인터넷미디어전공 실습 II
(C++)

강의노트 #6: 6장 클래스 특징 II
- 콜론(:) 초기화
- const 와 static
- 포함된 객체(embeded object)
- 인라인(in-line) 함수
- 실습문제

2007. 4. 11

담당교수: 조 재 수

E-mail: jaesoo27@kut.ac.kr

학습 내용

- 콜론(:) 초기화
- const 와 static
- 포함된 객체(embeded object)
- 인라인(in-line) 함수
- 헤더파일과 구현파일의 분리
- 클래스에 포함되는 다른 내용(enum, typedef)
- 실습문제

콜론 [:] 초기화

- 생성자 함수에서 대입연산자를 이용한 초기화

```
class Date
{
      int year, month, day;
  public:
    Date(int yy, int mm, int dd);
     ....
};


Date::Date(int yy, int mm, int dd)
{
  year = yy;
  month = mm;
  day = dd;
}
```

- 생성자 함수에서 콜론(:) 연산자를 이용한 초기화

```
class Date
{
      int year, month, day;
  public:
    Date(int yy, int mm, int dd);
     ....
};
Date::Date(int yy, int mm, int dd) : year(yy), month(mm),
      day(dd)
{
 }

또는 클래스 선언 내에서,
class Date
{
      int year, month, day;
  public:
    Date(int yy, int mm, int dd) : year(yy), month(mm), day(dd)
    {
     };
     ....
};
```


콜론 [:] 초기화

Date(int yy, int mm, int dd) : year(yy), month(mm), day(dd)

- 콜론(:) 초기화를 멤버 이니셜라이저(member initializer)라고 하기도 한다.
- 멤버 이니셜라이져의 의미는 멤버 변수 year를 매개 변수 yy로, 멤버 변수 month을 매개변수 mm으로, 멤버 변수 day를 dd로 초기화하라는 뜻
- 이러한 멤버 이니셜라이저는 생성자의 몸체 부분보다 먼저 실행된다는 특징
- const 멤버 변수를 초기화할 수 있다는 특징도 지닌다.
- const 멤버 변수는 반드시 이 콜론 초기화(멤버 이니셜라이져)를 이용해서 초기화해야 한다.

콜론(:) 초기화의 유용성

- 클래스 내부에 포함되어 있는 객체 초기화
- const 키워드로 멤버 변수가 상수화된 경우 생성자 함수 초기화

콜론(:) 초기화의 유용성

```
class File
{
   Date d1; // 포함된 객체

 public:
   File(); // 생성자 함수
    .....
};
```

- 클래스 내부에 포함되어 있는 객체 초기화
  - 콜론(:) 연산자를 사용하여 데이터 멤버를 초기화하는 경우에 Date 클래스의 d1 객체가 생성될 때 3개의 매개변수를 갖는 Date 클래스의 생성자를 호출

```
class File
{
    Date d1;
 public:
   File(...) : d1(2005, 9, 29)
   {
    };
    .....
};
```

콜론(:) 초기화의 유용성

- 콜론(:) 연산자를 사용하는 경우, 포함된(embeded) 객체의 초기값으로, 둘러싸는(surrounding) 클래스의 생성자에서 넘겨지는 매개변수 값을 그대로 지정할 수도 있다.

```
class File
{
     Date d1;

  public:
    File(char* filename, int yy, int mm, int dd) : d1(yy, mm, dd)
    {
      };
     .....
};
```



```
/* 예제 6-1.cpp */
#include<iostream>
using namespace std;

class Student
{
   const int id;  // id를 상수화
    int age;
   char name[20];
   char major[30];

public:
 Student(int _id, int _age, char* _name, char* _major)
 {
      id=_id;  // 에러 발생
      age=_age;
      strcpy(name, _name);
      strcpy(major, _major);
 }

 void ShowData()
 {
      cout<<"이름: "<<name<<endl;
      cout<<"나이: "<<age<<endl;
      cout<<"학번: "<<id<<endl;
      cout<<"학과: "<<major<<endl;
 }
};

int main()
{
  Student Kim(200577065, 20, "Hong Gil Dong", "Computer Eng.");
  Student Hong(200512065,19,"Kim Sam Soon","Electronics Eng.");

  Kim.ShowData();
  cout<<endl;
  Hong.ShowData();

  return 0;
}
```

- 생성되는 객체마다 고유한 값으로 상수화
- 멤버 변수 앞에 const 키워드를 선언
  -> 멤버 변수의 상수화
- 컴파일 에러 이유는 const 로 선언된 변수가 생성자 함수가 호출되면서 id가 계속해서 변하게 되는 초기화를 하고 있기 때문

- 상수화 된 멤버 변수를 초기화 하는 방법이 바로 콜론(:)을 이용한 초기화 방법 또는 멤버 이니셜라이져를 이용하는 것이다.

콜론(:) 초기화의 유용성

```
class Student
{
    const int id;  // id를 상수화
    int age;
    char name[20];
    char major[30];

  public:
    Student(int _id, int _age, char* _name, char* _major) : id(_id)
    {
        age=_age;
        strcpy(name, _name);
        strcpy(major, _major);
    }
    void ShowData()
    {
        cout<<"이름: "<<name<<endl;
        cout<<"나이: "<<age<<endl;
        cout<<"학번: "<<id<<endl;
        cout<<"학과: "<<major<<endl;
    }
};
```


2. const 멤버함수 와 const 객체

```
class Date
{
  ...
   public:
       int GetYear(void) const;
       int GetMonth(void) const;
       int GetDay(void) const;
       void DisplayDate(void) const;
   ...
}
```

. const 멤버함수란 어떤 의미인가? 클래스 선언시, 멤버함수에서 객체의 데이터 멤버를 변경시킬 수 없도록 한다.

•Date 클래스에서 GetYear, GetMonth, GetDay, DisplayDate 멤버함수는 각각 const 멤버함수로 선언되어 있다.

• const 멤버함수를 만들려면 멤버함수의 매개변수 다음에 const 라는 예약어를 붙여주면 된다.

