

C++ 언어 >

C++ 강좌 18회 : 복사 생성자



김성엽 카페매니저

+ 구독

1:1 채팅

2022.07.11. 00:13 조회 411



댓글 19

URL 복사



[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.

이번 강좌에서는 동일한 클래스로 만들어진 객체들 간에 대입 연산자로 데이터를 복사할 때 주의해야 할 점과 객체를 생성할 때 사용한 대입 연산자로 인해 호출되는 복사 생성자(copy constructor)에 대해 설명하겠습니다.

재생 124

https://cafe.naver.com/twlab

카페매니저 김성엽

since 2019.10.02.

카페소개

카페관리

소통게

씨앗5단계

739

초대하기

즐거찾는 멤버

43명

게시판 구독수

13회

우리카페앱 수

2회

주제 컴퓨터/통신 > 프로그래밍 언어

지역 서울특별시 광진구

카페 글쓰기

카페 채팅

검색

즐거찾는 게시판

전체글보기

221

시스템게시판

공지 사항

자유게시판

후기 게시판

C++ 언어 >

C++ 강좌 18회 : 복사 생성자

김성엽 카페매니저

2022.07.11. 00:13 조회 88

댓글 0

URL 복사

알림

다

[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.

00:00

53:00

이번 강좌에서는 동일한 클래스로 만들어진 객체들 간에 대입 연산자로 데이터를 복사할 때 주의해야 할 점과 객체를 생성

18회차. 복사 생성자

1. 대입 연산자로 그룹 메모리의 값을 복사할 수 있는가?

대입 연산자는 기본적으로 **오른쪽에 있는 값을 왼쪽에 있는 변수(메모리)에 저장할 때 사용하는 연산자**입니다. 예를 들어, 아래와 같이 사용하면 b 변수에 저장된 값(7)이 a 변수에 저장되거나 a 변수에 상숫값 3이 저장됩니다.

```
int a, b = 7;

a = b;    // b 변수에 저장된 값 7이 a 변수에 저장
a = 3;    // a 변수에 상숫값 3이 저장
```

그런데 이런 규칙이 모든 변수에 적용되는 것은 아닙니다. 아래와 같이 **배열 문법으로 선언한 data, temp 변수는 대입 연산자에 변수 이름만 사용하면 오류**로 처리됩니다. 이것은 data, temp 변수가 단순히 배열 개념으로 만들어진 그룹 메모리라서 그런 것이 아니라 **변수 이름만 사용했을 때 각 변수의 시작 주소를 의미하고 이 주소는 변경이 불가능하기 때문**입니다. 즉, 아래의 표현은 temp 변수의 시작 주소로 data 변수의 시작 주소를 변경하겠다는 의미인데 data 변수의 주소는 변경이 불가능하기 때문에 오류가 발생하는 것입니다.

배열의 이름은 배열의 시작 주소이다!

1. 이 글을 읽기 전에 봐야 할 내용 이 글은 아래에 링크한 글에 연결되는 내용입니다. 따라서 아래의 글을...

blog.naver.com

```
int data[3] = { 0, };
int temp[3] = { 1, 2, 3 };

data = temp;    // 오류 발생! data 변수의 시작 주소는 변경할 수 없다.
```

그래서 배열로 선언된 변수의 경우 복사를 하고 싶다면 아래와 같이 **각 항목을 직접 대입**해야 합니다.

```
int data[3] = { 0, };
int temp[3] = { 1, 2, 3 };

// temp 변수의 각 항목을 data 변수의 각 항목에 대입한다.
data[0] = temp[0];
data[1] = temp[1];
data[2] = temp[2];
```

위 코드는 항목이 세 개라서 직접 적었지만 항목의 수가 많다면 아래와 같이 반복문을 사용하면 됩니다.

```
int data[7] = { 0, };
int temp[7] = { 1, 2, 3, 4, 5, 6, 7 };

// temp 변수의 각 항목을 data 변수의 각 항목에 대입한다.
for(int i = 0; i < 7; i++) data[i] = temp[i];
```

