

# C++ 강좌

Written by 노상수

이 자료를 보기전에..

[색깔]

- 노란색, 흰색 : 소스코드에 대한 간략한 설명 혹은 표시
- 빨간색 : 슬라이드 내에서 눈에 띄기 쉽게 강조하거나 더 깊이 공부해야 하는 부분
- 보라색 : 이 ppt내에서 추가적으로 다뤄지는 부분
- 초록색 : Class 이름
- 파란색 : 이 슬라이드 내에서 텍스트로 코드를 쓴 부분 혹은 지시어(keyword)

[식별자 명명 및 용어]

- 클래스의 이름은 최대한 'C'를 붙여서 사용하려고 했지만 중간중간 없는 것도 있음
- 본 자료에서는 변수나 함수의 정의를 구현이란 단어와 혼용. 단, 선언과 정의(구현)의 구분은 명확

띄엄띄엄 읽으면 이해가 어렵습니다



## \*목차\*

1. C++ 프로그램 만들기
2. 데이터 타입
3. 연산자
4. 제어문
5. 포인터와 레퍼런스
6. 함수
7. 구조체
8. 클래스 기초
9. 상속성
10. 다형성
11. 클래스 고급
12. 템플릿
13. 예외 처리와 선행처리기
14. 파일 입출력
15. 참고문헌



# 1. C++ 프로그램 만들기

## - C++ 프로그램 구성요소

1) 키워드(Keyword) : 프로그램에서 미리 정의된 예약어

- Ex) return, namespace, int, double 등

2) 문장(Statement) : 프로그램의 최소 실행 단위로서 명령문이라고도 한다. 세미콜론(;)으로 구분

- 명령문은 여러 문장으로 구성되어 하나의 단위로서 실행된다면 중괄호({, })로 묶을 수 있다

- 하나의 문장은 여러 토큰(token)으로 구성된다. 토큰이란 프로그램을 구성하는 요소들의 최소 단위, 공백문자(space, tab)로 구분된다

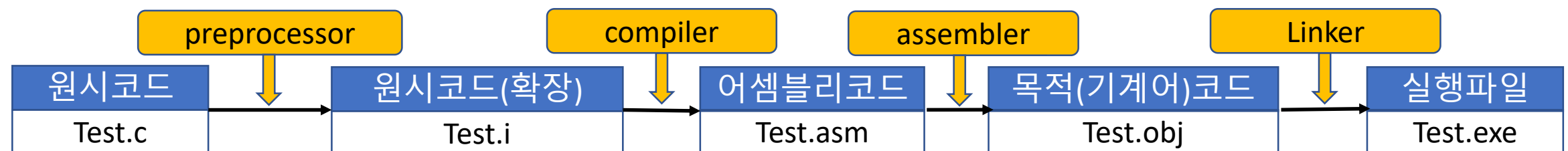
- Ex) return 0; 이라는 하나의 문장은 'return'과 '0'의 토큰으로 구성된다

- Ex) int number = 3;은 'int', 'number', '=', '3'으로 구성되고 'int'와 '='은 각각 C++의 데이터 타입(Data type), 연산자(Operator)에 해당하는 키워드이다. 'number'와 '3'은 키워드는 아니고 각각 변수(variable)와 상수(constant)이다.

3) Main함수 : 모든 C++프로그램은 main함수에서 시작, main함수가 끝이 나면 프로그램도 종료가 된다

## - C++ 프로그램 작성과정

1) 실행파일은 어떻게 만들어 질까? 소스코드 작성 후 컴파일 같은 일련의 과정 속에 어떤 일이 벌어질까?



# 1. C++ 프로그램 만들기

## 1) Preprocessor :

- 소스파일 내부의 #로 시작되는 명령어에 대한 전처리 과정
- test.c -> test.i

## 2) Compiler :

- 전처리된 소스파일을 어셈블리어로 변환하는 과정
- test.i -> test.asm

## 3) Assembler :

- 변환된 어셈블리어를 기계어로 변환하는 과정
- test.asm -> test.obj

## 4) Linker :

- 실행가능한 파일을 만드는 과정
- test.obj -> test.exe

## ★기계어는 무엇? 어셈블리어는 무엇?★

우리가 원시소스 코드를 작성할 때 쓰는 언어는 C나 C++이라고 알고있다. 이를 Level로 분류하자면 사람이 직관적으로 인지할 수 있는 High-Level인 고급언어에 속한다. 반면에 기계어는 기계가 인지할 수 있는 Low-Level의 언어이다. 기계어라 한다면 숫자 0과 1로만 이루어져 있다. 어셈블리어는 고급언어와 기계어 사이에 위치하면서 기계어와 1대1 대응하며 연산, 주소 부분 등에 대해 기호로 표기한 언어이다.



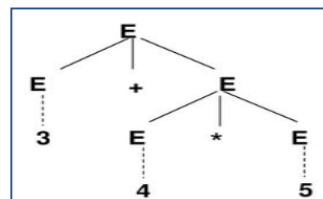
# 1. C++ 프로그램 만들기

## ★컴파일에 대한 좀 더 자세한 고찰★

### - 컴파일 단계

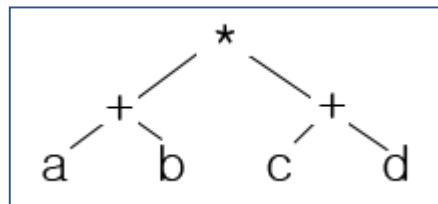
1. 어휘분석(lexical analysis) : 소스 프로그램을 읽어 일련의 토큰(token)을 생성, 주석은 무시
2. 구문분석(syntax analyzer) 혹은 파서(parser) : 어휘분석 단계의 출력인 토큰을 받아 소스 프로그램에 대한 Error Checking을 하고, 올바른 문장에 대해서 구문구조(syntactic structure)를 트리 형태로 만듦

- parse tree : 문법에 따라 확장 및 유도되는 트리 Ex)  $3 + 4 * 5$



$E \rightarrow E + E \mid E * E \mid E \mid N$   
 $N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \dots 9$   
같은 규칙

- abstract syntax tree(요즘 대부분의 컴파일러가 사용) Ex)  $a + b * c + d$



3. 중간코드 생성(intermediate code generation) : 구문분석 출력인 추상 구문 트리를 입력으로 받아 의미 검사 (semantic checking)을 행하고 그에 해당하는 중간 코드를 생성

- a) 형 검사(type checking) : 각 연산자(operator)가 소스 언어(Ex C++)의 정의에 맞는 피연산자(operand)를 가지는가를 검사 - 배열 인덱스에는 실수가 들어가면 안됨
- b) 형 변환(type conversion) : Ex) 실수와 정수의 혼합연산을 허용하는 경우
- c) 우측의 예시는 간단한 중간코드의 예시이다.

### 중간언어예시

```
lod a //load a
lod b //load b
lod c //load c
lod d //load d
add a b //a = a+b
add c d //c = c+b
mul a c //a = a*b
```



## 1. C++ 프로그램 만들기

4. 코드 최적화(code optimization) : 일련의 비효율적인 코드들을 구분해서 좀 더 효율적인 코드로 개선

- 지역 최적화 : 블록 내에서 최적화

- a) 컴파일 시간 상수 연산(constant folding)
- b) 중복된 load, store 명령문 제거
- c) 식(expression)의 대수학적 간소화
- d) 연산 강도 경감(strength reduction)
- e) 불필요한 코드블록(null sequence) 삭제

- 전역 최적화 : 블록 사이의 최적화

- a) 공통 부분식 축약
- b) loop내에서 변하지 않는 값은 loop 밖으로 이동
- c) 도달할 수 없는 코드(unreachable) 제거

- 최적화는 수행 위치에 따라 precode optimization, post optimization으로 나뉜다.

- a) precode : 중간 코드를 이용해서 최적화를 수행
- b) post : 목적 코드 생성 후에 최적화, 기계 의존적인 최적화 방법

- 기계 의존적 최적화 : 연속적인 명령어들을 의미적으로 동등한 하나의 명령어 또는 처리속도가 빠른 명령어로 대체하여 기계어 코드의 성능을 향상시키는 방법

5. 목적 코드 생성기(target code generator) : 중간코드를 입력으로 받아 그와 의미적으로 동등한(1대1 대응하는) 목적 기계(target machine)에 대한 코드를 생성(기계어)

- 목적 코드를 생성하기 위해 행해지는 일

- a) 목적 코드 선택 및 생성
- b) 레지스터의 운영
- c) 기억장소(register) 할당
- d) 기계 의존적인 코드 최적화



## 2. 데이터 타입

- C++에는 흔히 자료형이라고 하는 데이터 타입이 있다. 표현하고자 하는 데이터의 형태(문자, 숫자, Bool) 또는 데이터의 크기에 따라 여러가지로 나뉜다.
- 상수(constant) : 항상 그대로인 수, 변하지 않는다
  - a) 문자열 상수 : "abcde abcde\0" 같은 문자의 집합, 끝이 항상 null 문자이다.
  - b) 문자 상수 : 0~255사이의 하나의 유니코드 문자를 표현하고 1byte가 할당된다.
    - Escape Sequence : '\n', '\\' 같은 제어문자나 인쇄할 수 없는 문자를 표현하는데 사용된다.
  - c) 숫자 상수 : 0 ~ N, 크기에 따라 데이터 타입이 결정
  - d) Boolean 상수 : true & false라는 키워드로 표기하며 참, 참이 아닌 값으로 표현된다.
- 변수(variable) : 변하는 값
  - a) 변수의 필요성 : 프로그램에서 사용하는 데이터의 위치를 쉽게 찾기 위함 즉 의미 있는 변수명을 부여할 것!
  - b) 변수의 선언(declare)과 초기화(initialization), 대입 :
    - b-1) 선언 : [타입] [변수명];
    - b-2) 초기화 : [타입] [변수명] = [값];
    - b-3) 대입 : [변수명] = [값]; ★이때 변수명을 lvalue, 값을 rvalue라고 칭할 수 있다.
  - c) 초기화의 중요성 : 변수를 선언하고 초기화를 하지 않을 시, 메모리 상에서 변수가 저장된 위치에 우연히 남아있는 값이 저장될 수도 있다. 이를 쓰레기 값(garbage) 이라고 한다.
- 상수 변수(constant variable) : 상수에 이름을 부여 하는 것
  - a) 상수 변수의 필요성 : 프로그램에 상수가 많을 경우 명확한 의미를 알기 어렵고 추후 수정도 어렵다.
  - b) 선언과 초기화 : [const] [타입] [변수명] = [값] ;
  - c) 상수는 #define으로 정의하여 매크로로 쓰기도 한다.
  - d) 초기화된 상수 변수는 값을 변경할 수 없다.





## 2. 데이터 타입

### - 데이터 타입(Data type)

a) 정수형 : short (2byte), int (4byte), long (4 or 8byte), long long (8byte)

★long 타입은 32bit linux os 에서 4byte, 64bit linux os 에서 8byte를 가진다.

b) 실수형 : float (4byte), double (8byte), long double (8 or 16 byte)

c) 문자형 : char (1byte)

d) 문자열 : null로 끝나는 문자의 배열(array)로 정의되고 데이터 타입은 따로 없다 하지만 C++표준 라이브러리에서 string 이라는 문자열에 대한 데이터 타입이 제공된다.

e) Boolean : bool (1byte)

### - 데이터 타입 변환(casting) : 한 데이터 타입의 값이 다른 데이터 타입의 값으로 바뀌는 것

★큰 크기 데이터 -> 작은 크기 데이터 형 변환 시 데이터 손실을 불러올 수 있다.

a) 명시적 형 변환 : 형변환에 필요한 데이터 타입을 직접 명시한다.

Ex) double d = 10.0; int i = (int)d; → C스타일 형 변환, C++은 좀더 스마트한 명시적 형변환을 지원

b) 묵시적 형 변환 : 다른 타입의 변수를 대입 했을 때 컴파일러가 자동으로 변환시킨다. (단 데이터 손실에 대한 경고문이 뜸)

c) **Overflow & Underflow** : 기존 자료형이 표현할 수 있는 데이터보다 큰 값 혹은 작은 값이 대입될 경우 생기는 데이터 손실



## 2. 데이터 타입

### ★Overflow에 대한 고찰★

Ex) short 데이터 타입의 표현 범위 : -32768 ~ +32767

```
short befrOverflow = 32767;  
short aftrOverflow = befrOverflow + 1;
```

```
cout << befrOverflow << endl;  
cout << aftrOverflow << endl;
```

```
befrOverflow = 32767  
aftrOverflow = -32768
```

befrOverflow의 결과 값과 aftrOverflow의 결과 값을 비교해보자

# aftrOverflow의 값이 이상하다!!

befrOverflow	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= 32767
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	= 1
aftrOverflow	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 32768이 아닌가?
MSB(Most Significant Bit)																	LSB(Least Significant Bit)

비트를 있는 그대로 계산한다면 32768이 맞다! 그러나 short는 기본적으로 **signed** 이다. 최상위 bit(MSB)가 양수, 음수를 결정짓는 역할이기 때문에 short 형이 양수를 표현하기 위해서는 MSB의 값은 0이 되어야 한다. 그렇지만 현재 위의 값을 보면 MSB는 1이다. 즉 표현할 수 있는 양수의 최대치를 넘어 Overflow가 된 상황이다. MSB가 1이라는 것은 음수를 뜻한다.



## 2. 데이터 타입

### ★2의 보수, 1의 보수?★

- 위의 Overflow 예제에서 MSB가 1인 값은 음수라고 말했다. 그리고 Overflow된 값은 short가 표현할 수 있는 최솟값 이었다. 즉 32767에서 -32768로 되었다는 말인데, 여기서 많은 사람들이 오해를 한다. MSB가 0이면 양수의 최대 값은 **0** 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1로 표기되니까 MSB가 1이고 절대 값으로 따졌을 때 가장 큰 음수의 값은 **1** 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 이 아닌가? 하고 말이다. 틀렸다. **1** 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 은 -1이다. 이것은 우리가 MSB를 기준으로 양수 음수를 판단할 때 흔히 하는 실수이다.

Question) 그렇다면 음수의 값은 어떻게 알 수 있을까?

Answer) 음수를 표현하는 방식에는 절대 부호값(Sign - Magnitude), 1의 보수(1's Complement), 2의 보수(2's Complement)가 있다.

Question) 왜 음수를 표현할까?

Answer) 데이터 연산은 CPU의 ALU에서 진행, 덧셈 연산만이 가능하도록 구현되어 있다. 따라서 뺄셈 연산은 음수를 더하는 방식이다.



## 2. 데이터 타입

### 1) 부호 절대값 (Sign – Magnitude)

부호 절대값 방식은 최상위비트가 0이면 양수 1이면 음수로 나머지 비트는 절대치를 표현한다.

0 0 0 0 = +0	1 0 0 0 = -0
0 0 0 1 = +1	1 0 0 1 = -1
0 0 1 0 = +2	1 0 1 0 = -2
0 0 1 1 = +3	1 0 1 1 = -3
0 1 0 0 = +4	1 1 0 0 = -4
0 1 0 1 = +5	1 1 0 1 = -5
0 1 1 0 = +6	1 1 1 0 = -6
0 1 1 1 = +7	1 1 1 1 = -7

- a) 0이 두개이다. (+0, -0)
- b) 뺄셈기를 따로 설계해야 한다. Ex)  $4 + (-3) = 1$  이 정상 그러나  $0100 + 1011 = ?$  즉  $0100 - 1011$ 을 해야 0001이 나온다.
- c) (뺄셈기가 존재한다고 가정할 때) 덧셈 뺄셈 시 최상위 비트(부호비트)와 나머지 비트를 구분해야 한다. Ex)  $0100 - 1011$ 에서 부호비트를 제외한 나머지 비트의 연산을 고려할 때 절대치가 큰 값에서 작은 값을 빼기 때문에 부호비트는 0, 나머지 비트는 001이 되어 0001이 나오게 된다. 반대로 절대값이 작은 값에서 큰 값을 뺄 때?  $-2 - (-3)$  ? 3에서 2를 빼고 부호비트를 1로 취해준다.
- d) 비교연산의 문제 Ex)  $-3 < -2$  을 이항하고  $-3 - (-2) < 0$  에서 부호비트는 0이 되고 나머지 비트는 001이 되어  $1 < 0$  이라는 모순이 생긴다.  
즉 부호 절대값으로 음수를 사용하고자 한다면 항상 부호비트를 고려해야하는 불편함이 생긴다.

### 2) 1의 보수 (1's Complement)

1의 보수는 부호 절대값 방식에서 단순히 음수의 순서를 뒤집은 것이다. 단순히 보자면 어떤 수 A의 모든 비트를 1로 만들어 주기 위해 2진수 A의 모든 비트를 반전 시킨 값이다.

0 0 0 0 = +0	1 0 0 0 = -7
0 0 0 1 = +1	1 0 0 1 = -6
0 0 1 0 = +2	1 0 1 0 = -5
0 0 1 1 = +3	1 0 1 1 = -4
0 1 0 0 = +4	1 1 0 0 = -3
0 1 0 1 = +5	1 1 0 1 = -2
0 1 1 0 = +6	1 1 1 0 = -1
0 1 1 1 = +7	1 1 1 1 = -0

- a) 최상위 비트에 따라 음수, 양수로 구분되지만 부호와 절대값을 따로 계산하지 않아도 된다. 단 캐리 발생시 LSB에 +1을 해준다. Ex)  $3 + (-2) = 0011 + 1101 = 0000 + 0001(\text{Carry}) = 0001$
- b) 그러나 0이 두개인 것과 carry비트를 고려해야 하는 문제점이 남아있다.



## 2. 데이터 타입

1) 2의 보수 (2's Complement)  
1의 보수를 취하고 LSB에 1을 더한다.

0 0 0 0 = +0	1 0 0 0 = -8
0 0 0 1 = +1	1 0 0 1 = -7
0 0 1 0 = +2	1 0 1 0 = -6
0 0 1 1 = +3	1 0 1 1 = -5
0 1 0 0 = +4	1 1 0 0 = -4
0 1 0 1 = +5	1 1 0 1 = -3
0 1 1 0 = +6	1 1 1 0 = -2
0 1 1 1 = +7	1 1 1 1 = -1

- ★ MSB가 0이면 양수, 1이면 음수 성질 유지
- ★ 덧셈연산으로 뺄셈 가능
- ★ Carry bit 처리 불필요
- ★ 음수의 비교연산 모순 해결
- ★ 0이 두개인 모순 해결



## 2. 데이터 타입

### ★명시적 형 변환에 대하여★

- 기존의 C-Style의 casting은 어떠한 타입이든 안전장치 없이 바꿔준다. 혹은 타입 변환에서 에러가 발생시 어떤 것인지 확인할 수가 없다. C++ Style casting은 4가지의 casting 기능을 제공하는데 각각의 casting은 그 기능에 맞게 엄격하다. 또한 복잡한 코드에서 어떤 종류의 형 변환인지 쉽게 알아볼 수 있다.

1. `const_cast` : 상수 성질을 없애거나 부여
2. `reinterpret_cast` : 어떠한 포인터 타입도 어떠한 포인터 타입으로 변환
3. `static_cast` : 논리적으로 변환 가능한 타입을 변환 Ex) `int -> double` ( o ) `int -> struct` ( x )
4. `dynamic_cast` : 상속의 계층관계를 가로지를 때, 즉 up-casting or down-casting



## 2. 데이터 타입

1. `const_cast` : 포인터 또는 레퍼런스의 상수 성질을 잠깐 제거해주는 데 사용 (함수 포인터는 불가) 혹은 상수 성질을 부여(잘 사용 하지 않음)

```
void constCasting()
{
    const char msg[] = "error";
    //msg[0] = 'a'; // lvalue가 변경할 수 있는 식이 아니다.
    char* msgPtr = const_cast<char*>(msg); ①
    msgPtr[0] = 'a';
    cout << "msgPtr : " << msgPtr << endl;
}
```

결과

Microsoft Visual Studio 디버그 콘솔

msgPtr : error

- 1) msg의 const를 제거한다.
- 2) msgPtr을 통해 값을 수정한다.

2. `static_cast` : c-style casting과 비슷한 의미, 형 변환을 가지는 캐스트 연산자이다. 다만 조금 더 엄격(restrictive)하다.

- C style cast 와 `static_cast`의 비교 : 기본 자료형과 포인터 사이의 형 변환이 되지 않고(공통) 에러 감지 시기가 다르다(차이)

```
char cNum = 'a';
int* pNum;

pNum = static_cast<int*>(&cNum); ← 컴파일 에러
pNum = (int*)&cNum; ← 런타임 에러
*pNum = 5;
```

- 포인터 단에서의 이러한 형 변환은 사실 큰 의미가 없다. 의미 있는 경우는 상속 관계의 객체에 대한 업, 다운캐스팅 이다.



### 3. 연산자

기능별 분류	연산자	결합순서	우선순위
일차식	( ) [ ] -> .	→	1
단항 연산자	! ~ ++ -- - + (형명) * & sizeof	←	2
승제 연산자	* / %	→	3
가감 연산자	+ -	→	4
시프트 연산자	<< >>	→	5
비교 연산자	< <= > >=	→	6
등가 연산자	== !=	→	7
비트 연산자	&	→	8
	^	→	9
		→	10
논리 연산자	&&	→	11
		→	12
조건 연산자	? :	←	13
대입 연산자	= += -= *= /= %= >>= <<= &= ^= !=	←	14
coma 연산자	,	→	15

★new, delete 연산자 = 객체를 동적으로 할당, 해제하는 연산자





## 4. 제어문

### 1) 조건문 (Conditional statement)

#### a) if (조건식) { 명령문; }

- 조건식의 결과가 참이면 명령문을 수행, 거짓이면 수행하지 않음

#### b) if(조건식) else if(조건식) else

- 순차적으로 여러 조건을 판단하여 명령문을 수행

#### c) switch & case문

- 조건이 많을 때 효율적으로 사용이 가능

```
void ifStatement(int nScore)
{
    if (nScore == 10)
        cout << "학점 A+" << endl;
    else if (nScore == 9)
        cout << "학점 A" << endl;
    else if (nScore == 8)
        cout << "학점 B+" << endl;
    else if (nScore == 7)
        cout << "학점 B" << endl;
    else
        cout << "학점 C+" << endl;
}
```

```
void switchcaseStatement(int nScore)
{
    switch (nScore)
    {
        case 10:
            cout << "학점 A+" << endl;
            break;
        case 9:
            cout << "학점 A" << endl;
            break;
        case 8:
            cout << "학점 B+" << endl;
            break;
        case 7:
            cout << "학점 B" << endl;
            break;
        default:
            cout << "학점 C+" << endl;
            break;
    }
}
```

Main문

```
int Score = 6;
ifStatement(Score);
switchcaseStatement(Score);
```

결과

```
학점 C+
학점 C+
```

★switch & case문에서 break;를 쓰지 않는다면?

int nScore = 10일 때 모든 case문의 학점을 출력하게 된다.

