이다
 이때 깔때기 역할의 객체는 무엇으로 구현하느냐에 따라 다양한 옵션이 존재할 수 있으며, 그때그때의 상황에 따라 적절히 선택할 수 있다

virtual class를 깔때기로 사용할 때의 예시는 다음과 같다. 다음의 예시는 C#으로 작성되었음에 유의하여 보길 바란다

```

public interface IMapGenUnitInfo
{
    bool BIIsInfoValid { get; }
}
public interface IMapGenUnitInfoWithCubeDataStructure
{
    byte this[Vector3Int targetPos] { get; }
    byte this[int x, int y, int z] { get; }

    byte[,] CubeData { get; }

    Vector3Int TLPos { get; }
    Vector3Int CubeSize { get; }
}
public interface IMapFieldProperties
{
    float MapDensityRate { get; set; }
    float MineralRate { get; set; }

    float DefaultMapDensityRate { get; }
    float DefaultMineralRate { get; }
}
public interface IMapGenChildUnit
{
    List<Guid> AttachedParentUnitGuids { get; }
}
public interface IMapGenNamedUnit
{
    string Name { get; set; }

    int NameLengthLimit { get; }
}

public interface IBiomeUnitInfo : IMapGenUnitInfoWithCubeDataStructure,
    IMapFieldProperties,
    IMapGenNamedUnit { }

// -----
public abstract class MapGenUnitInfo : ObjectWithGUID, IMapGenUnitInfo, IDisposable
{
    private NoiseUtil.INoiseUtilParam m_noiseUtilParam;

    public abstract bool BIIsInfoValid { get; }

    public virtual NoiseUtil.INoiseUtilParam NoiseUtilParam { get { ... } set { ... } }

    public abstract void CleanUpDependency();

    public void Dispose() { ... }
    protected abstract void Dispose(in bool bisDispose);
}

// -----
public sealed class BiomeUnitInfo : MapGenUnitInfo, IBiomeUnitInfo
{
    private string m_name;

    private byte[,] m_biomeUnitInfluenceCube;
    private Vector3Int m_TLPos, m_cubeSize;

    private float m_mapDensityRate, m_mineralRate;

    public BiomeUnitInfo(in Vector3Int LPPos, in Vector3Int cubeSize) { ... }
    ~BiomeUnitInfo() { ... }

    public override bool BIIsInfoValid { get { ... } }

    public byte this[Vector3Int pos] { get { ... } set { ... } }
    public byte this[int x, int y, int z] { get { ... } set { ... } }

    public byte[,] CubeData { get { ... } set { ... } }

    public Vector3Int TLPos { get { ... } set { ... } }
    public Vector3Int CubeSize { get { ... } set { ... } }

    public float MapDensityRate { get { ... } set { ... } }
    public float MineralRate { get { ... } set { ... } }

    public float DefaultMapDensityRate { get { ... } }
    public float DefaultMineralRate { get { ... } }

    public string Name { get { ... } set { ... } }

    public int NameLengthLimit { get { ... } }

    public override void CleanUpDependency() { }

    protected override void Dispose(in bool bisDispose) { ... }
}

```

virtual class의 다음 쓰임새는 템플릿과 함께 사용하여 그 자체로서 사용하기보단 어느 정도의 대리 구현을 하고, 민감한 부분만을 자식 클래스에서 사용하거나 재정의할 수 있도록 허용하는 방식으로 사용되기도 한다

그 예시는 다음과 같다



```
public class SingleTon<_T> : IDisposable where _T : class
{
    protected static _T m_instance;

    public static _T Instance
    {
        get
        {
            if (m_instance == null)
            {
                m_instance = default;
            }
            return m_instance;
        }
        set
        {
            if(value == null)
            {
                m_instance = value;
            }
        }
    }

    public virtual void Dispose()
    {
        Instance = null;
    }
}

public sealed class MapGenerator : SingleTon<MapGenerator>, IMapGenerator
{
    private byte[,] m_biomeUnitResultCube;
    private byte[,] m_areaUnitResultCube;
    private uint[,] m_roomUnitResultCube;
    private uint[,] m_tileUnitResultCube;

    ...

    public override void Dispose()
    {
        m_biomeUnitResultCube = null;
        m_areaUnitResultCube = null;
        m_roomUnitResultCube = null;
        m_tileUnitResultCube = null;

        BiomeUnitMapGenerator.Instance.Dispose();
        AreaUnitMapGenerator.Instance.Dispose();
        RoomUnitMapGenerator.Instance.Dispose();
        TileUnitMapGenerator.Instance.Dispose();

        BiomeUnitInfoResourceManager.Instance.Dispose();
        AreaUnitInfoResourceManager.Instance.Dispose();
        RoomUnitInfoResourceManager.Instance.Dispose();
        TileUnitInfoResourceManager.Instance.Dispose();

        InvalidMapGenUnitInfoCleanManager.Instance.Dispose();

        base.Dispose();
    }
}
```

virtual class를 다양하게 활용하는 방법은 디자인 패턴의 종류가 얼마나 많은지를 보면 알 수 있듯 너무나 다양하고 과연 어떤 방법이 보다 합리적인지는 논란이 많은 부분이다. 따라서 추가적인 정보를 얻고 싶다면 디자인 패턴과 관련된 책이나 자료들을 찾아보기 바란다

