객체지향설계
Fall 2018

임성수 교수

Lecture 2: 클래스

수업 내용

1. 구조체와 클래스 

2. 생성자와 소멸자

3. 상수화

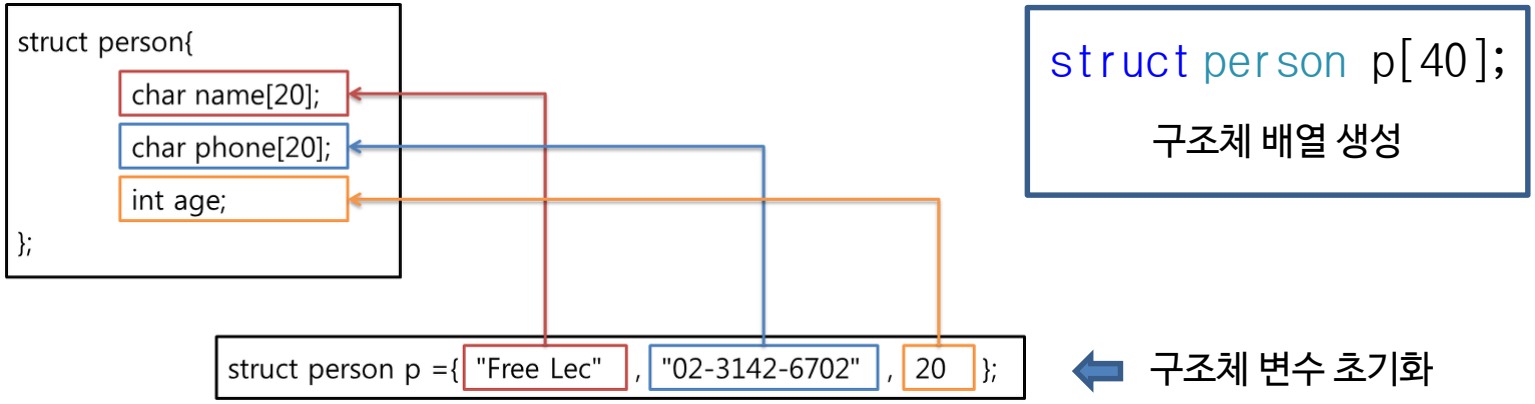
집합적 데이터 형

배열(Array)

- 같은 종류의 변수를 묶어서 정리한 구조

구조체(Struct)

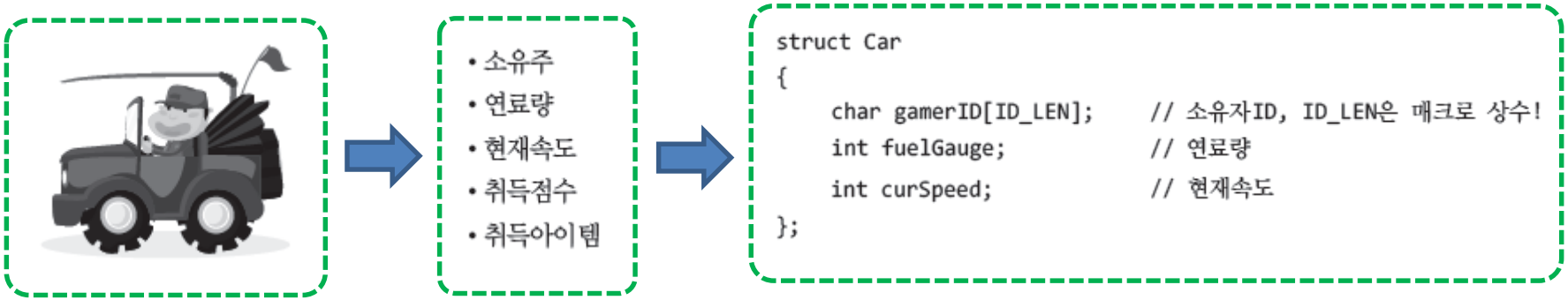
- 여러 종류의 **변수**를 묶어서 정리한 구조
- 구조체 배열 정의 가능



