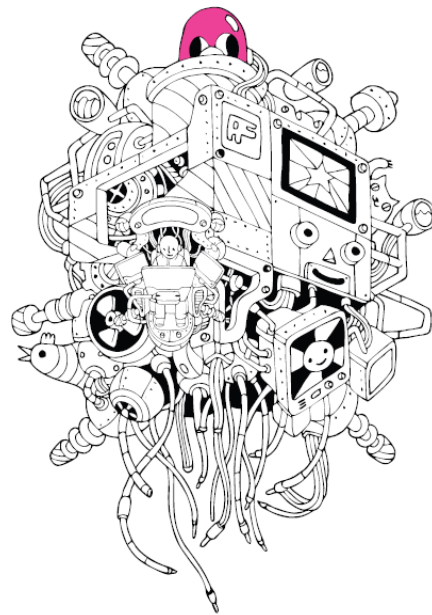


열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 11. 연산자 오버로딩2

열혈 C++ 프로그래밍



Chapter 11-1. 반드시 해야 하는 대입 연산자의
오버로딩

윤성우 저 열혈강의 C++ 프로그래밍 개정판

객체간 대입연산의 비밀: 디폴트 대입 연산자

복사 생성자의 호출

```
int main(void)
{
    Point pos1(5, 7);
    Point pos2=pos1;
    . . . .
}
```

대입 연산자의 호출

```
int main(void)
{
    Point pos1(5, 7);
    Point pos2(9, 10);
    pos2=pos1;
    . . . . pos2.operator=(pos1);
}
```

```
class First
{
private:
    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)
{
    num1=ref.num1;
    num2=ref.num2;
    return *this;
}
```

멤버 대 멤버의 복사를 진행하는 디폴트 대입
연산자 삽입!

First 클래스의 디폴트 대입 연산자

디폴트 대입 연산자의 문제점

해결책이 되는 대입 연산자의 오버라이딩

```
class Person
{
private:
    char * name;
    int age;
public:
    Person(char * myname, int myage)
    {
        int len=strlen(myname)+1;
        name=new char[len];
        strcpy(name, myname);
        age=myage;
    }
    void ShowPersonInfo() const
    {
        cout<<"이름: "<<name<<endl;
        cout<<"나이: "<<age<<endl;
    }
    ~Person()
    {
        delete []name;
        cout<<"called destructor!"<<endl;
    }
};
```

```
Person& operator=(const Person& ref)
{
    delete []name;    // 메모리의 누수를 막기 위한 메모리 해제 연산
    int len=strlen(ref.name)+1;
    name= new char[len];
    strcpy(name, ref.name);
    age=ref.age;
    return *this;
}
```

이전에 공부한 디폴트 복사 생성자의 문제점과 동일한 문제점을 확인할 수 있다!

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

실행결과

```
이름: Lee dong woo
나이: 29
이름: Lee dong woo
나이: 29
called destructor!
```

상속 구조에서의 대입 연산자 호출

```
class First
{
private:
    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;
    }
};
```

```
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;
        num3=ref.num3;
        num4=ref.num4;
        return *this;
    }
    */
};
```

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

디폴트 대입 연산자는 기초 클래스의 대입연산자를 호출해준다. 그러나 명시적으로 대입 연산자를 정의하게 되면, 기초 클래스의 대입 연산자 호출도 오른쪽의 예와 같이 별도로 명시해야 한다.

이니셜라이저의 성능 향상 도움

```
class BBB
{
private:
    AAA mem;
public:
    BBB(const AAA& ref) : mem(ref) { }
};

class CCC
{
private:
    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;
}
```

이니셜라이저를 이용해서 멤버를 초기화하면, 함수호출의 수를 1회 줄일 수 있다!

실행결과

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

열혈 C++ 프로그래밍



Chapter 11-2. 배열의 인덱스 연산자 오버로딩

윤성우 저 열혈강의 C++ 프로그래밍 개정판

배열보다 나은 배열 클래스

```
int main(void)
{
    int arr[3]={1, 2, 3};
    cout<<arr[-1]<<endl;
    cout<<arr[-2]<<endl;
    cout<<arr[3]<<endl;
    cout<<arr[4]<<endl;
    . . . .
}
```

이렇듯 기본 배열은 접근에 대한 경계검사를 진행하지 않는다.