• 이때 const 예약어는 멤버함수를 정의할 때도 사용해야 한다.

```
// 예제 6-2.cpp 컴파일 에러가 있는 예제
1:  #include <iostream>
2:  using namespace std;
3:
4:  class Count
5:  {
6:     int cnt;
7:  public :
8:    Count() : cnt(0){}
9:    int* GetPtr() const{
10:         return &cnt;    // Compile Error
11:   }
12:
13:   void Increment(){
14:         cnt++;
15:   }
16:
17:   void ShowData() const {
18:         ShowIntro();  // Compile Error
19:         cout<<cnt<<endl;
20:   }
21:   void ShowIntro() {
22:         cout<<"현재 count의 값 : "<<endl;
23:   }
24: };
25:
26: int main()
27: {
28:   Count count;
29:   count.Increment();
30:   count.ShowData();
31:
32:   return 0;
33: }
```

- 컴파일 하면 10번째 줄과 18번째 줄에서 컴파일 에러를 발생
- 상수화된 함수는 상수화되지 않은 함수 호출을 허용하지 않는다.
- 상수화된 함수는 멤버 변수의 포인터를 리턴하는 것도 허용하지 않는다.
- 상수화된 함수에서 이러한 것을 허용하지 않게 하는 이유는 포인터를 전달받은 영역에서는 포인터를 이용하여 멤버변수의 조작이 가능하기 때문이다.
- 다른 함수를 호출하여 그 함수에서 데이터를 조작할 수 있기 때문에 상수함수에서는 이러한 것을 허용하고 있지 않다.
- 예제에서 ShowIntro() 함수에서 실제 멤버 변수의 변경이 없음에도 불구하고, 멤버 변수를 조작할 가능성을 지니고 있기 때문에 ShowIntro() 함수의 호출을 허용하지 않는 것이다.
- 컴파일 에러가 발생하지 않도록 수정해 보자?


```
// 예제 6-3.cpp 컴파일 에러를 수정
1:  #include <iostream>
2:  using namespace std;
3:
4:  class Count
5:  {
6:     int cnt;
7:  public :
8:    Count() : cnt(0){}
9:    const int* GetPtr() const{
10:         return &cnt;
11:   }
12:
13:   void Increment(){
14:         cnt++;
15:   }
16:
17:   void ShowData() const {
18:         ShowIntro();
19:         cout<<cnt<<endl;
20:   }
21:   void ShowIntro() const {
22:         cout<<"현재 count의 값 : "<<endl;
23:   }
24: };
25:
26: int main()
27: {
28:   Count count;
29:   count.Increment();
30:   count.ShowData();
31:
32:   return 0;
33: }
```

- 먼저 상수함수에서 포인터를 리턴할 때 그 포인터를 이용한 변경이 불가능하도록 const 키워드를 이용하여 const 포인터를 넘겨준다.
- 상수 함수에서 다른 함수를 호출할 때 그 함수를 상수 함수로 만들어 멤버 변수의 조작이 없음을 컴파일러에게 알려주면 된다.

```
// 예제 6-4.cpp const 객체 예제

#include<iostream>
 using namespace std;

 class Student
 {
     int id;
      int age;
     char name[20];
     char major[30];
  public:
     Student(int _id, int _age, char* _name, char*
       _major)
   {
         id=_id;
         age=_age;
         strcpy(name, _name);
         strcpy(major, _major);
   }

     void SetMajor(char * _major){
      strcpy(major, _major);
    }

   void ShowData()
   {
        cout<<"이름: "<<name<<endl;
        cout<<"나이: "<<age<<endl;
        cout<<"학번: "<<id<<endl;
         cout<<"학과: "<<major<<endl;
   }
};
```

```
int main()
{
   const Student Kim(200577065, 20, "Kim", "Computer
     Eng.");

   Kim.SetMajor("Internet Eng.");
   Kim.ShowData();
  cout<<endl;

  return 0;
}
```

- Const 객체란? Const 객체 선언은 일반변수에 대한 const 선언과 같이 객체를 상수화하는 것이다.
- 컴파일 에러가 발생하는 이유는 const 객체 Kim의 멤버 변수를 변경할 수 없기 때문
- const 객체는 const 함수가 아닌 ShowData()를 호출할 수 없다.
- ShowData() 함수를 상수화하면 컴파일 에러 없이 호출 가능하다.
- 컴파일 에러가 발생하지 않도록 수정해 보자.

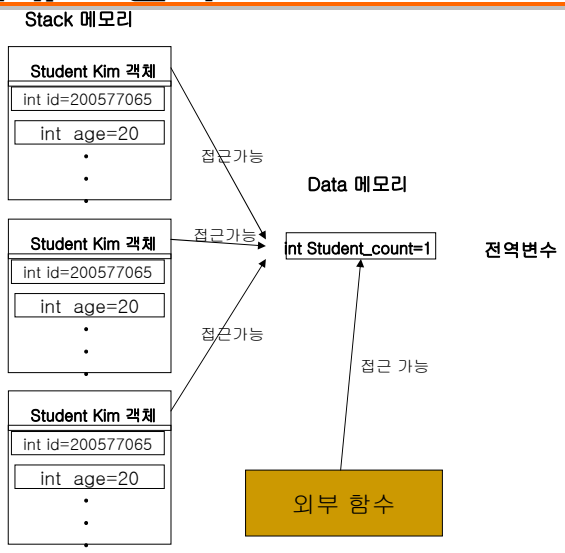
3. Static 클래스 멤버

```
/* 예제 6-5.cpp */
#include<iostream>
using namespace std;

int Student_count = 0;  // 전역변수

class Student
{
  int id;
  int age;
  char name[20];
  char major[30];
public:
     Student(int _id, int _age, char* _name, char* _major)
     {
       id=_id;
       age=_age;
       strcpy(name, _name);
      strcpy(major, _major);
      Student_count++;
      cout << Student_count << " 번째 Student 객체 생성" << endl;
     }

int main()
{
   Student Kim(200577065, 20, "Kjm", "Computer Eng.");
   Student Cho(200577067, 21, "Cho", "Multimedia Eng.");
   Student Hong(200577068, 222, "Hong", "Internetsoft Eng.");


   return 0;
}
```