구조체

배경

- 연관 있는 데이터를 하나로 묶으면 프로그램 구현 및 관리가 용이
- 자료형으로 묶어서 관리하면 연관 있는 데이터의 생성 및 소멸 시점, 이동 및 전달 시점과 방법이 일치하게 할 수 있음



구조체와 클래스

구조체

- 클래스의 기초가 됨: 프로그램 구현 및 관리를 용이하게 함
- 멤버 변수(property) 선언 가능
- 접근 제어: 구조체는 public, 클래스는 별도 지정 가능

클래스

- 구조체의 상위 호환: **함수**를 표현할 수 있는 **구조체**
- 멤버 변수, 멤버 함수(method) 선언 가능
- 접근 제어: 별도 지정하지 않으면 private 적용
- 클래스를 통해 **객체지향 프로그램** 작성 가능

객체, 멤버 변수, 멤버함수

객체: 클래스를 통해 생성된 인스턴스(instance)

멤버 변수: 클래스 내에 선언된 변수, 객체마다 값이 다름

멤버 함수: 클래스 내에 정의된 함수

클래스(class)
- class Animal {…};

객체(object)
- Animal cat;

멤버 변수(member variable)
- cat.name = "나비"
- cat.family = "코리안 숏 헤어"
- cat.age = 1
- cat.weight = 0.1

멤버 함수(member function)
- cat.Mew()
- cat.Eat()
- cat.Sleep()
- cat.Play()

구조체

선언

- 사용자가 새로운 형을 정의하고, 새로운 형의 변수를 선언 가능
- 멤버 변수: 구조체를 구성하는 변수

키워드 ➡ `struct USERDATA`

멤버 변수

```
#include "pch.h"
#include <iostream>
using namespace std;

struct USERDATA
{
    int nAge;
    char szName[32];
};
```

구조체

초기화 및 접근

- 초기화: 중괄호를 사용한 초기화 리스트를 사용하여 초기화
- 접근: 멤버 참조 연산자(.)를 사용하여 접근

```
int main(void)
{
    USERDATA user = {67,"김충남"};
    cout << user.nAge << " " << user.szName;
return 0;
}
```

사용자가 구조체의 멤버 및 구성을 알아야 함
제작자가 구조체를 변경할 시 사용자 코드도 수정 필요

실행 결과

```
67 김충남
```

구조체

인터페이스 함수

- 인터페이스: 사용자가 구조체의 멤버 및 구성을 몰라도 됨
- 인터페이스 함수: 제작자가 따로 완성된 함수를 제공

```
void PrintData(USERDATA *pUser)
{
    cout << pUser->nAge << " " << pUser->szName;
}

int main(void)
{
    USERDATA user = {67,"김충남"};
    PrintData(&user);

    return 0;
}
```

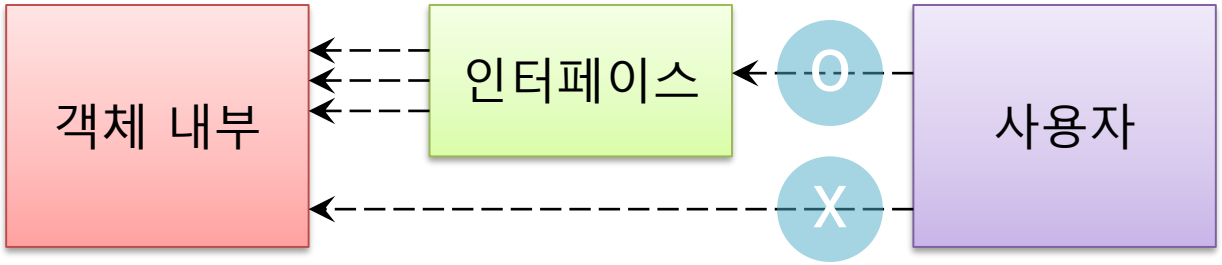
실행 결과

```
67 김충남
```

인터페이스

객체지향 패러다임

- 제작자가 구조체를 변경해도 사용자가 일일이 알 필요 없음
- 내부 구조, 관계를 몰라도 인터페이스 함수를 통해서 연결
- **구조체 + 함수 → 클래스**



