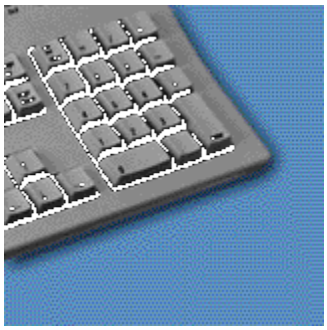


4. C++의 주제들 II

<장도비라>
우리는 2,3장에서 MFC의 소스 분석에 필요한 C++의 주제와 전처리 명령어들에 대해서 살펴보았다. 이 장은 5장부터 시작될 본격적인 MFC의 코드 분석에 앞서 C++의 주제들을 살펴보는 마지막 장이다. 이 장에서는 복사 생성자, 가상함수와 RTTI에 대해서 살펴볼 것이다. 이 주제들은 MFC에서 뿐만이 아니라 일반적으로도 매우 중요한 주제들이므로 모두를 완벽하게 이해해야 한다.
</장도비라>

이 장에서 설명하는 내용들은 2,3장에 이어 MFC의 소스 분석에 필요한 C++의 고급 주제들이다. 이 장 이후의 내용에서는 필자는 독자들이 C++의 모든 기능에 자유하다고 가정한다. 예를 들면 이 장 이후의 내용 중에

'A타입에 대한 객체를 ASibling타입으로 dynamic_cast<>()한다'

라는 문장이 있다면, 필자는 독자들이 dynamic_cast<>()에 자유하다고 가정할 것이므로 dynamic_cast<>()에 대한 추가적인 설명은 하지 않을 것이다. 이 장에서 설명할 내용은 다음과 같다.

- 복사 생성자(copy constructor)
- 가상함수(virtual function)
- RTTI(run-time type information)

복사 생성자(copy constructor)

<절도비라>
이 절에서는 생성자 중에서 객체가 복사되는 경우에 호출되는 특별한 복사 생성자의 원리와 동작을 이해한다. 또한 복사 생성자를 잘못 사용하는

경우 메모리 릭(memory leak)이 발생하는 과정을 자세하게 살펴본다.
</절도비라>

<저자 한마디>
복사 생성자의 원리를 이해하는 것은 매우 매우 매우 중요하다. 원리를 이해하지 못한다면 소스의 어딘가에서 발생한 메모리 릭을 찾느라, 몇일 밤을 고생할 것이고, 몇일 밤을 고생하더라도 메모리 릭이 일어난다는 현상만 알뿐 왜 메모리 릭이 일어나는지에 대해서 도무지 감을 잡을 수 없을 것이다. 예를 들면, STL(standard template library)의 컨테이너(container)가 어떤 타입을 포인터가 아니라 객체로 유지한다고 하자. 이 객체가 누군가에 의해 변경되면서 내부에서 동적 메모리 할당을 사용하였다. 이런! 이 객체에 복사 생성자가 없다면 엄청난 수의 메모리 릭이 발생할 것이다.
</저자 한마디>

구조체를 함수의 파라미터로 전달하는 경우를 생각해 보자. 구조체를 함수에 전달할 때, 값에 의한 전달(call by value)의 경우에도 구조체를 전달하기보다는 구조체의 포인터를 전달한다. 그것은 구조체의 크기가 클수록 이점이다. 크기가 1024바이트인 구조체의 경우, 구조체를 모두 전달한다면, 스택에 1024바이트가 모두 푸시(push)되어야 한다. 하지만, 구조체의 포인터를 전달한다면, 4바이트 포인터만 스택에 푸시하면 되는 것이다.

C++에서 객체가 전달되는 경우도 같은 상황을 생각해 볼 수 있다. 객체를 모두 전달하는 것보다는 객체의 포인터만을 전달하는 것이 확실히 이득이다. 물론 객체의 내용이 바뀌기를 원치 않는다면 조심스럽게 함수의 몸체를 코딩해야 할 것이다.

하지만, 분명히 객체가 전달되어야 하는 경우가 있다. 이 때 미묘한 문제가 발생할 수 있다. 함수의 파라미터로 전달되는 객체는 이미 메모리에 생성되어 있을 것이다. 그리고 이 객체가 파라미터로 전달될 때는 이미 만들어진 객체의 내용이 스택에 복사(copy)되어 전달될 것이다. 이 때 생성자가 호출되어야 하는가? 또한, 함수를 빠져 나올 때 파괴자는 호출되어야 하는가?

아래의 [예제 4.1]을 보자. CStr클래스는 문자열을 관리한다. 생성자는 동적으로 메모리를 할당하여, 문자열을 복사하며, 파괴자는 동적으로 할당된 메모리를 해제한다. 문자열의 시작 위치는 str이 가리킨다. 멤버 함수 Print()는 문자열 str을 출력한다. main()에서는 객체 s를 만들어 문자열을 "hello"로 초기화한다. 객체 s는 함수 f()에 전달되며, s의 문자열이 출력된다. 컴파일 시간 에러는 없다. 하지만, 실행 시간 에러(run-time error)가 발생한다. 어디가 잘못인가?

[예제 4.1] 잘못된 코드

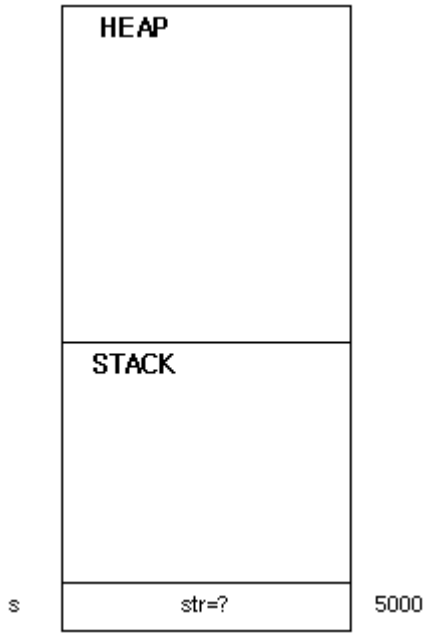
```
#include <stdio.h>
#include <string.h>
#include <conio.h>

class CStr {
    char *str;
  public:
    CStr(char *s="") {
        str=new char[strlen(s)+1];//EOS를 위해 +1이 필요하다.
        strcpy(str,s);//str=s;라고 쓰지 않도록 하자.
        printf("constructor\n");
    }
    ~CStr() {
        delete[] str;//delete str;이라고 쓰지 않도록 하자.
        printf("destructor\n");
    }
    void Print() { printf("%s\n",str); }
};//class CStr

void f(CStr s) {
    s.Print();
}

void main() {
    CStr s("hello");
    f(s);
}
```

실행 결과는 다음과 같다.

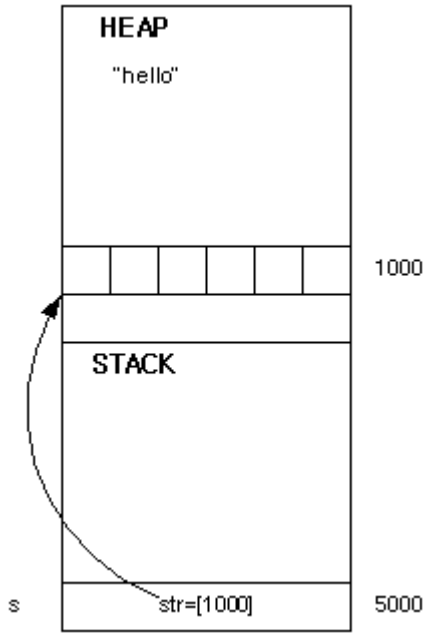
```
constructor
hello
destructor
(에러 발생)
```

즉 프로그램은 정상적으로 종료하지 않고 **GPF(general protection fault)**를 발생시킨다. 세 번째 출력인 destructor는 언제 출력된 것일까? 이것은 main()에서 s객체가 파괴될 때가 아니라, 함수 f()에서 복사된 객체 s가 파괴될 때 실행된 것이다. 마지막의 에러는 main()이 객체 s의 파괴자를 호출할

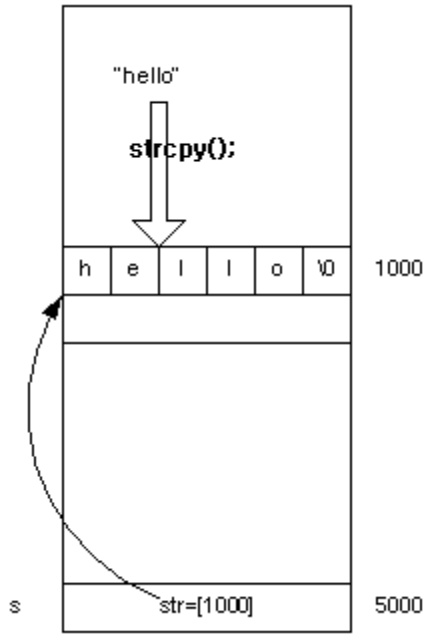
때 발생한다. 이미 해제된 메모리를 해제하려 한 것이다! 이제 프로그램을 자세히 따라가 보자.



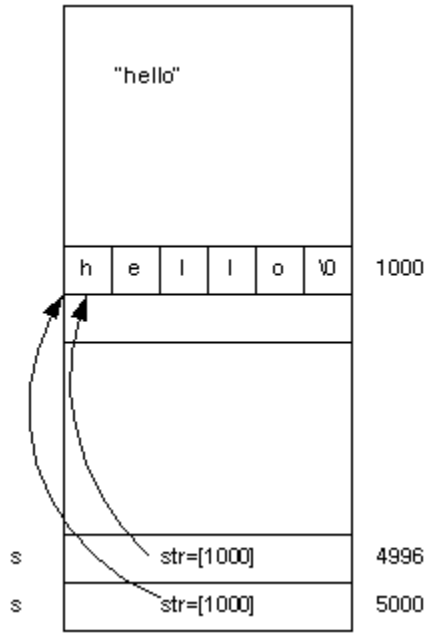
[그림 4.1] CStr s("hello");: main()함수의 객체 선언에 의해 객체 s가 스택에 만들어진다. s의 멤버 str은 아직 초기화되지 않았다. 이제 메모리가 할당되었으므로 생성자가 호출될 것이다.



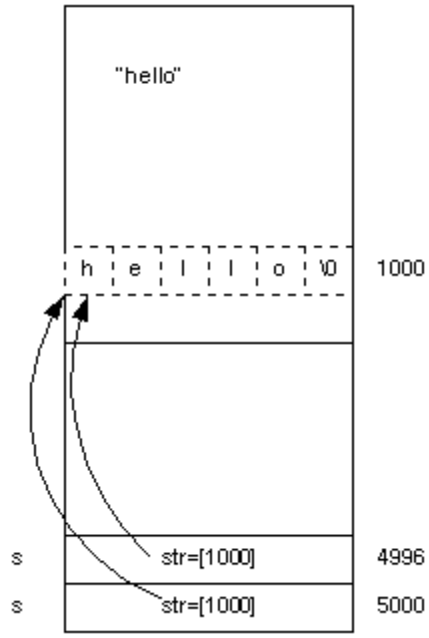
[그림 4.2] str=new char[strlen(s)+1];: 생성자가 호출된다. 6바이트를 동적으로 할당한다. 시작 주소를 [1000]이라 가정하자. 파라미터로 전달된 "hello"는 C 컴파일러가 문자열을 위해 관리하는 힙 메모리에 잠시 복사되어 있다. s의 멤버 str은 [1000]으로 설정된다.



