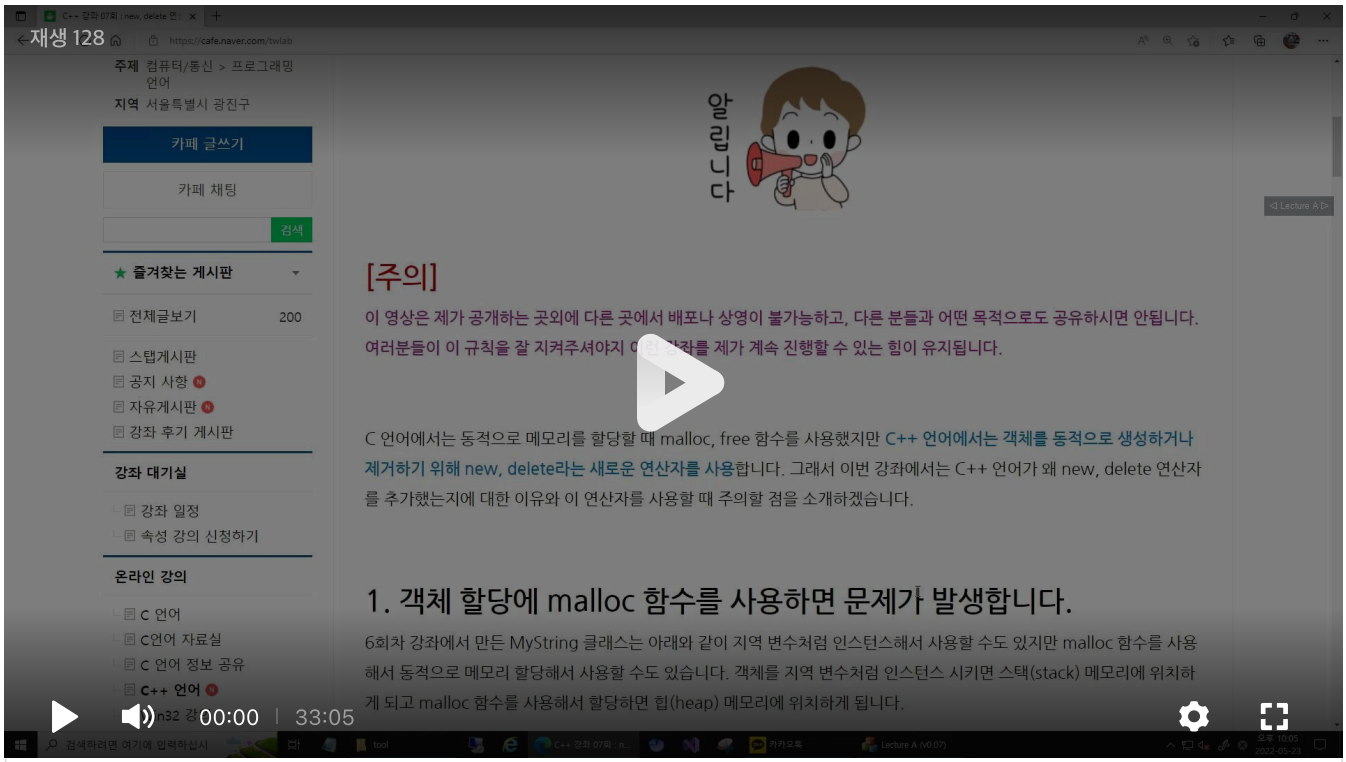
C++ 강좌 07회 : new, delete 연산자

김성엽 카페매니저 + 구독 1:1 채팅
2022.05.23. 14:53 조회 423

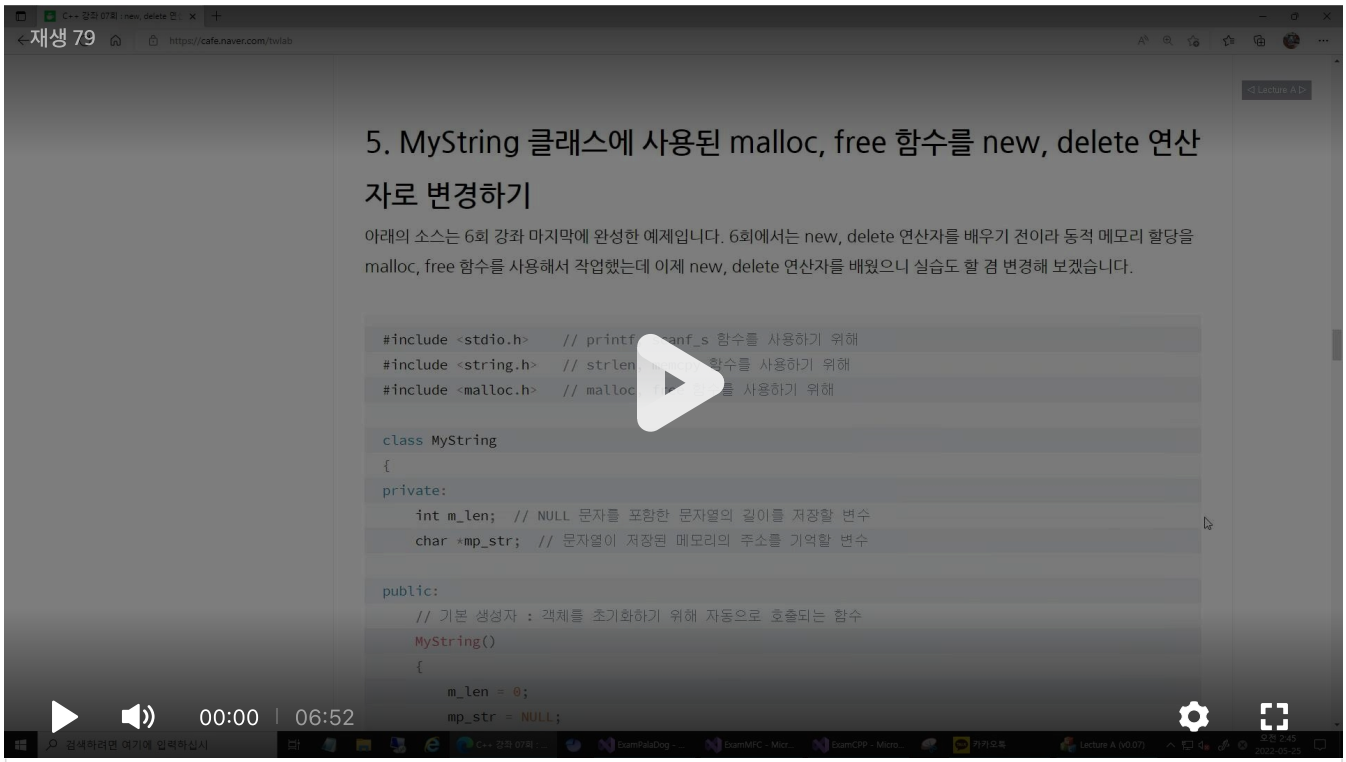
[주의]

이 영상은 제가 공개하는 곳외에 다른 곳에서 배포나 상영이 불가능하고, 다른 분들과 어떤 목적으로도 공유하시면 안됩니다. 여러분들이 이 규칙을 잘 지켜주셔야지 이런 강좌를 제가 계속 진행할 수 있는 힘이 유지됩니다.

C 언어에서는 동적으로 메모리를 할당할 때 malloc, free 함수를 사용했지만 C++ 언어에서는 객체를 동적으로 생성하거나 제거하기 위해 new, delete라는 새로운 연산자를 사용합니다. 그래서 이번 강좌에서는 C++ 언어가 왜 new, delete 연산자를 추가했는지에 대한 이유와 이 연산자를 사용할 때 주의할 점을 소개하겠습니다.