클래스

선언: 접근 제어 지시자(public/private/protected), 멤버 함수가 추가됨

키워드 클래스 이름

```
class Userdata
{
public:
    int nAge;
    char szName[32];

    void Print(void)
    {
        cout << user.nAge << " " << user.szName;
    }
};
```

접근 제어

멤버 변수

멤버 함수

클래스

초기화 및 접근

- 멤버 변수: 다양한 초기화 방식
  - 생성자 함수를 이용한 초기화 가능
- 멤버 함수: 클래스의 멤버 함수를 사용

```
int main()
{
    Userdata user = {67, "김충남"};
    user.Print();

    return 0;
}
```

실행 결과

```
67 김충남
```

멤버 변수 초기화

멤버 초기화 리스트(Member Initialization List)를 활용한 멤버 변수 초기화

```
class Userdata
{
public:
    Userdata()
        : nAge(67), szName("김충남")
    {}

    int nAge;
    char szName[32];

    void Print(void)
    {
        cout << nAge << " " << szName;
    }
};
```

생성자 초기화 목록
함수 원형 다음에 콜론(:) 기입 후
멤버 변수(초기값) 형식으로 기술

```
int main()
{
Userdata user;
user.Print();

return 0;
}
```

접근 제어 지시자

접근 제어

- 제작자가 만든 클래스의 특정 요소에 사용자가 접근할 수 없게 제어
- 별도로 언급하지 않는다면 기본적으로 private 적용

지시자	설명
public	멤버에 관한 모든 외부 접근 허용
protected	멤버에 관한 모든 외부 접근 차단 상속 관계에 있는 서브클래스의 접근은 허용
private	멤버에 관한 모든 외부 접근 차단 상속 관계에 있는 서브클래스의 접근도 차단

수업 내용

1. 구조체와 클래스
2. 생성자와 소멸자 
3. 상수화

생성자와 소멸자

정의 및 특징

- 클래스의 인스턴스(객체)가 생성/소멸될 때 자동으로 호출되는 함수
  - 생성: 메모리 공간 할당 → **생성자** 호출
  - 소멸: **소멸자** 호출 → 메모리 공간 반환
- 기본(Default) 생성/소멸자: 명시하지 않아도 생성/소멸 시 자동 호출
- 생성자는 중복 선언 가능
- 리턴을 선언하지 않음
- 클래스 이름과 함수 이름이 동일
  - 클래스: class Ctest { … };
  - 생성자: Ctest();
  - 소멸자: ~Ctest();

```
class Userdata
{
public:
    int nAge;
    char szName[32];
    Userdata() {}; // 자동으로 생성
};
```

생성자(Constructor)

생성자, 소멸자 선언의 예

```
#include "pch.h"
#include <iostream>
using namespace std;

class CTest
{
public:
    CTest()
    {
        cout << "생성자 호출" << endl;
    }
    ~CTest()
    {
        cout << "소멸자 호출" << endl;
    }
};
```

```
int main()
{
    cout << "Begin" << endl;
    CTest a;
    cout << "End" << endl;

return 0;
}
```

실행 결과

```
Begin
생성자 호출
End
소멸자 호출
```

실행 순서

C: main()함수가 가장 먼저 호출

C++: **전역 변수로 선언한 클래스의 생성자**가
main() 함수보다 먼저 호출

```
int main()
{
    cout << "Begin" << endl;
    CTest a;
    cout << "End" << endl;

return 0;
}
```

```
CTest a;

int main()
{
    cout << "Begin" << endl;
    cout << "End" << endl;

return 0;
}
```