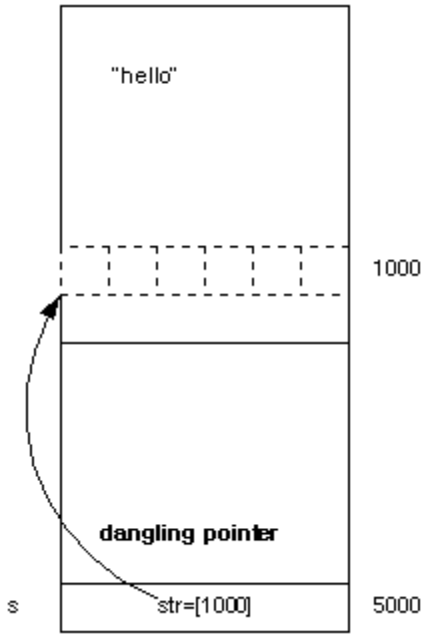
[그림 4.3] strcpy(str,s);: 임시 메모리에 있는 문자열 s가 str이 가리키는 곳에 복사된다. 끝에 \0도 같이 복사된다는 것에 주의하라. 그리고 printf()에 의해 "constructor"가 출력된다.



[그림 4.4] f(s);: 함수 f()로 객체 s가 전달된다. 객체 s의 내용이 그대로 복사되어(bitwise copy) 스택이 1개의 복사본이 만들어진다. 5000번지의 s는 main()의 객체 s이며, 4996번지의 s는 f()의 객체 s이다. **s의 내용이 복사된 것이지 생성된 것이 아니므로, s의 생성자는 호출되지 않는다**. f()의 str역시 [1000]을 가리키고 있다는 사실에 주목하라.



[그림 4.5] s.Print();: 멤버 함수 CStr::Print()가 호출되어 "hello"가 출력된다. 이제 함수 f()를 빠져 나간다. f()의 객체 s가 스택에서 해제되기 전, 파괴자가 호출된다. **객체는 메모리에서 사라지기 바로 전 항상 파괴자가 호출된다는 사실을 상기하자**. 파괴자는 1000번지를 해제한다. 이제 1000번지는 유효한 메모리 공간이 아니다. 문제의 핵심은, 파라미터로 전달된 객체의 생성자는 호출되지 않지만, 파괴자는 호출된다는 것이다.



[그림 4.6] Dangling pointer: 이제 main()의 객체 s의 멤버 str이 가리키는 곳은 무효한(invalid) 곳이다. 무효한 곳을 가리키는 포인터를 **댕글링 포인터(dangling pointer)**라 한다. main()이 종료하기 직전, 객체 s가 스택에서 해제되기 전 파괴자가 호출될 것이고, 파괴자는 1000번지를 메모리 해제하려 할 것이다. 이것은 실행시간 에러를 유발한다.

이 문제 - 동적으로 메모리 할당을 하는 생성자를 가진 객체를 함수의 파라미터로 직접 전달하는 문제 - 에 대한 해결책은 두 가지이다.

- 객체를 전달하지 않고 포인터 혹은 레퍼런스를 전달한다.
- 복사 생성자(copy constructor)를 만들어 둔다.

첫 번째 해결책은 함수 f()를 다음과 같이 수정하는 것이다.

```
...
void f(CStr& s) {
    s.Print();
}
```

만약 함수 f()를 void f(CStr* s) { s->Print(); }처럼 수정한다면, main()에

서는 f(s); 를 f(&s);로 수정하여야 할 것이다. 규칙은 다음과 같다.

"생성자에서 동적으로 메모리를 할당하는 객체를 함수의 파라미터로 전달할 때는 항상 포인터 혹은 레퍼런스를 전달하라."

더 일반적인 해결책은 **복사 생성자(copy constructor)**를 만드는 것이다. 복사 생성자는 객체가 만들어질 때가 아니라, 이미 만들어진 객체가 다

<주의>복사 생성자를 사용하면 안 되는 클래스를 만들 때에도 복사 생성자를 만드는 것이 필요하다. 그 클래스의 복사 생성자에서는 ASSERT()로 무조건 실패하도록 만들어 진다. 이러한 설계는 잘못된 복사 생성자의 사용으로 인한 프로그램의 크래시(crash)를 막는다.</주의>

른 곳으로 복사될 때 호출되는 특별한 생성자를 의미한다. 객체가 복사되는 경우는

① 함수의 파라미터로 객체를 전달
② 선언에서 초기화 문장
③ 오버로드된 연산자 혹은 함수가 객체를 리턴하는 경우
④ 임시 객체가 복사되는 경우

등에 발생한다.

복사 생성자는 일반 생성자와 다르므로, 특별한 문법 규칙을 필요로 한다. 복사 생성자를 정의하는 방법은 다음과 같다.

- **반드시 자신의 상수 클래스 타입의 레퍼런스를 파라미터로 받는다.**

그러므로 CTest의 복사 생성자는 다음과 같이 정의될 것이다.

<여기서 잠깐>이것은 실제로 CTest 클래스를 CTest 클래스로 타입 변환하는 **사용자 정의 타입 변환(user-defined type conversion)**을 정의한 것이다. 예를 들면, class X를 class Y로 타입 변환하기 위해서는 class Y의 멤버 함수에 Y(const X& x); 를 추가하면 된다.</여기서 잠깐>

```
CTest::CTest(const CTest& s) {...
```

위 문장은 s가 변경되지 못함과, s의 레퍼런스가 전달됨을 지시한다. 복사 생성자는 항상 위와 같이 첫 번째 파라미터만이 명시되어야 한다.

그러면, 이 예에서 복사 생성자는 어떻게 정의되어야 할까? 객체 s가 복사될 때 str이 가리키는 내용 역시 복사되도록 생성자를 정의하여야 할 것이다. 이렇게 포인터가 가리키는 내용까지 복사하는 것을 **깊은 복사(deep copy)**라 한다.

```
    CStr (const CStr &s) {//copy constructor
        str=new char[strlen(s.Get())+1];
        strcpy(str,s.Get());
        printf("copy constructor\n");
    }
```

CStr::Get()은 단순히 str을 리턴한다. 단, s가 상수이므로, Get()역시 상수 함수로 선언되어야 할 것이다. 수정된 소스는 [예제 4.2]와 같다.

[예제 4.2] 깊은 복사(Deep Copy)를 지원하는 수정된 소스

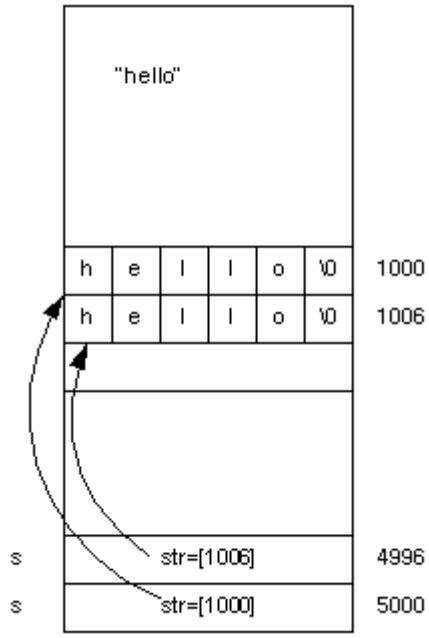
```
#include <stdio.h>
#include <string.h>

class CStr {
    char *str;
  public:
    CStr(char *s="") {
        str=new char[strlen(s)+1];
        strcpy(str,s);
        printf("constructor\n");
    }
    CStr (const CStr &s) {//copy constructor
        str=new char[strlen(s.Get())+1];
        strcpy(str,s.Get());//const 객체는 const 멤버 함수만을 호출할 수
                            //있다.
        printf("copy constructor\n");
    }
    ~CStr() {
        delete[] str;
        printf("destructor\n");
    }
```

```
    char *Get() const {//함수 뒤의 const는 이 함수가 어떠한 멤버도 변경
                       //시켜서는 안됨을 지시한다. '변경자'와 'this포인
                       //터'에서 자세히 설명할 것이다.
        return str;
    }
    void Print() { printf("%s\n",str); }
};//class CStr

void f(CStr s) {
    s.Print();
}

void main() {
    CStr s("hello");
    f(s);
}
```

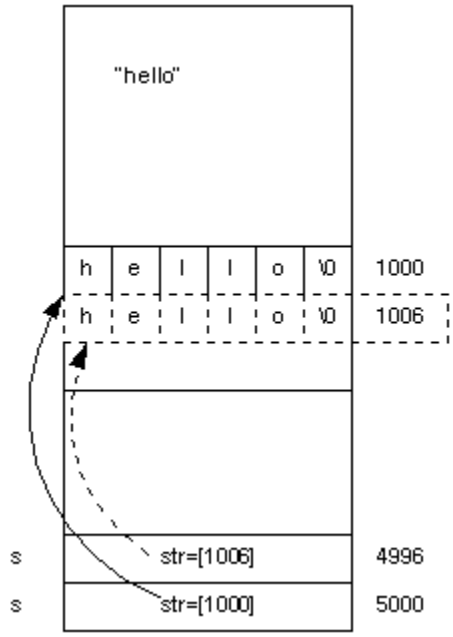
실행 결과는 다음과 같다.

```
constructor
copy constructor
hello
destructor
destructor
```

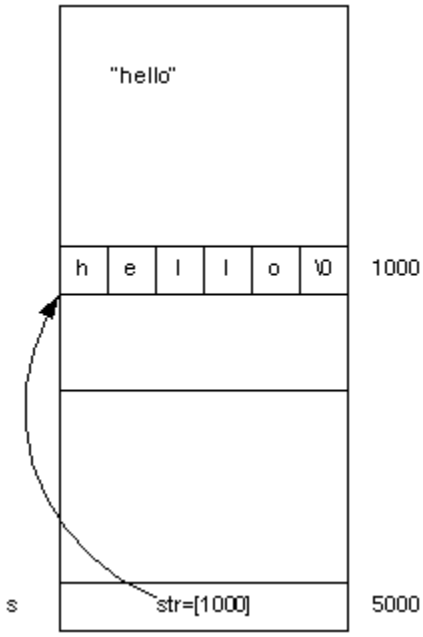
이제 실행시간 에러(run-time error)는 더 이상 발생하지 않는다. 복사 생성자가 정의된 후의 프로그램 동작은 아래 그림과 같다.



[그림 4.7] f(s);: 이제 main()의 s가 복사될 때 복사 생성자가 호출된다. 복사 생성자는 새로 메모리를 할당하여 main()의 s의 멤버 str이 가리키는 str을 복사한다. 새로 할당된 메모리를 [1006]이라 하자. main()의 s는 [1000]을 가리키고, f()의 s는 [1006]을 가리킨다.



[그림 4.8] 파괴자 호출: f()에서 호출된 파괴자는 [1006]을 해제한다.



[그림 4.9] main()의 s: main()의 s가 가리키는 1000번지는 여전히 유효하다. str은 더 이상 댕글링이 아니다.