배열 클래스를 기반으로 생성되는 배열 객체는 배열과 동일한 기능을 하되, 경계검사의 기능을 추가한 객체이다.
연산자 오버로딩을 통해서 다음과 같이 배열처럼 접근이 가능한 객체이다.

```
arrObject[2];
```

이는 다음과 같이 해석이 된다.

```
arrObject.operator[ ] (2);
```

때문에 다음의 형태로 오버로딩 해야 한다.

```
int operator [ ] (int idx) { . . . }
```



배열 클래스의 예

```
class BoundCheckIntArray
{
private:
    int * arr;
    int arrlen;
public:
    BoundCheckIntArray(int len) :arrlen(len)
    {
        arr=new int[len];
    }
    int& operator[] (int idx)
    {
        if(idx<0 || idx>=arrlen)
        {
            cout<<"Array index out of bound exception"<<endl;
            exit(1);
        }
        return arr[idx];
    }
    ~BoundCheckIntArray()
    {
        delete []arr;
    }
};
```

```
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;
}
```

실행결과

```
11
22
33
44
55
Array index out of bound exception
```

배열 클래스의 안전성 확보

배열은 저장소의 일종이고, 저장소에 저장된 데이터는 유일성이 보장되어야 하기 때문에 배열 객체를 대상으로 하는 복사와 관련된 연산은 모두 불가능하게 해야 할 필요도 있다(물론 상황에 따라서).

```
int main(void)
{
    BoundCheckIntArray arr(5);
    for(int i=0; i<5; i++)
        arr[i]=(i+1)*11;

    BoundCheckIntArray cpy1(5);
    cpy1=arr;      // 안전하지 않은 코드(이유는 이어서 바로 설명)
    BoundCheckIntArray copy=arr;    // 역시! 안전하지 않은 코드
    . . . . .
}
```

복사와 관련된 연산의 제한을 위해서 복사 생성자와 대입 연산자를 **private**으로 선언한 예!

```
class BoundCheckIntArray
{
private:
    int * arr;
    int arrlen;
    BoundCheckIntArray(const BoundCheckIntArray& arr) { }
    BoundCheckIntArray& operator=(const BoundCheckIntArray& arr) { }
public:
    . . . . .
}
```

const 함수를 이용한 오버로딩의 활용

함수의 **const** 유무는 함수 오버로딩의 조건이 된다!

```
int operator[] (int idx) const
{
    if(idx<0 || idx>=arrlen)
    {
        cout<<"Array index out of bound exception"<<endl;
        exit(1);
    }
    return arr[idx];
}
```

const 참조자로 참조하는 경우의 함수 호출을 위해서 정의된 함수! 그런데 이 함수를 대상으로는 멤버변수의 값을 변경할 수 없으니, **const**로 선언되지 않은 다음 함수가 추가로 정의되어야 한다.

```
int& operator[] (int idx)
{
    if(idx<0 || idx>=arrlen)
    {
        cout<<"Array index out of bound exception"<<endl;
        exit(1);
    }
    return arr[idx];
}
```

그래서 일반적으로 **operator[]** 함수는 **const** 함수와 일반함수가 동시에 정의된다.



객체의 저장을 위한 배열 클래스1

```
class BoundCheckPointArray
{
private:
    Point * arr;
    int arrlen;

    BoundCheckPointArray(const BoundCheckPointArray& arr) { }
    BoundCheckPointArray& operator=(const BoundCheckPointArray& arr) { }

public:
    BoundCheckPointArray(int len) :arrlen(len)
    {
        arr=new Point[len];
    }
    Point& operator[] (int idx)
    {
        if(idx<0 || idx>=arrlen)
        {
            cout<<"Array index out of bound exception"<<endl;
            exit(1);
        }
        return arr[idx];
    }
    Point operator[] (int idx) const
    {
        if(idx<0 || idx>=arrlen)
        {
            cout<<"Array index out of bound exception"<<endl;
            exit(1);
        }
        return arr[idx];
    }
    int GetArrLen() const { return arrlen; }
    ~BoundCheckPointArray() { delete []arr; }
};
```

```
int main(void)
{
    BoundCheckPointArray arr(3);
    arr[0]=Point(3, 4);
    arr[1]=Point(5, 6);
    arr[2]=Point(7, 8);

    for(int i=0; i<arr.GetArrLen(); i++)
        cout<<arr[i];

    return 0;
}
```

[3, 4]

[5, 6]

[7, 8]

실행결과

저장의 대상이 객체인 배열 클래스이다. 객체의 저장 방법은 두 가지이다. 객체를 통째로 저장하는 방법이 있고 객체의 주소 값을 저장하는 방법이 있다. 왼쪽의 클래스는 객체를 통째로 저장하는 배열 클래스이다.

