

## 목차

목차.....	1
문서 내 표기법 .....	1
본문 수정 사항 .....	1
3장 Hello World.....	3
5장 타입1 .....	6
6장 연산자.....	9
7장 분기와 반복.....	11
9장 배열.....	15
10장 구조체 .....	17
11장 포인터 .....	20
12장 배열과 구조체와 포인터.....	22
13장 복합 타입의 모든 것.....	25
14장 함수1 .....	27
15장 함수2.....	32
16장 동적 메모리 할당 .....	35
17장 문자열 .....	38
18장 헤더 파일과 구현 파일.....	41
20장 객체지향 프로그래밍.....	44
21장 클래스와 객체 .....	45
22장 상속과 포함 .....	51
23장 다형성과 가상 함수.....	53
24장 예외 처리 .....	56
26장 접근 범위와 존속 기간.....	58
27장 타입 2.....	60
28장 네임스페이스 .....	63
29장 템플릿 .....	64
30장 입출력 .....	65

## 문서 내 표기법

- 정답 부분은 **녹색**으로 적었습니다.
  - 정답을 적는 칸이 없는 문제는, 보기 자체를 녹색으로 바꾼 경우도 있습니다.
- 변경된 부분은 **빨간색**으로 적었습니다.
  - 폰트가 잘못되어서 바꾼 경우도 해당합니다.
- 추가된 부분은 **파란색**으로 적었습니다.
- 네모 칸 안을 채우는 문제는 네모 칸 안에 답을 적었습니다.
- 괄호 안에 들어갈 말을 보기에서 고르는 문제는 괄호 안에 정답의 번호를 적었습니다.
- 주관식/서술형의 경우는 별도의 테이블을 만들어서 적었습니다.

## 본문 수정 사항

- 소스코드는 Courier New로 되어야 하는데, 문제에 그렇게 되지 않는 경우가 종종 있습니다. 그런 경우 폰트를 바꾸고 빨간색으로 바꿨습니다.
- 6장 이것만은 알고 갑시다. 3번 문제 3번째 열.
  - $? = d ? f + i;$  로 되어 있는 것  $? = d - f + i;$  로 수정.
  - 3번 : 문제의 코드에 `int sum = 1;` 이 추가되었음
- 14장 이것만은 알고 갑시다.
  - 2번의 세번째 문제 : 호출된 -> 호출하는 으로 수정
- 23장 이것만은 알고 갑시다.
  - 2번 문제 : 이형성 -> 다형성 으로 수정

- 26장 이것만은 알고 갑시다.
  - 1번 문제 : 수정한 부분 빨간색으로 표시했습니다.

### 3장 Hello World

이것만은 알고 갑시다.

1. 아래 프로그램은 틀린 곳이 몇 군데 있다. 올바르게 수정하자.

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

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

2. 다음 빈 칸에 들어갈 알맞은 말을 보기에서 고르시오.

(  ) 객체를 사용해서 문자열을 화면에 출력할 수 있다. 이 객체를 사용하기 위해서는 준비 과정이 필요하다.

ㄱ) main   ㄴ) include   ㄷ) cout   ㄹ) return

우리가 정말로 하려고 하는 일은 (  ) 함수 안에 넣는다.

ㄱ) main   ㄴ) iostream   ㄷ) using   ㄹ) namespace

문장은 (  )에 의해서 구분된다.

ㄱ) 쉼표(,)   ㄴ) 마침표(.)   ㄷ) 괄호(<)   ㄹ) 세미콜론(;

#### 4장 변수

##### Vitamin Quiz 음악을 숫자로 보관하는 방법

그림을 숫자로 보관할 수 있다고 치자. 그러면 음악은 어떻게 숫자로 보관할 수 있을까?

음악은 연속되는 소리라고 생각할 수 있다. 적당한 시간 간격을 뒤서 매 간격마다 소리의 크기를 재서 그 값을 보관함으로써 음악을 보관할 수 있다. 음악 파일의 속성을 보면 44 kHz 등의 값을 볼 수 있는데, 이는 1초에 44000번만큼 소리의 크기를 재서 보관했다는 뜻이다.

##### Exercise 4-1 변수의 정의 연습

조금 전의 예제에 변수 d를 추가하고 400을 넣자. 그리고 변수 d의 값을 화면에 출력하자.

예제에 다음과 같은 코드를 추가한다.

```
int d;  
d = 400;  
cout << d << "\n"
```

##### Exercise 4-2 변수 끼리 대입하기

조금 전의 예제에 변수 f를 추가하고 3000을 넣자. 다시 변수 d의 값을 변수 f에 넣고 화면에 출력하자.

예제에 다음과 같은 코드를 추가한다.

```
int f = 3000;  
f = d;  
cout << f << "\n";
```

이것만은 알고 갑시다

1. 아래 괄호를 채워보자. 마지막 두 개의 괄호는 '있다', '없다' 중에서 고르자. '있을까?'는 안 된다.

- C++에서 모든 정보는 ( 숫자 )로 표현한다.
- ( 변수 )는 숫자를 보관할 수 있는 공간(방)이다.
- ( cout ) 객체를 사용해서 변수의 값을 화면에 출력할 수 있다.
- 변수의 값을 다른 변수에 대입할 수 ( 있다. )
- 같은 이름의 변수가 두 개 존재할 수 ( 없다. )

2. 아래 변수 이름 중에서 올바르지 않은 이름을 찾아보자.

- 25th\_birthday : 숫자가 앞에 나올 수 없다.
- my first variable : 공백을 포함할 수 없다.
- i
- error! : ! 를 사용할 수 없다.
- volatile : 이미 정의된 키워드를 사용할 수 없다.
- int3
- l\_like\_a\_very\_long\_variable\_because\_it\_is\_very\_clear

3. 소스 코드와 그 실행결과를 보여주려고 했는데 출판사에서 원고를 독촉하는 바람에 소스 코드를 완성하지 못했다. 필자를 위해서 소스 코드를 완성해 주길 바란다!

```
#include [ <iostream> ]  
using namespace std;  
  
int main()  
{  
    int a = 1, b = 2, c = 3, d = 4, e = 5;
```

<pre>[ cout &lt;&lt; "      " &lt;&lt; a &lt;&lt; "\n      " &lt;&lt; a &lt;&lt; b &lt;&lt; "\n      "     &lt;&lt; a &lt;&lt; b &lt;&lt; c &lt;&lt; "\n      " &lt;&lt; a &lt;&lt; b &lt;&lt; c &lt;&lt; d     &lt;&lt; "\n" &lt;&lt; a &lt;&lt; b &lt;&lt; c &lt;&lt; d &lt;&lt; e &lt;&lt; "\n"          ];  return 0; }</pre>	
실행결과	1 12 123 1234 12345

## 5 장 타입 1

### Vitamin Quiz 타입의 범위 계산하기

조금 전의 예제를 통해서 각 정수 타입의 크기를 확인했다. 각 타입 별로 몇 가지의 상태를 가질 수 있는지 계산해보자. 자세한 방법은 바로 앞의 '여기서 잠깐' 코너에 실려있다.

short int, unsigned short int -> 2 바이트 -> 65536 상태  
int, unsigned int, long int, unsigned long int -> 4 바이트 -> 4294967296 상태

### Vitamin Quiz 값과 해석

동일한 값이라도 해석하기에 따라서 다르게 보이는 예는 실생활에서 얼마든지 찾아볼 수 있다. 몇 가지 예를 찾아보자.

마이크로소프트 워드로 작성한 문서 파일을 가정해보자. 이 파일을 워드 프로그램에서 열면 제대로 보이지만, 메모장 프로그램에서 열면 이상한 문자밖에 보이지 않는다. 동일한 파일이지만 이 파일을 해석하는 방법에 따라 다르게 보인다.

### Exercise 5.1 모든 타입의 크기 확인

앞의 예제에서는 정수 타입의 크기만 확인했다. 모든 타입의 크기를 확인할 수 있는 프로그램을 작성해보자.

```
cout << "bool" << " -> " << sizeof(bool) << "\n";
cout << "signed char" << " -> " << sizeof(signed char) << "\n";
cout << "unsigned char" << " -> " << sizeof(unsigned char) << "\n";
cout << "signed short" << " -> " << sizeof(signed short) << "\n";
cout << "unsigned short" << " -> " << sizeof(unsigned short) << "\n";
cout << "signed int" << " -> " << sizeof(signed int) << "\n";
cout << "unsigned int" << " -> " << sizeof(unsigned int) << "\n";
cout << "signed long" << " -> " << sizeof(signed long) << "\n";
cout << "unsigned long" << " -> " << sizeof(unsigned long) << "\n";
cout << "float" << " -> " << sizeof(float) << "\n";
cout << "double" << " -> " << sizeof(double) << "\n";
cout << "long double" << " -> " << sizeof(long double) << "\n";
```

### Exercise 5.2 정수를 사용한 문자 출력

앞의 예제에서 정수 65를 문자로 해석하면 'A'가 된다는 사실을 알았다. 마찬가지로 66은 'B'가 되고, 67은 'C'가 된다. 이런 사실을 이용해서 아래와 같은 문자열을 출력하는 프로그램을 작성해보자.

```
HELLO WORLD
```

```
char h = 72;
```

```
char e = 69;
char l = 76;
char o = 79;
char w = 87;
char r = 82;
char d = 68;
cout << h << e << l << l << o << " " << w << o << r << l << d << endl;
```

### Exercise 5.3 문제가 발생하는 형변환 정리

형변환 도중에 문제가 발생할 수 있는 경우를 정리해봤다. 구체적으로 어떤 경우에 어떤 문제가 생기는지 적어보자.

double -> float	float 타입이 보관하기에는 정밀도가 높은 실수라면 근사 값으로 변환된다. 비슷하지만 정확한 값은 아니다.
float -> short	float 변수의 정수 부분이 short 변수가 담을 수 있는 크기보다 큰 경우 이상한 값으로 바뀔 수 있다.
int -> float	int 타입의 값이 큰 경우 정밀도를 잃어버릴 수 있다. 비슷한 값이지만 정확한 값은 아니다.

이것만은 알고 갑시다

1. C++의 주요 타입을 나열해 보았다. 아래와 같이 비슷한 성격을 갖는 타입으로 묶어보자.

bool, int, double, long, unsigned char, wchar\_t, unsigned int, char, float, short, unsigned short, unsigned long

정수를 담는 용도의 타입	int, long, unsigned int, short, unsigned short, unsigned long
문자를 담는 용도의 타입	char, unsigned char, wchar_t
실수를 담는 용도의 타입	double, float
0과 양수만 담을 수 있는 타입	bool

2. 빈 칸을 채워보자.

- 1바이트는 ( 8 ) 비트다.
- 타입의 크기를 알아내기 위해서 ( sizeof ) 연산자를 사용한다.

3. 변수에 저장한 값이 같더라도 타입에 따라 다르게 해석한다는 점을 확인하기 소스코드다. 아래와 같은 실행 결과가 나올 수 있도록 소스 코드의 빈 곳을 채워보자.

```
#include <iostream>
using namespace std;
int main()
{
    float f = 65.5f;

    cout << "float = " << f << endl;
    cout << "int = " << [(int)]f << endl;
    cout << "char = " << [(char)]f << endl;

    return 0;
}
```

실행결과

float = 65.5

int = 65

char = A



## 6 장 연산자

### Vitamin Quiz

다음의 코드는 temp라는 변수를 사용해서 a와 b의 값을 바꾸고 있다. 제 3의 변수를 사용하지 않고 오직 a, b만 사용해서 a, b의 값을 바꿔보자. (힌트: 비트단위 XOR 연산을 사용)

```
int a, b, temp;
a = 3;
b = 5;

// a와 b의 값을 바꾼다.
temp = a;
a = b;
b = temp;
```

비트 단위 XOR 연산은 연산을 다시 한 번 수행했을 때 원래의 값을 복원하는 특징을 갖는다. 예를 들어,  $3 \oplus 5$  는 6 인데, 다시  $6 \oplus 5$  는 3 이 된다. 즉  $x \oplus y \oplus y$  는 x 가 된다고 정리할 수 있다. 이런 특징을 이용하면 아래처럼 할 수 있다.

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

### Exercise 6.1 연산자를 사용한 입장 제한

이 책을 읽고 실력이 급성장한 여러분은 게임 회사에 들어갔다. 입사한 지 얼마 안되어 테트리스 개발팀에서 아래와 같은 조건을 만족하지 못하는 사용자의 입장을 제한하는 코드를 작성해달라고 부탁해왔다. 여러분의 실력을 발휘할 때가 되었다. 관계연산자와 논리 연산자를 사용해서 아래 네모 칸을 채우자.

테트리스 게임의 초보 채널 입장을 위한 조건

1. 나이가 12세 이상이어야 함
2. 내공이 1400 ~ 1800 사이인 사람
3. 불량 사용자가 아닌 사람

```
int age; // 나이
int rate; // 내공
bool baduser; // 불량사용자인지 여부
```

// 새로 입장하려는 사용자에게 대해서 위의 변수가 채워졌다고 가정

```
bool ok; // 입장이 가능한지 여부
ok = [ age >= 12 && rate >= 1400 && rate <= 1800 && !baduser];
```

### Exercise 6.2 10진수, 2진수, 16진수 변환 연습

이 변환 연습은 많이 할 수록 좋다. 실제로 쓸 일도 많고, 조금만 연습해도 금방 익숙해진다. 빈 칸을 채우자.

2진수	16진수	10진수
1101	D	13
10100011	A3	163
110011101001	CE9	3305
1010	0xA	10
10011011	0x9B	155
1111101100	0x3EC	1004

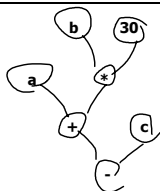
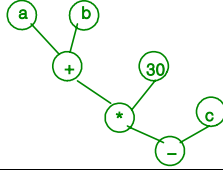
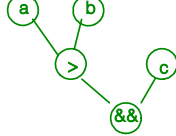
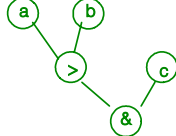
### Exercise 6.3 컴퓨터가 되어보자

'빨간색 부분의 값 바꾸기' 예제를 복습해보려고 한다. 조금 전에는 컴퓨터가 필요한 비트 단위 연산을 모두 해주었지만, 이번에는 여러분이 직접 책에 적어가면서 계산해보자.

<code>color = 0x1234;</code>	0001001000110100
<code>0x07ff</code>	0000011111111111
<code>color_temp = color &amp; 0x07ff;</code>	0000001000110100
<code>red = 30;</code>	00000000000011110
<code>red_temp = red &lt;&lt; 11</code>	1111000000000000
<code>color_finished = color_temp   red_temp</code>	1111001000110100

이것만은 알고 갑시다

1. 아래 연산들이 어떤 순서로 수행되는지 그림을 그려보자.

<code>a + b * 30 - c</code>	
<code>(a + b) * 30 - c</code>	
<code>a &gt; b &amp;&amp; c</code>	
<code>a &gt; b &amp; c</code>	

2. 다음 두 연산의 결과를 계산해보자. 책을 꼼꼼하게 읽었다면 굳이 2진수로 변환할 필요도 없이 5초 이내에 답을 적을 수 있다.

<code>32 &gt;&gt; 1</code>	16
<code>-32 &gt;&gt; 1</code>	-16

3. 아래 연산들의 결과는 어떤 타입인지 적어보자.