복사 생성자가 호출되는 경우

복사 생성자가 정의되었을 때, 복사 생성자가 호출되는 경우는 아래의 네 가지이다.

- **함수의 파라미터로 객체의 전달**: 이미 만들어진 객체가 함수의 파라미터로 전달될 때, 객체가 스택에 복사되면서 복사 생성자가 호출된다. 위의 예에서 f(s); 같은 경우가 이에 해당한다. 함수 f()의 프리픽스(prefix) 코드는 객체 s를 메모리 할당한 후, 복사 생성자를 호출한다.

- **함수가 객체를 리턴**: 함수가 객체를 리턴하는 경우, 임시 객체가 대입문에 의해 복사될 때, 복사 생성자가 호출된다. 예를 들어 함수 f()의 원형이 CStr f(CStr s); 인 경우 CStr v("hello"),u; u=f(v); 의 문장에서 함수 f()가 호출될 때, 함수 f()가 리턴되어 임시 객체가 u에 복사될 때 복사 생성자는 2번 호출된다(operator=()이 정의되어 있지 않다고 가정). 함수가 객체를

리턴하는 경우는 operator=() 함수의 리턴 타입이 객체일 경우도 성립하며, 생성자를 직접 호출하여 임시 객체를 만드는 경우도 성립한다. 예를 들면, operator=()의 리턴 타입이 CStr일 때, CStr s("hello"); s=CStr("abcdef"); 에서 복사 생성자가 호출될 것이다.

- **객체의 선언에 사용된 대입연산자**: 객체를 선언하면서 초기화하는 대입 연산자는 복사 생성자를 호출한다. 예를 들면, CStr s("hello"); CStr t=s; 에서 두 번째 객체 선언문 t=s;는 복사 생성자를 호출한다.

- **임시 객체의 복사**: CStr s("hello"); 처럼 이미 객체 s가 만들어 졌으며, CStr에 오버로드된 연산자 함수 operator==()이 정의되었다고 가정하자. operator==() 함수는 2개의 객체가 같은지 비교한다. 그러면 if (s==CStr("world")) ... 라는 문장은 생성자를 직접 호출하여, 객체 s와 비교한다. 이처럼 생성자를 직접호출해서 만들어진 객체를 임시 객체라고 한다. 임시 객체는 이미 만들어진 객체를 초기화하기 위해서도 가끔 사용한다. s=CStr("world"); 라는 문장은 operator=()이 오버로드되어 있지 않다면, 복사 생성자를 호출한다.

CStr s("hello"); CStr t=s; 는 복사 생성자를 호출하지만, CStr s("hello"); CStr t; t=s;는 복사 생성자를 호출하지 않는다는 사실을 주의하라. 선언문에 사용된 대입 연산자와 일반 대입 연산자는 다르게 번역된다. 그렇다면 t=s;는 어떻게 동작할까? 이러한 대입문은 비트 단위 복사(bitwise copy)가 이루어진다. 즉, s의 내용은 1비트도 틀림없이 그대로 t로 복사된다.

대입 연산자가 비트 단위 복사로 번역되는 것은 심각한 문제를 야기하는 경우가 있다. 아래의 [예제 4.3]은 비트 단위의 복사가 심각한 논리적 에러를 유발한다는 것을 보여준다. operator=()이 정의되어 있지 않다면 객체간의 대입은 비트 단위의 복사, 즉 단순한 memcpy()처럼 동작한다. 결과를 보지 말고, 문제점을 찾아보라.

[예제 4.3] 논리적 에러를 포함한 코드

```
#include <stdio.h>
#include <string.h>
```

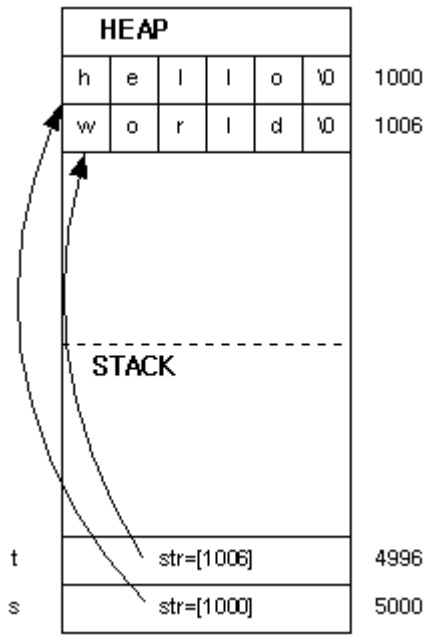
```
class CStr {
    char *str;
  public:
    CStr(char *s="") {
        str=new char[strlen(s)+1];
        strcpy(str,s);
        printf("normal constructor\n");
    }
    CStr (const CStr &s) {//copy constructor
        str=new char[strlen(s.Get())+1];
        strcpy(str,s.Get());
        printf("copy constructor\n");
    }
    ~CStr() {
        delete[] str;
        printf("destructor\n");
    }
    char *Get() const { return str; }
    void Print() { printf("%s\n",str); }
};//class CStr

void f(CStr s) {
    s.Print();
}

void main() {
    CStr s("hello"),
         t("world");
    CStr u=s;//객체의 선언문에 사용된 대입 연산자는 복사 생성자를 호출한
             //다.

    f(u);//함수의 파라미터로 전달된 객체는 복사 생성자를 호출한다.
    s=t;//객체 간의 대입문은 복사 생성자를 호출하지 않는다. 다만 그대로
        //복사(bitwise copy)된다.
    s.Print();
}
```

독자들은 실행 결과가 다음과 같을 것이라 예상할 것이다.

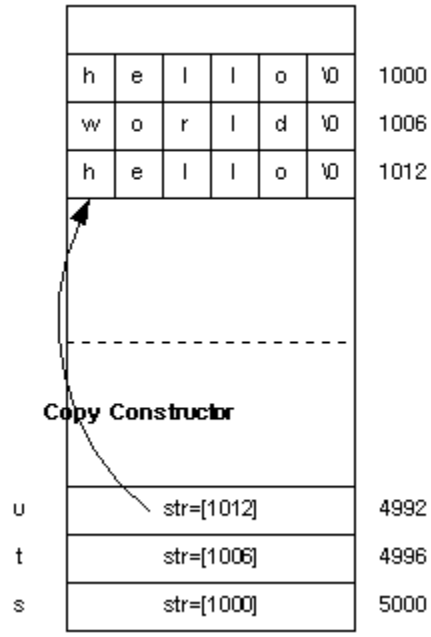
```
        normal constructor
        normal constructor
```

```
copy constructor
copy constructor
hello
destructor
world
destructor
destructor
destructor//이 문장이 출력되기 전 에러가 발생한다.
```

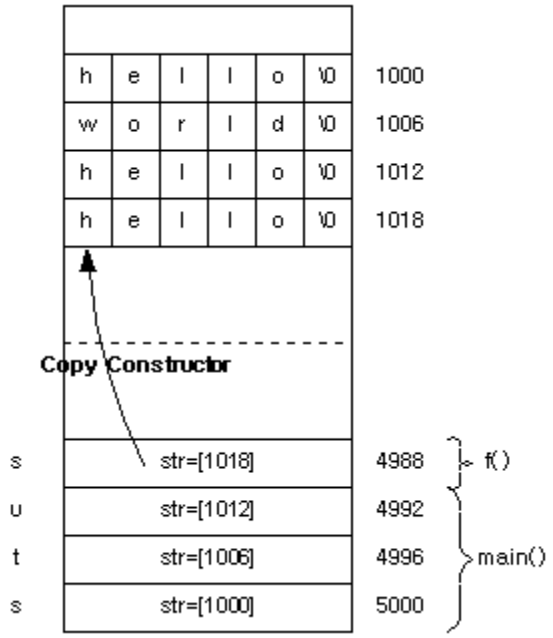
하지만, 마지막의 "destructor"는 출력되지 않는다. 왜 그러한 일이 발생하는가? 메모리 상태를 자세히 따라가 보자.



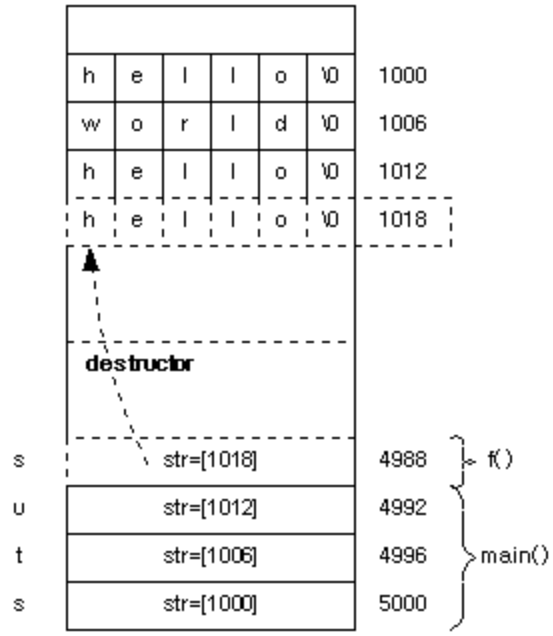
[그림 4.10] CStr s("hello"),t("world");: 객체 s와 t가 만들어지고, 생성자가 호출되어 문자열을 위한 메모리를 동적으로 할당한 후, "hello"와 "world"를 각각 복사한다.



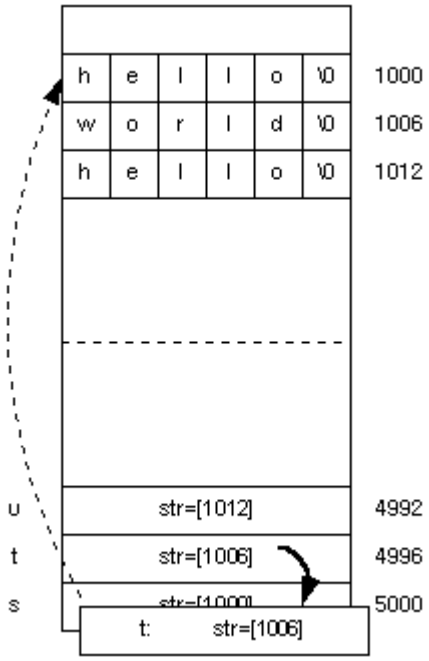
[그림 4.11] CStr u=s;: 객체 u를 위한 복사 생성자가 호출된다. s의 str이 가리키는 "hello"가 복사되어 u의 str이 가리키는 곳으로 복사된다.



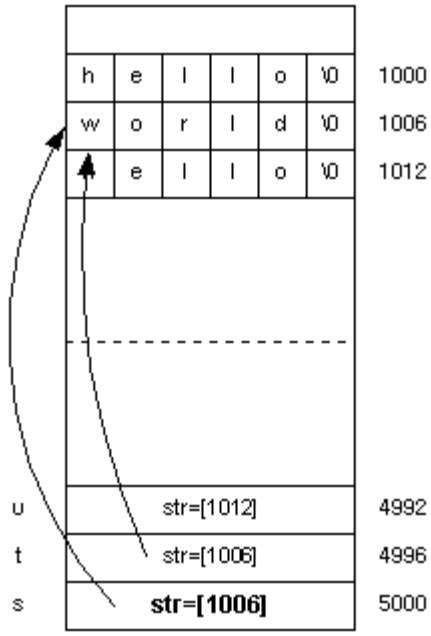
[그림 4.12] f(u);: main()의 객체 u가 f()의 객체 s로 복사되면서 s의 복사 생성자가 호출된다. 이제 힙에 "hello"는 3곳에 "world"는 1곳에 존재한다.