- 클래스 멤버 변수를 static으로 선언하는 것이 어떠한 의미를 지니는 것인가?
- C++가 static 멤버 변수를 지원하는 이유는 전역변수의 사용을 피하기 위해서이다.
- 전역변수를 사용하는 클래스는 OOP와 C++의 기본적인 캡슐화 원칙에 위배되는 것이다.
- 예제6.5.cpp은 C++에서 전역변수가 필요한 상황을 보여준다.
- 실행결과

1번째 Student 객체 생성
2번째 Student 객체 생성
3번째 Student 객체 생성

Static 클래스 멤버



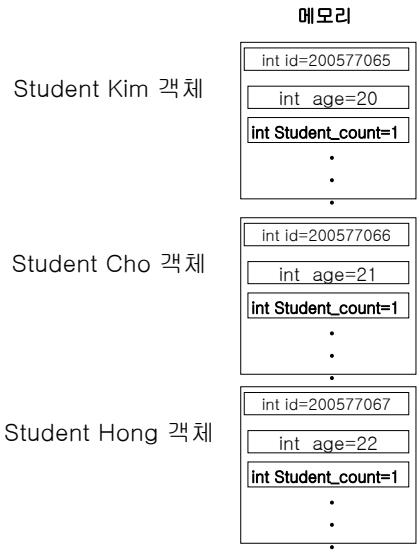
[그림 6-1] 개별 객체와 전역변수와의 메모리 구조


```
/* 예제 6-6.cpp */
#include<iostream>
using namespace std;


class Student
{
 int id;
 int age;
 int Student_count;


 char name[20];
 char major[30];


public:
Student(int _id, int _age, char* _name, char* _major)
{
    Student_count = 0;  // 초기 0으로 초기화
    id=_id;
    age=_age;
    strcpy(name, _name);
    strcpy(major, _major);


    Student_count++;
    cout << student_count <<  " 번째 Student 객체 생성" << endl;
  }
};


int main()
{
    Student Kim(200577065, 20, "Kjm", "Computer Eng.");
    Student Cho(200577067, 21, "Cho", "Multimedia Eng.");
    Student Hong(200577068, 22, "Hong", "Internetsoft Eng.");


      return 0;
}
```

- 실행결과는 다음과 같다.

1번째 Student 객체 생성
1번째 Student 객체 생성
1번째 Student 객체 생성

Static 클래스 멤버의 필요성



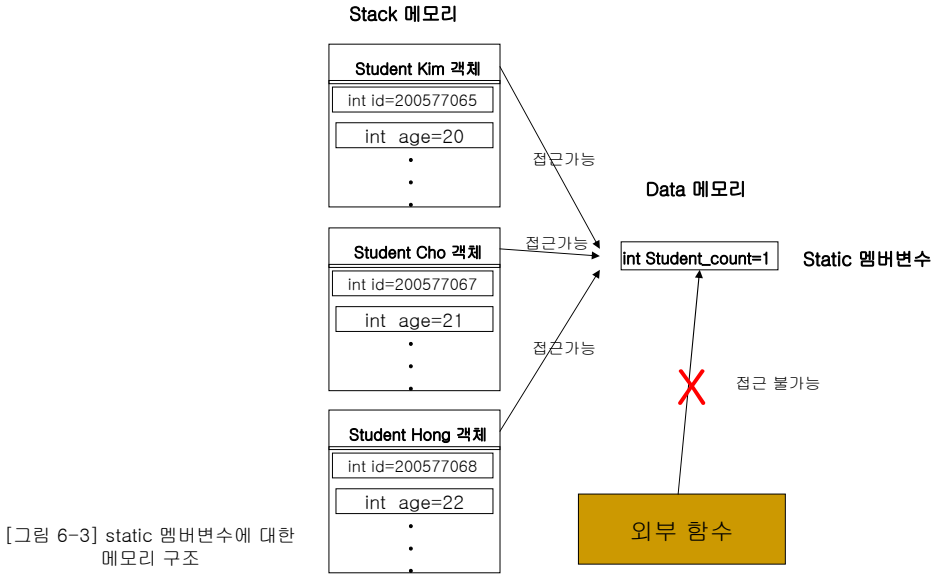
- 실행결과는 객체가 생성되면서 생성자 함수가 호출된다.
- 각 객체마다 Student_count가 0으로 초기화된 후, 1증가되어 객체가 생성될 때 마다 Student_count는 1이 된다.
- Student_count 멤버 변수는 각 객체가 1개를 공유하는 것이 아니라 그림 6-2와 같이 개별적으로 그 변수를 메모리에 독립적으로 가지고 있다.

[그림 6-2] Student 객체 멤버변수 Student_count에 대한 메모리 구조


```
/* 예제 6-7.cpp */
#include<iostream>
using namespace std;


class Student
{
 int id;
 int age;
 static int student_count;


 char name[20];
 char major[30];


public:
     Student(int _id, int _age, char* _name, char* _major)
     {
        id=_id;
        age=_age;
        strcpy(name, _name);
        strcpy(major, _major);


      student_count++;
     cout << Student_count <<  " 번째 Student 객체 생성"
      << endl;
     }

     void ShowData()
     {
          cout<<"이름: "<<name<<endl;
          cout<<"나이: "<<age<<endl;
          cout<<"학번: "<<id<<endl;
          cout<<"학과: "<<major<<endl;
     }
};
```

```
int Student::student_count = 0;  // static 멤버 변수 초기화


 int main()
 {
    Student Kim(200577065, 20, "Kjm", "Computer Eng.");
    Student Cho(200577067, 21, "Cho", "Multimedia Eng.");
    Student Hong(200577068, 222, "Hong", "Internetsoft
      Eng.");


       return 0;
}
```

Static 클래스 멤버의 필요성



[그림 6-3] static 멤버변수에 대한 메모리 구조

Static 클래스 멤버변수의 특징

- main 함수가 호출되기도 전에 메모리 공간에 올라가서 초기화 된다.
- public으로 선언이 된다면 객체 생성이전에도 접근이 가능
- 메모리 상에는 개별적인 객체의 멤버로 메모리를 할당되어 존재하는 것이 아니다. 선언되어 있는 클래스 내에서 직접 접근할 수 있는 권한이 부여
- static 멤버 변수의 생성자 함수에서 초기화하는 것이 아니고 전역변수와 같이 프로그램이 시작하면서 단 한 번만 초기화되어야 한다.
- private 속성을 갖는 static 멤버라도 클래스의 외부에서 초기화 할 수 있다.
- static 데이터 멤버의 초기화 구문에서 static 데이터 멤버의 데이터형이 함께 지정되어야 한다.