C++ 강좌 7회. new, delete 연산자



C++ 강좌 07회-1보강 : new, delete 연산자 실습

1. 객체 할당에 malloc 함수를 사용하면 문제가 발생합니다.

6회차 강좌에서 만든 MyString 클래스는 아래와 같이 지역 변수처럼 인스턴스해서 사용할 수도 있지만 malloc 함수를 사용해서 동적으로 메모리 할당해서 사용할 수도 있습니다. 객체를 지역 변수처럼 인스턴스 시키면 스택(stack) 메모리에 위치하게 되고 malloc 함수를 사용해서 할당하면 힙(heap) 메모리에 위치하게 됩니다.

```
MyString my_str;  // my_str 객체를 인스턴스 시킨다.
MyString *p = (MyString *)malloc(sizeof(MyString));  // 객체를 동적으로 할당한다.
```

그런데 위 코드에 대해 설명하면서 지역 변수처럼 사용한 코드에는 '객체를 인스턴스 시킨다'고 이야기 했고, malloc 함수를 사용한 코드에는 '객체를 동적으로 할당한다'고 이야기 했습니다. 이렇게 구분해서 이야기한 이유는 my_str 객체를 만든 때는 객체가 메모리에 할당되고 생성자가 자동으로 호출되기 때문에 객체가 인스턴스 되었다고 이야기 한 것입니다. 그런데 malloc 함수를 사용하면 MyString 자료형과 상관없이 sizeof(MyString) 크기만큼만 메모리를 할당하겠다는 뜻으로만 사용되어 MyString 클래스의 생성자가 호출되지 않습니다. 따라서 malloc 함수를 사용하면 단순히 메모리만 할당되기 때문에 인스턴스라는 표현을 사용하지 못하고 객체를 동적으로 할당한다고 이야기를 한 것입니다.

좀더 보충해서 설명하면 malloc 함수는 C 언어용 함수이기 때문에 객체에 대한 개념이 적용되어 있지 않습니다. 그래서 함께 사용되는 자료형을 단순히 크기로만 계산하여 자료형에 따른 특성이 반영되지 않습니다. 예를 들어, 위 코드에서 malloc 함수는 인자로 단순히 할당할 메모리 크기를 사용하기 때문에 MyString이 아닌 sizeof(MyString)을 사용하는 것입니다. 그리고 malloc 함수는 반환 주소에 대한 자료형도 자료형이 정해져 있지 않다는 void *를 사용하기 때문에 (MyString *)로 형 변환 연산자를 함께 사용해야 합니다.

결국 malloc 함수를 사용해서 객체를 메모리 할당하면 단순 할당만 될 뿐, 생성자가 함께 호출되지 않아 객체가 초기화 되지 않습니다. 따라서 초기화 되지 않은 객체를 사용하다가 문제가 발생하게 될 것입니다. 그리고 동일한 이유로 동적 할당된 객체를 free 함수로 해제하게 되면 객체 자체를 할당한 메모리는 해제됩니다. 하지만 객체가 제공하는 파괴자가 자동으로 호출되지 않아 객체가 내부적으로 관리하던 메모리는 정리되지 않습니다. 따라서 **동적으로 할당된 객체를 free 함수를 사용해서 해제하면 메모리 손실이 발생**할 수 있습니다.

2. 객체에는 new, delete 연산자를 사용합시다.

malloc, free 함수는 메모리를 할당하는 함수이지 객체의 생성자나 파괴자를 자동으로 호출해주는 개념이 없습니다. 그리고 이 함수에 생성자나 파괴자를 호출하는 기능을 추가하고 싶어도 malloc, free 함수는 우리가 만든 클래스보다 먼저 작성되고 컴파일되어 라이브러리에 포함된 함수라서 우리가 만든 클래스의 존재를 알수 없습니다. 즉, 우리가 만든 클래스보다 malloc, free 함수가 더 먼저 만들어지기 때문에 malloc, free 함수를 만드는 시점에 우리가 만든 클래스를 알게 한다는 것은 사실상 불가능합니다.

