C++ 정리

[C++ 언어의 주요한 설계 목적]
C언어와의 호환성
	장점 : 기존의 c 프로그램 코드 그대로 사용
	단점 : 캡슐화의 원칙이 무너짐
객체 지향 개념 도입
	캡슐화, 상속, 다형성을 통해 sw 재사용
실행시간의 효율성 극대화
	실행시간을 저하시키는 요소 해결 ex) 인라인 함수
엄격한 타입 체크
	실행 시간 오류의 가능성 줄임
	디버깅 편리

[C++ 표준 라이브러리]
3개 그룹
	c 라이브러리
		기존 c 표준 라이브러리 수용
		이름이 c로 시작하는 헤더파일 ex) #include <cstring>
	c++ 입출력 라이브러리
	c++ STL 라이브러리
		제네릭 프로그램을 지원하기 위한 라이브러리

[namespace 키워드]
개발자가 자신만의 이름 공간을 생성할 수 있도록 함
이름 충돌 해결
namespace 안에 다른 namespace 사용 가능

[std::]
c++표준에서 정의한 namespace
표준 라이브러리가 들어감
ex)std::cout

[cout 객체]
[iostream] 헤더파일에 선언
표준 c++ 출력 스트림 객체

[<< 연산자]
스트림 삽입 연산자
	오른쪽 피연산자를 왼쪽 스트림 객체에 삽입

[cin 객체]
표준 c++ 입력 스트림 객체
Enter 또는 Space 키가 입력될 때 까지 입력된 키를 입력 버퍼에 저장
	Backspace키로 입력된 키 삭제

[>> 연산자]
스트림 추출 연산자
	입력 스트림에서 값을 읽어 변수에 저장

[c++ 자료형]
bool 타입(1 byte)
	0은 거짓, 0이 아니면 참
	참은 1, 거짓은 0으로 출력
	true / false 상수를 출력하려면 boolalpha 조작자 사용
조작자 : 입력 또는 출력 방식을 바꿔주는 함수
-조작자 사용을 위해 #include <iomanip> 사용
	endl : 스트림 버퍼를 모두 출력하고 다음 줄로 넘어감
	oct : 정수 필드를 8진수 기반으로 출력
		ex) cin >> oct >> data;
	dec : 정수 필드를 10진수 기반으로 출력
	hex : 정수 필드를 16진수 기반으로 출력
	fixed : 실수 필드를 고정소수점 방식으로 출력
	boolalpha : boolean 값이 출력될 때, “true” 또는 “false” 문자열로 출력
	setprecision(int np) : 출력되는 수의 유효숫자 자리수를 np개로 설정
		소수점 별도 카운트
		ex) cout << “고정 소수점 출력” << fixed << setprecision(2) << 12.44725 << endl;
			실행결과 : 12.45 *반올림 유의*
	setw(int minWidth) : 필드의 최소 너비를 minWidth로 지정

[열거형 클래스 enum class]
기존 c언어의 enum의 경우 열거 타입 값이 자동으로 정수로 변환
enum 클래스
	엄격한 타입 적용
		자동으로 열거형 값이 정수로 변환되지 않음
		정수 타입으로 선언하려면 명시적 형 변환 필요
	얼거 타입 값을 사용할 때 반드시 범위지정연산자(::)를 붙여야 함
ex)
```
enum class Color {Blue = 1, Green, Red, Black};
Color color = Color::Blue;
if (static_case<int>(color) == 1)
    cout << “Blue” << endl;
```

[c++ 캐스팅]
const_cast
	포인터 또는 참조형의 const 속성 추가 또는 제거
static_cast
	명시적 형 변환
	컴파일 시 타입 검사
dynamic_cast
	실행 시간에 타입 검사
reinterpret_cast

[auto]
변수 타입은 초기화한 값에 맞춰 컴파일 때 결정
함수 매개변수, 구조체나 클래스의 멤버 변수로 사용 불가
ex) auto value = 20; ==> int형
auto로 선언하면서 반드시 초기화 필요
일반 함수의 반환 값 다입으로 사용가능
	ex) auto 함수명(매개변수) {...}
람다 표현식에서 사용
	ex) auto lambda = [](int a, int b){return a+b;};

[typeid 연산자]
#include <typeinfo> 선언
데이터 타입 정보 반환
ex) typeid(변수).name() ==>타입의 이름을 문자열로 반환

[범위 기반 for 문]
c++ STL의 컨테이너, 배열, initializer_list 등에 사용
ex) for (int a : v) {...} ==> v안의 요소들은 차례로 하나씩 a에 대입하여 반복

[initializer_list]
#include <initializer_list> 선언
동일 타입의 요소를 여러 개 보관하는 템플릿 클래스
여러 개의 인수를 받는 함수를 간단히 작성
ex)
void ABC(initializer_list<int> value) { ... } ==> int 타입의 요소를 list로 value 안에 넣는다
int main() {
	ABC({1,2,3,4,5});
	ABC({1,2,3});
}

[uniform initialization]
{}를 사용하여 변수, 배열, 객체 초기화
ex)

```
class Exam {
public :
    Exam() {cout << "Exam 디폴트 생성자" << endl;}
};
int main() {
    Exam e();          ==> X 생성자 호출이 아니고 함수호출이다
    Exam e{};          ==> O 디폴트 생성자 호출
    int d{3.6};        ==> X 축소 변환 불가
    int f{3};          ==> O f를 3으로 초기화
    int arr[] {3,4,5}; ==> O 배열 초기화
    method({4.9});     ==> X 축소 변환 불가
}
void method(int i) { ... }
```