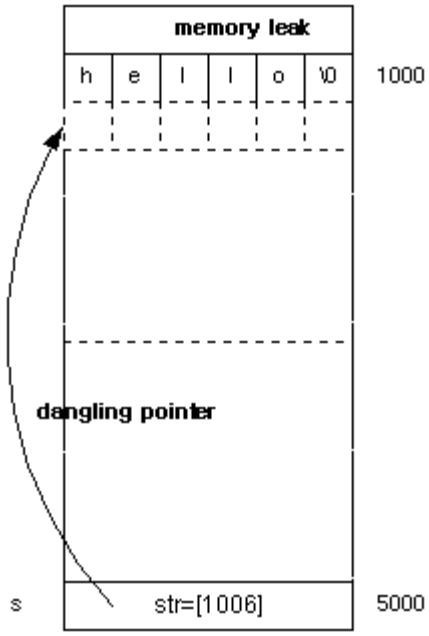
[그림 4.13] f()의 종료: 함수 f()가 리턴되기 전, 객체 s가 메모리에서 해제되므로, 파괴자가 호출된다. 1018번지는 메모리 해제된다.



[그림 4.14] s=t;: 객체 t가 s로 비트 단위 복사된다. 원래 s가 가지던 내용은 t의 내용으로 바뀌어진다. 원래 s의 str이 [1000]에서 [1006]으로 바뀌어지는 것에 주목하라. 이제 힙에 할당된 1000번지를 가리키던 포인터는 존재하지 않는다.



[그림 4.15] s.Print();: 이제 s의 str은 [1000]의 "hello"를 가리키지 않는다. [1006]의 "world"를 가리키므로, "world"가 출력된다.

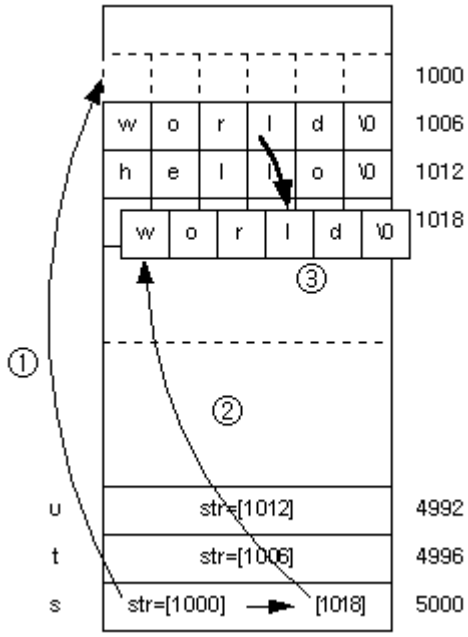


[그림 4.16] main()의 종료: main()이 종료하기 전 u,t와 s의 순으로 객체가 파괴될 것이다. u는 1012번지를 해제한다. t는 1006번지를 해제한다. s가 이미 해제된 [1006]을 다시 해제하려 하므로, 실행시간 에러가 발생한다. 또한, [1000]의 6바이트는 해제되지 않은 채로 남아 있다.

왜 이와 같은 문제가 발생하는가? 그것은 객체의 대입문이 비트 단위 복사(bitwise copy)를 수행하기 때문이다. 동적으로 메모리를 관리하는 객체 사이의 대입문은 특별하게 코딩되어야 한다.

이 예에서는 ① s가 이미 가리키는 [1000] 번지를 해제한 후, ② 다시 메모리를 할당하여 ③ t의 str이 가리키는 문자열을 대입하는 방식을 사용해야 할 것이다. 이 문제를 해결하기 위해서 operator=()을 정의해야 한다. =연산자는 아래와 같은 기능을 수행한다.

```
...
① delete[] str;
② str=new char[strlen(s.Get())+1];
③ strcpy(str,s.Get());
printf("= operator\n");
return *this;
```



[그림 4.17] =연산자의 바른 동작: 먼저 이전 메모리를 해제한다. 그리고 새로운 메모리를 할당한다. 마지막으로 문자열을 새로운 메모리로 복사한다.

이제 복사 생성자를 사용하는 규칙은 다음과 같다.

복사 생성자 규칙(copy constructor rule)

"생성자에서 동적으로 메모리를 할당하는 경우, 반드시 복사 생성자를 만들라. 또한, 반드시 대입 연산자(=)를 오버로딩하여, 레퍼런스를 리턴하도록 하라!"

이제, 복사 생성자 규칙을 완벽히 따르는 완벽한 CStr 클래스의 소스는 아래 [예제 4.3]과 같다. 오버로드된 =연산자가 레퍼런스를 리턴하지 않아도 되지만, 그것은 분명히 비효율적이다. 레퍼런스를 리턴하는 경우와 그렇지 않은 경우의 출력 결과를 비교하라.

[예제 4.4] 복사 생성자의 지원

```
#include <stdio.h>
```

```
#include <string.h>

class CStr {
    char *str;
  public:
    CStr(char *s="") {
        str=new char[strlen(s)+1];
        strcpy(str,s);
        printf("normal constructor\n");
    }
    CStr (const CStr &s) {//copy constructor
        str=new char[strlen(s.Get())+1];
        strcpy(str,s.Get());
        printf("copy constructor\n");
    }
    ~CStr() {
        delete[] str;
        printf("destructor\n");
    }
    char *Get() const { return str; }
    void Print() { printf("%s\n",str); }
    CStr& operator=(CStr &s) {//return reference to object
    //CStr operator=(CStr &s) {//return object itself
        delete[] str;
        str=new char[strlen(s.Get())+1];
        strcpy(str,s.Get());
        printf("= operator\n");
        return *this;
    }
};//class CStr

void f(CStr s) {
    s.Print();
}

void main() {
    CStr s("hello"),
         t("world");
    CStr u=s;

    f(u);
    s=t;
    s.Print();
```

```
}
```

- 출력 결과: =연산자가 레퍼런스를 리턴하는 경우, 즉 소스가 CStr& operator=(CStr &s) 인 경우.

```
normal constructor
normal constructor
copy constructor
copy constructor
hello
destructor
= operator
world
destructor
destructor
destructor
```

- 출력 결과: =연산자가 레퍼런스를 리턴하지 않는 경우, 즉 소스가 CStr operator=(CStr &s) 인 경우.

```
normal constructor
normal constructor
copy constructor
copy constructor
hello
destructor
= operator
copy constructor//복사와 대입의 여분의 오버헤드가 필요하다.
destructor
world
destructor
destructor
destructor
```

위의 예에서 보다시피, 오버로드된 연산자가 객체를 리턴하는 경우, 대부분 레퍼런스를 리턴하는 것이 효율적이다.

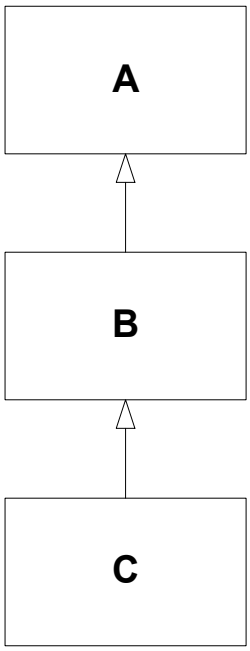
가상함수(virtual function)

<절도비라>
이 절에서는 가상 함수의 원리와 가상 함수를 이용한 설계에 대해서 알아본다. 또한, 베이스 클래스에서 파괴자를 가상으로 선언해야 하는 이유를 살펴보고, 생성자와 소멸자에서는 가상 함수가 호출되지 않음을 살펴본다. 가상 함수는 MFC의 메시지 맵에서 중요하게 사용될 뿐만 아니라, 일반적으로도 빈번하게 사용하므로 원리를 이해하는 것은 아무리 강조해도 지나치지 않다.
</절도비라>

가상 함수를 살펴보기 전에 먼저 '**subtype의 원리**'를 살펴보자. 이 원리는

"원래의 타입(original type)이 있어야 하는 자리에는, 그 타입의 서브타입(subtype)이 올 수 있다."

는 원리이다. 이 원리는 아주 중요하며, 많은 응용을 가진다. 먼저 아래의 [예제 4.5]를 살펴보자. 이 예제는 아래 [그림 4.18]의 계층 구조를 가지는 클래스를 구현한 것이다.



[그림 4.18] B는 A를 상속 받는다. C는 B를 상속 받는다. 프로그래머는 A 클래스의 포인터로 B 혹은 C 클래스 객체의 멤버 함수를 호출할 생각이다. 이러한 응용은 빈번하며 중요하다.

[예제 4.5] 그림 4.18에 해당하는 상속관계의 구현

```
#include <stdio.h>
#include <iostream.h>

class A {
    int i;
public:
    A() {
        i=0;
    }
    void Print() { cout << i << endl; }
};//class A

class B : public A {
    int i;
public:
    B() {
        i=1;
```

```
    }
    void Print() { cout << i << endl; }
};//class B

class C : public B {
    int i;
public:
    C() {
        i=2;
    }
    void Print() { cout << i << endl; }
};//class C

void main() {
    A* ap;
    B b;
    C c;
    ap=&b;//이 문장이 가능한가?
    ap->Print();//가능하다면, 무엇이 호출되는가?
    ap=&c;
    ap->Print();
}//main
```

독자들은 위 프로그램의 출력결과를 예측하였는가? 결과는 아래와 같다.

```
0
0
```

위 결과가 다소 의외일는지 모른다. 하지만, 이것은 이제 설명할 **가상 기능**을 위한 적절한 예이다. 위의 예에서 살펴보아야 할 사항은 두 가지이다.

- A클래스의 포인터에 B클래스의 포인터를 대입하는 것이 가능한가? 즉 아래의 문장은 에러가 아닌가?

  ```
  ap=&b;
  ```

  위의 문장은 에러가 아니다.

- 위의 문장이 가능하다면, ap->Print()는 A의 멤버를 호출하는가? B의 멤버를 호출하는가?

  A의 멤버 함수 Print()를 호출한다.

위의 두 가지 상황에 대한 설명이 매우 불만인 독자는 가상 기능을 이용해 이 문제를 해결할 수 있으므로, 걱정하지 말라. 먼저, 첫 번째 사항은 '**서브타입의 원리(subtype principle)**'이다.

할당문의 왼쪽이 A타입이라면, 할당문의 오른쪽은 당연히 A타입이 와야 한다. 예를 들어,

```
int i=1,j=2;
char c=3;
```

과 같은 변수 선언에서

```
i=c;
```

는 엄격한 의미에서 잘못이다. 명시적인 타입 변환(explicit type conversion)을 해서 아래와 같은 대입문을 사용하는 것이 바람직하다.

```
i=(int)c;
```

이 문제를 클래스에 적용해 보자. 왼쪽에 A*타입이 왔을 때, 오른쪽에는 A*타입이 와야 한다. 또한 **서브 타입의 원리**에 의해, 오른쪽에는 A보다 **아래 타입(subtype)**의 주소가 오는 것이 가능하다. 즉,

"그 타입이 쓰여야 할 자리에는 그 타입의 서브 타입이 올 수 있다."