  예) int Student::Student_count = 0;
- static 데이터 멤버를 public 속성으로 지정하여 클래스 외부의 함수에서 접근해야 할 필요가 있다면 클래스명과 영역결정 연산자 '::'를 static 데이터 멤버 앞에 붙임으로써 개별적인 객체의 데이터 멤버의 값이 변경되는 것이 아니라 클래스의 전체 객체의 static 데이터 멤버의 값이 변경되는 것이라는 것을 명확히 해 줄 필요가 있다.
- 이러한 특징으로 인하여 특히 static 멤버 변수가 어느 한 객체의 멤버 변수가 아니기 때문에 **클래스 변수라고** 표현을 하기도 한다.

```
/* 예제 6-8.cpp */
#include <iostream>
using namespace std;

class myclass {
  public:
     static int i;
     void seti(int n) { i = n; }
       int geti() { return i; }
};

int myclass::i;  // static 멤버 초기화 구문

int main()
{
  // i를 myclass 객체 생성 없이 직접 설정한다.
  myclass::i = 100; // 어떤 객체 참조없이 전역변수처럼 접근가능
  cout << "i = " << myclass::i << '₩n'; // 100을 출력한다.

  myclass o1, o2;  // 객체 생성
  o1.i = 200;     // 객체 o1을 통한 static 변수 접근
  cout << "i = " << o1.i << '₩n'; // 200을 출력한다.

  o2.seti(300);   // 객체 o2의 멤버변수를 통한 static 변수 접근
  cout << "i = " << o2.i << '₩n'; // 300을 출력한다.
  cout << "o1.i = " << o1.i << endl;
  cout << "o2.i = " << o2.i << endl;
  cout << "myclass::i = " << myclass::i << endl;

  cout << "o1.i의 주소: " << &o1.i << endl;
  cout << "o2.i의 주소: " << &o2.i << endl;
  cout << "myclass::i의 주소: " << &myclass::i << endl;

  return 0;
}
```

- 실행결과

i=100
i=200
i=300
o1.i=300
o2.i=300
myclass::i=300
o1.i의 주소: 00477738
o2.i의 주소: 00477738
myclass::i의 주소: 00477738

☞ 참고로 static 멤버 초기화 구문을 넣어주지 않으면 컴파일 에러는 생기지 않지만, Build 과정에서 link 에러가 발생

static 멤버 변수의 필요성 및 사용 예

- 은행의 예금계좌를 의미하는 Account라는 클래스 작성에서
  - 고객명과 고객의 예금 잔액을 저장할 데이터 멤버
  - 현재의 **이자율에 따라 이자액을** 계산하여 그 금액을 예금 잔액에 추가하는 멤버함수
  - 클래스의 각 객체는 특정 고객의 예금 계좌를 나타낸다.
- 이자율을 어떻게 나타낼 것인가?
- 이자율은 수시로 변하기 때문에 당연히 상수를 사용할 수 없다.
- 현재의 이자율을 저장할 변수가 필요
  1. **이자율을 클래스의 데이터 멤버에 포함시키는 경우**
  2. **이자율을 전역변수에 저장하는 경우**
  3. static 데이터 멤버

static 멤버 변수의 필요성 및 사용 예

1. **이자율을 클래스의 데이터 멤버에 포함시키는 경우**
   - 각각의 객체에는 이자율을 저장하는 데이터 멤버를 갖게 된다.
   - 모든 객체에서 같은 이자율을 사용하기 때문에 똑같은 정보를 모든 객체에서 갖고 있다는 것은 메모리의 낭비가 심하다.
   - 모든 객체에 똑 같은 이자율을 초기화하는 것은 매우 번거로운 작업이 될 수 있다.
   - 이자율은 수시로 변하기 때문에 이자율이 변할 때 마다 모든 객체의 이자율을 수정해 주어야 한다는 부담이 있다.
   - 결론적으로 클래스에 이자율을 나타내는 데이터 멤버를 지정하는 것은 매우 비효율적이고, 또한 객체마다 이자율이 달라질 위험성까지 있다.

2. **이자율을 전역변수에 저장하는 경우**
   - 전역변수에 이자율을 저장한다면 클래스의 객체마다 이자율을 갖고 있어야 하는 불합리한 점은 없어진다.
   - 전역변수를 사용할 때는 예금 계좌 클래스의 객체에서 뿐만 아니라 그 외에 다른 함수에서도 이 변수에 접근할 수 있게 된다.
   - 우리가 지금까지 살펴본 전역변수의 위험성을 그대로 안게 되며, 이것은 객체지향 프로그래밍(OOP)의 기본적인 원리에 위배되는 방법이 된다.

