스마트 포인터

- 학습지식 개요/요점
  - 자동 파괴자 auto_ptr 이란
  - auto_ptr의 필요성
  - auto_ptr의 사용방법
  - auto_ptr의 내부구조
  - shared_ptr와 auto_ptr의 비교

자동 파괴자 auto_ptr이란

❑ 자동 파괴자 auto_ptr

- C++에서는 객체 생성 시 사용했던 동적 메모리 또는 시스템 자원을 소멸 시 자동으로 소멸할 수 있는 매커니즘을 제공한다.
- 범위를 벗어난 변수는 스택에서 제거되며, 객체의 파괴자가 호출되어 자신이 사용하던 자원을 알아서 정리한다.

❑ 동적 메모리 해제 시 문제점

- **일반 파괴자는 스택으로 할당된 객체에 대해서는 소멸을 하지만, 동적으로 할당한 메모리에 대해서는 책임지지 않는 문제점이 있다.**
- **실수로 delete문을 빼먹은 경우**

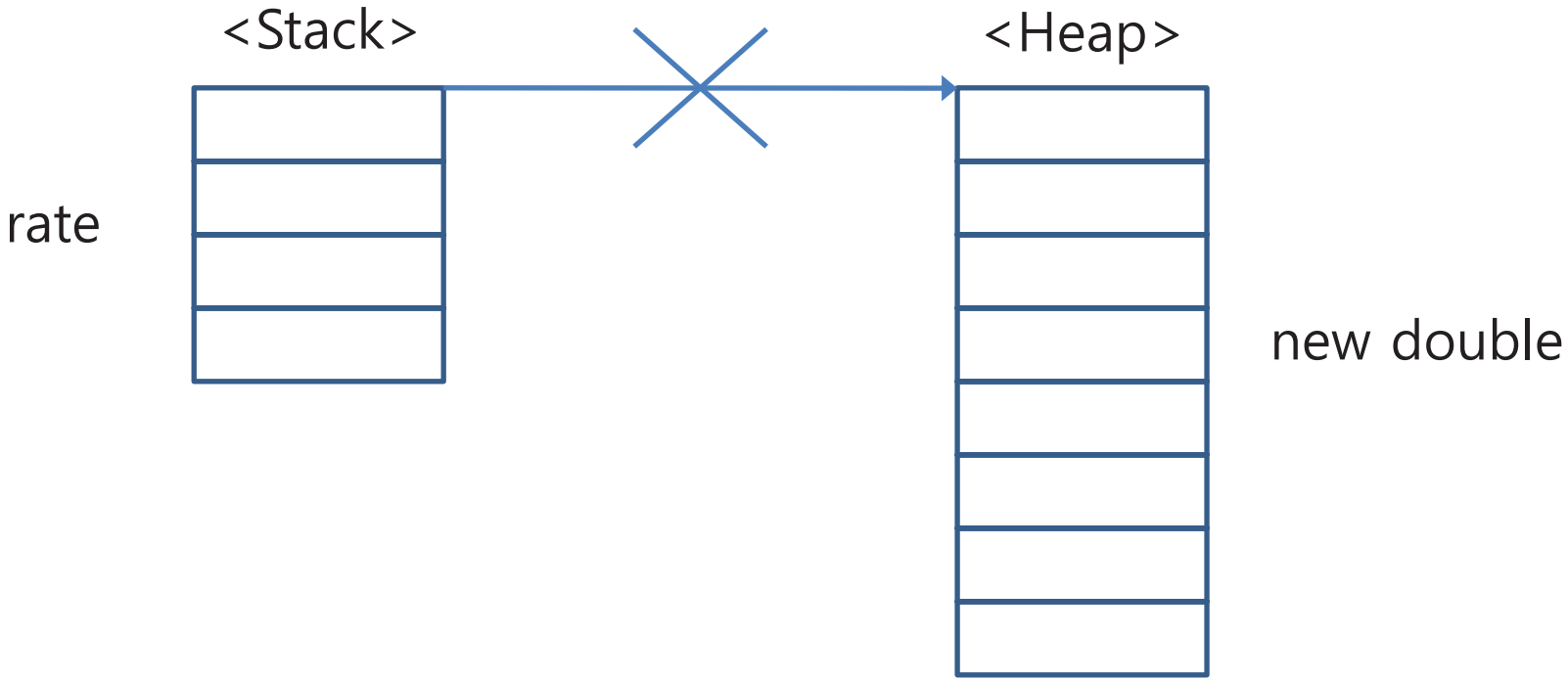
```
#include <iostream>
using namespace std;

void main()
{
    double *rate;

    rate=new double;
    *rate=3.1415;
    cout << *rate << endl;
    // delete rate;
}
```