는 것이다. 그러므로 아래 문장은 컴파일 시간 에러가 발생한다.

```
A a;
B* bp;
bp=&a;//error!
```

왼쪽이 B*이므로 오른쪽에는 B* 혹은 C*는 올 수 있어도, A는 B의 **상위타입(supertype)**이기 때문에 A*는 올 수 없다.

두 번째 사항은 다소 불만이다. 비록 포인터가 A*이지만, 대입된 포인터가 B*였다면, B의 멤버 함수 Print()가 호출되는 것이 자연스럽다. 하지만, 이러

한 정보를 컴파일 시간에 모두 알 수 없다. 컴파일러는 ap->Print()의 바인딩을 위해 단지 ap만을 참조■한다. ap는 A*이므로

<여기서 잠깐>ap는 **심벌 테이블(symbol table)**에 정보가 저장되어 있다. 컴파일러는 함수 주소를 얻기 위해 단지 심벌 테이블만을 참조한다.
</여기서 잠깐>

Print()는 A의 멤버가 호출된다.

이것이 B*라고 알려줄 수 는 없는가? 그럴 수 없다. 그래서 C++은 해결책을 제시하였다. 바로 **실행 시에 멤버 함수의 주소를 결정**하는 것이다. 그렇게 하려면, 함수의 헤더 앞에 virtual을 붙여서 이 함수의 바인딩이 실행 시에 일어나야 할 것임을 알려주어야 한다.

수정된 소스와 결과는 아래 [예제 4.6]과 같다. 수정된 부분은 단지 멤버 함수 Print()에 virtual을 붙인 것뿐이다.

[예제 4.6] 가상함수의 역할

```
#include <stdio.h>
#include <iostream.h>

class A {
    int i;
public:
    A() {
        i=0;
    }
    virtual void Print() { cout << i << endl; }
};//class A

class B : public A {
    int i;
public:
    B() {
        i=1;
    }
    virtual■ void Print() { cout << i << endl; }
```

<여기서 잠깐>virtual 키워드는 베이스 클래스에서 한 번만 지정하면 된다. 그러므로 여기서나 아래 클래스 C에서 명시적으로 virtual을 적을 필요는 없다. 하지만, 이 함수가 virtual인 것을 명확히 하기 위해 - 문서화에 도움이 된다 - 모두 virtual을 붙여 쓰는 것이 좋다.
</여기서 잠깐>

```
};//class B

class C : public B {
    int i;
public:
    C() {
        i=2;
    }
    virtual void Print() { cout << i << endl; }
};//class C

void main() {
    A* ap;
    B b;
    C c;
    ap=&b;
    ap->Print();
    ap=&c;
    ap->Print();
}//main
```

결과는 다음과 같다.

```
1
2
```

실행 결과 외에 차이가 있다면, **조금 늦어진 속도**이다. 실행시간에 가상 함수 테이블을 접근해서 함수의 포인터를 얻어오므로 멤버 함수 호출이 한 스텝 늦어진 것이다.

지금까지의 내용을 한 마디로 요약하면 다음과 같다.

"베이스 클래스의 포인터를 이용하여 하위 클래스(derived class)의 멤버 함수를 호출하기 위해서는, 멤버 함수가 가상이어야 한다. 즉, 하위 클래스에서 오버라이딩(overriding)할 것 같은 함수는 베이스에서 모두 가상으로 선언하라."

베이스 클래스의 멤버함수에서의 가상 함수의 호출

가상 함수의 호출은 그 호출의 시점이 어떤 곳이든 간에, 항상 실행 시에 바인딩 된다는 것을 주의해야 한다. 실행시간 바인딩을 살펴보기 위해 다음과 같은 CBase 클래스를 고려해 보자.

```
class CBase
{
public:
    void Print()
    {
        printf( "pre virtual\n" );
        OverrideMe();
        printf( "post virtual\n" );
    }//Print()

    virtual void OverrideMe()
    {
        printf( "CBase\n" );
    }//OverrideMe()
};//class CBase
```

CBase의 멤버 함수 Print()에서 가상 함수 OverrideMe()를 호출하고 있다. 이를 상속 받은 CDerive클래스가 OverrideMe()함수를 오버라이딩 한 경우, 놀랍게도 베이스 클래스의 OverrideMe()는 CDerived에서 새롭게 정의된 OverrideMe()를 호출한다. 이러한 구조는 강력한 객체 지향 모델을 설계할 수 있도록 한다. 아래의 [예제 4.7]은 베이스 클래스에서 호출한 가상 함수 OverrideMe()가 실행 시간에 바인딩 되어 CBase를 상속 받은 CDerived의 가상 함수인 OverrideMe()가 호출되는 것을 보여준다.

[예제 4.7] 가상 함수를 이용한 객체 지향 모델

```
#include <stdio.h>

class CBase
{
public:
    void Print()
    {
        printf( "pre virtual\n" );
        OverrideMe(); // 가상 함수는 항상 실행시에 바인딩된다.
// 즉, 이 문장은 항상 CBase의 OverrideMe()함수를 호출하는 것은 아니다
        printf( "post virtual\n" );
    }//Print()

    virtual void OverrideMe()
    {
        printf( "CBase\n" );
    }//OverrideMe()
};//class CBase

class CDerive : public CBase
{
public:
    void Draw()
    {
        printf( "draw\n" );
    }//Draw()

    virtual void OverrideMe()
    {
        CBase::OverrideMe(); // 가상 함수의 일반적인 설계 규칙은 베이
//스 클래스의 가상 함수에 기능을 확장하는 것이다.
        printf( "CDerive\n" );
    }//OverrideMe()
};//class CDerive

void main()
{
    CDerive d;

    d.Print();
}//main()
```

실행 결과는 다음과 같다.

```
pre virtual
CBase
CDerive
post virtual
```

가상함수를 사용할 때 한 가지 주의할 점이 있다. 가상 함수라 할지라도 생성자와 소멸자에서 호출된 가상 함수의 바인딩은 컴파일 시간에 일어난다. 이 주제는 아래 절에서 다시 다룬다.

가상 소멸자(virtual destructor)

생성자는 상위 클래스에서 하위 클래스의 순서로 호출된다. 한 조상을 여러 개의 경로를 통하여 상속받는 경우, 한 부모 클래스의 생성자가 여러 번 호출되는 것을 피하기 위하여, 클래스를 가상으로 만들 수 있다.

소멸자는 하위 클래스에서 상위 클래스의 순서로 호출된다. 하지만, 상위 클래스의 포인터가 동적으로 할당된 하위 클래스의 객체를 가리킬 때, 하위 클래스의 소멸자는 호출되지 않는다. 컴파일 시간(compile time)에 상위 클래스의 포인터가 하위 클래스를 가리키고 있다는 것을 어떻게 알 수 있는가? 그러므로 이러한 소멸자의 호출이 실행 시간(run time)에 일어나도록 해야 한다. 그렇다. 소멸자를 가상으로 선언해야 한다.

아래의 [예제 4.8]을 참고하라. 만약 B의 상위 클래스 A에서 소멸자를 virtual로 선언하지 않으면, 어떻게 출력될지를 예상해 보라. 그리고 실행해 보라. 또한 하위 클래스 B에서는 소멸자에 virtual을 붙이지 않아도 됨에 불구하고, 가상 소멸자임을 명확히 하기 위해, virtual을 붙였음에 주목하라.

[예제 4.8] 가상 소멸자

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>

class A {
    char* a;
```

```
public:
    A(char* s) {
        a=new char[strlen(s)+1];
        strcpy(a,s);
    }
    virtual ~A() {//가상 소멸자
        delete[] a;
        printf("Destructor of class A\n");
    }
    virtual void Print() {
        printf("%s\n",a);
    }
};

class B : public A {
    char* b;
public:
    B(char* s,char* t) : A(t) {
        b=new char[strlen(s)+1];
        strcpy(b,s);
    }
    virtual ~B() {
        delete[] b;
        printf("Destructor of class B\n");
    }
    virtual void Print() {
        A::Print();
        printf("%s\n",b);
    }
};

void main() {
    A* pA;
    pA=new B("hello","world");
    pA->Print();
    delete pA;//소멸자가 가상이 아니면 A클래스의 소멸자만이 호출된다.
}
```

실행 결과는 아래와 같다.

```
world
hello
```

```
Destructor of class B
Destructor of class A
```

<여기서 잠깐>A와 B의 소멸자가 가상이 아닌 경우의 실행결과는 다음과 같다.

```
world
hello
Destructor of class A
```

</여기서 잠깐>

위의 경우 소멸자를 가상으로 선언하지 않으면 문제는 심각하다. 생성자에서 동적으로 메모리를 할당하고 있기 때문에, 소멸자가 가상으로 선언되지 않으면, 메모리 샘(leak)이 발생할 것이다. 규칙은 다음과 같다.

"상위 클래스로 사용될 것 같은 클래스의 생성자에서 동적으로 메모리를 할당하는 경우, 소멸자를 가상으로 선언하라."

생성자, 소멸자에서의 가상함수 호출

가상함수를 사용할 때 무심코 흘려버릴 수 있는 주의 사항이 있다. 그것은 생성자 소멸자에서의 가상 함수 호출이다. 아래의 [예제 4.9]는 CBase의 생성자, 소멸자 및 일반 함수에서 가상 함수 DoVirtual()을 호출하고 있다. CBase를 상속받은 CDerive 객체를 만들면 출력결과는 어떻게 될까?

[예제 4.9] 생성자,소멸자에서의 가상 함수의 호출

```
#include <stdio.h>

class CBase
{
public:
    CBase()
    {
        DoVirtual(); // 가상함수를 호출할 것인가? No!
    }//CBase()

    virtual ~CBase()
    {
```

```
        DoVirtual(); // 가상함수를 호출할 것인가? No!
    }//~CBase()

    void DoIt()
    {
        DoVirtual(); // 가상함수를 호출할 것인가? Yes!
    }//DoIt()

    virtual void DoVirtual()
    {
        printf( "CBase\n" );
    }//DoVirtual()
};//class CBase

class CDerive : public CBase
{
public:
    virtual void DoVirtual()
    {
        printf( "CDerive\n" );
    }//DoVirtual()
};//class CDerive

void main()
{
    CDerive b;

    b.DoIt();
}//main()
```

출력 결과는 다음과 같다.