★ default:

앞서 정의된 case의 조건에 충족하지 못할 때 default에 정의된 명령문 실행



## 4. 제어문

### 1) 반복문 (loop statement)

#### a) while(조건식) { 명령문; }

- 조건식을 불만족 할 때 까지 명령문 반복

#### b) do{ 명령문; } while(조건식);

- 무조건 한 번은 명령문을 실행하고 조건문 검사

#### c) for( 초기값; 조건식; 증감 ) { 명령문; }

- 조건식을 불만족 할 때 까지 명령문 반복

- tip) for문과 while문은 반복횟수에 따라 주로 다르게 사용된다. for문은 반복횟수가 보다 구체적일 때 사용.

- tip) for문의 실행순서 : 초기값 -> 조건식 -> 명령문 -> 증감 -> 조건식 -> 명령문 -> 증감

- tip) for문의 증감 : 꼭 증감일 필요는 없다. 사용처에 따라 명령문 이후의 제2의 명령문이라 생각해도 된다.

- tip) 반복문의 무분별한 중첩은 프로그램 실행시간을 매우 많이 늘릴 수 있기 때문에 주의하여 사용한다.

while	main
<pre>void whileStatement(int endNum) {     int startNum = 0;     while (startNum &lt;= endNum)     {         cout &lt;&lt; "currentNum : " &lt;&lt; startNum &lt;&lt; endl;         startNum++;     } }</pre>	<pre>int end = 5; whileStatement(end);</pre>
	<b>결과</b>
	currentNum : 0
	currentNum : 1
	currentNum : 2
	currentNum : 3
	currentNum : 4
	currentNum : 5



## 4. 제어문

for	main														
<pre>void forStatement(int endNum) {     for (int index = 0; index &lt; endNum; index++)     {         cout &lt;&lt; "current index : " &lt;&lt; index &lt;&lt; endl;     } }</pre>	<pre>int end = 5; forStatement(end);</pre>														
	<table><tr><th colspan="2">결과</th></tr><tr><td>currentNum</td><td>: 0</td></tr><tr><td>currentNum</td><td>: 1</td></tr><tr><td>currentNum</td><td>: 2</td></tr><tr><td>currentNum</td><td>: 3</td></tr><tr><td>currentNum</td><td>: 4</td></tr><tr><td>currentNum</td><td>: 5</td></tr></table>	결과		currentNum	: 0	currentNum	: 1	currentNum	: 2	currentNum	: 3	currentNum	: 4	currentNum	: 5
결과															
currentNum	: 0														
currentNum	: 1														
currentNum	: 2														
currentNum	: 3														
currentNum	: 4														
currentNum	: 5														
do while	main														
<pre>void dowhileStatement(int endNum) {     int startNum = endNum;     do     {         cout &lt;&lt; "currentNum " &lt;&lt; startNum &lt;&lt; endl;         startNum++;     } while (startNum &lt;= endNum); }</pre>	<pre>int end = 5; dowhileStatement(end);</pre>														
	<table><tr><th colspan="2">결과</th></tr><tr><td>currentNum</td><td>5</td></tr></table>	결과		currentNum	5										
결과															
currentNum	5														



## 5. 포인터와 레퍼런스

- 포인터(pointer)란 말 그대로 무언가를 가리키는 것 이다. 포인터 값은 메모리 주소를 의미하며 포인터 변수는 변수(포인터 데이터)를 저장한다.

- 포인터 사용 이유

- 배열과 같은 연속된 데이터에 대한 접근과 조작이 용이하다.
- 메모리 동적 할당/해제를 통해 유동적인 메모리 관리가 가능하다.
- 복잡한 자료구조를 효율적으로 처리할 수 있다.
- call by address 에 의한 전역 변수의 사용을 억제한다.

- 포인터의 선언

`[Type]* [Variable] = nullptr;`

- 포인터의 초기화 : 포인터 변수에 주소 값을 저장한다. & 참조 연산자는 주소 값(포인터 데이터)를 반환한다.

`[Type]* [Variable] = &[Variable2];`

```
int num = 10;
int* numPtr = &num;
```

← 정수 값 10을 저장하는 변수 num 선언 및 초기화

← 변수 num의 주소를 가리키는 포인터변수 numPtr 선언 및 초기화

주소	0x003ef990
변수	numPtr
값	0x003ef99c

주소	0x003ef99c
변수	num
값	10

num	10
numPtr	0x003ef99c {10}
&numPtr	0x003ef990 {0x003ef99c {10}}

- 포인터 변수가 가리키는 주소의 데이터 접근 : 역 참조 연산자 \*를 이용.

-tip) 포인터 변수는 항상 4byte의 크기를 가진다.

```
std::cout << *numPtr << std::endl;
```

Microsoft Visual Studio 디버그 콘솔

10



## 5. 포인터와 레퍼런스

### - 포인터 사용 예시

#### 1. 포인터로 배열접근

```
void access2Array(void)
{
    int nArray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int* arrayPtr = nArray; 1
    for (int i = 0; i < sizeof(nArray) / sizeof(int); i++)
    {
        std::cout << *(arrayPtr + i) << " "; 2
        std::cout << arrayPtr[i] << " "; 3
        std::cout << *(arrayPtr++) << " "; 4
    }
    std::cout << std::endl;
}
```

- 1) arrayPtr에 nArray배열의 첫 번째 주소 값을 저장
- 2) 연속된 주소의 데이터는 주소+i로 접근이 가능하다. 이때 역 참조 연산자 \*로 데이터 접근
- 3) 배열의 첫 번째 주소를 가지는 arrayPtr은 배열처럼 데이터에 접근이 가능하다.
- 4) arrayPtr에 후위연산자와 \*연산자로 데이터에 접근이 가능하다.



## 5. 포인터와 레퍼런스

### - 포인터 사용 예시

#### 1. 포인터로 배열접근

```
void access2Array(void)
{
    int nArray[10] = { 1, 2, 3, 4,5,6,7,8,9,10 };
    int* arrayPtr = nArray; 1
    for (int i = 0; i < sizeof(nArray) / sizeof(int); i++)
    {
        std::cout << *(arrayPtr + i) << " "; 2
        std::cout << arrayPtr[i] << " "; 3
        std::cout << *(arrayPtr++) << " "; 4
    }
    std::cout << std::endl;
}
```

#### 결과

Microsoft Visual Studio 디버그 콘솔

1 2 3 4 5 6 7 8 9 10

- 1) arrayPtr에 nArray배열의 첫 번째 주소 값을 저장
- 2) 연속된 주소의 데이터는 주소+ index 로 접근이 가능하다. 이때 역 참조 연산자 \*로 데이터 접근
- 3) 배열의 첫 번째 주소를 가지는 arrayPtr은 배열처럼 데이터에 접근이 가능하다.
- 4) arrayPtr에 후위연산자와 \*연산자로 데이터에 접근이 가능하다.



## 5. 포인터와 레퍼런스

### ★접근 방법에 따른 속도차이★

```
void access2Array(void)
{
    int nArray[10] = { 1, 2, 3, 4,5,6,7,8,9,10 };
    int* arrayPtr = nArray;

    for (int i = 0; i < sizeof(nArray) / sizeof(int); i++)
    {
        std::cout << *(arrayPtr + i) << " "; 1
        std::cout << arrayPtr[i] << " ";      2
        std::cout << *(arrayPtr++) << " ";    3
    }
    std::cout << std::endl;
}
```

#### - 접근 방식

- 1)번과 3)번은 '포인터 주소'에 접근 후 값(데이터)에 접근하는 방식
- 2)번은 배열의 인덱스로 접근하는 방식, 각각의 요소에 대한 주소 값이 컴파일 시 계산이 된다.

#### - 속도 차이

- 1), 3)과 2)의 속도차이는 뭐가 빠를까?

→ 정답은 2)가 빠르다. 1, 3번은 주소 접근 후 데이터에 접근을 하기 때문에 두번의 access를 요하는 간접 접근 방식이고 2번의 경우에는 각각의 index로 데이터에 바로 접근하는 직접 접근 방식이기 때문이다.



## 5. 포인터와 레퍼런스

- 포인터 사용 예시

2. 포인터로 동적할당

```
void dynamicAllocateArray(int indexNum)
{
    //indexNum = 5
    int* arrayPtr = new int[indexNum] {0, }; ①
    for (int nIndex = 0; nIndex < indexNum; nIndex++)
        cout << arrayPtr[nIndex] << " ";
    cout << endl;

    for (int nIndex = 0; nIndex < indexNum; nIndex++)
    {
        arrayPtr[nIndex] = nIndex; ②
        /*(arrayPtr + nIndex) = nIndex; ③
        /*(arrayPtr++) = nIndex; ④
    }
    for (int nIndex = 0; nIndex < indexNum; nIndex++)
        cout << arrayPtr[nIndex] << " ";
}
```

결과

 Microsoft Visual :

```
0 0 0 0 0
0 1 2 3 4
```

1) 인자로 넘어온 indexNum 개수만큼 new 연산자로 연속되는 메모리 할당 및 첫 번째 주소 저장, 모두 0으로 초기화

2), 3), 4) 위 슬라이드와 동일한 방식으로 데이터에 접근 후 데이터 저장





## 5. 포인터와 레퍼런스

- 포인터 사용 예시

### 3. 포인터로 인자 넘기기

main

```
int a = 10, b = 20;  
cout << "a : " << a << " , " << "b : " << b << endl;  
pointerParameterSwap(&a, &b);  
cout << "a : " << a << " , " << "b : " << b << endl;
```

결과

C:\ Microsoft Visual Stud

```
a : 10 , b : 20  
a : 20 , b : 10
```

함수 소스

```
void pointerParameterSwap(int* num1, int* num2) ①  
{  
    int temp;  
    temp = *num1; ②  
    *num1 = *num2;  
    *num2 = temp;  
}
```

- 1) 함수를 호출하면 a와 b의 주소를 인자를 넘겨준다. 함수 내부에서는 `int* num1=&a, int* num2=&b`가 된다.
- 2) 데이터에 접근을 하기 위해 \* 연산자를 쓴다.

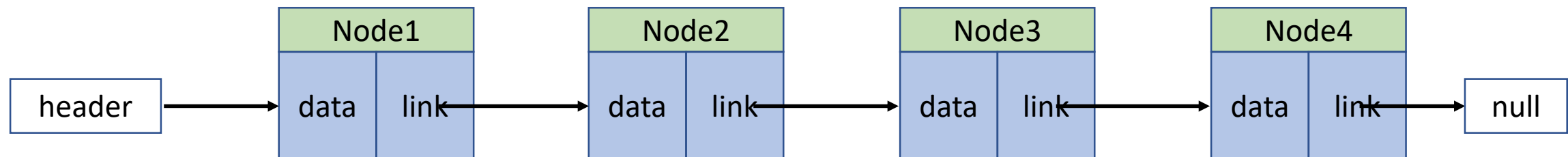


## 5. 포인터와 레퍼런스

### - 포인터 사용 예시

#### 4. 자료구조 예시

- Linked List : 데이터를 저장하기 위한 자료구조이며 배열, 포인터로 구현할 수 있다. 배열은 구현은 쉽지만 데이터의 삽입, 삭제가 불편하고 배열의 크기가 정해져 있어서 가변적인 데이터 보관이 어렵다. 포인터로 Linked List를 구현하면 구현은 복잡하지만 배열의 단점을 모두 극복한다.



#### 5. 함수 포인터

- 메모리 주소를 가리킨다는 본질적인 내용에서 좀 더 확장되어, 변수만이 아닌 메모리에 올라온 함수도 가리킬 수 있는 함수 포인터가 있다. (6장에서 설명)



## 5. 포인터와 레퍼런스

- 레퍼런스(reference)란 일종의 별명이다. 포인터와 같은 역할을 하는 것 처럼 보이지만 레퍼런스 변수로서의 공간이 따로 할당되는 것이 아닌. 원본 변수의 메모리를 공유하는 특징을 가지고 있다. 따라서 선언과 동시에 반드시 초기화가 이루어져야 한다.
- 레퍼런스 사용 이유
  - a) 함수 인자로 null 값을 받을 일이 없다. (포인터는 null 값이나 지정한 데이터타입이 아닌 것을 받을 수도 있다.)
  - b) 포인터는 참조하는 대상을 변경할 수 있어 실수가 일어날 가능성이 있지만 레퍼런스는 한 번 참조한 대상이 소멸하기 전에는 값이 변하지 않는다.
- 레퍼런스의 선언과 초기화  
[Type]& [Variable] = [Variable2];

```
int num = 10;  
int& numRef = num;
```

← 원본 변수 선언&초기화

← 레퍼런스 변수 선언&초기화

이름	값
num	10
numRef	10
&num	0x0039fb34 {10}
&numRef	0x0039fb34 {10}

Ref, 원본의 주소가 같다.

- Issue : 레퍼런스는 const pointer 이다? 선언과 동시에 초기화 말고는 다르다. 즉 NULL 값을 가질 수 없다.

```
int num = 10;  
int* const numRef = &num;
```

이름	값
&num	0x003ffe60 {10}
numRef	0x003ffe60 {10}
&numRef	0x003ffe54 {0x003ffe60 {10}}

Ref, 원본의 주소가 다르다.



## 5. 포인터와 레퍼런스

- 3 types of const pointer

```
int num = 10;  
  
int* const numRef = &num; 1  
const int* numRef2 = &num; 2  
const int* const numRef3 = &num; 3
```

- 1) int 타입 데이터를 가리키는 const 포인터 : 초기화 이후 다른 주소 값 대입 불가
- 2) const int 타입 데이터를 가리키는 포인터 : \*연산자로 참조 값 변경 불가
- 3) const int 타입 데이터를 가리키는 const 포인터 : 1), 2) 두개의 속성을 지님

Ex)

```
int num = 10;  
int anotherNum = 20;  
const int* ptr1 = &num; //가능  
*ptr1 = 5; //ptr1이 const int를 가리키므로 불가능  
ptr1 = &anotherNum; //가능
```

```
int *const ptr2; //불가능. const 포인터는 선언과 동시에 초기화  
int *const ptr3 = &num; //가능  
*ptr3 = 6; //가능  
ptr3 = &anotherNum; //불가능 const 포인터이므로 가리키는 대상을 바꿀 수 없음
```



## 5. 포인터와 레퍼런스

★포인터 변수를 레퍼런스 매개변수로 쓸 수 있는가?★

→ Yes

→ 디버깅시 우리가 생각해 봐야 하는 것은?

```
int main(void)
{
    int* arrPtr = new int[10]{ 0, };
    { ... }
    pointerReferPara(arrPtr);
    for (int i = 0; i < 10; i++)
        cout << arrPtr[i] << endl;
    return 0;
}

void pointerReferPara(int* (&arr))①
{
    for (int i = 0; i < 10; i++)
        arr[i] = i;
}
```

1) 포인터 변수에 대한 call by reference 코드를 어떻게 작성할 것인가?  
→ 포인터 타입(int \*)이지만 변수는 &(레퍼런스)이다. 라고 생각하자.

2) 매개변수 arr자체의 주소는 arrPtr과 같은 주소 값 이어야 한다.  
→ arr과 arrPtr의 주소가 같다.

3) 매개변수 arr이 가리키는(가지는) 주소 값은 arrPtr과 같은 값이어야 한다.

이름	값
&arrPtr ②	0x000000d7367df5d8 {0x00000232872b5500 {0}}
arrPtr ③	0x00000232872b5500 {0}
&arr ②	0x000000d7367df5d8 {0x00000232872b5500 {0}}
arr ③	0x00000232872b5500 {0}



## 5. 포인터와 레퍼런스

- RAII (Resource Acquisition Is Initialization) : 우리는 new 연산자를 이용해서 동적할당(Dynamic Memory Allocation)을 하면 delete로 할당을 해제한다. 그러나 이는 프로그래머가 직접적인 책임을 가지고 있으며 1. 코딩에 실수를 범할 수 있고(delete를 못하거나, 이미 소멸한 객체를 또 다시 delete(double free 버그)), 2. 1번과 더불어 이미 해제된 메모리를 여전히 가리키고 있는 Dangling Pointer문제, 3. 자원의 할당과 해제 명령문 사이에 exception이 발생하기라도 한다면 해제를 못하는 경우도 생겨 누수가 발생한다. 이 때 자원의 안전한 사용을 위해서 객체가 쓰이는 Scope를 벗어나면 자원을 해제하여 메모리 누수를 방지하는 기법이다. 그 결과 프로그램이(프로그래머가 아닌) 자원을 효율적으로 관리를 할 수 있다.

C++ 11 부터 지원하는 스마트 포인터가 그 역할을 한다. 스마트 포인터 변수는 메모리 영역 중 스택에 정의되기 때문에 지역변수의 수명과 일치한다. 아래의 세가지 포인터는 메모리와 객체를 관리하는 전략이 다르다.

1) unique\_ptr : 하나의 스마트 포인터만이 특정 객체를 소유, 때문에 소유권에 대한 복사는 불가능하고 move() 멤버함수를 통해 이전은 가능하다.

2) shared\_ptr : 참조 횟수가 계산되는(reference count) 포인터이다. 특정 객체에 새로운 shared\_ptr이 추가될 때마다 1씩 증가, 수명이 다하면 1씩 감소한다. 모든 shared\_ptr의 수명이 다하여 참조 횟수가 0이 된다면 delete 키워드로 자동 해제한다.

3) weak\_ptr : 하나 이상의 shared\_ptr 참조가 있는 객체에 대한 접근을 제공, 소유자의 수(참조 횟수)에는 포함되지 않는다. 서로가 서로를 가리키는 shared\_ptr가 있다면 참조 횟수는 절대 0이 되지 않기 때문에 메모리는 영원히 해제되지 않는 문제점 때문에 이러한 순환 참조(circular reference)를 제거하기 위해서 사용된다.



## 5. 포인터와 레퍼런스

### 예제 클래스 Person

```
class Person
{
private:
    string name;
    int age;
public:
    ① Person(const string& name, int age) { this->name = name; this->age = age; }
    ② ~Person() {}
    ③ void ShowPersonInfo() { cout << name << "의 나이는 " << age << endl; }
};
```

1. 생성자 (이름과 나이 초기화)
2. 소멸자 (스마트 포인터이므로 일부러 구현 x)
3. 이름과 나이 출력하는 멤버함수

### 1) unique\_ptr

```
int main()
{
    ① unique_ptr<Person> son = make_unique<Person>("형욱", 27);
      son->ShowPersonInfo();

    ② unique_ptr<Person> kim = move(son);
    ③ son->ShowPersonInfo();
      return 0;
}
```

1. son에 Person객체 할당
  2. kim에 son이 가리키는 객체 소유권 이전, son = nullptr
  3. son은 null이므로 액세스 위반 에러
- ★ unique\_ptr의 활용으로 디자인 패턴 중 single tone pattern이 있다.

#### 예외가 발생함

예외가 throw됨: 읽기 액세스 위반입니다.  
**this**이(가) nullptr였습니다.

세부 정보 복사 | [Live Share](#) 세션을 시작합니다.

#### ▲ 예외 설정

- ☒ 이 예외 형식이 throw되면 중단
- 다음에서 발생한 경우 제외:
- ☐ C++Essence.exe

[예외 설정 열기](#) | [조건 편집](#)



## 5. 포인터와 레퍼런스

### 2) shared\_ptr

```
① shared_ptr<Person> son = make_shared<Person>("형욱", 27);
② cout << "참조 횟수 " << son.use_count() << endl;

③ shared_ptr<Person> kim = son;
④ shared_ptr<Person> lee = kim;
⑤ cout << "참조 횟수 " << son.use_count() << endl;
```

1. son에 Person 객체 할당
2. son이 가리키는 객체에 대한 참조 횟수 출력
3. son이 가리키는 객체를 kim가 가리킴(공유)
4. kim이 가리키는 객체를 lee가 가리킴(공유), 결국 모두 같은 객체를 가리킴
5. 2번과 동일

Microsoft Visual Studio 디버그 콘솔

```
참조 횟수 1
참조 횟수 3
```

```
lee.reset();
kim.reset();
son.reset(); ①
cout << "참조 횟수 " << son.use_count() << endl; ②
```

1. 모든 shared\_ptr 참조 해제
2. son이 가리키는 객체에 대한 참조 횟수

```
참조 횟수 0
```

#### 예제 클래스 Person

```
class Person
{
private:
    string name;
    int age;
public:
    Person(const string& name, int age) { this->name = name; this->age = age; }
    ~Person() {}
    void ShowPersonInfo() { cout << name << "의 나이는 " << age << endl; }
    ① shared_ptr<Person> nextPerson;
};
```

1. 기존 클래스에 shared\_ptr 멤버변수 추가





## 5. 포인터와 레퍼런스

### 2) shared\_ptr

```
① shared_ptr<Person> son = make_shared<Person>("형욱", 27);
② shared_ptr<Person> kim = make_shared<Person>("기철", 25);
   cout << "kim의 참조 횟수" << kim.use_count() << endl;
   cout << "son의 참조 횟수" << son.use_count() << endl;
   //
③ son->nextPerson = kim;
④ kim->nextPerson = son;
   cout << "kim의 참조 횟수" << kim.use_count() << endl;
   cout << "son의 참조 횟수" << son.use_count() << endl;
   //
⑤ son.reset();
⑥ cout << "kim의 참조 횟수" << kim.use_count() << endl;
   cout << "son의 참조 횟수" << son.use_count() << endl;
```

1. son에 Person 객체 할당

2. son에 Person 객체 할당

→ son과 kim은 각각의 객체를 가리키고 있다. 즉 각각의 reference counting은 1이다.

3. son의 멤버변수가 kim과 동일한 객체를 가리킨다.

4. kim의 멤버변수가 son과 동일한 객체를 가리킨다.

→ 서로가 서로를 가리키는 형태가 되며(순환 참조), 각각의 reference counting은 2가 된다.

5. son을 reset() 시킴으로써 null로 만들었다.

→ 6번의 결과는 어떻게 될까? son의 reference counting은 0이 되었지만 kim은 여전히 2이다. 그럼 아직도 kim에 해당하는 객체를 kim말고도 무엇인가가 가리키고 있다는 뜻인데,,,, 다음 슬라이드를 보자

Microsoft Visual Studio 디버그 콘솔

kim의	참조	횟수	1
son의	참조	횟수	1
kim의	참조	횟수	2
son의	참조	횟수	2
kim의	참조	횟수	2
son의	참조	횟수	0



## 5. 포인터와 레퍼런스

### 2) shared\_ptr

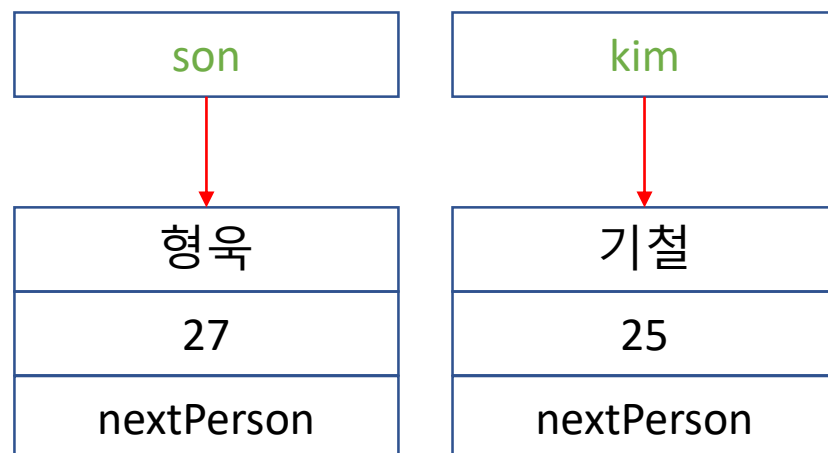
```
cout << kim->nextPerson.use_count() << endl;
```

위 코드는 전 슬라이드에 이어 추가한 구문이다. kim의 멤버변수 nextPerson이 가리키던 객체는 son이 가리키던 객체였다. 아까 분명 son.reset()으로 참조해제를 했을 텐데?? 아니다! kim의 멤버변수가 son이 가리키던 객체를 가리킴으로써, reference counting 횟수가 2가 된 것을 보지 않았던가?