❑ 동적 메모리 해제 시 문제점

- rate 변수는 동적으로 생성한 실수형 변수의 메모리를 저장하는 포인터 변수이다.
- rate 변수의 동적 메모리를 해제하는 delete 수행을 주석 처리한다.
- rate 변수는 종료 시에 자동으로 메모리가 소멸되지만, 이 변수가 가리키는 주소 메모리는 자동으로 해제되지 않는다. 즉, 메모리 누수(Memory Leak)가 발생한다.
- 동적으로 할당된 메모리는 이름이 없으므로 포인터를 잃어버리면 참조할 수 없어서 해제가 어렵게 된다.



❑ 동적 메모리 해제 시 문제점

- **짧은 코드에서는 delete문을 빼먹는 실수는 하지 않을 것이다.**
- **두 번째 경우는 예외 발생 시 메모리 해제 코드를 건너뛰는 경우이다.**

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

void main()
{
        int* a;
        int b;
        cout<<"나누는 수 입력: ";
        cin>>b;
        try{
                a = new int(10);
                if(b==0) throw b;
                cout<<"나누기결과: " <<*a/b<<endl;
                delete a;
        }
        catch(int ex){
                cout<<"나누어야할수가"<<ex<<" 이므로 연산을 수행할 수 없습니다."<<endl;
        }
}
```

auto_ptr의 필요성

- **동적 메모리 해제 시 문제점**
  - 정상적인 실행흐름이면 new/delete가 짝을 이루어 할당 해제가 수행되지만, 예외 조건에 의해 예외처리 시에는 catch 문 수행 후 delete는 수행하지 못하게 된다.
  - 지역객체가 가리키는 메모리까지 해제되는 것은 아니기 때문에 메모리 누수(Memory Leak)가 발생한다.
  - 이러한 메모리 누수는 양이 많지 않아 당장은 별 문제가 되지 않지만, 오랫동안 실행되는 프로그램은 시스템 자원을 갉아먹기 때문에 나중에는 심각한 문제가 될 수 있다.
  - 이러한 문제를 해결하기 위해 만들어진 것이 바로 auto_ptr 이다.

auto_ptr의 사용방법

❑ auto_ptr의 형태

- **auto_ptr은 동적으로 할당된 메모리도 자동으로 해제하는 포인터의 레퍼 클래스이다.**
- **auto_ptr 템플릿은 memory 헤더 파일에 정의되어 있으므로 사용시 헤더를 선언한다.**
- **auto_ptr 템플릿은 내부에 다음과 같이 정의되어 있다.**

```
template<typename T> class auto_ptr
```

- **포인터가 가리키는 대상체의 타입 T 를 인수로 받아 들이며 T* 형의 포인터를 관리한다.**
- **생성자로 전달한 포인터는 소멸자에서 delete로 해제하므로 포인터 뿐만 아니라 포인터가 가리키는 메모리도 자동으로 해제된다.**

실습 예제 및 수행가이드

- **auto_ptr 예제**

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

void main()
{
        auto_ptr<double> rate(new double);
        *rate = 3.1415;
        cout<<*rate<<endl;
}
```

- **새로운 double형 변수를 동적으로 할당하여 생성자로 전달했다.**
- **rate의 소멸자에서는 delete를 자동으로 호출하므로 함수가 끝날 때 rate를 따로 해제할 필요가 없으며 해제되지도 않는다.**
- **rate 객체 자체는 포인터가 아니기 때문에 delete rate 코드를 추가하면 컴파일 에러가 난다.**

auto_ptr의 내부구조

❑ 포인터의 래퍼 클래스 만들기

- **auto_ptr 템플릿은 포인터를 클래스로 감싸서 파괴자가 자동으로 해제할 수 있도록 만든 레퍼 클래스이다.**
- **auto_ptr 클래스의 기능을 기반으로 내부 구조를 유추하여 래퍼 클래스를 만들어보자.**
- **smartPointer는 인수로 전달한 타입 T에 대한 포인터를 감싸는 래퍼 클래스이다.**
- **객체 정리 시 소멸자를 통해서 포인터가 delete 될 수 있도록 처리하였고, *, ->에 대한 연산자 오버로딩을 하였다.**

```
template <class T>
class smartPointer
{
private:
        T *p;
public:
        smartPointer(T *sp) : p(sp){}
        ~smartPointer(){delete p;}
        T& operator*() const{return *p;}
        T* operator->() const{return p;}
};
```