그리고 배열 문법을 사용해서 만든 변수는 **그룹지어진 항목들이 메모리에 순서대로 나열되기 때문에 아래와 같이 원하는 만큼 메모리를 복사하는 memcpy 함수를 사용해서 복사**해도 됩니다.

```
int data[7] = { 0, };
int temp[7] = { 1, 2, 3, 4, 5, 6, 7 };

// data 변수의 크기(28bytes)만큼 temp 변수의 시작 주소에서 값을 복사해서 data 변수에 복사한다.
memcpy(data, temp, sizeof(data));
```

이처럼 **배열 문법으로 그룹지어진 메모리는 대입 연산자를 사용해서 한 번에 내용을 복사할 수 없습니다.**

하지만 배열과 달리 **구조체 문법으로 그룹지어진 메모리는 대입 연산자를 사용해서 전체 내용을 다른 변수에 복사하는 것이 가능합니다.** 예를 들어, 아래와 같이 구조체를 사용해서 Tipsware라는 사용자 정의 자료형을 만들고 이 자료형으로 a, b 변수를 선언했습니다. 그리고 a 변수의 각 항목에는 0, 1, 2를 저장하고 b 변수의 각 항목에는 100, 50, 70을 저장했습니다. 이런 상태에서 대입 연산자를 사용해 **b 변수의 값을 a 변수에 대입해도 정상적으로 data, temp1, temp2 항목에 값이 복사**된다는 뜻입니다.

```
#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;   // std::를 생략하기 위해서

struct Tipsware
{
    int data;
    short temp1;
    short temp2;
};

int main()
{
    // Tipsware 구조체로 a, b 변수를 만들고 초기화한다.
    Tipsware a = { 0, 1, 2 }, b = { 100, 50, 70 };
    // a 변수의 값을 출력한다.
    cout << "a: " << a.data << ", " << a.temp1 << ", " << a.temp2 << endl;

    // b 변수의 값을 a 변수에 대입한다.
    a = b;

    // a 변수의 값을 다시 출력한다.
    cout << "a: " << a.data << ", " << a.temp1 << ", " << a.temp2 << endl;

    return 0;
}
```

위 예제는 아래와 같이 실행됩니다. a = b; 명령이 수행된 후에 a 변수의 값이 b 변수와 동일해진 것을 확인할 수 있습니다.

```
a: 0, 1, 2
a: 100, 50, 70
```

이 창을 닫으려면 아무 키나 누르세요...

어떻게 보면 구조체 변수도 배열 변수처럼 그룹지어진 메모리이기 때문에 아래와 같이 **각 항목을 한 개씩 지정해서 복사하는 것이 원칙**적인 표현일 것입니다. 그런데도 배열로 선언된 변수는 오류가 발생하고 구조체로 선언된 변수는 오류가 발생하지 않는 이유가 무엇일까요?

```
a.data = b.data;
a.temp1 = b.temp1;
a.temp2 = b.temp2;
```

그 이유는 매우 단순합니다. **배열로 선언된 변수는 이름만 적었을 때 '주소' 표현이고 구조체 변수는 이름만 적었을 때 일반 변수처럼 '값'을 의미하는 표현**이기 때문입니다. 즉, 구조체 표현도 아래와 같이 적으면 오류가 발생합니다. 결국 $a = b$;가 처리된 것은 기본 크기의 메모리든 그룹 메모리든 해당 크기만큼 다른 메모리에 복사하겠다는 뜻이기 때문에 컴파일러가 오류없이 값이 복사되는 코드를 처리해준 것입니다.

```
// a, b 변수는 Tipsware 구조체로 선언된 변수라고 가정
&a = &b; // 오류 발생! &a는 a의 주소를 의미하는 상숫값으로 처리되기 때문에 값을 대입하는 것이 불가능
```

그리고 조금 더 설명하면 **기계어는 기본적으로 1, 2, 4, 8 단위로 값을 복사하기 때문에 구조체의 크기가 8바이트를 넘어서게 되면 문제**가 될 것입니다. 예를 들어, Tipsware 구조체의 크기가 20바이트로 가정하고 $a = b$; 명령을 사용하면 컴파일러가 이 명령어를 기계어로 번역할 때, 값의 복사 단위가 8바이트를 넘어서기 때문에 문제가 될 것입니다. 그래서 **컴파일러는 구조체로 만들어진 변수가 8바이트보다 큰 경우, 해당 값을 8바이트 단위로 잘라서 여러 번 나누어 복사**합니다. 따라서 a, b 변수가 각 20바이트 크기의 메모리라면 8바이트씩 3번으로 나누어 b 변수의 값을 a 변수에 복사하게 됩니다.