다음 옵션으로 일반적인 class를 사용하는 옵션을 보자면 보통 class 그 자체는 만약 계층 구조에 포함된다면 계층 구조의 말미에 위치하거나 계층 구조에 포함되지 않고 단독으로서 존재하게 된다. 특히나 만약 일반적인 클래스를 C++에서 사용할 경우 혹여나 해당 클래스를 다른 객체가 상속받게 되면 문제가 발생할 수 있기에 적절히 막는 것이 바람직할 수도 있다

마지막 옵션으로 struct로 분리하는 옵션은 일반적인 class와 같이 다른 객체가 struct 객체를 상속받는 형태는 의미론적으로도 맞지 않고, 따라서 C#에서도 class가 struct를 상속받는 것은 허용하지 않는다. 만약 관점을 struct로 분리한다는 것은 해당 struct를 멤버 변수로서 포함 관계를 구현한다는 것을 말한다는 것을 알 수 있다

- 이미 검증된 코드의 재활용으로 비용 절감

이미 검증된 코드라는 것은 적절한 많은 테스트가 이루어지고 그에 뒤따르는 보수가 이루어진 코드라는 것이다. 물론 당연하게도 해당 코드가 포함하고 있는 데이터와 동작들이 적절하게 포함되고 통제되었는지도 검증되었다는 것이다

보통 이러한 검증된 코드를 생산하는 것은 큰 비용이 들어가게 된다. 때문에 개발 과정에서 모든 코드를 완벽하게 검증하는 것은 현실적으로 불가능하고 차선책으로서 검증된 코드 조각을 상속받거나 포함하는 방식으로 새롭게 생산한 코드에서 검증해야될 부분을 줄여나가는 방식으로 진행하게 된다
그렇기에 여러번 재사용될 코드 조각을 생산하여 테스트와 보수를 하고 이를 최대한 재활용 함으로써 프로젝트의 비용을 절감하는 의도로 계층 구조를 활용하기도 하는 것이다

- 조립식으로 설계 가능하고 한 곳에서 관리하여 설계 및 변경이 용이하게

이 의도는 설계에 변경 사항이 생겼을 때 이를 적용하도록 강제함으로서 혹은 자동으로 적용되도록 할 수 있는 것이다

이 경우를 설명하기 위해 앞서 사용되었던 다음의 예를 다시 살펴보겠다

```
_interface IShape
{
    virtual void Draw()
    {
        std::cout << "IShape::Draw()" << std::endl;
    }
};

_interface IObjectWithGuid
{
    virtual Guid GetGuid() = 0;
    virtual bool SetGuid(Guid guid) = 0;
};
```

```

class Circle : public IShape, public IObjectWithGuid
{
private:
    Guid m_guid;

public:
    void Draw() override {}

    Guid GetGuid() override {}
    bool SetGuid(Guid guid) override {}
};

class ObjectGuidManager
{
    // ...

public:
    bool AddObject(const IObjectWithGuid* targetObject);
};

int main()
{
    IObjectWithGuid* circle = new Circle();
    circle->GetGuid();
    //circle->Draw();           // Error

    ObjectGuidManager guidManager;
    guidManager.AddObject(circle);
}

```

위와 같은 경우에서 만약 IObjectWithGuid에 해당 interface를 상속받는 객체가 지니고 있는 Guid가 유효한 값인지를 판단하는 BIIsValid()라는 함수의 선언을 추가하고자 한다고 해보자. 그러면 IObjectWithGuid를 상속받는 객체들은 반드시 BIIsValid()를 오버라이딩하는 방향으로 코드를 업데이트하도록 강제할 수 있다. 만약 오버라이딩 하지 않는다면 프로젝트의 빌드 자체가 되지 않을 것이다
 만약 이러한 변경 사항이 interface가 아닌 abstract class에 순수 가상 함수의 형태로 나타나도 같은 효과를 가질 것이다

그러나 만약 변경 사항이 interface가 아닌 virtual class 혹은 일반적인 class의 가상 함수나 일반적인 함수의 형태로 나타나고, 이 객체를 다른 객체들이 상속받는다면 기본적으로 상속받는 객체들은 아무것도 하지 않아도 자동으로 모든 변경사항이 적용된다. 그러다가 만약 상속받는 객체에서 조금의 커스텀이 필요하다면 그때 가상 함수를 오버라이딩해서 사용하는 것이다

- 코드간 결합 정도 조절

계층 구조는 클래스간의 관계를 설명한다. 관계는 포함 관계와 상속 관계를 모두 포함하는데, 이때 계층 구조의 건전성 혹은 유지보수성을 판단하는 잣대 중 하나로 코드간의 결합 정도가 있다

코드간의 결합은 강한 결합과 약한 결합이라는 두 개의 개념으로 분화되는데, 먼저 강한 결합이라 함은 한 코드가 다른 코드에 곧바로 연결되어서 한 부분에서의 변경 사항이 곧바로 다른 코드 부분에 영향을 미치게 되는 것이다
간단한 예제는 다음과 같다

```
class Data
{
    // ...

public:
    void Clear() {}

};

class Container
{
private:
    Data m_data;

public:
    void ClearData() { m_data.Clear(); }
};
```