실습 예제 및 수행가이드

- **smartPointer 예제**

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

template <class T> class smartPointer
{
    private:
            T *p;
    public:
            smartPointer(T *sp) : p(sp){}
            ~smartPointer(){delete p;}
            T operator*() const{return *p;}
            T* operator->() const{return p;}
};

void main()
{
            smartPointer<string> pStr(new string("test"));
            cout<<pStr.operator *()<<endl; //cout<<*pStr<<endl;
            cout<<pStr->size()<<endl; //cout<<pStr->size()<<endl;
}
```

auto_ptr의 내부구조

- **smartPointer 클래스 분석**
  - **생성자에서 T* p를 선언하는데 이 포인터는 생성자에서 초기화된다.**
  - **객체 소멸 시 소멸자 내부에서는 포인터 p를 delete 시킴으로써 p와 p가 가리키는 힙 메모리까지 모두 정리된다.**
  - **오버로딩한 * 연산자를 객체로 호출하면 내부에서는 *p를 리턴한다.**
  - **오버로딩한 -> 연산자를 객체로 호출하면 내부에서는 p를 리턴한다.**

shared_ptr과 auto_ptr의 비교

- **shared_ptr이란**
  - **shared_ptr는 C++ 11 표준 라이브러리에 추가된 스마트 포인터**
  - **auto_ptr의 단점을 보완함.**
  - **사용 형태는 auto_ptr과 동일함**

실습 예제 및 수행가이드

- **shared_ptr 예제**

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

void DoSomething( )
{
    auto_ptr<int> ptr1= auto_ptr<int>( new int(10) );
    shared_ptr<int> ptr2= shared_ptr<int>( new int(50) );
    int *ptr3 = new int(100);

    cout << *ptr1 << endl;
    cout << *ptr2 << endl;
    cout << *ptr3 << endl;
    delete ptr3;
}
void main()
{
            DoSomething();
}
```

shared_ptr과 auto_ptr의 비교

- **shared_ptr와 auto_ptr의 차이점**
  - **auto_ptr : 유일한 소유권 개념이 있어서 객체가 복사되는 순간 원래의 auto_ptr 객체는 바로 NULL 처리가 된다.**

```
class A;
auto_ptr<A> aObj(new A());
auto_ptr<A>bObj(aObj); //aObj는 NULL이 되고, bObj만 객체를 가리킨다.
```

  - **shared_ptr : 참조 카운팅 방식을 사용. 특정 자원을 가리키는 참조 카운트를 유지하고 있다가 이것이 0이 되면 해당 자원을 자동 삭제해준다.**

실습 예제 및 수행가이드

- **shared_ptr 예제**

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

void DoSomething( )
{
        auto_ptr<int> ptr_a1= auto_ptr<int>( new int(10) );
        auto_ptr<int> ptr_a2= ptr_a1;
        auto_ptr<int> ptr_a3= ptr_a2;

        shared_ptr<int> ptr_s1= shared_ptr<int>( new int(50) );
        shared_ptr<int> ptr_s2= ptr_s1;
        shared_ptr<int> ptr_s3= ptr_s2;

        int *ptr_1 = new int(100);
        int *ptr_2 = ptr_1;
        int *ptr_3 = ptr_2;

        cout << *ptr_a1<<','<<*ptr_a2<<','<<*ptr_a3 << endl;
        cout << *ptr_s1<<','<<*ptr_s2<<','<<*ptr_s3 << endl;
        cout << *ptr_1<<','<<*ptr_2<<','<<*ptr_3 << endl;

       delete ptr_1;
}
```

실습 예제 및 수행가이드

- **shared_ptr 예제**

```
void main()
{
        DoSomething();
}
```