그리고 이렇게 8바이트로 나누어 복사하는 방법은 구조체의 크기가 커지면 성능이 저하되는 문제가 생기기 때문에 **최적화가 적용되면 아래와 같이 memcpy 함수를 사용하는 것과 유사한 코드로 변경**됩니다. 이것은 구조체로 선언한 a, b 변수도 그룹 메모리이고 동일한 형식이기 때문에 b 변수의 시작 주소부터 Tipsware 자료형의 크기만큼 값을 a 변수에 복사하는 방법을 사용하는 것입니다. 그래서 동일한 구조체로 선언된 변수들은 $a = b$; 처럼 간단하게 사용해도 되지만 아래와 같이 메모리를 복사하는 방법을 사용하는 개발자들도 많습니다.

```
memcpy(&a, &b, sizeof(Tipsware));
```

하지만 구조체를 사용한 대입 표현은 **크기가 같더라도 서로 다른 자료형으로 만들어진 변수 간에는 적용되지 않습니다**. 예를 들어, 아래와 같이 8바이트로 크기는 같지만 형식이 다른 Tipsware, Tipsoft 구조체를 정의하고 **Tipsware 자료형으로 a 변수를 선언하고 Tipsoft 자료형으로 b 변수를 선언했다면 $a = b$;는 오류 처리**됩니다.

```

struct Tipsware
{
    int data;
    short temp1;
    short temp2;
};

struct Tipssoft
{
    short data1;
    short data2;
    int temp;
};

int main()
{
    Tipsware a = { 0x01, 0x02, 0x03}; // Tipsware 구조체로 a 변수를 만들고 초기화한다.
    Tipssoft b = { 0x11, 0x22, 0x33}; // Tipssoft 구조체로 b 변수를 만들고 초기화한다.

    a = b; // 오류 발생! 서로 자료형이 달라서 처리 안 됨

    return 0;
}

```

그리고 이 오류는 아래와 같이 **강제로 자료형 변환을 시도해도 해결되지 않습니다**. 왜냐하면 C, C++ 언어의 컴파일러는 '**주소**'에 대한 자료형 변환은 대부분 허용하지만 '**값**'에 대한 자료형 변환은 기본 자료형(char, short, long, ...)에 대해서만 부분적으로 허용하고 구조체로 만든 사용자 정의 자료형인 경우에는 허용하지 않습니다.

```

a = (Tipsware)b;

```

그래서 아래와 같이 **주소를 사용한 자료형 변환을 사용하면 복사가 허용**됩니다. 물론 a, b 변수의 크기가 같다는 가정하에 사용해야 합니다.

```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;   // std::를 생략하기 위해서

struct Tipsware
{
    int data;
    short temp1;
    short temp2;
};

struct Tipssoft
{
    short data1;
    short data2;
    int temp;
};

int main()
{
    Tipsware a = { 0x01, 0x02, 0x03 }; // Tipsware 구조체로 a 변수를 만들고 초기화한다.
    Tipssoft b = { 0x11, 0x22, 0x33 }; // Tipssoft 구조체로 b 변수를 만들고 초기화한다.

    cout.unsetf(ios::dec); // 10진법 출력을 해제한다.
    cout.setf(ios::hex);   // 16진법 출력을 설정한다.

    // a 변수의 값을 출력한다.
    cout << "a: " << a.data << ", " << a.temp1 << ", " << a.temp2 << endl;

    // a, b 변수의 크기가 같기 때문에 주소를 사용한 자료형 변환을 사용해서
    // b 변수의 값을 a 변수에 대입한다.
    a = *(Tipsware *)&b;

    // a 변수의 값을 다시 출력한다.
    cout << "a: " << a.data << ", " << a.temp1 << ", " << a.temp2 << endl;

    return 0;
}

