Part

01

기초 다지기

Chapter

01

객체지향 방법론

*** 학습목표**

- 프로그램 설계 차원에서 객체지향 방법론의 근본적인 개념을 이해한다.
- C++의 객체지향적 요소를 파악한다.
- 객체지향 설계에서 인터페이스와 구현을 분리하는 이유를 명확히 이해한다.
- 절차적 방법론과 객체지향 방법론의 근본적인 차이를 이해한다.

Point

첫 주제로 객체지향 프로그래밍(Object-Oriented Programming)을 제시하는 이유는 연관성 때문이다. 자료구조 교과에서 중요시되는, 이른바 추상 자료형 개념과 객체지향 프로그래밍의 클래스 개념은 완벽히 일치한다. 그리고 자바나 C++ 등 객체지향 언어로 쓰인 자료구조 서적 대부분은 이러한 추상 자료형을 강조하기 위해 쓰인 것이기도 하다.

그러나 이처럼 객체지향 언어만 사용할 경우, 해당 언어의 난이도에서 문제가 발생한다. 절차적 언어로 간단히 설명될 수 있는 부분까지도 복잡한 객체지향 언어 문법으로 설명해야만 하기 때문이다. 자료구조를 설명하기 위해 사용된 언어가 너무 복잡하여, 자료구조보다 오히려 그 언어 자체의 특성을 학습하는 많은 시간을 소모하는 것은 결코 바람직하지 않다.

이 책에서는 필요에 따라 C와 C++를 혼용하기로 한다. 객체지향 언어인 C++는 주로 추상 자료형 개념을 표현하는 데 사용하며, 절차적 언어인 C는 그러한 개념을 구체적으로 구현하는 데 사용한다. 단언컨대, C++ 언어란 그리 어려운 것이 아니다. 문법적으로는 C와 거의 유사하다. 단지, C++ 언어가 나오게 된 배경, 즉 객체지향 개념만 이해하면 된다. 더구나 이 책에서 사용하는 C++는 자료구조를 설명하기에 충분할 만큼의 문법적 요소만 제시한다. 이 책의 목적이 C++ 언어를 학습하는 데 있지 않기 때문이다. 이러한 점을 감안하여 독자는 두 가지 언어를 모두 이해해야 할 것이다.

자바, C++ 등 객체지향 언어는 이미 많은 프로그래머가 애용하는 언어가 된지 오래다. 이러한 객체지향 언어를 사용하여 프로그래밍을 하려면 무엇보다 객체지향 개념을 제대로 인식하는 것이 중요하다. 프로그램 도구로써, 객체지향 언어를 사용하고 있더라도 그 안에 객체라는 개념이 녹아있지 않을 때는 무의미하다. 문법적으로는 객체지향 언어일지 몰라도 프로그램 설계 자체가 객체지향이 아니기 때문이다.

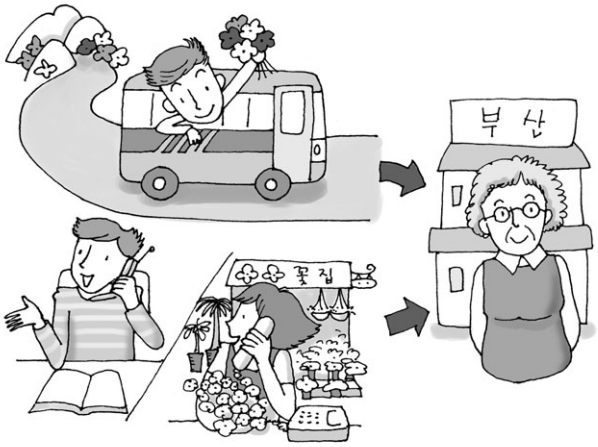
객체지향이란 일종의 프로그램 설계기법이다. 프로그램 언어가 아니다. 이 장에서는 객체지향의 기본적인 개념을 설명하고, 이러한 개념을 수용하여 프로그래밍 할 수 있게 하는 도구로써, C++ 언어의 기본 문법을 제시하기로 한다.

Section

01 객체지향 개념

"부산에 사시는 할머니께 꽃을 보내고 싶다"는 문제를 생각해보자. 물론, 이 문제를 해결하기 위한 수많은 방법이 존재할 것이다.

예를 들어 보자. 일단 주머니 사정을 확인한다. 만약, 여유가 있으면 인근 꽃가게에 전화해서 꽃배달 서비스를 신청한다. 그렇지 않으면 수첩을 뒤져 돈을 빌릴 사람을 찾아본다. 빌릴 사람이 없으면 운동화를 꺼내 신고 인근 야산으로 향한다. 원하는 꽃을 구한 다음 포장센터로 간다. 그리고 강남 고속버스 터미널로 가서 고속버스를 탄다. 고속버스에서 내려, 할머니 동네로 가는 마을버스를 탄다.



[그림 1-1] 부산에 사시는 할머니께 꽃을 보내는 방법

이처럼 어떤 문제를 해결하기 위한 과정을 순차적으로 설계하는 것을 **절차적 방법론**(Procedural Method)이라 한다. 이러한 방법론을 수용하는 **절차적 언어**(Procedural Language)에는 기존의 프로그램 언어인 파스칼(Pascal), 포트란(FORTRAN), C 등이 있다. 이들 언어에서는 어떤 문제를 해결하기 위한 모든 과정을 일일이 순차적인 명령어(Statement)로 나타내야 한다.

절차적 방법론
(Procedural Method)

절차적 언어
(Procedural Language)

객체지향적 방법론(Object-Oriented Method)

다른 사람(객체, Object)

대리인(Agent)

요구사항(Request)

책임(Responsibility)

위 문제에 대한 **객체지향적 방법론**(Object-Oriented Method)은 꽃집주인에게 전화를 걸어서 "무슨 꽃, 몇 송이를, 어디 주소로 보내 달라"고 하는 것이다. 일을 해야 할 내(주체, Subject)가 그 일을 **다른 사람**(객체, Object)에게 떠넘기는 것이다. 즉, 꽃집주인(예를 들어, 김꽃분)이라는 적당한 **대리인**(Agent)을 찾아서 "무슨 꽃, 몇 송이를, 어디 주소로 보내 달라"는 **요구사항**(Request)을 담은 메시지를 전해주면(Message Passing), 나머지 일은 김꽃분의 **책임**(Responsibility)이 된다. 다시 말해, 김꽃분은 내 일을 대신할 대리인이자 객체로서 내가 전한 메시지를 받는 사람(Receiver)이 된다.

김꽃분은 내 메시지를 받고서는 여러 방법(Method)을 동원해서 자신의 책임을 다할 것이다. 내 입장으로서는 그녀가 어떤 방법을 사용하는지 전혀 모를(Information Hiding, 정보 은닉)뿐만 아니라 알 필요도 없다. 그녀가 꽃을 직접 포장할지, 택배로 보낼지, 아니면 부산에 있는 꽃집주인에게 전화를 걸어서 책임을 다시 전가할지는 내가 신경쓰지 않아도 되는 부분이다. 김꽃분이 일처리를 어떻게 하는지, 방법에 관해 내가 일일이 신경 쓰려면 직접 하지, 뭐 하러 김꽃분에게 일을 시키겠는가!

객체지향 개념은 이처럼 단순하다. 내가 모든 것을 일일이 다 처리해 버리는 주체지향(主體指向)이 아니라 남을 시켜서 처리한다는 객체지향(客體指向)이다.

