

클래스와 객체

클래스와 객체

- 한 주간 잘 지내셨나요?
- 오늘은 C++ 클래스와 객체에 대하여 알아보려고 합니다
- 이미 자바라는 객체 지향 언어를 통하여 클래스와 객체에 대한 경험을 했습니다.
- 이번 시간에는 클래스, 생성자, 소멸자, 위임생성자, 인라인 함수 등에 대하여 알아보도록 하겠습니다.
- 자바와 비교해 가면서 C++ 언어에서는 어떤 방식으로 클래스가 구현이 되는지 차이점을 알아가도록 하겠습니다
- 지난 시간 복습 후 시작하도록 하겠습니다.
- 준비 되셨나요?

학습 목차

- 클래스와 객체
- 생성자
- 소멸자
- 접근지정자
- 인라인 함수
- C++ 구조체
- 바람직한 C++ 프로그램 작성법

학습 목표

- 클래스 선언과 구현을 이해할 수 있다
- C++ 소멸자를 이해할 수 있다
- 위임생성자와 타겟 생성자를 이해할 수 있다
- 인라인 함수를 이해할 수 있다
- 헤더 파일 중복 문제를 해결할 수 있다
- 바람직한 C++ 프로그램 개발을 할 수 있다

C++ 클래스와 C++ 객체

- 객체는 상태(state)와 행동(behavior)으로 구성
- 클래스
 - 객체를 만들어내기 위해 정의된 설계도, 틀
 - 멤버 변수(상태)와 멤버 함수(행동) 선언
- 객체
 - 객체는 생성될 때 클래스의 모양을 그대로 가지고 탄생
 - 멤버 변수와 멤버 함수로 구성
 - 메모리에 생성, 실체(instance)라고도 부름
 - 하나의 클래스 틀에서 찍어낸 여러 개의 객체 생성 가능
 - 객체들은 상호 별도의 공간에 생성

C++ 클래스 만들기

- 클래스 작성
 - 멤버 변수와 멤버 함수로 구성
 - 클래스 선언부와 클래스 구현부로 구성
- 클래스 선언부(class declaration)
 - class 키워드를 이용하여 클래스 정의, 선언 종료 시 **;(세미콜론)**
 - 멤버 변수와 멤버 함수 선언
 - 멤버 함수는 **원형(prototype) 형태로 선언**
 - 멤버에 대한 접근 권한 지정
 - private, public, protected 중의 하나
 - **디폴트는 private**
 - public : 다른 모든 클래스나 객체에서 멤버의 접근이 가능함을 표시
- 클래스 구현부(class implementation)
 - 클래스에 정의된 모든 멤버 함수 구현

C++ 클래스 만들기

- 클래스 선언(정의)과 구현 예

```
class Circle {
public: //접근 지정자
    int radius; // 멤버 변수 선언
    double getArea(); // 멤버 함수 선언
}; //세미콜론으로 끝남
```

클래스 선언과 구현 분리 이유

- 클래스 재사용
- 컴파일 시 클래스 선언부만 필요

클래스 선언부

클래스 이름

범위지정연산자 : 같은 이름의 함수가 다른
클래스에 존재할 수 있음

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

클래스 구현 : 멤버 함수 구현

C++ 클래스 만들기

객체 생성과 멤버 접근

(1) Circle donut;

객체 이름

객체가 생성되면
메모리가 할당된다.

int radius

double getArea() {...}

donut 객체

// 멤버 변수 접근

(2) donut.radius = 1;

int radius

double getArea() {...}

donut 객체

// 멤버 함수 접근

(3) double area = donut.getArea();

main()

area

int radius

double getArea() {...}

donut 객체

Circle 클래스의 객체 생성 및 활용

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

```
class Circle { //Circle 선언 – 클래스 정의
public:
    int radius;      //멤버 변수 선언
    double getArea(); //멤버 함수 선언
};
```

```
//Circle 클래스 구현 – 멤버 함수 구현
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle donut;
    donut.radius = 1;
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;
```

```
    Circle pizza;
    pizza.radius = 30;
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
}
```