결국 **우리가 만든 클래스와 동일 시점에서 컴파일되는 동적 할당 개념이 제공되어야 하기 때문에 함수 개념으로는 구현이 불가능**합니다. 그래서 **C++ 언어에서는 컴파일 시점은 동일하게 사용하는 new, delete라는 새로운 연산자로 동적 메모리 할당 기술은 제공**합니다. 이렇게 연산자로 제공되면 연산자는 일반 문법이기 때문에 해당 소스에서 클래스와 함께 명령문으로 사용되고 컴파일됩니다. 따라서 new, delete 연산자는 사용자가 정의한 클래스와 동일 시점에서 컴파일되기 때문에 우리가 만든 클래스의 특성을 반영(생성자, 파괴자 호출)할 수 있습니다.

여기서 **new 연산자가 malloc 함수를 대체**하고 **delete 연산자가 free 함수를 대체**하게 됩니다. 예를 들어, MyString 클래스를 동적으로 할당했다가 해제하려면 아래와 같이 코드를 구성하면 됩니다. 이렇게 코드를 구성하면 **new 연산자를 통해 객체가 동적으로 할당되고 기본 생성자가 호출**되며, 객체의 사용이 끝나 **delete 연산자를 사용하면 객체 파괴자가 호출되고 객체를 할당한 메모리가 해제**됩니다.

```
MyString *p = new MyString;  // 객체를 인스턴스 시킨다.  (객체 할당 + 생성자 호출)
// p->MyString(); 처럼 기본 생성자가 자동으로 호출된다.

// p->~MyString(); 처럼 파괴자가 자동으로 호출된다.
delete p;  // 객체를 할당한 메모리를 해제한다.
```

malloc, free 함수를 사용하기 위해서는 malloc.h 헤더 파일을 추가해야 하지만 new, delete 연산자는 문법이라서 별도의 헤더 파일을 추가할 필요가 없습니다.

3. new, delete 연산자는 일반 자료형에도 사용이 가능합니다.

C 언어에서 **int data[5]; 형식의 메모리를 동적으로 할당**하면 아래와 같이 코드를 사용해야 합니다. 여기서 malloc 함수에 사용된 할당된 크기는 sizeof(int)*5 대신 sizeof(int[5])라고 사용해도 되고 sizeof(20)라고 사용해도 됩니다.

```
int *p;
p = (int *)malloc(sizeof(int)*5);  // 20 바이트 메모리 할당

// ...사용 코드 생략 ...

free(p);  // 할당된 메모리 해제
```

위 코드에서도 malloc 함수는 자료형의 특성(배열, 구조체, ...)은 반영하지 않고 순수하게 할당할 크기를 인자로 받습니다. 그리고 이렇게 할당된 주소는 개발자가 원하는대로 사용할 수 있게 void * 자료형으로 반환하기 때문에 4 바이트 크기로 구분해서 사용하고 싶다면 int * 자료형의 포인터를 선언해서 (int *) 형변환을 사용해서 주소를 대입하면 됩니다.

이처럼 C 언어는 연산자를 사용하지 않고 **함수를 사용해서 자료형 형식과 상관없이 동적 메모리 할당은 구현했는데 이것은 동적 메모리 할당이라는 개념이 C 언어 문법에 종속되지 않게 하려는 의도**였습니다. 예를 들어, 새로운 운영체제 개발에 C 언어가 사용되었을 때, 해당 운영체제에서 동적 메모리 할당의 개념이 바뀌어도 C 언어 컴파일러를 수정하는 것이 아니라 malloc 함수만 수정해서 사용 가능하게 만드는 것이 목표였기 때문입니다. 결국 C 언어가 운영체제가 제공하는 기술에 대해 독립적으로 설계되어야지 C 언어가 이식성이 좋은 언어가 되기 때문입니다.