위와 같은 코드에서 Container.ClearData()의 동작은 다양한 요소들에 의해 유효하지 않게 될 가능성이 존재한다. 흔히 발생할 수 있는 것들을 나열해보자면 우선, Data.Clear()의 함수명이 바뀔 가능성이 존재한다. 또한, Data.Clear()의 내부 동작이 Container가 가정하던 동작과 다른 동작으로 바뀔 수도 있으며, 심지어 Data.Clear()가 이름은 존재하지만 호출이 유효하지 않은 문맥으로 처리되게 선언이 변경될 가능성 혹은 Data.Clear() 자체가 삭제될 가능성도 존재한다

즉, Container는 Data에 전적으로 모든 걸 의존하게 된다. 그런데 그렇다고 Container가 Data가 옳은 데이터와 동작을 가지고 있는지 관리/감독한다고 하면 그것은 Container의 역할을 넘어서는 월권과 같은 행위이기에 실질적으로 Data의 완결성이라 하는 것은 오로지 Data가 단독으로 책임지게 된다. 이렇게 되면 문제점이 Data에 가해지는 제약이 아무것도 없기에 Data 입장에서는 무엇을 하던 본인의 입장에서는 여전히 유효한 동작으로 간주될 수 있다

강한 결합의 반대는 약한 결합인데, 이는 어떤 코드 조각이 곧바로 다른 코드 조각에 연결되는 것이 아니라 인터페이스라는 거름망 혹은 바늘구멍을 거쳐서 연결되게 하는 것이다. 간단한 예제로서 앞서 여러 차례 등장했던 코드를 다시 가져오도록 하겠다

```
_interface IShape
{
    virtual void Draw()
    {
        std::cout << "IShape::Draw()" << std::endl;
    }
};
```

```

__interface IObjectWithGuid
{
    virtual Guid GetGuid() = 0;
    virtual bool SetGuid(Guid guid) = 0;
};

class Circle : public IShape, public IObjectWithGuid
{
private:
    Guid m_guid;

public:
    void Draw() override {}

    Guid GetGuid() override {}
    bool SetGuid(Guid guid) override {}
};

class ObjectGuidManager
{
    // ...

public:
    bool AddObject(const IObjectWithGuid* targetObject);
};

int main()
{
    IObjectWithGuid* circle = new Circle();
    circle->GetGuid();
    //circle->Draw();      // Error

    ObjectGuidManager guidManager;
    guidManager.AddObject(circle);
}

```

약한 결합은 느슨한 결합이라고도 하는데, 이는 사용측인 `main()` 내의 `circle->GetGuid()`와 공급측인 `Circle.GetGuid()`는 각자 서로를 의식할 필요가 전혀 없이 그저 본인이 할 일만을 하면 되기에 느슨해진다는 것이다. 다만 사용측과 공급측 사이의 관계의 건전성을 유지하고 각각에게 어느정도의 제약을 가하는 것은 중간에 있는 인터페이스가 책임을 가지는 것이다.

즉, 느슨한 결합에서는 인터페이스가 양쪽의 관계를 중계하면서 사용측에는 최소한 인터페이스에 선언된 동작들에 대해서는 적절한 호출과 동작이 이루어질 것이라는 혹은 필요하다면 인터페이스 자체적으로 어느정도의 필터링을 거친 결과를 제공한다는 가정을 제공하고, 공급측에는 인터페이스에 선언된 동작들을 그대로 물려받거나 적절히 오버라이딩해야만 한다는 제약을 가함으로서 최소한의 동작을 빠짐없이 구현한다는

가정을 보장해야만 한다는 것이다

그렇기에 인터페이스는 구현함에 있어 당연하게도 신중이 고려되고 설계되어야만 하는데, 인터페이스를 interface, abstract class, virtual class 중 어느 걸 사용할 지라던지 순수 가상 함수, 가상 함수, 일반적인 함수 중 어떤 걸 사용할 지 등이 대표적이고 기본적인 선택 사항이 될 수 있을 것이다

여기엔 추가적으로 인터페이스 단계 그 자체 혹은 인터페이스 단계와 사용측과 공급측의 관계를 어떻게 구성하고 운용하느냐에 따라 어느 정도의 비용과 성능으로 어떤 의도와 효과를 낼 수 있느냐가 다르기에 이 부분의 설계만 해도 매우 다양한 논의가 등장할 수 있고, 이런 설계 부분에서 여러 프로젝트에서 발견된 공통적으로 비슷하게 등장하는 패턴을 모아 정립한 것이 디자인 패턴이라고 할 수 있다

디자인 패턴은 프로그래밍 언어에 의존되지 않는 설계의 영역이기에 보통 의사 코드로 작성되며, 만약 프로그래밍 언어로 작성된다면 유명하고 C++, Java 등으로 주로 쓰인다. 그런데 주의할 점이 디자인 패턴 관련 책이나 자료에 적힌 코드를 프로젝트에 그대로 적용하려고 하면 대개 완벽히 딱 들어맞지는 않는다는 것이다. 중요한 점은 구조와 장단점, 비용 등을 이해하고 적절히 변형하고 때로 여러 개를 합쳐서 사용해야만 한다는 것이다

