10. 연산자 오버로딩1

11주차

연산자 오버로딩1
(FirstOperationOverloading.cpp)

402-404p

```
#include <iostream>
using namespace std;
class Point  {
        int xpos, ypos;
public:
        Point(int x=0, int y=0) : xpos(x), ypos(y) {  }
        void ShowPosition() const {
                cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
        }
        Point operator+(const Point &ref) {
        {                       Point pos(xpos+ref.xpos, ypos+ref.ypos);
                return pos;
        }
};
int main(void) {
        Point pos1(3, 4); Point pos2(10, 20);
        Point pos3=pos1.operator+(pos2);
        pos1.ShowPosition(); pos2.ShowPosition(); pos3.ShowPosition();
        return 0;
}
```

'operator' 키워드와 '연산자'를 묶어서 함수이름으로 정의하면 해당 연산자를 이용한 호출도 허용함.

Point pos3 = pos1 + pos2;
와 동일!.

404-410p

연산자를 overloading하는 2가지 방법
(GFunctionOverloading.cpp)

```
#include <iostream>
using namespace std;
class Point {
private:
        int xpos, ypos;
public:
        Point(int x = 0, int y = 0) : xpos(x), ypos(y) {  }
        void ShowPosition() const {
                cout << '[' << xpos << ", " << ypos << ']' << endl;
        }
        friend Point operator+(const Point& pos1, const Point& pos2);
};
Point operator+(const Point& pos1, const Point& pos2) {
        Point pos(pos1.xpos + pos2.xpos, pos1.ypos + pos2.ypos);
        return pos;
}
int main(void) {
        Point pos1(3, 4);  Point pos2(10, 20);
        Point pos3 = pos1 + pos2;
        pos1.ShowPosition(); pos2.ShowPosition(); pos3.ShowPosition();
        return 0;
}
```

연산자 overloading 함수의 인자는 const 참조자로 하자! -> 복사생성자 거치지 않고, 값 수정 방지.

전역함수에서 private 멤버에 접근할 수 있게 함.

전역함수로 연산자 overloading

410-411p

연산자 오버로딩의 제약 사항

.	멤버 접근 연산자
.*	멤버 포인터 연산자
::	범위 지정 연산자
?:	조건 연산자(3항 연산자)
sizeof	바이트 단위 크기 계산
typeid	RTTI 관련 연산자
static_cast	형변환 연산자
dynamic_cast	형변환 연산자
const_cast	형변환 연산자
reinterpret_cast	형변환 연산자

오버로딩 불가능!

=	대입 연산자
()	함수 호출 연산자
[]	배열 접근 연산자(인덱스 연산자)
->	멤버 접근을 위한 포인터 연산자

멤버함수의 형태로만 오버로딩 가능! (객체 대상으로 진행해야 의미 있기 때문)

411-412p

연산자 오버로딩의 주의 사항

- 본래의 의도를 벗어난 형태의 연산자 오버로딩은 만들지 말자!
- 연산자의 우선순위와 결합성은 바뀌지 않는다.
- 매개변수의 디폴트 값 설정 불가.
  - 매개변수의 자료형에 따라서 호출되는 함수가 결정되므로.
- 기본형의 연산자는 overload 할 수 없음.

422-424p

전위 단항연산자와 후위 단항연산자

- C언어의 연산 특성을 그대로 반영하여 함수 작성하기
  - C언어에서 아래는 허용함
    ++(++num)
    --(--num)
  - C언어에서 아래는 허용안함
    (num++)++
    (num--)--

- 함수 overload시 전위 단항연산자와 후위 단항연산자 구분 방법
  - 전위는 단항임을 고려하여 연산자 개수가 한 개 적음.
  - 후위는 오른쪽 연산자 자리에 (사용하지 않는) int 타입 인자를 선언.