하지만 C++ 언어는 운영체제를 만드는 언어가 아니라 운영체제에서 동작하는 응용 프로그램을 편하게 만들기 위해 만들어진 언어입니다. 따라서 이식성이 떨어지더라도 더 편하게 개발할 수 있는 형태로 문법이 발전했고 메모리 할당 같은 기술도 함수가 아닌 연산자로 바뀐 것입니다. 즉, 메모리 할당을 하는 함수가 연산자가 되었다는 뜻은 메모리 할당 기능 자체가 C++ 언어의 문법이 되었다는 뜻입니다.

그래서 **new, delete 연산자는 객체뿐만 아니라 일반 자료형이나 사용자 정의 자료형에도 그대로 사용이 가능**합니다. 예를 들어, 위 코드를 new, delete 연산자를 사용하게 수정하면 다음과 같습니다. 연산자를 사용하게 되면 자료형을 체크하는 시점이 컴파일하는 시점이기 때문에 단순 크기가 아닌 자료형의 특성을 그대로 반영할 수 있습니다. 그래서 sizeof 연산자를 사용하지 않고 자료형 표현을 사용해서 메모리를 할당하는 것이 가능합니다.

```
int *p;
p = new int[5];  // 20 바이트 메모리 할당

// ... 사용 코드 생략 ...

delete[] p;  // 할당된 메모리 해제
```

malloc, free 함수를 사용하는 코드와 비교했을 때 **sizeof 연산자와 형변환 연산자를 사용하는 코드가 줄어서 코드가 더 간단하게 표현**됩니다. 따라서 C++ 언어에서는 기본 자료형이나 사용자 정의 자료형에 malloc, free 함수를 사용하는 것도 가능하지만 대부분의 개발자가 new, delete 연산자를 사용합니다.

4. delete 연산자를 사용할 때 주의할 점

delete 연산자는 new 연산자의 형식에 영향을 받습니다. new 연산자에 [ ] 기호 사용 여부에 따라 delete 연산자에 [ ] 기호 사용 여부가 정해집니다.

예를 들어, 위 예제에서 20 바이트의 메모리를 그둡 할당하기 위해 new 연산자에 [ ] 기호를 사용했습니다. 그래서 delete 연산자를 사용할 때도 [ ] 기호를 사용한 것입니다. 이것은 선택 사항이 아니라 반드시 지키는 것이 좋습니다. 의도적으로 그둡 할당한 메모리를 부분해제하는 경우에 [ ] 기호를 생략하는 경우도 있지만 이런 경우는 거의 없습니다. 따라서 대부분 new 연산에서 [ ] 기호를 사용했는데 delete 연산에서 [ ] 기호를 사용하지 않았다면 개발자가 실수한 것입니다. 이렇게 실수를 하면 할당한 메모리가 다 해제되지 않아서 메모리 손실이 발생합니다.

반대로 아래와 같이 4바이트 단일 메모리를 할당할 때는 new 연산에 [ ]기호를 사용하지 않습니다. 따라서 delete 연산자를 사용해서 메모리를 해제할 때도 [ ] 기호를 사용하지 않습니다. 만약 new 연산자에 [ ] 기호를 사용하지 않고 메모리를 할당했는데 개발자가 delete 연산자를 사용할 때 [ ] 기호를 사용해서 메모리를 해제하면 할당된 메모리보다 더 많은 메모리를 해제하려고 시도하다가 메모리 오듀가 발생하게 됩니다.

```
int *p = new int;  // 4 바이트 메모리 할당

// ... 사용 코드 생략 ...

delete p;  // 할당된 메모리 해제
```

그런데 생각보다 이 규칙을 지키지 않아서 메모리에 문제가 생기는 경우가 많습니다. 따라서 이 규칙을 지키는 어려움이 있다면 new, delete 연산을 사용할 때 반드시 [ ] 기호를 사용하도록 하는 것도 방법입니다. 예를 들어, 위와 같은 코드도 아래와 같이 [ ] 기호를 사용해서 표현하는 것이 가능하기 때문에 계속 문제가 생긴다면 이 방법을 사용해보세요.

```
int *p = new int[1];  // 4 바이트 메모리 할당

// ... 사용 코드 생략 ...

delete[] p;  // 할당된 메모리 해제
```

아니면 new 연산자에서 [ ] 기호 사용 유무에 상관없이 그냥 해제하고 싶다면 free 함수를 사용해도 됩니다. 객체 개념을 사용할 때는 객체의 파괴자 함수 호출이 안되어서 문제가 되겠지만 일반 자료형은 free 함수로 안전하게 해제가 가능합니다.