[그림 1-2] 코끼리를 냉장고에 넣는 방법

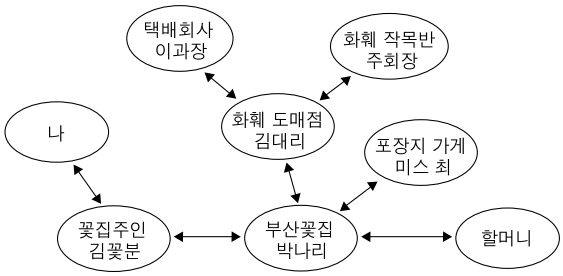
코끼리를 냉장고에 넣는 방법 중 하나가 바로 "조교에게 시킨다"다. 조교가 어떻게 할지 모를 뿐만 아니라 알고 싶지도 않지만, 어쨌든 "조교가 알아서 하라"는 것은 객체지향적 발언이다. 다시 말해, 객체지향 개념은 모든 걸 혼자 다 처리하려 들지 말고 다른 객체의 서비스

를 이용하라는 것이다. 그렇지만 처음부터 끝까지 모든 과정을 일일이 내가 알아서 프로그래밍해야 하는, 기존의 절차지향적 방식에 익숙한 사람에게는 이러한 프로그램 설계 방식이 일종의 어려움으로 느껴질 수도 있다.

프로그램으로 말하자면 객체지향적 설계는 일단 객체를 설정하는 것에서 출발한다. 예를 들어, 비즈니스 프로그램이라면 수금관리, 인사관리, 재고관리 등의 객체를 설정할 수 있다. 따라서 메인 함수에서는 "야, 수금관리 객체야! 지난 달 수금액 합계가 얼마나 되니?"라는 메시지를 던지고, 결과가 되돌아오면 이를 다시 인사관리 객체에게 던지면서 "야, 인사관리 객체야! 수금 실적이 좋으니, 이번 달 보너스는 100% 지급해"라는 메시지(실제로는 프로그램 명령문)를 던지는 식이다.

만약, 이것이 규모가 큰 프로젝트라면 객체별로 여러 프로그래머가 나누어 짜게 된다. 따라서 내가 만약 메인 함수를 작성한다면 수금관리 객체는 다른 사람이 작성할 수도 있다. 이 경우, 나는 수금관리 객체가 내부적으로 어떤 방법을 사용하는지 알 필요도 없고 알아서도 안 된다. 단지 메시지를 던져 일만 시키면 된다. 그러나 여기서 객체지향이라는 것은 "누가 프로그램을 작성하는가"의 문제가 아니다. 즉, "내가 작성하는가, 남이 작성하는가"의 의미가 아니다. 즉, 모든 프로그램을 나 혼자 짜더라도 전체적인 프로그램을 설계할 때 수금관리, 인사관리, 재고관리라는 개념적인 객체를 설정하였는가, 아닌가의 문제다. 당연히 절차적 설계기법에서는 이처럼 객체를 설정할 필요가 없다.

객체지향 방법론이 바라보는 세상은 온통 객체들만으로 구성되어 있다. 그리고 그 객체 사이를 부지런히 오가는 것은 메시지다. 각각의 객체는 외부로부터 메시지가 들어올 때만 응답하되, 자신만이 아는 방식으로 메시지를 처리한다. 인근 꽃집주인이 부산에 있는 다른 꽃집주인에게 부탁할 수 있듯이, 어떤 객체가 메시지를 처리하는 과정에서 필요하다면 또 다른 객체에게 메시지를 전달해서 일을 시킬 수도 있다.



[그림 1-3] 객체지향의 메시지 전달

[그림 1-3]에서 꽃집주인 김꽃분이 부산꽃집의 박나리에게 메시지를 던져 일을 시키면, 부산꽃집 박나리는 다시 화훼 도매점 김대리에게 자신의 가게로 생화를 갖다달라는 메시지를 전달할 수도 있다. 김대리 역시 회훼 작목반의 주회장과 택배회사 이과장에게 메시지를 전달해서 일을 시킬 수 있다. 일을 시킬 때에는 그 실행 결과를 돌려 줄 것을 요구할 수도 있지만 그렇지 않고 단지 일을 시키기만 할 수도 있다. 어찌 되었든 우리가 일상 생활에서 겪는 일 대부분이 타인의 서비스를 요구하는 부분이라는 측면에서 객체지향적 사고방식은 우리가 생각하는 방식과도 일치한다.

시뮬레이션 (Simulation, 모의실험)

객체지향 설계기법이 많이 사용되는 것이 **시뮬레이션(Simulation, 모의실험)** 프로그램이다. 예를 들어, 전쟁 시뮬레이션(War Game)은 아군과 적군 간에 전투력과 관련된 모든 인수(Factor)를 데이터로 하여 최종적으로 어느 쪽이 승리할 것인지를 예측하는 프로그램이다. 이를 위해서는 지휘부, 육군, 해군, 공군 외에도 장비, 날씨, 전쟁상황, 승패판정 등의 수많은 유무형의 객체를 설정해볼 수 있다. 어느 순간에 적의 공격징후를 포착한 '전쟁상황' 객체가 '지휘부' 객체에게 적이 침입했다는 메시지(Message, Event)를 넘긴다. '지휘부' 객체는 이 메시지에 응답하여 '육군', '해군', '공군' 등 객체에게 전투를 준비하라는 메시지를 넘긴다. '육군' 객체는 다시 이 메시지에 응답하여 '장비' 객체에게 이동 명령 메시지를 넘긴다.

이처럼 객체지향 설계에서는 이벤트 하나가 발생함으로 말미암아 모든 객체 간에 서로 뒤엉키면서 메시지가 오가는 상황으로 바뀐다. 이렇게 되면 프로그램을 실행해보기 전까지는 전체적인 메시지 흐름을 미리 한 눈에 조망하여 어느 쪽이 승자인지 예측하는 것은 불가능해진다. 결국, 프로그래머로서는 절차적 프로그램처럼 전체적인 프로그램의 흐름에 주목하기보다 객체 하나하나의 코딩에 충실해야 한다. 즉, 각각의 객체에 대해서, 이러한 메시지가 오면 이러한 반응을 보인다고 작성하면 그만이다. 객체 사이에 수많은 메시지가 오간 결과, 최종 승패는 '승패판정' 객체가 알려줄 것이기 때문이다.

"각자의 역할에만 충실하라. 그러면 이 세상은 돌아간다."

Section

2 객체 클래스와 상속

유사한 특징을 지닌 객체를 묶어서 그룹지은 것이 객체 클래스(Object Class)다. 그리고 **인스턴시에이션(Instantiation)**이란 '추상적 개념으로부터 구체적 실례 하나를 만듦'을 의미한다. 따라서 해당 클래스에 속하는 실례를 클래스 인스턴스(Class Instance) 또는 객체(Object)라 한다. 꽃 파는 사람이라는 유사한 특징을 지닌 사람을 그룹지어 '꽃집주인' 클래스로 정의하면, 해당 클래스에 속하는 김꽃분은 객체가 된다. 다시 말해, 객체 김꽃분은 클래스 꽃집주인에 속하는 많은 사람 중 하나다.

