C++ 스터디

7. 클래스(2)

D.Com

Passing Object

- 벡터나 객체를 전달할 때!
  - C++의 기본 설정은 “pass by value”이다. 이는 객체나 벡터를 복사하여 전달하기에 메모리를 많이 차지하고 프로그램의 실행 속도를 줄이게 된다. (벡터나 객체가 엄청나게 크다고 생각해봐라!)
  - 그러므로 “pass by reference”를 이용해 객체나 벡터의 값들은 그대로 두고 주소 값 만을 전달해줌으로써 효율적인 프로그램 작성이 가능하다.
  - “pass by reference”를 사용하면서 의도치 않은 함수 내의 값 변경을 막아 안정성을 위해서는 값을 수정하지 않는 함수는 const를 사용하여 수정이 불가능하게 해준다.

```
void print_fraction(const SimpleRational& f) {
    std::cout << f.get_numerator() << "/" << f.get_denominator();
}
```

Passing Object

- 다른 함수(예를 들어 위의 print_fraction)에서 객체(위의 f)를 받아올 때, 이 객체의 멤버 함수 또한 const로 선언해야만 사용 가능하다.
- 즉 const 키워드를 사용한 객체는 멤버 함수 또한 const 함수만 호출할 수 있다.

```
1   #include <iostream>
2   using namespace std;
3
4   class Point {
5   private:
6       double x;
7       double y;
8   public:
9       Point(double _x = 0, double _y = 0) :
10          x(_x), y(_y) {};
11
12      double getX() const { return x; }
13      double getY() const { return y; }
14  };
15
16  void print(const Point& p) {
17      cout << p.getX() << endl;
18      cout << p.getY() << endl;
19  }
```

D.Com

객체의 포인터

- 기존과 같은 방식으로 객체의 포인터 또한 이용할 수 있다.
- 포인터 정의

  Point* pt2;
- Operator '&'

  pt2 = &pt1;
- Operator '*'

  (*pt2).x = 1.0;

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

class Point {
public:
    double x;
    double y;
};

int main() {
    Point pt1;
    Point* pt2;

    pt2 = &pt1;

    pt1.x = 100.0;
    pt1.y = 200.0;

    cout << (*pt2).x << " : " << (*pt2).y << endl;

}
```

```
Microsoft Visual Studio
100 : 200

C:₩Users₩bluej₩Deskto
 종료되었습니다.
이 창을 닫으려면 아무
```

D.Com

객체의 포인터 (-> operator)

- Operator '->'을 이용하면 포인터 객체를 더 쉽게 이용가능 하다.
- 아래의 두 문장은 같다.

  (*pt2).x = 1.0;

  pt2-> x = 1.0;

- 마찬가지로 멤버 함수 또한 '->' 연산자를 사용해 이용 가능하다.

  (*pt2).set_x(1.0);

  pt2-> set_x(1.0);

```
1   #include <iostream>
2   using namespace std;
3
4   class Point {
5   public:
6     double x;
7     double y;
8   };
9
10  int main() {
11    Point pt1;
12    Point* pt2;
13
14    pt2 = &pt1;
15
16    pt1.x = 100.0;
17    pt1.y = 200.0;
18
19    cout << pt2->x << " : " << pt2->y << endl;
20
21  }
```

D.Com

객체의 포인터 (new, delete operator)

- 마찬가지로 기존과 동일하게 동적할당에 new와 delete 연산자를 사용 가능하다.
- 동적 할당

  Point* pt2;

  pt2 = new Point;
- 동적 할당 해제

  delete pt2;

- 이러한 객체의 동적할당은 후에 상속과 연계하여 다형성을 위해 사용된다.

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

class Point {
public:
    double x;
    double y;
};

int main() {
    Point* pt2;

    pt2 = new Point;

    pt2->x = 100.0;
    pt2->y = 200.0;

    cout << pt2->x << " : " << pt2->y << endl;

    delete pt2; // 동적할당 후 해제는 필수!!!
}
```

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

class Point {
    int x;
    int y;
};

int main() {
    Point* p_arr;
    p_arr = new Point[10];

    delete[] p_arr;
}
```

어레이와 함께 쓸 수도 있다.

D.Com

객체의 포인터 (this 포인터)

- this 포인터는 “그 객체 자신의 주소”를 나타낸다.
- 파라미터 중복을 피하고 주소 특정 등의 목적을 위해 사용한다.

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

class Point {
    double x;
    double y;
public:
    void set_x(double _x) {
        this->x = _x; // x = _x; 이렇게 대체 가능
    }
    void set_y(double _y) {
        this->y = _y;
    }
    void address() {
        cout << this << endl;
    }
};

int main() {
    Point pt1;
    pt1.set_x(100);
    pt1.set_y(200);
    Point* pt2 = &pt1;
    cout << pt2 << endl;
    pt2->address(); // -> 연산자를 사용한 멤버함수 사용
}
```

```
선택 Microsoft Visual
008FFBB4
008FFBB4

C:₩Users₩bluej₩Desk
ug₩Project1.exe(252
이 창을 닫으려면 0
```