5. MyString 클래스에 사용된 malloc, free 함수를 new, delete 연산자로 변경하기

아래의 소스는 6회 강좌 마지막에 완성한 예제입니다. 6회에서는 new, delete 연산자를 배우기 전이라 동적 메모리 할당을 malloc, free 함수를 사용해서 작업했는데 이제 new, delete 연산자를 배웠으니 실습도 할 겸 변경해 보겠습니다.

```
#include <stdio.h>    // printf, scanf_s 함수를 사용하기 위해
#include <string.h>   // strlen, memcpy 함수를 사용하기 위해
#include <malloc.h>   // malloc, free 함수를 사용하기 위해

class MyString
{
private:
    int m_len;  // NULL 문자를 포함한 문자열의 길이를 저장할 변수
    char *mp_str;  // 문자열이 저장된 메모리의 주소를 기억할 변수

public:
    // 기본 생성자 : 객체를 초기화하기 위해 자동으로 호출되는 함수
    MyString()
    {
        m_len = 0;
        mp_str = NULL;
    }

    // 생성자 : 지정된 문자열을 사용해서 객체를 초기화하는 생성자 함수
    MyString(const char *ap_str)
    {
        mp_str = NULL;
        CopyDynamicString(ap_str);
    }

    // 파괴자 : 객체를 정리하기 위해 자동으로 호출되는 함수
    ~MyString()
    {
        CleanUp();
    }

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 이미 할당된 메모리가 있으면 제거한다.
        CleanUp();

        // 매개 변수로 전달된 문자열의 길이를 구한다.
        m_len = strlen(a_str) + 1;
        // 문자열을 저장하기 위해 동적 메모리를 할당한다.
        mp_str = (char *)malloc(m_len);
        // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
        if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // 이 객체가 관리하는 문자열 정보를 출력하는 함수
    void ShowStringInfo()
    {
        // 관리하는 문자열의 정보를 출력한다.
        if (mp_str != NULL) printf("%s : %d\n", mp_str, m_len);
```

```
        else printf("보관된 문자열이 없습니다.\n");
    }

    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            free(mp_str);
            mp_str = NULL;
        }
    }
};

int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    MyString my_str; // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    my_str.CopyDynamicString(str);
    // 문자열이 잘 복사되었는지 확인한다.
    my_str.ShowStringInfo();

    // 문자열 상수를 인자로 가지는 생성자 사용
    MyString temp_str("tipsware lab");
    // 문자열이 잘 복사되었는지 확인한다.
    temp_str.ShowStringInfo();

    return 0;
}
```

수정 순서는 다음과 같습니다.

◆ malloc, free 함수를 사용하지 않기 때문에 malloc.h 헤더 파일을 포함할 필요가 없습니다.
◆ CopyDynamicString 함수에 사용된 malloc 함수를 new 연산자를 사용해서 아래와 같이 변경합니다.

```
// mp_str = (char *)malloc(m_len);  // 이전 코드
mp_str = new char[m_len];
```

◆ CleanUp 함수에 사용된 free 함수를 delete 연산자를 사용해서 아래와 같이 변경합니다.

```
// free(mp_str);  // 이전 코드
delete[] mp_str;
```