412-418p

전위 단항 연산자의 오버로딩
(OneOpndOverloading.cpp 1/2)

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

class Point
{
private:
	int xpos, ypos;
public:
	Point(int x = 0, int y = 0) : xpos(x), ypos(y)
	{ }
	void ShowPosition() const
	{
		cout << '[' << xpos << ", " << ypos << ']' << endl;
	}
	Point& operator++()
	{
		xpos += 1;
		ypos += 1;
		return *this;
	}
	friend Point& operator--(Point& ref);
};
```

- 매개변수 개수가 1개 적다

- 복사생성자 거치지 않고 연속적인 연산 가능하게 참조 형태로 this 를 반환.

412-418p

전위 단항 연산자의 오버로딩
(OneOpndOverloading.cpp 2/2)

```
Point& operator--(Point& ref)
{
	ref.xpos -= 1;
	ref.ypos -= 1;
	return ref;
}

int main(void)
{
	Point pos(1, 2);
	++pos;
	pos.ShowPosition();
	--pos;
	pos.ShowPosition();

	++(++pos);
	pos.ShowPosition();
	--(--pos);
	pos.ShowPosition();
	return 0;
}
```

- 전역변수로써 단항연산자 overloading

연이어 호출 가능.

후위 단항 연산자의 오버로딩
(PostOpndOverloading.cpp 1/2)

```
#include <iostream>
using namespace std;
class Point {
private:
        int xpos, ypos;
public:
        Point(int x = 0, int y = 0) : xpos(x), ypos(y) {  }
        void ShowPosition() const {
                cout << '[' << xpos << ", " << ypos << ']' << endl;
        }
        Point& operator++() {
                xpos += 1;
                ypos += 1;
                return *this;
        }
        const Point operator++(int) {
                const Point retobj(xpos, ypos);    // const Point retobj(*this);
                xpos += 1;
                ypos += 1;
                return retobj;
        }
        friend Point& operator--(Point& ref);
        friend const Point operator--(Point& ref, int);
};
```

멤버 함수로 overload한
전위 증가와 후위증가 연산자.

값 증가에 앞서 반환할 const
객체 만들어 뒀다가 반환.

전역함수로써 overload한
전위 증가와 후위증가 연산자.

후위 단항 연산자의 오버로딩
(PostOpndOverloading.cpp 1/2)

```
Point& operator--(Point& ref) {
        ref.xpos -= 1;
        ref.ypos -= 1;
        return ref;
}
const Point operator--(Point& ref, int) {
        const Point retobj(ref);
        ref.xpos -= 1;
        ref.ypos -= 1;
        return retobj;
}
int main(void) {
        Point pos(3, 5);
        Point cpy;
        cpy = pos--;
        cpy.ShowPosition();
        pos.ShowPosition();

        cpy = pos++;
        cpy.ShowPosition();
        pos.ShowPosition();
        return 0;
}
```

전역 함수로 overload한
전위 감소와 후위감소 연산자.

값 증가에 앞서 반환할 const
객체 만들어 뒀다가 반환.

교환법칙의 성립을 위한 구현 (CommuMultipleOperation.cpp)

425-428p

```
#include <iostream>
using namespace std;
class Point {
        int xpos, ypos;
public:
        Point(int x = 0, int y = 0) : xpos(x), ypos(y) {  }
        void ShowPosition() const {
                cout << '[' << xpos << ", " << ypos << ']' << endl;
        }
        Point operator*(int times) {
                Point pos(xpos * times, ypos * times);
                return pos;
        }
        friend Point operator*(int times, Point& ref);
};
Point operator*(int times, Point& ref) { return ref * times; }
int main(void) {
        Point pos(1, 2), cpy;
        cpy = 3 * pos;
        cpy.ShowPosition();
        cpy = 2 * pos * 3;
        cpy.ShowPosition();
        return 0;
}
```

Pre-defined 타입이 왼쪽 인자로 오려면 전역함수 사용 필수!
구현은 클래스 타입이 왼쪽에 오는 함수를 호출해도 됨. (교환법칙)

교환법칙 지원으로 자연스러운 구문이 가능해 짐.

429-433p

cout과 endl 흉내내기
(IterateConsoleOutput.cpp 1/2)

```
#include <iostream>

namespace mystd {
        using namespace std;

        class ostream {
        public:
                ostream& operator<< (char* str) {
                        printf("%s", str);
                        return *this;
                }
                ostream& operator<< (char str) {
                        printf("%c", str);
                        return *this;
                }
                ostream& operator<< (int num) {
                        printf("%d", num);
                        return *this;
                }
                ......
```

cout과 endl 흉내내기
(IterateConsoleOutput.cpp 2/2)

429-433p