실행 결과

Begin
생성자 호출
End
소멸자 호출

생성자 호출
Begin
End
소멸자 호출

소멸자(Destructor)

정의

- 객체 소멸 시 메모리 반환을 위하여 자동 호출되는 함수

객체 소멸

- 선언된 블록 범위가 끝나면 자동 소멸

```
int main()
{
    cout << "Begin" << endl;
    CTest a;
    cout << "End" << endl;

return 0;
}
```

객체(a)가 선언된 블록 범위 수행 후 소멸자를 통해 소멸

실행 결과

```
Begin
생성자 호출
End
소멸자 호출
```

실행 순서

main() 함수가 호출되기 전 생성자 호출 가능
main() 함수가 종료된 후 소멸자 호출
main() 함수의 마지막 구문 실행 후 내부 선언된 지역 변수들 소멸

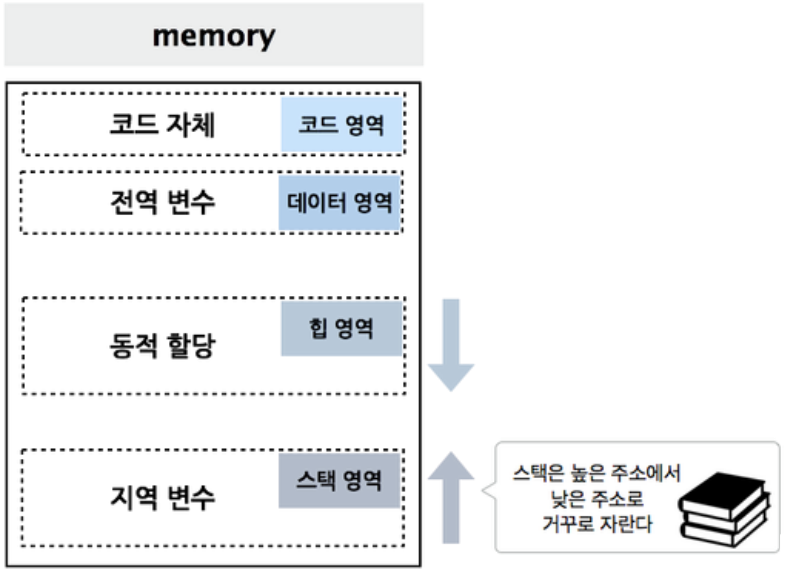
```
int main()
{
    cout << "Begin" << endl;
    CTest a;
    cout << "Test" << endl;
    CTest b;
    cout << "End" << endl;

    return 0;
}
```

main() 함수
끝까지 실행 후
지역 변수들 소멸

실행 결과

```
Begin
생성자 호출
Test
생성자 호출
End
소멸자 호출
소멸자 호출
```

메모리 할당



메모리

- 데이터와 명령어를 저장하는 공간

정적(static) 할당

- 프로그램 실행 시 미리 메모리를 할당 받고 종료 후 회수
- 메모리 크기가 고정되어 조절이 어려움

동적(dynamic) 할당

- 프로그램 실행 동안 요청하여 메모리 공간 할당 및 해제
- 경제적이고 조절이 쉽지만 따로 해제해야 함

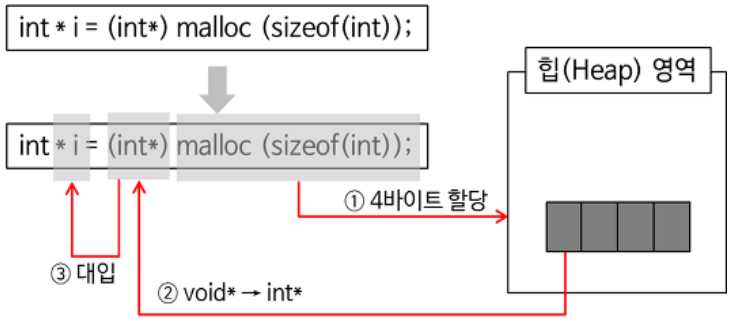
동적 할당

C 프로그램

- 함수 사용: malloc() / free()
- 할당하려는 메모리 크기를 알아야 함
  - 타입 *변수이름 = (타입*)malloc(sizeof(타입)*메모리크기)
  - 예) int *i = (int *)malloc(sizeof(int));