<code>short a = 20; float f = 30.0f; ? = a + f;</code>	float
<code>char c = 'A'; short s = 5; ? = c + s;</code>	int
<code>int i = 3; float f = 20.0f; double d = 99.9; ? = d - f + i;</code>	double

## 7 장 분기와 반복

Vitamin Quiz switch/case와 if

다음 절을 읽기 전에 switch/case와 if 문을 비교해보자. 아래에서 switch/case를 사용하는 게 좋은 경우는 S로, if가 좋은 경우는 I로 표기하자.

S	하나의 변수를 여러 대상 값과 비교하는 경우
I	관계연산을 통해서 분기해야 하는 경우
I	여러 변수의 값을 확인하면서 분기해야 하는 경우
I	한 번의 분기만 있으면 되는 경우

Vitamin Quiz for, while, do while

for, while, do while을 비교해보자. 아래에서 for를 사용하는 게 좋은 경우는 F로, while이 좋은 경우는 W, do while이 좋은 경우는 D로 표기하자.

D	반드시 한 번은 반복해야 하는 경우
F	시작 값과 끝 값, 증가치가 정해져 있는 경우
W	특정 상황이 발생할 때까지 계속해서 반복해야 하는 경우

Exercise 7.1 학점 정하기

if, else if, else 명령을 사용해서 점수에 따라 A~F의 성적을 출력하는 프로그램을 작성하자. 점수의 기준은 다음과 같다.

91~100 점: A  
81~90 점: B  
71~80 점: C  
61~70 점: D  
0~60 점: F

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

int main()
{
    unsigned int score = 85;
    char grade = '?';

    if (score > 90 && score <= 100)
        grade = 'A';
    else if (score > 80 && score <= 90)
        grade = 'B';
    else if (score > 70 && score <= 80)
        grade = 'C';
    else if (score > 60 && score <= 70)
        grade = 'D';
    else
        grade = 'F';

    cout << "score = " << score << ", grade = " << grade <<
    "\n";
    return 0;
}
```

Exercise 7.2 학점 별 안내 문구 출력

switch/case 명령을 사용해서 학점 별로 안내 문구를 출력하는 프로그램을 작성하자. 학점 별 안내 문구는 다음과 같다.

A: 참 잘했어요  
B: 잘했어요  
C, D: 분발하세요  
F: 찻찻

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

int main()
{
    char grade = 'C';

    switch( grade )
    {
        case 'A':
            cout << "참 잘했어요\n";
            break;
        case 'B':
            cout << "잘했어요\n";
            break;
        case 'C':
        case 'D':
            cout << "분발하세요\n";
            break;
        case 'F':
            cout << "찻찻\n";
            break;
        default:
            cout << "알 수 없는 성적입니다.\n";
    }

    return 0;
}
```

Exercise 7.3 while로 구구단을 외자  
중첩된 while 명령을 사용해서 구구단을 출력해보자.

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

int main()
{
    int i = 1;
    while (i <= 9)
    {
        cout << "==== " << i << " ====\n";

        int j = 1;
        while (j <= 9)
        {
            cout << i << " x " << j << " = " << i * j << "\n";
            ++j;
        }
        ++i;
    }

    return 0;
}
```

이것만은 알고 갑시다

1. if/elseif/else로 이루어진 비교 구문을 switch/case를 사용하게 고쳐보자. switch/case를 사용하도록 고칠 수 없는 경우라면 왜 그런지 적어보자.

if/elseif/else	switch/case
<pre>char key; if ( 'w' == key ) {     cout &lt;&lt; "전진" &lt;&lt; endl; } else if ( 'a' == key ) {     cout &lt;&lt; "좌회전" &lt;&lt; endl; } else if ( 's' == key ) {     cout &lt;&lt; "후진" &lt;&lt; endl; } else if ( 'd' == key ) {     cout &lt;&lt; "우회전" &lt;&lt; endl; } else {     cout &lt;&lt; "알 수 없는 키" &lt;&lt; endl; }</pre>	<pre>char key; switch( key ) {     case 'w':         cout &lt;&lt; "전진" &lt;&lt; endl;         break;     case 'a':         cout &lt;&lt; "좌회전" &lt;&lt; endl;         break;     case 's':         cout &lt;&lt; "후진" &lt;&lt; endl;         break;     case 'd':         cout &lt;&lt; "우회전" &lt;&lt; endl;         break;     default:         cout &lt;&lt; "알수없는키" &lt;&lt; endl;         break; }</pre>
<pre>int rate; bool baduser; if ( rate &gt;= 1400 &amp;&amp; rate &lt;= 1800 &amp;&amp; !baduser ) {     cout &lt;&lt; "입장 가능!" &lt;&lt; endl; } else {     cout &lt;&lt; "입장 불가능!" &lt;&lt; endl; }</pre>	<p>switch/case에서는 관계연산자를 사용한 비교를 할 수 없다.</p>

2. 삼항연산자를 사용한 조건식을 if/else를 사용하게 고쳐보자.

삼항연산자	if/else
<pre>bool largefile; unsigned int filesize; largefile = filesize &gt; 1024 * 1024 ? true : false;</pre>	<pre>bool largefile; unsigned int filesize; if ( filesize &gt; 1024 * 1024 )     largefile = true; else     largefile = false;</pre>

3. for 명령을 사용한 코드를 while을 사용하게 고쳐보자.

for	while
<pre>int factorial = 1; int sum = 1; for ( int i = 1; i &lt;= 5; ++i ) {     sum *= i; }</pre>	<pre>int factorial = i; int i = 1; while ( i &lt;= 5 ) {     sum *= i;     ++i; }</pre>

4. 왼쪽의 문제 상황에 알맞은 명령을 짝지어 보자

1부터 10까지의 덧셈처럼 구  
간이 정해져 있는 반복이 필  
요한 경우

학점에 따라서 알맞은 문자열  
을 출력하는 예제와 같이 정  
해진 몇 개의 입력 값을 기준  
으로 분기가 필요한 경우

주어진 조건을 만족하는 동안  
반복하되, 적어도 한 번은 꼭  
실행해야 하는 경우

다양한 조건이나 범위, 논리  
연산을 사용하여 분기가 필요  
한 경우

간단한 반복만 필요한 경우



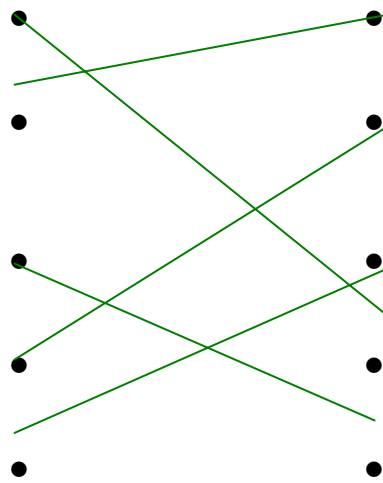
switch/case

if/elseif/else

while

for

do/while



## 9 장 배열

Vitamin Quiz 배열의 전체 크기는 어떻게 될까?

다음 절을 읽기 전에 배열의 크기에 대해서 논리적으로 생각하는 시간을 갖자. 예를 들어 int 타입의 원소 3개를 갖는 배열을 생각해 보자. 보통의 32비트 시스템이라면 int 타입의 크기는 4바이트다. 그렇다면 이 배열은 크기는 얼마일까?

각 원소의 크기를 모두 합친 값이 배열의 크기가 된다. 각 원소의 크기가 4바이트이므로 원소 3개를 갖는 배열의 크기는 12바이트가 된다.

### Exercise 9.1 배열을 사용해서 평균 구하기

이번에는 '도전 퀴즈왕' 게임을 만드는 개발팀에서 여러분에게 도움을 요청했다. 게임에 참여한 10명의 평균 점수를 구해서, 평균보다 못한 사람을 탈락시키는 기능이 필요하다고 한다. 아래와 같이 배열에 점수가 보관되어 있다. 10명의 평균을 구하는 프로그램을 작성해 보자.

```
unsigned int scores[10] = { 10, 100, 94, 36, 72, 88, 60, 60, 80, 24 };  
unsigned int scores[10] = { 10, 100, 94, 36, 72, 88, 60, 60, 80, 24 };  
  
// 총점을 구한다  
int sum = 0;  
for ( int i = 0; i < 10; ++i )  
    sum += scores[i];  
  
// 평균을 구한다  
int ave = sum / 10;  
cout << "ave = " << ave << endl;
```

### Exercise 9.2 문자열 뒤집기

'도전 퀴즈왕' 게임에 나오는 문제 중에 뒤집어진 글씨를 보고 빨리 알아 맞추는 유형이 있다고 한다. 아래와 같은 문자열이 주어졌을 때, 문자를 뒤집어서 화면에 출력하는 프로그램을 작성해 보자. (문자열의 내용이 무슨 뜻인지는 인터넷을 찾아보자.)

```
char example[] = "scientia est potentia";  
char exampel[] = "scientia est potentia";  
  
// 배열의 크기를 구한다.  
// (널문자를 고려해서 1을 뺀다)  
int len = sizeof(str) - 1;  
  
// 배열의 원소를 역순으로 탐색하면서 문자열을 출력한다.  
for (int i = len - 1; i >= 0; --i)  
{  
    cout << str[i];  
}  
  
cout << "\n";
```

이것만은 알고 갑시다

1. 아래 이어지는 예제 코드에서 잘못된 부분을 찾아서 설명해 보자.

<pre>int arr[3]; arr[1] = 10; arr[2] = 20;</pre>	원소의 인덱스는 0~2가 된다. arr[3]은 존재하지 않으며, 대신에 arr[0]이 존재한다.
--	---

arr[3] = 30;	
char arr[10]; arr[20] = 'a';	배열의 크기를 벗어난 인덱스를 사용하면 프로그램이 비정상 종료할 수 있다.
float arr[5]; arr = { 0.1f, 0.2f, 0.3f };	배열의 초기화 리스트는 변수를 정의할 때만 사용할 수 있다.

2. 다음 프로그램의 결과로 출력되는 값은?

<pre>int arr1[10]; cout &lt;&lt; "arr1[5] = " &lt;&lt; arr1[5] &lt;&lt; endl; int arr2[10] = { 1, 2, 3, 4, 5 }; cout &lt;&lt; "arr2[5] = " &lt;&lt; arr2[5] &lt;&lt; endl;</pre>	<p>arr1[5] 에는 쓰레기 값이 들어있다. 배열을 초기화 하지 않았기 때문이다. arr2[5]에는 0 이 들어있다. 배열을 초기화 했기 때문인데, 비록 arr2[0]~arr2[4] 만 초기화 되었다 할지라도 나머지 원소들은 0 으로 초기화 한다.</p>
--	--

3. 아래 배열의 크기를 byte 단위로 적어보자.

char c[5];	5 bytes
char c[] = "Hello?";	7 bytes
double d[3];	24 bytes
short s[4];	8 bytes



## 10 장 구조체

Vitamin Quiz 틀과 붕어빵은 생활 속에

붕어빵 틀과 붕어빵의 관계는 우리의 생활 곳곳에서 발견할 수 있다. 일상 생활에서 볼 수 있는 붕어빵 틀과 붕어빵의 예를 찾아보자.

공장에서 만드는 모든 제품들을 예로 들 수 있다. 노트북, 핸드폰, 밥그릇 등등 공장에는 붕어빵 틀의 역할을 하는 설비가 있고 그 결과 붕어빵에 해당하는 제품이 생산된다.

Vitamin Quiz 구조체 변수의 덧셈?

구조체도 하나의 타입이지만 기본 타입과 다른 점이 많이 있다. 구조체 변수끼리 덧셈을 하는 것이 가능할까? 아니라면 그 이유가 무엇일까?

불가능하다. 그 이유는 우리가 만든 타입이기 때문이다. 그래서 컴퓨터는 덧셈을 어떻게 수행해야 할지 모른다. 연산자 오버로딩을 배우면 덧셈을 어떻게 수행해야 하는지 컴퓨터에게 알려줄 수 있다.

Exercise 10.1 모든 타입의 멤버를 갖는 구조체

지금까지 배운 모든 타입의 멤버를 갖는 구조체를 정의해보자.

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

struct AllType
{
    signed char      sc;
    unsigned char uc;
    signed short  ss;
    unsigned short  us;
    signed int     si;
    unsigned int    ui;
    signed long     sl;
    unsigned long   ul;
    float          f;
    double          d;
    long double     ld;
};

int main()
{
    AllType at;

    at.sc = -100;
    at.uc = 200;
    at.ss = -30000;
    at.us = 60000;
    at.si = -2000000000;
    at.ui = 4000000000;
    at.sl = -2000000000;
    at.ul = 4000000000;
    at.f = 1234.56f;
    at.d = 12345678.9;
    at.ld = 12345678.9;

    return 0;
}
```

### Exercise 10.2 만약 세상에 구조체가 없었다면.....

Point 구조체를 사용해서 10개의 점을 만들어보자. 초기 값은 여러분 마음대로 해도 좋다. 그리고 나서 이 프로그램과 동일한 프로그램을 구조체를 사용하지 않고 작성해보자.

Point 구조체를 사용한 예제는 간단하므로 생략한다. 다음은 구조체를 사용하지 않고 10개의 점에 관한 정보를 보관한 프로그램이다. Point 구조체를 흉내내서 만들기는 했지만, 억지스러운 구현이라고 볼 수 있다. 커다란 프로젝트의 경우에는 소스 코드가 매우 지저분해져서 유지보수하기가 힘들어진다.

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

int main()
{
    int x[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int y[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i = 0; i < 10; ++i)
    {
        cout << "( " << x[i] << ", " << y[i] << ")\n";
    }

    return 0;
}
```

이것만은 알고 갑시다

1. 다음 문장의 빈 곳에 들어갈 알맞은 말을 보기에서 고르시오.

구조체를 사용해서 여러 변수들을 ( ☐ ) 관리할 수 있다.

ㄱ) 재밌게    ㄴ) 배열처럼    ㄷ) 모아서    ㄹ) 모질게

구조체와 구조체의 변수는 ( ☐ )와/과 ( ☐ )의 관계다

ㄱ) 붕어빵, 붕어빵 틀    ㄴ) 붕어빵 틀, 붕어빵    ㄷ) 잉어빵, 붕어빵 틀    ㄹ) 아버지, 아들

구조체는 ( ☐ ) 타입이다.

ㄱ) 내    ㄴ) 내장(built-in)    ㄷ) 사용자 정의    ㄹ) 외장

2. 아래의 출력 결과가 나올 수 있도록 구조체를 초기화 부분을 완성해보자.

```
struct Student
{
    char bloodType;
    int stdNumber;
    float grade;
};

int main()
{
    Student s = [{ 'A', 337, 4.5f }];
    cout << s.bloodType << " " << s.stdNumber << " " << s.grade << endl;
    return 0;
}
```

실행결과

A 337 4.5

3. 앞서 나온 Student 구조체 변수의 각 멤버를 복사하는 부분을 완성해보자.