[구조화된 바인딩]
C++ 17버전 이상
데이터 멤버에 사용
단 접근 지정자 public
tuple 분리 map 사용 시 유용
ex)

```
#include <iostream>
#include <string>
using namespace std;
struct Entry {
    string name;
    int value;
};
Entry read_entry() {
    string s;
    int i;
    cin >> s >> i;
    return {s, i};
}
int main() {
    //1번 방법
    auto e = read_entry();                  // 함수 호출
    cout << e.name << e.value << endl;   //구조체.멤버 로 접근
    //2번 방법
    auto [n, v] = read_entry();             //구조화된 바인딩
    cout << n << v << endl;
}
```

ex)

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

int main() {
    tuple t {“abc”, 23, 41.51}  //튜플 초기화

    auto[str, i, d] = t;   //구조화된 바인딩. 튜플 분리
    //str == “abc”, i == 23, d == 41.51
    return 0;
}
```

[배열] : array / vector 클래스

array / vector 클래스는 컨테이너 클래스

Fixed size array : **array**

-sequence container(순서가 있는 저장장치), limited size(제한된 사이즈를 가짐), random access, 반복자와 알고리즘 함수 사용

-std::array -> stack, compile time, fast allocation

-#include <array> 선언

ex) std::array<int, 100> arr; => int 타입의 100개 array를 arr로 생성

-크기를 미리 정해서 영역을 잡아둔다

Flexible size array : **vector**

-sequence container, huge size(큰 사이즈를 가짐), random access

-std::vector -> heap, runtime, slow allocation(reserve()로 미리 영역을 정해두고 시작함으로 해결)

데이터가 늘어나면 heap 영역 어딘가에 할당한 공간에서 데이터를 담아 다시 복사

-#include <vector> 선언

ex) std::vector<int> arr(100); //100의 영역을 미리 가지는 int 타입의 vector를 arr로 생성

-어느 정도 공간을 정해두고 vector의 크기가 커질수록 공간을 늘려나간다

-vector를 사용하는 이유

stack의 크기가 작을 수 있다

stack은 넣고 빼는 것이 순차적으로 이루어지기 때문에 데이터양이 많아지면 시간이 걸릴 수 있다

array

```
#include <array>
#include <algorithm>
```

배열.size() ==>배열 크기

배열.at() ==>유효 범위 검사. <=> 배열[]

배열.front() ==>첫 번째 원소

배열.back() ==>마지막 원소

배열.empty() ==>빈 배열이면 true, 아니면 false

sort(배열.begin(), 배열.end()); ==>배열 원소 정렬

vector
#include <vector>
벡터.emplace_back() ==>벡터 마지막에 ()원소 추가
벡터.size() ==>원소가 들어있는 크기
벡터.capacity() ==>벡터 전체 영역
벡터.pop_back() ==>벡터 마지막 데이터 리턴

배열의 이름 = 배열의 시작 주소

[C++의 문자열 표현 방식]
C 스트링 방식 => 주소를 포인터로 받아 문자열 제어
	'\0'(또는 NULL) 으로 끝나는 문자 배열
	ex)char name1[6] = {''G', 'r', 'a', 'c', 'e', '\o'};
		문자열O
	char name2[5] = {''G', 'r', 'a', 'c', 'e', '\o'};
		문자열X, 단순 배열
	char name3[10] = "Grace";
		name3 [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
			G r a c e \0 ----null----
string 클래스
	C++ 표준 라이브러리
	#include <string> 선언
	std 안에 string 정의
	문자열의 크기에 따른 제약 없음
		string 클래스가 스스로 문자열 크기에 맞게 내부 버퍼 조절
		가변 크기의 문자열
	null로 끝나지 않는다

	ex)
	string str; //빈 문자열을 가진 스트링 객체
	string address1("강원도 춘천시");
	string address2("강원도 춘천시", 6); //6byte만 받음. 한글은 2byte //address2 = "강원도"
	string address3(address1); //address3 == address1

	getline(cin, str, 'c'); //문자 'c'에서 입력 중지
		getline()은 string 타입의 문자열을 입력받기 위해 제공되는 전역함수
		세 번째 인수는 구분기호(생략시 \n)
		공백이 포함된 문자열 입력가능

	char text[] = {"L', 'O', 'V', 'E', '\n'}; //c 스트링 방식 문자열
	string title(text); //c스트링 으로부터 스트링 객체 생성

	auto string1 = "Hello Cpp"; //const char*
	auto string2 = "Hello Cpp"s; //str::string

string 객체의 동적 생성과 소멸

new / delete를 이용하여 문자열을 동적 생성/반환 가능

ex)string *p = new string("C++"); //string 객체 동적 생성

cout << *p; //"C++" 출력

p -> append(" Great"); //구조체 포인터 사용하여 문자열 붙임

cout << *p; //"C++ Great" 출력

delete p; //스트링 객체 반환

숫자 -> 문자열 변환

to_string(숫자)

문자열 -> 숫자 변환

stoi("1234aa") //스트링을 int형으로 변환 //1234

stol("3456") //스트링을 long형으로 변환 //"a3456"이면 Error

stod("12.34") //double형으로 변환

stof("34.45") //float형으로 변환

[문자열 다루기]

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string s1 = "C++ programming", s2 = "language";
    cout << "문자열 비교" << s1.compare(s2) << endl;     //-1
        //0이면 같다.   0이 아니면 틀리다
    cout << "문자열 연결" << s1.append(s2) << endl;   //C++ programminglanguage
    s1 = "C++ programming";
    cout << "문자열 삽입" << s1.insert(3, "really") <<endl;  //C++really programming
        //인덱스 3번 자리에 추가
    cout << "문자열 대치" << s1.replace(3, 6, "study");  //C++study programming
        //3부터 6개의 문자를 "study"로 바꾼다
    cout << "문자열 길이" << s1.length() << " " << s1.size();  //20 20
    cout << "문자열 삭제" << s1.erasse(0, 4);   //tudy programming
        //0부터 4개의 문자 삭제
    s2 = s1;   //문자열 복사 (깊은 복사)
    s1.clear();  //문자열 전체 삭제
    cout << s2; //tudy programming

    return 0;
}
```

```
#include <iostream>
#include <string>
#include <locale>  //문자를 다루는 함수 라이브러리 ex)isdigit(), toupper(), isalpha(), islower()
using namespace std;
int main() {
    s1 = "C++ programming";

    cout << "문자열 검색" << s1.find("p")  //4
        //문자나 문자열 있으면 인덱스 출력   //없으면 -1
    cout << "문자열 추출" << s1.substr(2, 4);  //+ pr
        //2에서 4개의 문자 추출
    cout << "문자열 추출" << s1.substr(2);  //+ programming
        //2에서 끝까지 추출
    cout << "문자열의 각 문자 다루기" << s1.at(5);  //r
        //인덱스 5에 있는 문자 반환
    cout << "문자 다루기" <<static_cast<char>(toupper('a'));  //A
        //대문자 변환
    if(isdigit(s1.at(6)))           //숫자인지 판별
        cout << "숫자";
    else if(isalpha(s1.at(6)))   //문자인지 판별
        cout << "문자";
    return 0;
}
```