객체의 저장을 위한 배열 클래스2

```
class BoundCheckPointPtrArray
{
private:
    POINT_PTR * arr;
    int arrlen;

    BoundCheckPointPtrArray(const BoundCheckPointPtrArray& arr) { }
    BoundCheckPointPtrArray& operator=(const BoundCheckPointPtrArray& arr) { }

public:
    BoundCheckPointPtrArray(int len) :arrlen(len)
    {
        arr=new POINT_PTR[len];
    }
    POINT_PTR& operator[] (int idx)
    {
        if(idx<0 || idx>=arrlen)
        {
            cout<<"Array index out of bound exception"<<endl;
            exit(1);
        }
        return arr[idx];
    }
    POINT_PTR operator[] (int idx) const
    {
        if(idx<0 || idx>=arrlen)
        {
            cout<<"Array index out of bound exception"<<endl;
            exit(1);
        }
        return arr[idx];
    }
    int GetArrLen() const { return arrlen; }
    ~BoundCheckPointPtrArray() { delete []arr; }
};
```

```
int main(void)
{
    BoundCheckPointPtrArray arr(3);
    arr[0]=new Point(3, 4);
    arr[1]=new Point(5, 6);
    arr[2]=new Point(7, 8);

    for(int i=0; i<arr.GetArrLen(); i++)
        cout<<*(arr[i]);

    delete arr[0];
    delete arr[1];
    delete arr[2];
    return 0;
}
```

[3, 4]

[5, 6]

[7, 8]

실행결과

객체의 주소 값을 저장하는 형태의 배열이다.
앞서 보인 객체를 통째로 저장하는 배열보다
일반적이다.

열혈 C++ 프로그래밍



Chapter 11-3. 그 이외의 연산자 오버로딩

윤성우 저 열혈강의 C++ 프로그래밍 개정판

new 연산자 오버로딩에 대한 상세한 이해

new 연산자가 하는 일

1. 메모리 공간의 할당
2. 생성자의 호출
3. 할당하고자 하는 자료형에 맞게 반환된 주소 값의 형 변환

이 중에서 1번에 해당하는 메모리 공간의 할당 작업만 오버로딩의 대상이 된다. 즉, 2번과 3번의 역할은 고정이다. 오버로딩이 불가능하다.

```
void * operator new (size_t size)
{
    void * adr=new char[size];
    return adr;
}
```

클래스의 멤버함수 형태로 오버로딩 된 new 연산자의 예



delete 연산자 오버로딩에 대한 이해와 예제

```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y) { }
    friend ostream& operator<<(ostream& os, const Point& pos);

    void * operator new (size_t size)
    {
        cout<<"operator new : "<<size<<endl;
        void * adr=new char[size];
        return adr;
    }
    void operator delete (void * adr)
    {
        cout<<"operator delete ()"<<endl;
        delete []adr;
    }
};
```

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

```
int main(void)
{
    Point * ptr=new Point(3, 4);
    cout<<*ptr;
    delete ptr;
    return 0;
}
```

```
operator new : 8
[3, 4]
operator delete ()
```

실행결과

```
void operator delete (void * adr)
{
    delete []adr;
}
```

객체가 생성된 상태가 아닌데도 불구하고 new 연산자를 오버로딩 하고 있는 멤버함수의 호출이 가능한 이유는 다음과 같다.

new 연산자와 delete 연산자를 오버로딩 하고 있는 함수는 자동 static으로 선언이 된다.