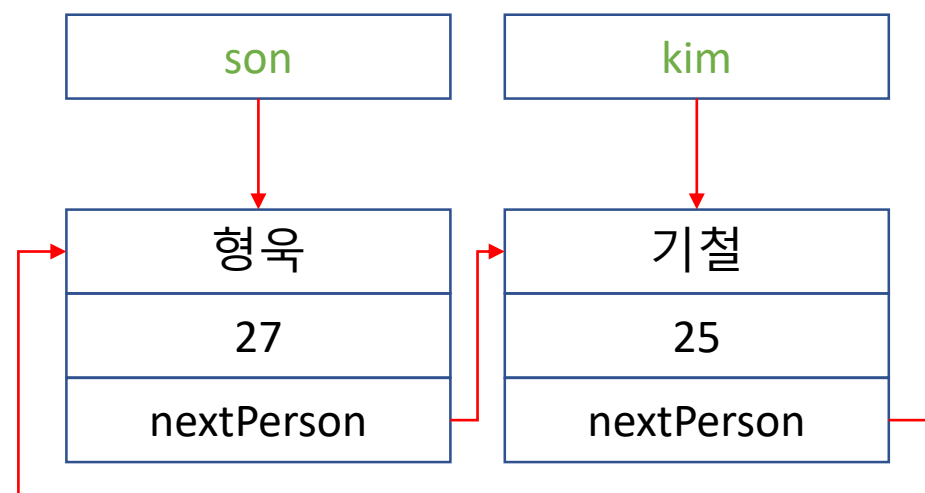
son.reset()을 하면 son은 가리키고 있던 객체를 더 이상 가리키지 않게 되고(reference counting = 0) 그 객체의 reference counting이 1로 줄어든 것 뿐이다. 당연히 son이 가리키던 객체는 메모리에 남아있다...

son이 가리키던 객체가 메모리에 남아있기 때문에 여전히 nextPerson은 kim이 가리키던 객체를 가리키고, kim의 reference counting은 2로 그대로 있던 것이다.

Slide 33-1,2 son과 kim에 객체 할당

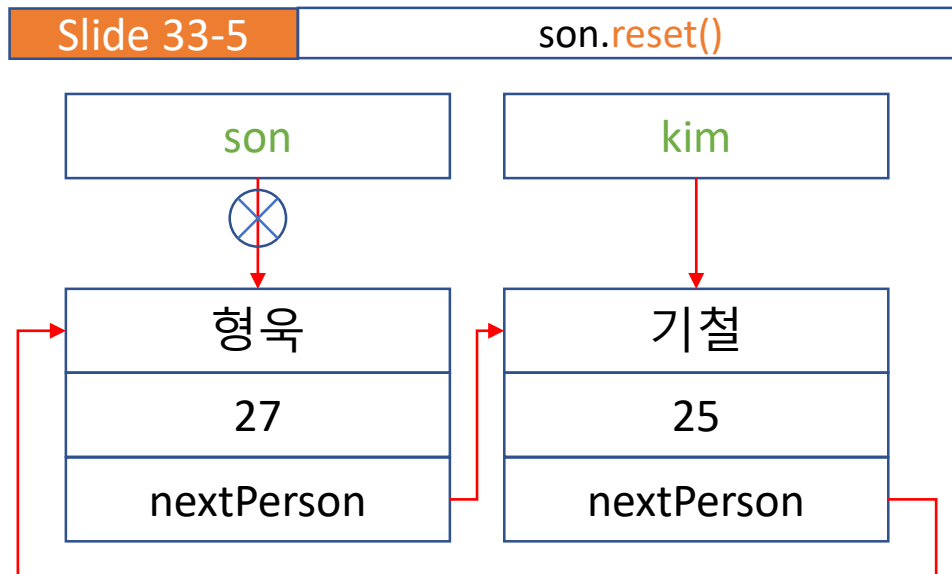


Slide 33-3,4 각 멤버변수가 서로의 객체를 가리킴



## 5. 포인터와 레퍼런스

### 2) shared\_ptr



## 5. 포인터와 레퍼런스

### 3) weak\_ptr

```
class Person
{
private:
    string name;
    int age;
public:
    Person(const string& name, int age) { this->name = name; this->age = age; }
    ~Person() {}
    void ShowPersonInfo() { cout << name << "의 나이는 " << age << endl; }
    //shared_ptr<Person> nextPerson;
    ① weak_ptr<Person> nextPerson;
};

int main()
{
    shared_ptr<Person> son = make_shared<Person>("형욱", 27);
    shared_ptr<Person> kim = make_shared<Person>("기철", 25);
    cout << "kim의 참조 횟수" << kim.use_count() << endl;
    cout << "son의 참조 횟수" << son.use_count() << endl;
    //
    son->nextPerson = kim;
    kim->nextPerson = son;
    cout << "kim의 참조 횟수" << kim.use_count() << endl;
    cout << "son의 참조 횟수" << son.use_count() << endl;
    //
    son.reset();
    cout << "kim의 참조 횟수" << kim.use_count() << endl;
    cout << "son의 참조 횟수" << son.use_count() << endl;

    cout << kim->nextPerson.use_count() << endl;
}
```

Microsoft Visual Studio 디버그 콘솔

```
kim의 참조 횟수 1
son의 참조 횟수 1
kim의 참조 횟수 1
son의 참조 횟수 1
kim의 참조 횟수 1
son의 참조 횟수 0
0
```

1. 기존의 멤버변수가 shared\_ptr → weak\_ptr로 바뀌었다.



## 5. 포인터와 레퍼런스

### 3) weak\_ptr

결론부터 말하자면 weak\_ptr로 가리키는 객체는 reference counting을 증가시키지 않는다. 이 점을 알고 나면 위의 결과를 바로 이해할 것이다.



## 6. 함수

- 함수(function)이란 말 그대로 기능이며 특정한 작업을 수행하기 위한 명령문의 그룹이다. 함수는 표준함수와 사용자 정의 함수로 구분된다.

a) 표준 함수(standard function)는 C와 C++에서 표준 라이브러리로 제공하며 함수의 기능과 호출 방법만 알고 있으면 언제든지 사용 가능하다.

b) 사용자 정의 함수(user defined function) : 개발자가 필요에 따라 만들어 사용하는 함수이다. 선언과 정의를 해야 하고 보통 선언은 헤더파일, 정의는 소스파일에 한다.

- 함수 선언 방법

[반환할 데이터 타입] [함수명](parameter);

- 함수 정의 방법

[반환할 데이터 타입] [함수명](parameter)

```
{  
    명령문;  
    return 반환할 값;  
    //반환할 값이 void 즉 없다면 returns; 혹은 생략  
}
```

- 함수 호출 방법

[함수명](인자값);



## 6. 함수

- 함수 포인터 : 함수의 위치를 가리키는 포인터

일반적으로 사용자 정의 함수를 작성하여 컴파일을 한다면 함수의 위치가 로드되어 함수의 이름만으로도 호출이 가능하지만 dll이나 플러그인 모듈 같이 런타임에 동적으로 가져다 쓰는 경우는 함수의 위치를 가리키는 함수 포인터를 사용한다.

```
void display(string msg) ①
{
    cout << msg << endl;
}

void myfunc(void (*pFunction)(string)) ②
{
    //실행 중 에러가 발생하면 호출 측에 알려준다.
    pFunction("에러 발생!!");
}

void runFunction()
{
    myfunc(display); ③
}
```

결과

Microsoft Visual Studio

에러 발생!!

- 1) 원형함수 정의
- 2) 함수 포인터를 매개변수로 하는 함수 정의
- 3) 호출



## 6. 함수

### ★clean code★ 작성하기

- 4장 5장의 예제 함수를 보고 좀 더 clean한 코드를 작성할 수 있는 여지가 있다. 뭘까?

#### 4-c 중첩if문

```
void ifStatement(int nScore)
{
    if (nScore == 10)
        cout << "학점 A" << endl;
    else if (nScore == 9)
        cout << "학점 A" << endl;
    else if (nScore == 8)
        cout << "학점 B+" << endl;
    else if (nScore == 7)
        cout << "학점 B" << endl;
    else
        cout << "학점 C+" << endl;
}
```

반환형을 토대로 함수 호출 후 상태 값 체크  
Ex) true면 성공, false면 실패

함수 내에서 read-only로 쓰이는 매개변수는 call by reference  
혹은 const 키워드를 붙여 주기

상수는 가독성이 좋은 enum 으로

#### 5-3 포인터 매개변수

```
void pointerParameterSwap(int* num1, int* num2)
{
    int temp;
    temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

인자로 넘어온 포인터 변수가 null값을 가리키는지 반드시 확인!

```
if(!num1 || !num2)
    return;
```





## 6. 함수

### - 함수 포인터의 별칭

기존 함수 포인터의 사용방식은 코드 가독성을 어렵게 할 수도 있어서 typedef로 별칭을 만들어 보다 쉬운 표현으로 사용하기도 한다.

typedef란 사용자 정의 자료형을 말한다. 쉽게 말해서 줄임말이다.

Ex) 버스정류장 -> 버정

Ex) typedef string s : 기존 string 자료형을 s로 쓰겠다. -> typedef 버스정류장 버정 : 버스정류장을 버정으로 쓰겠다.

```
typedef void (*displayPointer)(string);
```

```
displayPointer pTemp = display;  
pTemp("에러 발생!!");
```

```
typedef void (*displayPointer)(string);
```

→ string 매개변수 1개를 가진 함수에 대해 displayPointer 라는 별칭으로 함수 포인터 자료형을 선언하겠다.

```
displayPointer pTemp = display;
```

→ pTemp라는 이름의 함수 포인터 변수를 선언하고 display 함수를 가리킨다.



## 7. 구조체

- 구조체(structure)란 서로 다른 종류의 데이터 항목을 결합하여 하나의 단위로 정의한 데이터 타입이다. 그렇기에 C++에서 기본 제공하는 데이터 타입이 아닌 **사용자 정의 데이터 타입**이라고 볼 수 있다.

- 구조체 선언 및 정의 : `[struct] [태그명] { 멤버 데이터 };`

Ex) `struct Employee {  
 string name;  
 string phoneNo;  
 double pay;  
};`

- typedef문을 이용하여 선언 할 수도 있다. C 같은 경우 typedef 를 사용하지 않으면 구조체 변수를 선언할 때 일일이 struct 키워드를 같이 사용해야만 했다.

Ex) `typedef struct [태그명] { 멤버 데이터 } [별칭];`

tip) C++에서는 typedef 를 쓰지 않아도 태그명만 가지고도 구조체 변수를 선언할 수 있기 때문에 굳이 쓸 필요는 없다.

- 구조체 변수 초기화

`Employee person1 = {"손형욱", "010-1234-5678", 100};`

- C++에서의 구조체는 C와 비교하여 멤버 데이터에 함수, 클래스를 넣을 수 있다. 구조체의 모든 멤버 데이터에 대한 default 접근지정자는 public이며 이후 설명할 클래스 같은 경우 private이다.

- 멤버 데이터 접근

구조체 역시 new 연산자를 통해 동적할당을 할 수 있으며 만약 포인터를 이용한 동적할당은 -> 연산자를 통해 데이터 멤버변수에 접근한다. 정적할당은 . 연산자를 사용한다.

Ex) `person1.name = "아무개";`

Ex) `Employee* person2 = new Employee; // 혹은 new Employee{초기화할 데이터};`



## 8. 클래스 기초

1) 객체지향(object orientation) : 객체라는 개념을 중심으로 소프트웨어 시스템을 구축하는 것.

(C는 함수 단위의 절차지향 프로그래밍 언어)

### ★2) 객체지향의 특징 :

- 추상화(abstraction) : 정보은닉의 수단 중 하나로 공통의 속성이나 기능을 묶어 **클래스**로 정의하는 것  
Ex) 각종 핸드폰들이 공통적으로 가지는 기능을 모아 만든 틀(전화걸기 받기, 번호입력) 아이폰 3gs~11, 갤럭시s1~10 모두 가지고 있는 기능이다.
- 캡슐화(encapsulation) : 실제로 구현 및 동작되는 부분에 대해 데이터를 처리함에 있어 객체의 데이터를 직접 접근할 수 없고 메서드(method)를 통해서만 가능하다.  
Ex) 전화번호를 입력하면 그 번호는 객체 내부에서 동작
- 상속성(inheritance) : 기존 클래스를 확장한 파생 클래스를 정의하는 것  
Ex) 아이폰6s는 추상화 단계의 기능 뿐만 아니라 3gs부터의 기능들을 포함하여 만들어 졌다.
- 다형성(polymorphism) : **오버라이딩** 으로 다양한 형태로 나타낼 수 있는 특징  
Ex) 음성인식 기능 -> 아이폰 : 시리 호출, 갤럭시 : 빅스비 호출

3) 객체 (object) :

- 소프트웨어 시스템을 나누는 **단위**
- **특성(상태)**과 **행위**를 가지는 어떠한 실체  
Ex) 특성(attribute) : 해당 객체에 저장된 정보, 행위(behavior) : 객체가 행동하거나 반응하는 방법  
Ex) 자동차 객체의 특성 : 차체, 엔진, 스티어링, 변속기, 바퀴 등의 부품을 가지고 있음  
자동차 객체의 행위 : 좌회전하다, 우회전하다, 정지하다, 가속하다 등등의 행위를 가지고 있음

4) 객체간 의사소통 : 자동차를 운전하기 위해서(자동차의 행위를 하기 위해서) 그러한 명령을 보내는 상대 객체가 있어야 하고 그건 운전자 이다. 이 경우 운전자(주체)는 자동차(객체)에게 '메시지 보내기'를 한다.



## 8. 클래스 기초

- 5) 클래스(class) : 유사한 특징과 행위를 갖는 객체를 표현하는 모형(틀) 혹은 템플릿. 또한 이러한 관념을 표현한 것을 추상화(abstraction)이라고 하며 객체의 추상화가 클래스이다. 모든 객체는 클래스를 기반으로 생성되며 객체를 클래스의 인스턴스(instance)라고도 말한다.
- 6) 추상적인 데이터 타입(abstract data type) : 프로그래밍 관점에서 클래스를 정의하는 것은 새로운 데이터 타입을 정의하는 것과 같다. 이를 추상적인 데이터 타입 즉 사용자 정의 데이터 타입인 것이다. 7장의 구조체 역시 마찬가지이다. 그러나 구조체로 정의된 데이터 타입은 문제를 일으킬 요소가 많다.

```
//구조체
typedef struct _Date {
    int year;
    int month;
    int day;
}Date;
```

- 1) **Date** 멤버변수(year, month, day)는 따로 초기화를 해주지 않으면 유효하지 않은 날짜(쓰레기 값 혹은 0값)를 가질 수도 있다. 그리고 구조체는 이걸 막을 방법이 없다.
- 2) **Date** tomorrow = today + 1; 또한 구조체는 이런 타입의 연산을 할 수가 없다.
- 3) 메모리를 절약하기 위해 구조체를 bit field 타입으로 정의했고 프로그래밍을 했다면 후에 그 구조를 바꾸기가 쉽지 않다.

### ★ 클래스(class)와 구조체(struct)의 명백한 차이점 ★

기존 C의 구조체보다 기능이 향상된 C++의 구조체는 멤버 함수, 클래스를 가지며 접근지정자를 통한 캡슐화가 실현 가능하다. 구조체와 클래스 그 자체만을 두고 비교한다면 거의 동일한 기능 때문에 개념확립이 모호해질 수 있다.

다만 클래스와 구조체의 명확한 차이 두가지

1. 클래스의 Default 접근지정자는 private, 구조체는 public이다.
2. 클래스는 상속을 통한 객체지향이 실현 가능하다.

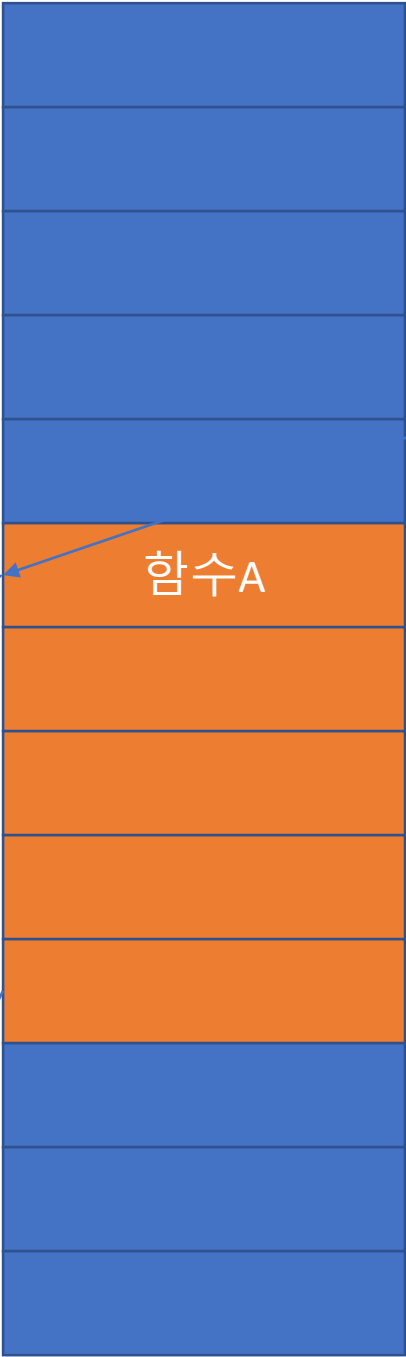


memory

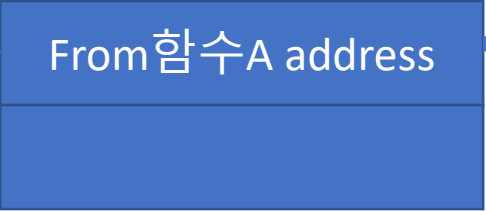
code



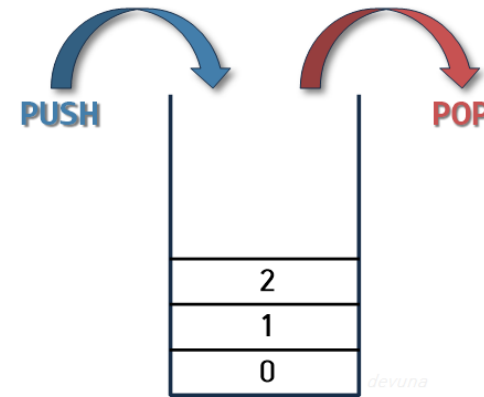
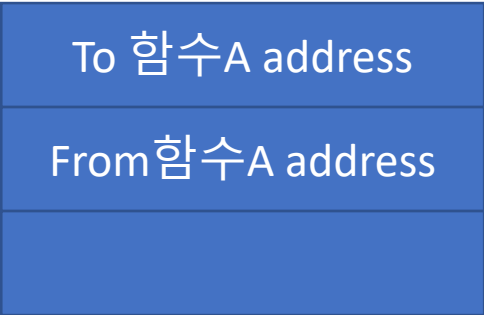
To 함수A address



Call stack



Stack



LIFO (Last In First Out) 방식

Last In First Out

Queue



First In First Out



## 8. 클래스 기초

```
class CDate {    클래스 선언
private:
    int year;
    int month;
    int day;
public:
    CDate();
    CDate(int yy, int mm, int dd);
    ~CDate();
    void setDate(int yy = 2020, int mm = 1, int dd = 1);
    int getYear(void);
    int getMonth(void);
    int getDay(void);
    void displayDate(void);
};

int max(int x, int y);
int min(int x, int y);
```

클래스 선언

멤버 변수

생성자, 소멸자

멤버 함수

- 1) 객체 생성시 생성자를 통해 클래스 멤버변수에 유효하지 않은 값을 걸러낼 수 있다.
- 2) 연산자 오버로딩을 통해 객체 간의 연산을 할 수 있다.

3) 접근지정자

```
CDate::CDate() 생성자 정의
{
    this->year = 1990;
    this->month = 1;
    this->day = 1;
}
```

- 1) 생성자(constructor) : 객체가 생성될 때 호출되는 멤버 함수이며, 멤버 변수에 대한 초기화를 담당한다.
- 2) 소멸자(destructor) : 객체가 소멸될 때 뒤처리를 담당하는 멤버 함수이다. 보통은 동적할당 된 메모리를 해제한다.
- 3) 생성자, 소멸자는 반환 값, return이 없다.



## 8. 클래스 기초

6) 접근 지정자(access specifier) : 접근 지정자는 해당 클래스의 객체의 멤버를 다른 외부 객체에서 접근하려고 할 때 접근이 허용되는지를 지정한다. 즉 추상화와 캡슐화를 실현하기 위함이다.

a) public : 멤버가 외부에 공개되어 어디에서라도 멤버에 접근할 수 있게 한다. 일반적으로 public 접근 지정자를 지정하여 객체의 외부와의 상호작용할 수 있게 한다.

b) private : 멤버를 비공개로 지정하여 외부 객체에서 멤버에 접근할 수 없게 한다. private 멤버에 접근할 수 있는 함수는 해당 클래스의 멤버 함수 뿐이다.

3) protected : 상속 관계에서 파생클래스만이 접근할 수 있는 멤버를 지정한다.

- tip) private으로 감춰진 멤버 변수에 대한 write는 setter(), read는 getter()라 칭한다. get/set은 public 지정자에서 정의되며 클래스 외부에서 접근이 가능하다.

- tip) 일반적으로 데이터 멤버는 private 지정자로 정의하는 것이 좋다. 클래스 내부의 데이터를 외부에 노출시키지 않음으로써 데이터의 유효성, 쉬운 내부 데이터 구조 변경이 보장된다.

7) 멤버 함수 정의 : 클래스 내에 선언된 멤버 함수를 정의할 때는 반드시 영역 결정 연산자(scope resolution operator) :: 를 붙여줘야 한다. 이는 멤버함수가 클래스 내에 포함된다는 것을 의미하기 때문이다.

Ex) 멤버함수 정의 예

```
void CDate::setDate(int yy, int mm, int dd)
{
    int days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    this->year = max(1990, yy);
    this->month = max(1, mm);
    this->month = min(this->month, 12);
    this->day = max(1, dd);
    this->day = min(this->day, days[month]);
}
```

★왜 멤버함수를 사용하여 멤버변수에 접근하는가?

- 캡슐화 실현
- 연산자 오버로딩을 통해 객체에 연산 기능 부여
- 클래스 데이터 구조 변경 용이
- 데이터 멤버에 잘못된 데이터가 들어가지 않게 보장 가능





## 8. 클래스 기초

8) 인스턴스 생성과 **생성자** : 클래스를 선언하는 일은 컴파일러에게 새로운 데이터 타입을 사용하겠다고 알리는 것 뿐이지 실제로 메모리가 할당되는 것은 아니다. 메모리 할당 시점은 클래스 변수를 정의할 때이다. 이 때 클래스 변수를 인스턴스(instance) 즉, 객체(object)라고 한다. 객체도 변수와 같이 정적, 동적할당이 존재한다.

- **생성자(constructor) 특징** : 객체가 생성될 때 호출 됨

- a) 멤버변수 값을 설정하거나, 메모리를 동적 할당 받거나, 파일, 네트워크의 연결 등 객체를 사용하기전 필요한 조치를 할 수 있도록 하기 위함
- b) 생성자는 오직 한번만 실행된다.
- c) 생성자이름은 클래스 이름과 **동일**하다. 이로 인해 다른 멤버 함수와 쉽게 구별된다.
- d) 생성자를 선언&정의 할 때는 리턴 타입을 명시하지 않고 return 명령어도 쓰지 않는다.
- e) 생성자는 **오버로딩**을 통해 중복 가능하다.

아래는 **오버로딩**을 통해 정의된 두 개의 생성자 이다.