```
int main()
{
```

```
Student s1, s2;
s1.bloodType = 'O';
s1.stdNumber = 337;
s1.grade = 3.3f;

// s1의 각 멤버를 s2에 복사
[  s2.bloodType = s1.bloodType;
  s2.stdNumber = s1.stdNumber;
  s2.grade = s1.grade;          ];
return 0;
}
```

## 11 장 포인터

Vitamin Quiz void\* 타입의 포인터 변수의 크기는?

void\* 타입의 포인터 변수의 크기는 몇 바이트일까? sizeof() 연산자를 사용해서 구해보자.

아래와 같은 코드를 실행해보자. 32 비트 시스템이라면 타입이 상관없이 모든 포인터 변수의 크기는 32 비트가 된다.

```
cout << sizeof( void* ) << endl;
```

Vitamin Quiz 포인터 변수를 가리키는 포인터 변수

int\* 타입의 포인터 변수를 가리키는 포인터 변수를 만들 수 있다. 이 포인터 변수의 타입은 어떻게 될까?

int\*\* 처럼 정의할 수 있다. 다음 코드를 참고하자.

```
int a;
int* pa = &a;
int** ppa = &pa;
```

Exercise 11-1 일부러 틀리기

앞의 예제를 조금 수정해서 다른 타입의 변수를 가리키도록 해보자. 어떤 예러가 나는지 기억해두자. 또, unsigned int\* 타입의 포인터 변수는 int 타입의 변수를 가리킬 수 있는지도 확인해보자.

다른 타입의 변수를 가리킬 수 없다는 점을 꼭 기억해두자. unsigned int\* 타입의 포인터 변수 역시 int 타입의 변수를 가리킬 수는 없다.

Exercise 11-2 빈칸 채우기

앞의 예제를 보지 말고 빈칸을 채워보자. 포인터 변수를 어떻게 정의하면 아래와 같은 결과가 나올까?

```
float f1 = 10.0f;
float f2 = 20.0f;
[const float* ] p = &f1;
p = &f2; // OK
*p = 30.0f; // FAIL
```

이것만은 알고 감시다

1. 각 타입 별로 보관할 수 있는 데이터의 종류를 적어보자.

short, int, long	정수
char, wchar_t	문자
float, double	실수
int*	int타입 변수의 주소

2. 빈 칸에 들어갈 알맞은 말을 고르시오.

메모리는 (    e    ) 단위로 주소가 부여되어 있다.

ㄱ) 한 비트    ㄴ) 가변적인    ㄷ) 아주 작은    ㄹ) 한 바이트

포인터 변수 p가 변수 a의 주소를 가지고 있는 상황을 "p가 a를 (    e    ) 있다"라고 표현한다.

ㄱ) 무시하고    ㄴ) 먹고    ㄷ) 가리키고    ㄹ) 가지고

포인터 변수의 이름 앞에 ( & )을/를 붙이면 '포인터가 가리키는 변수'의 의미가 된다.

ㄱ) !    ㄴ) &    ㄷ) \*    ㄹ) ^

3. 각각의 프로그램에서 잘못된 부분을 찾아서 그 이유를 적어보자.

<pre>char c = 'A'; int* pc = &amp;c;</pre>	pc는 int 타입 변수의 주소만 보관할 수 있다.
<pre>const float f1 = 99.9f; float f2 = 0.1f; f1 = f1 + f2;</pre>	f1을 const로 정의했으므로 f1의 값을 수정할 수 없다.
<pre>int a = 2580; const int * p1 = &amp;a; int const * p2 = &amp;a; int * const p3 = &amp;a;  *p1 = 1234; *p2 = 2345; *p3 = 3456;</pre>	const int * 로 선언한 경우 포인터가 가리키는 변수의 내용을 수정할 수 없다.

4. 각각의 프로그램에서 빈 곳을 채워보자.

<pre>int* p = [ NULL ]; int a = 1004; p = &amp;a;  if ( [ NULL ] != p )     *p = 1005;</pre>
<pre>char c = 'A'; int i = 65; float f = 65.0f; [void]* p; p = &amp;c; p = &amp;i; p = &amp;f;</pre>

## 12 장 배열과 구조체와 포인터

Vitamin Quiz 변수 `i` 없이 탐색할 수 있을까?

정수 `i`를 사용하지 않고 포인터 `p`만 사용해서 배열을 탐색할 수 있게 수정해달라는 요청이 들어왔다. 앞의 예제를 수정해서 배열의 모든 원소의 값을 출력하는 프로그램을 작성해보자. 배열 `nArray`와 포인터 `p` 외에 그 어떤 변수를 정의해서도 안 된다.

```
int main()
{
    int nArray[10];
    int* p = &nArray[0];

    while ( p != &nArray[10] ) // 마지막 원소를 넘어서면 멈춘다.
    {
        // 첫번째 원소와의 차이를 통해 인덱스를 구한다.
        *p = p - &nArray[0];

        // 다음 원소를 가리키게 한다.
        ++p;
    }

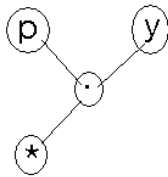
    return 0;
}
```

Vitamin Quiz 연산자 우선 순위

앞의 예제에서 `(*p).y`가 아니라 `*p.y`라고 하면 어떤 오류가 발생하는지 확인해보자. 각각의 경우에 어떻게 해석되는지 그림을 그려서 표현해보자.

`(*p).y`는 "포인터 `p`가 가리키는 구조체 변수의 멤버 `y`"를 의미한다. 반면에 `*p.y`라고 하면 "구조체 변수 `p`의 멤버 `y`가 가리키는 변수의 값"을 의미한다. 멤버를 의미하는 `.` 연산자가 가리키는 변수를 의미하는 `*` 연산자보다 우선 순위가 높기 때문이다.

그림을 그려보면 이렇다. `*`는 단항연산자이므로 입력을 하나만 받는다.



Exercise 12-1 배열을 가리키는 포인터를 사용해서 배열 탐색하기

조금 전의 예제를 수정해서, 포인터 `p`를 사용해서 배열의 모든 원소의 값을 출력하는 프로그램을 작성해보자.

```
for ( int i = 0; i < 20; ++i )
{
    cout << (*p)[i] << endl;
}
```

Exercise 12-2 사용자 정보 채우기

최고의 게임 사이트 '뇌자극 게임'에서 사용자의 정보를 보관하는 프로그램의 작성을 의뢰해왔다. 며칠간의 분석 끝에 다음과 같은 사용자 구조체를 만들기로 했다. 아래 나오는 정보를 이용해서 사용자 3명의 정보로 초기화하는 프로그램을 작성해보자. ( 배열의 초기

화 기능을 사용하자.)

```
struct User
{
    char userID[20];
    char passwd[20];
    int  scores_per_stage[5];
    unsigned long magicPt;
    unsigned long healthPt;
};

User user[3] = [ {{ "denzang", "sd933k", {80, 56, 72, 86, 91}, 300, 10010 },
                  { "zzazang", "!!sd487", {100, 98, 100, 100, 91}, 20000, 19000 },
                  { "gochuzang", "df321#4", {34, 54, 70, 48, 54}, 400, 5000 }} ];
```

정보

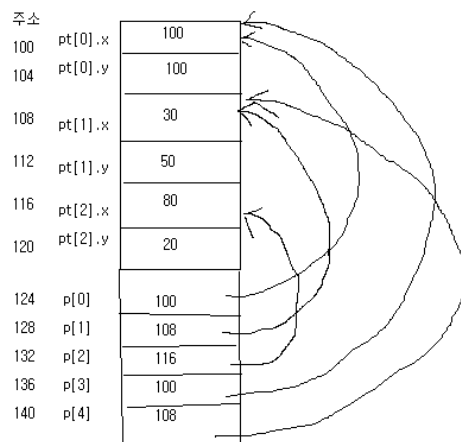
! 이 정보는 p.269의 필드 테스트 3번 문제의 테이블을 옮겨 주세요 !

### Exercise 12-3 메모리 상태 확인하기

다음의 소스 코드가 실행되었을 때의 메모리 상태(layout)를 그림으로 그려보자.

```
struct Point
{
    int x, y;
};

Point pt[3] = { {100, 100}, {30, 50}, {80, 20}};
Point* p[5] = { &pt[0], &pt[1], &pt[2], &pt[0], &pt[1] };
```



이것만은 알고 갑시다

1. 각각의 예제에서 최종적으로 r이 갖는 값을 맞춰보자.

<pre>int a = 100;    // a의 주소는 0x200 int *p = &amp;a; int *r = p + 2;</pre>	<p>0x208</p> <p>int 변수의 크기는 4이므로 p + 2 는 실제로 0x200 + 8이 된다.</p>
<pre>short arr[5]; short *p = &amp;arr[3]; short *q = &amp;arr[0]; int r = p - q;</pre>	<p>3</p> <p>두 원소의 인덱스 차이값이 결과로 나온다.</p>

2. 아래 프로그램에서 출력되는 결과는 무엇일까?

```
float scores[5] = {0};
for ( int i = 0; i < 5; ++i )
```

```

{
    if ( scores == &scores[i] )
    {
        cout << i << endl;
        break;
    }
}

```

0

배열의 이름은 첫번째 원소의 주소와 같다.

3. 아래와 같이 배열의 제일 앞 원소를 가리키는 포인터 p가 있을 때, p를 사용해서 3번째 원소의 값을 바꿀 수 있도록 빈 칸을 채워보자. 본문에서 배운 두 가지 방식 모두 적어보자.

```

float arr[5];
float* p = &arr[0];
[p[3] ] = 30.3f;    // 3번째 원소에 접근

```

4. 배열을 가리키는 포인터를 정의해보자. 연산자의 우선 순위를 조심할 필요가 있다.

```

long arr[20];
[long (*p)[20]] = &arr;

```

5. 포인터를 사용해서 구조체의 멤버에 접근하는 예제다. p를 사용하여 x의 값을 30으로 바꿀 수 있도록 빈 칸을 채워보자. 본문에서 배운 두 가지 방식 모두 적어보자.

```

struct ABC
{
    int x;
} abc;
ABC* p = &abc;
[ p->x ] = 30;    // abc.x의 값을 30으로 바꾼다.

```



## 13 장 복합 타입의 모든 것

Vitamin Quiz 포인터 변수에 대한 레퍼런스

레퍼런스는 어떤 타입의 변수라도 참조할 수 있다. `int*` 타입의 변수를 참조하는 레퍼런스 변수를 정의해보자.

```
int* p = NULL;
int*& r = p;
```

Exercise 13-1 공용체의 특징 이해

다음 코드에서 최종적으로 `uni.i`가 갖는 값은 어떻게 될까? 16진수로 답해보자.

```
union UNI
{
    int i;
    char c;
} uni;
uni.i = 0x12345678;
uni.c = 0x90;
```

0x12345690

`uni.c`의 값을 변경하는 것은 `uni.i`의 마지막 바이트를 변경하는 것과 같다.

Exercise 13-2 열거체 연습

여러분이 좋아하는 10개의 국가에 대한 심볼을 정의하는 열거체를 만들어보자.

```
enum { KOREA, SWITERLAND, FRANCE };
예를 들어서 3개국만 정의했다.
```

Exercise 13-3 배열 타입에 대한 별명

`typedef`를 사용해서 "int 타입의 원소를 10개 갖는 배열" 타입을 `ARR_INT_10`이라는 새로운 타입 이름으로 정의해보자.

```
typedef int ARR_INT_10[10];
```

Exercise 13-4 다차원 배열에 구구단 결과 보관하기

2차원 배열과 중첩된 반복 명령을 사용해서 구구단의 결과를 2차원 배열에 보관하는 프로그램을 작성해보자. 여러분이 만들어낸 배열은 다음과 같이 사용할 수 있어야 한다.

```
cout << "3 x 9 = " << arr[3][9] << endl;

int arr[10][10];
for ( int i = 1; i <= 9; ++i )
{
    for ( int j = 1; j <= 9; ++j )
    {
        arr[i][j] = i * j;
    }
}

cout << "3 x 9 = " << arr[3][9] << endl;
```

이것만은 알고 갑시다

1. 다음 괄호 안에 들어갈 알맞은 말을 고르시오.

공용체의 모든 멤버들은 메모리 공간을 (   e   )한다.

ㄱ) 제거   ㄴ) 기부   ㄷ) 대여   ㄹ) 공유

소스 코드에서 숫자 값을 직접 사용하는 대신에 알아보기 쉬운 이름을 가진 (          ) 사용하는 것이 좋다.

ㄱ) 철수를    ㄴ) 구조체의 멤버를    ㄷ) 열거체의 심볼들을    ㄹ) 배열을

열거체에서 정수 타입으로의 (          ) 형변환은 가능하지만 그 반대의 경우에는 (       ) 형변환을 필요로 한다.

ㄱ) 빠른, 느린    ㄴ) 암시적, 명시적    ㄷ) 명시적, 암시적    ㄹ) 느린, 빠른

레퍼런스는 다른 변수의 (          )처럼 행동한다.

ㄱ) 별명    ㄴ) 엄마    ㄷ) 물주    ㄹ) 빗쟁이

구조체의 비트 필드는 내부적으로 비트 단위 논리 연산을 사용하기 때문에 우리가 직접 비트 단위 논리 연산을 사용하는 것보다 (          ).

ㄱ) 빠르다    ㄴ) 빠르지 않다    ㄷ) 빠를까?    ㄹ) 멍있다

2. 아래 코드에서 주석이 되어 있는 1, 2 번 줄의 차이점에 대해서 설명해보자.

```
int target1 = 20;
int target2 = 30;
int& ref = target1; // 1
ref = target2;      // 2
```

1.ref는 target1의 별명이 된다.  
2.target1에 target2의 값을 대입한다.

3. 각각의 프로그램이 올바르게 동작할 수 있도록 빈 칸을 채워보자.

```
[typedef char[] MY_CHAR;    ]
MY_CHAR str = "Hello, World!";
```

```
char c = '1';
char* pc = &c;
[char** ppc] = &pc;
```

4. 아래와 같은 구조체에서 멤버 a, b, c가 각각 3, 4, 1 비트를 차지하도록 수정하자. b, c 사이에 1비트의 여유 공간도 추가하자.

```
struct Flags
{
    int a : 3;
    int b : 4;
    unsigned int : 1;
    bool c : 1;
};
```

## 14 장 함수 1

Vitamin Quiz 포인터 변수를 사용하는 좋은 습관

Exercise 14-4에 나오는 Sub() 함수는 안전성과 관련한 문제가 있다. 필자는 이미 포인터 타입의 변수를 사용할 때 주의할 점을 설명한 바 있다. Sub() 함수가 안전성을 지닌 함수가 되게 수정해보자.

```
void Sub( int i, int* p, int& r)
{
    i = 10;
    if ( p )
        *p = 20;
    r = 30;
}
```

Vitamin Quiz 이차원 배열의 전달

이차원 배열을 인자로 전달할 때 내부적으로 사용하는 포인터의 타입은 무엇일까? 가상의 코드를 작성하고 메모리 구조를 그려보자. 그리고 왜 대괄호의 맨 앞만 비워주어야 하는지 생각해보자.

```
void Using2DArray( int (*arr)[3] )
{
    // 중간 생략
}

int main ()
{
    int arr[5][3];
    Using2DArray( arr );
    return 0;
}
```

arr[5][3]처럼 생성한 배열을 풀어서 생각하면, arr[3]과 같은 배열이 5개 있는 것이다. 그러므로 arr[3] 배열에 대한 포인터를 사용해서 가리킬 수 있다. 정수의 포인터를 사용해서 정수의 배열을 가리킬 수 있었던 것과 같은 원리다.

Exercise 14-1 이름 10번 써 오기

여러분의 이름을 출력하는 함수를 만들어보자. 반복 명령을 사용해서 함수를 10번 호출해보자.