인스턴시에이션 (Instantiation)

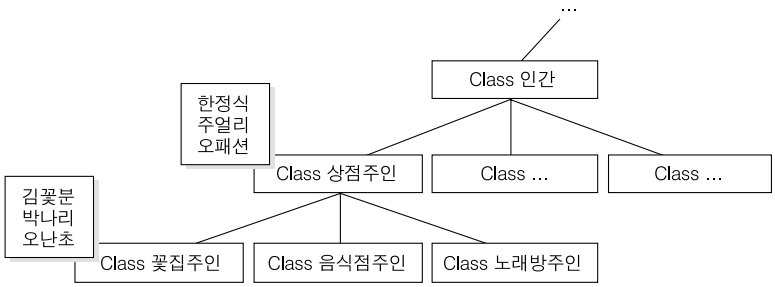
같은 클래스에 속한 모든 객체는 동일한 메시지에 대해 동일하게 반응한다. 즉, 동일한 **메소드(Method, 방법)**를 사용한다. 예를 들어, 꽃집주인 클래스에 속한 객체 김꽃분에게 "꽃을 보내 달라"라는 메시지를 보냈을 때 "네, 알겠습니다. 보내겠습니다"라는 메소드를 동원한다면 다른 꽃집주인 박나리도 똑같은 메소드를 동원한다는 의미다. 그러나 클래스가 다르면 동일한 메시지에 대해서 달리 반응할 수도 있다. 전화를 잘못 걸어서 치과의사 클래스에 속한 객체인 김치과에게 "꽃을 보내 달라"라는 메시지를 보내면 "전화를 잘못 거셨습니다"라는 메소드를 동원할 수도 있다. 동일한 메시지에 대해서, 클래스에 따라서 달리 반응할 수 있게 하는 것도 객체지향에서 말하는 **다형성(多形性, Poly-morphism)**의 일례라 할 수 있다.

메소드(Method, 방법)

다형성(多形性, Poly-morphism)

꽃집주인은 상점주인의 특수한 경우(Specialization)다. 상점주인에는 꽃집주인도 있지만 음식점주인, 노래방주인 등 많은 경우가 있다. 또, 상점주인은 인간이라는 카테고리의 특수한 경우다. 인간은 상점주인만 있는 것이 아니다. 이런 식으로 클래스를 바라보면 클래스 간에 **계층구조(Hierarchy)**가 존재하는 것을 알 수 있다. 다시 말해, 어떤 클래스(Child Class, Sub Class, Derived Class)는 더 높은 일반적인 클래스(Parent Class, Super Class, Base Class)에 속하고, 그 클래스는 다시 더 높고 포괄적인 클래스에 속한다고 볼 수 있다. [그림 1-4]를 보면 '인간 → 상점주인 → 꽃집주인' 순으로 일반적인 클래스에서 특수한 클래스로 이행한다.

계층구조(Hierarchy)



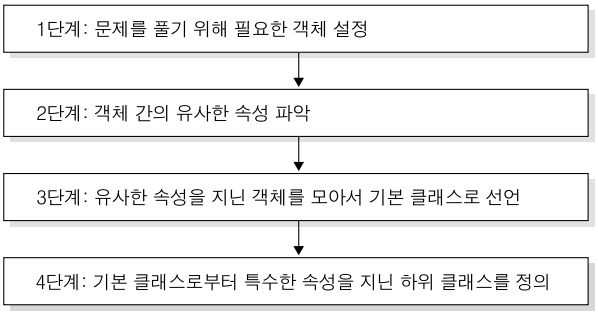
[그림 1-4] 클래스 계층구조

속성(Attribute)

상속(Inheritance)

객체지향 언어에서 클래스 간에 계층구조를 선언하는 것은 상위 클래스의 **속성**(Attribute)을 **상속**(Inheritance) 받기 위함이다. 예를 들어, 모든 상점주인은 물건값을 받으면 영수증을 주는 속성(Attribute)이 있다. 따라서 상점주인 클래스에서 "물건값을 받는다"라는 메시지에 대응하여 "영수증을 준다"라는 메소드를 한번 선언하였으면, 꽃집주인 클래스에서는 그러한 메소드를 굳이 다시 선언하지 않아도 자동으로 "영수증을 준다"라는 메소드가 실행된다. 즉, 상위 클래스에서 메소드를 선언하였을 때, 어떤 클래스가 그 하위 클래스라고 선언하기만 하면 자동으로 상위 메소드를 상속받게 된다. 자신의 소속을 밝힘으로써 자신이 속한 단체의 재산을 거저 얻어오는 행위라고도 할 수 있다.

따라서 어떤 문제를 풀기 위해서 객체지향적으로 설계하려면 다음과 같은 과정을 거쳐야 한다.



[그림 1-5] 객체지향 설계 과정

Section

03 C++와 객체지향

소프트웨어 규모가 커질수록 혼자서 일일이 모든 프로그램을 짜기는 어려워진다. 따라서 메인 함수를 짜는 사람과 수금관리 객체를 짜는 사람이 서로 다를 수 있다. 다른 객체의 내부 프로그램이 어떻게 돌아가는지 모르면서도(Information Hiding) 프로그램이 가능한 것이 객체지향의 특성이다. 즉, 수금관리 객체가 어떤 방법(Method)을 사용하는지 몰라도 메인 함수는 단지 "지난 달 수금액 합계가 얼마나 되니?"라는 메시지를 던지고 해당 객체로부터 응답(Response)만 받아오면 된다. 이 메시지는 이런 의미므로 이렇게 해석하여 처리해달라는 사전 약속만 한다면 객체별로 서로 다른 사람이 프로그래밍해도 아무런 문제가 없다는 것이다.

단위성(Modularity)

내가 짜든, 남이 짜든 간에 객체 단위로 프로그램이 작성되기 때문에 프로그램의 **단위성(Modularity)**이 높아진다. 큰 문제 전체를 하나로 놓고 한꺼번에 모든 세부사항을 처리하는 것보다는 그 문제를 잘라서 작은 단위(Module)로 분할한 다음 이들 개별 단위별로 구현하는 것이 훨씬 수월한 방법이다. 해결하려는 문제의 크기가 줄어들기 때문이다. 즉, 문제 해결에 필요한 주요 단위를 객체로 설정하고 객체 단위로 구현에 집중하는 것이다. 이처럼 객체 단위로 잘라 놓으면 나중에 다른 문제를 해결할 때에도 필요한 객체만 가져다가 조합함으로써 **재사용성(Reusability)**을 높일 수 있다.

재사용성(Reusability)

인터페이스 파일 (Interface File, Header File)

구현 파일 (Implemenation File)

객체지향 언어의 일종인 C++ 프로그램 소스 파일은 일반적으로 객체별로 나뉘어 구성된다. 객체 하나에 대해서 확장자 .h로 끝나는 **인터페이스 파일(Interface File, Header File)**과, 확장자 .cpp로 끝나는 **구현 파일(Implemenation File)** 등 두 가지를 작성해야 한다.

인터페이스 파일은 문자 그대로 객체를 불러서 사용할 외부 사용자와의 인터페이스를 위한 파일이다. 여기에는 해당 객체가 할 수 있는 일, 즉 외부에서 객체에게 던질 수 있는 메시지들이 나열되어 있다. 따라서 외부 사용자 입장에서는 구현 파일은 보지 않고, 인터페이스 파일만 보고도 필요한 메시지를 던질 수 있다. 이 경우, 실제로 그 메시지를 구현하는 사람은 그냥 갖다 쓰는 사람을 위해서 해당 메시지가 무엇을 위한 것인지 친절하고도 정확한 주석

(Comment)을 써두는 것이 좋다. 카드놀이 프로그램용 객체 card를 선언하기 위한 인터페이스 파일인 card.h를 예로 들어 보자.