```
CDate::CDate()
{
    ★ this->year = 1990;
    this->month = 1;
    this->day = 1;
}

CDate::CDate(int yy, int mm, int dd)
{
    setDate(yy, mm, dd);
}

void CDate::setDate(int yy, int mm, int dd)
{
    int days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    this->year = max(1990, yy);
    this->month = max(1, mm);
    this->month = min(this->month, 12);
    this->day = max(1, dd);
    this->day = min(this->day, days[month]);
}
```

```
CDate today;
CDate anyday(2020, 02, 17);
```

- 객체 today는 CDate() 생성자를 호출
- 객체 anyday는 CDate( parameter) 생성자를 호출

```
CDate* today = new CDate();
CDate* anyday = new CDate(2020, 2, 17);
```

- 동적할당도 동일하게 생성자를 호출



## 8. 클래스 기초

- (참고) 위임생성자(delegating constructor) : 생성자가 다른 생성자를 호출  
한 클래스의 생성자들은 대개 객체를 초기화하는 비슷한 코드가 중복되는 경우가 많다. 위 슬라이드의 생성자  
코드는 위임 생성자로 코드를 간소화 할 수 있다.

위임생성자 →

타겟생성자 →

```
CDate::CDate() : CDate(1990, 1, 1) {};  
CDate::CDate(int yy, int mm, int dd)  
{  
    setDate(yy, mm, dd);  
}  
void CDate::setDate(int yy, int mm, int dd)  
{  
    int days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  
    this->year = max(1990, yy);  
    this->month = max(1, mm);  
    this->month = min(this->month, 12);  
    this->day = max(1, dd);  
    this->day = min(this->day, days[month]);  
}
```

1. 생성자 호출  
2. 매개변수 생성자 호출,  
yy, mm, dd에 1990, 1, 1 전달

- 객체 생성 및 매개변수 없는 생성자 호출

```
CDate* today = new CDate();
```

이름	값
today	0x011ef3e0 {year=1990 month=1 day=1}
year	1990
month	1
day	1



## 8. 클래스 기초

### ★생성자는 꼭 있어야 하는가?★

생성자를 선언하지 않은 클래스는 어떻게 될까? 사실 **생성자가 없는 클래스는 근본적으로 존재할 수가 없다!!!!**  
다만 생성자를 정의하지 않은 클래스에 대한 객체를 생성할 때 컴파일러가 매개변수가 없는 기본 생성자(default constructor)를 만들어 삽입하고, 삽입된 생성자를 호출하도록 컴파일 한다.

Ex) 생성자가 선언되지 않은 CCircle클래스

```
class CCircle {
public:
    double getArea();
private:
    int radius;
};
int main(void)
{
    CCircle c1;
    return 0;
}
```

1. 컴파일

2. 컴파일러에 의해  
자동 삽입

```
class CCircle {
public:
    double getArea();
    //CCircle();
private:
    int radius;
};
/*
CCircle::CCircle()
{
}
*/
int main(void)
{
    CCircle c1;
    return 0;
}
```

3. 호출

★매개변수가 있는 생성자가 단 하나라도 선언되어 있는 클래스는 기본 생성자가 자동으로 생성되지 않는다.



## 8. 클래스 기초

9) 객체의 멤버함수, 변수에 대한 접근 : 일반적인 변수로 정의되는 객체는 . 연산자, 포인터로 정의되는 객체는 -> 연산자를 이용한다.

```
class CAccess {
private:
    int a;
    void privateFunc() {}
    CAccess() { this->a = 1; this->b = 1; };
public:
    int b;
    CAccess(int x) { this->a = x; this->b = x; }
    void publicFunc() {};
};

int runExample()
{
    CAccess objA; ①
    CAccess objB(100);
    objB.a = 10; ②
    objB.b = 20;
    objB.publicFunc();
    objB.privateFunc(); ③
    return 0;
}
```

1) 생성자가 private에 정의, runExample(외부) 함수에서 접근 불가능

2) a는 private에 정의, 접근 불가능

3) privateFunc()는 private에 정의, 접근 불가능



## 8. 클래스 기초

10) 소멸자(destructor) : C++ 객체는 생성 이후 소멸되면서 객체 메모리는 시스템으로 반환된다. 생성자와 마찬가지로 객체가 소멸할 때 소멸자가 호출된다.

- 소멸자(destructor)의 특징 : 객체가 소멸할 때 호출 됨

★객체는 언제 소멸하는가? 객체는 기본적으로 기본 데이터 타입의 변수와 같은 영역을 가지기 때문에 객체를 생성한 블록이 끝나면 소멸된다. 전역으로 생성된 객체는 프로그램이 끝날 때, 동적으로 생성한 객체는 delete 연산자를 호출할 때 소멸된다.

a) 객체가 사라질 때, 동적으로 할당 받은 메모리를 운영체제에 돌려주거나, 열어 놓은 파일을 저장하고 닫거나, 연결된 네트워크를 해제하는 등 객체가 사라지기 전에 필요한 조치를 하기 위함

b) 소멸자의 이름은 클래스 이름 앞에 ~를 붙인다.

c) 소멸자를 선언&정의 할 때는 리턴 타입을 명시하지 않고 return 명령어도 쓰지 않는다.

d) 소멸자는 오직 한 개만 존재하며 매개변수를 가지지 않는다.

e) 소멸자가 선언되어 있지 않으면 기본 소멸자(default destructor)가 자동으로 생성된다. 단 아무 일도 하지 않는다.

- 아래는 멤버변수 radius를 10으로 동적생성 및 초기화 하여 getArea()로 면적을 구하는 클래스이다.

### 클래스 선언

```
class CCircle {
public:
    double getArea();
    CCircle();
    ~CCircle();
private:
    const double PI = 3.1415;
    int* radius;
};
```

### 클래스 구현 및 호출

```
double CCircle::getArea()
{
    return (*this->radius) * (*this->radius) * PI;
}

CCircle::CCircle() 객체 c1을 생성하면서 생성자
                  호출
{
    cout << "생성" << endl;
    this->radius = new int(10); 생성자에서 할당한 메모리
                              반환
}

CCircle::~~CCircle() main문이 종료되면서 호출
{
    cout << "소멸" << endl;
    delete this->radius;
}

int main(void)
{
    CCircle c1;
    cout << c1.getArea() << endl;
```

### 결과

Microsoft Visual Studio 디버그 콘솔

생성  
314.15  
소멸

## 8. 클래스 기초

지금까지 예제 코드를 보면서 **this**라는 키워드를 많이 보았을 것이다.

11) **this** 포인터 : 객체 자신에 대한 포인터로서 클래스의 멤버 함수 내에서만 사용된다. **this**는 객체의 멤버 함수가 호출될 때, 컴파일러에 의해 보이지 않는 **매개변수**로 전달되는 객체에 대한 주소.

**쉽게 말해서 객체의 주소와 함께 호출되는 thiscall 이라고 한다.**

- **this** 포인터는 멤버 변수와 매개변수의 이름이 같을 때 사용
- 객체 자신의 주소를 반환할 때 사용 : 연산자 **오버로딩**에서 많이 사용된다.
- **정적 멤버** 함수(static member function)에서는 **this**를 사용할 수 없다.



---

## 소프트웨어 명세서

전화기

Class Name : KPhone

이제 클래스로 명세화 하시고  
클래스를 설계 및 구현하세요.

기능

Send() : 전화를 걸다

Receive() : 전화를 받다

Numbers() : 전화번호를 입력한다.

Kphone.h  
Kphone.cpp

멤버변수

Cancel() : 전화를 끊다

private:

strNumber : 전화번호

isConnected : 전화중인가?



## 소프트웨어 명세서

학생 성적

Class Name : StudentScore

이제 클래스로 명세화 하시고  
클래스를 설계 및 구현하세요.

기능

SetStudentName(string name) : 학생 이름

SetSubjectScore(string subject, int score) : 과목, 성적 입력

DoCalc() : 결과 산출

홍길동 = 국어:90, 영어:90, 수학:98, 미술:88, 음악:94, 역사:99  
총합 = ??, 평균 = ??, 최소 = ??, 최대 = ??

이순신 = 국어:90, 영어:90, 수학:98, 미술:88, 음악:94, 역사:99  
총합 = ??, 평균 = ??, 최소 = ??, 최대 = ??

신사임당 = 국어:90, 영어:90, 수학:98, 미술:88, 음악:94, 역사:99  
총합 = ??, 평균 = ??, 최소 = ??, 최대 = ??

StudentScore.h  
StudentScore.cpp

멤버 변수

private:

GetSum() : 과목 총 합

GetAvg() : 과목 평균

GetMin() : 과목 최소

GetMax() : 과목 최대

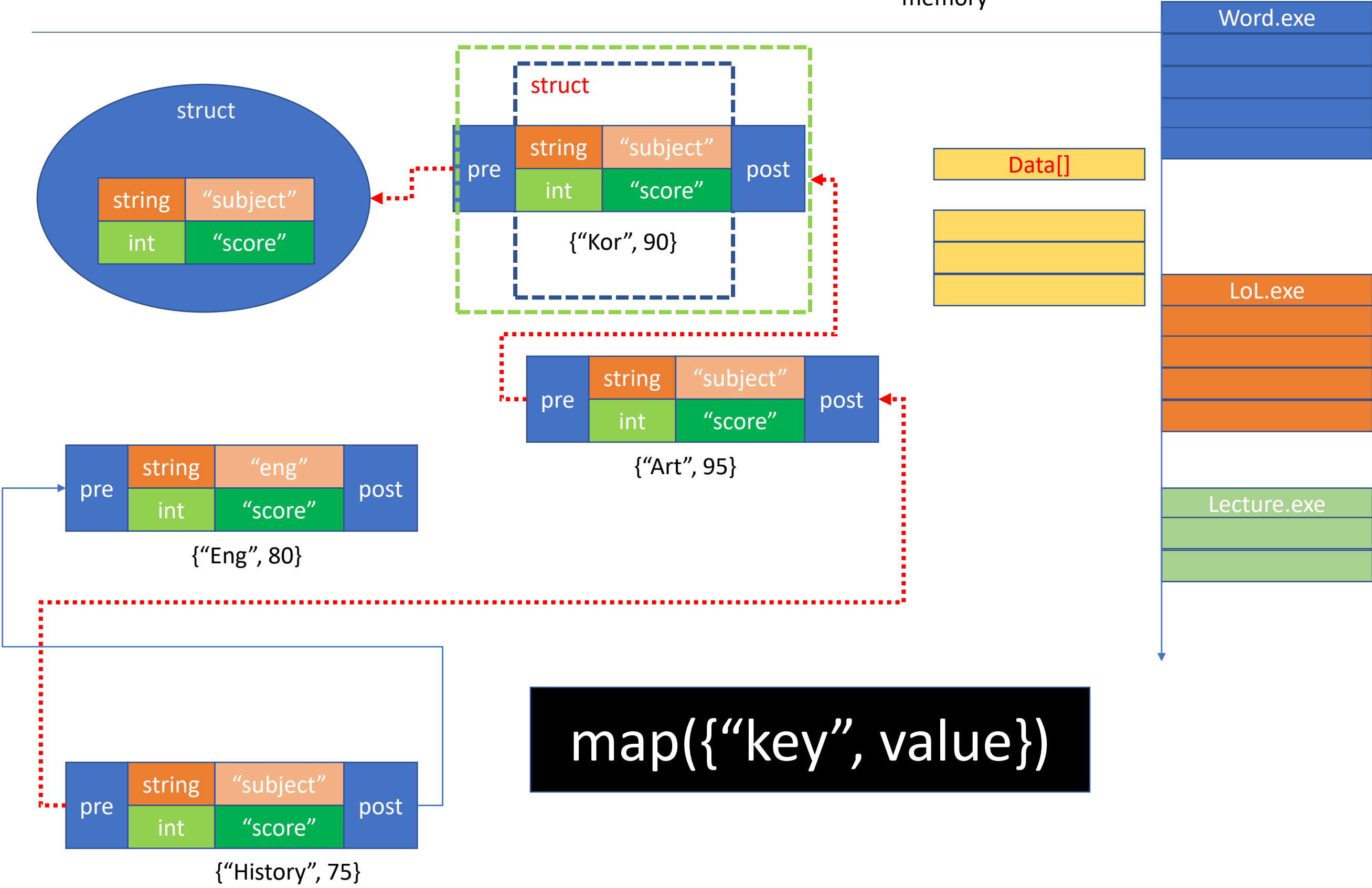
```
Struct StudentInfo  
{  
    string name;  
    int eng  
    int math  
    int kor  
    int sum  
    float avg  
    int min  
    int max  
}
```

StudentInfo stStudentInfo





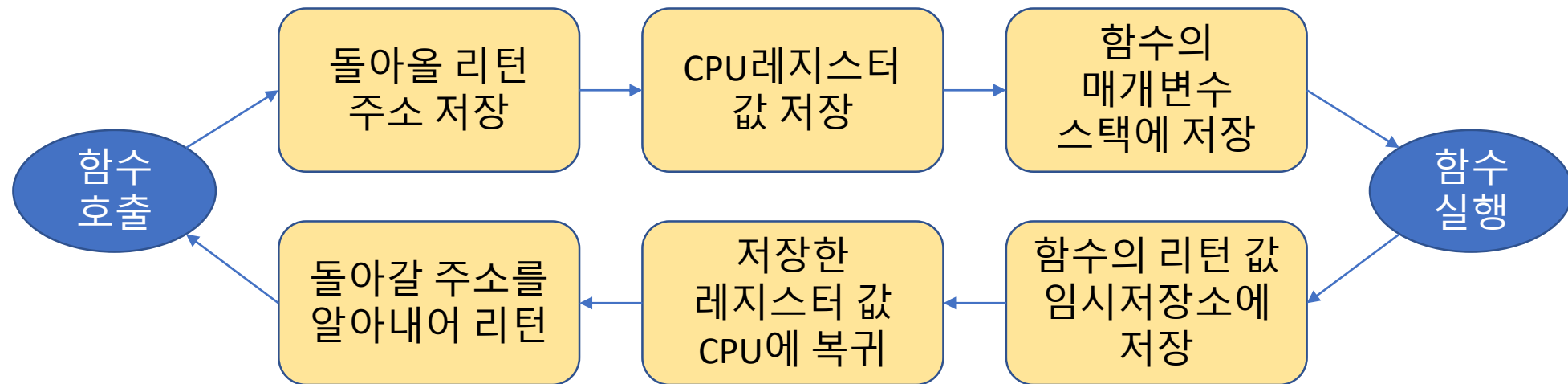
memory



## 8. 클래스 기초

### ★inline 함수★

- 컴파일러가 인라인 함수를 호출하는 곳에 인라인 함수의 코드를 그대로 삽입하여 함수 호출이 일어나지 않게 하는 것



- 함수 호출에 따른 시간 **오버헤드(overhead)** 시간이 무시할 수 없는 비중을 차지하는 경우 inline을 통해 실행시간을 개선할 수 있다.

```
int odd(int x)
{
    return x % 2;
}

int main()
{
    int sum = 0;
    for (int index = 0; index <= 10000; index++)
    {
        if (odd(index)) 10001번 호출
            sum += index;
    }
    cout << sum;
}
```

확장 컴파일

```
inline int odd(int x)
{
    return x % 2;
}

int main()
{
    int sum = 0;
    for (int index = 0; index <= 10000; index++)
    {
        if (index%2)
            sum += index;
    }
    cout << sum;
}
```



## 8. 클래스 기초

### ★멤버 함수의 인라인 선언과 자동 인라인★

```
class CCircle {  
public:  
    inline double getArea() { return PI * radius * radius; }  
    inline CCircle() { this->radius = 1; }  
    ~CCircle() {};  
private:  
    const double PI = 3.1415;  
    int radius;  
};
```

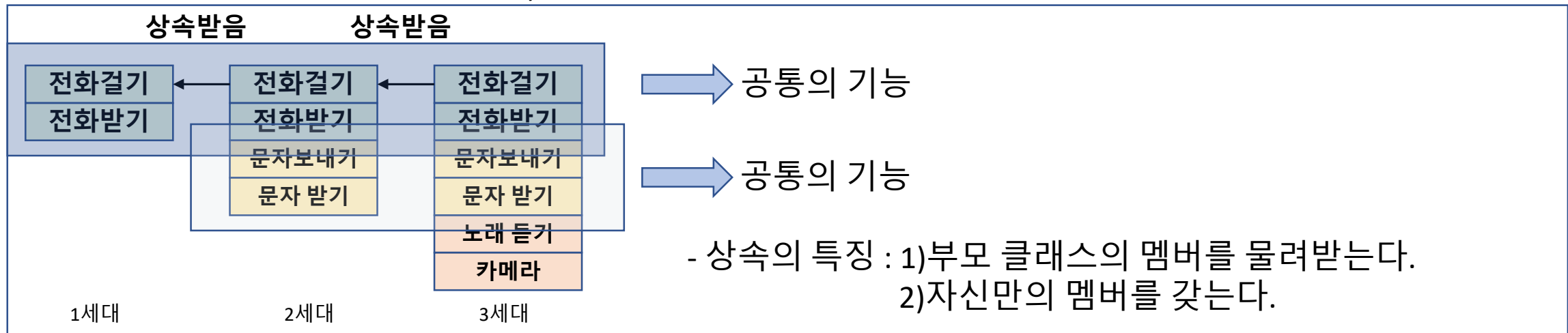
선언부에 정의한다면 자동  
inline함수로 인식한다.



## 9. 상속

- 상속(inheritance) : 객체나 클래스가 마치 부모와 자식 같은 관계를 맺어 계층적인 구조를 이루는 것이다. 부모 클래스는 기초 클래스(base class) 혹은 슈퍼 클래스(super class)라고 하며, 자식 클래스 파생 클래스(derived class) 혹은 서브 클래스(sub class)라고도 한다.

- 상속을 쉽게 이해할 수 있는 예시 Ex) 전화기, 도형



- 상속의 목적과 장점

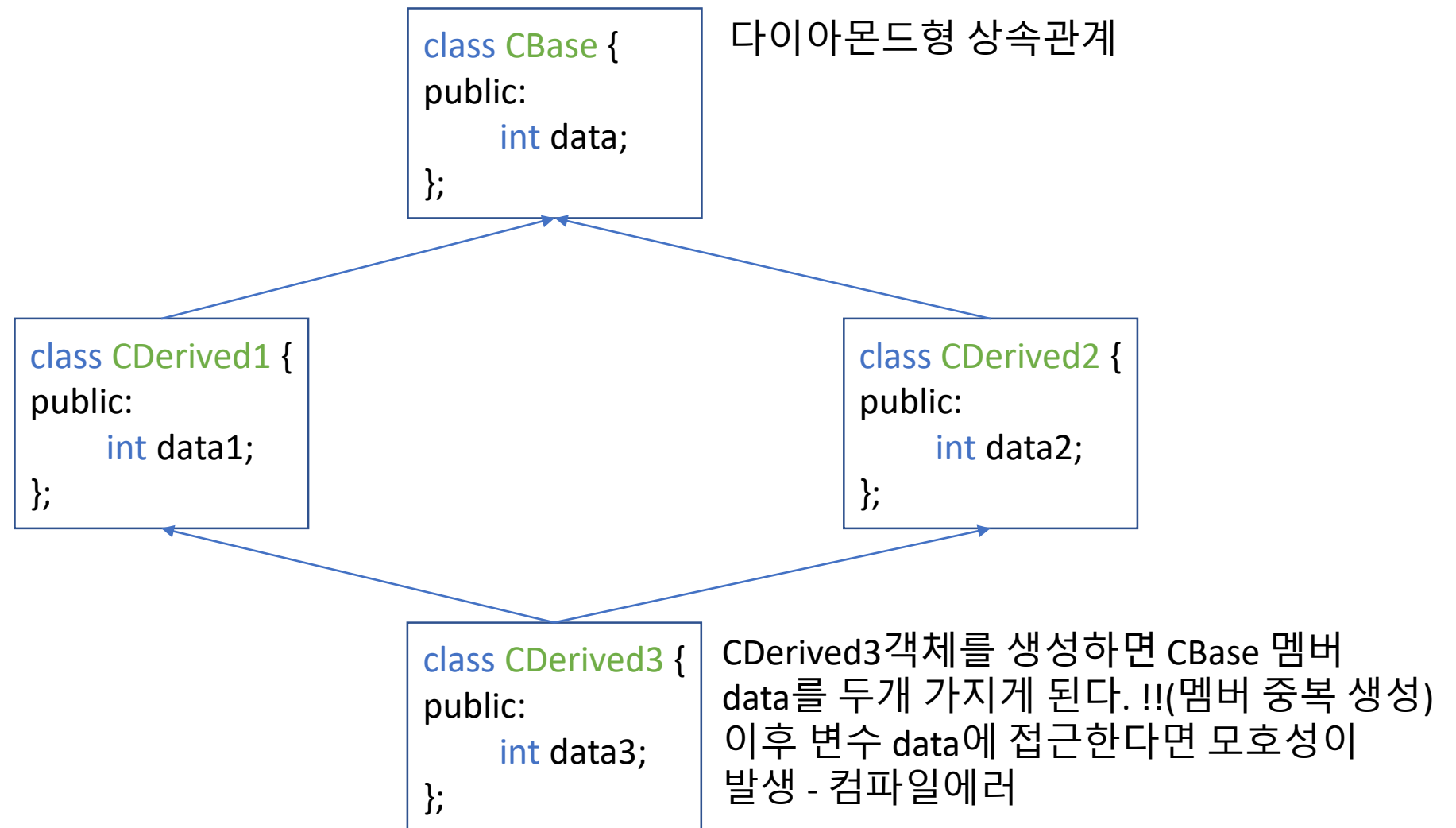
- 간결한 클래스 작성<공통적인 코드는 상위 클래스에>
- 클래스 간의 계층적 분류 및 관리의 용이함
- 클래스 재사용과 확장을 통한 소프트웨어 생산성 향상

- 주의 : 기능이 필요하다는 이유만으로 연관성 없는 클래스를 상속해서는 안 된다. 마구잡이식 상속은 객체지향의 의미를 퇴색시킨다. 즉 클래스 본연의 기능과 목적에 맞게 상속을 해야 한다. Ex) 애플 워치가 1세대(위 그림) 기능을 가진다 해서 1세대를 상속받는 일은 없어야 할 것, 또한 **다중상속 ... 다음장에 계속**



## 9. 상속

### ★다중상속의 문제점★



- CDerived1, CDerived2가 CBase를 **virtual 키워드로 상속** 받는다면 문제가 해결된다.

Ex) `class CDerived1 : virtual public CBase { ... };`



# 9. 상속

Ex) 직원 클래스