```
void DisplayName()
{
    cout << "이현창\n";
}

int main ()
{
    for ( int i = 0; i < 10; ++i )
        DisplayName();

    return 0;
}
```

Exercise 14-2 또 이름 10번 써 오기

여러분의 이름을 반환하는 함수를 만들어보자. 함수를 호출하고 반환값을 출력하는 작업을 반복 명령을 사용해서 10번 반복해보자.

```
char* GetName()
{
    return "이현창";
}

int main ()
{
    for ( int i = 0; i < 10; ++i )
    {
        char* name = GetName();
        cout << name << endl;
    }

    return 0;
}
```

### Exercise 14-3 중간값 구하기

카드게임 개발팀에서 5개의 정수 중에서 중간 값을 구하는 함수의 작성을 요청해왔다. 아마도 포커 게임에 쓰려는 것 같다. 다음과 같이 호출해서 사용할 수 있는 함수를 작성해보자.

```
int mid = MidValue( 9, 1, 15, 3, 7); // mid에는 7이 들어간다.
```

다음은 MidValue()를 구현한 예다. MidValue() 함수 안에서 값이 큰 순서대로 인자를 배열에 보관하는데, 이와 같이 크기 순서대로 값을 나열하는 작업을 정렬(Sorting)이라고 부른다. 정렬에는 여러 가지 방법이 있는데, 필자가 사용한 것은 거품 정렬(Bubble Sorting)이다. 이 정렬 방법은 알고리즘이 간단한 반면에 수행 효율이 좋지 않은 것으로 유명하다.

여러분이 직접 코드를 분석해서 거품 정렬의 알고리즘을 파악해내는 것도 가능하겠지만, 인터넷이나 알고리즘 서적을 통해서 공부하는 것이 훨씬 능률적일 것이다. 중요한 것은 MidValue() 내에서 어떤 방식을 사용하던지 간에 main() 함수에는 아무런 영향을 미치지 않는다는 점을 잊지 않는 것이다.

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

int MidValue(int v1, int v2, int v3, int v4, int v5)
{
    int sorted[5] = {v1, v2, v3, v4, v5};

    // 제일 큰 순서대로 배열에 담는다.
    for (int i = 0; i < 4; ++i)
    {
        for (int j = 0; j < 4 - i; ++j)
        {
            if ( sorted[j] > sorted[j+1] )
            {
                // sorted[j]와 sorted[j+1]의 값을 맞바꾼다
                int temp = sorted[j];
                sorted[j] = sorted[j + 1];
                sorted[j + 1] = temp;
            }
        }
    }
}
```

```

        // 가운데 있는 값을 반환한다.
        return sorted[2];
    }

int main()
{
    int mid = MidValue(9, 1, 15, 3, 7);

    cout << "mid = " << mid << "\n";

    return 0;
}

```

#### Exercise 14-4 코드 분석하기

다음 코드를 분석해서 최종적으로 변수 a, b, c에 남는 값을 맞춰보자.

```

void Sub( int i, int* p, int& r)
{
    i = 10;
    *p = 20;
    r = 30;
}

int main()
{
    int a = 0, b = 0, c = 0;

    Sub( a, &b, c );
    cout << a << " " << b << " " << c << endl;

    return 0;
}

```

0 20 30

a의 경우만 아무런 영향을 받지 않는다.

#### Exercise 14-5 또 중간값 구하기

카드게임 개발팀에서 중간값 구하는 함수를 개선해달라고 요청해왔다. 기존의 함수도 매우 훌륭했지만 더 많은 개수의 정수를 입력 받을 수 있기를 원한다. 그래서 이번에는 배열을 통해서 정수 값들을 입력 받기로 결정했다. 다음과 같이 호출해서 사용할 수 있는 함수를 만들어보자.

- 규칙 1. 첫 번째 인자로 배열을 넘기고, 두 번째 인자로 배열의 길이를 넘긴다.
- 규칙 2. 입력한 정수가 짝수 개라서 정확하게 가운데 있는 정수가 없는 경우, 보다 작은 수를 중간값으로 반환한다.

```
int arr[10] = {8, 10, 7, 2, 16, 9, 1, 0, 3, 5};
```

```

// mid에는 5가 들어간다.
int mid = MidValue( arr, 10 );

```

```

// mid에는 8이 들어간다.
mid = MidValue( arr, 5 );

```

```

#include <iostream>
using namespace std;

int MidValue(const int arr[], int len)
{
    // 원본 배열을 보존하기 위해서

```

```

// 배열의 복사본을 만들어서 작업한다.
int sorted[100];
for (int i = 0; i < len; ++i)
    sorted[i] = arr[i];

// 제일 큰 순서대로 배열에 담는다.
for (int i = 0; i < len - 1; ++i)
{
    for (int j = 0; j < len - 1 - i; ++j)
    {
        if ( sorted[j] > sorted[j+1] )
        {
            // sorted[j]와 sorted[j+1]의 값을 맞바꾼다
            int temp = sorted[j];
            sorted[j] = sorted[j + 1];
            sorted[j + 1] = temp;
        }
    }
}

// 가운데 있는 값을 반환한다.
return sorted[ (len - 1) / 2];
}

int main()
{
    int arr[10] = {8, 10, 7, 2, 16, 9, 1, 0, 3, 5 };
    int mid = MidValue( arr, 10);
    cout << "mid = " << mid << "\n";

    mid = MidValue( arr, 5);
    cout << "mid = " << mid << "\n";

    return 0;
}

```

#### Exercise 14-6 구조체 전달의 문제점 지적하기

다음 소스 코드를 분석해보자. Func() 함수의 두 가지 문제점을 을 지적하고 개선해보자.

```

struct MainInfo
{
    char name[20];
    int val1;
    int val2;
    double val3;
};

void Func( MainInfo mi )
{
    // mi 의 값을 읽기만 한다.
}

```

문제점 1. 매번 MainInfo 변수를 복사해야 하므로 비효율적이다.  
 문제점 2. MainInfo& 처럼 고친다면, 원본 인자가 훼손될 수 있다.  
 함수의 원형을 아래처럼 고친다.  
 void Func( const MainInfo& mi )

이것만은 알고 갑시다

1. 함수를 호출하기 위해서는 그 함수의 원형을 알고 있어야 한다. 호출할 함수의 원형을 알게 만드는 방법 두 가지를 적어보자.

방법1	자신의 앞에 정의된 함수는 호출할 수 있다.
방법2	자신의 앞에 원형을 선언한 함수는 호출할 수 있다.

2. 다음 괄호 안에 들어갈 알맞은 말을 고르시오.

함수는 (      ㄴ      ) 반환 값을 가질 수 있다.

ㄱ) 두 개의    ㄴ) 하나의    ㄷ) 무한대의

다른 함수에 있는 변수는 접근할 수 (      ㄴ      ).

ㄱ) 있다    ㄴ) 없다

호출하는 함수 안에서 넘겨준 값을 (    A    )이라고/라고 부르고, 호출된 함수 안에서 (    A    )의 값을 대입 받는 변수를 (    B    )이라고/라고 부른다.

ㄱ) A-임자, B-매개상수    ㄴ) A-매개변수, B-인자    ㄷ) A-인자, B-매개변수

3. 포인터 타입의 인자를 사용해서 함수 안쪽의 결과 값을 얻기 위한 단계를 정리해 보았다. 순서대로 번호를 붙여보자.

3	함수 안에서 결과를 넘겨줄 때는 매개 변수가 가리키는 곳에 값을 넣어준다.
1	함수의 매개 변수는 포인터 타입으로 정의한다.
2	인자를 넘겨줄 때는 값을 담고 싶은 변수의 주소를 넘겨준다.

4. 레퍼런스 타입의 인자를 사용해도 함수 안쪽의 결과 값을 얻어올 수 있으며, 포인터를 사용하는 것보다 권장된다. 순서대로 번호를 붙여보자.

1	함수의 매개 변수는 레퍼런스 타입으로 정의한다.
2	인자를 넘겨줄 때는 값을 담고 싶은 변수를 그대로 넘겨준다.
3	함수 안에서 결과를 넘겨줄 때는 매개 변수에 값을 넣어 준다.

5. 배열의 전달은 내부적으로 포인터를 사용한다. 하지만 이 사실에 상관 없이 매개 변수가 배열인 것처럼 사용하면 된다. 순서대로 번호를 붙여보자.

1	인자로 넘겨줄 때는 배열의 이름을 넘겨준다.
3	인자로 넘어온 배열을 사용할 때는 그냥 평범한 배열을 사용하듯이 하면 된다.
2	매개 변수의 타입을 적어줄 때 '배열의 원소 개수' 부분은 적지 않는다.

6. 아래와 같이 UsingArray() 함수에서 배열의 내용을 변경할 수 없도록 함수의 원형을 완성해보자. ( 첫 번째 대괄호만 네모박스로 대체해주세요 )

```
void UsingArray( [const char ] arr[] )
{
    arr[3] = 'A';    // 이 줄에서만 오류가 발생하게 만들자
    char c = arr[2];
}
```

7. 구조체를 인자로 전달할 때는 다음과 같이 하면 좋은데, 각각에 대하여 그 이유를 적어보자.

구조체를 인자로 넘겨줄 때는 레퍼런스를 사용한다.	임시 변수를 생성하고 복사하는 작업을 없앨 수 있어서 효율적이다.
함수의 안쪽에서 구조체의 내용을 읽기만 한다면 const와 레퍼런스를 사용하자.	const를 사용하지 않고 레퍼런스로 만들면 원본 인자의 값을 실수로 훼손할 수 있다.

## 15 장 함수 2

### Vitamin Quiz 오버로딩 vs 디폴트 인자

아래 코드에서 볼 수 있듯이 오버로딩을 써야 할 지 디폴트 인자를 써야 할지 고민되는 경우가 있다. 이런 경우에는 어떤 기준으로 결정할 수 있을지 생각해보자.

```
// 디폴트 인자를 사용한 구현
void Func( int a, int b = 100 );

void Func2()
{
    Func( 50 );
    Func( 50, 50 );
}
```

```
// 오버로딩을 사용한 구현
void Func( int a );
void Func( int a, int b );

// 바로 위의 Func2()와 완전히 똑같다.
void Func2()
{
    Func( 50 );
    Func( 50, 50 );
}
```

생각보다는 명확한 기준을 생각할 수 있다. 인자를 한 개 받는 경우와 두 개 받는 경우에 하는 일이 조금 다르다면 오버로딩을 쓸 수 밖에 없다. 하는 일이 같다면 디폴트 인자를 쓰는 것이 좋은데, 그래야 불필요한 코드의 중복을 막을 수 있다.

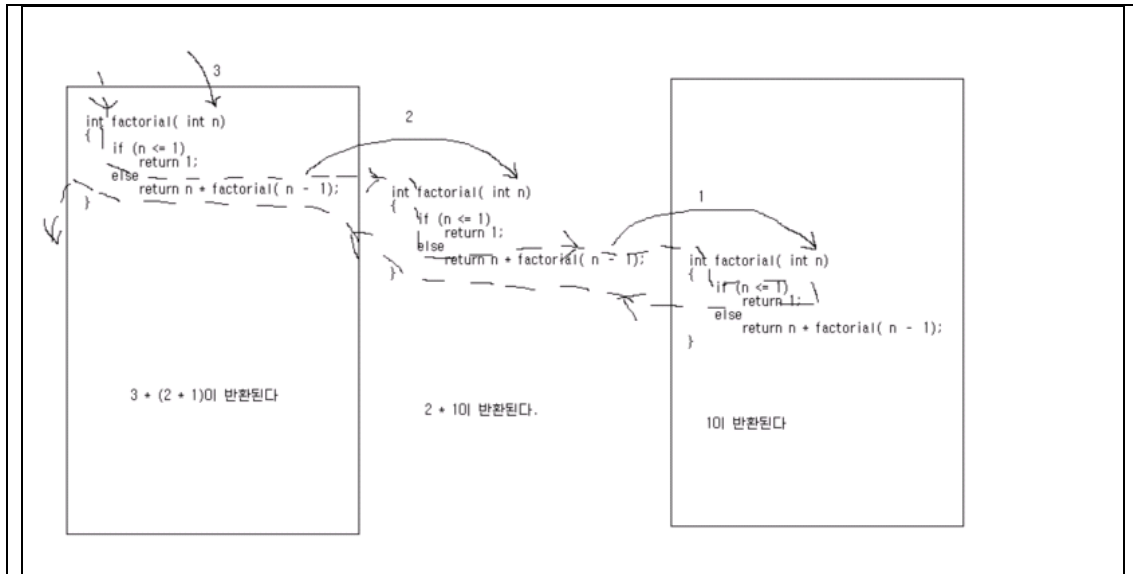
### Exercise 15-1 재귀호출을 이해해보자

다음은 팩토리얼의 수학적 표현이다. 그리고 이어지는 코드는 이 표현을 바탕으로 해서 팩토리얼을 구해주는 함수를 재귀 호출을 사용해서 구현한 것이다. [그림 15-5]와 같이 이 함수를 호출하는 과정을 그림으로 그려보자.

```
factorial(!)의 수학적 표현
n! = 1 (for n=0)
n! = n * (n-1)! (for n>0)

int factorial( int n )
{
    if ( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```





### Exercise 15-2 전체 시나리오 이해하기

'함수 포인터 활용하기' 절의 시작에서 얘기했던 것처럼 함수 포인터의 사용에는 3 명의 주인공이 등장한다. 3명의 주인공 A, B, C는 아래와 같이 정리 두었다. 조금 전의 예제에서 A, B, C에 해당하는 함수를 연결해보자.

<b>A : 함수의 포인터를 넘겨주는 쪽</b> B가 어떤 함수를 호출해야 하는지 알고 있다.	main()
<b>B : 함수의 포인터를 넘겨받는 쪽</b> 언제, 어떤 인자를 사용해서 함수를 호출해야 하는지 알고 있다.	ImportantFunc()
<b>C : 함수의 포인터가 가리키는 함수</b> B가 필요로 하는 일을 어떻게 하는지 알고 있다.	ForWindowsNT()

이것만은 알고 갑시다

1. 각각의 프로그램에서 잘못된 부분을 지적하고 설명해보자.

<code>int Func( char c, float f );</code> <code>float Func( char c, float f );</code>	반환 값의 타입만 다른 함수들은 오버로드 할 수 없다.
<code>void Create( int id, void* parent = 0, int option );</code>	option에 디폴트 값이 없으므로 parent를 디폴트 인자로 만들 수 없다.
<code>void Move( int first, int second = 100 );</code> <code>void Move( int first );</code>	Move( 1 ); 과 같이 호출했을 때 어느 함수를 호출해야 할지 알 수 없다.

2. 오버로드한 함수 중에서 어떤 함수를 호출할지 결정하는 규칙을 정리해보았다. 우선 순위가 높은 것부터 번호를 매겨 보자.

1	정확하게 일치하는 경우(Exact Match)
2	승진에 의한 형변환(Promotion)
3	표준 형변환(Standard Conversions)
4	사용자에 의한 형변환(User-defined Conversions)

3. 다음과 같은 원형을 갖는 함수의 주소를 보관할 수 있는 함수 포인터를 정의해보자. ( 반드시 typedef을 사용해서 타입도 정의하자. )

<code>int MyFunc( int a, char str[] );</code>
<code>typedef int (*FUNC_PTR)(int, char[] );</code>



## 16 장 동적 메모리 할당

Vitamin Quiz 틀린 부분 찾기

다음 소스 코드에서 잘못된 점을 지적해보자.