오버로딩 된 delete 연산자 내에서 반드시 해야 할 일

operator new & operator new []

두 가지 형태의 new 연산자 오버로딩

```
void * operator new (size_t size) { . . . . }  
void * operator new[] (size_t size) { . . . . }
```

두 가지 형태의 delete 연산자 오버로딩

```
void operator delete (void * adr) { . . . . }  
void operator delete[] (void * adr) { . . . . }
```

```
ostream& operator<<(ostream& os, const Point& pos)  
{  
    os<< '['<<pos.xpos<< ", "<<pos.ypos<< ']'<<endl;  
    return os;  
}  
int main(void)  
{  
    Point * ptr=new Point(3, 4);  
    Point * arr=new Point[3];  
    delete ptr;  
    delete []arr;  
    return 0;  
}
```

```
class Point  
{  
private:  
    int xpos, ypos;  
public:  
    Point(int x=0, int y=0) : xpos(x), ypos(y) { }  
    friend ostream& operator<<(ostream& os, const Point& pos);  
    void * operator new (size_t size)  
    {  
        cout<<"operator new : "<<size<<endl;  
        void * adr=new char[size];  
        return adr;  
    }  
    void * operator new[] (size_t size)  
    {  
        cout<<"operator new [] : "<<size<<endl;  
        void * adr=new char[size];  
        return adr;  
    }  
    void operator delete (void * adr)  
    {  
        cout<<"operator delete ()"<<endl;  
        delete []adr;  
    }  
    void operator delete[] (void * adr)  
    {  
        cout<<"operator delete[] ()"<<endl;  
        delete []adr;  
    }  
};
```

실행결과



```
operator new : 8  
operator new [] : 24  
operator delete ()  
operator delete[] ()
```

포인터 연산자 오버로딩

```
class Number
{
private:
    int num;
public:
    Number(int n) : num(n) { }
    void ShowData() { cout<<num<<endl; }
    Number * operator->()
    {
        return this;
    }
    Number& operator*()
    {
        return *this;
    }
};

int main(void)
{
    Number num(20);
    num.ShowData();

    (*num)=30;
    num->ShowData();
    (*num).ShowData();
    return 0;
}
```

`(*num)=30;`  `(num.operator*())=30;`
`(*num).ShowData();`  `(num.operator*()).ShowData();`

`num->ShowData();`  `num.operator->() -> ShowData();`

20
30
30

실행결과

스마트 포인터(Smart Pointer)

```
class SmartPtr
{
private:
    Point * posptr;
public:
    SmartPtr(Point * ptr) : posptr(ptr)
    { }

    Point& operator*() const
    {
        return *posptr;
    }
    Point* operator->() const
    {
        return posptr;
    }
    ~SmartPtr()
    {
        delete posptr;
    }
};
```

```
int main(void)
{
    SmartPtr sptr1(new Point(1, 2));
    SmartPtr sptr2(new Point(2, 3));
    SmartPtr sptr3(new Point(4, 5));
    cout<<*sptr1;
    cout<<*sptr2;
    cout<<*sptr3;

    sptr1->SetPos(10, 20);
    sptr2->SetPos(30, 40);
    sptr3->SetPos(50, 60);
    cout<<*sptr1;
    cout<<*sptr2;
    cout<<*sptr3;
    return 0;
}
```

```
Point 객체 생성
Point 객체 생성
Point 객체 생성
[1, 2]
[2, 3]
[4, 5]
[10, 20]
[30, 40]
[50, 60]
Point 객체 소멸
Point 객체 소멸
Point 객체 소멸
```

실행결과

포인터 처럼 동작하는, 포인터보다 다소 똑똑하게 동작하는 객체를 가리켜 스마트 포인터라 한다.
위의 스마트 포인터는 자신이 참조하는 객체의 소멸을 대신해주는 똑똑한 포인터이다.