C++ 프로그램

- 함수 대신 연산자 사용: new / delete
- 복잡한 함수 사용보다 편의성 높음
- 할당하려는 메모리 크기를 몰라도 됨
  - 타입 *변수이름 = new 타입
  - 예) int *i = new int;



객체의 동적 할당

동적 객체의 생성과 소멸

- new/delete 연산자를 통해 생성자/소멸자 호출
- 객체가 생성 및 소멸하는 시점을 코드에서 명확히 파악 가능

객체 생성 ➡

객체 삭제 ➡

```
int main()
{
    cout << "Begin" << endl;
    CTest *pData = new CTest;
    cout << "Test" << endl;
    delete pData;
    cout << "End" << endl;

    return 0;
}
```

실행 결과

```
Begin
생성자 호출
Test
소멸자 호출
End
```

동적 객체의 소멸

동적 객체의 생성과 소멸

- 배열로 생성한 객체는 배열로 삭제함
- 그렇지 않으면 첫 요소 소멸 후 메모리 누수 버그 발생

```
int main()
{
cout << "Begin" << endl;
CTest *pData = new CTest[3];
delete [] pData;
cout << "End" << endl;

return 0;
}
```

```
int main()
{
cout << "Begin" << endl;
CTest *pData = new CTest[3];
delete pData;
cout << "End" << endl;

return 0;
}
```

실행 결과

```
Begin
생성자 호출
생성자 호출
생성자 호출
소멸자 호출
소멸자 호출
소멸자 호출
End
```

```
Begin
생성자 호출
생성자 호출
생성자 호출
소멸자 호출
```

Memory Leak!

참조자(Reference)

정의

- C에는 없는 형식으로 포인터와 구조적으로 유사
- 포인터를 잘못 사용해서 생기는 문제들을 줄여줌
- 선언과 동시에 초기화
  - 타입 &변수이름 = 원본;
  - 예) int &m_Ref = a;

포인터 vs 참조자

- 포인터: NULL을 허용 / 참조자: NULL이 될 수 없음
- 포인터: 주소값을 할당 / 참조자: 참조 대상을 그대로 할당
- 포인터: 참조 대상 변경 가능 / 참조자: 참조 대상 변경 불가

참조자 멤버 변수 초기화

참조 형식의 멤버 변수

- 참조자는 반드시 선언과 동시에 초기화
- 생성자 초기화 목록을 이용하여 초기화

참조형 멤버는 생성자 초기화 목록으로 초기화 ➡

참조형 멤버는 객체 생성 시 초기화 ➡

```
class RefTest
{
public:
    RefTest(int &rParam) : m_Ref(rParam) {}
    int GetData(void)
    {
        return m_Ref;
    }

private:
    int &m_Ref;
};
```

참조자 멤버 변수 초기화

포인터와 달리 참조 대상 변경 불가

참조 대상의 원본 값 수정은 가능

```
int main()
{
int a = 10;
RefTest t(a);
cout << t.GetData() << endl;

a = 20;
cout << t.GetData() << endl;

return 0;
}
```

참조 대상 a의
원본 값 수정

실행 결과

```
10
20
```

생성자 오버로딩

생성자 다중 정의

- 생성자 다중 정의를 통해 사용자 코드 간편화 가능
- 컴파일러가 매개변수를 보고 어떤 생성자를 호출할지 결정

```
class OverTest
{
public:
   OverTest(int x) : m_nData(x) {};
   OverTest(int y, int z) : m_nData(y + z) {};
   int GetData(void)
   {
      return m_nData;
   }

   private:
   int m_nData;
};
```

```
int main()
{
   OverTest a(10);
   OverTest b(3, 4);

   cout << a.GetData() << endl;
   cout << b.GetData() << endl;

   return 0;
}
```