평소 별로 찾아보지 않더라도 들어볼 정도로 유명한 패턴들은 다음과 같다

- **Single-Ton**
- **Disposable**
- **Factory**
- **Event(Event Queue)**
- **Observer**

9.1.2.2.1. 상속 접근 제어

타 언어들의 경우 상속이라 함은 그냥 상속 그 자체로서 다른 의미가 추가될 여지가 없다. 그러나 C++의 경우 상속 자체에도 접근 제어 속성을 부여할 수 있게 허용한다. 문제는 클래스 내 멤버 접근 제어와 상속 접근 제어가 함께 사용되면서 부모 클래스의 멤버 접근 제어가 자식 클래스로 내려오면서 조금씩 다른 접근 제어로 취급받게 되는 경우가 존재한다

이를 표로 정리하면 다음과 같다

자식 클래스가 부모 클래스를 상속받을 때 부모의 멤버 접근 제어의 취급 변화		상속 접근 제어		
		public	protected	private
멤버 접근 제어	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X

위의 표를 볼 때 어떤 것이 인터페이스로서 외부에 보여지는지, 어떤 것이 파생되어 아래쪽으로 흘러갈 수 있는지, 어떤 것이 해당 클래스 고유의 것(private)으로 취급되는지를 주의깊게 보는 것이 좋다

- *public* 상속

public 상속의 경우 부모 클래스의 *public*, *protected* 멤버를 현재의 접근 제어 그대로 자식 클래스로 끌어오는 것을 의미한다. 다만, 부모 클래스의 *private* 영역은 자식 클래스에서 접근할 수 없다

```
class A
{
public:
    int m_publicData;

protected:
    int m_protectedData;

private:
    int m_privateData;
};

class B : public A
{
public:
    void Test()
    {
        m_publicData = 1;
        m_protectedData = 2;
        //m_privateData = 3;      // Error
    }
};

class C : public B
{
public:
    void Test()
    {
        m_publicData = 1;
        m_protectedData = 2;
        //m_privateData = 3;      // Error
    }
};

int main()
{
    B b;
    b.m_publicData = 1;
    //b.m_protectedData = 2;      // Error
    //b.m_privateData = 3;      // Error
```

```
        return 0;
}
```

주의할 점은 public 상속이 아닌 이상 클래스 계층 구조에서의 업/다운 캐스팅이 불가함을 기억하길 바란다

```
class A {};

class Public_B : public A {};
class Protected_B : protected A {};
class Private_B : private A {};

int main()
{
    A* a = new Public_B;
    //A* a = new Protected_B;      // Error
    //A* a = new Private_B;       // Error

    return 0;
}
```

- *protected* 상속

protected 상속의 경우 부모 클래스의 *public*, *protected* 멤버를 *protected* 멤버 접근 제어 속성으로서 자식 클래스로 끌어오는 것을 의미한다. 다만, 부모 클래스의 *private* 영역은 자식 클래스에서 접근할 수 없다

```
class A
{
public:
    int m_publicData;

protected:
    int m_protectedData;

private:
    int m_privateData;
};

class B : protected A
{
public:
    void Test()
    {
```

```

        m_publicData = 1;
        m_protectedData = 2;
        //m_privateData = 3;      // Error

    }

};

class C : public B
{
public:
    void Test()
    {
        m_publicData = 1;
        m_protectedData = 2;
        //m_privateData = 3;      // Error
    }
};

int main()
{
    B b;
    //b.m_publicData = 1;      // Error
    //b.m_protectedData = 2;      // Error
    //b.m_privateData = 3;      // Error

    return 0;
}

```

- *private* 상속

private 상속의 경우 부모 클래스의 *public*, *protected* 멤버를 *private* 멤버 접근 제어 속성으로서 자식 클래스로 끌어오는 것을 의미한다. 다만, 부모 클래스의 *private* 영역은 자식 클래스에서 접근할 수 없다

```

class A
{
public:
    int m_publicData;

protected:
    int m_protectedData;

private:
    int m_privateData;

```

```

};

class B : private A
{
public:
    void Test()
    {
        m_publicData = 1;
        m_protectedData = 2;
        //m_privateData = 3;      // Error

    }
};

class C : public B
{
public:
    void Test()
    {
        //m_publicData = 1;      // Error
        //m_protectedData = 2;  // Error
        //m_privateData = 3;    // Error
    }
};

int main()
{
    B b;
    //b.m_publicData = 1;      // Error
    //b.m_protectedData = 2;  // Error
    //b.m_privateData = 3;    // Error

    return 0;
}

```

9.1.2.2.2. 구현 상속

이곳에서 말하는 구현 상속이란 멤버 포함에 의한 구현 + 클래스 구현 상속을 함께 말하는 것이다. 여기서의 클래스는 Interface가 아닌 Abstract Class, Virtual Class, 일반적인 Class이다