[string_view 클래스]
다양한 타입의 문자열 처리
C++ 17에서만 동작
string과 c 스트링을 한 함수로 받을 수 있다

```
#include <string_view>

string_view extratExt(string_view filename) {
    return filename.substr(filename.find('.'));
}
int main() {
    string filename = "C:\temp\file.string";
    cout << extractExt(filename);    //.string

    const char* cfilename = "C:\temp\file.cstring";
    cout << extracExt(filename);     //.cstring

    cout << extractExt("C:\temp\file.literal");    //.literal
    reutn 0;
}
```

컴파일 시 "g++ 파일이름.cpp -std=c++17

[참조 reference]
이미 존재하는 객체나 변수에 대한 별명
참조 활용
	참조 변수
	참조에 의한 호출
	참조 리턴
참조 변수
	참조자 & 사용	(= 연산자의 왼쪽에 오면 reference, 오른쪽에 오면 주소)
	이미 존재하는 변수에 대한 다른 이름(별명)을 선언
	새로운 공간을 할당하지 않음
	초기화로 지정된 원본 변수의 공간 공유
참조 변수의 선언 가능 범위
	선언 시 반드시 원본 변수로 초기화
		상수 대상 불가
		생성과 동시에 누군가를 참조해야 함
		NULL 초기화 불가능
	참조 변수의 배열 불가
		배열의 요소는 가능
const 참조자로 const 변수 참조, 상수 참조
	const 선언이 들어간 변수들은 참조변수에도 const 필요
	const 참조자는 상수 참조 가능
		상수를 메모리 공간에 임시적으로 저장하기 때문에 소멸되지 않는다
		함수 매개변수 형이 const 참조자인 경우 상수 전달 가능
		ex)Adder(100, 200);
		 int Adder(const int &num1, const int &num2){
		 return num1 + num2;
		 }
포인터 타입 변수에 대한 참조
	포인터 역시 변수
	주소 값을 저장
참조를 사용한 함수 호출
	call by reference
	함수의 매개변수를 참조 타입으로 선언 시
		참조 매개변수(reference parameter)
		참조 매개변수의 이름만 생기고 공간이 생기지 않음
		참조 매개변수는 실 인자 변수의 공간을 공유하여 값을 참조
		참조 매개변수에 대한조작은 실 인자 변수의 조작과 같다
참조에 의한 함수 호출 예)

```
void swap(int &a, int &b){
    int tmp;
    tmp = a; a = b; b = tmp;
}
int main() {
    int m=2; n=9;
    swap(m,n);
}           //m = 9, n = 2
```

주소에 의한 호출 예)

```
void swap(int *a, int *b){
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int m=2, n=9;
    swap(&m, &n);   //매개변수가 포인터이면 주소값을 인자로 줘야함
}       //m = 9, n = 2
```

값에 의한 호출 예)

```
void swap(int a, int b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int m=2, n=9;
    swap(m,n);
}       //m = 2, n = 9   //바뀌지 않는다
```

const 참조자를 이용한 상수 참조

const 참조자를 사용하여 상수를 참조하면 임시 변수를 만들어 참조자가 참조하게 함
함수 내에서 const 참조자를 이용한 값의 변경 허용X
ex) int arr[] = {1,2,3,4,5,6};
//1. 배열에서 범위기반 for 문 : 참조자를 사용한 값 변경

```
for (int &elem : arr)
    elem += 1;
```

//2. 값 변경 및 복사 방지 배열 요소 사용 : const와 참조 사용

```
for (const int &elem : arr)
    cout << elem << " ";    //1 2 3 4 5 6
```

참조 리턴

변수의 값을 리턴하는 것이 아니라 공간에 대한 참조 리턴
ex)int num1 = 1;
int &num2 = Ref(num1);

```
int& Ref(int &ref) {
    ref++;
    return ref;
}  //num1, num2, ref 모두 같은 공간 공유  //단 ref는 함수 종류시 사라짐
```

반환형이 참조형인 경우
반환되는 대상을 참조 또는 변수로 받을 수 있다
반환형이 일반변수이면 참조X
지역변수를 참조의 형태로 반환불가. 변수가 소멸될 때 참조대상이 없어지므로

[클래스와 객체]
클래스
	객체를 만들어내기 위해 정의된 설계도, 틀
	멤버 변수와 멤버 함수 선언
객체
	객체는 생성될 때 클래스의 모양을 그대로 가지고 탄생
	멤버 변수와 멤버 함수로 구성
	메모리에 생성
	하나의 클래스에서 여러개의 객체 생성 가능

클래스 선언부
	class 키워드를 사용하여 클래스 정의, 선언 종료 시 세미콜론(;)
	멤버에 대한 접근 관한 지정
		디폴트값 private
클래스 구현부
	클래스에 정의된 모든 멤버 함수 구현
클래스 선언과 구현 분리 이유
	클래스 재사용
	컴파일 시 클래스 선언부만 필요