```
int main()
{
    // 세 개의 float 값을 담을 공간을 할당한다.
    float* p = new float [3];

    // 세 값의 평균을 구한다.
    float ave = (p[0] + p[1] + p[2]) / 3;

    // 결과를 출력한다.
    cout << ave << endl;
    return 0;
}
```

동적으로 할당한 메모리를 해제하지 않았다.  
`delete[] p;` 를 추가해야 한다.

Exercise 16-1 상위권 점수만 출력하자

앞의 예제를 조금 수정해서, int 타입 대신에 float 타입의 점수를 입력 받게 수정해보자.  
또, 평균만 구하는 대신에 평균보다 높은 점수만 출력하게 수정해보자.

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

int main()
{
    // 몇개의실수를입력할지 물어본다.
    int size;
    cout << "몇개의실수를입력하시겠소? ";
    cin >> size;

    // 필요한만큼의메모리를할당한다.
    float* arr = new float [size];

    // 실수를입력받는다.
    cout << "실수를입력하시오.\n";
    for (int i = 0; i < size; ++i)
        cin >> arr[i];

    // 평균을계산하고출력한다.
    float sum = 0;
    for (int i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    float ave = sum / size;
    cout << "합= " << sum << ", 평균= " << ave << "\n";

    // 평균보다높은점수를출력한다.
    for (int i = 0; i < size; ++i)
    {
        if ( arr[i] > ave )
            cout << arr[i] << endl;
    }
}
```

```

    }

    // 사용한메모리를해제한다.
    delete[] arr;

    return 0;
}

```

#### Exercise 16-2 문자열을 이동하기

빙고 게임 개발팀에서 문자열을 이동하는 함수 작성을 요청해왔다. 게임 화면 상단에 전광판이 있는데 여기에 출력하는 문자열이 계속해서 흘러가길 원한다. 아래는 여러분이 만들 함수를 사용하는 코드다. 함수를 한 번 호출할 때마다 문자열이 왼쪽으로 한 칸씩 이동하고, 제일 왼쪽의 문자는 제일 오른쪽으로 이동한다. ShiftLeftString() 함수를 작성해보자.

```

char message[] = "BINGO JJANG!!!";

// copy는 "INGO JJANG!!!B"가 된다.
char* copy = ShiftLeftString( message, 14 );

// copy는 "NGO JJANG!!!BI"가 된다.
char* copy2 = ShiftLeftString( copy, 14 );

#include <iostream>
using namespace std;

char* ShiftLeftString( char* msg, int len )
{
    // 결과 문자열을 보관할 공간을 할당한다.
    char* copy_msg = new char[ len + 1 ];

    // 원본 문자열의 첫번째 글자를 빼고 복사한다.
    for ( int i = 0; i < len - 1; ++i )
    {
        copy_msg[i] = msg[ i + 1 ];
    }

    // 원본 문자열의 첫번째 글자를 제일 뒤에 복사한다.
    copy_msg[ len - 1 ] = msg[ 0 ];

    copy_msg[ len ] = NULL;

    return copy_msg;
}

int main ()
{
    char message[] = "BINGO JJANG!!!";

    char* copy = ShiftLeftString( message, 14 );
    char* copy2 = ShiftLeftString( copy, 14 );

    cout << copy << endl;
    cout << copy2 << endl;

    delete[] copy;
    delete[] copy2;

    return 0;
}

```

이것만은 알고 갑시다

1. 다음 중에서 동적 메모리 할당의 특징을 모두 골라보자.

- ㄱ. 할당하는 메모리의 크기를 동적으로(프로그램의 실행 도중에) 결정할 수 있다.
- ㄴ. 메모리의 할당과 해제 시점을 개발자 마음대로 정할 수 있다.
- ㄷ. 메모리의 할당 시점은 마음대로 정할 수 있지만 해제 시점은 그렇지 않다.
- ㄹ. 할당하는 메모리의 크기를 정적으로(프로그램 작성 중에) 결정할 수 없다.

2. 빈칸에 들어갈 알맞은 말을 고르시오.

new를 사용해서 할당한 메모리는 (    ☐    )을/를 사용해서 해제하고 new[]를 사용해서 할당한 메모리는 (    ☐    )을/를 사용해서 해제해야 한다.

ㄱ) delete, delete    ㄴ) []delete, delete    ㄷ) delete, delete[]    ㄹ) delete[], []delete

3. 다음 빈칸에 공통적으로 들어갈 말을 적어보자. (주관식)

( ☐ ) 포인터를 해제하는 것은 안전하다.

메모리를 해제한 후에는 메모리를 가리키던 포인터의 값을 ( ☐ )로 초기화하자.

## 17 장 문자열

### Vitamin Quiz 함수의 인자 분석하기

strcpy() 함수의 원형은 다음과 같다. 왜 첫 번째 인자는 char\* 타입인데, 두 번째 인자는 const char\* 타입인지 설명해보자. 또한 두 번째 인자에 const 속성이 없는 경우에 어떤 문제점이 발생하는지 설명해보자.

```
char *strcpy( char *strDestination, const char *strSource);
```

첫번째 인자는 strcpy() 함수 안에서 값을 변경할 수 있고, 두 번째 인자는 변경할 수 없다. 다시 말해 첫번째 인자는 읽고 쓰는 용도이며, 두 번째 인자는 읽기 용도이다. 만약에 const 속성을 주지 않는다면 strcpy() 함수의 개발자가 실수로 strSource 가 가리키는 문자열의 내용을 변경하는 경우에 컴퓨터로부터 조언을 받을 수가 없다. 다시 말해서, 오류 메시지가 발생하지 않기 때문에 자신의 실수를 그 즉시 발견할 수가 없다.

### Exercise 17-1 strlen() 직접 구현하기

strlen() 함수와 동일한 기능을 하는 함수를 직접 만들어보자. 아래 strlen() 함수의 원형을 참고해도 좋다.

```
int strlen( const char* str);
int my_strlen(const char* str)
{
    int i = 0;
    while( *(str + i) != NULL)
    {
        ++i;
    }

    return i;
}
```

### Exercise 17-2 욕 판별 함수 작성하기

웃놀이 게임 개발팀에서 채팅 내용이 욕인지 구별하는 함수의 작성을 요청해왔다. 아래는 함수의 원형과 사용이 금지된 욕의 리스트가 있다. 인자로 넘겨진 문자열이 욕 리스트에 있는 문자열과 일치하는 경우 true를 반환하도록 구현해보자.

```
bool IsTermOfAbuse( const char* pChatMessage );
```

바보, 병신, 나쁜 애, 미친 애

```
bool IsTermOfAbuse( const char* pChatMessage )
{
    if ( 0 == strcmp( "바보", pChatMessage ) ||
        0 == strcmp( "병신", pChatMessage ) ||
        0 == strcmp( "나쁜 애", pChatMessage ) ||
        0 == strcmp( "미친 애", pChatMessage ) )
        return true;
    return false;
}
```

### Exercise 17-3 욕 판별 함수 개선하기

웃놀이 게임 개발팀에서 욕 판별 함수의 개선을 요청해왔다. 기존의 함수는 "너 바보야" 라고 말하는 경우에도 제대로 판단을 할 수가 없었다. 이제는 욕 리스트에 있는 문자열과 일치하는 경우 뿐만 아니라, 욕이 포함되어 있는 경우에도 true를 반환하도록 구현해보자. 아래 함수의 원형과 욕의 리스트가 있다.

bool IsTermOfAbuse( string sChatMessage );
바보, 병신, 나쁜, 미친
<pre> bool IsTermOfAbuse( const char* pChatMessage ) {     string s = pChatMessage;     if ( string::npos != s.find("바보")            string::npos != s.find("병신")            string::npos != s.find("나쁜")            string::npos != s.find("미친") )         return true;     return false; } </pre> 참고로 find() 함수는 원하는 문자열을 찾지 못한 경우, string::npos 를 반환한다.

Exercise 17-4 절대 경로의 디렉토리 부분만 출력하기	
<p>틀린그림찾기 개발팀에서 그림 파일들을 관리하는 프로그램을 작성 중인데, 그 때문에 파일의 절대 경로에서 디렉토리 부분만 추출하는 함수가 필요하다고 한다. 사용자로부터 파일의 절대 경로를 입력 받은 후에 디렉토리 부분만 출력하는 프로그램을 작성해보자. 아래 입력과 출력의 예가 있다.</p>	
입력 예	d:\My Library\Temp\test.jpg
출력 예	d:\My Library\Temp\
<pre> #include &lt;iostream&gt; using namespace std;  int main () {     // 절대 경로를 입력받는다.     char path[256];     cin.getline( path, 255 );      // 가장 우측의 역슬래쉬를 찾는다.     int len = strlen( path );     for ( int i = len - 1; i &gt;= 0; --i )     {         if ( '\\ ' == path[i] )         {             // 역슬래쉬의 오른쪽에 널문자를 넣는다.             path[i + 1] = NULL;             break;         }     }      cout &lt;&lt; path &lt;&lt; endl;      return 0; } </pre>	

이것만은 알고 갑시다		
1. 아래는 C 스타일과 C++ 스타일의 문자열 사용법을 비교해 봤다. 빈 칸을 채워보자.		
	C 스타일	C++ 스타일
문자열 보관	char* 타입의 변수	string 클래스
문자열 비교	strcmp()	==, != 연산자
문자열 길이	strlen()	size() 함수
문자열 복사	strcpy()	= 연산자

문자열 결합	strcat()	+ 연산자
--------	----------	-------

2. C 스타일 문자열과 C++ 스타일의 문자열을 변환하는 방법을 정리했다. 빈 칸을 채워 보자.

C 스타일에서 C++ 스타일로 변환	string 클래스에 C 스타일의 문자열을 대입한다.
C++ 스타일에서 C 스타일로 변환	string 클래스의 c_str() 함수를 사용. 이렇게 얻은 문자열은 읽기 전용이다.

3. 아래와 같이 C 스타일과 C++ 스타일의 문자열이 있을 때, 사용자의 입력을 문자열에 넣는 코드를 완성해보자. (>> 연산자를 사용하자 )

```
char cs[20];
string cpps;

[ cin >> cs ]; // 사용자의 입력을 cs 에 넣는다.
[ cin >> cpps ]; // 사용자의 입력을 cpps 에 넣는다.
```

4. >> 연산자를 사용해서 문자열을 입력 받은 경우의 문제점을 정리해봤다. 빈 칸을 채워 보자.

문제점 1	C 문자열의 경우, 준비한 공간보다 문자열이 길면 문제가 생긴다.
문제점 2	공백을 포함한 문자열을 받을 수 없다.

5. getline() 함수를 사용해서 문자열을 입력 받은 경우에는 메모리의 크기를 지정할 수 있으므로 긴 문자열을 입력해도 안전하다. 아래에 3번 문제와 같은 코드가 있다. getline() 함수를 사용해서 코드를 완성해보자.

```
char cs[20];
string cpps;

[ cs.getline( cs, 20 ) ]; // 사용자의 입력을 cs 에 넣는다.
[ getline( cin, cpps ) ]; // 사용자의 입력을 cpps 에 넣는다.
```



## 18 장 헤더 파일과 구현 파일

Vitamin Quiz 같은 이름의 다른 구조체

Exercise 18-2를 풀어봤다면, Point 구조체의 정의가 Example1.cpp와 Example2.cpp에 한 번씩 등장한다는 사실을 알 수 있다. 만약에 아래와 같이, 서로 다른 구현 파일에서 서로 다른 Point 구조체를 정의한다면 결과는 어떻게 될까?

```
// main.cpp
#include <iostream>
using namespace std;

struct Point
{
    int x, y;
};
void Add100( Point& p );

int main()
{
    Point pt = {0,0};
    Add100( pt );

    cout << pt.x << " " << pt.y << endl;
    return 0;
}
```

```
// sub.cpp
struct Point
{
    float x, y;
};

void Add100( Point& p )
{
    p.x += 100;
    p.y += 100;
}
```

두 파일에서 서로 다른 Point 구조체를 알고 있기 때문에, 엉뚱한 결과가 나오게 된다. 공유하는 구조체는 꼭 헤더파일에 두자.

Vitamin Quiz 헤더 파일의 중복 포함을 막는 또 다른 방법

Visual C++은 헤더 파일의 중복 포함을 막는 또 다른 방법을 제공한다. MSDN에서 #pragma once에 대한 정보를 검색해보고 조금 전의 예제를 수정해보자. ( MSDN 사용법은 5부에서 다룬다.)

파일의 앞뒤로 있는 #ifndef, #define, #endif 를 엮고 파일의 제일 앞에 #pragma once를 한 번 써주면 된다. 자세한 설명은 MSDN을 통해 확인하자.

Exercise 18-1 예제 검토

앞의 예제에는 example.cpp, A.cpp, B.cpp 세 개의 구현 파일이 존재한다. 각각의 구현 파일에서 #include 지시문이 해석된 후의 코드를 적어보자. 이 코드를 보면 앞서 정리한 4개의 규칙을 이해하는 데 도움이 될 것이다. 아래는 #include 지시문이 해석된 후의 B.cpp의 코드를 적어두었다.

```
// #include 가 해석된 B.cpp
void B1();
void B2();
```

```
void B1()
{
}
void B2()
{
}
```

```
// #include 가 해석된 A.cpp
void A1();
void A2();
void B1();
void B2();
```

```
void A1()
{
    A2();
}
void A2()
{
    B1();
    B2();
}
```

```
// #include 가 해석된 example.cpp
void A1();
void A2();
void B1();
void B2();
```

```
int main()
{
    A1();
    B1();

    return 0;
}
```

## Exercise 18-2 또 예제 검토

앞의 예제에는 Example1.cpp와 Example2.cpp 두 개의 구현 파일이 있다. 각각의 구현 파일에서 #include 지시문이 해석된 후의 코드를 적어보자.

```
// #include가 해석된 Example1.cpp
struct Point
{
    int x, y;
};
double Distance(const Point& pt1, const Point& pt2);

int main()
{
    // 중간 생략
}

// #include가 해석된 Example2.cpp
struct Point
{
    int x, y;
};
double Distance(const Point& pt1, const Point& pt2);
```

```
double Distance(const Point& pt1, const Point& pt2)
{
    // 중간 생략
}
```

이것만은 알고 갑시다

1. 아래에 a.h와 a.cpp 파일이 있다. #include 지시문이 해석된 후의 a.cpp의 내용을 적어 보자.

// a.h void Func();	// #include 가 해석된 a.cpp void Func();
// a.cpp #include "a.h"	
void Func() { }	void Func() { }

2. 서로 다른 구현 파일간에 함수를 공유하기 위한 처리를 정리했다. 빈칸을 채워보자.

- |   |                                      |
|---|--------------------------------------|
| 1 | 공유될 함수가 있는 구현 파일의 이름을 따서 헤더 파일을 만든다. |
| 2 | 이 헤더 파일에 공유할 함수의 [원형]을 적어준다.         |
| 3 | 공유될 함수를 호출할 구현 파일에서는 이 헤더파일을 포함한다.   |
| 4 | 구현 파일에 자기 자신에 대한 헤더 파일을 포함한다.        |

3. 서로 다른 구현 파일 에 구조체를 공유하기 위한 처리를 정리했다. 빈칸을 채워보자.