static 멤버 변수의 필요성 및 사용 예

3. static 데이터 멤버
   - 하나의 클래스에 속해 있는 객체들에서만 사용할 수 있는 일종의 클래스 범위의 전역변수와 같은 성질을 갖는다.
   - 클래스에 속해 있는 모든 객체에서는 쉽게 접근이 가능하고 클래스의 외부 함수에서는 접근할 수 없다.
   - 모든 객체는 단지 하나의 같은 데이터를 공유한다.

```
/* 예제 6-9.cpp */
#include <iostream>
using namespace std;

class Account
{
      char *name;
      double amount;
      static double currentRate;
  public:
      Account();
      Account(char *_name,double _amount)
      {
        name = new char[strlen(_name)+1];
        strcpy(name, _name);
        amount = _amount;
      }
      ~Account(){
          delete name;
        }
      void earnInterest()
      {
          amount += currentRate * amount;
      }
      void showAmount()
      {
       cout << name << ": " << amount << " 이자율 : "
           << currentRate << endl;
      }
};
```

```
double Account::currentRate = 0.0015;  // static 멤버 변수 초기화


int main()
{
  Account Cho("Cho", 50000), Kim("Kim", 200000),
        Hong("Hong",1000);


  Cho.earnInterest();
  Kim.earnInterest();
  Hong.earnInterest();


  Cho.showAmount();
  Kim.showAmount();
  Hong.showAmount();


 return 0;
}
```

▪실행결과는 다음과 같다.

```
-----------------------------------------
Cho: 50075  이자율: 0.0015
Kim: 200300 이자율: 0.0015
Hong: 1001.5 이자율: 0.0015
-----------------------------------------
```

▪static 멤버는 모든 객체에서 필요한 공통되는 자원을 구현하거나 객체에 대한 상태 정보를 관리하는데 유용하게 사용 된다.

static 멤버 함수의 필요성 및 사용 예

```
class Airplane
{
    static int count;
  public:
      Airplane()
      {
        count++;
      }
      static int howMany()
      {
        return count;
      }
      ~Airplane()
      {
         count--;
      }
   };

   int Airplane::count = 0;
```

- 다음 코드는 어느 순간에 그 클래스의 객체가 얼마나 생성되었는지를 알려준다.
- static int howMany() 함수는 static 멤버함수를 의미한다.

Static 멤버 함수

```
#include<iostream>
 using namespace std;
class Account
{
      char *name;
      double amount;
      static double currentRate;
  public:
      Account();
      Account(char *_name,double _amount)
      {
        name = new char[strlen(_name)+1];
        strcpy(name, _name);
        amount = _amount;
      }
      void earnInterest()
      {
         amount += currentRate * amount;
      }
      static void SetInterestRate(double newValue){
          currentRate = newValue;
      }
      static double GetInterestRate(void)
      {
        return  currentRate;
      }
     void showAmount()
     {
       cout << name << ": " << amount << " 이자율 : "
          << currentRate << endl;
     }
};
```

- public 속성의 static 데이터 멤버에 접근하는 것과 마찬가지로 static 멤버 함수를 호출하는 방법도 두 가지가 있다.
- 하나는 다른 멤버 함수와 같이 객체 생성 후 객체에 '.' 연산자를 사용하여 호출하는 것
- 다른 하나는 멤버함수 앞에 클래스명과 영역 결정연산자 '::'를 붙여 호출하는 방법이 있을 수 있다.

```
int main(void)
{
   Account Kim("Kim", 40000), Cho("Cho", 30000);
  Account::SetInterestRate(0.002);
  Kim.showAmount();
 Cho.showAmount();

 Kim.SetInterestRate(0.002);
 Kim.showAmount();
Cho.showAmount();

 return 0;
}
```

- 명확히 클래스에 속해 있는 전체 객체에 대하여 static 데이터 멤버의 값이 변경된다는 것을 표현하기 위해 클래스를 이용하는 두 번째 방법을 사용하는 것이 좋다.

Static 멤버 함수

- static 멤버함수의 필요성은 무엇인가?

  예를 들어 SetInterestRate 멤버함수가 static으로 지정되어 있지 않다고 하자. 또한 아직 Account 클래스의 어떠한 객체도 생성되어 있지 않다면 어떻게 private 속성의 static 데이터 멤버 currentRate의 값을 변경시킬 것인가? 이런 경우에 static 멤버함수가 필요하게 된다.
- static 멤버함수는 클래스의 객체가 생성되지 않은 경우에도 사용할 수 있다.
- static 멤버함수는 아주 제한적으로 사용된다.
- static 멤버함수는 클래스의 특정 인스턴스 즉, 객체에 영향을 미치지 않으므로 this 포인터를 사용할 수 없다.
- static 멤버함수는 static 이 아닌 데이터 멤버에 접근할 수도 없고, static이 아닌 다른 멤버함수도 호출할 수 없다.

전역변수 vs. static 멤버 vs. 비 static 멤버에 대하여

- 우리가 객체 또는 클래스에 대하여 데이터에 접근하거나 함수를 구현할 때, 위와 같이 3가지 중 어떤 것으로 할지 선택을 할 수 있다.
  - 하나는 전역변수와 전역함수를 사용하는 것이고,
  - 다른 하나는 static 클래스 멤버를 사용하는 것,
  - 나머지 하나는 static이 아닌 클래스 멤버를 사용하는 것이다.

- 전역변수와 전역함수는 데이터 또는 함수가 전체 프로그램에 걸쳐 공유되면서 논리적으로 어떤 클래스에도 속하지 않을 때만 사용하여야한다.
- 전역변수나 전역함수를 사용할 때 반드시 지켜야 하는 점은 항상 사용을 최소화하여야 한다는 것이다.
- 보통 프로그램을 개발하는 과정에서 전역변수나 전역함수를 대체할 만한 클래스가 발견된다.

- static 이 아닌 멤버는 클래스의 각 객체에서 고유한 데이터를 저장하고 그 데이터에 대해 처리를 할 때 사용된다.
- static 멤버는 클래스에 속해 있는 객체들에 대해서 공통되는 데이터를 저장하고 그 데이터에 대해 처리를 할 때 사용된다.
- static 멤버는 일반적인 클래스 멤버와 전역변수 또는 전역함수의 중간적인 성격을 갖고 있다고 생각할 수 있다.