() 연산자의 오버로딩과 펑터(Functor)

() 연산자의 오버로딩

- 객체를 함수처럼 사용할 수 있게 하는 오버로딩.
- 객체의 멤버함수를 함수처럼 호출할 수 있게 하는 오버로딩.

adder가 객체의 이름이라면

- adder(2, 4); 와 같이 함수처럼 사용을 한다.
- 그리고 이는 adder.operator()(2, 4); 로 해석이 된다.

```
class Adder
{
public:
    int operator()(const int& n1, const int& n2)
    {
        return n1+n2;
    }
    double operator()(const double& e1, const double& e2)
    {
        return e1+e2;
    }
    Point operator()(const Point& pos1, const Point& pos2)
    {
        return pos1+pos2;
    }
};
```

이렇듯 함수처럼 호출이 가능한 객체를 가리켜
'Functor'라 부른다.

```
4
5.2
[10, 13]
```

실행결과

```
int main(void)
{
    Adder adder;
    cout<<adder(1, 3)<<endl;
    cout<<adder(1.5, 3.7)<<endl;
    cout<<adder(Point(3, 4), Point(7, 9));
    return 0;
}
```

펑터(Functor)의 위력

```
class SortRule
{
public:
    virtual bool operator()(int num1, int num2) const =0;
};
```

```
class AscendingSort : public SortRule    // 오름차순
{
public:
    bool operator()(int num1, int num2) const
    {
        if(num1>num2)
            return true;
        else
            return false;
    }
};
```

```
class DescendingSort : public SortRule    // 내림차순
{
public:
    bool operator()(int num1, int num2) const
    {
        if(num1<num2)
            return true;
        else
            return false;
    }
};
```

```
void SortData(const SortRule& functor)
{
    for(int i=0; i<(idx-1); i++)
    {
        for(int j=0; j<(idx-1)-i; j++)
        {
            if(functor(arr[j], arr[j+1]))
            {
                int temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}
```

위의 함수는 본문의 **DataStorage** 클래스의 멤버함수이다. 이 함수가 호출이 되면 버블정렬이 되는데, 인자로 무엇이 전달되느냐에 따라서 오름차순 정렬이 될 수도 있고, 내림차순 정렬이 될 수도 있다.

임시객체로의 자동 형 변환

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

실행결과

```
Number(int n=0)
Number(int n=0)
operator=()
30
```

```
int main(void)
{
    Number num;
    num=30;
    num.ShowNumber();
    return 0;
}
```

num=Number(30); // 1단계. 임시객체의 생성

num.operator=(Number(30)); // 2단계. 임시객체를 대상으로 하는 대입 연산자의 호출

위 main 함수의 다음 문장 처리과정

num=30;

형 변환 연산자의 오버로딩

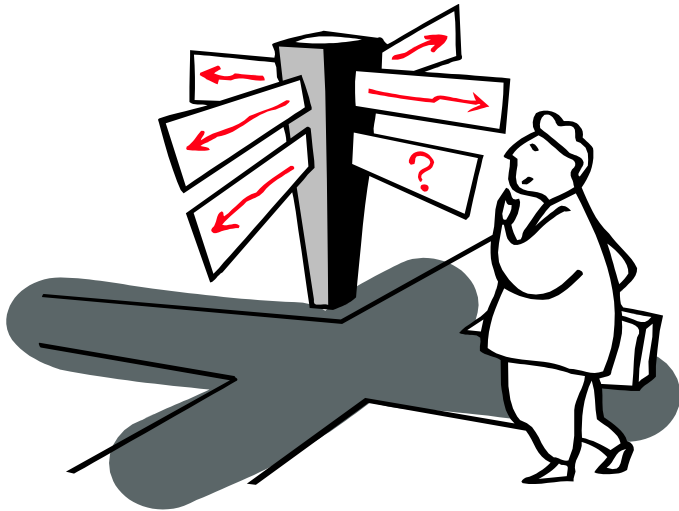
```
class Number
{
private:
    int num;
public:
    Number(int n=0) : num(n)
    {
        cout<<"Number(int n=0)"<<endl;
    }
    Number& operator=(const Number& ref)
    {
        cout<<"operator=()"<<endl;
        num=ref.num;
        return *this;
    }
    operator int ()    // 형 변환 연산자의 오버로딩
    {
        return num;    해당 객체가 int형으로 변환되어야 하는 상황에서 호출되는 함수
    }
    void ShowNumber() { cout<<num<<endl; }
};
```

```
int main(void)
{
    Number num1;
    num1=30;
    Number num2=num1+20;
    num2.ShowNumber();
    return 0;
}
```

```
Number(int n=0)
Number(int n=0)
operator=()
Number(int n=0)
50
```

실행결과





Chapter 11이 끝났습니다. 질문 있으신지요?