```
CBase
CDerive
CBase
```

생성자와 소멸자에서는 가상함수를 호출하지 않는다. 마지막으로 다음 문제를 고려해 보자.

"생성자에서 호출한 함수가 내부에서 가상 함수를 호출하는 경우는 어떻게 동작할까?"

답은 부당한 가상 함수의 호출로 프로그램이 죽는다는 것이다. 독자들이 꼭 재현해 보기 바란다.

RTTI(Run-Time Type Information)

<절도비라>

RTTI는 클래스의 문자열 이름으로 객체를 만들거나, 만들어진 객체의 타입을 얻고 서로 비교하기 위한 방법이다. 이 절에서는 RTTI를 살펴보고 이를 직접 구현한다. 그리고 최종적으로 매크로를 이용해서 RTTI 코드를 재사용 가능하게 만든다. 또한, MFC에서 RTTI가 어떻게 유용하게 사용될지에 대해서도 간단하게 살펴본다.

</절도비라>

RTTI□란 실행 시간에 객체의 타입 정보를 얻는 C++ 진보된 방식이다. 표

<저자한마디>
MFC의 소스를 분석하면서 RTTI의 동작 원리를 이해했을 때의 놀라움을 잊을 수 없다. 사실 필자는 2000년부터 게임 개발에만 몰두하고 있어서 이제 MFC 코드를 작성할 일이 거의 없다. 하지만, MFC에서 사용한 많은 기교들이 다른 라이브러리에서도 거의 동일하게 구현되고 있으므로 많은 도움이 되었다. 대중에게 공개되는 프로그램을 5년이 넘게 개발하면서 느낀 사실은 프로그램은 - 더군다나 게임 프로그램은 - 어떠한 상황에서도 절대로 죽지 않아야 한다는 것이고, 프로그래밍에 대한 지식이 조금씩 늘수록 모르는 것은 더 빨리 늘어간다는 것이며, 이것은 평생 계속될 것이다.
</저자한마디>

준 위원회에서는 sizeof처럼 typeof를 넣자는 제안도 있었다. 필자는 후에 typeof가 C++에 추가될 것이라 생각한다. 만약 typeof라는 키워드가 존재한다면, 파라미터로 전달된 CObject 클래스를 상속받는 pObj타입의 객체를 스택이나 힙(heap)에 만들기 위한 Test()함수를 다음과 같이 작성할 수 있다.

```
void Test(CObject* pObj)
{
    typeof(pObj)    objTemp;
    ...
    CObject* pSecondObj = new typeof(pObj);
    ...
```

```
}//Test()
```

와우! 얼마나 놀라운 방식인가! 하지만, 현재의 C++표준은 위와 같은 방식을 지원하지 않는다. 하지만, MFC등 대부분의 잘 설계된 라이브러리에서는 typeof의 역할을 돕도록 - 즉 실행시간에 결정되는 타입의 객체를 생성하도록 - 클래스를 설계하고 있으며, 이것은 이제부터 살펴볼 내용이다. 이제부터 살펴보는 내용은 무척 중요하다. 왜냐하면, 이러한 방식은 업계의 표준으로 대부분의 라이브러리가 사용하기 때문이다□.

<저자 한마디>RTTI를 사용하는 필자가 접해본 라이브러리의 예로서, Boost 라이브러리, 게임 브리오 엔진, 넷이머스 엔진 등이 있다.</저자 한마디>

왜 이것이 필요한가?

사용자가 클래스의 이름을 입력하면, 클래스의 골격을 대충 만들어 주는 코드 자동 생성기(automatic code generator)를 생각해 보자.

```
Enter Class Name: MyClass↵
```

위의 예에서 사용자가 MyClass를 입력하면, 입력을 받은 프로그램은 MyClass란 클래스의 소스 코드 골격을 생성한다. 그리고 이 소스 코드에는 자신을 동적으로 만들기 위해 메모리를 할당하는 루틴이 포함되어 있을 것이다.

실행시에 메모리 할당이 실패했을 때, 아래와 같은 에러 메시지를 출력하는 부분을 구현하고 싶다고 하자.

"MyClass"를 만드는데 실패했습니다

어떻게 이것을 구현할 것인가? 이것은 **실행시 클래스에 관한 정보(run-time type information)**를 얻어야만 가능하다. 즉, 클래스의 이름을 문자열로 얻거나, 클래스의 크기 등을 정수로 얻을 수 있는 기능이 있어야 한다. 구현할 기능은 다음을 지원한다.

- 실행시에 알려지지 않은 클래스의 이름과 크기를 얻을 수 있어야 한다.

- 실행시에 알려지지 않은 클래스를 동적으로 생성할 수 있어야 한다.

먼저 클래스의 이름을 얻는 간단한 기능부터 구현해 보자.

실행시에 객체의 클래스 이름 얻기

아래의 예에서는 CObject 클래스가 이미 만들어져 있고, 사용자가 이 클래스를 상속받아 CMyClass를 만드는 경우, CMyClass의 객체가 만들어졌을 때, 이 객체의 이름을 출력하는 예제이다.

먼저 베이스 클래스인 CObject는 자신을 상속받은 하위 클래스의 이름을 얻기 위한 멤버 함수 GetClassName()을 가져야 한다. 또한 이것은 실행 시에 바르게 하위 클래스를 참조해야하므로, 가상 함수로 선언되어야 한다.

CObject의 소스는 아래와 같다.

```
class CObject {
public:
    virtual char* GetClassName() const {
        return NULL;
    }
};//class CObject
```

GetClassName()이 어떠한 값도 변경시키지 않는다는 것(즉 상수함수임을)을 명확히 하기 위해, 끝에 const를 추가하였다. 또한 CObject 자신은 단지 베이스 클래스로만 사용될 것이라 가정하였으므로, 자신의 이름은 NULL을 리턴하고 있다.

사용자가 CObject를 베이스 클래스로 하면서, RTTI를 지원하는 자신의 클래스 CMyClass를 만들기 위해서는 반드시 구현해야할 클래스의 멤버가 필요하다. 그것은 다음의 두가지 이다.

① 자신의 클래스의 이름을 저장하는 정적 멤버 변수
② 그 이름을 리턴하는 GetClassName()을 오버라이드(override)하는 것

전체 소스는 [예제 4.10]과 같다.

[예제 4.10] RTTI를 지원하기 위해 정적 멤버 함수의 구현

```
//RTTI(run time type information) example
#include <iostream.h>

class CObject {
public:
    virtual char* GetClassName() const {
        return NULL;
    }
};//class CObject

class CMyClass : public CObject {
public:
    static char lpszClassName[];//클래스의 이름은 메모리에 한 번만 할
                                //당하면 된다.
    virtual char* GetClassName() const {//베이스 클래스의 GetClassName을
                                        //오버라이드한다.
        return lpszClassName;//정적 멤버 변수를 리턴한다.
    }
};//class CMyClass

char CMyClass::lpszClassName[]="CMyClass";//정적 변수는 반드시 클래스의
                                          //외부에서 초기화되어야 한다.

void main() {
    CObject *p;

    p=new CMyClass;//subtype principle에 의해 하위 클래스를 가리키기 위
                   //해 상위 클래스의 포인터를 사용하는 것이 가능하다.
    cout << p->GetClassName() << endl;
      //동적으로 만들어진 객체의 이름을 구할 수 있다!
    delete p;
}//main
```

[예제 4.10]을 차근히 살펴보자.

```
class CMyClass : public CObject {
```

먼저 CMyClass를 사용자가 만들 때 반드시 CObject를 상속받도록 만들어야 한다. 만약 코드 자동 생성기가 이 부분을 생성한다면, 무조건 정해진 어떤 클래스를 상속받는 CMyClass를 만들 것이므로, 걱정하지 않아도 좋다(MFC의 어플리케이션 마법사(Application Wizard)는 사용자 클래스의 베이스로

CObject를 사용한다).

```
static char lpszClassName[];
```

그리고 클래스의 이름을 저장할 정적 변수(static variable)가 클래스의 멤버로 추가되어야 한다. 또한 이 변수는 클래스의 외부에서 클래스의 이름과 같은 문자열 "CMyClass"로 초기화되어야 한다.

```
char CMyClass::lpszClassName[]="CMyClass";
```

그리고 이름을 얻는 베이스 클래스의 GetClassName()을 오버라이드한다.

```
virtual char* GetClassName() const {//베이스 클래스의 GetClassName을
                                    //오버라이드한다.
    return lpszClassName;//정적 멤버 변수를 리턴한다.
}
```

CMyClass의 GetClassName()은 단순히 자신의 정적 변수 lpszClassName을 리턴한다.

자 이제 main()부분의 사용법을 살펴보자. 먼저 CMyClass를 동적으로 만들기 위해 CObject* 변수 p를 선언한다.

```
void main() {
    CObject *p;

    p=new CMyClass;//subtype principle에 의해 하위 클래스를 가리키기 위
                   //해 상위 클래스의 포인터를 사용하는 것이 가능하다.
    cout << p->GetClassName() << endl;
      //동적으로 만들어진 객체의 이름을 구할 수 있다!
    delete p;
}//main
```

p가 새로 만들어진 CMyClass의 객체를 가리킨다. p->GetClassName()은 가상 함수이므로, CObject::GetClassName()이 호출되지 않고, CMyClass::GetClassName()을 호출하므로 출력 결과는

```
CMyClass
```

이다.

매크로의 사용

이런 작업을 빈번하게 해야 한다면, 다음과 같은 두 개의 매크로를 사용하여 일을 간단히 할 수 있다.

```
#define DECLARE_CLASSNAME(s) static char lpszClassName[]
#define IMPLEMENT_CLASSNAME(s) char s##::lpszClassName[]=(#s)
```

위의 매크로 함수는 토큰을 결합하고 파라미터를 스트링으로 만들기 위해 각각 ##와 # 연산자를 사용한다. 이 두개의 매크로는 단순한 정적 변수 선언 문장과 정적 변수의 초기화를 대신한다.

매크로를 사용한 소스를 [예제 4.11]에 다시 리스트하였다.

[예제 4.11] 매크로의 사용

```
//RTTI(run time type information) example
#include <iostream.h>

#define DECLARE_CLASSNAME(s) static char lpszClassName[]
#define IMPLEMENT_CLASSNAME(s) char s##::lpszClassName[]=(#s)

class CObject {
public:
    virtual char* GetClassName() const {
        return NULL;
    }
};//class CObject

class CMyClass : public CObject {
public:
    //static char lpszClassName[];
    DECLARE_CLASSNAME(CMyClass);
    virtual char* GetClassName() const {
        return lpszClassName;
    }
};//class CMyClass
```

```
//char CMyClass::lpszClassName[]="CMyClass";
IMPLEMENT_CLASSNAME(CMyClass);

void main() {
    CObject *p;

    p=new CMyClass;
    cout << p->GetClassName() << endl;
    delete p;
}//main
```

무엇이 좋아졌는가?

위의 소스에서 main()의 내부를 제외한 다른 모든 부분은 코드 자동생성기가 자동으로 생성한 코드라는 점에 유의해야 한다. 만약 사용자가 코드 생성기에게 동적으로 클래스의 이름을 얻는 코드를 생성하도록 코드 생성기의 옵션(option)을 설정했다고 하자. 그러면 코드 생성기는 DECLARE_CLASSNAME과 IMPLEMENT_CLASSNAME 매크로를 위에서처럼 추가할 것이다. 사용자가 이 옵션을 선택하지 않았다고 하자. 그러면 코드 자동생성기는 단순히 DECLARE_CLASSNAME과 IMPLEMENT_CLASSNAME을 코드에 포함시키지 않을 것이다.

동적 생성을 지원하려면?

이제 실제로 동적 생성을 지원하는 클래스를 만들기 위해서, 코드 자동 생성기가 해 주어야 하는 작업을 구체적으로 살펴보자.

실제 MFC 프로그램에서는 위에서 살펴본 lpszClassName을 **CRuntimeClass 구조체**가 대신한다. 이 구조체는 ①클래스 이름과, ②객체의 크기에 대한 데이터 멤버를 가지며, 대상 클래스를 동적으로 생성하기 위해, ③함수 포인터와 ④CreateObject() 멤버 함수를 가진다.

CRuntimeClass 구조체는 아래와 같다.