❶ 인터페이스 파일

[코드 1-1] card.h 인터페이스 파일

```
enum suits {diamond, clover, heart, spade}  타입 suit는 카드무늬 집합 중 하나
enum colors {red, black}                    타입 colors는 카드색 집합 중 하나

class card{
public:
        card();                     생성자 함수
        ~card();                    소멸자 함수
        colors  Color();            현재 카드의 색깔을 되돌려 주는 함수
        bool    IsFaceUp();         앞면이 위인지 아래인지 되돌려주는 함수
        int     Rank();             카드에 쓰인 숫자를 되돌려주는 함수
        void    SetRank(int x);     카드의 숫자를 x로 세팅하는 함수
        void    Draw();             카드를 화면에 그려내는 함수
        void    Flip();             카드를 뒤집는 멤버 함수
private:
        bool    Faceup;             그림이 위로 향하고 있는지 나타내는 변수
        int     Rval;               카드 숫자를 나타내는 변수
        suits   Sval;               카드 종류를 나타내는 변수
};
```

C++의 인터페이스 파일은 객체 클래스를 선언하는 부분이다. 클래스 선언은 class라는 키워드로 시작된다. 클래스 선언은 public 섹션과 private 섹션 등 두 부분으로 구성된다. public은 '공유'라는 뜻으로 그 내부에 선언된 메시지를 외부 사용자가 공유할 수 있다는 의미다. 따라서 외부 사용자는 여기에 선언된 메시지를 사용하여 이 클래스의 객체에게 일을 시킬 수 있다. 이 예에서는 메시지 여덟 개가 선언되어 있다.

멤버 함수
(Member Function)

객체지향 개념에서의 메시지(Message)를 C++에서는 **멤버 함수**(Member Function)라고 부른다. 클래스 선언은 인터페이스 역할만 하면 충분하므로 멤버 함수의 구체적인 구현 내

용은 나타나지 않고 함수명(Function Prototype)만 표시한다. 카드 객체를 불러서 사용하는 외부에서는 구체적인 구현 내용을 볼 필요가 없기 때문이다. 대신, 외부에서 이 파일만 보고 메시지를 던져야 하는 사람을 위해서 메시지가 구체적으로 무엇을 해주는 것인지 쉽게 알아볼 수 있도록 명확하고도 자세한 주석을 적어 주어야 한다.

생성자(Constructor)

멤버 함수 중, 클래스명과 동일한 함수명을 사용하되 리턴 타입을 선언하지 않는 함수가 **생성자(Constructor)** 함수다. 이 함수는 문자 그대로 새로운 객체를 만드는 함수로써, 객체 선언 시에 자동으로 불려져 온다. 예를 들어, card Mycard;라고 선언하면 이 생성자 함수에 의해 card 클래스의 객체 Mycard가 생성된다. 반면에, 객체를 없애고 그 객체가 사용하던 메모리 공간을 운영체제에게 반납하기 위한 함수가 **소멸자(Destructor)** 함수다. 소멸자 함수는 클래스명 앞에 물결(Tilde, ~) 표시로 구분하며, 생성자 함수와 마찬가지로 리턴 타입을 선언하지 않는다.

소멸자(Destructor)

C++ 클래스 선언의 두 번째 섹션은 private이라는 키워드로 시작한다. 문자 그대로 이 부분은 사적인 것으로서 외부 사용자와 공유하는 부분이 아니다. 다시 말해, 이 섹션에 선언된 변수나 함수를 읽고 쓰려고 외부에서 직접 접근(Access)할 수 없다. 이 부분을 접근할 자격을 지닌 것은 자체 클래스의 멤버 함수 뿐이다. 따라서 외부에서 이 부분을 접근하는 유일한 방법은 멤버 함수 호출을 통하는 것이다. 일반적으로 private 섹션에 나타나는 것은 멤버 함수를 수행하는 데 필요한 객체 내부의 저장 공간 즉, 변수들로서 이를 C++에서는 **멤버 데이터(Member Data)**라고 부른다. 객체 자신의 변화상태를 나타내는 변수라는 의미에서 객체지향에서는 이를 **상태 변수(State Variable, Instance Variable)**라고도 한다.

멤버 데이터
(Member Data)

상태 변수
(State Variable,
Instance Variable)

외부 사용자 입장에서 보면 private 섹션은 무의미하다. 즉, 아예 읽어볼 필요가 없는 부분이고 따라서 그러한 것은 구현 파일에 포함시켜야 한다. 따라서 외부 사용자 입장에서는 불러 쓸 수 있는 public 섹션만 읽어보면 그만이다. 그럼에도 불구하고 C++의 인터페이스 파일에 private 섹션을 두는 이유는 C++ 자체가 처음부터 객체지향 언어로서 설계되지 않고, 기존의 C 문법에 추가적인 요소를 반영하여 만들어졌기 때문이다.

요약하자면, C++의 객체 클래스 선언은 두 부분으로 나뉜다. 공유 가능한 public 섹션과 공유 불가능한 private 섹션이다. 각각의 섹션에는 멤버 함수 뿐만 아니라 멤버 데이터도 들어갈 수 있다. 외부 사용자 입장에서 볼 때, 멤버 함수는 일종의 동사에 해당하며 그 객체가 처리할 수 있는 메시지에 해당한다. 멤버 데이터는 일종의 명사에 해당하며, 객체가 메시지를 실행하기 위해 필요로 하는 변수를 말한다.

❷ 구현 파일

메소드(Method)

외부에서는 메시지를 사용해서 객체에게 일을 시키면 되지만, 최종적으로 누군가는 그 메시지를 처리하는 **메소드(Method)**를 구현해야만 한다. C++에서는 메시지명 자체가 메소드명이 된다. 즉, '메시지 = 메소드 = 멤버 함수' 라 할 수 있다. 카드 객체의 메시지를 구현한 card.cpp 파일의 일부를 살펴보자.

[코드 1-2] card.cpp 구현 파일

```
#include "card.h"
card::card()
{     Sval = diamond;          카드 종류는 다이아몬드
      Rval = 7;                카드 숫자는  7로
      Faceup = TRUE;           앞면을 위로
}
int card::Rank()
{     return Rval;             현재의 카드 숫자를 되돌려 줌
}
```

구현 함수에서는 클래스명을 먼저 쓰고 콜론(:) 두개를 붙인 다음 이어서 메시지명을 써야 한다. 이렇게 함으로써 어느 클래스에 속한 함수인지를 나타낼 수 있다. 같은 이름의 메시지라 할지라도 클래스에 따라서 처리 방식이 달라질 수 있기(Polymorphism) 때문이다. 위 생성자 함수는 멤버 데이터를 초기화하는 데 있어서 카드 종류는 다이아몬드, 카드 숫자는 7, 앞면을 위로 향하게 하고 있다. Rank( ) 함수는 현재의 카드 숫자를 되돌려 주는 함수로서 상태 변수 Rval에 그 값이 저장되어 있다. 멤버 함수는 private 섹션을 자유로이 접근할 수 있으므로 그 값을 읽어서 되돌려주면 된다.

[표 1-1] C++ 소스 파일의 구성

인터페이스 파일(.h)	구현 파일(.cpp 또는 .c 또는 .cc)
클래스 선언	클래스 구현
#define: 매크로 정의 #include: 다른 인터페이스 파일 포함 typedef: 타입 선언	#include: 자체 헤더 파일 포함 데이터/ 객체 선언

이처럼 C++ 소스파일은 인터페이스 파일(Interface File, 헤더 파일)과 구현 파일(Implementation File)로 구성되어 있다. 각 파일에 들어가는 요소는 [표 1-1]과 같다. 매크로 정의, 타입 선언, 다른 인터페이스 파일 등의 요소는 일반적으로 인터페이스 파일에 포함시킨다. card.cpp 파일이 #include "card.h"로 시작하는 것처럼, 구현 파일의 첫 부분에는 항상 인터페이스 파일을 포함시킨다.