전역변수 vs. static 멤버 vs. 비 static 멤버에 대하여

- 참고로 많은 사람들이 static 멤버와 const 멤버와 혼동을 하게 된다.
- static 데이터 멤버는 static 멤버함수에서만 접근이 가능한 것으로 잘못 생각하기 쉽다.
- static이 아닌 멤버함수에서도 얼마든지 static 데이터 멤버에 접근할 수 있다.
- const 객체에서는 const 멤버함수만 사용할 수 있다는데서 그러한 혼동을 하게 된다.
- 단, static 멤버함수는 static 데이터 멤버에만 접근할 수 있다.

6.4 포함된 객체(embeded object)

- 한 클래스의 객체는 다른 클래스의 객체들과 분리되어 존재할 수 없으며, 항상 다른 객체와 관련을 맺고 있다.
- 이러한 다른 객체와의 관련성을 나타내는 것 중의 하나가 다른 객체를 데이터 멤버로 포함하고 있는 경우이다.
- 예를 들어, 사각형을 표현하는 방법을 생각해 보자.
- 이 경우에 우리는 사각형을 표현하는 클래스를 작성할 때 모두 3 가지 방법을 사용할 수 있다.
  1. 사각형은 좌측 상단 위치와 우측 하단 위치를 나타내는 정보로 표현가능: 4개의 int 변수로 표현
  2. 2개의 좌표로 표현가능: 좌상단 좌표와 우하단 좌표
  3. 3개의 변수로 표현가능: 좌측 상단 위치와 폭과 높이

포함된 객체(embeded object)

1. 4개의 좌표로 표현

```
class Rectangle
{
    int x1, y1, x2, y2;
public:
    .......
};
```

2. 두개의 좌표로 표현

```
class Rectangle
{
    Point leftupper;
    Point rightdown;
 public:
    ........
};
```

3. 1개의 좌표와 2개의 int 변수

```
class Rectangle
{
    Point leftupper;
    int width, height;
  public:
    ......
};

class Point
{
    int x, y;
 public:
    ......
};
```

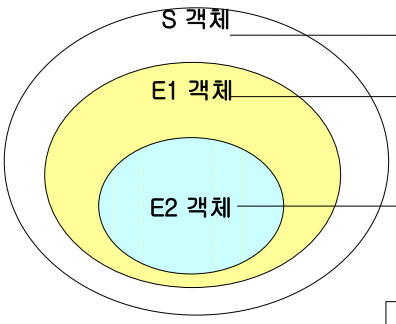
포함된 객체(embeded object)

```
class Point
{
    int _x, _y;
public:
    Point(int x = 0, int y = 0) : _x(x), _y(y)
    {
        cout << "Point 클래스의 생성자 함수 호출." << endl;
    }
    Point(const Point& point)
    {
       _x = point._x;
       _y = point._y;
       cout << " Point 클래스의 복사 생성자 함수 호출.“ << endl;
    }
   ~Point()
   {
      cout << " Point 클래스의 소멸자 함수 호출.“ << endl;
   }
   int GetX(void){   return _x; }
   int GetY(void){   return _y; }
   void SetX(int x){  _x = x; }
   void SetY(int y){  _y = y; }
};
```



```
class Rectangle
{
        Point _leftupper;
        int  _width, _height;
public:
    Rectangle() : _width(0), _height(0), _leftupper(0, 0)
   {
      cout << "Rectangle 클래스의 기본 생성자 함수 호출.“ << endl;
    }
  Rectangle(int width, int height, Point point) : _width(width),  _height(height), _leftupper(point)
 {
   cout << "Rectangle 클래스의 3개의 매개변수 생성자 함수 호출.“ << endl;
 }
 Rectangle(int width, int height, int x, int y) : _width(width),  _height(height), _leftupper(x, y)
{
 cout << "Rectangle 클래스의 4개의 매개변수 생성자 함수 호출.“ << endl;
}
 ~Rectangle()
{
  cout << " Rectangle 클래스의 소멸자 함수 호출.“ << endl;
}
void SetCorner(Point point){   _leftupper = point; }
Point GetCorner(void){  return _leftupper; }
void SetSize(int width, int height){
      _width = width;
      _height = height;
}
void display(void)
{
   cout << "Rectangle at x:  " << _leftupper.GetX()  << "y:  " << _leftupper.GetY();
  cout << "Height :  " << _height << " Width: " << _width << endl;
}
};
```

포함된 객체를 갖는 클래스의 생성과 소멸 순서



둘러싸는(surrounding) 객체

포함된 (enbeded) 객체

가장 내부에 포함된 (enbeded) 객체

S 객체 생성시

객체생성순서: E2 -> E1 -> S
객체소멸순서: S -> E1 -> E2

포함된 객체를 갖는 클래스의 생성과 소멸 순서

```
#include <iostream>
#include "embeded.h"
//  Rectangle Class와 Point Class 포함한
    헤더파일


int main(void)
{
    Point p1(25, 35);
    cout << endl;


   Rectangle r1(1,2,p1);
   r1.display();
   cout << endl;


   return 0;
}
```

- 실행결과는 다음과 같다.

Point 클래스의 생성자 함수 호출.

Point 클래스의 복사 생성자 함수 호출. (1)
Point 클래스의 복사 생성자 함수 호출. (2)
Rectangle 클래스의 3개의 매개변수 생성자 함수 호출.
Point 클래스의 소멸자 함수 호출 (3)
Rectangle at x: 25 y: 35 Height: 2 Width: 1

Rectangle 클래스의 소멸자 함수 호출. (4)
Point 클래스의 소멸자 함수 호출. (5)
Point 클래스의 소멸자 함수 호출. (6)

6.5 인라인(inline) 함수

- C++에서는 함수가 실제로 호출되는 대신에, 호출하는 그 라인에 삽입하도록 함수를 정의하는 것이 가능
- 인라인 함수란 호출하는 그 라인에 삽입하도록 함수를 정의하는 것을 말하고 장점은 함수를 호출하고 값을 반환하는 메커니즘과 관련된 오버헤드(overhead)가 없다.
- 단점으로는 인라인 함수가 길고 자주 호출되면 프로그램이 거대해진다는 것
- 이런 이유 때문에 일반적으로 짧은 함수는 인라인 함수로 선언
- 인라인 함수를 선언하기 위해서는 함수 정의 앞에 inline 지정자를 붙이면 된다.