D.Com

오버로딩

- C++은 하나의 함수(이름이 같은)에 대해 여러 정의를 가지도록 허용하고 있다. 이를 이용하여 이름은 같지만 입력 파라미터에 따라 여러 역할을 하는 함수나 연산자(operator)를 만들 수 있다. (반드시 입력 파라미터는 달라야 한다. - 무슨 함수인지 구분을 위해)
- 즉 함수 오버로딩이란, 서로 다른 여러 개의 함수가 하나의 이름을 공유(연결)하는 것이다. (이름이 반드시 같아야 한다.)
- 함수 오버로딩은 서로 다른 변수 타입을 대상으로 하지만 기본적으로는 같은 작업을 하는 함수만을 만드는 것이 좋다.
- 연산자 ('+', '<<', '=' 등등)에 대해서도 오버로딩이 가능하다!

D.Com

오버로딩 (함수 예시)

- 기존의 표준 라이브러리의 함수도 오버로딩이 물론 가능하다
- 파라미터는 다르지만 같은 기능을 하는 함수 오버로딩의 예시이다.

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

void print(string a) { // 파라미터가 다름
    for (auto elem : a) {
        cout << elem << ' ';
    }
    cout << '₩n';
}

void print(int a) { // 파라미터가 다름
    string b = to_string(a); // a를 스트링으로
    for (auto elem : b) {
        cout << elem << ' ';
    }
    cout << '₩n';
}

int main() {
    print("dasddsa");
    print(1211231);
}
```

```
Microsoft Visual Stud
d a s d d s a
1 2 1 1 2 3 1

C:₩Users₩bluej₩Des
 종료되었습니다.
이 창을 닫으려면 0
```

D.Com

오버로딩 (연산자 오버로딩)

- Point 클래스를 떠올려 보자.

```
Point p1(3, 4);
Point p2(4, 5);
Point p3 = p1 + p2;
```

- 위의 예시와 같이 객체와 객체의 덧셈 연산을 정의하려면 연산자 오버로딩이 필요하다.
- 1 + 2를 예로 들자면 + 연산자는 숫자 1과 2를 인자로 받는 함수처럼 볼 수 있다. 그러므로 오버로딩을 통해 다른 인자를 받도록 할 수 있다.

오버로딩 (연산자 오버로딩)

- 이러한 기능을 하는 연산자를 오버로딩 하고 싶다.

$$(x_1, y_1) + (x_2, y_2) = (x_1 + y_1, x_2 + y_2)$$

- 연산자 함수를 선언하는 형식은 대략적으로는 다음과 같다. (꼭 따를 필요 x 예시임)

연산자	함수 선언	리턴 값
+,-,/,*	클래스명 operator연산자(const 클래스명& 변수명)	객체
=	클래스명& operator=(const 클래스명& 변수명)	객체
〉,〈,〉=,〈=,==,!=	bool operator연산자(const 클래스명& 변수명)	bool

오버로딩 (연산자 오버로딩)

- 이런 식으로 하면 된다.
- 제일 우측은 포인터 내부의 연산자로 만들어 좌측의 객체는 this를 통해 접근하고 있다.
- 포인터 내부 연산자가 되면 1개의 객체만을 인자로 받을 수 있다.

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

class Point
{
public:
    Point(int _x, int _y) :x(_x), y(_y) {};
    Point() {};
    int x;
    int y;
};

Point operator+(const Point& p1, const Point& p2) {
    Point result;
    result.x = p1.x + p2.x;
    result.y = p1.y + p2.y;
    return result;
}

int main() {
    Point p1(1, 2);
    Point p2(2, 3);
    Point p3 = p1 + p2;
    cout << p3.x << ' ' << p3.y << endl;
}
```

```
Microsoft Visual S
3 5

C:₩Users₩bluej₩
종료되었습니다.
이 창을 닫으려면
```

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

class Point
{
public:
    Point(int _x, int _y) :x(_x), y(_y) {};
    Point() {};
    int x;
    int y;
    void operator+(const Point& p1) {
        this->x = this->x + p1.x;
        this->y = this->y + p1.y;
        // 기존의 것에 더해주기만 하니 리턴 필요x
    }
};

int main() {
    Point p1(1, 2);
    Point p2(2, 3);
    p1 + p2;
    cout << p1.x << ' ' << p1.y << endl;
}
```

```
선택 Micr
3 5

C:₩Users₩b
종료되었
이 창을 닫
```

D.Com

오버로딩 (연산자 오버로딩 예시)