```
                .....
                ostream& operator<< (double e) {
                        printf("%g", e);
                        return *this;
                }
                ostream& operator<< ( ostream& (*fp)(ostream& ostm) ) {
                        return fp(*this);
                }
        };
        ostream& endl(ostream& ostm) {
                ostm << '₩n';
                fflush(stdout);
                return ostm;
        }
        ostream cout;
}
int main(void) {
        using mystd::cout;
        using mystd::endl;
        cout << 3.14 << endl << 123 << endl;
        endl(cout);
        return 0;
}
```

cout객체 안에서 다양한 기본 자료형을 대상으로 << 연산자를 overloading 하였다.

연속적으로 사용하기 위해 반환 타입을 ostream & 로 했음.

std 영역에서도 endl은 함수다.

<<, >> 연산자 overloading
(PointConsoleOutput.cpp 변형 1/2)

434-437p

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

class Point
{
private:
	int xpos, ypos;
public:
	Point(int x = 0, int y = 0) : xpos(x), ypos(y)
	{ }
	void ShowPosition() const
	{
		cout << '[' << xpos << ", " << ypos << ']' << endl;
	}
	friend ostream& operator<<(ostream&, const Point&);
	friend istream& operator>>(istream&, Point&);
};
```

ostram이나 istream의 >>, << 연산자를 추가할 수 없으므로 전역함수에 의한 overloading만 가능!

434-437p

<<, >> 연산자 overloading
(PointConsoleOutput.cpp 변형 2/2)

```
ostream& operator<<(ostream& os, const Point& pos) {
	os << '[' << pos.xpos << ", " << pos.ypos << ']' << endl;
	return os;
}

istream& operator>>(istream& is, Point& pos) {
	is >> pos.xpos >> pos.ypos;
	return is;
}

int main(void) {
	Point pos1(1, 3);
	cout << pos1;
	Point pos2(101, 303);
	cout << pos2;

	cout << "Point 입력: ";
	cin >> pos1;
	cout << "입력 내용은 " << pos1;
	return 0;
}
```

11. 연산자 오버로딩2

12주차

default 대입연산자와 그의 문제점
(AssignShallowCopyError.cpp 1/2)

442-449p

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class Person {
        char* name;
        int age;
public:
        Person(const char* myname, int myage) {
                int len = strlen(myname) + 1;
                name = new char[len];
                strcpy(name, myname);
                age = myage;
        }
        Person& operator=(const Person& ref) {
                delete [] name;
                int len=strlen(ref.name)+1;
                name= new char[len];
                strcpy(name, ref.name);
                age=ref.age;
                return *this;
        }
```

대입연산자를 정의하지 않으면 default 대입연산자가 얕은 복사 함 -> 할당 받은 메모리 주소 복사 시 문제 발생.

복사생성자와 유일하게 다른 점 : 기존의 메모리를 해제하는 작업 필요.

깊은 복사.

default 대입연산자와 그의 문제점
(AssignShallowCopyError.cpp 2/2)

442-449p

```
        ......
        void ShowPersonInfo() const {
                cout << "이름: " << name << endl;
                cout << "나이: " << age << endl;
        }
        ~Person() {
                delete [] name;
                cout << "called destructor!" << endl;
        }
};

int main(void) {
        Person man1("Lee dong woo", 29);
        Person man2("Yoon ji yul", 22);
        man2 = man1;
        man1.ShowPersonInfo();
        man2.ShowPersonInfo();
        return 0;
}
```

Default 대입연산자 사용될 경우, 얕은 복사 때문에 여기서 문제 생김.

대입연산자 사용.

449-453p

상속 구조에서의 대입연산자
(InheritAssignOperation.cpp 1/3)

- **생성자**
  - 유도클래스의 생성자에서 명시하지 않아도 기초클래스의 생성자가 호출됨.
- **대입연산자**
  - 유도클래스의 대입연산자에서 명시하지 않으면 기초클래스의 대입연산자가 호출되지 않음.
  - => **대입연산자 구현 시 반드시 기초클래스의 대입연산자를 호출해 주기!**

```
#include <iostream>
using namespace std;
class First {
        int num1, num2;
public:
        First(int n1 = 0, int n2 = 0) : num1(n1), num2(n2) {  }
        void ShowData() { cout << num1 << ", " << num2 << endl; }
        First& operator=(const First& ref)  {
                cout << "First& operator=()" << endl;
                num1 = ref.num1;
                num2 = ref.num2;
                return *this;
        }
};
```

449-453p

상속 구조에서의 대입연산자
(InheritAssignOperation.cpp 2/3)