```
class CEmployee { 기본 클래스
public:
    CEmployee(string name, string address, string telNo, CDate joinDate);
    void displayEmployee();
    //virtual double payCheck();
private:
    string m_name;           //사원명
    string m_address;        //주소
    string m_telNo;          //H.P
    CDate m_joinDate;        //입사일
};

CEmployee::CEmployee(string name, string address, string telNo, CDate joinDate)
{
    this->m_name = name;
    this->m_address = address;
    this->m_telNo = telNo;
    this->m_joinDate = joinDate;
}

void CEmployee::displayEmployee()
{
    cout << "name = " << this->m_name << endl;
    cout << "address = " << this->m_address << endl;
    cout << "telNo = " << this->m_telNo << endl;
    cout << "joinDate = "; this->m_joinDate.displayDate();
}
```

기본클래스구현

클래스구조
Cemployee(class)
CEmployee() 생성자
~CEmployee() 소멸자
m_name
m_address
m_telNo
m_joinDate
displayEmployee()

- 1) 생성자 구현 : 멤버 변수 초기화
- 2) 멤버함수 구현 : 멤버 변수 상태 값 출력



## 9. 상속

Ex) 정규직원 클래스 (직원 클래스를 상속 받음)

```
class CRegularEmployee : public CEmployee { 파생 클래스
public:
    CRegularEmployee(string name, string address,
        string telNO, CDate joinDate, double salary);
    double payCheck();
protected:
    double m_salary;
};

CRegularEmployee::CRegularEmployee(string name, string address, string telNo,
    CDate joinDate, double salary) :
    CEmployee(name, address, telNo, joinDate), m_salary(salary){
    1
}

double CRegularEmployee::payCheck()
{
    2
    return this->m_salary;
}
```

파생클래스구현

기존클래스  
영역

CRegularEmployee(class)
CEmployee() 생성자
~CEmployee() 소멸자
m_name
m_address
m_telNo
m_joinDate
displayEmployee()

+

파생클래스  
영역

CRegularEmployee()
~CRegularEmployee()
m_salary
payCheck()

- 1) 생성자 구현 : 파생클래스의 객체가 생성되면서 생성자를 호출하는데, 이 때 기본 클래스의 생성자를 명시적으로 호출하기 위해 변형한 사례, 파생 클래스 내의 기본 클래스 영역 초기화 및 파생클래스의 salary 멤버변수 초기화
- 2) 멤버 함수 구현 : 급여 값 반환
- 3) 파생클래스 객체는 기존 클래스 영역에서 private지정자에 선언된 멤버 이외의 모든 멤버에 접근 가능



부모

```
class CBase {  
public:  
    int data;  
};
```

```
class CDerived1 {  
public:  
    int data1;  
};
```

부모

자식





## 9. 상속

Ex) 테스트 Program

```
void testProc()
{
    1 CRegularEmployee regularEmp1("손형욱", "화성시", "1234-5678",
      CDate(2019, 11, 1), 1000);
    2 regularEmp1.displayEmployee();
    3 std::cout << regularEmp1.payCheck() << std::endl;
}
```

1) CRegularEmployee 파생 클래스에 대한 regular 객체 생성

2) 상속받은 CEmployee 클래스의 displayEmployee() 호출  
→ displayEmployee() 기본클래스의 public 지정자에서 선언되었기 때문에 가능하다.

3) CRegularEmployee 파생 클래스의 payCheck() 호출

 Microsoft Visual Studio 디버그 콘솔

```
name = 손형욱
address = 화성시
telNo = 1234-5678
joinDate = 2019-11-1
1000
```



## 9. 상속

### ★up casting & down casting★

<class의 구조를 한눈에 보기위해 선언과 동시에 구현함>

**업 캐스팅**은 기본 클래스가 파생클래스를 가리키는 것이다.

우측의 예제는 과자(CSnack)라는 기본 클래스와 그 과자를 상속받는 과자A, B(CSnackA~B)가 있다. 이 과자를 먹는 아이(CChild) 클래스도 존재한다.

과자를 먹기 위해서는 봉지를 뜯어야 한다. 따라서 아이(CChild) 클래스에 뜯기(release), 먹기(eat) 멤버 함수가 존재한다. 이 멤버 함수의 매개변수의 타입은 모두 CSnack\* 이다.

Question) 그냥 각각의 과자에 뜯기 먹기를 추가하면 되지 않나?

Answer) 객체의 기능 및 장점에 위배된다.(코드가 길어지거나 재사용성이 떨어짐)

Up casting을 하는 이유는 코드를 더욱 구조적으로 탄탄하게 또 generic하게 짜기 위함이다. 즉 기본 클래스로부터 파생된 여러 파생클래스를 일관성 있게 관리 할 수 있다. 쉽게 말하면 아이가 어떤 과자를 먹든지 코드가 일관되게 동작해야 한다.

```
void runUpcastingProc()
{
    CSnackA snackA;
    CSnackB snackB;
    CChild child;

    child.releaseSnack(&snackA);
    child.eatSnack(&snackA);

    child.releaseSnack(&snackB);
    child.eatSnack(&snackB);
}
```

### 결과

Microsoft Visual Studio 디버그 콘솔

```
SnackA봉지를 뜯었다!!
그리고 먹는다
SnackB봉지를 뜯었다!!
그리고 먹는다
```

```
class CSnack {
public:
    CSnack() {}
    void setStatus() { this->snackStatus = "밀봉해제"; }
    void setSnack(string m_snackName)
    {
        this->snackName = m_snackName;
        this->snackStatus = "밀봉";
    }
    void display() { cout << snackName << snackStatus << endl; }
    string getName() { return this->snackName; }
private:
    string snackName;
    string snackStatus;
};

class CSnackA : public CSnack {
public:
    CSnackA() { setSnack("SnackA"); }
};

class CSnackB : public CSnack {
public:
    CSnackB() { setSnack("SCnackB"); }
};

class CChild {
public:
    CChild() {}
    void releaseSnack(CSnack* snack) {
        snack->setStatus();
        cout << snack->getName() + "봉지를 뜯었다!!" << endl;
    }
    void eatSnack(CSnack* snack) {
        cout << " 그리고 먹는다 " << endl;
    }
};
```



## 9. 상속

### ★up casting & down casting★ 코드 설명

```
class CChild {
public:
    CChild() {}
    void releaseSnack(CSnack* snack) {
        snack->setStatus();
        cout << snack->getName() + "봉지를 뜯었다!!" << endl;
    }
    void eatSnack(CSnack* snack) {
        cout << " 그리고 먹는다 " << endl;
    }
};

void runUpcastingProc()
{
    CSnackA snackA;
    CSnackB snackB;
    CChild child;

    child.releaseSnack(&snackA);
    child.eatSnack(&snackA);

    child.releaseSnack(&snackB);
    child.eatSnack(&snackB);
}
```

release와 eat함수의 매개변수를 보면 **CSnack\*** snack이다. 함수를 호출하고 인자 값이 넘어가면 **CSnack\*** snack = &snackA 이 되고 CSnack이 snackA(CSnack으로 자동 형 변환)를 가리킨다.

이렇게 되면 snack은 CSnack의 멤버 변수에 접근이 가능하다. 때문에 쉽고 간편하게 밀봉해제를 할 수 있는 것이다. SnackA와 SnackB는 CSnack의 멤버를 확장하고 자신만의 **고유기능(상태, 행위)**을 가진다. 그러나 공통적으로 동작하는 행위에 대한 접근을 개별적으로 한다면 코드가 상당히 길어지고 복잡해진다. 이를테면 각각의 과자 A, B 클래스에서 과자의 밀봉상태를 해제 하는 함수를 만들던가 하는..

과자에 대한 객체가 수 없이 많아진다면 일일이 관리하기가 힘들 것이다.

- down casting : **기본 클래스가 가리키던 객체를 파생클래스의 포인터로 가리키는 것**

업캐스팅을 하면 하나의 기본클래스로 여러 파생클래스를 관리할 수 있어 편하지만 파생클래스 각자의 고유 기능을 접근하기에는 불가능하다. 그 때는 다운캐스팅을 한다. 중요한 점은 다운캐스팅을 할 때는 반드시 명시적 형 변환이 따라줘야 한다. 기본 클래스는 하난데 파생클래스는 여러 개가 있을 수 있는 상황에서 **기본 클래스가 가리키던 객체가 무엇인지 모르기 때문이다.**



## 9. 상속

### - dynamic cast

dynamic cast는 상속관계에 있는 기본 클래스(base)와 파생 클래스(derived) 사이에서 클래스 형 변환을 하기 위해 C++에서 지원하는 명시적 캐스팅이다. (주로 **down casting**)

dynamic\_cast는 이름 그대로 동적인 형 변환이며, 런 타임 시(프로그램 동작) 형 변환에 쓰이는 클래스 타입에 대한 질의를 한다. (**RTTI** : Run Time Type Information) 객체의 캐스트가 유효한지 타입체크를 한다는 말이다. 만약 유효하지 않는다면 null값을 return 하게 된다. 단 RTTI는 가상함수(virtual function)을 가지는 클래스만 사용이 가능하다. 단점이라 한다면, 상속관계가 깊은 클래스에서 실행속도가 느리다.

**쉽게 생각해보자!! down casting에 대한 이해는 조금 뒤에 나온다.**

```
Derived* pD = dynamic_cast<Derived*>(pB);
```

내가 다운캐스팅을 진행하고자 할 때 pB가 가리키고 있는 객체가 Derived 클래스가 아니다? null값을 뱉어내는 것이다.

우리는 dynamic\_cast 말고도, typeid, type\_info 연산자를 통해 RTTI를 이룩할 수 있다.  
자세한 내용은 구체적 예시를 담은 곳에서 확인하기

<https://genesis8.tistory.com/109>

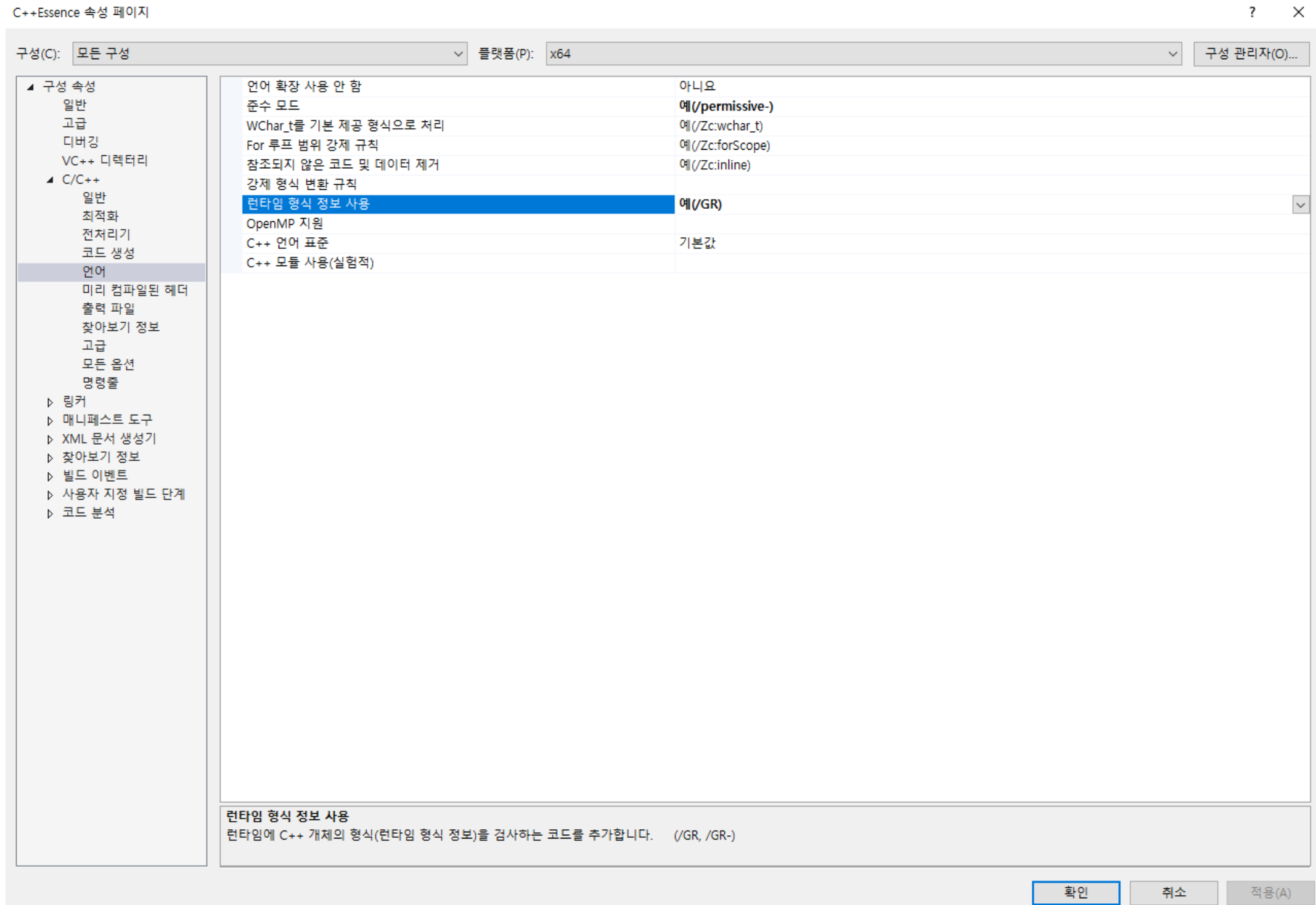
<https://vallista.tistory.com/entry/C-RTTI-RunTime-Type-Information>

static\_cast는 안되는가? 된다 단, 프로그래머는 부모 자식 간 상속관계를 정확히 파악하고, 형 변환 연산 이전에 자신이 구현한 코드에서 무엇이 어떤 객체를 가리키고 있는지 알고 있어야만 한다.



## 9. 상속

RTTI 설정 : 프로젝트 속성 -> C/C++ -> 언어 -> 런타임 형식 정보 사용 -> 예



## 9. 상속

- reinterpret cast

reinterpret\_cast는 임의의 Type끼리 큰 제약 없이 형 변환을 시킨다. 때문에 위험성이 높으며 그 리스크는 프로그래머의 몫이다. reinterpret\_cast를 진행할 때는 형 변환 연산에 관여하는 Type과 형 변환의 목적을 명확히 하고 사용해야 한다.

Ex) 정수형 값을 포인터 타입으로 바꾸어 메모리 주소 값을 가리키게 함

```
int addr = 0x6AFA;  
int* pAddr = reinterpret_cast<int*>(addr);
```

pAddr이 가지는 값은 0x6AFA 이다.



## 10. 다형성

### ★가상함수 & 오버라이딩 그리고 함수 재정의★

전 슬라이드에서 클래스의 공통의 기능과, 업캐스팅에 대해 설명을 했다. 그런데 만약에 공통의 기능을 하지만 다른 작업을 수행하여 다른 결과를 도출 한다면?

예를 들어 기본 클래스인 자동차 클래스 내에 연료소비()의 기능이 있고, 파생 클래스의 각각의 자동차가 각기 다른 연료를 사용하는 경우가 있을 것이다. 또한 각각의 다른 연료는 다른 물질을 배출해낸다. 그렇다면 우리는 이 각각의 파생 클래스에 대해 연료소비()에 대한 정의를 다르게 내려야 한다. 기본 클래스의 연료소비()에서 확장하여 다른 기능을 구현하는 것인데 이를 메서드 재정의(redefine)라고 한다. 다음의 예시코드를 보자.

```
class CCar {
public:
    CCar() {}
    void fuel() { cout << "자동차 꺾대기입니다." << endl; }
};

class Ci8 : public CCar {
public:
    Ci8() {}
    void fuel() { ①
        cout << "i8 연료 : 가솔린" << endl;
    }
};

class Csonata : public CCar {
public:
    Csonata() {}
    void fuel() { ①
        cout << "sonata 연료 : 전기 + 가솔린" << endl;
    }
};
```

```
void runRedefineProc()
{
    Ci8 i8, *pCi8;
    pCi8 = &i8;
    pCi8->fuel(); ②

    CCar* pCar;
    pCar = pCi8; ③
    pCar->fuel();
}
```

- 1) 각 클래스에 대해 fuel() 멤버함수 재정의
- 2) 파생클래스 타입 포인터로 파생클래스 객체 가리키기
- 3) 기본클래스 타입 포인터로 파생클래스 객체 가리키기(up casting)

2), 3)에 대한 결과는?



## 10. 다형성

### 결과

Microsoft Visual Studio 디버그 콘솔

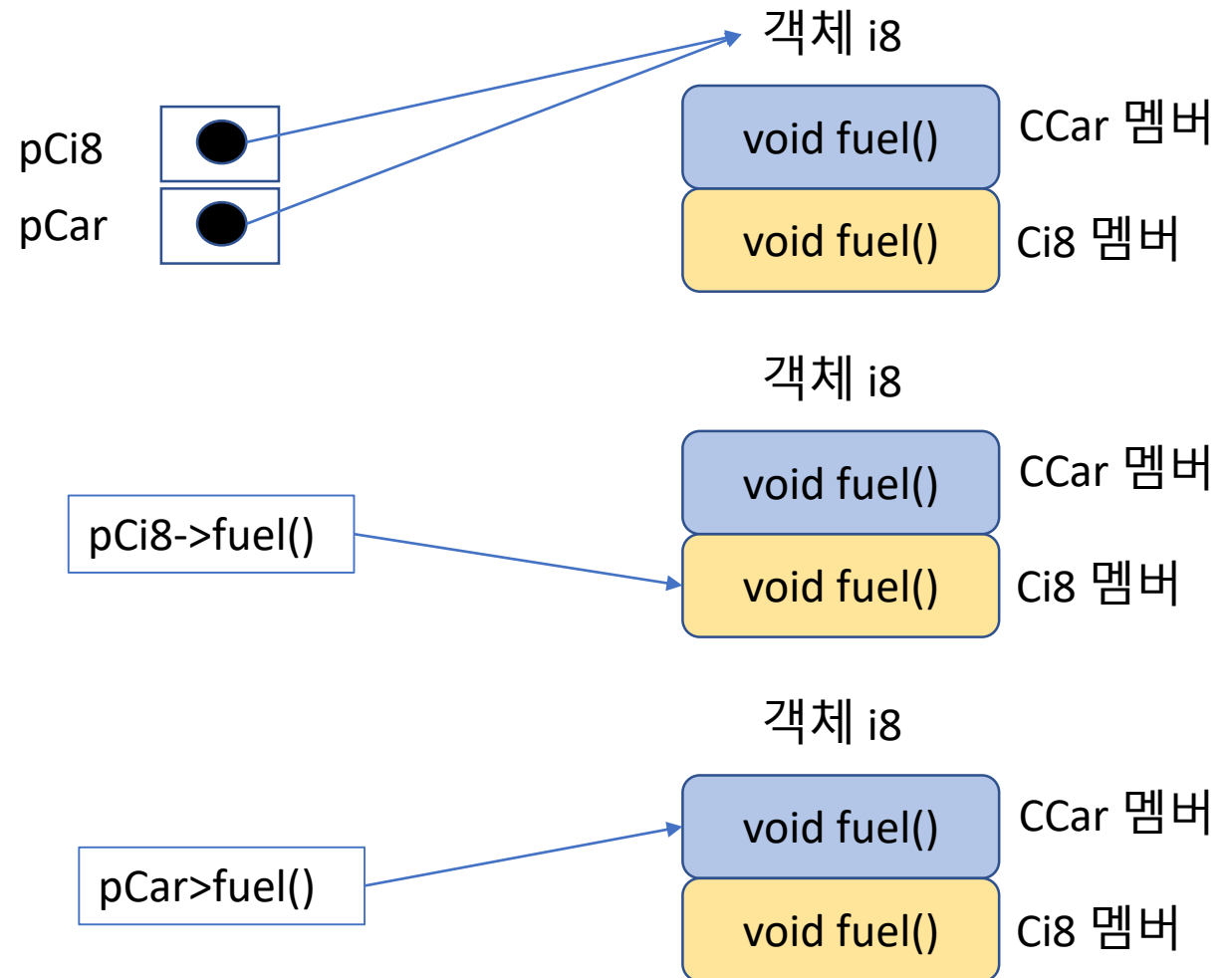
```
i8연료 : 가솔린  
컬데기 클래스
```

그림과 같이 객체 i8은 두 개의 fuel()  
함수를 가지고 있다.

Ci8\* pCi8 = &i8 의 경우 Ci8타입의 포인터  
변수가 Ci8 타입의 객체를 가리키기 때문에  
컴파일러는 Ci8 멤버 함수를 우선적으로  
**바인딩** 한다.

반면에

CCar\* pCar = \*pCi8의 경우 CCar타입이  
Ci8을 가리키는 경우인데, 이 때  
컴파일러는 fuel()멤버 함수에 대해 CCar  
멤버 함수를 우선적으로 **바인딩** 한다.



- tip) `pCi8->CCar::fuel()` 은 `CCar`의 `fuel()`을 호출한다. 범위 지정 연산자 `::`로 재정의된 함수를 구분하여 호출 가능하다.

- tip) 결론은 어떤 타입의 포인터가 가리키고 있냐에 따라 어떤 함수를 호출하는지 달라진다.

이쯤 되면 원하는 함수를 호출하기 위해 어떤 타입의 포인터로 어떤 타입의 인스턴스를 가리켜야 할지 막 헷갈리기 시작한다.





## 10. 다형성

그럴 땐 가상함수 선언을 통한 오버라이딩을 한다. 각각의 클래스마다 항상, 언제든지 그 고유의 기능을 사용하고 싶을 때 말이다. 가상함수 선언에 필요한 키워드는 virtual이고 멤버 함수 선언 앞에 붙인다.

Ex) [virtual] [반환타입] [함수명](매개변수);

- virtual 키워드는 컴파일러에게 자신에 대한 호출 바인딩을 실행 시간까지 미루도록 지시하는 키워드이다. 즉 가상함수가 호출되면 실행 중에 객체 내에 오버라이딩된 가상 함수를 동적으로 찾아 호출하는 것이다. 앞선 예제 함수 재정의(redefine)가 컴파일 시간 다형성(compile time polymorphism)이라면(정적 바인딩), virtual 키워드를 이용한 오버라이딩은 실행 시간 다형성(run time polymorphism)을(동적 바인딩)실현시킨다. 즉 어떤 타입의 포인터가 가리키든 상관 없이 참조되는 객체의 멤버 함수가 호출 된다.

헤더	소스	호출
<pre>class CCar { public:     CCar() {}     virtual void fuel(); };  class Ci8 : public CCar { public:     Ci8() {}     virtual void fuel(); };  class Csonata : public CCar { public:     Csonata() {}     virtual void fuel(); };</pre>	<pre>void CCar::fuel() {     cout &lt;&lt; "껍데기 클래스" &lt;&lt; endl; }  void Ci8::fuel() {     cout &lt;&lt; "i8연료 : 가솔린" &lt;&lt; endl; }  void Csonata::fuel() {     cout &lt;&lt; "sonata연료 : 전기+가솔린" &lt;&lt; endl; }</pre>	<pre>void runOverrideProc() {     Ci8 i8, * pCi8;     pCi8 = &amp;i8;     pCi8-&gt;fuel();      CCar* pCar;     pCar = pCi8;     pCar-&gt;fuel(); }</pre> <div>결과</div> <div>Microsoft Visual Studio 디버그 콘솔</div> <div>i8연료 : 가솔린 i8연료 : 가솔린</div>



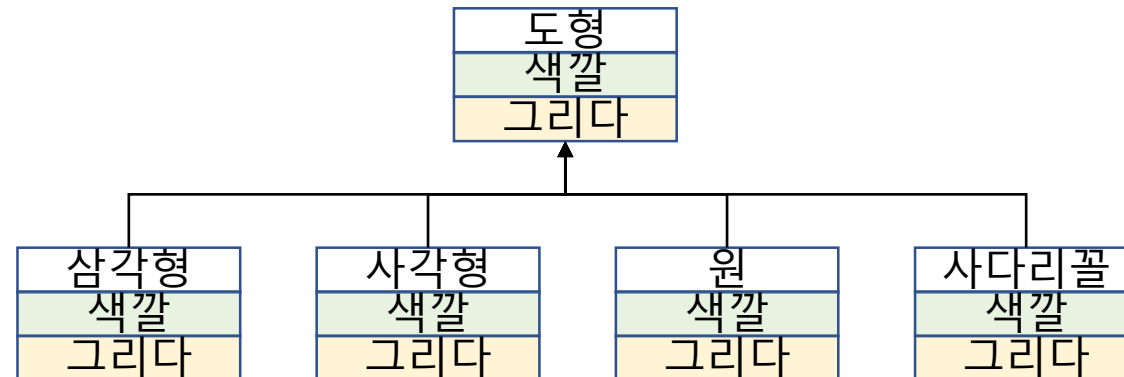
## 10. 다형성

- 오버라이딩(overriding)의 특징
  - a) 함수의 이름과 매개 변수의 타입, 개수, 리턴 타입 모두 일치해야 한다.
  - b) 파생 클래스에서 가상함수를 선언할 때 virtual 키워드를 생략해도 된다.
  - c) 가상함수는 접근지정자 public, private, protected에 제약이 없다.
  - d) 오버라이딩 함수를 호출 할 때 범위 지정자 연산자 ::를 통해서 해당 범위에 속한 함수를 호출할 수도 있다.



## 10. 다형성

지금까지의 설명으로 가상함수를 만드는 목적이 파생 클래스들이 자신의 고유한 기능에 맞게 가상함수를 재정의 하도록 하는 것임을 알았다. 가상함수는 상속받는 파생 클래스에서 구현해야 할 일종의 인터페이스 역할을 하는 것이다. 즉 객체 지향의 **다형성**(polymorphism)을 실현하는 키워드 인 것이다. 아직도 헷갈린다면 다음 클래스 구조를 보자.



각각의 삼각형, 사각형, 원, 사다리꼴은 도형이란 기본 클래스의 파생 클래스이고 이들은 그리다 라는 행위를 한다. 삼각형 클래스는 삼각형을 그려야 하고 사각형 클래스는 사각형을 그리는 것 그것이 다형성이며, 가상함수를 통해 오버라이딩으로 구현할 수 있는 것이다.

- tip) override, final 지시어

1) override 지시어 : 프로그래머의 실수로 가상함수 이름을 잘못 코딩할 때