클래스 선언과 구현 예시)
```
class Circle {
public:
    int radius;                    //멤버 변수 선언
    double getArea();              //멤버 함수 선언
};
double Circle::getArea() {    //멤버 함수 구현
    return 3.14 * radius * redius;
}
```

객체 생성과 멤버 접근
```
	Circle dunut;    //객체 생성
	donut.redius = 1;  //멤버 변수 접근
	double area = donut.getArea();  //멤버 함수 접근
```

[생성자 constructor]
객체가 생성되는 시점에서 자동으로 호출되는 멤버 함수
클래스 이름과 동일한 멤버 함수

생성자의 목적
	객체가 생성될 때 객체가 필요한 초기화를 위해
생성자는 크게 매개변수가 있는 생성자, 기본 생성자, 복사 생성자로 구분
생성자 선언
	클래스 정의에서 선언
	const 한정자 사용 불가

생성자 함수의 특징

- 생성자 이름
  - 반드시 클래스 이름과 동일
- 생성자는 리턴 타입 없음
- 객체 생성 시 오직 한 번만 자동으로 호출
  - 임의 호출 불가
  - 각 객제마다 생성자 실행
- 생성자 중복 가능
  - 중복된 생성자 중 하나만 실행
- 생성자가 선언되어 있지 않으면 기본 생성자 자동으로 생성
  - 기본 생성자 : 매개변수 없는 생성자
  - 생성자가 하나라도 선언된 클래스의 경우 기본 생성자 자동 생성 X

생성자 예)

```
class Circle {
pubic:
    int radius;
    Circle();                       //디폴트 생성자
    Circle(int r);                  //매개변수 있는 생성자
    Circle(const Circle& cr);       //복사 생성자
    double gerArea();
};
Circle::Circle() {
    radius = 1;
}
Circle::Circle(int r) {
    radius = r;
}
Circle::Circle(const Circle& cr) {
    radius = cr.radius;                 //복사만 하고 변경 불가
}
//gerArea() 구현 생략
int main() {
    Circle donut;    //디폴트 생성자 호출
    double area = donut.gerArea();
    Circle pizza(30);  //매개변수 있는 생성자 호출
    //==Circle pizza{30};
```

명시적 디폴트 생성자, 삭제된 생성자

```
ex)class Circle {
public:
    int radius;
    Circle() = default;    //명시적 디폴트 생성자 //컴파일러가 자동으로 생성
                           //빈 껍데기 생성자를 구현할 필요없다// 객체 배열 사용 시 유용
    Circle(intn r);        //매개변수 있는 생성자
};
```

ex)class Circle {
public:
    Circle() = delete;   //삭제된 생성자 //컴파일러가 디폴트 생성자를 생성하지 않음
                          //정적 메소드로만 구성되는 클래스 정의 시 주로 사용
};

생성자가 다른 생성자 호출(위임 생성자)
        여러 생성자에 중복 작성된 코드의 간소화
        타겟 생성자와 이를 호출하는 위임생성자로 나누어 작성
        타겟 생성자 : 객체 초기화를 전담하는 생성자
        위임 생성자 : 타겟 생성자를 호출하는 생성자, 객체 초기화를 타겟 생성자에 위임
        ex)
        Circle::Circle() {
            radius = 1;
            cout << radius;
        }
        Circle::Circle(int r) {
            radius = r;
            cout << radius;
        }
        ==>
        Circle::Circle() : Circle(1) { }   //Circle(int r)의 생성자 호출 //위임 생성자
        Circle::Circle(int r) {   //타겟 생성자
            radius = r;
            cout << radius;
        }
멤버 변수 초기화 방법
class Point {
    int x, y;
public:
    Point();
    Point(int a, int b);
};
//1.생성자 코드에서 멤버 변수 초기화
Point::Point() {x=0; y=0;}
Point::Point(int a, int b) {x=a; y=b;}
//2.클래스 선언부에서 직접 초기화 (c++ 11부터 가능)
class Point {
    int x = 0; y = 0;   //생성자에서 멤버 초기화가 있을 경우 직접 초기화 보다 우선순위를 가진다
public:
    ....;
};

```
//3. 생성자 initializer 사용
//멤버 변수 x, y를 0으로 초기화
Point::Point() : x{0}, y{0} { }
Point::Point(int a, int b) : x{a}, y{b} { }   //멤버 변수 x=a, y=b로 초기화
```

생성자 initializer 사용시 주의사항

나열한 데이터 멤버는 initializer에 나열한 순서가 아닌 클래스 정의에 작성한 순서대로 초기화

생성자 initializer를 사용해야하는 경우

디폴트 생성자가 없는 객체 멤버 초기화

ex)

```
class Cell{
    double cd;
public:
    cell(double d) : cd(d) { };
};
class Spread{
    Cell mcel;     //객체(생성되어야 사용가능 - 생성자를 호출해야 함)
public:
    Spread() : mcel(4.5) { }  //생성자 initializer를 사용하여 해결
};
```

[소멸자]

객체가 소멸되는 시점에서 자동으로 호출되는 함수

오직 한 번만 자동 호출, 임의로 호출 불가

객체 메모리 소멸 직전 호출됨

소멸자의 목적

객체가 사라질 때 마무리 작업

실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 및 닫기, 네트워크 닫기 등

소멸자 특징

소멸자 함수의 이름은 클래스 이름 앞에 ~ 를 붙인다

소멸자는 리턴 타입 선언 불가

한 클래스 내에 오직 하나만 작성 가능

소멸자는 매개변수 없는 함수

소멸자가 선언되어 있지 않으면 기본 소멸자 자동 생성

컴파일러에 의해 생성된 기본 소멸자 : 아무것도 하지 않고 단순 리턴

ex)

```
class Circle{
    Circle();
    Circle(int r);
    ~Circle();     //소멸자 함수 선언
};
Circle::~Circle() {         //소멸자 함수 구현
    ......;
}
```