구현 상속과 인터페이스 상속에 대한 사사로운 점들은 이미 위에서 논의한 바 있다. 때문에 이번에는 구현 상속과 인터페이스 상속의 특징에 대해 조금 더 정돈된 형태로 기술해보고자 한다

구현 상속의 특징은 다음과 같다

- 구현 상속의 가장 주된 장점이자 특징은 상위 클래스의 구현 세부사항이 하위 클래스로 전파된다는 점이다
- 구현 상속의 경우 일반적으로 데이터와 함수 구현을 포함하는 클래스를 하위 클래스가 상속받는 것을 말한다. 이런 경우 당연하게도 상위 클래스에서 변경 사항이 발생할 경우 하위 클래스들까지도 모두 재컴파일되어야만 한다
- 구현 상속 관계에 놓이는 클래스는 때로는 필요 이상의 데이터 혹은 동작을 포함하고 있을 때가 있을 수 있으나 이러한 클래스는 제한된 접근을 위한 요구사항을 만족하기 어려울 수 있다. 예를 들어 어떤 클래스는 데이터는 필요 없이 선별된 특정한 동작만이 필요할 수 있다. 그렇다고 이러한 클래스를 지나치게 세분화한다면 계층 구조의 관리가 어려울 수 있다. 또한 지나친 세분화는 구현 상속의 장점을 약화시킬 수 있다

9.1.2.2.3. 인터페이스 상속

인터페이스 상속의 특징은 다음과 같다

- 인터페이스 상속은 구현 세부 사항을 외부로부터 철저히 숨기며, 그저 특정 관점에서의 동작의 집합을 제공한다는 점이 주된 장점이자 특징이다
- 인터페이스 상속의 경우 데이터와 동작의 정의를 포함하지 않는다. 그렇기에 그 특성상 만약 인터페이스에 변경 사항이 발생할 경우 하위 클래스들은 재컴파일될 필요 없이 인터페이스만이 재컴파일되면 된다

9.1.2.3. 다중 상속

여기서부터는 앞서 정한 인터페이스 등의 용어 대신 일반적인 C++ 용어로 사용하겠다

9.1.2.3.1. 다중 인터페이스

일반적으로 인터페이스를 표현할 때에는 추상 클래스를 사용한다

가변 상태가 존재하지 않는 클래스는 다중 상속되더라도 파생 클래스의 입장에서는 단독 상속만이 존재하는 것이나 진배없다. 가변 상태가 존재하지 않는 모든 클래스는 다중 상속 관계망에서 인터페이스로서 활용될 때 오버헤드가 거의 발생되지 않는다

9.1.2.3.2. 다중 구현 클래스

여기서 다중 구현 클래스란 가변 사항을 포함할 수도 있는 추상 클래스가 아닌 모든 클래스를 다중 상속받는 것을 말한다

앞서 다중 인터페이스에서도 말했듯 가변 상태가 존재하지 않는 기반 클래스를 다중 상속할 때에는 오버헤드가 거의 발생되지 않는다

다중 구현 클래스는 가장 평범한 사용 사례는 일반적으로 기반 클래스의 동작을 물려받음으로서 파생 클래스를 정의할 때 프로그래머의 작업량을 줄여주는 것이다. 프로그래머는 오직 가상 함수의 재정의만을 수행하면 될 뿐이다 주의할 점은 이러한 방식의 다중 구현 클래스의 사용법은 구현 세부사항의 은닉이라는 목적과 충돌할 때가 존재한다는 것이다

9.1.2.3.3. 모호성 해결

다중 상속을 사용하다보면 다수의 기반 클래스가 동일한 이름의 함수를 포함하고 있는 때가 있다. 그런 때에는 호출하고자 시도하는 함수가 클래스 계층 구조 내에서 정확히 어디에 있는 버전을 지칭하는 것인지를 결정해야만 한다

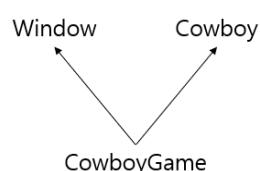
주의할 점은 여기서 말하는 모호성 해결은 순수 가상 함수와 가상 함수의 모호성 해결이 아닌 일반적인 함수의 모호성 해결 방식이라는 점이기에 헷갈리면 안된다

클래스 계층 구조상에서 함수 모호성 해결의 핵심은 함수를 호출한 객체의 정적 타입부터 시작하여 점점 위쪽으로 거슬러올라가며 재귀적으로 호출을 시도한 함수와 일치하는 함수를 찾고자 시도한다는 것이다. 여기서 정확히 일치하는 함수란 함수의 시그니처가 일치하는 것이 아니라 단순히 이름이 일치하는 것이다. 즉, 심지어 다음과 같은 코드도 CowboyGame::SomeFunc() 내에서 호출된 Draw의 모호성으로 인해 컴파일되지 않는다

```
class Window
{
public:
    void Draw(int num);           //  Draw the window
};

class Cowboy
{
public:
    void Draw();                 //  Draw the gun
};

class CowboyGame : public Window, public Cowboy
{
public:
    void SomeFunc()
    {
        //Draw();                  // Error : Which Draw() is called?
    }
};
```