```
class Shape {
public:
    virtual void draw(); //가상함수
};

class Rect : public Shape {
public:
    void draw();
};
```

다른 함수로 인식

```
class Shape {
public:
    virtual void draw(); //가상함수
};

class Rect : public Shape {
public:
    void draw() override;
};
```

컴파일 에러 :  
draw에 대한 함수  
정의가 없다!!



## 10. 다형성

2) final 지시어 : 파생 클래스에서 오버라이딩을 할 수 없게 하거나, 클래스의 상속 자체를 금지할 수 있다.

Ex) 어떤 클래스가 있는 API를 제공한다 할 때 제공되는 클래스의 상속이나 멤버함수의 override를 금지시키고자 할 때

Ex1)

```
class Shape {  
public:  
    virtual void draw() final; //가상함수  
};  
  
class Rect : public Shape {  
public:  
    void draw() override;  
};
```

1) 가상함수에 final 지시어가 붙어 draw()는 오버라이딩이 불가능 하다.

2) 따라서 Rect 클래스의 draw()는 오버라이딩이 불가하다.

Ex2)

```
class Shape final {  
public:  
    virtual void draw(); //가상함수  
};  
  
class Rect : public Shape {  
};
```

1) 클래스 선언 뒤 final 지시어를 붙이면 해당클래스는 다른 클래스에서 상속이 불가능하다.

2) 따라서 Shape을 상속 받고자 하는 Rect클래스 구문에서 컴파일 에러가 생긴다.



## 10. 다형성

- 가상 소멸자 : 상속 관계의 클래스 사이에서 기본 클래스의 소멸자 앞에 **virtual** 키워드를 붙여 소멸자를 선언하는 것

왜 하는지에 대한 간단한 예시를 보자

소스	결과
<pre>class Base { public:     Base() { cout &lt;&lt; "Base::생성자 호출" &lt;&lt; endl; }     ~Base() { cout &lt;&lt; "Base::소멸자 호출" &lt;&lt; endl; } };  class Derived : public Base { public:     Derived() { cout &lt;&lt; "Derived::생성자 호출" &lt;&lt; endl; }     ~Derived() { cout &lt;&lt; "Derived::소멸자 호출" &lt;&lt; endl; } };  void runConsDesSequenceProc() {     Base* pBase = new Derived(); ①     delete pBase;                ② }</pre>	<p>Microsoft Visual Studio 디버그 콘솔</p> <pre>Base::생성자 호출 Derived::생성자 호출 Base::소멸자 호출</pre> <p>1) 기본 클래스 타입의 포인터로 파생클래스를 가리키는 업 캐스팅 -&gt; 생성자 호출</p> <p>2) 메모리 할당 해제 -&gt; 소멸자 호출</p>

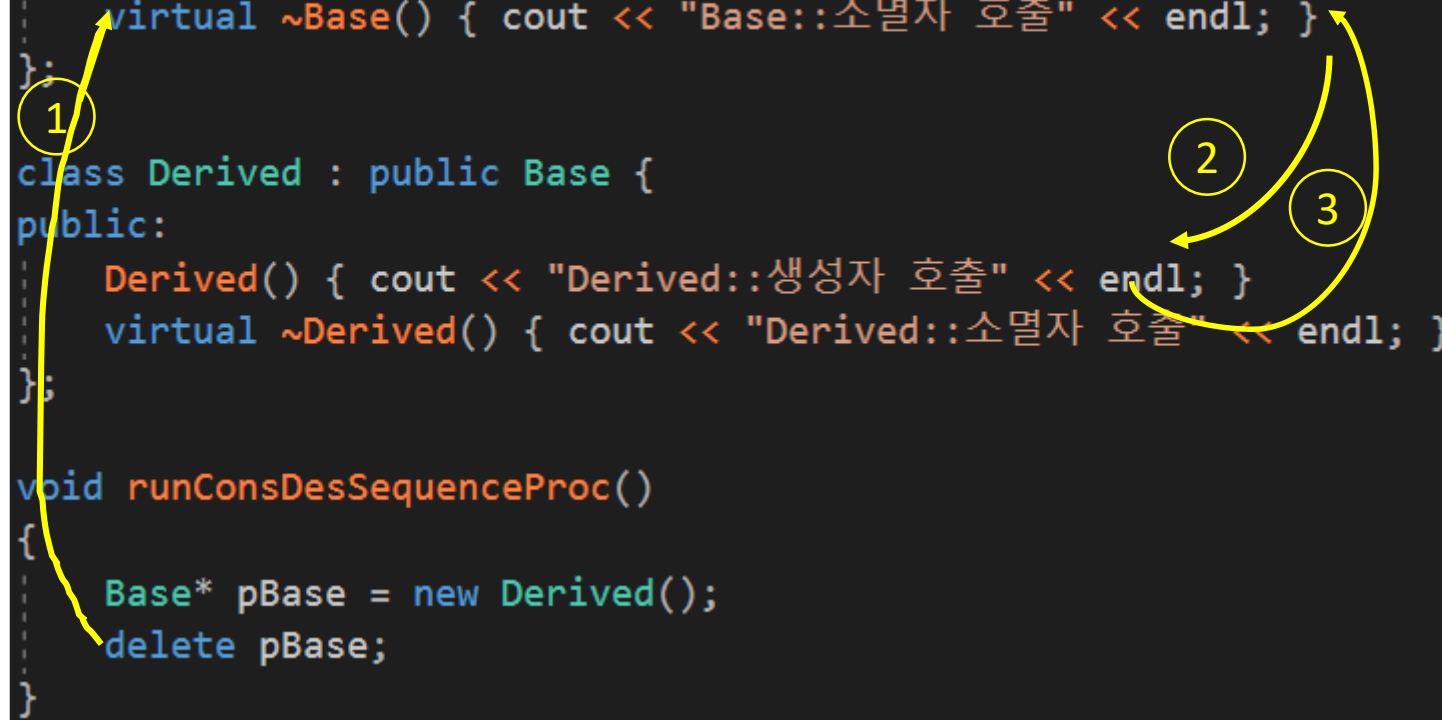
- 결과를 보면 Derived 소멸자 호출을 하지 않는다. 이유는 pBase가 **Base** 타입이므로 정적 바인딩을 통해 **Base**의 소멸자만 호출한다. 만약 Derived의 생성자에서 어떤 포인터 멤버 변수에 동적 메모리 할당을 한 상태라면 메모리 해제가 안되는 결과를 초래한다. 때문에 소멸자 역시 **virtual** 키워드를 사용하여 가상함수로 정의해 준다.



## 10. 다형성

- 가상 소멸자 : 상속 관계의 클래스 사이에서 기본 클래스의 소멸자 앞에 **virtual** 키워드를 붙여 소멸자를 선언하는 것

왜 하는지에 대한 간단한 예시를 보자

소스	결과
<pre>class Base { public:     Base() { cout &lt;&lt; "Base::생성자 호출" &lt;&lt; endl; }     virtual ~Base() { cout &lt;&lt; "Base::소멸자 호출" &lt;&lt; endl; } };  class Derived : public Base { public:     Derived() { cout &lt;&lt; "Derived::생성자 호출" &lt;&lt; endl; }     virtual ~Derived() { cout &lt;&lt; "Derived::소멸자 호출" &lt;&lt; endl; } };  void runConsDesSequenceProc() {     Base* pBase = new Derived();     delete pBase; }</pre> 	<pre>Microsoft Visual Studio 디버그 콘솔 Base::생성자 호출 Derived::생성자 호출 Derived::소멸자 호출 Base::소멸자 호출</pre> <p>1)</p>

. 결론적으로 기본 클래스의 소멸자를 virtual로 선언한다면 어떤 타입의(기본 클래스든 파생 클래스든) 포인터로 소멸하든 기본, 파생클래스의 소멸자를 모두 실행하는 정상적인 과정이 진행된다.



## 10. 다형성

### - 추상 클래스와 순수 가상 함수

기본 클래스에 작성된 가상 함수는 실행할 목적보다는, 파생 클래스에 **대한 인터페이스 역할을** 한다. 이를 바탕으로 동적바인딩이 일어나는데, 실행이 목적이 아니라면 굳이 기본 클래스의 가상함수를 **구현**할 필요가 없다.

- 순수 가상 함수(pure virtual function) : 함수의 구현(정의) 없이 선언만 있는 가상 함수이다. 순수 가상함수의 선언은 **Ex) [virtual] [반환 값] [함수명](매개변수) = 0; 이다.**

### - 추상 클래스(abstract class) :

- a) 최소 하나 이상의 순수 가상 함수를 가지고 있는 클래스
- b) 추상 클래스는 상속을 위한 기본 클래스로 활용하는 것이 목적이며 틀 그 자체로써 구현할 함수의 원형을 나타내는 인터페이스 역할을 한다.
- c) 추상 클래스는 구현되지 않은 멤버 함수가 존재하기에 객체로 인스턴스화 시킬 수 없다.
- d) 추상 클래스를 상속받는 파생 클래스도 순수 가상 함수에 대한 구현이 없다면 객체화 시킬 수 없다.
- e) 추상 클래스를 이용하여 응용프로그램의 **설계**와 **구현**을 분리할 수 있다.



## 10. 다형성

### - 추상클래스 예제 코드

소스	결과
<pre>class CCalculator { <b>계산기 추상 클래스</b> public:     virtual int add(int a, int b) = 0;     virtual int subtract(int a, int b) = 0;     virtual double average(int nArray[], int size) = 0;     virtual ~CCalculator() {}; };  class COMvCalculator : public CCalculator { <b>구현</b> public:     int add(int a, int b) { return a + b; }     int subtract(int a, int b) { return a - b; }     double average(int nArray[], int size) {         double sum = 0;         for (int index = 0; index &lt; size; index++)             sum += nArray[index];         return sum / size;     }     virtual ~COMvCalculator() {}; };  void runCalProc() {     int nArray[] = { 1,2,3,4,5 };     CCalculator* pCal = new COMvCalculator();      cout &lt;&lt; "10 + 20 = " &lt;&lt; pCal-&gt;add(10, 20) &lt;&lt; endl; <b>//30</b>     cout &lt;&lt; "10 - 20 = " &lt;&lt; pCal-&gt;subtract(10, 20) &lt;&lt; endl; <b>//-10</b>     cout &lt;&lt; "(1+2+3+4+5)/5 = " &lt;&lt; pCal-&gt;average(nArray, 5) &lt;&lt; endl;     delete pCal; }</pre>	<pre>Microsoft Visual Studio 디버그 콘솔 10 + 20 = 30 10 - 20 = -10 (1+2+3+4+5)/5 = 3</pre>





## 10. 다형성

### ★오버로딩★

- 함수 오버로딩(function overloading) : 매개 변수 타입이나 개수가 다르지만, 이름이 같은 함수들이 중복 작성 되는 것

- 연산자 오버로딩(operator overloading) : 우리는 + 연산기호를 보면 당연하게도 숫자나 문자열 더하기 연산을 떠올린다. 그러나 피연산자에 따라 서로 다른 연산을 하도록 동일한 연산자를 중복해서 작성하는 것이 가능하다. 같은 + 연산자로 인해 다른 결과를 도출할 수 있다는 점을 보면 이것도 다형성의 일종이다.

7장 구조체에서 구조체 변수 간의 연산이 안된다는 점을 다시 보자. 우리는 연산자 오버로딩을 통해 실현 시킬 수 있다!! 뿐만 아니라 다른 피연산자끼리 연산도 가능하다.

### - 연산자 중복 특징

- a) C++ 언어에 본래 있는 연산자만 중복 가능(+ - \* / % == != && 등)
- b) 연산자 중복이란 피연산자의 타입이 다른 연산을 새로 정의하는 것이다.
- c) 연산자 중복은 함수를 통해 이루어진다. 이를 연산자 함수(operator function)라고 부른다.
- d) 연산자 중복은 반드시 클래스와 관계를 가진다. 이 말은 피연산자는 반드시 객체여야 한다는 소리다.
- e) 연산자 중복으로 피연산자의 개수를 바꿀 수 없다.
- f) 연산자 중복으로 연산의 우선순위를 바꿀 수 없다.
- g) 모든 연산자가 중복이 가능한 것은 아니다('.' '\*' '::' '3항 연산' 불가능)
- h) 연산자 함수는 클래스의 멤버 함수 혹은 전역 함수로 구현하여 클래스 내의 **프렌드 함수**로 선언한다.



## 10. 다형성

Ex1) 이항 연산자(+) 중복 - 소스

```
typedef struct _stPower
{
    int kick;
    int punch;
}stPower;

class CPower {
public:
    CPower() { this->absPower.kick = 0; this->absPower.punch = 0; }
    CPower(int m_kick, int m_punch)
    {
        this->absPower.kick = m_kick;
        this->absPower.punch = m_punch;
    }
    CPower operator+ (CPower op2)
    {
        CPower temp;
        temp.absPower.kick = this->absPower.kick + op2.absPower.kick;
        temp.absPower.punch = this->absPower.punch + op2.absPower.punch;
        return temp;
    }
    void show() { cout << this->absPower.kick << " , " << this->absPower.punch; }
private:
    stPower absPower;
};
```

컴파일러에 의해 변환  
 $c = a + b; \rightarrow c = a. + (b);$

복사 생성자에 대한 내용

1 operator function

호출

```
void runOperatorOverloadingProc()
{
    CPower a(10, 20);
    CPower b(20, 30);

    CPower c = a + b;
    c.show();
}
```

결과

Microsoft Visual Studio 디버그 콘솔

30 , 50

1) 구조체  $c = a + b$ 에서 우측 피연산자  $b$ 는 매개변수로 전달되며 왼쪽 피연산자  $a$ 는 객체 자기 자신이 된다. 반환 타입이  $CPower$  이므로 리턴 되는 객체는  $c$ 에 복사대입이 된다.



## 10. 다형성

Ex2) 비교 연산자(==) 중복 - 소스

```
bool operator== (CPower op2)
{
    if (this->absPower.kick == op2.absPower.kick &&
        this->absPower.punch == op2.absPower.punch)
        return true;
    return false;
}
```

호출

```
void runOperatorOverloadingProc()
{
    CPower a(10, 20);
    CPower b(20, 30);

    if (a == b)
        cout << "a와 b는 같다" << endl;
    else
        cout << "a와 b는 같지 않다" << endl;
}
```

결과

Microsoft Visual Studio 디버그 콘솔

a와 b는 같지 않다

Ex3) 단항 연산자(!) 중복 - 소스

```
bool operator!()
{
    if (!this->absPower.kick && !this->absPower.punch)
        return true;
    return false;
}
```

호출

```
void runOperatorOverloadingProc()
{
    CPower a(0, 0);
    if (!a)
        cout << "a파워가 0이다" << endl;
    else
        cout << "a파워가 0이 아니다" << endl;
}
```

결과

Microsoft Visual Studio 디버그 콘솔

a파워가 0이다



## 10. 다형성

### Ex4) 단항 연산자 소스

```
CPower& operator+= (int op2)
{
    CPower의 주소 값이 반환 값
    this->absPower.kick += op2;
    this->absPower.punch += op2;
    return *this; 객체 자신의 참조(주소)를 리턴!!
}
```

### 호출

```
void runOperatorOverloadingProc()
{
    CPower a(10, 20);
    a += 3;

    a.show();
}
```

### 결과

Microsoft Visual Studio 디버그 콘솔

```
13 , 23
C:\Users\WOM\Documents\카카오톡
```



## 10. 다형성

### - 프렌드 함수(friend function)

클래스 내부에 friend 키워드로 선언된 함수이며 이 함수는 클래스 외부에서 전역으로 구현된다.

- 프렌드 함수의 필요성 : 클래스의 멤버 함수로는 적합하지 않지만 클래스의 private, protected 멤버에 접근해야 하는 경우(대표적으로 연산자 오버로딩)

### - 프렌드 함수의 선언

a) 클래스 외부에 구현된 함수를 프렌드로 선언

헤더	소스
<pre>class Rect; <b>전방 참조</b> bool equals(Rect r, Rect s);  class Rect { public:     Rect(int width, int height) { this-&gt;width = width; this-&gt;height = height; }     friend bool equals(Rect r, Rect s); private:     int width;     int height; };</pre> <div data-bbox="478 1209 1795 1332">Rect 클래스 내에 equals 프렌드 함수가 있기에 private인 width와 height에 접근할 수 있는 것이다.</div>	<pre>bool equals(Rect r, Rect s) {     if (r.width == s.width &amp;&amp;         r.height == s.height)         return true;     return false; }</pre>
호출	결과
<pre>int main(void) {     Rect rect1(20, 30);     Rect rect2(20, 30);     if (equals(rect1, rect2))         cout &lt;&lt; "두 Rect는 같다" &lt;&lt; endl;     else         cout &lt;&lt; "두 Rect는 다르다" &lt;&lt; endl; }</pre>	<div data-bbox="1069 1481 1725 1641">Microsoft Visual Studio 디버그 콘솔 두 Rect는 같다</div>



## 10. 다형성

b) 다른 클래스의 멤버 함수를 프렌드로 선언

헤더

```
class Rect;
class RectManager {
public:
    bool equals(Rect r, Rect s);
};
class Rect {
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend bool RectManager::equals(Rect r, Rect s);
private:
    int width;
    int height;
};
```

소스

```
bool RectManager::equals(Rect r, Rect s)
{
    if (r.width == s.width &&
        r.height == s.height)
        return true;
    return false;
}
```

호출

```
int main(void)
{
    Rect rect1(20, 30);
    Rect rect2(20, 30);
    RectManager rectMan;

    if (rectMan.equals(rect1, rect2))
        cout << "두 rect는 같다" << endl;
    return 0;
}
```

결과

Microsoft Visual Studio 디버그 콘솔  
두 rect는 같다



## 10. 다형성

c) 다른 클래스의 모든 멤버 함수를 프렌드로 선언

헤더

```
class Rect;
class RectManager {
public:
    bool equals(Rect r, Rect s);
};
class Rect {
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend RectManager; 클래스 자체를 friend로 선언해버린다
private:
    int width;
    int height;
};
```



## 10. 다형성

c) 다른 클래스의 모든 멤버 함수를 프렌드로 선언

헤더