[생성자/소멸자 실행 순서]
객체가 선언된 위치에 따른 분류
	지역 객체
		함수 내에 선언된 객체로서 함수가 종료하면 소멸
	전역 객체
		함수의 바깥에 선언된 객체로서 프로그램이 종료할 때 소멸
객체 생성 순서
	전역 객체는 프로그램에 선언된 순서로 생성
	지역 객체는 함수가 호출되는 순간에 순서대로 생성
객체 소멸 순서
	함수가 종료하면, 지역 객체가 생성된 순서의 역순으로 소멸
	프로그램이 종료하면, 전역 객체가 생성된 순서의 역순으로 소멸
new를 이용하여 동적으로 생성된 객체의 경우
	new를 실행하는 순간 객체 생성
	delete 연산자를 실행할 때 객체 소멸

[접근 지정자]
캡슐화의 목적
	객체 보호, 보안
	객체의 상태를 나타내는 데이터 멤버(멤버 변수)에 대한 보호
	중요한 멤버는 다른 클래스나 객체에서 접근할 수 없도록 보호
	외부와의 인터페이스를 위해서 일부 멤버는 외부에 접근 허용
접근 지정자 종류
	private : 동일한 클래스의 멤버 함수에만 허용
	public : 모든 다른 클래스에 허용
	protected : 클래스 자신과 상속받은 자식 클래스에만 허용
접근 지정자는 각각의 멤버에 붙이는 것이 아니라 그룹 단위로 지정
멤버 변수는 private 지정이 바람직하다

[상수형 함수]
멤버 변수에 읽기 접근은 가능하지만 쓰기는 허용되지 않는 메소드
상수형 함수만 호출 가능
선언 방법
	함수 원형 뒤에 const 예약어 사용
		ex)int getArea() const;
상수화 된 대상에 대한 쓰기 작업을 허용하고 싶을 때
	mutable : 멤버 변수 선언에 사용
		ex)mutable int radius;
	const_cast<> : 참조 또는 포인터에 사용
		ex)void test(const int &param) {
			int &newParam = const_cast<int &>(param);
		 }

[함수 호출에 따른 시간 오버헤드]
작은 크기의 함수를 호출하면, 함수 실행 시간에 비해 호출을 위해 소요되는 부가적인 시간 오버헤드가 상대적으로 크다 -간단한 함수는 함수를 사용하지 않는 것이 더 유용할 수 있다

[인라인 함수]
inline 키워드로 선언된 함수
인라임 함수를 호출하는 곳에 인라인 함수 코드를 확장 삽입
	코드 확장 후 인라인 함수 소멸
인라인 함수 호출
	함수 호출에 따른 오버헤드 X
	프로그램 실행 속도 개선
	컴파일러에 의해 이루어짐
인라인 함수의 목적 : 실행 속도 향상
	자주 호출되는 짧은 코드의 함수 호출에 대한 시간 소모를 줄임
컴파일러는 inline 처리 후 확장된 c++ thtm 파일을 컴파일한다
ex)
```
inline int add(int x){
    return (x%2);
}
int main() {
    int sum = 0;
    for (int i = 1; i <= 10000; i++) {
        if(add(i))            //==if(i%2)  //컴파일러의 의해 inline 함수의 코드 확장 삽입
            sum += i;
    }
    cout << sum;
}
```
인라인 제약 사항
	컴파일러가 판단하여 inline 요구를 수용할지 결정
		recursion(재귀), 긴 함수, static, 반복문, goto 문 등을 가진 함수 불가
장점 : 프로그램 실행 시간이 빨라진다
단점 : 컴파일 된 전체 코드 크기 증가

자동 인라인 함수
	클래스 선언부에 구현된 멤버 함수
		컴파일러에 의해 자동 인라인 처리
		생성자를 포함, 모든 함수가 자동 인라인 함수 가능
	ex)
```
class Circle {
    int radius;
public:
    Circle() {          //자동 인라인 함수
        radius = 1;
    }
};
```

[c++ 구조체]
모든 것이 클래스와 동일
클래스와 다른점
	구조체의 디폴트 접근 지정 : public
	클래스의 디폴트 접근 지정 : private
c 언어와의 호환성 때문에 사용
구조체 객체 생성
	ex)
	struct abc {
	private:
	 ....
	public:
	 ....
	};
	abc stobj;	//구조체 객체 생성

[바람직한 c++ 프로그램 작성법]
클래스를 헤더파일과 cpp 파일로 분리하여 작성
	클래스마다 분리 저장
	인터페이스 파일 - 클래스 선언부
		헤더파일(.h)에 저장
	구현 파일 - 클래스 구현 부
		cpp 파일에 저장
		클래스가 선언된 헤더 파일 include
	어플리케이션 파일
		객체를 인스턴스화하고 객체를 활용하는 main 함수의 코드가 포함
		필요시 클래스가 선언된 헤더파일 include
목적 : 클래스 재사용

헤더 파일의 중복 include 문제 해결방법
	1.조건 컴파일 사용
		조건 컴파일 문의 상수는 클래스의 이름 대문자로 사용
	//Circle.h에서
	#ifndef CIRCLE_H
	#define CIRCLE_H
	class Circle { ....; };
	#endif

	2.#pragma once 지시자 사용
	//Circle.h에서
	#pragma once
	class Circle {
	.../
	};

[객체 포인터]
객체의 주소 값을 가지는 변수
포인터로 멤버를 접근할 때
	1.객체포인터 -> 멤버
	2.(*객체포인터).멤버
ex)Circle donut;
double d = donut.getArea();

Circle *p;	//객체에 대한 포인터 선언
p = &donut;	//포인터에 객체 주소 저장
d = p -> getArea();	//멤버 함수 호출
//== d = (*p).getArea();