3. 메시지 전달 요령

객체별로 인터페이스 파일과 구현 파일을 작성했다면 메인 함수를 비롯한 다른 객체들은 다음과 같은 형식으로 원하는 객체에 메시지를 던질 수 있다.

```
int main()
{   card MyCard;              내 카드 객체 하나 만들기
    MyCard.Flip();            내 카드여! 뒤집어라.
    cout << MyCard.Rank();    내 카드여! 현재 네 숫자가 얼마인지 화면에 찍어라.
     ...
}
```

어떤 클래스에 속한 객체를 선언하려면 클래스명 다음에 객체명을 선언하면 된다. 위 예에서 card는 클래스명이고 MyCard는 해당 클래스에 속한 객체명이다. 이는 마치 C 언어에서의 int x;와 유사하다. 앞의 int는 타입명이고 뒤의 x는 해당 타입의 변수기 때문이다. 즉, 타입명에 해당하는 것이 클래스명이며, 변수에 해당하는 것이 객체다. 생성자 함수는 card MyCard;라고 선언하는 순간 자동으로 불려온다. [코드 1-2]의 생성자 함수에서는 새로운 카드가 생성될 때 그 숫자 값을 임의로 7이란 숫자로 초기화했으므로 이 프로그램을 실행하면 결과적으로 숫자 7이 찍히게 된다.

C++에서 메시지 전달(Message Passing)은 메시지를 받을 객체 다음에 점을 찍고 메시지명을 대면 된다. 즉, MyCard.Flip( )에서 MyCard는 객체고 Flip은 그 객체가 수행할 수 있는 메시지다. 필요하다면 괄호 안에 파라미터(Parameter, Argument)를 같이 넣어서 전달하면 된다. 객체의 멤버 데이터를 지정할 때에도 객체명 다음에 점을 찍고 변수명을 지정한다. 예를 들어, MyCard.Rval이라고 하면 객체 MyCard의 상태 변수 중 RVal을 지칭한다. 메시지, 즉 멤버 함수 다음에는 파라미터가 없더라도 빈 괄호를 넣는 것이 상례다. 괄호가 없을 경우에는 그것이 멤버 함수인지 멤버 데이터인지 헷갈리기 때문이다.

클래스 하나에 객체 여러 개를 선언할 수 있다. 만약, 위 main 함수에서 card MyCard, YourCard;라고 선언하였다면 실제로 같은 클래스에 객체 두 개가 생성되는 것이다. 이 경우, 멤버 데이터는 해당 객체별로 따로 존재한다. 따라서 MyCard.SetRank(10); YourCard.SetRank(5);라고 하였다면 MyCard 객체의 멤버 데이터 Rval 값은 10으로, YourCard 객체의 멤버 데이터 Rval 값은 5로 설정된다.

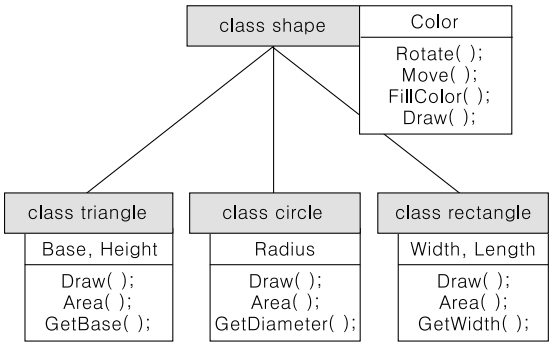
❹ 상속과 다형성

삼각형 객체를 만들기 위해 다음과 같이 클래스 삼각형(Triangle)을 선언해보자.

[코드 1-3] 삼각형 객체 선언

```
class triangle: public shape{
public:
    triangle();             생성자 함수
    ~triangle();            소멸자 함수
    void Draw();            현재 객체를 화면에 그리는 함수
    float Area();           현재 객체의 면적을 계산하는 함수
    float GetBase();        현재 객체의 밑변 길이를 되돌려주는 함수
private:
    float Base;             밑변의 길이를 나타내는 변수
    float Height;           높이의 길이를 나타내는 변수
};
```

코드 첫 줄에 class triangle: public shape라고 한 것은 클래스 triangle이 클래스 shape의 하위 클래스임을 밝힌 것이다. 이는 [그림 1-6]과 같이 클래스 shape의 하위 클래스로서 클래스 triangle, 클래스 circle, 클래스 rectangle을 가정한 것이다.



[그림 1-6] 클래스 계층구조의 예

클래스 속성(Attribute)은 멤버 함수와 멤버 데이터 모두를 일컫는다. 예를 들어, [그림 1-6]의 클래스 shape의 속성은 Rotate 등의 멤버 함수 외에도 멤버 데이터인 변수 Color까지 포함하는 개념이다. 따라서 하위 클래스가 상위 클래스의 속성을 이어받는다는 것은 멤버 함수와 멤버 데이터 모두를 이어받는다는 것이다.

먼저, 멤버 함수를 상속받는 경우를 보자. triangle T;라고 선언하고 T.Rotate( ); 명령을 내릴 수 있다. 클래스 Triangle을 정의한 [코드 1-3]에는 분명 Rotate라는 멤버 함수가 없음에도 이러한 명령을 내릴 수 있는 것은 상위 클래스인 class shape에 Rotate라는 멤버 함수가 정의되고 구현되어 있기 때문이다. 즉, 상속에 의한다.

객체와 어떤 멤버 함수가 연결되는지는 주의를 요한다. 예를 들어, T.Rotate( );라고 하면 일단 T의 클래스인 triangle 클래스에 해당 함수가 있는지 먼저 검색된다. 만약, 해당 함수가 있다면 triangle 클래스에 선언된 함수가 직접 실행된다. 그 곳에 없을 때에 한해서만 그 상위 클래스인 shape 클래스로 가서 다시 그 함수가 있는지 찾게 된다. 이런 식으로 계속적으로 상위 클래스로 올라가면서 가장 먼저 선언된 것이 사용된다. 이러한 객체와 메시지의 연결(Binding)은 컴파일 시(Compile Time)에 일어나는 것이 아니라 실행 시(Run Time)에 일어난다는 의미에서 이를 **동적 연결(Dynamic Binding)**이라 한다.

동적 연결
(Dynamic Binding)

예를 들어, T.Draw( );라고 하면 triangle 클래스의 Draw 함수가 실행된다. 이 경우, shape 클래스의 Draw 함수는 triangle 클래스에 Draw 함수가 정의되어 있지 않을 때 실행된다. 하위 클래스에서 상위 클래스와 동일한 함수를 정의할 때에는 그 상위 클래스의 함수를 덮어 씌우고자 하는 것이다. 이는 상위 클래스의 함수가 일반적인 함수일 때, 하위 클래스가 그 함수를 자체 특성에 맞게 특화(Specialization)시킬 때 사용하는 기법이다.

동일한 메시지를 서로 다른 클래스의 객체에게 던졌을 때 다르게 반응하는 것이 다형성이다. 예를 들어, triangle T; circle C; rectangle R;이라고 선언하고 T.Draw( ); C.Draw( ); R.Draw( );라고 하면 Draw라는 명령은 동일하지만 삼각형, 원, 사각형 등 서로 다른 그림이 그려진다. 각각의 클래스에 Draw( )라는 동일한 메시지를 보냈지만 그 메시지가 서로 다른 방법으로 구현되어 있기 때문이다. Draw라는 동일한 메시지에 대해서 객체가 알아서 자신의 특성을 반영한다는 점에서 이는 프로그래머에게 매우 유용한 기능이다.