6.5 인라인(inline) 함수

```
inline void Func1()
{
  cout << " 여곳은 인라인 함수1입니다.₩n";
  cout << " 인라인 함수를 사용하는 효과는?₩n";
}

inline void Func2()
{
  cout << " 여곳은 인라인 함수2입니다.₩n";
  cout << " 인라인 함수를 사용하는 효과는?₩n";
}

int main()
{
 // inline 함수를 호출한다.
 cout << "인라인 함수 1을 호출" << endl;
 Func1();

 cout << "인라인 함수 2를 호출" << endl;
 Func2();
 return 0;
}
```

컴파일러

```
int main()
{
 // inline 함수를 호출한다.
 cout << "인라인 함수 1을 호출" << endl;
 // Func1();
  cout << " 여곳은 인라인 함수1입니다.₩n";
  cout << " 인라인 함수를 사용하는 효과는?₩n";

 cout << "인라인 함수 2를 호출" << endl;
 //  Func2();
  cout << " 여곳은 인라인 함수2입니다.₩n";
  cout << " 인라인 함수를 사용하는 효과는?₩n";

 return 0;
}
```

[그림 6-5] 인라인(inline) 함수의 효과

인라인(inline) 함수

```
// 인라인 함수의 예
#include<iostream>
using namespace std;


inline int even(int x)
{
   return !(x%2);
    }

int main()
{
if (even(10)) cout << "10 is even₩n";
if (even(11)) cout << "11 is even₩n";


return 0;
}
```

- 예제는 inline을 사용할 때의 중요한 사실 하나를 지적하는데, 인라인 함수는 처음 호출되기 전에 선언되어야 한다는 것
- 만약 그렇게 하지 않는다면, 컴파일러는 함수가 인라인 함수로 확장될 것이라는 것을 알 수가 없다.
- 이것이 바로 even() 함수가 main() 함수 이전에 선언된 이유이다.

인라인(inline) 함수

- 매개변수를 가진 매크로보다 inline을 사용할 때의 이점은 두 가지가 있다.
  - 첫째로, 짧은 함수를 인라인으로 확장하기 위한 좀 더 구조적인 방법을 제공한다.
  - 두 번째로, 인라인 함수는 매크로 확장보다 컴파일러에 의해 더욱 최적화될 수 있다.
- C++ 프로그래머는 짧은 함수 호출과 관련된 오버헤드를 피하기 위해, 매개변수를 갖는 매크로를 사용하는 대신 inline 함수를 사용한다.
- 중요한 점은, inline 지정자는 컴파일러에게 요청하는 것이지 명령하는 것이 아니라는 것이다.
- 만약 이런저런 이유로 컴파일러가 요청을 완수하지 못한다면 보통의 함수처럼 컴파일 되고, inline 요청은 무시된다.

```
// 예제 6-1, 인라인된 멤버 함수를 설명한다.

#include<iostream>
using namespace std;


class samp {
      int i, j;
public:
  samp(int a, int b);
  int divisible(); // 정의에서 인라인으로지정된다.
};


samp::samp(int a, int b)
{
  i = a;
  j = b;
}

/* i가 j로나누어떨어지면 1을반환한다. 이 멤버함수는 인라인으로 확장된다. */

inline int samp::divisible()
{
  return !(i %j);
}

int main()
{
  samp ob1(10, 2), ob2(10, 3);

// 이것은 참이다.
 if (ob1.divisible())
  cout << "10 divisible by 2₩n";
  // 이것은 거짓이다.
 if (ob2.divisible())
  cout << "10 divisible by 3₩n";

  return 0;
}
```


6.6 헤더파일과 구현파일의 분리

- 클래스 디자인 과정에서 클래스 정의를 헤더파일과 구현파일에 나누어 기술하면 소스코드가 간결해지고, 소스코드의 관리 및 재사용이 용이한 장점

6.6 헤더파일과 구현파일의 분리

```
// 파일명: Example.cpp
#include <iostream>
using namespace std;

// Point 클래스를 정의한다.
class Point
{
      // 멤버 변수
      int x, y;
public:
      // 멤버 함수
      void Print();

        // 생성자들
      Point();
      Point(int initialX, int initialY);
      Point(const Point& pt);

      // 접근 함수
      void SetX(int value)
      {
            if (value < 0)
                  x = 0;
            else if (value > 100)
                  x = 100;
            else
                  x = value;
      }
      void SetY(int value)
      {
            if (value < 0)
                  y = 0;
            else if (value > 100)
                  y = 100;
            else
                  y = value;
      }
      int GetX() {return x; };
      int GetY() {return y; };
};
```

```
Point::Point(const Point& pt)
{
      x = pt.x
      y = pt.y
}


Point::Point(int initialX, int initialY)
{
      SetX(initialX);
      SetY(initialY);
}


Point::Point()
{
      x = 0;
      y = 0;
}
void Point::Print()
{
      cout << "( " << x << ", " << y << ") ₩n";
}



int main()
{
      // 객체를 생성한다.
      Point pt(-50, 200);


      // pt의 내용을 출력한다.
      pt.Print();


      return 0;
}
```