```
struct CRuntimeClass {
    char m_lpszClassName[21];
```

```
    int m_nObjectSize;
    CObject* (*pfnCreateObject)();

    CObject* CreateObject();
};//struct CRunTimeClass

CObject* CRuntimeClass::CreateObject() {
    return (*pfnCreateObject)();//함수 포인터를 이용하여
                                //간접적으로 함수를 호출한다.
}//CRuntimeClass::CreateObject()
```

CreateObject()는 단순히 자신의 데이터 멤버인 함수포인터를 이용하여 무엇일지 모르는 - 사용자에 의해 정해진 객체를 생성하는 - 함수를 호출하고 있다.

이제 준비 작업은 끝났다. 이제 남은 문제는 다음과 같다.

사용자가 정의하는 클래스가 자신을 동적으로 생성할 수 있도록 어떻게 코드를 준비하는가?

아마도 사용자가 정의한 클래스의 이름이 CAlpha라면 CAlpha의 멤버 함수 중에 아래와 같은 문장이 반드시 있어야 할 것이다.

```
new CAlpha;
```

하지만 이러한 동적 생성 문장은 클래스의 이름에 상관없이 일관된 방법으로 호출되어야 할 것이다.

객체를 생성하는 함수가 항상 존재해야 하므로 해답은 정적 멤버 함수(static member function)를 사용하는 것이다. 즉, 클래스의 멤버 함수로 자신을 동적으로 생성하는 CreateObject()를 추가한다. 그리고, 정적 CRuntimeClass 객체(예를 들면 classCObject)를 데이터 멤버로 추가한다. 마지막으로, classCObject의 멤버 pfnCreateObject를 실제 자신을 생성하는 함수의 주소인, &CreateObject로 초기화하는 것이다.

CreateObject()의 주소를 얻기 위해, '**멤버 함수의 주소를 얻는 방법**'을 사용하지 않은 것에 주목하라. 즉, CRuntimeClass의 함수 포인터 멤버가 다음과 같이 선언되어서는 안된다.

```
COBject* (CObject::*pfnCreateObject)();
```

그 이유는 CreateObject() 역시 정적 멤버 함수이기 때문이다.
이제 CObject의 실제 소스를 살펴보자.

```
class CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const { return NULL; }
      //파생 클래스에서 반드시 구현할 필요가 없으므로
      //순수가상함수가 아니다.
    static CRuntimeClass classCObject;
    virtual ~CObject(){}
protected:
    CObject(){ printf("CObject constructor\n"); }
};//class CObject

CRuntimeClass CObject::classCObject={
    "CObject",sizeof(CObject),NULL
};
```

CObject는 CRuntimeClass타입의 정적 멤버 classCObject를 리턴하는 GetRuntimeClass()를 가진다. 위의 경우 CObject는 단순히 베이스 클래스로만 사용될 것이므로, 단순히 NULL을 리턴한다.

```
virtual CRuntimeClass* GetRuntimeClass() const { return NULL; }
```

CRuntimeClass타입의 정적 멤버

```
static CRuntimeClass classCObject;
```

는 외부에서 자신을 동적으로 생성하기 위해 사용한다. 이 변수는 정적 변수이므로 클래스 외부에서 적당히 초기화된다. CObject는 단순히 베이스 클래스로만 사용될 것이므로, 세 번째 값은 NULL로 초기화하였다.

```
CRuntimeClass CObject::classCObject={

    "CObject",sizeof(CObject),NULL
};
```

CRuntimeClass의 사용은 CObject를 상속받은 다른 클래스를 살펴보면, 확실해 진다. 아래의 CAlpha는 베이스 클래스로 CObject를 사용한다.

```
class CAlpha : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCAlpha;//정적 멤버 객체의 주소를 리턴한다.
    }
    static CRuntimeClass classCAlpha;
      //DECLARE_DYNAMIC(CAlpha)
    static CObject* CreateObject();
      //DECLARE_DYNCREATE(CAlpha)
protected:
    CAlpha() { printf("CAlpha constructor\n"); }
};//class CAlpha

CRuntimeClass CAlpha::classCAlpha = {//IMPLEMENT_DYNAMIC(CAlpha)
    "CAlpha", sizeof(CAlpha), CAlpha::CreateObject
};
```

CAlpha의 GetRuntimeClass()는 정적 멤버의 주소를 리턴한다. 정적 멤버 classCAlpha는 다음과 같이 초기화된다.

```
        CRuntimeClass CAlpha::classCAlpha = {//IMPLEMENT_DYNAMIC(CAlpha)
            "CAlpha", sizeof(CAlpha), CAlpha::CreateObject
        };
```

구조체를 초기화하는 세 번째 값에 주목하라. 세 번째 값은 CRuntimeClass의 세 번째 멤버인 함수 포인터를 CAlpha를 동적으로 생성하는 CreateObject()의 주소로 초기화한다. CreatObject의 소스는 아래와 같다.

```
        CObject* CAlpha::CreateObject() {
            return new CAlpha;//자신을 동적으로 생성한다.
        }
```

어떻게 동적 생성이 가능한가?

선언되지도 않았고, 심지어 이름이 정해지지 않았을지도 모르는 객체를 동적으로 생성하는 문장을 자동으로 생성하는 것이 어떻게 가능한가? 이것은

정적 멤버(static member)의 특성에 있다. 정적 멤버는 객체의 생성과는 상관없이 항상 메모리에 존재한다는 것을 상기하자. 그러므로, 위의 예에서 CAlpha의 정적 멤버인 classCAlpha와 CreateObject()는 CAlpha의 객체가 하나도 만들어지지 않았지만 존재한다□.

<여기서 잠깐>멤버 함수는 - 그것이 정적이든 아니든 - 클래스와 상관없이 항상 존재한다. 하지만, 일반 멤버 함수는 묵시적으로 전달되는 첫 번째 파라미터 this의 값을 결정해야 하므로, 객체가 없으면 호출할 수 없다. 하지만, 정적 멤버 함수는 그것의 선언이 클래스 내부일 뿐, 일반 함수와 같다. 즉 this를 파라미터로 받지 않는다.</여기서 잠깐>

그러므로, 이미 존재하는 classCAlpha를 이용하여, CreateObject()를 호출함으로써, 동적 생성이 가능하게 된 것이다. 여기서 주의할 것은 필요한 포인터는 CreateObject()가 리턴하는 포인터이지 classCAlpha의 포인터가 아니라는 것이다(classCAlpha의 포인터는 항상 일정하다).

아래의 예를 참고하라. 아래의 예는 CAlpha와 CBeta를 동적으로 생성하는 예를 보여주고 있다. 새로 사용된 RUNTIME_CLASS 매크로에 주목하라.

매크로 사용 전 소스

[예제 4.12] 매크로 사용 전 소스

```
//RTTI(run time type information) example
#include <stdio.h>
#include <iostream.h>

#define RUNTIME_CLASS(class_name) (&class_name::class##class_name)

class CObject;

//{{struct CRuntimeClass-----------------------------------
struct CRuntimeClass {
    char m_lpszClassName[21];
    int m_nObjectSize;
    CObject* (*pfnCreateObject)();

    CObject* CreateObject();
};//struct CRunTimeClass
```

```
CObject* CRuntimeClass::CreateObject() {
    return (*pfnCreateObject)();//함수 포인터를 이용하여
                                //간접적으로 함수를 호출한다.
}//CRuntimeClass::CreateObject()
//}}struct CRuntimeClass----------------------------------

//{{class CObject-----------------------------------------
class CObject {
    //CObject는 순수 가상함수를 포함하지 않으므로
    //추상 클래스가 아니다. 하지만, 생성자가 protected로 설정되었으므로,
    //CObject객체를 생성할 수 없다.
public:
    virtual CRuntimeClass* GetRuntimeClass() const { return NULL; }
      //파생 클래스에서 반드시 구현할 필요가 없으므로
      //순수가상함수가 아니다.
    static CRuntimeClass classCObject;
      //DECLARE_DYNAMIC(CObject)
    virtual ~CObject(){}
protected:
    CObject(){ printf("CObject constructor\n"); }
};//class CObject

CRuntimeClass CObject::classCObject={//IMPLEMENT_DYNAMIC(CObjec
    "CObject",sizeof(CObject),NULL
};
//}}class CObject-----------------------------------------

//{{class CAlpha------------------------------------------
class CAlpha : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCAlpha;
    }
    static CRuntimeClass classCAlpha;
      //DECLARE_DYNAMIC(CAlpha)
    static CObject* CreateObject();
      //DECLARE_DYNCREATE(CAlpha)
protected:
    CAlpha() { printf("CAlpha constructor\n"); }
};//class CAlpha

CRuntimeClass CAlpha::classCAlpha = {//IMPLEMENT_DYNAMIC(CAlpha)
```

```
    "CAlpha", sizeof(CAlpha), CAlpha::CreateObject
};

CObject* CAlpha::CreateObject() {//IMPLEMENT_DYNCREATE(CAlpha)
    return new CAlpha;
}
//}}class CAlpha----------------------------------------

//{{class CBeta----------------------------------------
class CBeta : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCBeta;
    }
    static CRuntimeClass classCBeta;
      //DECLARE_DYNAMIC(CBeta)
    static CObject* CreateObject();
      //DECLARE_DYNCREATE(CBeta)
protected:
    CBeta() { printf("CBeta constructor\n"); }
};//class CBeta

CRuntimeClass CBeta::classCBeta = {//IMPLEMENT_DYNAMIC(CBeta)
    "CBeta", sizeof(CBeta), CBeta::CreateObject
};

CObject* CBeta::CreateObject() {//IMPLEMENT_DYNCREATE(CBeta)
    return new CBeta;
}
//}}class CBeta----------------------------------------

void main() {
    //Create CAlpha
    CRuntimeClass* pRTCAlpha=RUNTIME_CLASS(CAlpha);
    CObject* pObj1;

    pObj1=pRTCAlpha->CreateObject();
      //struct RuntimeClass의 CreateObject()가 호출되지만
      //IMPLEMENT_DYNCREATE 매크로에 의해 CAlpha의 &OnCreate()가
      //대입되어 있으므로, 결국은 CAlpha의 OnCreate()가 호출되어
      //동적으로 객체를 생성하게 된다.
      //문제를 푸는 열쇠는 바로 '함수 포인터'이다.
    printf("CAlpha class=%s\n",
```

```
        pObj1->GetRuntimeClass()->m_lpszClassName);

    //Create CBeta
    CRuntimeClass* pRTCBeta=RUNTIME_CLASS(CBeta);
    CObject* pObj2;

    pObj2=pRTCBeta->CreateObject();
    printf("CBeta class=%s\n",
        pObj2->GetRuntimeClass()->m_lpszClassName);
      //일관된 방법으로 클래스를 만들고 해제하는 것에 주목하라.

    delete pObj1;
    delete pObj2;
}//main
```

실행 결과는 다음과 같다.

```
CObject constructor
CAlpha constructor
CAlpha class=CAlpha
CObject constructor
CBeta constructor
CBeta class=CBeta
```

[예제 4.12]에서 main()을 보면 RUNTIME_CLASS(CAlpha)를 이용하여, CRuntimeClass 객체의 시작 주소를 얻는다.