상위 클래스의 상태 변수나 메시지들이 private 섹션에 정의되어 있다면, 이는 상위 클래스의 자체에 속하는 객체들만 접근할 수 있다. 그러나 때에 따라서는 하위 클래스가 상위 클래스의 상태 변수나 메시지를 사용해야 할 때가 있다. 이를 가능하게 하려면 상위 클래스에서 이들을 protected 섹션 내부에 정의하면 된다. protected 섹션에 의해서 다른 클래스 객체와 자신의 하위 객체의 접근성을 차별화시킨다. 즉, 다른 객체는 접근할 수 없지만 같은 가족인 하위 클래스 객체는 접근할 수 있게 한 것이다.

5. 간단한 계층구조

급여 관리를 위한 가장 단순한 계층구조를 설계해보자. 객체지향 설계의 가이드라인에 의해서 다음과 같이 진행해 볼 수 있다.

1단계. 문제를 풀기위해 필요한 객체를 설정한다.

① 풀타임 직원

② 파트타임 직원

2단계. 객체들 간의 유사한 속성을 파악한다.

① ID 번호

② 성명

③ 부서명

3단계. 유사한 속성을 지닌 객체를 모아서 기본 클래스를 선언한다.

```
class Employee
{
protected:
        long myIdNum;           직원 사원번호
        char Name[12];          성명
```

```
        int DeptCode;           부서코드
        .....
public:
        .....                   필요한 멤버 함수 정의
};
```

4단계. 기본 클래스로부터 특수 속성을 지닌 하위 클래스를 정의한다.

```
①
class FullTimeEmployee : public Employee
{ public:
        ...                     풀타임 직원용 멤버 함수
  protected:
        double mySalary;
};
②
class PartTimeEmployee : public Employee
{ public:
        ...                     파트타임 직원용 멤버 함수
  protected:
        double WeeklyWage;
        int HoursWorked;
};
```

6. 연산자 오버로딩

객체지향 언어의 다형성(Polymorphism)은 여러가지 형태로 나타난다. 동일한 메시지에 대해 객체에 따라 서로 달리 반응하는 것도 다형성이지만 연산자(Operator)의 의미를 다르게 정의하는 것도 다형성이다. 예를 들어, 할당을 위한 Equal(=)을 보자. 우리가 일반적으로 int x; x = 2;에 의해 정수 변수 x에 정수값 2를 할당하는 것은 별 무리가 없다. 그러나 카드 클래스에 속하는 객체를 card C1, C2;라고 선언하고 프로그램 중간에 C1 = C2;처럼 할당하였다고 해보자.

이 경우, 프로그래머로서는 C2의 값이 C1으로 복사되기를 원하는 것이다. 그러나 C2는 단순한 데이터 타입이 아니라 카드 클래스의 객체다. 따라서 이 경우에는 카드 클래스 객체 내부의 모든 상태 변수 값이 복사되어야 한다. 즉, C2의 상태 변수 FaceUp, Rval, Sval의 현재 값이 C1로 각각 복사되어야 한다. 이렇게 하려면 Equal 연산자에 또 다른 의미를 부여해야만 한다. 즉 x = 2;를 수행할 때 사용되는 연산자와 C1 = C2를 수행할 때 사용되는

연산자 오버로딩
(Operator
Overloading)

연산자는 달리 정의되어야 한다. 이처럼 동일한 연산자에 두 개 이상의 의미를 부여하는 것을 **연산자 오버로딩**(Operator Overloading)이라 한다. 오버로딩을 위해서는 [코드 1-4]와 같은 함수를 추가하면 된다.

[코드 1-4] 오버로딩을 위한 함수

```
card.h 파일.....................................................
void operator = (card C);

card.cpp 파일...................................................
void card::operator = (card C)
{ FaceUp = C.FaceUp;
  Rval = C.Rval;
  Sval = C.Sval;
}
```

우선 헤더파일의 클래스 선언 내부에 멤버 함수로서 void operator = (card C);라고 정의한다. 연산자의 우변에 card C라고 한 것은 연산자 우변에 card 클래스의 객체가 올 때에는 operator =이라는 멤버 함수를 사용하라는 것이다. 이 멤버 함수는 별도로 구현 파일에 정의해줘야 한다. 위 구현 파일에서 FaceUp = C.FaceUp;이라고 한 것은 우변에 넘겨진 객체의 FaceUp 변수 값을 현재 이 메시지를 수행하는 객체의 FaceUp 변수에 복사하라는 것이다. C1 = C2;라고 할 때에는 당연히 우변이 C2가 되므로 C2.FaceUp 값이 C1.FaceUp으로 복사된다.

프로그램 실행 시에 일반적인 Equal 연산자 함수를 부를 것인지, 아니면 위에 정의된 연산자 함수를 부를 것인지는 소스 코드의 형태에 의해 파악된다. Equal 연산자 우변에 객체가 없으면 당연히 일반 연산자가 불려 오고, 반대로 객체가 있으면 위에 정의된 연산자가 불려 온다.

Section

04 절차적 설계와의 비교

객체지향 언어는 '객체'를 강조하는 반면, 파스칼이나 C와 같은 절차적 언어(Procedural Language)는 '작업'을 강조한다. 객체지향 역시 결국은 작업(Operation, Message, Method, Member Function)을 구현해야 하므로 어찌 들으면 같은 말인 것 같으나 이 두 가지 사이에는 "내 카드 객체에게 뒤집으라는 작업을 시킬 것인가, 아니면 뒤집으라는 함수에게 내 카드를 전달할 것인가"라는 엄연한 차이점이 존재한다. 전자가 객체지향적 접근방식이라면 후자는 절차적인 접근방식이다.

절차적 설계에서는 어떤 작업을 함수로 정의하고 이를 반복해서 호출함으로써 재사용성을 높인다. 예를 들어, 어떤 문제를 해결하기 위해 최소치를 구하는 계산이 필요하면 최소치 함수를, 또 평균값을 구하는 계산이 필요하면 평균치 함수를 만들어서 라이브러리 형식으로 재사용한다.

반면에 객체지향 설계에서는 이러한 모든 함수를 수행할 수 있는 객체를 설정하는 것을 우선시 한다. 일단, 계산기라는 객체 클래스를 만들고 거기에 일반적으로 계산기가 수행해야 하는 모든 작업을 선언한 다음, 내 계산기라는 객체를 선언하면 내 계산기는 최소치, 평균치를 포함하여 일반적인 계산기 클래스가 할 수 있는 모든 작업을 수행할 수 있게 된다. 이처럼 어떤 객체를 설정하여 그 객체가 할 수 있는 모든 기능을 한 군데 모아 놓는 접근 방법이 재사용성 면에서는 더욱 유리하다.

구조면에서 볼 때, 객체지향 언어는 객체가 수행할 수 있는 작업과 작업에 필요한 변수를 묶어서 한꺼번에 정의한다. 다시 말해, public 섹션에는 작업을, private 섹션에는 변수를 선언하여 그 두 개를 클래스 하나로 묶어버린다. private 섹션의 변수는 해당 public 섹션의 작업에만 사용되게 함으로써 어떤 변수가 어떤 작업에 이용되는지가 분명해진다.