6.6 헤더파일과 구현파일의 분리

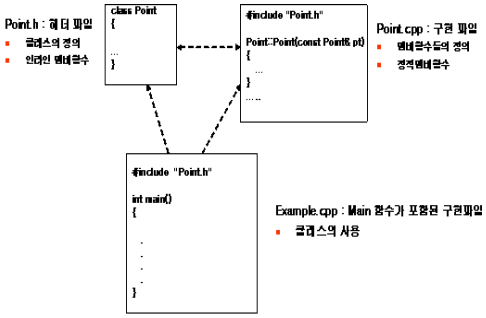
```
// 파일명: Point.h
#ifndef  POINT_H  // 헤더파일이 중복해서 포함
       되는 경우를 위한 처리
#define  POINT_H


// Point 클래스를 정의한다.
class Point
{
      // 멤버 변수
      int x, y;
public:
      // 멤버 함수
      void Print();

      // 생성자들
      Point();
      Point(int initialX, int initialY);
      Point(const Point& pt);

      // 접근 함수
      void SetX(int value)
      {
            if (value < 0)
                  x = 0;
            else if (value > 100)
                  x = 100;
            else
                  x = value;
      }
      void SetY(int value)
      {
            if (value < 0)
                  y = 0;
            else if (value > 100)
                  y = 100;
            else
                  y = value;
      }
      int GetX() {return x; };
      int GetY() {return y; };
};
#endif
```

```
// 파일명: Point.cpp
#include "Point.h"
#include <iostream>
using namespace std;


Point::Point(const Point& pt)
{
      x = pt.x
      y = pt.y
}


Point::Point(int initialX, int initialY)
{
      SetX(initialX);
      SetY(initialY);
}


Point::Point()
{
      x = 0;
      y = 0;
}
void Point::Print()
{
      cout << "( " << x << ", " << y << ")
       ₩n";
}
```

```
// 파일명: Example.cpp
#include "Point.h"


int main()
{
      // 객체를 생성한다.
      Point pt(-50, 200);


      // pt의 내용을 출력한다.
      pt.Print();


      return 0;
}
```

6.6 헤더파일과 구현파일의 분리

- ☞ Point.h 파일: Point 클래스의 정의 부분이 포함되었다.
- ☞ Point.cpp 파일: Point 클래스에서 정의된 멤버함수들을 포함하고 있다.
- ☞ Example.cpp 파일: Point 클래스를 사용해서 활용하는 코드들을 포함하고 있다.



[그림 6-6] 분리된 세파일의 관계

6.6 헤더파일과 구현파일의 분리

- 헤더파일과 구현파일로 분리하면 다음과 같은 장점이 있다.
  - 소스코드가 간결해져서 전체 프로그램을 이해하기 쉽다.
  - 관련된 내용이 함께 모여 있으므로 필요한 부분을 찾아보기 쉽다.
  - 소스코드의 관리와 재사용이 용이하다.

6.7 클래스에 포함되는 다른 내용

```
// 파일명: Point.h
#ifndef  POINT_H   // 헤더파일이 중복해서 포함되는
        경우를 위한 처리
#define  POINT_H


// Point 클래스를 정의한다.
class Point
{
      // 멤버 변수
      int x, y;
public:
     enum { MIN_X = 0, MAX_X = 100, MIN_Y = 0,
       MAX_Y = 100};


      // 멤버 함수
      void Print();


      // 생성자들
      Point();
      Point(int initialX, int initialY);
      Point(const Point& pt);
```

```
      // 접근 함수
      void SetX(int value)
      {
            if (value < MIN_X)
                  x = MIN_X;
            else if (value > MAX_X)
                  x = MAX_X;
            else
                  x = value;
      }
      void SetY(int value)
      {
            if (value < MIN_Y)
                  y = MIN_Y;
            else if (value > MAX_Y)
                  y = MAX_Y;
            else
                  y = value;
      }
      int GetX() {return x; };
      int GetY() {return y; };


};
#endif
```


6.7 클래스에 포함되는 다른 내용

```
/ 파일명: Point.h
#ifndef  POINT_H   // 헤더파일이 중복해서
      포함되는 경우를 위한 처리
#define  POINT_H


// Point 클래스를 정의한다.
class Point
{
 public:
     enum { MIN_X = 0, MAX_X = 100, MIN_Y
       = 0, MAX_Y = 100};
     typedef int COOR_T; // 좌표의 타입


      // 멤버 함수
      void Print();


      // 생성자들
      Point();
      Point(COOR_T initialX, COOR_T
      initialY);
      Point(const Point& pt);
```

```
    // 접근 함수
    void SetX(COOR_T value)
    {
          if (value < MIN_X)
                x = MIN_X;
          else if (value > MAX_X)
                x = MAX_X;
          else
                x = value;
    }
   void SetY(COOR_T value)
   {
          if (value < MIN_Y)
                y = MIN_Y;
          else if (value > MAX_Y)
                y = MAX_Y;
          else
                y = value;
    }
     COOR_T GetX() {return x; };
     COOR_T GetY() {return y; };
private:
// 멤버 변수
  COOR_T x, y;
};
#endif
```

6.7 클래스에 포함되는 다른 내용

```
/// 파일명: Point.cpp

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

Point::Point(const Point& pt)
{
    x = pt.x
    y = pt.y
}
Point::Point(COOR_T initialX, COOR_T initialY)
{
    SetX(initialX);
    SetY(initialY);
}
Point::Point()
{
    x = 0;
    y = 0;
}
void Point::Print()
{
    cout << "( " << x << ", " << y << ") ₩n";
}
```

- 멤버변수 x, y의 타입과 관련된 것은 모두 COOR_T 형으로 변환했다.
- 이 소스코드는 기존의 소스코드와 완벽하게 동일한 것이다. COOR_T가 결국은 int 형이기 때문이다.
- 하지만 나중에 x, y의 형을 float로 바꾸고 싶다면 다음과 같이 수정하면 된다.

typedef float COOR_T;

- 이렇게 멤버함수와 멤버변수만이 아니라 열거체(enum), typedef 같은 것들도 클래스와 관련된 것이라면 클래스의 정의 안쪽에 위치시키는 것이 좋다.

실습문제

- 은행계좌관리시스템 version 2.0을 version 2.1로 upgrade 해 보자.
  - const 또는 static 키워드를 이용하여 프로그램을 좀더 안정되게 해 보자.
  - 모든 계좌에 이자율(0.035)을 추가하여, 현재 금액에서 이자율을 계산한 잔액을 출력하는 함수를 추가해 보자. (단 여기서 이자율은 기간과 무관하게 현재의 잔액에 대하여만 생각하자.)

질문 & 답변

Thank You !

수고하셨습니다.