[nullptr]
Null Pointer를 의미
	포인터가 아무것도 가리키고 있지 않음
ex) double *p = nullptr;
널 포인터를 NULL 매크로나 상수 0 사용시 문제점
	NULL 매크로나 상수 0을 사용하여 함수에 인자로 넘기는 경우 int 타입으로 추론한다

[객체 배열의 생성 및 소멸]
ex) Circle c[3];	//Circle 타입의 배열 선언
객체 배열 선언
	1. 객체 배열을 위한 공간 할당
	2. 배열의 각 원소인 객체마다 디폴트 생성자 호출
		c[0]의 생성자, c[1]의 생성자, c[2]의 생성자 순으로 실행
		매개변수 있는 생성자 호출 불가
객체 배열 소멸
	배열의 각 객체마다 소멸자 호출
	생성의 역순으로 소멸
객체 배열의 초기화
	1.배열의 각 원소인 객체 각각에 생성자를 지정하는 방법
	ex)
	Circle cirArr[3] = { Circle(10), Circle(20), Circle() };
	//cirArr[0] 객체가 생성될 때, 생성자 Circle(10) 호출
	//cirArr[1] 객체가 생성될 때, 생성자 Circle(20) 호출
	//cirArr[2] 객체가 생성될 때, 생성자 Circle() 호출
	2.
	ex)
	Circle cirArr[3];
	cirArr[0].setRadius(2);
	cirArr[1].setRadius(7);
	cirArr[2].setRadius(4);

[Stack vs Heap]

	Stack	Heap
function life cycle	stack frame	pointer
memory size	small	large
memoy allocation	complie time	runtime(dynamic)

stack memory allocation(할당)
	변수 선언을 통해 필요한 메모리 할당
	많은 양의 메모리는 배열 선언을 통해 할당
	int, float, double, small array, 작은 객체
	-빠르고 간단하다

heap memory allocation
	필요한 양이 예측되지 않는 경우
		프로그램 작성 시 메모리를 할등 받을 수 없음으로
	실행 중 운영체제로부터 heap 메모리를 할당받음
		heap 메모리는 운영체제가 소유하고 관리
	large array, 큰 객체 (약 1kb이상)
	runtime(dynamic) 할당 시

[동적 메모리 할당 및 반환]
new 연산자
	객체의 동적 생성
		heap 메모리에 객체를 위한 메모리 할당 요청
	객체 할당 시 생성자 호출
delete 연산자
	new로 할당받은 메모리 반환
	객체의 동적 소멸, 소멸자 호출 뒤 객체를 heap에 반환

크기와 형 변환 필요 없음
new / delete 연산자의 사용 형식
	데이터타입 *포인터변수 = new 데이터타입;
	delete 포인터변수;
ex).....

```
int main() {
    int *p = nullptr;
    p = new int;     //heap에 int 영역 할당
    *p = 5;          //할당한 영역에 5 대입

    delete p;        //객체 소멸  //delete 후 포인터 p는 살아있지만 접근하면 안된다
    p = nullptr;     //따라서 nullptr로 해제된 메모리를 다시 사용할 수 있는 실수를 방지한다
}
```

delete 사용시 주의사항
	적절하지 못한 포인터로 delete 하면 실행시간 오류 발생
		동적으로 할당 받지 못한 메모리 반환 시 오류
		동일한 메모리 중복 반환 시 오류

[배열의 동적 메모리 할당 및 반환]
배열 형태로 동적 생성한 것은 배열 형태로 삭제
배열 인덱스 표시 생략 불가

데이터타입 *포인터변수 = new 데이터타입[배열의 크기];
delete [] 포인터변수;

ex)
```
int *p = new int[5];    //heap 영역에 int[5] 배열 영역 할당
for(int i = 0; i<5; i++)
    p[i] = i;
delete [] p;              //delete 후 p는 살아있지만 사용X
```

[동적 할당 메모리 초기화]
ex)
```
int *p = new int{20};        //== new int(20);
char *ch = new char{'c'};
```

배열은 동적 할당 시 초기화 기본적으론 불가능
유니폼 초기화(uniform initializer)를 사용하면 가능
ex)int *pa = new int[4] {3, 4, 5, 6};

[객체의 동적 생성 및 반환]
new 연산자는 생성자를 호출하고, delete 연산자는 소멸자를 호출
클래스이름 *포인터변수 = new 클래스이름;
클래스이름 *포인터변수 = new 클래스이름(생성자 매개변수 리스트);
delete 포인터변수;   //생성한 순서엔 관계없이 사용가능

[객체 배열의 동적 생성 및 반환]
클래스이름 *포인터변수 = new 클래스이름[배열 크기]; //각 객체마다 자동으로 디폴트 생성자 실행
delete [] 포인터변수;   //포인터변수가 가리키는 객체 배열을 반환   //각 객체마다 소멸자 실행

[객체 배열의 사용, 배열의 반환과 소멸자]
동적으로 생성된 배열도 보통 배열처럼 사용
ex)
```
Circle *pArray = new Circle[3];
pArray[0].SetRadius(10);
....
```
포인터로 배열 접근
ex)
```
(pArray+1) -> getArea();
```
배열 소멸
ex)
```
delete [] pArray;      //생성의 역순으로 소멸자 실행
```

[클래스 멤버의 동적 생성]
클래스의 멤버도 동적 생성 가능
　　단 생성자에서 동적 할당되어야 하고, 소멸자에서 동적 메모리를 헤제해야 함
ex)
```
class Dog {
    string *name;
public:
    Dog(string n) : name{new string{n}} { }
  //Dog(string n) {
  //    name = new string{n};
  //}
    ~Dog() {
        delete name;      //반드시 delete 해야한다
    }
};
```

[객체 포인터 배열]
사용자가 Runtime 기간 동안 입력하는 경우
ex)
```
Circle *cir[3];
for(int i = 0; i < sizeof(cir); i++)
    cir[i] = new circle(r);
for(int i = 0; i < sizeof(cir); i++)
    delete cir[i];
```