- |   |                                  |
|---|----------------------------------|
| 1 | 구조체의 이름을 따서 새로운 헤더 파일을 만든다.      |
| 2 | 이 헤더 파일에 구조체의 [정의 부분]을 위치시킨다.    |
| 3 | 구조체를 사용하는 구현 파일마다 이 헤더 파일을 포함한다. |

4. Point 구조체를 널리 사용하기 위해서 point.h 파일에 정의했다. 필자가 너무 바쁜 나머지 헤더 파일의 중복 포함을 예방하기 위한 처리를 빼먹고 말았다. 다른 사람들이 point.h를 사용하기 전에 얼른 빈칸을 채워서 소스 코드를 완성해보자.

```
[#ifndef POINT_H
#define POINT_H ]

struct Point
{
    int x, y;
}

[#endif ]
```

5. #include 지시문을 사용할 때 <>를 사용하는 경우와 ""를 사용하는 경우를 비교해보자.

- |                |                                      |
|----------------|--------------------------------------|
| #include <xxx> | xxx는 표준 라이브러리의 헤더 파일이 존재하는 폴더에서 찾는다. |
| #include "xxx" | xxx는 구현 파일이 존재하는 폴더에서 찾는다.           |

## 20 장 객체지향 프로그래밍

이것만은 알고 갑시다

1. 다음 빈칸에 들어갈 알맞은 말을 고르시오.

객체지향 프로그래밍은 부품을 조립해서 완제품을 생산하듯이 (  )을/를 조립해서 프로그램을 만든다.

ㄱ) 공인형   ㄴ) 객체   ㄷ) 주체   ㄹ) 컴퓨터

(  )은/는 객체의 세부 구현을 숨김으로써 객체간의 연관성을 줄이는 개념이다.

ㄱ) 상속   ㄴ) 다형성   ㄷ) 정보 은닉   ㄹ) 클래스

(  )은/는 객체의 세부 구현을 캡슐로 감싸서 외부에 공개하지 않는 개념을 말하며, 정보 은닉을 달성하는 방법이다.

ㄱ) 상속   ㄴ) 캡슐화   ㄷ) 객체   ㄹ) 부모 클래스

(  )은/는 기존의 클래스를 기반으로 클래스의 기능을 그대로 물려받는 새 클래스를 만드는 개념이다.

ㄱ) 상속   ㄴ) 객체   ㄷ) 다형성   ㄹ) 공인형

(  )은/는 부모 클래스와 자식 클래스를 동일한 방법으로 다룰 수 있는 능력을 말한다.

ㄱ) 상속   ㄴ) 캡슐화   ㄷ) 정보 은닉   ㄹ) 다형성

2. 객체지향 프로그래밍의 장점을 정리해보았다. 빈칸을 채워보자.

- 다른 객체의 세부 사항에 영향을 받지 않고, 자신이 맡은 객체를 만드는 데만 노력을 집중할 수 (  ).
- 프로그램에 문제가 생기거나, 기능 개선이 이뤄지는 경우 해당 (  )만 교체해주면 된다.
- 만들어진 객체는 다른 프로젝트에 (  )할 수 있다.

## 21 장 클래스와 객체

### Vitamin Quiz 대입 연산자 복습

클래스의 경우에도 아래와 같이 연쇄적으로 대입 연산자를 쓸 수 있는지 확인해보자.

```
pt3 = pt2 = pt1;
```

가능하다.

### Vitamin Quiz 얇은 복사 복습

사실 DynamicArray 클래스의 구현에는 문제가 있다. 아래와 같이 사용하는 경우에 어떤 문제가 생길 수 있는지 지적하고, DynamicArray 클래스를 수정하자.

```
DynamicArray arr1(10);  
DynamicArray arr2 = arr1;
```

기본 제공하는 복사 생성자는 얇은 복사를 하므로, arr2.arr과 arr1.arr 은 동일한 메모리를 가리킨다. 그러므로 arr2와 arr1의 소멸자에서 각각 arr을 해제하면, 이미 해제된 메모리를 또 해제하는 문제가 발생한다. 아래와 같이 깊은 복사를 하도록 수정할 수 있다.

```
class DynamicArray  
{  
public:  
    int* arr;  
    int size;  
  
    DynamicArray(int arraySize);  
    DynamicArray::DynamicArray( const DynamicArray& r );  
    ~DynamicArray();  
};  
  
DynamicArray::DynamicArray(int arraySize)  
{  
    // 동적으로메모리할당한다.  
    arr = new int [arraySize];  
    size = arraySize;  
}  
  
DynamicArray::DynamicArray( const DynamicArray& r )  
{  
    arr = new int [ r.size ];  
    for ( int i = 0; i < r.size; ++i )  
        arr[i] = r.arr[i];  
    size = r.size;  
}  
  
DynamicArray::~~DynamicArray()  
{  
    // 메모리해제한다.  
    delete[] arr;  
    arr = NULL;  
    size = 0;  
}  
  
int main()  
{  
    DynamicArray arr1( 10 );
```

```

        DynamicArray arr2 = arr1;

        return 0;
    }

```

위 코드는 조금 더 개선할 수 있는 부분이 있다.

첫째, 복사 생성자와 대입 연산자는 항상 붙어다녀야 한다. 무슨 말이고 하니, 복사 생성자를 새로 구현해줬다면 대입 연산자도 구현해줘야 한다는 뜻이다. 연산자 오버로딩 부분에서 다루기로 한다.

둘째, 복사 생성자에서 arr의 내용을 복사하기 위해서 반복명령을 사용했는데 memcpy() 함수를 사용하는 것이 더욱 일반적이다. 사용법은 strcpy()와 크게 다르지 않다. MSDN을 통해서 공부해보자.

#### Exercise 21-1 내 생애 첫 번째 멤버 함수

조금 전의 예제에 새로운 멤버 함수를 추가해보자. 아래와 같은 원형으로, 이 함수를 호출하면 원점 (0, 0)에서 점 (x, y) 까지의 거리를 반환해야 한다.

```

float DistanceFromOrigin();

#include <iostream>
#include <math.h>
using namespace std;

class POINT
{
public:
    int x, y;

    void Print()
    {
        cout << "( " << x << ", " << y << ")\n";
    }

    float DistanceFromOrigin()
    {
        return sqrt( float(x*x) + float(y*y) );
    }
};

int main()
{
    POINT p = { 1, 1};
    cout << p.DistanceFromOrigin() << endl;

    return 0;
}

```

#### Exercise 21-2 당구공 클래스

당구 게임 개발팀에서 당구공 클래스의 제작을 요청했다. 아래 보이는 것 같이 이미 어느 정도 개발이 진행된 상태이다. 문제는 복사 생성자인데, 두 개의 당구공이 한 위치에 존재할 수는 없기 때문이다. 그래서 새로 생성하는 당구공은 이전 당구공의 바로 오른쪽에 위치해야 한다. 당구 게임 개발팀을 위해서 복사 생성자를 구현해보자.

```

class Ball
{
public:

```

```

int _x, _y;    // 공의 위치
int _radius;   // 공의 반지름

Ball( int x, int y );    // 기본 생성자
Ball( const Ball& b );   // 복사 생성자
// 이하 생략
};

Ball::Ball( const Ball& b)
{
    _radius = b._radius;
    _x = b._x + _radius * 2;
    _y = b._y;
}

```

#### Exercise 21-3 당구공 클래스에 접근 권한 설정

이전 Exercise에서 만들었던 당구공 클래스(Ball)에서 외부에서 접근을 막아야 하는 멤버와 접근을 허용해야 하는 멤버를 골라보자. 그리고 알맞은 접근 권한을 설정하자.

```

class Ball
{
public:
    Ball( int x, int y );
    Ball( const Ball& b );
private:
    int _x, _y;
    int _radius;
    // 이하 생략
};

```

#### Exercise 22-4 당구공 클래스에 접근자 추가

당구 개발팀에서 당구공 클래스의 수정을 요청해왔다. 당구대의 크기가 800x400 인데 당구공의 위치가 (800,400) 밖으로 나가는 경우가 있다고 한다. 당구공 클래스(Ball)에 접근자를 추가해서 \_x, \_y 값이 (800,400)을 넘어설 수 없도록 수정해보자.

```

Ball::SetX( int x )
{
    if ( x < 0 )
        _x = 0;
    if ( x > 800 )
        _x = 800;
}
Ball::SetY( int y )
{
    if ( y < 0 )
        _y = 0;
    if ( y > 400 )
        _y = 400;
}

```

#### Exercise 22-5 당구공 클래스 최적화

게임에 사용하는 당구공의 크기는 모두 같기 때문에, 굳이 당구공 객체마다 반지름 값을 가지고 있을 필요가 없다. \_radius 를 정적 멤버 변수로 고쳐보자.

```

class Ball
{
public:

```

```

    Ball( int x, int y );
    Ball( const Ball& b );
private:
    int _x, _y;
    static int _radius;
    // 이하 생략
};

int Ball::radius = 10;

```

#### Exercise 22-6 함수 포인터 연습

멤버 함수에 대한 포인터를 사용해서 Point 클래스의 Print() 함수를 호출해보자.

```

Point p = {100, 100};

typedef void (Point::*FUNC_PTR) ();
FUNC_PTR pfn = &Point::Print;
(p.*pfn) ();

```

#### Exercise 22-7 당구공 4개 생성하기

당구 게임 개발팀에서 객체를 배열에 넣어서 보관할 수 있으면, 왜 처음부터 안 가르쳐줬냐고 난리다. 나란히 놓여져 있는 당구공 객체 4개를 배열에 보관하는 코드를 작성해서 당구 게임 개발팀에 알려주자.

```

Ball arr[] = { Ball(0,0), Ball(20,0), Ball(40,0), Ball(60,0) };

```

이것만은 알고 갑시다

1. 일반 함수와 멤버 함수의 차이점을 정리해봤다. 빈칸을 채워보자.

- 멤버 함수를 호출하기 위해서는 ( **객체** )을/를 명시해주어야 한다.
- 멤버 함수 안에서는 객체의 이름을 명시하지 않고 ( **멤버** )에 접근할 수 있다.
- 멤버 함수 안에서는 외부에서 접근이 거부된 멤버에도 접근할 수 ( **있다** ).

2. 아래와 같은 결과가 출력될 수 있도록 기본 생성자, 복사생성자, 소멸자를 구현해보자.

```

class Dummy
{
public:
    [Dummy()
    {
        cout << "생성자\n";
    }
    Dummy( const Dummy& )
    {
        cout << "복사생성자\n";
    }
    ~Dummy()
    {
        cout << "소멸자\n";
    }
];

int main()
{
    cout << "d1 생성" << endl;
    Dummy d1;
    cout << "d2 생성" << endl;
    Dummy d2 = d1;
    cout << "끝" << endl;
}

```



```
return 0;
}
```

[실행결과]  
d1 생성  
생성자  
d2 생성  
복사생성자  
끝  
소멸자  
소멸자

3. 아래 프로그램은 생성자의 구현에 문제가 있다. 문제점을 지적하고 수정해보자.

```
#include "Room.h"
class Lobby
{
private:
    const int maxUser;
    Room rooms[10];
    Room& specialRoom;
public:
    Lobby()
    {
        maxUser = 100;
        specialRoom = rooms[0];
    }
};
```

레퍼런스 변수나 const 변수는 생성자의 초기화 리스트를 통해서 초기화해야 한다.

```
Lobby()
: maxUser(100), specialRoom( rooms[0] )
{
}
```

4. 접근 제어 키워드를 나열했다. 외부에서의 접근을 허용하는 키워드를 골라보자.

ㄱ. public    ㄴ. protected    ㄷ. private

5. 클래스를 만들 때 따라야 할 가이드라인을 정리해봤다. 빈 칸을 채워보자.

- 모든 경우를 감시해야 한다. 접근자를 잘 만들어 두었어도 생성자를 간과했다면 헛일이 되어 버린다.
- 모든 경우에 동일한 코드를 사용하게 만들어야 한다. ( 코드의 중복 )을 줄일 수록 문제가 줄어든다.

6. 다음 빈칸에 들어갈 알맞은 말을 고르시오.

정적 멤버는 ( ㄱ )의 소유며 모든 객체들은 같은 정적 멤버를 공유한다.

ㄱ) 클래스    ㄴ) 객체    ㄷ) 멤버    ㄹ) 생성자

클래스의 정의 안에서 정의한 함수는 자동적으로 ( ㄴ )이/가 된다.

ㄱ) 정적 멤버 함수    ㄴ) 인라인 함수    ㄷ) const 멤버 함수    ㄹ) 자동 함수

멤버 함수 안에서 ( ㄹ ) 포인터는 멤버 함수를 소유한 객체를 가리킨다.

ㄱ) NULL    ㄴ) 많은    ㄷ) 몇 몇    ㄹ) this

7. 클래스를 여러 파일에 나누어 담으면 다음과 같은 장점이 생긴다. 빈 칸을 채워보자.

- 소스 코드가 간결해져서 읽기에 편하다.
- 관련된 내용끼리 모여있으므로 필요한 부분을 찾아보기 쉽다.
- 소스 코드의 관리와 ( 재사용 )이/가 편하다.

8. 인라인 함수를 사용할 때에 따라야 할 가이드라인을 정리해봤다. 빈 칸을 채워보자.

- 함수의 내용이 몇 줄 정도로 아주 ( 짧은 ) 경우에만 인라인 함수로 만들자.
- 인라인 함수는 반드시 ( 헤더 ) 파일에 있어야 한다.

9. 멤버 함수를 const로 만드는 것의 의미를 정리했다. 빈 칸을 채워보자.

- 다른 개발자가 '아, 이함수는 멤버 변수의 값을 변경하지 않는구나' 라고 생각하게 만든다.
- 실수로 ( 멤버변수 )의 값을 바꾸려고 하면 컴퓨터가 오류 메시지를 통해서 알려준다.
- ( const ) 객체를 사용해서 이 함수를 호출할 수 있다.

10. 객체를 동적으로 할당하는 경우에 생성자와 소멸자의 호출 시점을 정리해보자.

생성자	new 연산자로 할당할 때
소멸자	delete 연산자로 해제할 때

11. 여기는 21장 필드테스트 문제 1번을 옮겨주세요. 타이핑 하기에 내용이 너무 많네요.

소스폴더를 참조(21장의 Exer1 폴더)

## 22 장 상속과 포함

### Vitamin Quiz 버그 수정

Rect 클래스에는 문제점이 있다. 예를 들어 `_topLeft`의 값이 (10, 10)이고 `_bottomRight`의 값이 (0, 0)이라면, `GetWidth()` 함수와 `GetHeight()` 함수는 각각 -10을 반환한다. 이런 경우에도 양수의 값을 반환하게 수정하자.

```
void Rect::GetWidth() const
{
    int result = ( _bottomRight.GetX() - _topLeft.GetX() + 1 );
    return ( result > 0 ? result : -result );
}
```

`GetHeight()` 도 같은 방식으로 구현할 수 있다. 이 외에 부호에 상관없이 절대값을 취하는 `abs()` 함수를 조사해 보는 것도 좋다.