생성자

- 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 생성자라는 3가지로 구분
- 생성자 선언
 - 생성자는 클래스의 멤버 함수이므로 클래스 정의에서 선언
 - 데이터 멤버를 초기화하는 변경 작업을 하므로 `const` 한정자를 붙일 수 없음

생성자 함수의 특징

- 생성자 이름
 - 반드시 클래스 이름과 동일
- 생성자는 리턴 타입을 선언하지 않는다.
 - 리턴 타입 없음. void 타입도 안됨
- 객체 생성 시 오직 한 번만 호출
 - 자동으로 호출됨. 임의로 호출할 수 없음. 각 객체마다 생성자 실행
- 생성자는 중복 가능
 - 생성자는 한 클래스 내에 여러 개 가능
 - 중복된 생성자 중 하나만 실행
- 생성자가 선언되어 있지 않으면 기본 생성자 자동으로 생성
 - 기본 생성자(디폴트 생성자) – 매개 변수 없는 생성자
 - 컴파일러에 의해 자동 생성
 - 생성자가 하나라도 선언된 클래스의 경우 - 기본 생성자를 자동 생성하지 않음

3 개의 생성자를 가진 Circle 클래스

```
class Circle {
public:
    int radius;
    Circle();           // 디폴트 생성자
    Circle(int r);      // 매개 변수 있는 생성자
    Circle(const Circle& cr); // 복사 생성자
    double getArea();
};
```

- Initializer-list 생성자 - STL에서 설명
- 이동 생성자 - 참조에서 설명

```
Circle::Circle() {
    radius = 1;
}
```

```
Circle::Circle(int r) {
    radius = r;
}
```

```
Circle::Circle(const Circle& cr) {
    radius = cr.radius;
}
```

//멤버 함수 getArea() 정의 생략

```
int main() {
    Circle donut;    // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;

    Circle pizza(30); // 매개 변수 있는 생성자 호출
    // 또는 Circle pizza{30}
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
}
```

*매개변수 없는 생성자 호출하여 객체 생성 시 주의
 Circle donut(); //error, 객체 생성과 함수 원형이 충돌
 //함수 원형에서만 사용
 Circle donut{}; //ok, 디폴트 생성자로 객체 생성

명시적 디폴트 생성자 & 삭제된 생성자

```
class Circle {  
public:  
    int radius;  
    Circle() = default;    // 명시적 디폴트 생성자-컴파일러가 자동으로 생성  
                           // 빈 껍데기 생성자를 구현할 필요 없음, 객체 배열 사용 시 유용  
    Circle(int r);        // 매개 변수 있는 생성자  
};
```

```
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 위임 생성자로 나누어 작성
 - 타겟 생성자 : 객체 초기화를 전담하는 생성자
 - 위임 생성자 : 타겟 생성자를 호출하는 생성자, 객체 초기화를 타겟 생성자에 위임

```
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

여러 생성자에 코드 중복

위임 생성자

타겟 생성자

```
Circle::Circle() : Circle(1) { } // Circle(int r)의 생성자 호출

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

호출

r에 1 전달

간소화된 코드

다양한 멤버 변수 초기화 방법

(1) 생성자 코드에서 멤버 변수 초기화

```
Point::Point() { x = 0; y = 0; }  
Point::Point(int a, int b) { x = a; y = b; }
```

```
class Point {  
    int x, y;  
public:  
    Point();  
    Point(int a, int b);  
};
```

(2) 클래스 선언부에서 직접 초기화(C++ 11부터 가능)