때문에 앞서 말한 클래스 계층 내에서의 재귀적인 함수 탐색은 다중 상속 구조에서 기반 클래스들이 같은 이름의 함수를 가지고 있을 때에는 그다지 힘을 발휘하지 못한다. 때문에 다중 상속 구조 내에서는 다음과 같이 정확히 어떤 기반 클래스의 함수를 호출하고자 하는 것인지 명시적으로 표시해주는 수밖에 없다

```
class Window
{
public:
    void Draw(int num);           //  Draw the window
};

class Cowboy
{
public:
    void Draw();                 //  Draw the gun
};

class CowboyGame : public Window, public Cowboy
{
public:
    void SomeFunc()
    {
        Window::Draw(10);
    }
};
```

그러나 위와 같이 함수를 호출할 때마다 기반 클래스를 일일이 명시한다면 실수의 여지가 다분하다. 때문에 파생 클래스에 기반 클래스의 모호한 함수들을 적절히 호출해주는 Wrapper 목적의 함수를 정의하는 방법이 쓰이기도 한다

```
class Satellite
{
public:
    Debug_Info GetDebugInfo();
};

class Displayed
{
public:
    Debug_Info GetDebugInfo();
};

class Com_Sat : public Satellite, public Displayed
```

```

{
public:
    Debug_Info GetDebugInfo()
    {
        Debug_Info sat_info = Satellite::GetDebugInfo();
        Debug_Info disp_info = Displayed::GetDebugInfo();

        return MergeDebugInfo(sat_info, disp_info);
    }

private:
    Debug_Info MergeDebugInfo(Debug_Info sat_info, Debug_Info
disp_info);
};
```

혹은 기존의 기반 클래스들과 파생 클래스 사이에 모호성 해결을 위한 인터페이스 클래스들을 끼워넣는 방식을 사용할 수도 있다. 설명을 위해 앞서 사용했던 *CowboyGame*의 예를 조금 수정한다면 다음과 같다

```

class Window
{
public:
    void Draw(int num); // Draw the window
};

class Cowboy
{
public:
    void Draw(); // Draw the gun
};

class WWindow : public /*or protected*/ Window
{
public:
    using Window::Window;
    virtual void Draw_Win(int num) = 0;
    void Draw(int num) { Window::Draw(num); }
};

class CCowboy : public /*or protected*/ Cowboy
{
public:
    using Cowboy::Cowboy;
    virtual void Draw_Cow() = 0;
```

```

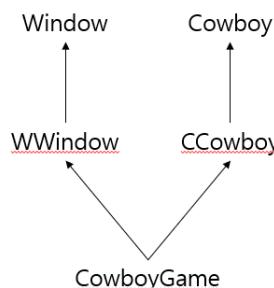
        void Draw() { Cowboy::Draw(); }

};

class CowboyGame : public WWindow, public CCowboy
{
public:
    void Draw_Win(int num) override { WWindow::Draw(num); }
    void Draw_Cow() override { CCowboy::Draw(); }

    void SomeFunc()
    {
        Draw_Cow();
    }
};

```



위의 방법은 어찌보면 우아하기까지한 방법이다

그 첫번째 이유는 만약 프로그래머가 단순히 Window::Draw()와 Cowboy::Draw()를 손쉽게 구분해서 사용하고자만 한다면 Cowboy에 곧바로 Draw_Win()과 Draw_Cow()를 정의할 수도 있을 것이다

그렇게 할 경우 Window와 Cowboy를 상속받는, 혹은 Draw를 포함하고 있는 다른 어떠한 클래스를 포함하여 다중 상속받는 다른 클래스를 설계할때마다 아무런 가이드라인과 제한없이 Wrapper 함수를 정의해야만 할 것이다. 다만 이러한 주먹구구식의 구현은 어쩌다 까먹고 빼먹거나 기반 클래스 내 Draw의 미묘하지만 컴파일러가 해결해줄 수 있는 변경 사항을 전체에 전파할 수 없는 실수를 발생시킬 여지가 다분하다.

그러나 위와 같이 기존의 기반 클래스는 그대로 두면서도 거기서 인터페이스 계층을 추출하게 되면 프로그래머가 기반 클래스와 인터페이스 계층만을 신경한다면 인터페이스 계층을 상속받는 모든 클래스는 필연적으로 실수의 여지 없이 건전한 Wrapper 함수를 가지도록 강제할 수 있다. 또한 만약 인터페이스가 만약 기존의 기반 클래스를 public 상속이 아닌 protected 상속을 받는다면 인터페이스 계층은 추상 클래스에서 public으로 선언된 극히 제한되고 선별된 요소만을 외부에 노출하는 말 그대로의 인터페이스로서도 동시에 사용이 가능한 것이다

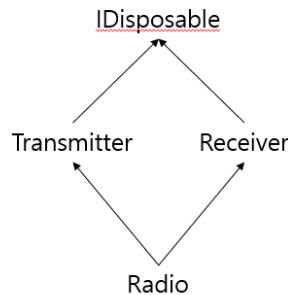
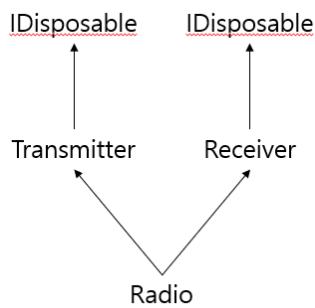
9.1.2.3.4. 기반 클래스의 반복 사용