### Vitamin Quiz 객체의 크기 확인하기

자식 클래스가 부모 클래스의 모든 멤버를 소유한다면 당연히 객체의 크기도 항상 커야 한다. `sizeof()` 연산자를 사용해서 `HTMLWriter` 클래스와 `DocWriter` 클래스의 크기를 비교해보자.

```
cout << sizeof( DocWriter ) << " " << sizeof( HTMLWriter ) << endl;
```

### Exercise 22-1 생성자와 소멸자의 순서

생성자와 소멸자의 순서에 대해서 필자가 한 말이 사실인지 확인하려면 어떻게 해야 할까? Rect 예제를 수정해서 호출 순서를 확인해보자.

지금까지 해왔던 것처럼 생성자와 소멸자에서 적절한 문자열을 출력하도록 하면 된다.

### Exercise 22-2 생성자와 소멸자의 순서

`HTMLWriter` 클래스와 `DocWriter` 클래스의 코드를 수정해서 생성자와 소멸자의 호출 순서를 확인해보자.

생성자와 소멸자에서 적절한 문자열을 출력하도록 수정한다.

### Exercise 22-3 멤버 함수도 확인하자

멤버 함수도 앞에서 배운대로 `private`, `protected`, `public`의 접근 제어 키워드에 영향을 받는지 확인해보자. `Parent` 클래스에 멤버 함수를 추가해서 확인해보자.

`Parent` 클래스에 각 접근 권한별로 멤버 함수를 추가해서 확인하면 된다. 멤버 변수와 똑같이 동작하는 것을 알 수 있다.

이것만은 알고 갑시다

1. 다음 빈칸에 들어갈 알맞은 말을 고르시오.

A 클래스가 B 클래스의 객체를 멤버 변수로 가지고 있다면 A 클래스는 B 클래스를 ( ☐ )한다고 말한다.

ㄱ) 사랑   ㄴ) 좋아   ㄷ) 포함   ㄹ) 상속

자식 클래스는 부모 클래스의 ( ☐ ) 멤버를 상속 받는다.

ㄱ) 플레티넘   ㄴ) 일부   ㄷ) 모든   ㄹ) 골드

두 개 이상의 부모 클래스를 동시에 상속 받는 경우를 ( ☐ )(이)라고 하며 모든

부모의 멤버를 상속 받는다.

ㄱ) 대단하다 ㄴ) 포함 ㄷ) 능력 있다 ㄹ) 다중 상속

2. 아래와 같은 상황에서는 초기화 리스트를 사용해야만 한다. 네모 칸 안에 triangle 클래스의 생성자를 구현해보자.

```
class point
{
public:
    point( int x, int y ); // point 클래스의 유일한 생성자
    // 중간 생략. 본문에 나오는 Point 클래스와 비슷하나 다른 클래스
};

class shape
{
public:
    shape( int color ); // shape 클래스의 유일한 생성자
    // 중간 생략
};

class triangle : public shape
{
public:
    triangle( point p1, point p2, point p3, int color )
    [ : pt1( p1 ), pt2( p2), pt3( p3 ), shape( color )
      {
      }
    ]

private:
    point pt1, pt2, pt3;
    // 중간 생략
};
```

3. 포함과 상속의 경우에 생성자와 소멸자의 호출 순서를 정리해보았다. 빈 칸을 완성해보자.

A 클래스가 B클래스를 포함하는 경우	B의 생성자 → A의 생성자 → A의 소멸자 → B의 소멸자
A 클래스가 B클래스를 상속 받은 경우	B의 생성자 → A의 생성자 → A의 소멸자 → B의 소멸자

4. Derived 클래스가 Base 클래스를 상속 받았다는 가정 하에 아래 표를 채워보자.

Base 객체를 Derived 객체에 대입	실패
Derived 객체를 Base 객체에 대입	성공
Base* 타입을 Derived* 타입으로 형변환	실패
Base& 타입을 Derived& 타입으로 형변환	실패
Derived* 타입을 Base* 타입으로 형변환	성공
Derived& 타입을 Base& 타입으로 형변환	성공

5. 각각의 접근 제어 키워드가 어떤 경우에 접근을 허락하는지 정리했다. 표를 채워보자.

[접근이 이루어질 수 있는 위치]

ㄱ. 자기 자신 ㄴ. 자식 클래스 ㄷ. 전혀 상관 없는 곳

public 멤버에 접근을 허용하는 위치	ㄱ, ㄴ, ㄷ
protected 멤버에 접근을 허용하는 위치	ㄱ, ㄴ
private 멤버에 접근을 허용하는 위치	ㄱ

## 23 장 다형성과 가상 함수

본문 수정

변경 전: 바로 위에서 배운 규칙이 동일하게 적용되기 때문이다.

변경 후: 바로 위에서 배운 규칙이 비슷하게 적용되기 때문이다.

본문 수정

변경 전: 그런데 우리가 생성자를 하나 만든 순간에 컴퓨터가 제공해주는 생성자들을 모두 사용할 수 없게 되어버린 것이다.

변경 후: 그런데 우리가 생성자를 하나 만든 순간에 컴퓨터가 제공해주는 디폴트 생성자를 사용할 수 없게 되어 버린 것이다.

Vitamin Quiz 다형성 복습

이제 DocWriter와 HTMLWriter 클래스가 완벽한 다형성을 갖게 되었다. 부품의 교체 차원에서 DocWriter와 HTMLWriter 클래스가 갖는 다형성을 설명해보자.

DocWriter와 HTMLWriter는 동일한 인터페이스를 가지고 있기 때문에, DocWriter나 HTMLWriter를 사용하는 코드는 동일해진다. 그래서 DocWriter를 사용하는 코드는 수정없이 HTMLWriter를 사용할 수 있다.

Exercise 23-1 다양한 도형 추가

Shape 클래스의 자식 클래스로 삼각형, 타원, 선 클래스를 만들어보자. 또한 다형성을 사용해서 이 클래스들의 객체를 하나의 배열에 담아서 사용하는 예제를 만들어보자.

삼각형 클래스를 추가해보았다. 타원과 선 클래스는 여러분이 직접 해보기 바란다

```
// 삼각형
class Triangle : public Shape
{
public:
    Triangle(int x, int y, int x1, int y1, int x2, int y2);
    virtual void Draw() const;

protected:
    int _x1, _x2, _y1, _y2;
};

Triangle::Triangle(int x, int y, int x1, int y1, int x2, int y2)
: Shape( x, y)
{
    _x1 = x1;
    _y1 = y1;
    _x2 = x2;
    _y2 = y2;
}

void Triangle::Draw() const
{
    cout << "[Triangle] Position = ( " << _x << ", " << _y << " )
    "
        "vertex1 = ( " << _x1 << ", " << _y1<< " ) "
        "vertex2 = ( " << _x2 << ", " << _y2<< " )\n";
}
```

```

int main()
{

    // 도형들을 담을 배열을 준비한다
    Shape* shapes[3] = {NULL};

    // 각 타입의 객체를 생성해서 배열에 보관한다.
    shapes[0] = new Circle( 100, 100, 50);
    shapes[1] = new Rectangle( 300, 300, 100, 100);
    shapes[2] = new Triangle( 200, 100, 50, 50, 150, 150);

    // 배열의 보관된 모든 객체를 그린다.
    for (int i = 0; i < 3; ++i)
        shapes[i]->Draw();

    // 배열의 보관된 모든 객체를 소멸시킨다.
    for (i = 0; i < 3; ++i)
    {
        delete shapes[i];
        shapes[i] = NULL;
    }

    return 0;
}

```

이것만은 알고 갑시다

1. 다음 빈칸에 들어갈 알맞은 말을 보기에서 고르시오.

(    **ㄷ**    )을/를 사용하면 부모 클래스의 포인터를 사용해서 자식 객체를 가리키고 있는 경우에도 알맞은 함수를 호출한다.

ㄱ) 생성자    ㄴ) 레퍼런스    ㄷ) 가상 함수    ㄹ) 인라인 함수

클래스에 하나 이상의 가상 함수가 있는 경우에는 (    **ㄴ**    )도 반드시 가상 함수로 만들어야 한다.

ㄱ) 생성자    ㄴ) 소멸자    ㄷ) 접근자    ㄹ) 합격자

(    **ㄱ**    )은/는 타입에 관계 없이 동일한 방법으로 다룰 수 있는 능력을 말한다.

ㄱ) 다형성    ㄴ) 상속    ㄷ) 포함    ㄹ) 오버라이딩

(    **ㄱ**    )는 함수의 원형만 존재하는 가상 함수로써 외부와 약속된 부분(인터페이스)로서의 역할만 하는 함수다. 자식 클래스는 이 함수를 반드시 오버라이드 해야 한다.

ㄱ) 순수 가상 함수    ㄴ) 불순 가상 함수    ㄷ) 인라인 함수    ㄹ) 소멸자

하나 이상의 순수 가상 함수를 가진 클래스를 (    **ㄷ**    )(이)라고 부르며 이 클래스의 객체를 생성하는 것은 불가능하다.

ㄱ) 순수 클래스    ㄴ) 불순 클래스    ㄷ) 추상 클래스    ㄹ) 구체 클래스

2. 가상 함수를 만드는 가이드라인을 정리했다. 빈 칸을 채워보자.

- 보통의 경우는 일반적인 멤버 함수로 만든다.
- **다형성**을 이용하려 한다면 멤버 함수를 ( **가상함수** )로 만든다.
- 외부와 약속된 부분(인터페이스)으로서의 역할만 갖게 하려면 ( **순수가상함수** )(으)로 만든다.

3. 아래 코드에서 잘못된 부분을 지적하고 설명해보자.

```

class A
{
public:
    void Func( int a ) {}
    void Func( int a, int b) {}
};

class B : public A
{
public:
    void Func( int c ) {}
};

int main()
{
    B b;
    b.Func( 10, 20 );
    return 0;
};

```

클래스 B에서 Func()을 오버라이드 하면, 클래스 A에 있는 모든 Func() 함수를 사용할 수 없게 된다. 아래와 같이 클래스 A의 Func() 함수를 호출하고 싶다고 명시적으로 적어줘야 한다.

```

b.A::Func( 10, 20 );

```

4. 아래 코드에서 잘못된 부분을 지적하고 설명해보자.

```

class A {};

class B
{
public:
    B(int i) {}
};

int main()
{
    A a;
    B b;
    return 0;
}

```

클래스 B에서 생성자를 정의하면, 컴퓨터가 기본으로 제공하는 디폴트 생성자를 사용할 수 없다. 디폴트 생성자도 직접 정의해주어야 한다.

## 24 장 예외 처리

### Vitamin Quiz 먼저 생각해보기

잠깐만 읽기를 멈추고 먼저 생각해보자. 반환 값을 사용한 예외 처리는 어떤 문제점을 가지고 있을까? 다음 절에서는 2 가지 문제점을 제시한다.

- 문제점 1. 본연의 소스 코드와 예외 처리 코드가 뒤엉켜서 지저분하고 읽기 어렵다
- 문제점 2. 예외 처리 때문에 반환 값을 본래의 용도로 사용할 수 없다.

### Vitamin Quiz 먼저 생각해보기

잠깐만 읽기를 멈추고 먼저 생각해보자. 조금 전의 예제처럼 생성자에서 예외가 발생한 경우는 어떻게 대비할 수 있을까? 여러분이 직접 예외 처리를 해보고 필자의 구현과 비교해보자.

본문에서처럼 예외를 처리하고 다시 던질 수 있다.

### Exercise 24-1 예외 처리 연습

아래에 나눗셈을 수행하는 코드가 있다. 인자 b의 값으로 0을 입력되는 경우에 "0으로 나누기" 라는 문자열 리터럴을 예외로 던지게 수정하자.

```
float Divide( int a, int b )
{
    return (float)a / (float)b;
}

float Divide( int a, int b )
{
    if ( 0 == b )
        throw "0으로 나누기";
    return (float)a / (float)b;
}
```

### Exercise 24-2 함수 실행과정 확인해보기

조금 전의 예제에서 실행의 흐름이 어떻게 이동하는지 확인해보자. 함수 A(), B(), C()의 시작과 끝에 문자열을 출력해보면 어디는 실행하고 어디는 건너뛰는지 알 수 있다. 예를 들어 A() 함수는 아래와 같이 고칠 수 있다.

```
void A()
{
    cout << "A() 시작" << endl;
    B();
    cout << "A() 끝" << endl;
}
```

다른 함수도 고치고 결과를 확인해보자.

### Exercise 24-3 auto\_ptr 적용

auto\_ptr을 사용해서 [예제 24-15]을 수정해보자.

이 문제는 함정이 있었다. auto\_ptr을 사용하도록 수정하는 것은 불가능하다. auto\_ptr의 소멸자에서는 delete()가 아닌 delete 연산자를 사용하여 메모리를 해제하므로, 예제와 같이 new()로 할당한 메모리를 대입하면 올바르게 해제되지 않는다.

이것만은 알고 갑시다

1. 반환 값을 사용한 예외 처리의 문제점을 정리했다. 빈 칸을 채워보자.



본연의 소스 코드와 예외 처리 코드가 뒤엉켜서 지저분하고 읽기 어렵다.

예외 처리 때문에 반환 값을 본래의 용도로 사용할 수 없다.

2. 다음 중 맞는 말을 모두 고르시오.

- ㄱ. 예외가 던져지면 실행의 흐름이 catch 블록으로 이동한다.
- ㄴ. 예외는 함수를 여러 개 건너서도 전달할 수 있다.
- ㄷ. 하나의 try 블록에 여러 개의 catch 블록이 따라올 수 있다.
- ㄹ. 예외 객체는 레퍼런스로 받는 것이 좋다.
- ㅁ. 뇌를 자극하는 C++은 참 좋은 책이다.

3. 다음 빈칸에 들어갈 알맞은 말을 보기에서 고르시오.

예외가 발생했을 때 리소스 릭(Resource Leak)이 발생하는 것을 막기 위해서 (    ㄷ    ) 포인터를 사용할 수 있다.

- ㄱ) NULL    ㄴ) void    ㄷ) 스마트    ㄹ) 함수

생성자에서 예외가 발생한 경우에는 (    ㄱ    )이/가 호출되지 않는다.

- ㄱ) 소멸자    ㄴ) 복사 생성자    ㄷ) 부모의 생성자    ㄹ) 가상 함수

소멸자의 밖으로 예외가 던져지는 것은 (    ㄷ    ) 막아야 한다.

- ㄱ) 가능하면    ㄴ) 경우에 따라    ㄷ) 반드시    ㄹ) 대충

new, new[] 연산자를 사용해서 메모리를 할당할 때 컴퓨터에 메모리가 부족하다면 (    ㄴ    ) 예외가 발생할 것이다. 따라서 반드시 이 예외를 처리해줘야 한다.

- ㄱ) 기분 나쁜    ㄴ) bad\_alloc    ㄷ) bad\_cast    ㄹ) out\_of\_memory

4. 여기는 p.727의 필드테스트 1번 문제를 옮겨주세요. 타이핑 하기에 너무 많네요.

소스 폴더 참조(24장의 Exer 프로젝트)

## 26 장 접근 범위와 존속 기간

Vitamin Quiz 멤버 함수 안에 정의한 변수는?