```
class Point {  
    int x=0; y=0; // 클래스 선언부에서 x, y를 0으로 직접 초기화  
                //생성자에서 멤버 초기화가 있을 경우 직접 초기화 보다 우선순위를 갖는다  
public:  
    ...  
};
```

다양한 멤버 변수 초기화 방법

(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=a로, y=b로 초기화
Point::Point(int a, int b) : x{a}, y{b} { }
```

```
class Point {
    int x, y;
public:
    Point();
    Point(int a, int b);
};
```

1) 생성자 initializer 사용시 주의 사항

- 나열한 데이터 멤버는 initializer에 나열한 순서가 아닌 클래스 정의에 작성한 순서 대로 초기화

//2) 생성자 initializer를 사용해야 하는 경우 – 디폴트 생성자가 없는 객체 멤버 초기화

```
class Cell {
    double cd;
public:
    Cell(double d) : cd(d) {};
};
```

const 데이터 멤버
레퍼런스 데이터 멤버
디폴트 생성자가 없는 베이스 클래스

```
class Spread {
    Cell mcel;
public:
    Spread() {} //컴파일 error, Cell에 디폴트 생성자가 존재하지 않음(mcel 초기화 불가)
    Spread() : mcel(4.5) {} //생성자 initializer를 사용하여 해결
};
```


소멸자

- 소멸자
 - 객체가 **소멸**되는 시점에서 **자동**으로 호출되는 **함수**
 - 오직 한번만 자동 호출, 임의로 호출할 수 없음
 - 객체 메모리 소멸 직전 호출됨
- 소멸자의 목적
 - 객체가 사라질 때 마무리 작업을 위함
 - 실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 및 닫기, 네트워크 닫기 등

소멸자 특징

- 소멸자 함수의 이름은 클래스 이름 앞에 ~를 붙인다.
 - 예) Circle::~~Circle() { ... }
- 소멸자는 리턴 타입이 없고, 어떤 값도 리턴하면 안됨
 - 리턴 타입 선언 불가
- 중복 불가능
 - 소멸자는 한 클래스 내에 오직 한 개만 작성 가능
 - 소멸자는 매개 변수 없는 함수
- 소멸자가 선언되어 있지 않으면 기본 소멸자 자동 생성
 - 컴파일러에 의해 기본 소멸자 코드 생성
 - 컴파일러가 생성한 기본 소멸자 : 아무 것도 하지 않고 단순 리턴

```
class Circle {
    Circle();
    Circle(int r);

    //소멸자 함수 선언
    ~Circle();    // 리턴 타입 없고 매개변수 없음
                  // 소멸자는 오직 하나만 존재
};

Circle::~~Circle() {    //소멸자 함수 구현
    .....
}
```

생성자/소멸자 실행 순서

- 객체가 선언된 위치에 따른 분류
 - 지역 객체
 - 함수 내에 선언된 객체로서, 함수가 종료하면 소멸된다.
 - 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

지역 객체와 전역 객체의 생성 및 소멸 순서

실행 결과는 무엇인가?

```
#include <iostream>
using namespace std;
class Circle {
public:
    int radius;
    Circle();
    Circle(int r);
    ~Circle();
    double getArea();
};
Circle::Circle() : Circle(1) { }
Circle::Circle(int r) : radius{r} {
    cout << "반지름 " << radius << " 생성" << endl;
}
Circle::~~Circle() {
    cout << "반지름 " << radius << " 소멸" << endl;
}
double Circle::getArea() { return 3.14*radius*radius; }
```

```
C:\WINDOWS\system32\cmd.exe
반지름 1000 생성
반지름 2000 생성
반지름 1 생성
반지름 30 생성
반지름 100 생성
반지름 200 생성
반지름 200 소멸
반지름 100 소멸
반지름 30 소멸
반지름 1 소멸
반지름 2000 소멸
반지름 1000 소멸
계속하려면 아무 키나 누르십시오 . . .
```

//전역 객체 생성

```
Circle globalDonut(1000);
Circle globalPizza(2000);
```

void f() {

//지역 객체 생성

```
Circle fDonut(100);
Circle fPizza(200);
}
```

int main() {

Circle mainDonut; //지역 객체 생성

Circle mainPizza(30);

f();

