

스마트포인트어 구현

김성익(noerror@hitel.net)
2005.03.25

개요

- 메모리 관리의 어려움
- 메모리 Leak의 문제점
- 자바의 가비지 컬렉션

템플릿(1)

- 데이터 타입(혹은 상수)때문에 메소드를 재정의 해야 하는 경우

```
float MIN(float a, float b)
{
    return a > b ? b : a;
}

int MIN(int a, int b)
{
    return a > b ? b : a;
}
```

- **template** 을 이용하면 다양한 데이터형에 적합한 구현이 가능

```
template <class _T>
_T MIN(_T a, _T b)
{
    return a > b ? b : a;
}

#endif
```

템플릿(2)

- 효율적인 프로그래밍 작업가능
- Generic 프로그래밍이란 ?

객체 생성/소멸자

- 객체를 더 이상 사용하지 않을 경우에는 소멸자 호출

```
#include <stdio.h>
class Object
{
public :
    Object() { printf("생성\n"); }
    ~Object() { printf("소멸\n"); }
};

void main()
{
    Object b;
}
```

- 생성자와 소멸자는 항상 짝을 이룸

기타

- 클래스 연산자 오버라이딩

Leak

- 생성을 했으나 해제를 안 한 경우

```
struct _TEST {  
    int value ;  
};  
  
void main()  
{  
    _TEST * a = new _TEST;  
    a->value = 10;  
}
```

- 빈번하게 발생
- 프로그래밍하는 작업자의 몫
- 근본적으로 자동화 가능하다면 ?

스마트포인터

- 객체(메모리, 리소스)의 생성과 소멸 자동화
- 일반 포인터 사용하듯이 사용
- 컴파일러에 의한 객체의 생성과 소멸 호출 이용
- 연산자 오버라이딩
- 템플릿을 이용한 generic한 형태로 구현

과제: 자원 자동 해제

- 생성과 소멸자를 이용하여 포인터 자동 해제처리

```
struct _TEST {
    int value;
};

class MyPointer
{
public:
    MyPointer() { m_pPointer = NULL; }
    ~MyPointer() {
        if (m_pPointer)
            delete m_pPointer;
    }
    void SetPointer(_TEST * t) {
        if (m_pPointer)
            delete m_pPointer;
        m_pPointer = t;
    }

private:
    _TEST * m_pPointer;
};
```

```
void main()
{
    MyPointer b;
    _TEST * a = new _TEST;

    b.SetPointer(a);

    a->value = 10;

    a = new _TEST;
    b.SetPointer(a);
    a->value = 11;
}
```

과제:일반 포인터 사용하듯이 사용

- 연산자 오버라이딩을 이용해서 그냥 포인터 사용하듯이 사용

```
struct _TEST {
    int value ;
} ;

class MyPointer
{
public :
    MyPointer() { m_pPointer = NULL; }
    ~MyPointer() {
        if (m_pPointer)
            delete m_pPointer;
    }
    operator = (_TEST * t) {
        if (m_pPointer)
            delete m_pPointer;
        m_pPointer = t;
    }
    _TEST * operator -> () { return m_pPointer; }

private :
    _TEST * m_pPointer;
} ;
```

```
void main()
{
    MyPointer a;
    a = new _TEST;
    a->value = 10;

    a = new _TEST;
    a->value = 11;
}
```

과제:일반적인 형태로 사용

- 템플릿을 사용하면 타입의 제한 없이 사용가능

```
template <class _T>
class MyPointer
{
public :
    MyPointer() { m_pPointer = NULL; }
    ~MyPointer() {
        if (m_pPointer)
            delete m_pPointer;
    }
    operator = (_T * t) {
        if (m_pPointer)
            delete m_pPointer;
        m_pPointer = t;
    }
    _T * operator -> () { return m_pPointer; }

private :
    _T * m_pPointer;
};
```

```
struct _TEST {
    int value ;
};

void main()
{
    MyPointer <_TEST> a;
    a = new _TEST;
    a->value = 10;

    a = new _TEST;
    a->value = 11;
}
```

과제: 자원 공유

- 단순히 자원 해제에만 사용하지 않고, 같이 참조했을 때 중복 소멸하지 않아야 한다

```
void main()
{
    CSmartPointer <_TEST> a;
    CSmartPointer <_TEST> b;
    a = new _TEST;
    a->value = 10;

    b = new _TEST;
    b->value = 20;

    b = a;
}
```

- 해당 자원을 다른 스마트 포인터에서 사용 중인지를 알아야 한다

- 자원 사용시 이중 링크드 포인트로 연결
*삽입해제의 손쉬움을 위해서
연결된 게 없다면 마지막 사용자*
- 최종 코드

```
template <class _T>
class CSmartPointer
{
public :
    CSmartPointer() {
        m_pPointer = NULL;
        m_pPrev = m_pNext = this;
    }
    ~CSmartPointer() {
        Release();
    }
    operator = (_T * t) {
        Release();
        m_pPointer = t;
    }
}
```

```

CSharedPointer <_T> & operator = (CSharedPointer <_T> & pt)
{
    Release();
    m_pNext = &pt;
    m_pPrev = pt.m_pPrev;
    pt.m_pPrev->m_pNext = this;
    pt.m_pPrev = this;
    m_pPointer = pt.m_pPointer;
    return *this;
}
_T * operator -> () { return m_pPointer; }

private :
void Release()
{
    if (m_pNext == this) // m_pPrev == this && m_pNext == this
    {
        if (m_pPointer != NULL)
            delete m_pPointer;
    }
    else
    {
        m_pPrev->m_pNext = m_pNext;
        m_pNext->m_pPrev = m_pPrev;
        m_pPrev = m_pNext = this;
    }
    m_pPointer = NULL;
}
_T * m_pPointer;
CSharedPointer <_T> * m_pPrev, * m_pNext;
};

```

응용 : 소멸자정의

- 엔진에 응용하거나, 핸들에 응용할 경우 다른 소멸자 필요

```
template <class _T> struct _release_mem {
public :
    void Release(_T *t) const { delete t; }
};

template <class _T, class _R = _release_mem <_T> >
class MyPointer
{
public :
    MyPointer() { m_pPointer = NULL; }
    ~MyPointer() {
        if (m_pPointer) {
            _R r;
            r.Release(m_pPointer);
        }
    }
    operator = (_T * t) {
        if (m_pPointer) {
            _R r;
            r.Release(m_pPointer);
        }
        m_pPointer = t;
    }
    _T * operator -> () { return m_pPointer; }

private :
    _T * m_pPointer;
};
```

```
struct _TEST {
    int value;
};

template <class _T> struct _free_mem {
public :
    void Release(_T *t) const { free(t); }
};

void main()
{
    MyPointer <_TEST> a;
    MyPointer <_TEST, _free_mem <_TEST> > b;
    a = new _TEST;
    a->value = 10;

    b = (_TEST*) malloc(sizeof(_TEST));
    b->value = 11;
}
```

질문

참고

- **Modern C++ Design: Generic Programming and Design Patterns Applied**
Andrei Alexandrescu, Addison Wesley, 2001