```
class Second : public First {
private:
        int num3, num4;
public:
        Second(int n1, int n2, int n3, int n4)
                : First(n1, n2), num3(n3), num4(n4) {  }
        void ShowData() {
                First::ShowData();
                cout << num3 << ", " << num4 << endl;
        }

        Second& operator=(const Second &ref) {
                cout<<"Second& operator=()"<<endl;
                First::operator=(ref);
                num3=ref.num3;
                num4=ref.num4;
                return *this;
        }
};
```

이 함수를 가리면:
Default 대입연사자가 기초클래스 대입연산자까지 호출하여 4개의 멤버변수가 모두 복사됨.

함수 정의하고 이 문장만 가리면:
기초클래스의 대입연산자를 자동으로 호출하지 않으므로 기초클래스가 가진 2개 멤버변수 값이 전달 안됨.

449-453p

상속 구조에서의 대입연산자
(InheritAssignOperation.cpp 3/3)

```
int main(void)
{
	Second ssrc(111, 222, 333, 444);
	Second scpy(0, 0, 0, 0);
	scpy = ssrc;
	scpy.ShowData();
	return 0;
}
```

453-455p

이니셜라이저의 성능 향상 도움
(ImproveInit.cpp 1/2)

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

class AAA {
private:
        int num;
public:
        AAA(int n = 0) : num(n) {
                cout << "AAA(int n=0)" << endl;
        }
        AAA(const AAA& ref) : num(ref.num) {
                cout << "AAA(const AAA & ref)" << endl;
        }
        AAA& operator=(const AAA& ref) {
                num = ref.num;
                cout << "operator=(const AAA &ref)" << endl;
                return *this;
        }
};
```

[결과]

```
AAA(int n=0)
**********
AAA(const AAA & ref)
**********
AAA(int n=0)
operator=(const AAA &ref)
```

이니셜라이저의 성능 향상 도움
(ImproveInit.cpp 2/2)

453-455p

```
class BBB {
        AAA mem;
public:
        BBB(const AAA& ref)
                : mem(ref) {  }
};
class CCC {
        AAA mem;
public:
        CCC(const AAA& ref) {
                mem = ref;
        }
};
int main(void) {
        AAA obj1(12);
        cout << "**********" << endl;
        BBB obj2(obj1);
        cout << "**********" << endl;
        CCC obj3(obj1);
        return 0;
}
```

복사생성자

Default constructor와 대입연산자.

[결과]

```
AAA(int n=0)
**********
AAA(const AAA & ref)
**********
AAA(int n=0)
operator=(const AAA &ref)
```

BBB 클래스는 initializer이용했으므로 AAA클래스의 생성자만으로 객체 생성이 완료됨.

CCC 클래스는 initializer이용 안하고 생성 후 대입했으므로 AAA클래스의 생성자와 대입연산자, 2개 함수가 호출됨.

456-459p

배열보다 나은 배열 클래스
(ArrayClass.cpp 1/2)

- 배열 역할 하는 클래스 구현 => 배열 원소에 접근 시 index값의 경계검사 가능.
- [] 연산자 overloading => 실제 배열처럼 사용 가능.

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

class BoundCheckIntArray {
private:
        int* arr;
        int arrlen;
public:
        BoundCheckIntArray(int len);
        int& operator[] (int idx);
        ~BoundCheckIntArray();
};
```

변경도 가능하도록 const 안붙였음.

456-459p

배열보다 나은 배열 클래스
(ArrayClass.cpp 2/2)

```
int main(void) {
        BoundCheckIntArray arr(5);
        for (int i = 0; i < 5; i++)
                arr[i] = (i + 1) * 11;
        for (int i = 0; i < 6; i++)
                cout << arr[i] << endl;
        return 0;
}
BoundCheckIntArray::BoundCheckIntArray(int len) :arrlen(len) {
        arr = new int[len];
}
int& BoundCheckIntArray::operator[] (int idx) {
        if (idx < 0 || idx >= arrlen) {
                cout << "Array index out of bound exception" << endl;
                exit(1);
        }
        return arr[idx];
}
BoundCheckIntArray::~BoundCheckIntArray() {
        delete [] arr;
}
```

일반 배열과 사용법이 같음.

[] 연산자 overloading 하면서 유효한 범위 체크.