}

접근 지정자

- 캡슐화의 목적
 - 객체 보호, 보안
 - C++에서 객체의 캡슐화 전략
 - 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
- 멤버에 대한 3 가지 접근 지정자
 - private : 동일한 클래스의 멤버 함수에만 제한함
 - 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 바람직함

```
class Circle {
public:
    int radius; //멤버변수를 보호하지 못함
    Circle();
    Circle(int r);
    double getArea();
};
```

```
Circle::Circle() {
    radius = 1;
}
Circle::Circle(int r) {
    radius = r;
}
```

```
int main() {
    Circle waffle;
    waffle.radius = 5;
}
```

(a) 멤버 변수를 public으로 선언한 나쁜 사례



```
class Circle {
private:
    int radius; //멤버변수를 보호함
public:
    Circle();
    Circle(int r);
    double getArea();
};
```

```
Circle::Circle() {
    radius = 1;
}
Circle::Circle(int r) {
    radius = r;
}
```

```
int main() {
    Circle waffle(5); // 생성자에서 radius 설정
    //Circle waffle{5};
    waffle.radius = 5; // private 멤버 접근 불가
}
```

(b) 멤버 변수를 private으로 선언한 바람직한 사례

다음 소스의 컴파일 오류가 발생하는 곳은 어디인가?

```
class PrivateAccessError {
private:
    int a;
    void f();
    PrivateAccessError();
public:
    int b;
    PrivateAccessError(int x);
    void g();
};

PrivateAccessError::PrivateAccessError() {
    a = 1;    // (1)
    b = 1;    // (2)
}

PrivateAccessError::PrivateAccessError(int x) {
    a = x;    // (3)
    b = x;    // (4)
}
```

```
void PrivateAccessError::f() {
    a = 5;    // (5)
    b = 5;    // (6)
}

void PrivateAccessError::g() {
    a = 6;    // (7)
    b = 6;    // (8)
}

int main() {
    PrivateAccessError objA;    // (9)
    PrivateAccessError objB(100); // (10)
    objB.a = 10;                // (11)
    objB.b = 20;                // (12)
    objB.f();                    // (13)
    objB.g();                    // (14)
}
```