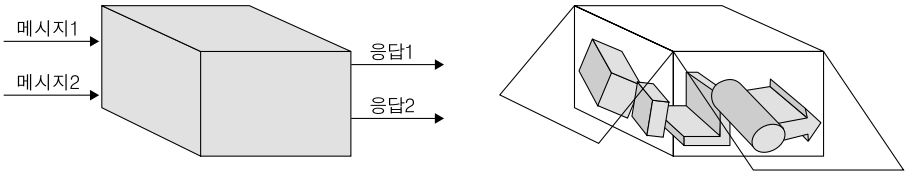
반면, 기존의 절차적 언어에서는 어떤 변수가 어떤 함수에 사용되는지 분간하기 어렵다. 예를 들어, 전역 변수나 지역 변수를 막론하고 절차적 프로그램에서는 함수 윗부분에 int A; long B; char C; bool D;라는 식으로 필요한 변수를 나열한 후, 그 아래 함수에서는 선언된 어

떤 변수를 가져다 써도 무방하기 때문이다. 때에 따라서는 변수 하나가 서로 다른 함수에 의해 재사용되기도 한다.

객체지향에서 무엇보다 중요한 점은 인터페이스와 구현의 분리다. 외부에서 객체를 사용하는 사람의 입장에서는 인터페이스만 알면 된다. 객체 내부가 구체적으로 어떻게 구현되는지는 몰라도 된다. 또 몰라야 한다. 이처럼 객체 내부가 외부에 공개되지 않도록 "외부와 벽을 쌓고 있다"라는 의미에서 이를 **인캡슐레이션(Encapsulation)**이라 한다.

인캡슐레이션
(Encapsulation)

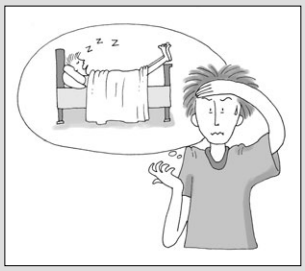
이는 캡슐로 둘러싸여 있다는 의미다. 다시 말해, [그림 1-7]처럼 외부 사용자의 눈에는 객체 내부가 전혀 들여다보이지 않는 일종의 블랙 박스(Black Box)로 보인다. 외부 사용자는 그 객체가 할 수 있는 작업이 어떤 것들이 있는지, 또 그 작업이 무엇을 하는 작업인지만 알고 있으면 된다. 필요하면 메시지를 던져서 일만 시키면 그만이기 때문이다. 반면, 객체 내부에서 그 객체를 구현하는 사람은 구체적으로 작업이 어떻게 구현되는지, 상태 변수가 어떻게 변하는지 알아야 한다.



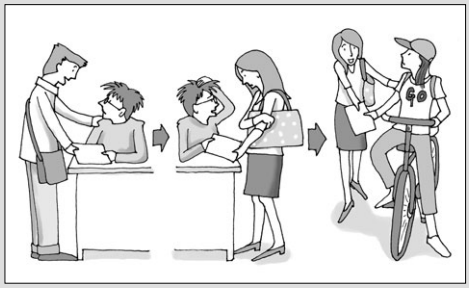
인터페이스 관점: Object = Black Box　　구현 관점: Object = Visible

[그림 1-7] 인터페이스 관점과 구현 관점

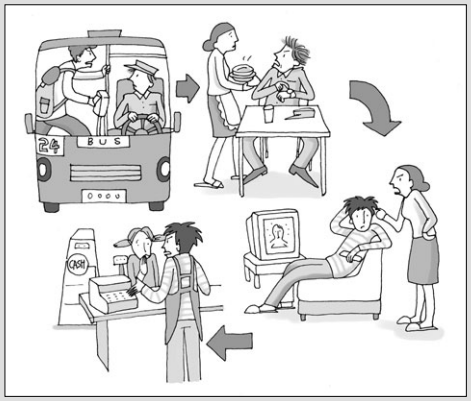
생활 속의 자료구조



컴퓨터공학과에 다니는 박군이 학생회장에 당선되었다. 학생들을 위해 무엇을 할까? 아니 무엇을 할 수 있을까? MT, 축제, 자금, 사전답사, 학회지, 여러 고민 끝에 그가 내린 결론은 한 가지였다. 기획부장, 총무부장, 섭외부장, 문화부장을 선출하기로 한 것이다. 적재적소에 인물을 배치하고 이 사람들만 잘 다루면 된다는 일종의 노하우다. 그런데 담당 부장을 뽑은 후, 한달이 지나도록 박군은 노심초사하고 있다. 과연 문화부장이 이번 학회지 기안을 제대로 진행할 수 있을지. 학생회비는 제대로 걷히고 있는지 자세히 알 수가 없어서다. 그러나 일을 시켜놓고 못 믿을 거라면 무엇 때문에 일을 맡겼는지 문화부장으로서도 고민이다.



컴퓨터공학과 최군은 비디오방 프로그램을 짜오라는 이번 숙제를 객체지향적으로 작성하기로 했다. 그래서 숙제가 나오자마자 향우회 후배인 이군에게 넘겨버렸다. 이군 역시 객체지향적으로 과감하게 CC인 조양에게 넘겨버렸다. 조양도 숙고 끝에 다른 남자 친구에게 넘겼다. 객체지향이란 결국 무한루프인가?



등교 길에 사소한 일로 버스 기사에게 화를 냈다. 버스 기사는 점심식사 중에 식당 주인에게 좋지 않은 소리를 했다. 식당 주인은 집에 가서 대학생 아들에게 잔소리를 했다. 대학생 아들은 편의점에서 아르바이트를 하면서 손님에게 불친절하게 대한다. 아마도 어떤 메시지는 누군가에 의해 단절되는 것이 좋을 것 같다.

요약

객체지향이란 객체에게 작업을 시키는 형태로 "프로그램의 조직화"를 말한다. 이를 위해서는 필요한 구성요소를 객체 클래스 단위로 조직화해야 한다. 조직화된 클래스에 속한 객체에게 작업을 시키려면 메시지를 전달하면 된다.

계층적으로 선언된 클래스에서 하위 클래스 객체는 상위 클래스 객체의 메시지를 상속받는다. 동일한 메시지라도 전달 받는 객체에 따라서 다르게 반응하도록 배려한 것이 객체지향의 다형성이다.

객체지향 언어는 인터페이스 파일과 구현 파일을 철저히 분리한다. 이렇게 함으로써 사용자 입장에서는 구현 파일 내용을 전혀 모르는 상태에서 단지 인터페이스 파일만 참조하여 메시지를 전달함으로써 객체에게 작업을 시킬 수 있게 된다.

절차적 설계에서는 객체라는 개념이 아예 존재하지 않는다. 반면 객체지향적 설계에서는 객체라는 개념 아래 그 객체가 처리할 수 있는 작업과 이에 필요한 데이터를 묶어서 캡슐화한다. 외부 사용자 입장에서는 이 캡슐 내부를 들여다 볼 수 없고, 단지 메시지를 통해 캡슐화된 객체에게 작업을 시킬 수 있을 뿐이다.

연습 문제

1 객체지향 프로그래밍에서 인캡슐레이션(Encapsulation)의 의미와 그것이 필요한 이유를 간략히 설명하라.

2 객체지향 프로그래밍을 일명 디스크리트 이벤트 시뮬레이션(Discrete Event Simulation)이라 부르는 이유를 간략히 설명하라.

3 다음 중 함수의 오버로딩을 설명한 것은 무엇인가?

가. 서로 다른 파라미터와 서로 다른 이름을 가진 함수
나. 동일한 파라미터와 서로 다른 이름을 가진 함수
다. 서로 다른 파라미터와 동일한 이름을 가진 함수
라. 동일한 파라미터와 동일한 이름을 가진 함수

4 다음 중 상속의 목적이 아닌 것은 무엇인가?