각 클래스가 직접적인 기반 클래스를 오직 하나만 가지고 있을 경우 클래스 계층 구조는 필연적으로 Linked-List와 같은 형태가 될 것이며 하나의 클래스는 계층 구조에서 단 한번만

등장하게 될 것이다(자기 자신을 상속받는 클래스를 불가하기에). 그러나 다중 상속을 사용하는 순간 동일한 클래스가 클래스 계층 구조 내에서 여러 번 등장할 가능성을 낳게 된다

```
_interface IDisposable {};  
  
class Transmitter : public IDisposable {};  
class Receiver : public IDisposable {};  
  
class Radio : public Transmitter, public Receiver {};
```

위의 코드에서 나타나듯 기반 클래스가 여러번 사용될 경우 상상할 수 있는 구조는 다음과 같은 둘 중 하나일 것이다



첫 번째 그림에서 보여주는 바는 `Radio`가 `IDisposable`의 객체를 내부적으로 두 개 가진다는 것이고, 두 번째 그림이 보여주는 바는 `Radio`가 `IDisposable` 객체를 내부적으로 하나만 가진다는 것이다

C++에서는 두 가지 경우가 모두 나타날 수 있지만 위의 코드와 같은 경우에는 첫 번째 그림과 같은 클래스 계층 구조를 가지게 된다. 즉, 개발하는 입장에서 `Transmitter`와 `Receiver`가 상속받는 각각의 `IDisposable`은 전혀 다른 객체이기에 기존에 개발하던 방식 그대로 개발하면 된다

9.1.2.3.5. 가상 기반 클래스

C++에서는 `virtual`이라는 키워드를 제공한다. 이는 보통 순수 가상 함수 혹은 가상 함수를 선언할 때 사용되지만 클래스 상속 관계를 정의함에 있어서도 사용되기도 한다

```

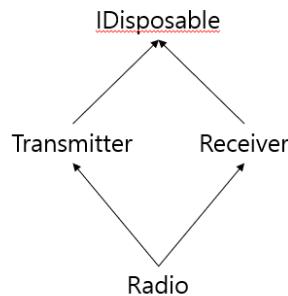
__interface IDispisable {};

class Transmitter : public virtual IDispisable {};
class Receiver : public virtual IDispisable {};

class Radio : public Transmitter, public Receiver {};

```

위와 같이 코드를 조금 수정할 경우에서야 다음의 그림과 같은 클래스 계층 구조를 가지게 된다



여 경우 Transmitter와 Receiver가 상속받는 IDisposable은 동일한 객체이다. 즉, Transmitter에서 수정한 IDisposable의 요소는 Receiver에서 접근하는 IDisposable 객체에도 영향을 미친다

간단한 예제는 다음과 같다

```

class IDispisable // Temporary threatened as class
{
public:
    int m_num;
};

class Transmitter : public virtual IDispisable {};
class Receiver : public virtual IDispisable {};

class Radio : public Transmitter, public Receiver {};

int main()
{
    Radio radio;
    radio.Transmitter::m_num = 10;
    std::cout << radio.Transmitter::m_num << std::endl;
    std::cout << radio.Receiver::m_num << std::endl;

    radio.Receiver::m_num = 20;
}

```

```

    std::cout << radio.Transmitter::m_num << std::endl;
    std::cout << radio.Receiver::m_num << std::endl;

    // Output
    // 10
    // 10
    // 20
    // 20

    return 0;
}

```

일반적인 기반 클래스와 가상 기반 클래스를 다음과 같이 혼용하여 사용할 수 있다

```

class IDispisable // Temporary threatened as class
{
public:
    int m_num;
};

class Transmitter : public virtual IDispisable {};
class Receiver : public virtual IDispisable {};

class Radio : public Transmitter, public Receiver, public IDispisable
{};

int main()
{
    Radio radio;
    radio.Transmitter::m_num = 10;
    std::cout << radio.Transmitter::m_num << std::endl;
    std::cout << radio.Receiver::m_num << std::endl;
    std::cout << radio.IDispisable::m_num << std::endl;

    radio.Receiver::m_num = 20;
    std::cout << radio.Transmitter::m_num << std::endl;
    std::cout << radio.Receiver::m_num << std::endl;
    std::cout << radio.IDispisable::m_num << std::endl;

    radio.IDispisable::m_num = 30;
    std::cout << radio.Transmitter::m_num << std::endl;
    std::cout << radio.Receiver::m_num << std::endl;
    std::cout << radio.IDispisable::m_num << std::endl;
}

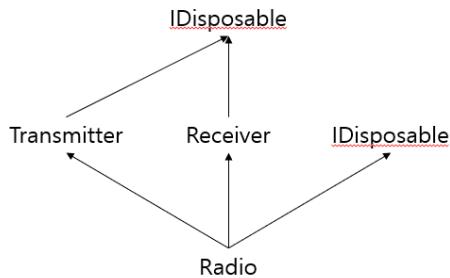
```

```

// Output
// 10
// 10
// 2364308      // Gargabe value
// 20
// 20
// 2364308      // Gargabe value
// 20
// 20
// 30

return 0;
}