상수형 함수

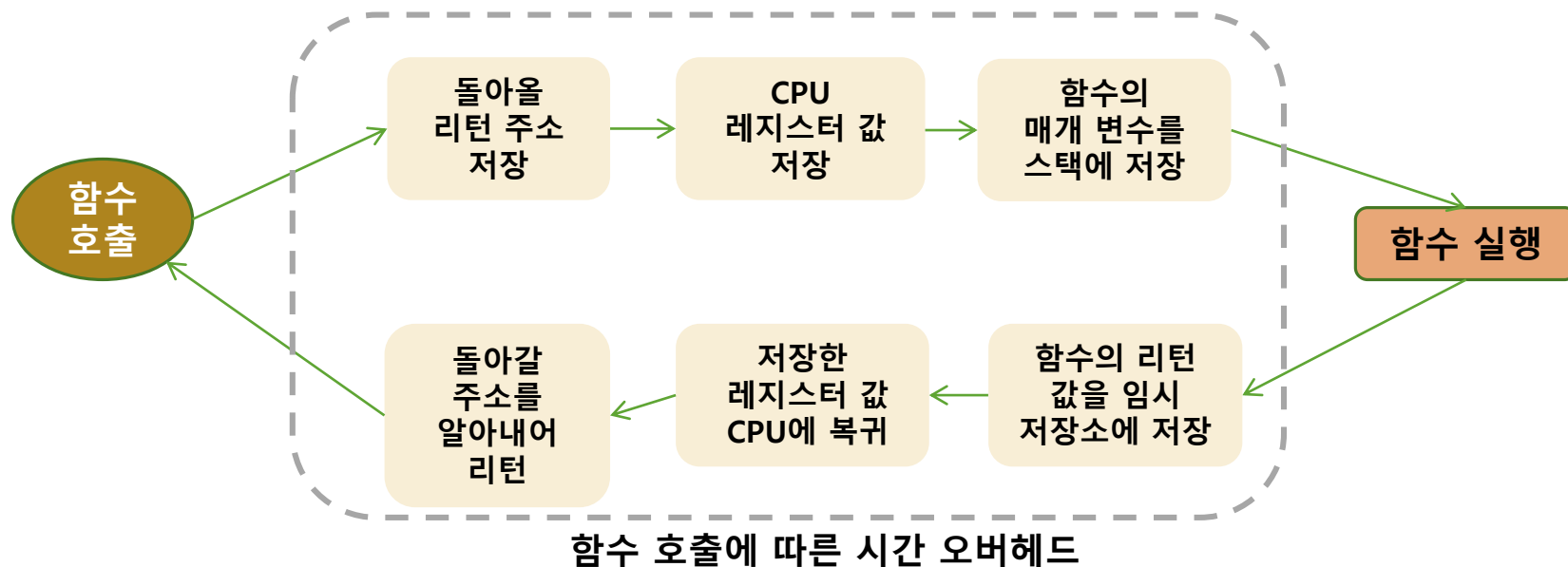
- 멤버 변수에 읽기 접근은 가능하지만 쓰기는 허용되지 않는 메소드
- 상수형 함수만 호출할 수 있다
- 선언 방법
 - 함수 원형 뒤에 **const 예약어** 사용
- 상수화 된 대상에 대한 쓰기 작업 허용
 - mutable – 멤버 변수 선언에 사용
`mutable int radius;`
 - `const_cast<>` - 참조 또는 포인터에 사용

```
void testFunc(const int &param){
    //상수형 참조가 일반 참조로 형 변환
    int &newParam = const_cast<int &>(param);
}
```

```
class Circle {
private:
    int radius; //mutable int radius (1)
public:
    Circle();
    Circle(int r);
    int getRadius() const; //상수형 함수 선언
    double getArea();
};

int Circle::getRadius() const{
    cout<<getArea(); //error, 상수형 함수만 호출
    radius = 50; //error, 멤버 변수 변경 금지
                //(1)처럼 멤버 변수를 선언하면 변경 가능
    return radius;
}
```


함수 호출에 따른 시간 오버헤드



작은 크기의 함수를 호출하면, 함수 실행 시간에 비해, 호출을 위해 소요되는 부가적인 시간 오버헤드가 상대적으로 크다.

함수 호출에 따른 오버헤드가 심각한 사례

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

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

```
int main() {
    int sum = 0;
```

```
    // 1에서 10000까지의 홀수의 합 계산
    for(int i=1; i<=10000; i++) {
        if(odd(i))
            sum += i;
    }
    cout << sum;
}
```

10000번의 함수 호출. 호출에 따른 엄청난 오버헤드 시간이 소모됨.

odd() 함수의 코드 $x\%2$ 를 계산하는 시간보다 odd() 함수 호출에 따른 오버헤드가 더 크며, 루프를 돌게 되면 오버헤드는 가중됩니다.



인라인 함수

- 인라인 함수
 - inline 키워드로 선언된 함수
- 인라인 함수에 대한 처리
 - 인라인 함수를 호출하는 곳에 인라인 함수 코드를 확장 삽입
 - 매크로와 유사
 - 코드 확장 후 인라인 함수는 사라짐
 - 인라인 함수 호출
 - 함수 호출에 따른 오버헤드 존재하지 않음
 - 프로그램의 실행 속도 개선
 - 컴파일러에 의해 이루어짐
- 인라인 함수의 목적
 - C++ 프로그램의 실행 속도 향상
 - 자주 호출되는 짧은 코드의 함수 호출에 대한 시간 소모를 줄임
 - C++에는 짧은 코드의 멤버 함수가 많기 때문

인라인 함수 사례

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

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

```
int main() {
    int sum = 0;

    for(int i=1; i<=10000; i++) {
        if(odd(i))
            sum += i;
    }
    cout << sum;
}
```

컴파일러는 inline 처리 후, 확장된 C++ 소스 파일을 컴파일 한다.