```cpp
#include <iostream>
using namespace std;
class Point {
    double x;
    double y;
public:
    Point();
    Point(int x, int y);
    void setPoint(int x, int y);
    int getX(void) const;
    int getY(void) const;
    Point operator+(const Point& point);
    Point& operator=(const Point& point);
};
Point::Point()
{
    x = y = 0;
}
Point::Point(int x, int y)
{
    this->x = x;
    this->y = y;
}
void Point::setPoint(int x, int y)
{
    this->x = x;
    this->y = y;
}
int Point::getX(void) const
{
    return this->x;
}
int Point::getY(void) const
{
    return this->y;
}
Point Point::operator+(const Point& point)
{
    Point result(this->x + point.getX(), this->y + point.getY());
    return result;
}
Point& Point::operator=(const Point& point)
{
    this->x = point.getX();
    this->y = point.getY();
    return *this;
}
ostream& operator<<(ostream& cout, const Point& point)
{
    return cout << "(" << point.getX() << "," << point.getY() << ")";
} // << 오퍼레이터에 대한 오버로딩은 외우는 게 맘편하다.

int main()
{
    Point* pP1, * pP2;
    pP1 = new Point;
    pP2 = new Point(1, 2);
    pP1->setPoint(10, 20);
    *pP2 = *pP1 + *pP2;
    cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
    cout << *pP2 << endl;
    delete pP1;
    delete pP2;
}
```

ostream? :
ostream은 출력 stream으로 이를 상속받은 클래스로는 iostream, ofstream이 있다.
이러한 연산자 오버로딩을 통해 cout, fout객체로 값을 넘겨줄 수 있다.

D.Com

static 멤버 데이터와 함수

- 때로는 한 클래스에서 생겨난 모든 객체에서 동일한 변수가 필요할 수 있다. 전역 변수나 상수는 특정 클래스에 종속되어 있지 않기에 static 멤버 변수를 사용한다.
- 예를 들어 Point 클래스가 만들어진 수를 세고 싶을 때 static 변수를 사용한다.

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

class Point {
    int x;
    int y;
    static int count_num;

public:
    Point(int _x, int _y) :x(_x), y(_y) { count_num++;}; //이용
    Point() { x = 0, y = 0, count_num++; }; //이용
    static int get_count() { return count_num; }
    //static 멤버함수를 통해 static 멤버변수에 접근

};

int Point::count_num = 0;
//static 멤버 변수는 전역공간에서 초기화

int main() {
    Point pt1;
    cout << pt1.get_count() << endl;
    Point pt2;
    cout << pt2.get_count() << endl;
}
```

```
Microsoft Visual Stu
1
2

C:₩Users₩bluej₩De
 종료되었습니다.
이 창을 닫으려면
```

D.Com

Friend (함수 예시)

- friend는 C++에서 지원하는 예외적인 기능의 키워드이다. 외부에서 접근 제어 지시자를 무시하고 class의 private 정보들에 접근할 수 있게 한다.
- 연산자 오버로딩, 특수 목적의 함수에 사용된다.
- friend 키워드는 특이하게도 당하는 쪽(class)에서 선언해준다.

```
#include <iostream>
using namespace std;
class Point {
    double x;
    double y;
    static int countCreatedObjects;
public:
    Point();
    Point(int x, int y);
    void setPoint(int x, int y);
    int getX(void) const;
    int getY(void) const;
    static int getCreatedObject(void);
    Point operator+(const Point& point);
    Point& operator=(const Point& point);
    friend std::ostream&
        operator<<(std::ostream& os, const Point& point);
    //이 함수를 프렌드 함수로 선언
};
// Initialize the static variables.
int Point::countCreatedObjects = 0;
Point::Point()
{
    x = y = 0;
    countCreatedObjects++;
}
Point::Point(int x, int y)
{
    this->x = x;
    this->y = y;
    countCreatedObjects++;
}
```

D.Com

Friend (함수 예시)

```
33  void Point::setPoint(int x, int y)
34  {
35      this->x = x;
36      this->y = y;
37  }
38  int Point::getX(void) const
39  {
40      return this->x;
41  }
42  int Point::getY(void) const
43  {
44      return this->y;
45  }
46  int Point::getCreatedObject(void)
47  {
48      return countCreatedObjects;
49  }
50  Point Point::operator+(const Point& point)
51  {
52      Point result(this->x + point.getX(),
53          this->y + point.getY());
54      return result;
55  }
56  Point& Point::operator=(const Point& point)
57  {
58      this->x = point.getX();
59      this->y = point.getY();
60      return *this;
61  }
```

```
62  ostream& operator<<(ostream& os, const Point& point)
63  {
64      return os << "(" << point.x << "," << point.y << ")";
65  } // 외부 함수에서 클래스 내부 private 변수 접근 가능
66  int main()
67  {
68      Point* pP1, * pP2;
69      cout << "Number of created object is : "
70          << Point::getCreatedObject() << endl;
71      pP1 = new Point;
72      pP2 = new Point(1, 2);
73      pP1->setPoint(10, 20);
74      *pP2 = *pP1 + *pP2;
75      cout << "[X:" << pP1->getX()
76          << "]" << "[Y:" << pP1->getY() << "]" << endl;
77      cout << *pP2 << endl;
78      cout << "Number of created object is : "
79          << Point::getCreatedObject() << endl;
80      delete pP1;
81      delete pP2;
82  }
```