```
CRuntimeClass* pRTCAlpha=RUNTIME_CLASS(CAlpha);
```

그리고, CRuntimeClass의 멤버 함수 CreateObject()를 호출하여 CAlpha를 만든 다음, CObject의 포인터 pObj1이 이를 가리키도록 하는 사실에 주목하라.

```
CObject* pObj1;

pObj1=pRTCAlpha->CreateObject();
```

사용자는 이제 pObj1을 통하여 자유롭게 CAlpha의 멤버들을 접근할 수 있다. 그리고 객체를 해제하기 위해서 다음과 같이 pRTCAlpah를 직접 접근해서는 안되다.

```
delete pRTCAlpha;
```

대신에 다음과 같이 pObj1을 접근해야 한다.

```
delete pObj1;
```

CObject와 CRuntimeClass가 미리 만들어져 있다면, 사용자가 입력한 CAlpha에 대해 이 클래스를 동적으로 생성하는 코드가 얼마나 간단해 졌는가에 주목하라. 이러한 작업은 매크로를 사용하여 더 간단히 줄일 수 있다.

멤버를 선언하기 위해,

```
DECLARE_
```

매크로를 사용하며, 구현하기 위해,

```
IMPLEMENT_
```

매크로를 사용하도록 하자. 그러므로 다음과 같은 매크로를 설계할 수 있다.

```
#define DECLARE_DYNAMIC(class_name)   static\
               CRuntimeClass class##class_name;
#define IMPLEMENT_DYNAMIC(class_name) CRuntimeClass\
                  class_name::class##class_name = {\
                      (#class_name),\
                      sizeof(class_name),\
                  class_name::CreateObject };
#define DECLARE_DYNCREATE(class_name) static CObject* CreateObject();
#define IMPLEMENT_DYNCREATE(class_name) CObject*\
                       class_name::CreateObject() {\
                  return new class_name; }
```

동적 타입 정보에 관한 매크로는 _DYNAMIC으로 끝난다. 동적 생성에 관한 매크로는 _DYNCREATE로 끝난다. 이것은 실제 MFC 코드 생성기가 사용하는 매크로이다. 우리는 MFC에서 이 매크로의 동작을 10장에서 살펴 볼 것이다.

매크로 사용 후 소스

[예제 4.13] 매크로 사용 후 소스

```
//RTTI(run time type information) example
#include <stdio.h>
#include <iostream.h>

//{{RTTI macros-----------------------------------------
#define RUNTIME_CLASS(class_name) (&class_name::class##class_name)
#define DECLARE_DYNAMIC(class_name)     static\
                CRuntimeClass class##class_name;
#define IMPLEMENT_DYNAMIC(class_name) CRuntimeClass\
                    class_name::class##class_name = {\
                        (#class_name),\
                        sizeof(class_name),\
                    class_name::CreateObject };
#define DECLARE_DYNCREATE(class_name) static CObject* CreateObject();
#define IMPLEMENT_DYNCREATE(class_name) CObject*\
                         class_name::CreateObject() {\
                    return new class_name; }
//}}RTTI macros-----------------------------------------

class CObject;//forward declaration of CObject

//{{struct CRuntimeClass-----------------------------------
struct CRuntimeClass {
    char m_lpszClassName[21];
    int m_nObjectSize;
    CObject* (*pfnCreateObject)();

    CObject* CreateObject();
};//struct CRunTimeClass

CObject* CRuntimeClass::CreateObject() {
    return (*pfnCreateObject)();//함수 포인터를 이용하여
                                //간접적으로 함수를 호출한다.
}//CRuntimeClass::CreateObject()
//}}struct CRuntimeClass-----------------------------------

//{{class CObject-------------------------------------------
class CObject {
```

```
    //CObject는 순수 가상함수를 포함하지 않으므로
    //추상 클래스가 아니다. 하지만, 생성자가 protected로 설정되었으므로,
    //CObject객체를 생성할 수 없다.
public:
    virtual CRuntimeClass* GetRuntimeClass() const { return NULL; }
      //파생 클래스에서 반드시 구현할 필요가 없으므로
      //순수가상함수가 아니다.
    DECLARE_DYNAMIC(CObject)
      //static CRuntimeClass classCObject;
    virtual ~CObject(){}
protected:
    CObject(){ printf("CObject constructor\n"); }
};//class CObject

CRuntimeClass CObject::classCObject={
    "CObject",sizeof(CObject),NULL
};
//}}class CObject----------------------------------------

//{{class CAlpha----------------------------------------
class CAlpha : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCAlpha;
    }
    DECLARE_DYNAMIC(CAlpha)
      //static CRuntimeClass classCAlpha;
    DECLARE_DYNCREATE(CAlpha)
      //static CObject* CreateObject();
protected:
    CAlpha() { printf("CAlpha constructor\n"); }
};//class CAlpha

IMPLEMENT_DYNAMIC(CAlpha)
  //CRuntimeClass CAlpha::classCAlpha = {
  //    "CAlpha", sizeof(CAlpha), CAlpha::CreateObject
  //};

IMPLEMENT_DYNCREATE(CAlpha)
  //CObject* CAlpha::CreateObject() {
  //    return new CAlpha;
  //}
//}}class CAlpha----------------------------------------
```

```
//{{class CBeta-----------------------------------------
class CBeta : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCBeta;
    }
    DECLARE_DYNAMIC(CBeta)
      //static CRuntimeClass classCBeta;
    DECLARE_DYNCREATE(CBeta)
      //static CObject* CreateObject();
protected:
    CBeta() { printf("CBeta constructor\n"); }
};//class CBeta

IMPLEMENT_DYNAMIC(CBeta)
  //CRuntimeClass CBeta::classCBeta = {
  //    "CBeta", sizeof(CBeta), CBeta::CreateObject
  //};

IMPLEMENT_DYNCREATE(CBeta)
  //CObject* CBeta::CreateObject() {
  //    return new CBeta;
  //}
//}}class CBeta-----------------------------------------

void main() {
    //Create CAlpha
    CRuntimeClass* pRTCAlpha=RUNTIME_CLASS(CAlpha);
    CObject* pObj1;

    pObj1=pRTCAlpha->CreateObject();
      //struct RuntimeClass의 CreateObject()가 호출되지만
      //IMPLEMENT_DYNCREATE 매크로에 의해 CAlpha의 &OnCreate()가
      //대입되어 있으므로, 결국은 CAlpha의 OnCreate()가 호출되어
      //동적으로 객체를 생성하게 된다.
      //문제를 푸는 열쇠는 바로 '함수 포인터'이다.
    printf("CAlpha class=%s\n",
         pObj1->GetRuntimeClass()->m_lpszClassName);

    //Create CBeta
    CRuntimeClass* pRTCBeta=RUNTIME_CLASS(CBeta);
    CObject* pObj2;
```

```
    pObj2=pRTCBeta->CreateObject();
    printf("CBeta class=%s\n",
         pObj2->GetRuntimeClass()->m_lpszClassName);
      //일관된 방법으로 클래스를 만들고 해제하는 것에 주목하라.

    delete pObj1;//delete pRTCAlpha는 왜 (논리적) 에러인가?
    delete pObj2;
}//main
```

위의 소스는 독자들의 편의를 위해 매크로의 원래 부분을 주석으로 처리했다. 실제 매크로의 강력함을 느끼려면 이 부분을 삭제하고 보면 확실해 진다. CRuntimeClass 및 CObject가 이미 만들어져 있다고(afx.h에) 가정하며, 주석을 지우고 소스를 다시 적어 보자.

매크로와 CRuntimeCass 및 CObject가 "afx.h"에 미리 만들어져 있다고 가정했을 때의 소스 코드

[예제 4.14] afx.h를 가정한 소스

```
#include <stdio.h>
#include <iostream.h>
#include "afx.h"

//{{class CAlpha-----------------------------------------
class CAlpha : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCAlpha;
    }
    DECLARE_DYNAMIC(CAlpha)
    DECLARE_DYNCREATE(CAlpha)
protected:
    CAlpha() { printf("CAlpha constructor\n"); }
};//class CAlpha

IMPLEMENT_DYNAMIC(CAlpha)
IMPLEMENT_DYNCREATE(CAlpha)
//}}class CAlpha-----------------------------------------
```

```
//{{class CBeta-----------------------------------------
class CBeta : public CObject {
public:
    virtual CRuntimeClass* GetRuntimeClass() const {
        return &classCBeta;
    }
    DECLARE_DYNAMIC(CBeta)
    DECLARE_DYNCREATE(CBeta)
protected:
    CBeta() { printf("CBeta constructor\n"); }
};//class CBeta

IMPLEMENT_DYNAMIC(CBeta)
IMPLEMENT_DYNCREATE(CBeta)
//}}class CBeta-----------------------------------------

void main() {
    //Create CAlpha
    CRuntimeClass* pRTCAlpha=RUNTIME_CLASS(CAlpha);
    CObject* pObj1;

    pObj1=pRTCAlpha->CreateObject();
    printf("CAlpha class=%s\n",
         pObj1->GetRuntimeClass()->m_lpszClassName);

    //Create CBeta
    CRuntimeClass* pRTCBeta=RUNTIME_CLASS(CBeta);
    CObject* pObj2;

    pObj2=pRTCBeta->CreateObject();
    printf("CBeta class=%s\n",
         pObj2->GetRuntimeClass()->m_lpszClassName);

    delete pObj1;
    delete pObj2;
}//main
```

독자들은 매크로에 관한 설명을 전혀 듣지 않고, 위의 [예제 4.14]를 접했을 때의 황당함에 대해 생각해 본 적이 있는가? 필자가 MFC의 소스를 처음 접했을 때 소스는 완전한 암호문이었다. 하지만, 메시지 맵(message map)에 관한 매크로와, 동적 생성에 관한 매크로에 대해 완전히 파악하고 난 뒤의 소스

는 간결함 그 자체였다!

요약

6장부터 본격적인 MFC의 소스 분석을 시작할 것이다. 그리고 10장에서 MFC의 RTTI를 직접 구현해 볼 것인데, 매크로의 원리를 다시 설명하지는 않을 것이므로 이 장의 내용을 이해하지 못한 독자들은 처음부터 다시 보기 바란다. 5장 이후의 내용에서 C++에 관한 주제는 더 이상 설명하지 않는다.

- **복사 생성자**는 이미 생성된 객체가 복사될 때 호출되는 특별한 생성자이다. 클래스 내부에서 동적 메모리 할당을 사용하는 경우 복사 생성자를 만들고, operator=()을 작성하며, 상속을 위해 소멸자를 가상으로 선언할 필요가 있다.
- **가상함수**는 함수 호출에 대한 주소 결정이 실행 시간에 일어난다. 이것을 늦은 바인딩이라고 하는데, 가상함수를 사용하면 이미 디자인된 프로그램의 세부 동작을 수정하거나 향상시키는 것이 가능하며 이것이 주 목적이다.
- C++의 RTTI지원을 받지 않는 **MFC의 RTTI**에 대해서 살펴보았다. 비주얼 C++ 6에 포함된 MFC 버전 4.2는 C++의 RTTI가 지원되기 전부터 존재했다.

[문서의 끝]