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

```
int main() {
    int sum = 0;

    for(int i=1; i<=10000; i++) {
        if(i%2)
            sum += i;
    }
    cout << sum;
}
```

컴파일러에 의해
inline 함수의 코드
확장 삽입

인라인 제약 사항

- inline은 컴파일러에게 주는 요구 메시지
- 컴파일러가 판단하여 inline 요구를 수용할 지 결정
- recursion, 긴 함수, static, 반복문, goto 문 등을 가진 함수는 수용하지 않음

인라인 함수 장단점 및 자동 인라인

- 장점
 - 프로그램의 실행 시간이 빨라진다.
- 단점
 - 인라인 함수 코드의 삽입으로 컴파일 된 전체 코드 크기 증가
 - 통계적으로 최대 30% 증가
 - 짧은 코드의 함수를 인라인으로 선언하는 것이 좋음

자동 인라인 함수

- 자동 인라인 함수 : 클래스 선언부에 구현된 멤버 함수
 - inline으로 선언할 필요 없음
 - 컴파일러에 의해 자동으로 인라인 처리
 - 생성자를 포함, 모든 함수가 자동 인라인 함수 가능

```
class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
    double getArea();
};
```

```
inline Circle::Circle() {
    radius = 1;
}
```

```
Circle::Circle(int r) {
    radius = r;
}
```

```
inline double Circle::getArea() {
    return 3.14*radius*radius;
}
```



```
class Circle {
private:
    int radius;
public:
    Circle() { // 자동 인라인 함수
        radius = 1;
    }

    Circle(int r);
    double getArea() { // 자동 인라인 함수
        return 3.14*radius*radius;
    }
};

Circle::Circle(int r) {
    radius = r;
}
```

(a) 멤버함수를 inline으로 선언하는 경우

(b) 자동 인라인 함수로 처리되는 경우

C++ 구조체

- C++ 구조체
 - 상속, 멤버, 접근 지정 등 모든 것이 클래스와 동일
 - 클래스와 유일하게 다른 점
 - 구조체의 디폴트 접근 지정 – public
 - 클래스의 디폴트 접근 지정 – private
- C++에서 구조체를 수용한 이유?
 - C 언어와의 호환성 때문
 - C의 구조체 100% 호환 수용
 - C 소스를 그대로 가져다 쓰기 위해
- 구조체 객체 생성
 - struct 키워드 생략

```
struct StructName {
  private:
    // private 멤버 선언
  protected:
    // protected 멤버 선언
  public:
    // public 멤버 선언
};
```

```
structName stObj;    // (O), C++ 구조체 객체 생성
struct structName stObj; // (X), C 언어의 구조체 객체 생성
```

Circle 클래스를 C++ 구조체를 이용하여 재작성

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

struct StructCircle { // C++ 구조체 선언
private:
    int radius;
public:
    StructCircle(int r) { radius = r; } // 구조체의 생성자
    double getArea();
};

double StructCircle::getArea() {
    return 3.14*radius*radius;
}

int main() {
    StructCircle waffle(3);
    cout << "면적은 " << waffle.getArea();
}
```


바람직한 C++ 프로그램 작성법

- 클래스를 헤더 파일과 cpp 파일로 분리하여 작성
 - 클래스마다 분리 저장
 - 인터페이스 파일 – 클래스 선언 부
 - 헤더 파일(.h)에 저장
 - 구현파일 – 클래스 구현 부
 - cpp 파일에 저장
 - 클래스가 선언된 헤더 파일 include
 - 어플리케이션 파일
 - 객체를 인스턴스화하고 객체를 활용하는 main 함수의 코드가 포함된 파일
 - 필요하면 클래스가 선언된 헤더 파일 include
- 목적
 - 클래스 재사용

