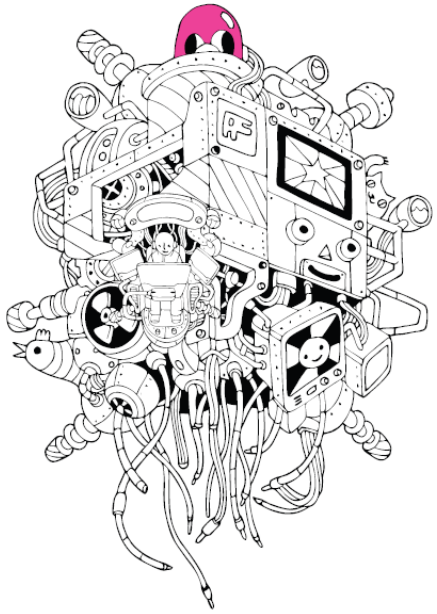
윤성우의
열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 11. 연산자 오버로딩2

윤성우의

열혈 C++ 프로그래밍

Chapter 11-3. 그 이외의 연산자 오버로딩

윤성우 저 열혈강의 C++ 프로그래밍 개정판

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).ShowData();
```

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

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.

펑터(Functor)의 위력


```
class DataStorage    // for int data
{
private:
   int * arr;
   int idx;
   const int MAX_LEN;
public:
   DataStorage(int arrlen) :idx(0), MAX_LEN(arrlen)
   {
      arr=new int[MAX_LEN];
   }
   void AddData(int num)
   {
      if(MAX_LEN<=idx)
      {
         cout<<"더 이상 저장이 불가능합니다."<<endl;
         return;
      }
      arr[idx++]=num;
   }
   void ShowAllData()
   {
      for(int i=0; i<idx; i++)
         cout<<arr[i]<<' ';
      cout<<endl;
   }
```

```
int main(void)
{
   DataStorage storage(5);
   storage.AddData(40);
   storage.AddData(30);
   storage.AddData(50);
   storage.AddData(20);
   storage.AddData(10);

   storage.SortData(AscendingSort());
   storage.ShowAllData();

   storage.SortData(DescendingSort());
   storage.ShowAllData();
   return 0;
}
```

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

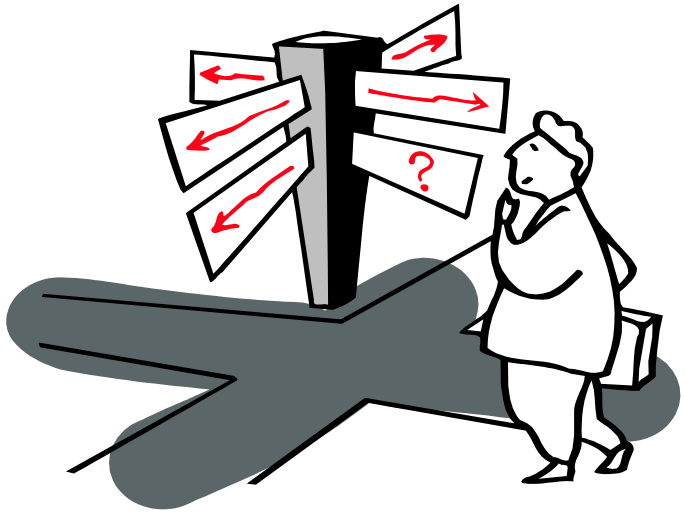

```
class Number
{
private:
    int num;
public:
    Number(int n=0) : num(n)
    {
        cout<<"Number(int n=0)"<<endl;
    }
    Number& operator=(const Number& ref)
    {
        cout<<"operator=()"<<endl;
        num=ref.num;
        return *this;
    }
    operator int ()     // 형 변환 연산자의 오버로딩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

해당 객체가 int형으로 변환되어야 하는 상황에서 호출되는 함수



Chapter 11이 끝났습니다. 질문 있으신지요?