실행 결과

```
10
7
```

수업 내용

1. 구조체와 클래스
2. 생성자와 소멸자
3. 상수화 

상수화

멤버 변수/함수의 상수화

- 키워드 const를 앞에 붙이면 값과 주소를 변경할 수 없게 함
- 상수화된 멤버 함수는 멤버 변수에 읽기 접근은 가능하지만 멤버 변수에 대한 대입 연산자, 단항 연산자 등 사용 불가

```
class ConstTest
{
private:
    const int t;
public:
    ConstTest(int a) {
        t = a;
    }
};
```

변수 t 값을 변경 불가
컴파일 에러 발생

Compile Error!

상수화

목적

- 상수형 메소드는 멤버 변수의 값을 수정할 수 없고,
  상수형 메소드가 아닌 멤버는 호출 불가
- 여러 처리 조건을 한번에 바꿀 수 있으므로 **유지보수에 유리**
  - 나중에 변경될 가능성이 있는 조건을 상수화
- 중간에 변경이 불가능하기 때문에 **오류 방지에 유리**
  - 프로그래밍 시 바뀌면 안 되는 값이 있을 때 활용

멤버 변수의 상수화

멤버 초기화 리스트(Member Initialization List)를 통한 초기화 가능

```
class ConstTest
{
private:
   const int t;
public:
   ConstTest(int a) : t(a) {};
   void Print(ConstTest *pData)
   {
      cout << t << endl; }
};

int main()
{
   ConstTest T(10);
   T.Print(&T);
}
```

실행 결과

```
10
```

멤버 함수의 상수화

상수형 메소드: 멤버 변수의 값을 읽을 수 있지만 쓸 수는 없음

```
class CTest
{
public:
    CTest(int nParam) : m_nData(nParam) {};
    ~CTest() {}

    int GetData() const
    {
        return m_nData;
    }
    int SetData(int nParam)
    {
        m_nData = nParam;
    }
private:
    int m_nData = 0;
};
```

상수형 메소드 선언

```
int main()
{
    CTest a(10);
    cout << a.GetData() << endl;

    return 0;
}
```

실행 결과

```
10
```

멤버 함수의 상수화

상수형 메소드: 멤버 변수의 값을 읽을 수 있지만 쓸 수는 없음

- const 예약어를 사용 시 메소드를 안정적으로 활용 가능

```
int GetData()
{
    m_nData = 20;
    return m_nData;
}
```

실행 결과

```
20
```

```
int GetData() const
{
    m_nData = 20;
    return m_nData;
}
```

Compile Error!

향후 일정

3주차: Chap. 1

4주차: Chap. 2

질문 및 답변

참고자료

객체지향 프로그래밍의 이해

객체에 대한 간단한 정의

- 사전적 의미 — 물건 또는 대상
- 객체지향 프로그래밍 — 객체 중심의 프로그래밍

객체지향 프로그래밍에서는 나, 과일장수, 사과라는 객체를 등장시켜서 두 개의 사과 구매라는 행위를 실체화한다.

객체지향 프로그래밍은 현실에 존재하는 사물과 대상, 그리고 그에 따른 행동을 있는 그대로 실체화 시키는 형태의 프로그래밍이다.

출처: 윤성우의 C++ 프로그래밍 강의자료

참고자료

객체를 이루는 것은 데이터와 기능입니다.

과일장수 객체의 표현

- 과일장수는 과일을 팝니다. 행위
- 과일장수는 사과 20개, 오렌지 10개를 보유하고 있습니다. 상태
- 과일장수의 과일판매 수익은 현재까지 50,000원입니다. 상태

과일장수의 데이터 표현

- 보유하고 있는 사과의 수 → int numOfApples;
- 판매 수익 → int myMoney;

과일장수의 행위 표현

```
int SaleApples(int money)      // 사과 구매액이 함수의 인자로 전달
{
     int num = money/1000;     // 사과가 개당 1000원이라고 가정
     numOfApples -= num;       // 사과의 수가 줄어들고,
     myMoney += money;         // 판매 수익이 발생한다.
     return num;               // 실제 구매가 발생한 사과의 수를 반환
}
```

이제 남은 것은 데이터와 행위를 한데 묶는 것!