```
class Rect;
class RectManager {
public:
    bool equals(Rect r, Rect s);
};
class Rect {
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend RectManager; 클래스 자체를 friend로 선언해버린다
private:
    int width;
    int height;
};
```





## 10. 다형성

★전방참조(forward reference), 후방참조(backward reference)★

C++에서는 변수나 함수, 클래스 이름을 먼저 선언한 후에 참조(사용) 하는게 원칙이다. (backward reference)

```
class Rect;
bool equals(Rect r, Rect s);

class Rect {
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend bool equals(Rect r, Rect s);
private:
    int width;
    int height;
};
```

Q) 이 구문이 없다면?

A) equals() 선언에서 컴파일러 : Rect는 뒤에 선언 및 구현되어 있는데 equals에서 미리 참조하기 때문이다. 컴파일러 입장에서 아직 선언되지 않은 이름을 참조(forward reference)하므로 에러가 발생한다.

class Rect를 미리 선언하여 전방참조(forward reference) 문제를 해결하고 이를 forward declaration이라고 한다.



## 11. 클래스 고급

- 정적 멤버(static member) : 여러 클래스들이 공유하는 멤버

친구들과 모임통장을 만들었다고 치면 모임통장 하나에 저축을 할 것이다. 그 계좌가 바로 바로 static 멤버 변수이다.

- 정적 멤버 특징 :

- a) static 멤버 함수는 non-static 멤버 변수에 접근 불가능 : 프로그램이 시작함과 동시에 static 멤버는 생성되었지만 non-static은 객체의 인스턴스화가 진행되지 않았기 때문에 메모리에서 찾을 수 없다.
- b) non-static 멤버 함수는 static 멤버 변수에 가능
- c) 생명 주기 : 프로그램이 시작할 때 생성, 종료할 때 소멸
- d) 사용 범위 : 선언되는 공간 안 전체
- e) static 멤버가 선언되어 있다면 반드시 전역에서 초기화 진행

선언 & 구현	호출	결과
<pre>class allAccount { public:     allAccount(string name) { this-&gt;name = name; }     static void saveMoney(int m_money)     {         currentMoney += m_money;     }     static void showMoney()     {         cout &lt;&lt; "현재 금액 : " &lt;&lt; currentMoney &lt;&lt; endl;     } private:     string name;     static int currentMoney; };  int allAccount::currentMoney = 0;</pre>	<pre>int main(void) {     CAllAccount person1("손형욱");     CAllAccount person2("김기철");      person1.saveMoney(100);     person2.saveMoney(300);      person1.showMoney();     person2.showMoney();      return 0; }</pre>	<div>Microsoft Visual Studio 디버그 콘솔</div> <div>현재 금액 : 400 현재 금액 : 400</div>



## 11. 클래스 고급

### - 상수 멤버 함수 & 상수 객체

기본 데이터 타입의 변수에 `const` 키워드를 사용하여 상수 변수를 정의할 수 있는 것 처럼, 클래스의 멤버 함수와 객체에도 `const` 키워드를 사용하여 상수 멤버 함수(const member function)과, 상수 객체(constant object)를 정의할 수 있다.

- 상수 멤버 함수(const member function) : 멤버 함수에서 객체의 멤버 변수를 변경할 수 없게 하는 것, 즉 읽기 전용 함수가 되는 것이다. 단 생성자와 소멸자에는 `const` 키워드를 붙일 수 없다. 항상 데이터의 초기화 혹은 소멸이 일어나야 하기 때문이다. 또한 상수화된 함수는 상수화된 함수만 호출할 수 있다. 애초에 데이터 변경을 배제해야 하기 때문이다.

선언부	정의부
Ex) <code>class CCircle {     ...     int getData() const;     void setData(); }</code>	Ex) <code>int CCircle::getData() const {     ...(명령문) }</code>

- 상수 객체(constant object) : 객체를 생성할 때 `const`를 붙이면 생성자와 소멸자에 의한 데이터 초기화 & 소멸 이외의 데이터 변경은 할 수 없다.

선언 및 호출
Ex) <code>const CCircle c1();     c1.setData(); // 에러</code>

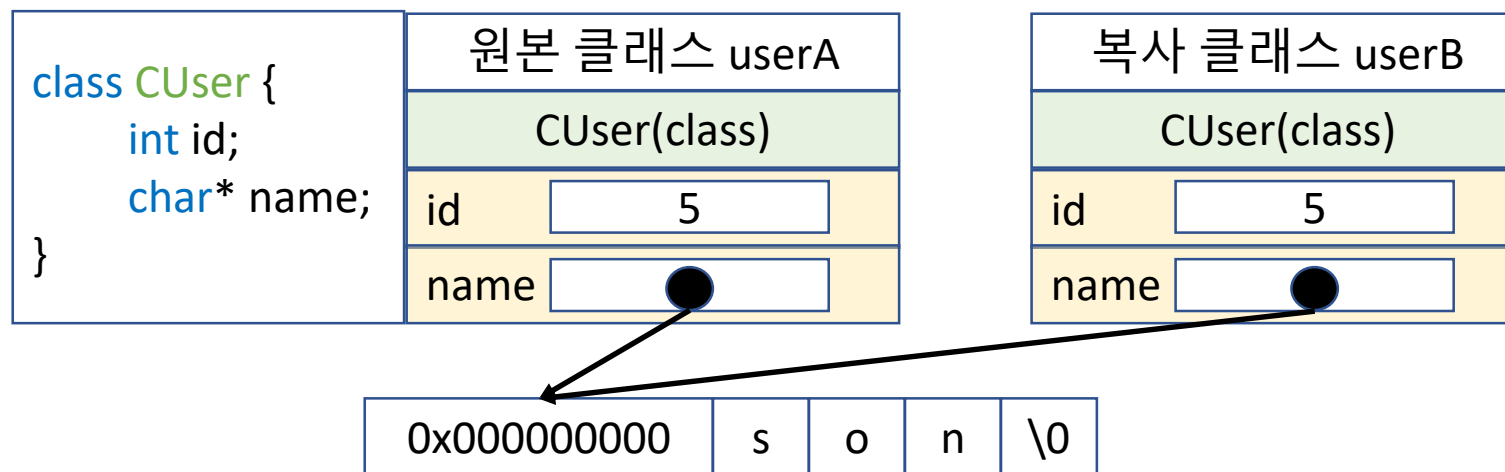


## 11. 클래스 고급

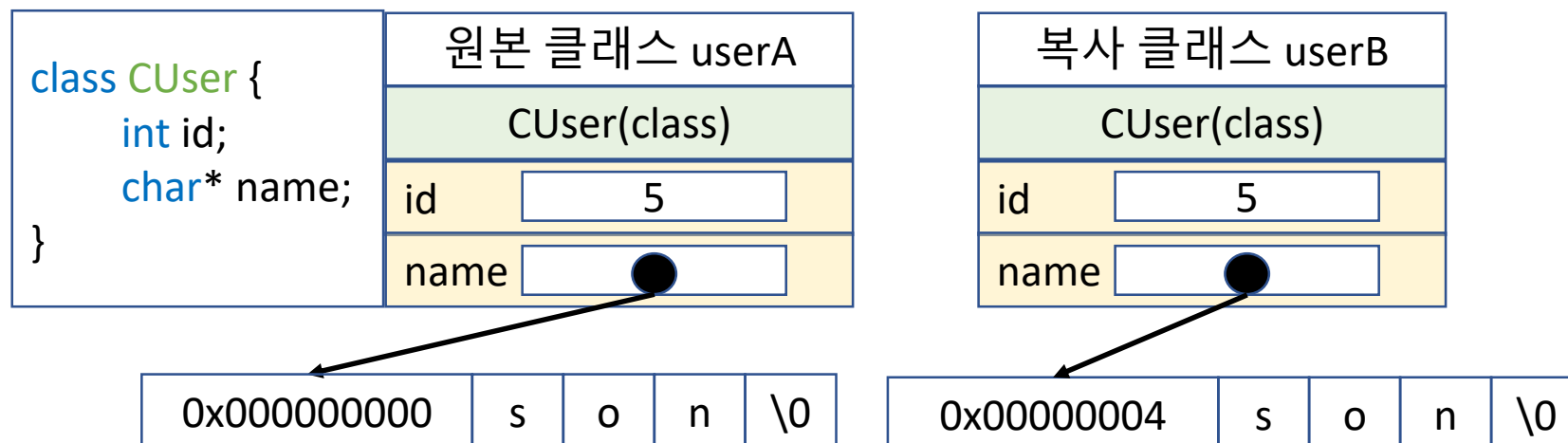
- 복사 생성자(copy constructor) : 객체를 복사할 때 호출되는 생성자
- **객체의 복사** : 객체가 생성될 때 원본 객체를 복사하여 생성되는 경우로 '='을 이용한 대입(치환)연산이 아니다.  
Ex) Circle cCopy(cOrigin) : cCopy는 cOrigin의 멤버를 그대로 복사한다.

### - 얕은 복사 & 깊은 복사

- a) 얕은 복사 : origin 객체 멤버 변수 중 동적 할당된 메모리가 있다면 객체 복사시에 copy본도 같은 메모리를 가리킨다.



- b) 깊은 복사 : 객체 복사시 origin 객체 멤버, copy 객체 멤버 모두 각자의 동적 할당 메모리를 가진다.



## 11. 클래스 고급

- 얽은 복사로 인한 메모리 공유문제는 발견하기도 어렵고 발견했다 치더라도 시간이 많이 소비되기 때문에 가능한 얽은 복사가 발생하지 않도록 해야 한다.
- 얽은 복사는 복사생성자를 직접 구현하지 않았을 때 컴파일러가 묵시적으로 default 복사생성자를 삽입함으로써 일어나기 때문에 반드시 복사생성자를 정의 해야 문제가 발생하지 않는다.

```
class CUser {
public:
    CUser(int m_id, const char* m_name) 생성자
    {
        this->id = m_id;
        this->name = new char[strlen(m_name) + 1]; ①
        strcpy(this->name, m_name); ②
        this->checkCopy = "원본";
        cout << this->checkCopy + "생성자 실행" << endl;
    }
    CUser(const CUser& m_copy) 복사 생성자 ③
    {
        this->id = m_copy.id;
        this->name = new char[strlen(m_copy.name) + 1];
        strcpy(this->name, m_copy.name);
        this->checkCopy = "복사";
        cout << this->checkCopy + "생성자 실행" << endl;
    }
    ~CUser() {
        cout << this->checkCopy + "소멸자 실행" << endl;
        delete[] this->name;
    }
private:
    int id;
    char* name;
    string checkCopy; ④
};
```

1) name에 매개변수 m\_name 사이즈만큼 메모리 할당

2) name에 m\_name 값 복사

3) 매개변수를 자세히 보면 const CUser& m\_copy 이다. call by reference를 통해 원본 객체의 주소를 그대로 공유하지만 const를 붙임으로써 읽기전용이 되는 것이다.

4) 생성자와 소멸자 호출에 있어서 원본과 복사본을 구별하기 위함이다.

- getter함수는 생략하였다.



## 11. 클래스 고급

### 호출

```
int main(void)
{
    CUser userA(10, "손형욱"); 원본 생성자 호출
    CUser userB(userA); 복사 생성자 호출

    return 0; 복사 소멸자, 원본 소멸자 차례로 호출
}
```

### 결과

Microsoft Visual Studio 디버그 콘솔

```
원본 생성자 실행
복사 생성자 실행
복사 소멸자 실행
원본 소멸자 실행
```



## 11. 클래스 고급

- 변환 생성자(conversion constructor), 변환 연산자(conversion operator)

a) 변환 생성자 :

- 기본데이터 타입을 사용자 정의 데이터 타입으로 타입 변환
- 다른 사용자 정의 데이터 타입을 사용자 정의 데이터 타입으로 타입 변환
- 변환 생성자는 기본 데이터 타입 매개변수를 하나만 받는다.

b) 변환 연산자 :

- 사용자 정의 데이터 타입을 기본 데이터 타입으로 타입 변환
- 사용자 정의 데이터 타입을 다른 사용자 정의 데이터 타입으로 타입 변환

변환 생성자 & 변환 연산자 - 소스

```
class CDistance {  
private:  
    int kilometer, meter;  
public:  
    CDistance() : kilometer(0), meter(0) {}  
    explicit CDistance(int newDist) { ① 변환 생성자  
        kilometer = newDist / 1000;  
        meter = newDist % 1000;  
    }  
    explicit operator int() { ② 변환 연산자  
        return kilometer * 1000 + meter;  
    }  
    void printDistance(const CDistance& d) {  
        cout << d.kilometer << " , " << d.meter << endl;  
        cout << "this " << kilometer << " , " << meter << endl;  
    }  
};
```

호출

```
void runConversionOperator()  
{  
    CDistance d = static_cast<CDistance>(3030);  
    d.printDistance(d);  
    int a = static_cast<int>(d);  
    cout << a << endl;  
}
```

호출

Microsoft Visual Studio 디버그 콘솔

```
3 , 30  
this 3 , 30  
3030
```



## 11. 클래스 고급

- 네임스페이스(namespace) :

```
#include<iostream>
```

```
using namespace std;
```

- using namespace std; : std(standard)라는 표준라이브러리의 네임스페이스를 사용하겠다 라는 뜻

- namespace : 이름이 소속되어 있는 공간이란 뜻, 식별자의 이름을 특정 영역 안에 그룹화 시켜 중복으로 인한 충돌이 발생하지 않도록 구분하는데 쓰임

Ex) 대규모 프로젝트시 유용하게 쓰인다.

```
Ex) namespace A {  
    class CCircle { ... };  
}
```

```
namespace B {  
    class CCircle { ... };  
}
```

- 두 개의 같은 식별자 CCircle로 정의된 class가 존재하는데 하나는 A영역, 나머지 하나는 B영역이다. 우리는 이 클래스를 사용할 때 영역을 지정하여 사용해야한다.

Ex) A::CCircle [클래스이름];





## 12. 템플릿 & 표준라이브러리

- 템플릿(template)이란 일반화(generic)과도 같은 말이다. 함수의 오버로딩을 구현하다 보면 약점에 부딪힐 수 있는데, 다음 예시를 보자

```
void swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

int형 자료형을 swap

```
void swap(double& a, double& b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

double형 자료형을 swap



```
template<class T>
void swap(T& a, T& b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

- 두 함수는 매개변수만 다른 점을 제외하고 동일한 코드이다. 그러나 오버로딩을 계속 진행하면 코드의 길이가 늘어나고, 프로그래머의 실수가 생길 수도 있기 때문에 이를 일반화하는 방식이 필요하다. 즉 중복함수에 대한 일반화를 제공하는게 template이다. template 키워드로 선언된 함수는 template function 또는 generic function이라고 부른다.

- 템플릿 선언

Ex) template <class T>

Ex) template <typename T>

template, class, typename : 키워드, T : generic type(기본으로 제공되는 데이터형이 아니다)



## 12. 템플릿 & 표준라이브러리

- template 스택 구현 : int면 int, char면 char 어떤 데이터 타입이든 제네릭한 Stack이 된다.

### 클래스 선언 및 정의

```
template<class T>
class myStack {
private:
    int tos;    //TopOfStack
    T data[100];
public:
    myStack();
    void push(T element);
    T pop();
};

template<class T>
myStack<T>::myStack()
{
    tos = -1;
}

template<class T>
void myStack<T>::push(T element)
{
    if (tos == 99)
    {
        cout << "stack is full ! ";
        return;
    }
    data[++tos] = element;
}

template<class T>
T myStack<T>::pop()
{
    if (tos == -1)
    {
        cout << "stack is empty ! ";
        return 0;
    }
    return data[tos--];
}
```

클래스 선언

멤버함수 정의

### 호출

```
void runTemplateStack()
{
    myStack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);

    cout << s.pop() << " ";
    cout << s.pop() << " ";
    cout << s.pop() << " ";
}
```

### 결과

Microsoft Visual Studio 디버그

30 20 10



## 12. 템플릿 & 표준라이브러리

### STL(Standard Template Library)

- 컨테이너, 반복자, 알고리즘으로 이루어져 있는 라이브러리
- 광범위하게 두루 이용되며 재사용 및 이식성이 좋다. 또한 다양한 자료구조, 알고리즘을 최적화하여 사용하기 편리하고, 동작속도도 빠르다. 검증된 코드로서 빠르고 효율적으로 코드작성이 가능하다.

#### 1) 컨테이너(container)

- 데이터를 저장하고 검색하기 위해 담아두는 자료 구조를 구현한 클래스

a) 순차 컨테이너(sequence container) : 연속적인 메모리 공간에 데이터를 순차적으로 저장, 임의의 위치에 삽입, 삭제 가능

Ex) vector(가변배열), list(linked list), deque

b) 어댑터 컨테이너(adaptor container) : 데이터를 미리 정해진 방식에 따라 관리

Ex) stack(FILO), queue(FIFO) // FIFO = First In First Out, FILO = First In Last Out

c) 연관 컨테이너() : 일정 규칙에 따라 자료를 조직화(정렬, 해시)하여 저장

Ex) map, set 등

#### 2) 반복자(iterator)

- 컨테이너 원소에 접근할 수 있는 포인터 객체이다. 컨테이너에 직접적으로 접근을 할 수 있지만 iterator는 포인터로 접근을 하기 때문에 동작시간을 단축시킬 수 있다.

#### 3) 알고리즘(algorithm)

- 컨테이너 원소에 대한 복사(copy), 검색(find, search), 삭제(remove), 정렬(sort) 등의 기능을 구현한 템플릿 함수, 통칭하여 알고리즘이라고 불림



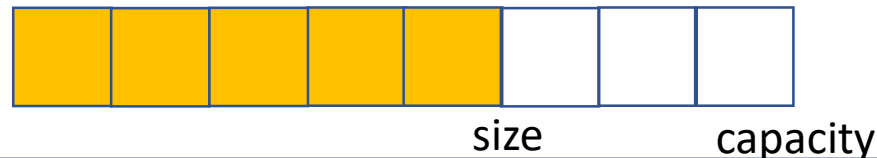
## 12. 템플릿 & 표준라이브러리

STL컨테이너(Vector); #include<vector>

- Vector는 순차 컨테이너로서 멤버함수 push\_back(data), pop(), 을 이용하여 데이터를 추가, 삭제 할 수 있다.
- Vector를 사용하기 앞서 Capacity와 Size의 의미를 알아야 한다.

a) Capacity : 할당된 메모리의 크기

b) Size : 메모리에 저장된 요소의 개수



```
vector<int> a;  
a.push_back(1);  
a.push_back(2);  
a.push_back(3);  
a.push_back(4);  
a.push_back(5);  
a.push_back(6);  
a.push_back(7);
```

이름	값
a	{ size=1 }
[capacity]	1
[allocator]	allocator
[0]	1
[Raw 뷰]	{_Mypair=allocator }

a.push\_back(1)

이름	값
a	{ size=4 }
[capacity]	4
[allocator]	allocator
[0]	1
[1]	2
[2]	3
[3]	4

a.push\_back(4)

이름	값
a	{ size=5 }
[capacity]	6
[allocator]	allocator
[0]	1
[1]	2
[2]	3
[3]	4

a.push\_back(5)

위 실행 결과를 보자.

- push\_back으로 요소가 늘어날 때마다 capacity가 늘어나고 size도 늘어난다. 이는 새로운 요소를 추가&삭제 할 때마다 메모리를 재할당 및 기존 메모리의 모든 요소를 복사하기 때문이다. 그런데 만약 vector의 메모리가 매우 크거나 빈번한 추가 삭제를 한다면? 오버헤드를 발생시키기 때문에 vector의 특징인 Size, Capacity를 이해할 필요가 있다.

=> 때문에 오버헤드가 예상된다면, vector의 capacity를 미리 할당 시키도록 한다. size가 capacity가 넘어설 때, capacity를 재 할당 한다. push\_back을 지양하고, 아래의 방법을 지향하자.

① //멤버함수를 이용한 capacity 할당

```
vector<int> a;  
a.reserve(10);
```

② //생성자를 통한 capacity 할당

```
vector<int> b(100, 0);
```



## 12. 템플릿 & 표준라이브러리

- 위 슬라이드의 1,2번 방식과 더불어 vector는 일반 배열의 인덱스 접근 방식으로도 사용 가능하다. 단 vector의 size보다 큰 값에 접근을 시도한다면 memory range error를 맞닥뜨린다.

vector 초기화 및 값 대입

```
vector<int> b(100, 0);

for (int i = 0; i < b.size(); i++)
{
    b[i] = i;
    cout << b[i] << endl;
}


b[101] = 1;
```

결과

Microsoft Visual Studio 디버그 콘솔

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

Microsoft Visual C++ Runtime Library

 Debug Assertion Failed!

Program:  
C:\Users\OMV\source\repos\C++ Essence\Debug\C++ Essence.exe  
File: C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.21.27702\include\vector  
Line: 1469

Expression: vector subscript out of range

For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)

중단(A)

다시 시도(R)

무시(I)



## 12. 템플릿 & 표준라이브러리

### - vector의 생성자

- a) `vector<int> v` : 비어있는 vector를 생성
- b) `vector<int> v(5)` : 기본 값(0)으로 초기화 된 5개의 요소를 가지는 vector를 생성
- c) `vector<int> v(5, 2)` : 2로 초기화 된 5개의 요소를 가지는 vector를 생성
- d) `vector<int> v2(v1)` : v1을 복사하는 복사 생성자

### - vector의 멤버 함수(자주 사용하는)

- a) `v.assign(5, 2)` : 5개의 원소 할당 및 2로 초기화
- b) `v.at(index)` : index번째 요소 참조, 앞서 인덱스로 접근하는 방법에서는 range error가 날 수 있지만 at은 범위를 점검하므로 안전한 방법 단 조금 느리다.
- c) `v.front()` : 첫번째 요소 참조
- d) `v.back()` : 마지막 요소 참조
- e) `v.clear()` : 모든 요소 제거, 단 할당된 메모리는 그대로 남아 capacity 유지, size는 0
- f) `v.push_back(data)` : 마지막 요소 뒤에 7 삽입
- g) `v.pop()` : 마지막 요소 제거
- h) `v.size()` : 현재 size를 return
- i) `v.begin()` : 첫 번째 요소 iterator return
- j) `v.end()` : 마지막 요소 다음 iterator return
- k) `v.reserve(5)` : 메모리를 할당하여 capacity = 5가 된다
- l) `v.resize(10)` : size를 10으로 변경 기존 size나 capacity보다 커질 경우 기존 data이외 늘어난 만큼 0으로 초기화
- m) `v.data()` : vector의 첫번째 요소 주소를 return 한다.

이 외에도 많은 멤버 함수가 있으니 사용할 때 찾아보길 바란다



## 12. 템플릿 & 표준라이브러리

STL컨테이너(map); #include<map>

- map은 연관 컨테이너로서 key와 value를 쌍(pair)으로 묶어 하나의 요소로 저장한다.
- map은 중복된 key를 허용하지 않고(unique key), key를 정렬하여 요소를 저장한다. 이 때 저장공간의 필요에 따른 동적할당을 한다.
- map의 요소들은 노드 기반으로 이루어져 있으며 **균형 이진 트리 구조**이다.

map은 언제 사용할까?