```
class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
    double getArea();
};
```

Circle.h

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

#include "Circle.h"

```
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius;
    cout << " 원 생성" << endl;
}
```

```
Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius;
    cout << " 원 생성" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

반지름 1 원 생성
donut 면적은 3.14
반지름 30 원 생성
pizza 면적은 2826

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

#include "Circle.h"

```
int main() {
    Circle donut;
    double area = donut.getArea();
    cout << "donut 면적은 ";
    cout << area << endl;
}
```

```
Circle pizza(30);
area = pizza.getArea();
cout << "pizza 면적은 ";
cout << area << endl;
}
```

main.cpp

컴파일

컴파일

Circle.cpp

Circle.obj

링킹

main.exe

main.obj

헤더 파일의 중복 include 문제

- 헤더 파일을 중복 include 할 때 생기는 문제

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

```
#include "Circle.h"
```

```
#include "Circle.h" // 컴파일 오류 발생
```

```
#include "Circle.h"
```

```
int main() {
    .....
}
```

circle.h(4): error C2011: 'Circle' : 'class' 형식 재정의



헤더 파일의 중복 include 문제 해결

조건 컴파일 문의 상수(CIRCLE_H)는 다른 조건 컴파일 상수와 충돌을 피하기 위해 클래스의 이름으로 하는 것이 좋음.

권장

1) 조건 컴파일 사용

Circle.h를 여러 번 include해도 문제 없게 하기 위함

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
    double
    getArea();
};

#endif    Circle.h
```

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

```
#include "Circle.h"
#include "Circle.h"
#include "Circle.h"
```

```
int main() {
    .....
}
```

main.cpp

컴파일 오류 없음

2) #pragma once 지시자 사용

```
#pragma once
class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
};
```

include 됨

```
#ifndef CIRCLE_H
#define CIRCLE_H

// Circle 클래스 선언
class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
    double getArea();
};

#endif
```

circle.h

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

#include "circle.h"

// Circle 클래스 구현. 모든 멤버 함수를 작성한다.
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성\n";
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성\n";
}

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

circle.cpp

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

#include "circle.h"

int main() {
    Circle donut;
    double area = donut.getArea();
    cout << " donut의 면적은 " << area << "\n";

    Circle pizza(30);
    area = pizza.getArea();
    cout << "pizza의 면적은 " << area << "\n";
}
```

main.cpp

컴파일

circle.obj

main.obj

링킹

main.exe

헤더 파일과 cpp 파일로 분리하기

- 아래의 소스를 헤더 파일과 cpp 파일로 분리하여 재 작성하라

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

class Adder { // 덧셈 모듈 클래스
    int op1, op2;
public:
    Adder(int a, int b);
    int process();
};

Adder::Adder(int a, int b) {
    op1 = a; op2 = b;
}

int Adder::process() {
    return op1 + op2;
}
```

```
class Calculator { // 계산기 클래스
public:
    void run();
};

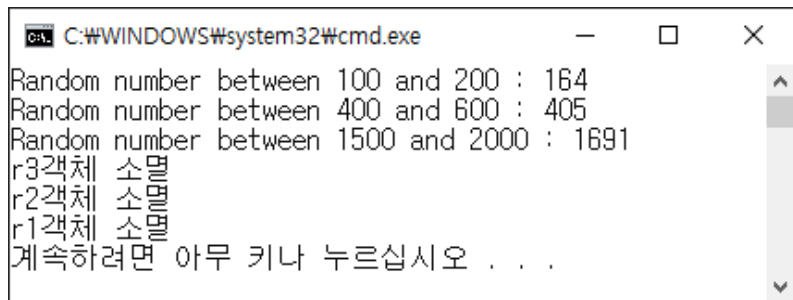
void Calculator::run() {
    cout << "두 개의 수를 입력하세요>>";
    int a, b;
    cin >> a >> b; // 정수 두 개 입력
    Adder adder(a, b); // 덧셈기 생성
    cout << adder.process(); // 덧셈 계산
}