◆ 객체를 동적 할당하는 것도 실습하기 위해, main 함수에 사용된 my_str, temp_str 객체를 인스턴스 해서 사용하는 코드를 모두 new, delete 연산자를 사용해서 동적 객체 할당 방식으로 변경합니다.

```
// 문자열 정보를 저장할 객체 (기본 생성자 사용)
// MyString my_str;  // 기존 코드
MyString *p_my_str = new MyString;
// 동적 메모리를 할당하고 str에 저장된 문자열을 복사
// my_str.CopyDynamicString(str);  // 기존 코드
p_my_str->CopyDynamicString(str);
// 문자열이 잘 복사되었는지 확인한다.
// my_str.ShowStringInfo();  // 기존 코드
p_my_str->ShowStringInfo();
// 동적 할당했던 객체를 제거한다.
delete p_my_str;

// 문자열 상수를 인자로 가지는 생성자 사용
// MyString temp_str("tipsware lab");  // 기존 코드
MyString *p_temp_str = new MyString("tipsware lab");
// 문자열이 잘 복사되었는지 확인한다.
// temp_str.ShowStringInfo();  // 기존 코드
p_temp_str->ShowStringInfo();
// 동적 할당했던 객체를 제거한다.
delete p_temp_str;
```

malloc, free 함수가 사용된 코드를 new, delete 연산자로 수정한 전체 소스는 다음과 같습니다.

```
#include <stdio.h>    // printf, scanf_s 함수를 사용하기 위해
#include <string.h>   // strlen, memcpy 함수를 사용하기 위해

class MyString
{
private:
    int m_len;  // NULL 문자를 포함한 문자열의 길이를 저장할 변수
    char *mp_str;  // 문자열이 저장된 메모리의 주소를 기억할 변수

public:
    // 기본 생성자 : 객체를 초기화하기 위해 자동으로 호출되는 함수
    MyString()
    {
        m_len = 0;
        mp_str = NULL;
    }

    // 생성자 : 지정된 문자열을 사용해서 객체를 초기화하는 생성자 함수
    MyString(const char *ap_str)
    {
        mp_str = NULL;
        CopyDynamicString(ap_str);
    }

    // 파괴자 : 객체를 정리하기 위해 자동으로 호출되는 함수
    ~MyString()
    {
        CleanUp();
    }

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 이미 할당된 메모리가 있으면 제거한다.
        CleanUp();

        // 매개 변수로 전달된 문자열의 길이를 구한다.
        m_len = strlen(a_str) + 1;
        // 문자열을 저장하기 위해 동적 메모리를 할당한다.
        mp_str = new char[m_len];
        // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
        if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // 이 객체가 관리하는 문자열 정보를 출력하는 함수
    void ShowStringInfo()
    {
        // 관리하는 문자열의 정보를 출력한다.
        if (mp_str != NULL) printf("%s : %d\n", mp_str, m_len);
        else printf("보관된 문자열이 없습니다.\n");
```

```
    }

    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            delete[] mp_str;
            mp_str = NULL;
        }
    }
};

int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    // MyString my_str;  // 기존 코드
    MyString *p_my_str = new MyString;
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    // my_str.CopyDynamicString(str);  // 기존 코드
    p_my_str->CopyDynamicString(str);
    // 문자열이 잘 복사되었는지 확인한다.
    // my_str.ShowStringInfo();  // 기존 코드
    p_my_str->ShowStringInfo();
    // 동적 할당했던 객체를 제거한다.
    delete p_my_str;

    // 문자열 상수를 인자로 가지는 생성자 사용
    // MyString temp_str("tipsware lab");  // 기존 코드
    MyString *p_temp_str = new MyString("tipsware lab");
    // 문자열이 잘 복사되었는지 확인한다.
    // temp_str.ShowStringInfo();  // 기존 코드
    p_temp_str->ShowStringInfo();
    // 동적 할당했던 객체를 제거한다.
    delete p_temp_str;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
tips
tips : 5
tipsware lab : 13

이 창을 닫으려면 아무 키나 누르세요...
```

댓글 등록순 최신순

관심글 댓글 알림

황금잉어가물치 3

new로 할당한 변수도 NULL을 확인해주고 delete 해제하는것이 맞는지요? malloc함수로 할당할 경우 NULL을 확인하고 값을 입력하거나 해제하는데 new와 delete도 NULL을 확인하는게 맞는지요? 답변부탁드립니다.

```
    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            delete[] mp_str;
            mp_str = NULL;
        }
    }
};
```

2022.11.12. 19:26 답글쓰기

김성엽 M 작성자

네~ 확인하고 해제하는 것이 맞습니다. 물론, 100% 할당을 확신한다면 체크 안하고 그냥 해제하기도 합니다.