- 정렬해야 한다.
- 많은 자료를 저장하고, 검색이 빨라야 한다
- 빈번하게 삽입, 삭제하지 않는다.

```
map<string, int> m;  
m.insert(make_pair("손형욱", 1));  
m.insert(make_pair("김기철", 2));  
m.insert(make_pair("오경탁", 3));  
m.insert(make_pair("임희영", 4));  
m.insert(make_pair("강경진", 5));  
m.insert(make_pair("김용찬", 6));
```

- 가나다 순으로 정렬이 되어 할당된 모습이다.

m	{ size=6 }
[compara...	less
[allocator]	allocator
["강경진"]	5
["김기철"]	2
["김용찬"]	6
["손형욱"]	1
["오경탁"]	3
["임희영"]	4

- 만약 Key값이 중복된 요소를 추가한다면 어떻게 될까?

```
map<string, int> m;  
m.insert(make_pair("손형욱", 1));  
m.insert(make_pair("김기철", 2));  
m.insert(make_pair("오경탁", 3));  
m.insert(make_pair("임희영", 4));  
m.insert(make_pair("강경진", 5));  
m.insert(make_pair("김용찬", 6));  
  
m.insert(make_pair("김용찬", 10));  
if (m.insert(make_pair("김용찬", 10)).second == false)  
    cout << "추가실패" << endl;
```

- 내용은 바뀌지 않고,  
콘솔창에는 “추가 실패”가  
입력된다.

m	{ size=6 }
[compara...	less
[allocator]	allocator
["강경진"]	5
["김기철"]	2
["김용찬"]	6
["손형욱"]	1
["오경탁"]	3
["임희영"]	4
[Raw 뷰]	{...}

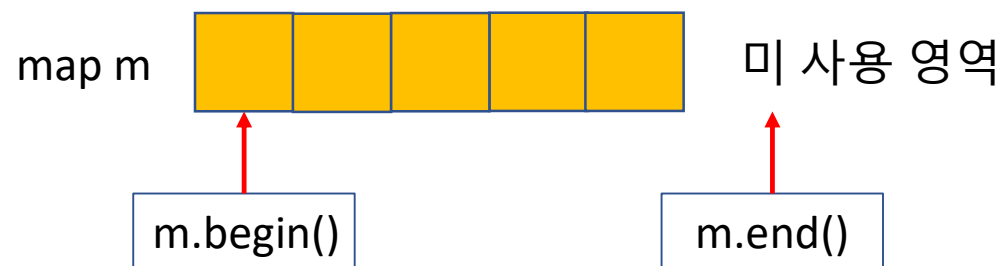


## 12. 템플릿 & 표준라이브러리

단 `m["김용찬"] = 10;`, operator를 이용한 이 명령문은 기존의 Key값 “김용찬”의 Value 값을 10으로 변경해준다. 중복된 key가 아니라면 추가도 가능하다.

- map 생성자
  - a) map<key 자료형, value 자료형> 변수명
  - b) map<key 자료형, value 자료형> 변수명(less or greater) : 오름차순 혹은 내림차순 선택
  - c) map<key 자료형, value 자료형> 변수명(변수명2) : 복사 생성자
- map 멤버 함수(자주 사용하는)
  - a) m.begin() : 첫 번째 요소 iterator return
  - b) m.end() : 마지막 요소 다음 iterator return
  - c) m.clear() : map의 모든 요소를 삭제
  - d) m.erase(key), m.erase(iterator, iterator): key를 포함한 요소 삭제, 범위 내 삭제
  - e) m.find(key) : key를 가진 요소의 iterator return
  - f) m.empty() : map이 비었으면 true, size >= 1 false return
  - g) m.size() : 현재 map의 요소 개수 return

- m.begin()과 m.end()의 이해를 돕기 위한 그림





## 12. 템플릿 & 표준라이브러리

### STL반복자(iterator)

- iterator는 STL 컨테이너의 요소에 접근(참조)할 수 있는 일종의 포인터이다.
- 컨테이너와 알고리즘을 묶어주는 인터페이스 역할도 한다.
- STL이 지원하는 어떤 컨테이너 공통적으로 쉽게 관리 할 수 있다.

### 반복자의 종류 및 기능

[http://tcpschool.com/cpp/cpp\\_iterator\\_category](http://tcpschool.com/cpp/cpp_iterator_category)

#### - vector와 iterator

```
vector<int>::iterator it; ①
for(② it = v.begin(); it != v.end(); ++it)
    cout << ③ *it << endl;
```

Microsoft Visual Studio

```
1
2
3
4
5
```

1. vector에 대한 iterator 선언
2. iterator 초기화
3. iterator가 가리키는 요소의 값 접근

#### - map과 iterator

```
② map<string, int>::iterator it; ①
for(② it = m.begin(); it != m.end(); ++it)
{
    cout << ③ "[" << it->first << "] = ";
    cout << ④ it->second << endl;
}
```

Microsoft Visual Studio

```
[강경지] = 5
[김기철] = 2
[김용찬] = 6
[손형욱] = 1
[오영택] = 3
[임희영] = 4
```

1. map에 대한 iterator 선언
2. iterator 초기화
3. key값 접근
4. value값 접근

각각의 컨테이너 요소에 순차적으로 접근하면서 요소의 값을 출력하는 코드이다.



## 12. 템플릿 & 표준라이브러리

STL알고리즘(algorithm); #include<algorithm>

- algorithm은 말그대로 컨테이너의 요소를 가지고 [어떤 작업을 수행] 하는 것이다.
- algorithm의 함수 원형은 대부분이 void algorithm( iter begin, iter end ) 이며 algorithm( iter begin, iter end, Pred pred) 형태이다. 이 때 pred는 알고리즘을 수행하기 위한 특정 조건이다.  
Ex) sort(정렬) 알고리즘의 경우 오름차순 혹은 내림차순 조건 같은

알고리즘 종류(자주 사용되는)

- a) sort, stable\_sort, partial\_sort : 컨테이너 속 요소들을 정렬
- b) binary\_search : 이미 정렬 되어 있는 컨테이너 에서 특정 데이터가 지정한 구간에 있는지 조사
- c) merge : 두 개의 정렬된 구간을 합침

★ 내부 로직을 몰라도 어떤 동작을 하는지, 제약사항은 무엇인지 반드시 알고 사용하자 ★

★ Microsoft docs를 살펴보면 여러 알고리즘 함수의 원형과 설명을 볼 수 있다. ★

sort 예시

```
std::srand(NULL);  
vector<int> v(100, 0);  
for (int i = 0; i < v.size(); i++)  
{  
    ① v[i] = rand();  
}  
② sort(v.begin(), v.end(), greater<int>());
```

- 1) 난수 생성
- 2) 내림차순으로 정렬

v { size=100 }	
[capacity]	100
[allocator]	allocator
[0]	24167
[1]	12236
[2]	31249
[3]	23636
[4]	15619
[5]	25727
[6]	12700
[7]	24323
[8]	14922
[9]	22774
[10]	6811

1) 난수생성 결과

v { size=100 }	
[capacity]	100
[allocator]	allocator
[0]	32285
[1]	32278
[2]	31891
[3]	31597
[4]	31111
[5]	30612
[6]	29533
[7]	29404
[8]	28937
[9]	28881
[10]	28871

2) 정렬 결과

★ Algorithm에서 제공하는 sort는 Quick Sort를 상황에 따라 merge, heap sort 등 혼합되어 쓰인다. ★



## 12. 템플릿 & 표준라이브러리

정렬을 공부하기 전 자료구조(자료를 저장 및 관리하는 방식)와 알고리즘(자료를 처리하는 방식)에 대한 이해를 바탕으로, 공간복잡도와 시간복잡도 개념에 입각하여 데이터 처리의 효율성에 대한 이해를 하면 좋다.

### 1. 공간복잡도란(space complexity)

어떤 알고리즘 내에서 사용되는 자원 할당량 정도를 말하며, 고정 할당, 가변 할당을 합쳐 계산한다. 사실 시대가 지남에 따라 하드웨어의 성능이 비약적으로 발전했고, 공간 복잡도에 대한 엄격함은 줄어 들었다. 그러나 부족한 자원을 매우 효율적으로 사용해야하는 임베디드의 경우는 조금 다를 수도 있겠다.

### 2. 시간복잡도란(time complexity)

어떤 알고리즘을 이루고 있는 연산이 몇 회 실행되는지 나타내는 지표이다. 연산에는 사칙(산술)연산, 변수에 값을 대입하는 대입연산, 비교연산, 이동연산 등 프로그램 내에서 코드레벨로 이루어지는 모든 행위가 연산에 포함된다고 생각하자. 연산의 횟수는 결국 프로그램 동작시간에 직결되는 것이다.

조금 더 쉬운 이해를 위해 간단한 코드를 보자.

```
int num = 0;
for( int i = 0; i < n; i++)
    num = i;
```

연산횟수

$i = 0$ (대입),  $i < n$ (대입),  $i++$ (증감),  $num = i$ (대입)  $\rightarrow 4 * n$  번이다.

```
int num = 0;
for(int i=0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        num = i + j;
    }
}
```

연산 횟수는 대략  $n^2$ , 앞선 예제보다 비약적으로 상승했다.



## 12. 템플릿 & 표준라이브러리

위 예시에서 for문이 하나 더 중첩되어 연산횟수가 비약적으로 늘어났다. 중요한건 정확한 연산 횟수보다는 연산의 증가율이다.  $n$ 이 아무리 커져도  $2n+1$  이나  $3n+5$ 는 CPU입장에서 큰 차이가 없다는 뜻이다.  $2n+1$ 과  $2n^2 + 1$ 을 놓고 따졌을 때는 이야기가 달라진다.

이러한 시간 복잡도는 Big-O (상한선 - 최악) 이하 빅 오, Big-Ω (하한선 - 최선) 이하 빅 오메가, Big-Θ(상, 하한선 사이) 이하 빅 세타로 표기할 수 있는데 보통 Big-O 표기법을 많이 사용한다.

Big-O notation의 수학적 정의

- 두 개의 함수  $f(n)$ 과  $g(n)$ 이 주어질 경우 모든  $n > n_o$ 에 대하여  $|f(n)| \leq c |g(n)|$ 을 만족하는 2개의 상수  $c$ 와  $n_o$ 가 존재하면  $f(n) = O(g(n))$ 이다. (Big-Θ는  $|f(n)| \geq c |g(n)|$ 이다)

즉  $f(n) = 2n + 1$ 은 Big-O notation 표기법으로  $O(n)$ ,  $f(n) = 2n^2 + 1$ 은  $O(n^2)$ 이다. 쉽게 생각하자면, 시간에 영향을 끼치는 요소 중 가장 큰 요소만 남기고 제하는 형식인데, “아무리 걸려도 이 정도 시간 안에는 끝난다”라고 인식하면 될 것 같다.



## 13. 예외 처리

### 예외처리(Exception Handling)

- 앞서 말한(5장) RAII(Resource Acquisition Is Initialization)에서 Exception을 언급했다. Exception은 프로그램 실행 중 일어나는 비정상적인 상황이다. 예를 들어 나누기 연산에서 제수가 0인 경우, vector의 범위를 벗어난 경우 등등.. 그리고 이를 처리하는게 예외 처리이다. try catch문이라고 들어봤을 것이다.

- try ~ throw ~ catch 예시

```
try
{
    ...
    if(예외 발견)
    {
        throw 예외 값;
    }
}
catch(처리할 예외 파라미터)
{
    예외 처리문;
}
...
```

```
int num = 0;
try
{
    if(num == 0)
    {
        throw 3;
    }
}
catch (int exception)
{
    cout << exception << endl;
}

Microsoft Visual Studio 디버그 콘솔
3
```

1. try문에서 던지는 3이라는 예외 값은 catch문에서 exception 매개변수로 받아 처리한다.
2. 인자로 받는 exception 값은 int형 말고도 여러 자료형(타입)이 가능하다.
3. throw로 던지는 자료형과 catch에서 인자로 받는 자료형은 같아야 한다. catch가 구현되어 있지 않으면 abort error 발생, throw에서 int형을 던지고 int를 받는 catch문이 1개 초과 시 컴파일 에러 발생, 즉 1대1 매칭이다.
4. try문에서 예외를 발견하지 않으면 catch로 넘어가지 않고 다음 명령을 실행한다.
5. throw문장은 반드시 try 블록 안에 있어야 한다.



## 13. 예외 처리

- 함수를 포함하는 try ~ catch문

```
int CalExp(int value, int exp)
{
    if (exp < 0)
        ② throw "exp가 음수";

    int result = 1;
    for (int tc = 0; tc < exp; tc++)
        result *= value;

    return result;
}

int main(void)
{
    int num = 0;
    try
    {
        ① CalExp(2, -5);
    }
    ③ catch(const char* exception)
    {
        cout << exception << endl;
    }

    return 0;
}
```

Microsoft Visual Studio 디버그 콘솔  
exp가 음수

- 실행흐름

1. main문 시작, CalExp(2, -5) 호출

2. exp는 -5로 음수, 함수내에서  
throw 값을 던진다.

3. catch문은 그 값을 받아  
처리한다.

쉽고 간단하다.



## 13. 예외 처리

- C++ 에서 제공하는 예외 객체(Exception Class) : #include <stdexcept>

- logic error : 내부 로직의 문제로 인해 예상하지 않은 값에 대한 예외
  - a) Domain error : 적절하지 않은 매개변수 값 Ex) sqrt( -4), -4의 제곱근은 2i 이지만 허수는 지원하지 않기 때문
  - b) Invalid argument : 의도하지 않은 매개변수 값에 대한 예외
  - c) Length error : 문자열의 max size를 넘어선 연산 시
  - d) Out of range : 유효하지 않은 메모리 접근 시
- runtime error : 실행하기 전까지 알 수 없는 예외
  - a) Range error : 연산 결과가 변수의 타입과 다를 경우
  - b) Overflow error : 데이터 연산 시 생길 수 있는 overflow
  - c) Underflow error : 데이터 연산 시 생길 수 있는 underflow

★ 위 객체들은 프로그래머가 예상하는 예외에 대한 네이밍 및 예외전달 메시지 내용을 확인할 수 있는 멤버함수(what())를 제공할 뿐이지 실제로 무언가 처리해주지는 않는다.

★예외처리를 남발하면 프로그램에 부하가 걸릴 수 있다.



## 13. 예외 처리

- 다음은 Microsoft docs에서 제시하는 예시다.

```
void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
        throw invalid_argument("MyFunc argument too large.");
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }
    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    return 0;
}
```

결과

Microsoft Visual Studio 디버그 콘솔

MyFunc argument too large.

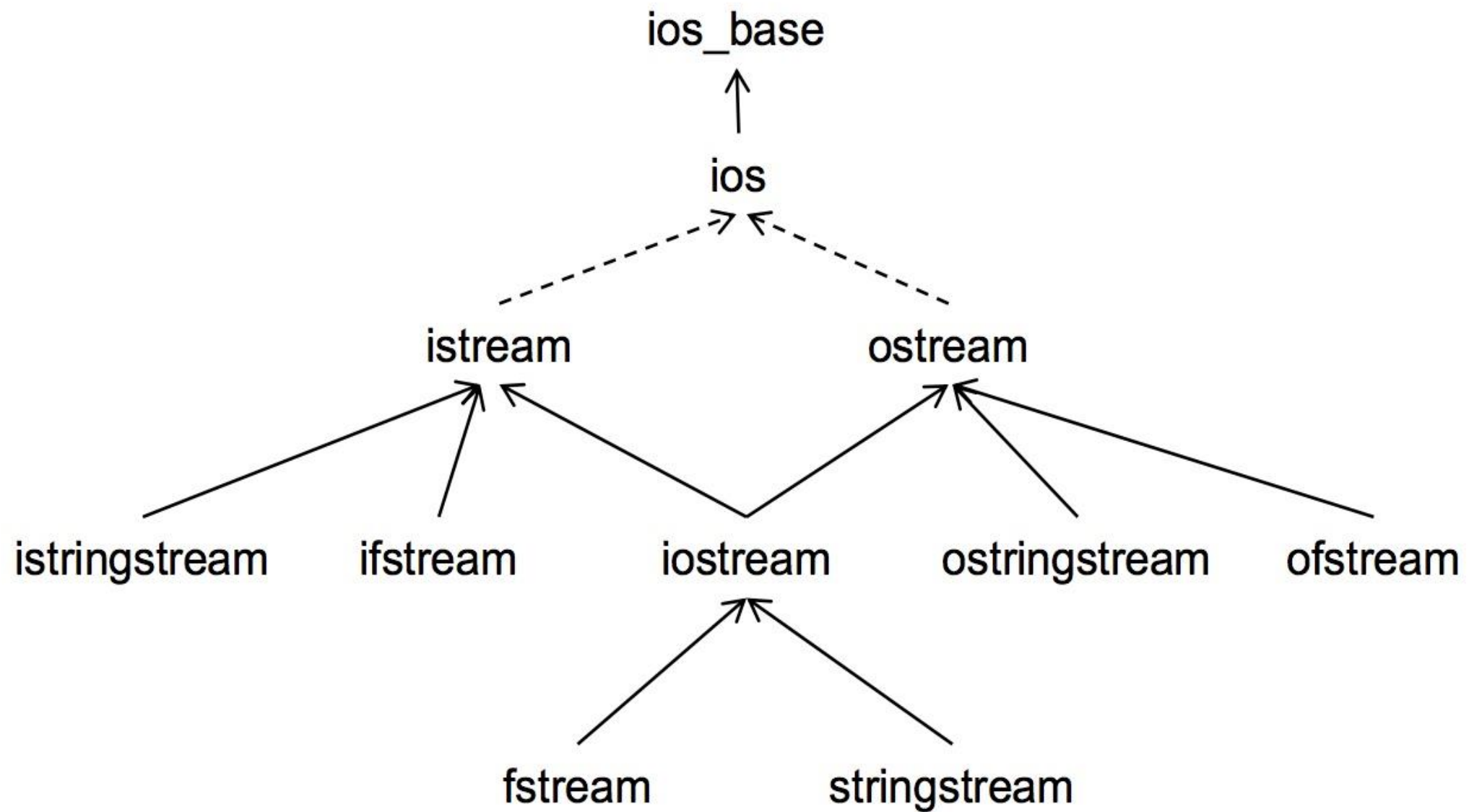
각 Exception Class에 대한 예제는 <http://www.cplusplus.com/reference/stdexcept/> 에 모두 나와있다.





## 14. 파일 입출력

## - C++ 파일 입출력 라이브러리 상속관계도



## 14. 파일 입출력

- 파일 입출력 스트림 : 파일과 프로그램을 연결하는 통로 `#include<fstream>`

- `ifstream` : 파일 -> 프로그램(read)
- `ofstream` : 프로그램 -> 파일(write)

- 파일 입출력 방식

- Text I/O (Input / Output) : ASCII, UNICODE 같은 문자로만 이루어진 데이터를 다루는 입출력 방식
- Binary I/O : 모든 데이터를 byte 수준으로 다루는 입출력 방식

1. 파일 입출력을 위해서는 스트림 객체를 생성해야 한다.

- `ofstream` fout;
- `ifstream` fin;

2. I/O를 위해서는 먼저 파일을 열어야 한다. 파일을 연다는 의미는 입출력 스트림에 프로그램과 파일을 연결하는 과정이다.

- `fout.open(매개변수);`
- `fin.open(매개변수);`

3. open 함수 원형

- `void open(const char* or const string& fileName, ios_base::openmode mode = ios_base::in or out);`
- open함수를 사용하지 않고 객체를 생성할 때 생성자로도 파일open이 가능하다.
- 파일 오픈의 유무는 반드시 확인해야한다. `if(fin.is_open())`

4. 매개변수

- 파일명에 해당하는 인자는 파일의 경로이다. 파일의 경로를 "C:\\users\\data\\abc.txt"로 예시를 들었을 때 역슬래시(\\)가 두개인 이유는 문자열 내에서 \과 함께 붙은 다른 문자가 \과 함께 특수문자로 인식이 될 수도 있기 때문이다.



## 14. 파일 입출력

### 5. 파일 입출력 모드

ios::in : read mode

ios::out : write mode

ios::ate : write를 위한 파일 열기, 열기 후 **파일포인터**를 파일의 끝에 둔다.

ios::app : 파일쓰기 시 항상 파일 끝에서 부터 쓰기 시작한다.

ios::trunc : 파일을 열 때 파일이 존재한다면 파일의 내용을 모두 지운다. ios::out 모드로 지정하면 **default**로 설정된다.

ios::binary : 바이너리 I/O로 파일을 연다. 이 모드가 지정되지 않으면 **default**모드는 텍스트I/O이다.

### 6. 파일 입출력 연산

a) >> , << 연산자 : 파일 입출력 시 자동 형 변환 지원

- fout << "abcde"; = abcde 문자열을 출력 스트림을 통해 파일에 저장

- fin >> 변수; 파일을 읽어 입력 스트림을 통해 변수에 저장

b) get(), put(char data)

- **int get()** : 파일에서 문자 하나(1 byte)를 읽고 **파일포인터**를 다음 위치로 이동 후 읽은 문자 **return**, 파일의 끝(EOF)에서 읽으면 -1 **return**

- **ostream& put(char data)** : 파일에 data를 기록, 표준 입출력 함수인 **put**과 헷갈리지 말 것

c) getline()

- **getline(istream fin, string line)** : 파일의 한 라인('\n', '\r' 을 만나는 곳)을 읽어 변수 line에 저장, **string** 헤더에서 지원하는 함수

- **getline(char\* line, int n)** : n 크기 만큼의 byte를 읽어 line에 저장(반드시 문자열의 끝 '\0' 공간을 신경 써야 함), **istream** 클래스 에서 지원하는 함수

- **string** 객체를 사용하는게 안전하고 편리하다

d) read / write : get, put(1 byte 단위)와 다르게 block단위로 입출력

- **istream& read(char\* s, int n)** : **istream**의 멤버 함수, 파일에서 n byte만큼 배열 s에 읽음, EOF시 중단

- **ostream& write(char\* s, int n)** : **ostream**의 멤버 함수, 배열 s에 있는 데이터 중 n byte를 파일에 저장

- **int gcount()** : 가장 최근에 읽은 byte수 **return**;



## 14. 파일 입출력

7. open으로 파일을 열었다면 작업종료 시 반드시 close로 파일을 닫아주어야 한다. 프로그램 비정상 종료 시 버퍼에 남아 있는 데이터가 제대로 처리되지 않을 수 있기 때문이다.

8. 파일포인터 : 쉽게 말해 깜빡이는 커서이다. 현재 포인터의 위치를 파악(Ex tellg(), tellp() ) 혹은 변경 (Ex seekg(), seekp() )하여 파일을 부분적으로 read & write할 때 쓰인다.



## 15. 참고문헌

---

오세만, [컴파일러 입문], 정익사, p22~p28

황기태, [명품 C++ Programming], 생능출판사

전병선, [C++ Essence], 와우북스

TCP School, [<http://tcpschool.com/>]