int main() {
    Calculator calc; // calc 객체 생성
    calc.run(); // 계산기 시작
}
```

두 개의 수를 입력하세요>>5 -20
-15

실습1. 클래스와 객체 활용 - 난수 생성하기

- 주어진 큰 값과 작은 값 범위 내의 난수 생성하기
 - 범위를 객체 생성시 매개변수로 전달- 매개변수 있는 생성자 필요
 - 난수 생성 결과 출력하는 멤버 함수 - 변경 작업이 필요 없으므로 const



```

C:\WINDOWS\system32\cmd.exe
Random number between 100 and 200 : 164
Random number between 400 and 600 : 405
Random number between 1500 and 2000 : 1691
r3객체 소멸
r2객체 소멸
r1객체 소멸
계속하려면 아무 키나 누르십시오 . . .
  
```

//C++ 난수 생성

#include <random>

random_device rd; //시드값을 얻기 위한 random_device 생성

mt19937 gen(rd()); //random_device를 통해 난수 생성 엔진 초기화

uniform_int_distribution<int> dis(low, high); //low~high 사이의 난수 분포 정의

Int value = dis(gen); //난수 엔진을 전달하여 범위내 생성된 난수를 value에 저장

```

int main()
{
    RandomInteger r1{ 100, 200, "r1" };
    r1.print();

    RandomInteger r2(400, 600, "r2");
    r2.print();

    RandomInteger r3(1500, 2000, "r3");
    r3.print();

    return 0;
}
  
```

실습2. 클래스와 객체 활용 - 계좌 관리

- 제시된 main()소스코드를 참조하여 계좌 관리를 위한 클래스 Account를 구현하시오.
 - 클래스 구현과 선언을 분리하여 작성하시오

```

C:\WINDOWS\system32\cmd.exe
menu : 1. 입금, 2. 출금 3. 조회 >> 1
입금액 >> 20000
C++의 입금 액은 20000
C++의 잔액은 70000
2021: 객체 소멸
계속하려면 아무 키나 누르십시오 . . .

```

```

C:\WINDOWS\system32\cmd.exe
menu : 1. 입금, 2. 출금 3. 조회 >> 2
출금액 >> 20000
C++의 출금 액은 20000
C++의 잔액은 30000
2021: 객체 소멸
계속하려면 아무 키나 누르십시오 . . .

```

```

C:\WINDOWS\system32\cmd.exe
menu : 1. 입금, 2. 출금 3. 조회 >> 3
C++의 잔액은 50000
2021: 객체 소멸
계속하려면 아무 키나 누르십시오 . . .

```

```

int main() {
    Account a("C++", 2021, 50000);
    int menu, money;
    cout << "menu : 1. 입금, 2. 출금 3. 조회 >> ";
    cin >> menu;

    switch (menu) {
        case MENU::DEPOSIT:
            cout << "입금액 >> ";
            cin >> money;
            a.deposit(money);
            cout << a.getOwner() << "의 입금 액은 " << money << endl;
            cout << a.getOwner() << "의 잔액은 " << a.inquiry() << endl;
            break;
        case MENU::WITHDRAW :
            cout << "출금액 >> ";
            cin >> money;
            cout << a.getOwner() << "의 출금 액은 " << a.withdraw(money) << endl;
            cout << a.getOwner() << "의 잔액은 " << a.inquiry() << endl;
            break;
        case MENU::INQI:
            cout << a.getOwner() << "의 잔액은 " << a.inquiry() << endl;
    }
    return 0;
}

```


Q & A

- "C++ 클래스와 객체"에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 3주 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 수고하셨습니다.^^