2022.11.12. 19:39 답글쓰기

황금잉어가물치 3

김성엽 감사합니다

2022.11.13. 02:01 답글쓰기

카일 3

강의 잘 들었습니다.^^

2022.11.25. 20:25 답글쓰기

김성엽 M 작성자

수고하셨습니다

2022.11.25. 21:51 답글쓰기

황금잉어가물치 3

new로 할당된 크기는 어떻게 구하나요?
예로 char *ch=new char[10];으로 했을때 sizeof(ch) 혹은 _msize(ch)로 new로 할당된 크기를 구할 수 있나요?
아니면 c++에서 new로 할당된 크기를 구할 수 있는 함수가 있는지 궁금합니다.

2022.12.04. 15:17 답글쓰기

김성엽 M 작성자

네~ _msize 함수로 구하면 됩니다. 아니면 아래의 자료처럼 new 연산자를 재정의해서 직접 관리해도 됩니다.
https://blog.naver.com/tipsware/221605375790

2022.12.04. 18:34 답글쓰기

황금잉어가물치 3

김성엽 감사합니다

2022.12.04. 18:40 답글쓰기

oceanide 2

안녕하세요. 좋은 강의 영상을 제공해주셔서 감사합니다.
한 가지 궁금한 사항이 있습니다.

C언어의 함수의 매개변수가 없는 경우, void를 명시적으로 적어서 사용했었습니다.
C++언어의 멤버함수에서 매개변수가 없는 경우에 void를 명시적으로 적어주지 않는 이유가
this 포인터가 넘어오기 때문이라고 생각하면 되는걸까요?

감사합니다.

2024.03.19. 10:46 답글쓰기

김성엽 작성자

아닙니다. C 언어도 매개 변수가 없는 경우 void 안적어도 됩니다. 그리고 CPP 함수도 인자가 없을때 void 적고 선언해도 됩니다. this 포인터는 어차피 명시적으로 매개 변수에 포함되지 않기 때문에 문법적으로는 매개변수가 아닙니다. 내부적으로 매개 변수처럼 넘어갈 뿐입니다.

2024.03.19. 11:14 답글쓰기

김성엽 작성자

C 언어 문법에서도 예전에는 매개 변수가 없음을 강조하기 위해 void 사용을 권장했지만 꼭 그렇게 사용해야 하는건 아닙니다. 그리고 이제 함수 개념에 익숙해진 개발자들이 많아서 굳이 void를 적을 필요는 없다고 생각합니다.

2024.03.19. 11:15 답글쓰기

oceanide 2

김성엽 네 답변 감사합니다 좋은 하루 되세요

2024.03.19. 11:17 답글쓰기

김성엽 작성자

사실 개인적으로 생략을 하는 것보다 원칙적인 표현을 지키는 것이 좋다고 생각하지만 오해를 만들만한 코드가 아니면 생략 가능한 표현은 생략해서 사용하는 것도 좋다고 생각합니다. 우리가 int 라고 적으면 signed int에서 signed가 생략된 표현인데 이 표현을 원칙적으로 signed int라고 다 적는 사람은 이제 없을거라고 생각합니다.

2024.03.19. 11:18 답글쓰기

oceanide 2

김성엽 네, 말씀하시려는 뜻은 이해했습니다.
기본적인 질문인데 자세히 답변해주셔서 다시 한번 감사드립니다.

2024.03.19. 11:23 답글쓰기

김성엽 작성자

oceanide

2024.03.19. 11:25 답글쓰기

dh221009

댓글을 남겨보세요

등록