[동적 메모리 할당과 메모리 누수(memory leak)]
ex)
```
char n = 'a';
char *p = new char[1024];
p = &n;       //할당받은 1024바이트의 메모리 누수 발생!
              //누수메모리는 반환 불가, 사용 불가
```

ex)
```
char *p;
for(int i = 0; i < 100000; i++) {
    p = new char[1024];   //p는 새로 할당 받는 1024 바이트의 메모리를 가리키므로
}                         //이전에 할당받은 메모리의 누수 발생
```

프로그램이 종료되면 운영체제는 누수 메모리를 모두 heap에 반환한다

[class 객체의 size]
C++에서의 정수의 size = 16~64 bits
　　일반적으론 4byte(32bits)
　　반드시 sizeof()로 사이즈를 확인해야 한다

ex)

```
class Dog{
    int a;          //4bytes
    double b;       //8bytes
};

main() {
    Dog dog;        //dog의 size는 12bytes일 수도 있고 아닐 수도 있다!
}
```

c++ objects의 전체 사이즈

1. 가장 큰 멤버의 배수의 위치에서 끝나야 한다
2. 모든 변수는 자신의 size의 배수의 위치에서 시작한다

Dog : X

0	1	2	3	4	5	6	7	8	9	10	11
int a				double d							

Dog : O

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
int a				빈 공간(패딩)				double b							

전체 사이즈 = 16bytes

만약

```
class Dog {
    double d;
    int a;
};
```

이면

Dog

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
double b								int a				빈 공간			

전체 사이즈 = 16bytes

가장 큰 멤버 double의 배수 16의 위치에서 끝나야 한다

객체의 사이즈가 다른 이유 = cache(memory) sharing 때문이다

cache는 데이터를 한번에 32byte 또는 64byte 등 2^n 단위로 값을 읽어오기 때문

[스마트 포인터]
포인터처럼 동작하는 클래스 템플릿
	시용이 끝난 메모리를 자동으로 해제
메모리 누수로부터 프로그램의 안전성 보장을 위해 제공
#include <memory> 선언

스마트 포인터 종류
	unique_ptr
	shared_ptr
	weak_ptr

unique_ptr
	하나의 스마트 포인트만 객체를 소유
	스마트 포인터가 영역을 벗어나거나 리셋되면 참조하던 resiurce 해제
	복사(copy), 대입(assign), 공유(share) 불가
	포인터에 대한 소유권 이전(move) 가능

shared_ptr
	하나의 특정 객체를 참조하는 스마트 포인터의 개수를 참조하는 스마트 포인터
	참조 카운트(reference count), use_count()
		해당 메모리를 참조하는 포인터가 몇 개인지 나타내는 값
	shared_ptr이 추가될 때 1씩 증가, 수명이 다하면 1씩 감소
	0이 되면 메모리 자동 해제
	ex)

weak_ptr
	하나 이상의 shared_ptr이 가리키는 객체를 참조할수 있지만 reference count를 늘리지 않는 스마트포인터
	순환 참조를 제거하기 위해 사용
		순환 참조 : shared_ptr이 서로 상대방을 가리키는 것으로 reference count가 0이 되지 않아 메모리가 해제되지 않는 형태
		메모리 누수 발생!
		그림)

unique_ptr 예제

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    //1
    unique_ptr<int> p(new int);       //비추
    *p = 99;                          //포인터의 초기화

    unique_ptr<int> ap = move(p);  //move()를 사용해 포인터 이동, *p는 이제 사용불가
                                   //unique_ptr<int> ip = p;   //에러!
    //2
    unique_ptr<double> sp = make_unique<double>(23.5);
    //*sp == 23.5;

    //3
    auto asp = make_unique<int []>(5);      //5개의 정수형 배열 형태의 unique_ptr 생성
    asp[1] = 10;     //일반 배열처럼 사용!
```

unique_ptr 동적 객체 생성

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;
class Person {
    string name;
    int age;
public:
    Person() = default;
    Person(string n, int a) : name(n), age(a) { }
    ~Person() { cout << "객체 소멸" << endl; }
    void disPlay() { cout << "name = " << name << " , age = " << age << endl; }
};
int main() {
    auto p = make_unique<Person>("unique", 20);
    p -> disPlay();       //일반적인 포인터와 동일하게 사용한다
}
```

실행결과

```
        name = unique, age = 20
        객체 소멸
```

unique_ptr 동적 객체 멤버 생성
객체의 delete는 자동으로 수행된다

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;
class Person {
    unique_ptr<string> name;
    unique_ptr<int> age;
public:
    Person();
    Person(string n, int a);
    ~Person();
    void disPlay() const;
};
Person::~Person() {
    cout << *name << " 객체소멸" << endl;
}
Person::Person() : Person("null", 0); { }
Person::Person(stirng n, int a) : name{make_unique<string>(n)}, age{make_unique<int>(a)} { }
/*또는
Person::Person(string n, int a) {
    name = make_unique<string>(n);
    age = make_unique<int>(a);
}
void Person::disPlay() const {
    cout << "name = " << *name << ", age = " << *age << endl;
}
int main() {
    unique_ptr<Person> p = make_unique<Person>("java", 30);
    p -> disPlay();
    return 0;
}
```

실행결과

```
name = java, age = 30
java 객체 소멸
```

shared_ptr, use_count() 예제

```
#include <iostream>
#include <memory>
using namespace std;
class Person {
    string name;
    int age;
public:
    Person() = default;
    Person(string n, int a) : name(n), age(a) { }
    ~Person() { cout << "메모리 해제" << endl; }
    void disPlay() { cout << "name = " << name << " , age = " << age << endl; }
};
void show(share_ptr<Person> sp) {
    sp -> disPlay();
    cout << "show().sp.use_count() : " << sp.use_count() << endl;
}
int main() {
    auto sp1 = make_shared<Person>("unique", 17);
    show(sp1);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;

    shared_ptr<Person> sp2 = sp1;      //또는 auto sp2 = sp1;
    show(sp2);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;
    cout << "sp2.use_count() : " << sp2.use_count() << endl;
  }
```

실행 결과