아래 코드를 보면 클래스의 멤버 함수 안에서 static으로 지역 변수를 정의했다. 프로그램의 결과를 예상해보자. 문제를 푸는 열쇠는 과연 이 static으로 정의한 지역 변수가 객체마다 하나씩 존재할 것인지, 아니면 한 변수가 모든 객체에 의해서 사용될지를 파악하는데 있다.

```
class Test
{
public:
    void Func()
    {
        static int a = 0;
        cout << "a = " << ++a << "\n";
    }
};

int main()
{
    Test t1, t2, t3;

    t1.Func();
    t2.Func();
    t3.Func();
    return 0;
}
```

실행시켜보면 알 수 있겠지만 변수 a의 값은 객체와는 상관없이 지속적으로 증가하게 된다. 객체마다 별도로 관리할 것이라고 생각할 수도 있지만 실제로는 그렇지 않다는 점을 꼭 기억해두자.

### Exercise 26-1 존속 기간 확인

생성자와 소멸자를 사용하면 지역 변수와 전역 변수의 존속 기간을 확인해볼 수 있다. 앞의 예제에서 변수 대신에 객체를 사용하고 객체의 생성자와 소멸자에서 적절한 문자열을 출력해서 지역 변수와 전역 변수의 존속 기간을 확인해보자.

생성자와 소멸자만 가진 간단한 클래스를 만들어서 확인한다. 여태까지 계속 해왔던 방식이라 별도의 소스 코드는 생략한다.

### Exercise 26-2 또 존속 기간 확인

생성자와 소멸자를 사용해서 static으로 정의한 지역 변수와 전역 변수의 존속 기간을 확인해보자.

생성자와 소멸자만 가진 간단한 클래스를 만들어서 확인한다. 여태까지 계속 해왔던 방식이라 별도의 소스 코드는 생략한다.

이것만은 알고 갑시다

1. 각 변수의 종류에 따라 접근 범위와 존속 기간을 정리했다. 빈칸을 채우자.

변수의 종류	접근 범위	존속 기간
지역 변수	변수를 정의한 함수 안쪽	변수를 정의한 함수가 끝날 때 소멸
전역 변수	프로그램 전체	프로그램이 종료할 때 소멸
블록 안에서 정의한 변수	블록 안쪽	블록이 끝날 때 소멸
static으로 정의한 지역 변수	변수를 정의한 함수 안쪽	프로그램이 종료할 때 소멸

static으로 정의한 전역 변수	변수를 정의한 파일 안쪽	프로그램이 종료할 때 소멸
for 명령의 괄호 안에서 정의한 변수	for 명령의 중괄호 안에서만 접근할 수 있는 것이 규칙이지만 잘 지켜지지 않고 있다.	for 명령이 끝날 때 소멸하는 것이 규칙이지만 잘 지켜지지 않고 있다.

2. 다음 빈칸에 들어갈 알맞은 말을 보기에서 고르시오.

다른 파일에서 정의한 전역 변수나 함수를 사용하기 위해서는 (   ㄴ  ) 키워드를 사용해서 다시 한 번 선언해준다. 함수의 경우에는 이 키워드를 생략 가능하다.

ㄱ) auto   ㄴ) extern   ㄷ) intern   ㄹ) global

레지스터 변수는 메모리가 아닌 (   ㄱ  )에 값을 보관한다.

ㄱ) 레지스터   ㄴ) 하드디스크   ㄷ) USB 메모리   ㄹ) 그래픽 메모리

C 언어로 작성한 함수를 사용하기 위해서는 (   ㄹ  ) 키워드를 사용해서 함수의 원형을 선언해준다.

ㄱ) intern "C"   ㄴ) external "C"   ㄷ) language "C"   ㄹ) extern "C"

3. 다음 중 옳은 것을 모두 고르시오.

- ㄱ. 변수를 정의한 위치가 다르다면 같은 이름을 가질 수 있다. 이때 새로 정의한 변수가 이전 변수를 숨긴다.
- ㄴ. static으로 정의한 전역 변수는 다른 파일에서 extern 키워드를 사용해야 접근할 수 있다.
- ㄷ. static으로 정의한 함수를 사용하기 위해서는 extern 키워드를 사용하더라도 접근할 수 없다.

## 27 장 타입 2

Vitamin Quiz 레퍼런스의 경우는 왜 예외를 던질까?

레퍼런스 변수의 경우에는 dynamic\_cast에 실패했을때 왜 bad\_cast 예외를 던질 수 밖에 없는지 조금 더 고민해보자.

레퍼런스 변수는 반드시 초기화되어야 하기 때문이다. 형변환에 실패한 경우 아무것도 아닌 값을 가져야 하는데, 레퍼런스 변수는 반드시 다른 변수를 사용해서 초기화해야 한다.

### Exercise 27-1 복소수의 뺄셈

Complex 클래스에 뺄셈 연산자를 구현하자.

아래와 같은 멤버 함수를 추가할 수 있다.

```
Complex operator-(const Complex& right)
{
    // 실수부와허수부를각각뺀다.
    int real = this->real - right.real;
    int imag = this->imaginary - right.imaginary;

    // 결과를보관한복소수객체를반환한다.
    return Complex(real, imag);
}
```

### Exercise 27-2 -- 연산자

Complex 클래스에 전치와 후치 -- 연산자를 구현하자.

```
// --c 의경우 (전치연산)
Complex operator--()
{
    // 실수부의값을먼저뺀다.
    this->real--;

    // 현재값을반환한다.
    return *this;
}

// c--의경우 (후치연산)
Complex operator--(int)
{
    // 현재값을먼저보관한다.
    Complex prev( this->real, this->imaginary );

    // 실수부의값을뺀다.
    this->real--;

    // 보관한값을반환한다.
    return prev;
}
```

### Exercise 27-3 일반함수로 구현한 복소수의 뺄셈

일반함수를 사용해서 Complex 클래스의 뺄셈 연산자를 구현하자.

```
Complex operator+(const Complex& left, const Complex& right)
{
    // 실수부와 허수부를 각각 뺀다.
```

```

int real = left.real - right.real;
int imag = left.imaginary - right.imaginary;

// 결과를 보관한 복소수 객체를 반환한다.
return Complex(real, imag);
}

```

이 함수를 클래스의 friend 함수로 지정하는 것도 잊지 말자.

이것만은 알고 갑시다

1. 전치형 ++ 연산자와 후치형 ++ 연산자를 오버로딩 하기 위한 함수의 원형을 정리했다. 빈 칸을 채워보자.

전치형 ++	operator++()
후치형 ++	operator++(int)

2. 다음 빈 칸에 들어갈 알맞은 말을 보기에서 고르시오.

일반 함수를 사용해서 연산자를 오버로딩 했을 때 클래스의 private, protected 멤버에 접근할 필요가 있다면 해당 함수를 (          )(으)로 설정한다.

ㄱ) virtual    ㄴ) const    ㄷ) inline    ㄹ) friend

const\_cast는 (          )나 volatile 속성을 제거하는 데 사용한다.

ㄱ) const    ㄴ) virtual    ㄷ) variable    ㄹ) 레퍼런스

(          )는 일반적으로 허용하지 않는 위험한 형변환을 할 때 사용한다.

ㄱ) const\_cast    ㄴ) static\_cast    ㄷ) dynamic\_cast    ㄹ) reinterpret\_cast

static\_cast는 평범한 형변환을 할 때 사용하는데 구체적으로 다음과 같은 규칙을 갖는다.

- 만약에 A 타입에서 B 타입으로 (          ) 형변환이 가능하다면 static\_cast를 사용해서 B 타입에서 A 타입으로 형변환할 수 있다.

ㄱ) 명시적    ㄴ) 암시적    ㄷ) 위험한    ㄹ) const\_cast

dynamic\_cast는 (          ) 관계에 있는 클래스들의 포인터나 레퍼런스간의 형변환에 사용한다.

ㄱ) 불륜    ㄴ) 친한    ㄷ) 포함    ㄹ) 상속

dynamic\_cast를 사용한 형변환이 올바르지 못한 경우 NULL 포인터를 반환하거나 (          ) 예외를 던진다.

ㄱ) 알 수 없는    ㄴ) 마구    ㄷ) 엄청난    ㄹ) bad\_cast

dynamic\_cast를 사용하기 위해서는 (          ) 옵션을 켜주어야 한다.

ㄱ) RTTI    ㄴ) RTTI    ㄷ) RRTI    ㄹ) RTI

생성자 앞에 (          ) 키워드를 붙이면 생성자를 사용한 암시적인 형변환이 불가능해진다.

ㄱ) extern    ㄴ) operator    ㄷ) void    ㄹ) explicit

3. 다음은 간략한 Integer 클래스다. main() 함수에 보이는 것처럼 bool 타입과 Integer 타입 간에 상호 형변환이 가능하도록 코드를 완성하자.

```

class Integer
{
public:
    int n;
    Integer() : n(0) {}

```

```
};

int main()
{
    Integer i;
    i = true;
    bool b = i;
    return 0;
}

class Integer
{
public:
    int n;
    Integer() : n(0) {}
    Integer(bool b) : n(b) {}
    operator bool() {return n ? true : false;}
};
```

4. C 스타일 형변환의 문제점을 정리했다. 빈 칸을 채우자.

형변환을 수행하는 코드가 눈에 띄지 않는다.  
 정확한 ( **의도** )를 파악하기 힘들다.

5. Complex 클래스를 다음과 같이 사용할 수 있게 연산자를 오버라이드하자.

**27장 소스 폴더 참조(Exer 프로젝트)**

## 28 장 네임스페이스

### Exercise 28-1 Using-Declaration

앞에서 using 키워드를 사용해서 네임스페이스 안쪽의 특정한 이름을 지정하는 방식을 배웠다. using 키워드를 사용해서 cout 객체를 지정하게 조금 전의 예제를 수정하자.

```
#include <iostream>

using std::cout;

int main()
{
    cout << "Hello, World\n";

    return 0;
}
```

이것만은 알고 감시다

1. 네임스페이스에 속한 이름을 일컫는 방법 3가지를 정리했다. 빈 칸을 채워보자. ( 참고로 왼쪽 열은 각각의 경우를 일컫는 전문 용어다.)

Qualified Name	늘 namespace 이름을 붙여서 사용한다.
Using-Directive	using namespace Cat처럼 사용할 네임스페이스를 지정한다.
Using-Declaration	using Cat::CreateAll처럼 사용할 이름만 지정한다.

2. 다음 빈칸에 들어갈 알맞은 말을 보기에서 고르시오.

C++의 표준 라이브러리 코드는 (    ) 네임스페이스 안에 정의되어 있다.

ㄱ) std    ㄴ) stb    ㄷ) stc    ㄹ) std

3. 다음 예제를 실행했을 때 출력하는 결과를 예상해보자.

buttons 를 사용하기 직전에 Keyboards::buttons 를 사용하겠노라고 선언했기 때문에 Keyboards::buttons 의 값인 200 이 출력된다.

## 29 장 템플릿

Vitamin Quiz. `AutoArray<float>`와 `AutoArray<int>`의 관계는?

`AutoArray<float>`와 `AutoArray<int>` 타입의 변수를 두 개 만들어보자. 두 변수 간에 대입이 가능할까? 가능하지 않다면 그 이유는 무엇일까?

대입이 가능하지 않다. `AutoArray<float>`와 `AutoArray<int>`는 전혀 상관이 없는 클래스이기 때문이다. 같은 템플릿 클래스를 기반으로 하지만 완전히 독립적인 2개의 클래스라고 볼 수 있다.

Exercise 29-1 `min()` 함수 만들기

입력된 두 값 중에서 작은 값을 반환하는 `min()` 함수를 템플릿을 사용해서 구현해보자.

```
template< typename T >
T min(T a, T b)
{
    return ( a < b ? a : b );
}
```

Exercise 29-2 `vector` 사용하기

앞의 예제에서 `list` 대신에 `vector`를 사용하게 수정하자.

`#include <list>`를 `#include<vector>`로 바꾸고, `list<int>`를 `vector<int>`로 고치면 된다.

이것만은 알고 갑시다

1. 다음 빈 칸에 들어갈 알맞은 말을 보기에서 고르시오.

템플릿은 (    **ㄱ**    ) 시간에 코드를 만들어낸다.

ㄱ) 한가한    ㄴ) 실행    ㄷ) 한    ㄹ) 컴파일

템플릿 함수의 구현은 (    **ㄷ**    ) 파일에 놓여야 한다.

- 일반적인 템플릿 함수
- 템플릿 멤버 함수
- 템플릿 클래스의 멤버 함수

ㄱ) 여러    ㄴ) 구현    ㄷ) 헤더    ㄹ) 실행

제너릭 프로그래밍에서 가장 중요한 것은 타입과 알고리즘을 (    **ㄱ**    )하는 것이다.

ㄱ) 사랑    ㄴ) 공부    ㄷ) 분석    ㄹ) 분리

2. 프로젝트 장에서 성적표 Ver 3.6의 링크드 리스트 클래스를 템플릿 클래스로 만들어보자.

(참고: 필자가 작성한 소스 코드는 29장 소스 폴더의 `exer` 프로젝트로 만들어두었다)

[소스 폴더 참조](#)



## 30 장 입출력

### Exercise 30-1 진법 변환

setbase 조종자를 사용해서 10진수 65를 각각 8진수와 16로 출력해보자.

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << setbase(8) << 10 << " " << setbase(16) << 10 <<
endl;
}
```

### Exercise 30-2 채팅 내용 저장하기

일본 사람들은 게임 중 채팅을 매우 즐긴다고 한다. 일본에 있는 낚시 게임 개발팀에서 채팅 내용을 파일로 저장하는 기능의 구현을 요청해 왔다. 사용자에게 한 문장씩 입력 받아서 파일에 저장하는 프로그램을 작성하자. 사용자가 "exit"라는 문자열을 입력하면 프로그램이 종료해야 한다.

```
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // 채팅 내용을 저장할 파일을 연다.
    ofstream file1( "chat.txt" );

    while( true )
    {
        // 채팅을 입력받는다.
        string chat;
        getline( cin, chat );

        // 채팅 내용이 "exit" 이면 반복을 끝낸다.
        if ( "exit" == chat )
            break;

        // 채팅 내용을 파일에 기록한다.
        file1 << chat << endl;
    }

    // 파일을 닫는다.
    file1.close();
}
```

이것만은 알고 갑시다

1. 다음 빈 칸에 들어갈 알맞은 말을 보기에서 고르시오.

C++의 (      r      )은/는 데이터가 흘러가는 통로를 추상화한 개념이다.

ㄱ) 버퍼    ㄴ) 정보은닉    ㄷ) 상속    ㄹ) 스트림

C++의 입출력은 내부적으로 (      r      )을/를 수행한다.

ㄱ) 컴파일 ㄴ) 파일 복사 ㄷ) 버퍼링 ㄹ) 양파링

setf() 함수를 비롯한 형식 지정 함수로 할 수 있는 모든 작업은 (    ㄱ    )을/를 사용해서 보다 간단하게 할 수 있다.

ㄱ) 조종자 ㄴ) 수행자 ㄷ) 도망자 ㄹ) 접근자

2. 다음과 같이 보관한 데이터가 있다. 이 데이터를 [그림 30-7]처럼 출력할 수 있게 각각 C 스타일과 C++ 스타일의 출력 방법을 사용해서 구현해보자.

30 장 소스 폴더 참조 (Exer 프로젝트)