가. 상위 클래스의 인터페이스를 재사용하기 위한 것이다.
나. 상위 클래스에서 구현 내용 일부를 재사용하기 위한 것이다.
다. 하위 클래스에서는 상위 클래스에 없는 작업만 정의하기 위한 것이다.
라. 상위 클래스의 프라이빗 상태 변수에 접근하기 위한 것이다.

5 다음 중 프로그램 출력 결과는 무엇인가?

```
class A {
public:
          int F () { return 1; };
};

class B: public A {
public:
          int F () { return 2; };
};

A a; B b; cout << a.F() << "," << b.F() << endl;
```

가. 1,1　　나. 1,2　　다. 2,1　　라. 2,2

6 어떤 클래스의 멤버 함수가 같은 클래스의 다른 멤버 함수를 사용할 수 있는가?

가. 사용할 수 없다.
나. public 멤버 함수만 사용할 수 있다.
다. private 멤버 함수만 사용할 수 있다.
라. public, private 할 것 없이 모두 사용할 수 있다.

연습 문제

7 서로 다른 클래스 두 개가 같은 이름의 멤버 함수를 가질 수 있는 경우는 어떤 때인가?

가. 가질 수 없다.
나. 두 클래스 모두 같은 이름일 때만 그렇다.
다. 메인 프로그램이 두 클래스를 모두 선언하지 않을 때만 그렇다.
라. 항상 가질 수 있다.

8 다음 중 맞는 것을 고르라.

가. 멤버 함수와 멤버 데이터 모두 private이다.
나. 멤버 함수는 private이고 멤버 데이터는 public이다.
다. 멤버 데이터는 private이고, 멤버 함수는 public이다.
라. 멤버 함수와 멤버 데이터 모두 public이다.
마. 모든 문항이 모두 옳지 않다.

9 기본 생성자(Default Constructor)가 필요한 주목적은 무엇인가?

가. 프로그램 하나에 여러 가지 클래스를 사용하려고
나. 함수 호출 시 함수에 전달되는 매개변수를 복사하려고
다. 각각의 객체가 선언될 때, 객체를 초기화하려고
라. 어떤 클래스에 몇 개의 객체가 선언되었는지 세려고

10 어떤 클래스에 대해서 Equal 기호에 대한 오버로딩 연산이 정의되어 있지 않다. 이 경우 a = b;에 의해 객체 간에 할당 명령을 가하면 어떻게 되는가?

가. 자동 할당 연산자가 사용된다.
나. 복사 생성자가 사용된다.
다. 컴파일러 오류(Compiler Error)가 발생한다.
라. 실행 오류(Run-time Error)가 발생한다.

11 헤더 파일에 함수 프로토타입이 int day_of_week(int year, int month, int day);로 선언되어 있다. 다음과 같은 명령을 순차적으로 수행했을 때 이 중 함수 호출 몇 개가 잘못된 명령인가?

```
cout << day_of_week();
cout << day_of_week(2004);
cout << day_of_week(2001, 10);
cout << day_of_week(1995, 10, 4);
```

가. 모두 제대로다.　　나. 한 개　　다. 두 개　　라. 세 개　　마. 모두 잘못되었다.

12 다음의 클래스 인터페이스 정의를 기준으로 할 때, 오류가 없는 것을 고르라.

```
class Date {
public:
          Date (int m, int d, int y);
          int Day ();
private:
          int month, day, year;
}
```

가. Date d(7,14,2002); cout << Day( );
나. Date d; d.Date(7,14,2002);
다. Date d(7,14,2002); cout << d.year;
라. Date d(7,14,2002); cout << d.Month( );

13 다음 클래스 선언을 참고로 물음에 답하라.

```
class Hello {
public:
          void Greet (string name);
};
```

이 함수가 불려지면 Greet라는 멤버 함수는 Hello, <name>을 출력한다. 여기서 <name>은 파라미터로 넘겨진 사람 이름을 뜻한다. 화면에 사람 이름을 입력하면 Hello <name>을 출력하는 완전한 프로그램을 작성하라.

14 클래스 복소수(Complex Number)를 정의하기 위한 다음 선언을 참조하여 아래 명령어의 출력을 써보라.

```
class Complexity {
public:
          Complexity ();
          Complexity (int, int);
          Complexity AddTo (Complexity);
          Complexity operator += (Complexity c);
          void Print ();
private:
          int myReal;
          int myImaginary;
};
```

```
Complexity::Complexity ()
{  myReal = 0;
   myImaginary = 0;
}

Complexity::Complexity (int r, int i)
{  myReal = r;
   myImaginary = i;
}

Complexity Complexity::AddTo (Complexity c)
{  myReal += c.myReal;
   myImaginary += c.myImaginary);
   return Complexity (myReal, myImaginary);
}

Complexity Complexity::operator += (Complexity c)
{ return AddTo (c);
}

void Complexity::Print ()
{  cout << myReal;
   if (myImaginary >= 0)
       cout << '+';
   cout << myImaginary << 'i' << endl;
}
```

가. Complexity c; c.Print( );
나. Complexity c(1,2); c.Print( );
다. Complexity c(1,-2); c.Print( );
라. Complexity a(1,2), b(3,4); a.AddTo(b); a.Print( );
마. Complexity a(1,2), b(-3,-4); a+=b; a.Print( );

15 자동 기본 생성자(Automatic Default Constructor)란 무엇인지 인터넷 자료를 찾아서 발표하라.

16 다음 C++ 프로그램이 화면에 무엇을 출력한다면 그것은 무엇인가? 만약, 이 프로그램이 실행시간 오류를 일으킨다면 어디서, 왜 일어나는지 지적하라.

```
#include <iostream.h>
class BaseClass {
public:
          char * getMessage() {
          return "Good morning, World!";
          }
};

class SubClass: public BaseClass {
public:
          char * getMessage() {
          return "Good evening, World!";
          }
};

int main(){
          BaseClass b, *p;
          SubClass s;
          p = &s;
          cout << p->getMessage() << endl;
          return 0;
}
```

17 다음 클래스 선언이 주어졌을 때, Pen::special과 ColorPen::CPspecial의 라인별로 X, P, CP, D라고 표시하라. 단, 멤버 함수 호출이 있든 없든 간에 해당 라인이 컴파일되지 않을 때는 X, 이 멤버 함수의 Pen 버전을 호출할 때에는 P, 이 멤버 함수의 ColorPen 버전을 호출할 때에는 CP, 컴파일은 되지만 멤버 함수를 호출하지 않을 때에는 D라고 표시한다.

```
class Pen {
public:
          Pen();
          void move(int x, int y);
          void penUp();
          void penDown();
          void special();
protected:
          int getX ();
          int getY ();
private:
```

```
            int x, y;
            bool isUp;
};

void Pen::special()
{   penDown();
    int savex = getX();
    move(3,5);
    bool saveStat = isUp;
    setColor(32);
    int c = color;
}

class ColorPen : public Pen {
public:
            ColorPen();
            void move(int x, int y);
            void setColor(int color);
            void CPspecial();
protected:
            int color;
};

void ColorPen::CPspecial()
{   penDown();
    int savex = getX();
    move(3,5);
    bool saveStat = isUp;
    setColor(32);
    int c = color;
}
```

18 객체지향 프로그래밍(OOP, Object Oriented Programming)은 "스위치문이 없는 프로그래밍"으로 알려져 있다. 객체지향의 무엇이 이를 가능하게 하는지 한 단어로 답하라.