D.Com

Friend (class 예시)

- class도 friend로 선언이 가능하다.

```cpp
#include <iostream>
using namespace std;
class Point {
    double x;
    double y;
    static int countCreatedObjects;
public:
    Point();
    Point(int x, int y);
    void setPoint(int x, int y);
    int getX(void) const;
    int getY(void) const;
    static int getCreatedObject(void);
    Point operator+(const Point& point);
    Point& operator=(const Point& point);
    friend std::ostream& operator<<
        (std::ostream& os, const Point& point);
    friend class SpyPoint; //spy point 클래스 friend 선언
};
int Point::countCreatedObjects = 0;
// Initialize the static variables.
Point::Point()
{
    x = y = 0;
    countCreatedObjects++;
}
Point::Point(int x, int y)
{
    this->x = x;
    this->y = y;
    countCreatedObjects++;
}
void Point::setPoint(int x, int y)
{
    this->x = x;
    this->y = y;
}
int Point::getX(void) const
{
    return this->x;
}
int Point::getY(void) const
{
    return this->y;
}
int Point::getCreatedObject(void)
{
    return countCreatedObjects;
}
Point Point::operator+(const Point& point)
{
    Point result(this->x + point.getX(),
        this->y + point.getY());
    return result;
}
Point& Point::operator=(const Point& point)
{
    this->x = point.getX();
    this->y = point.getY();
    return *this;
}
```

D.Com

Friend (class 예시)

```
62  std::ostream& operator<<
63  (std::ostream& os, const Point& point)
64  {
65      return os << "(" << point.x << "," << point.y << ")";
66  }
67  class SpyPoint {
68  public:
69      void printPoint(const Point& point);
70  };
71  void SpyPoint::printPoint(const Point& point)
72  { // class의 private변수에 접근가능
73      cout << "{X:" << point.x << "}" <<
74          "{Y:" << point.y << "}" << endl;
75  }
76  int main()
77  {
78      Point* pP1, * pP2;
79      SpyPoint SP;
80      cout << "Number of created object is : "
81          << Point::getCreatedObject() << endl;
82      pP1 = new Point;
83      pP2 = new Point(1, 2);
84      pP1->setPoint(10, 20);
85      *pP2 = *pP1 + *pP2;
86      cout << "[X:" << pP1->getX() << "]" << "[Y:"
87          << pP1->getY() << "]" << endl;
88      cout << *pP2 << endl;
89      cout << "Number of created object is : "
90          << Point::getCreatedObject() << endl;
91      SP.printPoint(*pP1);
92      SP.printPoint(*pP2);
93      delete pP1;
94      delete pP2;
95  }
```

D.Com

소멸자

- 소멸자는 생성자의 반대이다. 생성자가 클래스명() {}; 의 형태를 가진다면 소멸자는 ~클래스명() {} 의 형태를 가진다.
- 말 그대로 생성자가 class가 생성될 때 기능을 수행하는 함수라면 소멸자는 객체가 소멸될 때 기능을 수행하는 함수이다.
- 예를 들어 생성자에서 동적할당을 new 키워드를 이용해 하고 동적할당 해제(delete)를 소멸자에서 수행해 줄 수 있다.

소멸자 (예시)

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

class Myclass {
private:
    int num;
    static int now_class;

public:
    Myclass(int _num) {
        num = _num;
        now_class++;
    }
    ~Myclass() {
        now_class--;
    } // 소멸자
    static int get_now_class() { return now_class; }
};

int Myclass::now_class = 0;

int main() {
    Myclass m1(1);
    cout << m1.get_now_class() << endl;
    Myclass m2(2);
    cout << m2.get_now_class() << endl;
    Myclass* m3= new Myclass(3);
    cout << m3->get_now_class() << endl;
    delete(m3);
    cout << m2.get_now_class() << endl;
}
```

```
Microsoft Visual Studio 디
1
2
3
2

C:₩Users₩bluej₩Desktop
 종료되었습니다.
이 창을 닫으려면 아무
```

D.Com

과제 7

- 랩 7 ㄱ