```
        name = unique, age = 17
        show().sp.use_count() : 2
        sp1.use_count : 1                  //show() 함수를 stack frame에서 제거했기 때문!
        name = unique, age = 17
        show().sp.use_count() : 3
        sp1.use_count() : 2                //show() 함수를 stack frame에서 제거했기 때문!
        sp2.use_count() : 2
        메모리 해제
```

shared_ptr 메모리 leak 예제

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;
class Person {
    string name; int age;
public:
    Person(string n, int a) : name(n), age(a) { }
    ~Person() { cout << "메로리 해제" << endl; }
    void disPlay() { cout << "name = " << name << ", age = " << age << endl; }

    shared_ptr<Person> per;     //public 멤버변수 포인터    //메모리 leak 해결방법 = weak_ptr 사용!
};
void show(shared_ptr<Person> sp) {
    sp -> disPlay();
    cout << "show().sp.use_count() : " << sp.use_count() << endl;
}
int main() {
    auto sp1 = make_shared<Person>("unique", 17);
    show(sp1);
    cout << "sp1.use_count : " << sp1.use_count() << endl;

    sp1 -> per = sp1;
    show(sp1 -> per);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;
}
```

실행 결과

```
        name = unique, age = 17
        show().sp.use_count() : 2
        sp1.use_count() : 1
        name = unique, age = 17
        show().sp.use_count() : 3
        sp1.use_count() : 2
```

-> 서로가 서로를 참조하는 순환참조 형태가 됨

        메모리 leak 발생

c & c++ 동적 메모리 할당 예제

-c style

```
class Apple {
public:
    Apple() { cout << “yummy apple” << endl; }
    ~Apple() { cout << “finished apple” << endl; }
};
int main() {
    //heap int : c style
    int *a = (int *)malloc(sizeof(int));
    *a = 100;
    free(a);

    //heap int array : c style
    int *b = (int *)malloc(sizeof(int) * 3);
    b[0] = 100;
    free(b);

    //heap Apple : c style
    Apple *c = (Apple *)malloc(sizeof(Apple));        //sizeof() 사용 - 예측한 size와 다를 수 있기 때문
    free(c);

    //heap Apple array : c style
    Apple *d = (Apple *)malloc(sizeof(Apple) * 3);
    free(d);
}
```

==> 생성자와 소멸자가 자동으로 호출되지 않는다!

-c++ style

```
class Apple {
public:
    Apple() { cout << “yummy Apple” << endl; }
    ~Apple() { cout << “finished Apple” << endl; }
};
int main() {
    //heap int : c++ style
    int *a = new int;
    *a = 100;
    delete a;

    //heap int array : c++ style
    int *b = new int[3];
    b[0] = 100;
    delete [] b;

    //heap Apple : c++ style
    Apple *c = new Apple;
    delete c;

    //heap Apple array : c++ style
    Apple *d = new Apple[3];
    delete [] d;
}
```

==> 생성자와 소멸자 자동으로 호출된다
더 안정적

safer c++ 동적 메모리 할당 예제

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

class Apple {
public:
    Apple() { cout << “yummy Apple” << endl; }
    ~Apple() { cout << “finished Apple” << endl; }
};
int main() {
    cout << “== heap int : c++ style==” << endl;
    unique_ptr<int> a = make_unique<int>();
    *a = 100;

    cout << “== heap int array : vector<int> == ”  << en이;
    vector<int> b(3);
    b[0] = 55;          //또는 b.at(0) = 55;

    cout << “== heap Apple : memory leak 해결==” << endl;
    unique_ptr<Apple> c = make_unique<Apple>();

    cout << “== heap Apple array : memory leak 해결==” << endl;
    int count;
    cout << “ vector size ? :”;
    cin >> count;
    vector<Apple> d(count);       //생성자 자동 호출
}//소멸자 자동 호출
```

[this 포인터]

this

포인터, 개체 자신 포인터

클래스의 멤버 함수 내에서만 사용

static 사용 불가

컴파일러가 선언한 변수

this 포인터의 실체

컴파일러에서 처리

```
//개발자가 작성한 클래스
class Sample {
    int a;
public:
    void setA(int x) { this -> a = x; }
};

//컴파일러에 의해 변환된 클래스
class Sample {
    ....
public:
    void setA(Sample* this, int x) { this -> a = x; }
};
```

```
Sample ob;        ------------>        ob.setA(&ob, 5);
ob.setA(5);                           //임의 주소가 this 매개변수에 전달됨
```

this와 객체

각 객체 속의 this는 다른 객체의 this와 다름

this = 클래스로 만든 모든 객체에 공통적인 것이 아닌 각가 객체에 대한 자신의 포인터이다

```
class Circle {
    int radius;
public:
    Circle() { this -> radius = 1; }
    Circle (int radius) { this -> radius = radius; }
    void setRadius(int radius) { this ->radius = radius; }
};
int main() {
    Circle c1;
    Circle c2(2);
    Circle c3(3);

    c1.setRadius(4);
    c1.setRadius(5);
    c1.setRadius(6);
}
```

this가 필요한 경우

매개변수의 이름과 멤버변수의 이름이 같은 경우

ex)Circle(int radius) {

this -> radius = radius;

} //멤버변수//매개변수

멤버 함수가 객체 자신의 주소를 리턴할 때

연산자 중복에서 주로 사용

ex)class Sample {

public:

Sample* f() {

...

return this;

}

};

this 제약사항

멤버함수가 아닌 함수에서 this 사용불가

객체와의 관련성이 없기 때문

static 멤버 함수에서 this 사용불가

객체가 생기기 전에 static 함수 호출이 있을 수 있기 때문