```



위의 경우 `radio.Transmitter::m_num`과 `radio.Receiver::m_num`는 하나의 `IDisposable` 객체의 멤버를 지칭하는 것이며, `radio.IDisposable::m_num`은 `Transmitter`와 `Receiver`가 상속받는 `IDisposable` 객체와는 별개의 다른 `IDisposable` 객체의 멤버를 지칭하는 것이다

일반적으로 계층 구조 내에서 두 개 이상의 클래스가 데이터를 공유하는 세 가지 명백한 경우가 있을 수 있다

[1] 데이터를 비지역적으로 만든다(즉, 전역 혹은 네임스페이스 객체를 만든다)

[2] 데이터를 기반 클래스에 넣는다

[3] 객체를 어딘가에 할당하고 두 개의 클래스 각각에 포인터를 부여한다

[1]인 비지역적인 데이터는 대개의 경우 좋지 않은 선택이다. 데이터에 무슨 코드가 어떻게 접근하는지에 대해 제어할 수 있는 바가 없기 때문이다

[2]인 데이터를 기반 클래스로 넣는 방법은 손쉽게 구현이 가능하다는 장점이 존재한다. 그러나 이런 방식을 차용하고 설계를 발전시키다보면 갈수록 점점 공통 기반 클래스는 파생 클래스들에서 공통적으로 사용할 데이터들로만 가득차게 될 것이며 당연하게도 해당 기반 클래스는 읽기에 굉장히 난잡해질 것이다. 이러한 문제를 해결하기 위해 만약 공통 기반 클래스를 목적에 따라 나름대로 분리한다 하더라도 설계에 따라 공통 기반 클래스끼리의 상호 참조가 필요할 수 있으며, 그럴 경우 해당 상호 참조를 해결하기 위한 추가적인 설계가 첨가되어야만 할 수 있다

또한, 이러한 방식의 사용은 논리적으로 [1]과 매우 유사하기에 전역 혹은 네임스페이스 객체를 사용할 때 발생하는 문제가 거의 그대로 나타난다. [1]보다 조금 더 나은 점이라 한다면 그나마 기반 클래스라는 조금 더 특화된 네임스페이스로 둑어둔다는 점이나, getter/setter 등의 패턴을 사용함으로써 일련의 필터링 처리를 가미할 수 있다는 것 정도이다

[3]인 포인터를 통해 접근되는 객체의 공유는 보다 우아한 해결 방법이다

9.1.3. 멤버를 가리키는 포인터

일반적인 포인터라 함은 이미 선언 및 초기화가 이루어 실제로 메모리상에 존재하는 객체의 주소값을 담는 역할을 한다. 즉, 실제하는 객체를 가리킨다. 그러나 멤버를 가리키는 용도의 포인터는 실제하는 객체를 가리키지 않는다. 그보다는 클래스에서 가리키고자 하는 멤버의 오프셋 정보를 담는다고 보면 된다

멤버를 가리키는 주소값은 기본적으로 다음과 같이 완전히 한정된 클래스 멤버 이름에 주소 연산자를 적용하여 얻는다

```
&class-name::member-name
```

또한, 클래스의 멤버를 가리키는 포인터 형식은 기본적으로 다음의 형태를 따른다

```
class-name::*
```

다음의 예제 코드는 Data 클래스의 멤버 변수와 멤버 함수를 가리키는 포인터를 선언하고 활용하는 방식을 보여준다

```
class Data
{
public:
    const char* m_name;

    void Print(const char* msg) { std::cout << msg << std::endl; }

};

using Pnf = void (Data::*)(const char* );
using Pm = const char* Data::*;

int main()
{
    Data data;

    Pnf pf = &Data::Print;
    Pm pm = &Data::m_name;
```

```

(data.*pf) ("Hello");
data.*pm = "Cherno";

std::cout << data.m_name << std::endl;

// Output
// Hello
// Cherno
}

```

멤버를 가리키는 포인터를 사용할 때 유의해야만 할 점들은 다음과 같을 수 있다

- 멤버 함수가 virtual일 수 있다
- 멤버가 static일 수 있다

먼저 멤버 함수가 만약 virtual일 경우 멤버 함수 포인터와 조합된 객체의 동적 타입에 따라 적절한 함수가 선택되어 호출되게 된다

만약 멤버가 static이라면 static 멤버는 특정 객체와 별개로 존재하기에 이러한 경우에 static 멤버를 가리키는 포인터는 통상적인 포인터이다. 즉, 다음과 같다

```

class Data
{
public:
    const char* m_name;

    void Print0(const char* msg);
    static void Print1(const char* msg);
};

//void (*fptr0)(const char*) = &Data::Print0; // Error
void (Data::* pmf0)(const char*) = &Data::Print0;

void (*fptr1)(const char*) = &Data::Print1;
//void (Data::* pmf1)(const char*) = &Data::Print1; // Error

```