출처: 윤성우의 C++ 프로그래밍 강의자료

참고자료

'과일장수'의 정의와 멤버변수의 상수화

```
class FruitSeller
{
private:
    int APPLE_PRICE;
    int numOfApples;
    int myMoney;

public:
    int SaleApples(int money)
    {
        int num=money/APPLE_PRICE;
        numOfApples-=num;
        myMoney+=money;
        return num;
    }
};
```

변수 선언

함수 정의

과일 값은 변하지 않는다고 가정할 때
APPLE_PRICE는 다음과 같이 선언하는 것이 좋다!
const int APPLE_PRICE;

그러나 상수는 선언과 동시에 초기화 되어야 하기 때문에 이는 불가능하다. 물론 클래스를 정의하는 과정에서 선언과 동시에 초기화는 불가능하다.

추가

```
void InitMembers(int price, int num, int money)
{
    APPLE_PRICE=price;
    numOfApples=num;
    myMoney=money;
}
```

초기화를 위한 추가

추가

```
void ShowSalesResult()
{
    cout<<"남은 사과: "<<numOfApples<<endl;
    cout<<"판매 수익: "<<myMoney<<endl;
}
```

얼마나 파셨어요? 라는 질문과 답변을 위한 함수

출처: 윤성우의 C++ 프로그래밍 강의자료

참고자료 '나(me)'를 표현하는 클래스의 정의와 객체생성

'나'의 클래스 정의

```
class FruitBuyer
{
    int myMoney;       // private:
    int numOfApples;   // private:
public:
    void InitMembers(int money)
    {
        myMoney=money;
        numOfApples=0;     // 사과구매 이전이므로!
    }
    void BuyApples(FruitSeller &seller, int money)
    {
        numOfApples+=seller.SaleApples(money);
        myMoney-=money;
    }
    void ShowBuyResult()
    {
        cout<<"현재 잔액: "<<myMoney<<endl;
        cout<<"사과 개수: "<<numOfApples<<endl;
    }
};
```

상태

행위

일반적인 변수 선언 방식의 객체생성

```
FruitSeller seller;

FruitBuyer buyer;
```

동적 할당 방식의 객체생성

```
FruitSeller * objPtr1=new FruitSeller;

FruitBuyer * objPtr2=new FruitBuyer;
```

출처: 윤성우의 C++ 프로그래밍 강의자료

참고자료

사과장수 시뮬레이션 완료

```
int main(void)
{
    FruitSeller seller;
    seller.InitMembers(1000, 20, 0);
    FruitBuyer buyer;
    buyer.InitMembers(5000);
    buyer.BuyApples(seller, 2000);

    cout<<"과일 판매자의 현황"<<endl;
    seller.ShowSalesResult();
    cout<<"과일 구매자의 현황"<<endl;
    buyer.ShowBuyResult();
    return 0;
}
```

아저씨 사과 2000원어치 주세요.

아저씨 오늘 얼마나 파셨어요.. 라는 질문의 대답

너 사과 심부름 하고 나머지 잔돈이 얼마야.. 라는 질문의 대답

```
void BuyApples(FruitSeller &seller, int money)
{
    numOfApples+=seller.SaleApples(money);
    myMoney-=money;
}
```

FruitBuyer 객체가 FruitSeller 객체의 SaleApples 함수를 호출하고 있다. 그리고 객체지향에서는 이것을 '두 객체가 대화하는 것'으로 본다. 따라서 이러한 형태의 함수호출을 가리켜 '메시지 전달'이라 한다.

출처: 윤성우의 C++ 프로그래밍 강의자료