```

위 예제는 아래와 같이 출력됩니다. b 변수에 저장된 값을 바이트 크기에 맞춰서 16진수로 정확하게 적어보면 0x0011, 0x0022, 0x00000033입니다. b 변수의 첫 4바이트(data1, data2 항목이 합쳐진 메모리)가 a 변수의 data 항목에 복사 되기 때문에 0x00220011이 저장된 것입니다. 그리고 b 변수의 나머지 4바이트(temp 항목 메모리)가 a 변수의 temp1, temp2에 2바이트씩 나누어져 복사되어 temp1 항목에는 0x0033이 저장되었고 temp2 항목에는 0x0000이 저장된 것입니다.

```

a: 1, 2, 3
a: 220011, 33, 0

```

이 창을 닫으려면 아무 키나 누르세요...

만약, `a = (Tipsware *)&b;` 표현을 사용하는 것이 어렵다면 아래와 같이 `memcpy` 함수를 사용해도 됩니다. `memcpy` 함수는 주소와 크기만 사용하는 개념이기 때문에 자료형 변환을 별도로 할 필요가 없습니다.

```
memcpy(&a, &b, sizeof(Tipsware));
```

2. 대입 연산자로 그룹 메모리의 값을 복사할 때 주의해야 할 점

프로그램을 하다보면 항상 고정된 크기의 메모리만 사용할 수 없습니다. 그래서 구조체를 구성하는 항목에 고정 크기를 표현하는 일반 변수나 배열 변수 외에도 아래와 같이 포인터 변수를 사용하는 경우도 많습니다. 이 구조체는 정수를 저장하는 메모리를 동적으로 관리하기 위해 사용하는 정수의 개수를 `count` 항목에 저장하고 해당 개수만큼 정수를 저장하기 위해 동적 할당한 메모리의 주소를 `p` 항목에 저장하는 형식입니다.

```
struct Tipsware
{
    int count; // 할당된 정수의 개수를 저장
    int *p;    // 할당된 메모리의 주소를 저장
};
```

그래서 위 구조체는 아래와 같이 사용됩니다. `a` 변수에는 5개의 정숫값을 저장하기 위해 20바이트 메모리를 할당하고 10, 11, 12, 13, 14 값을 저장하고 출력했습니다. 그리고 `b` 변수에도 5개의 정숫값을 저장하기 위해 20바이트 메모리를 할당하고 20, 21, 22, 23, 24 값을 저장하고 출력했습니다. 그리고 이렇게 할당된 메모리는 프로그램 종료 전에 체크해서 해제하도록 했습니다.

```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

struct Tipware
{
    int count; // 할당된 정수의 개수를 저장
    int *p;    // 할당된 메모리의 주소를 저장
};

int main()
{
    // Tipware 구조체로 a, b 변수를 만들고 초기화한다.
    Tipware a = { 5, 0 }, b = { 5, 0 };

    // 20바이트 메모리를 동적 할당하고 그 주소를 a의 p 항목에 저장
    a.p = new int[a.count];
    cout << "a : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 10, 11, 12, 13, 14 저장
    for (int i = 0; i < a.count; i++) {
        *(a.p + i) = 10 + i;
        cout << *(a.p + i) << ", ";
    }
    cout << endl;

    // 20바이트 메모리를 동적 할당하고 그 주소를 b의 p 항목에 저장
    b.p = new int[b.count];
    cout << "b : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 20, 21, 22, 23, 24 저장
    for (int i = 0; i < b.count; i++) {
        *(b.p + i) = 20 + i;
        cout << *(b.p + i) << ", ";
    }
    cout << endl;

    // a 또는 b 변수의 p 항목에 주소가 저장되어 있다면 저장된 주소에 해당하는
    // 메모리를 해제한다.
    if (a.p) delete[] a.p;
    if (b.p) delete[] b.p;

    return 0;
}

```

위 예제는 아래와 같이 실행됩니다.

```

a : 10, 11, 12, 13, 14
b : 20, 21, 22, 23, 24,

```

이 창을 닫으려면 아무 키나 누르세요...

위 예제 코드처럼 동적 메모리를 직접 명시적으로 할당하고 해제하는 코드에서는 문제가 잘 발생하지 않지만, 아래와 같이 코드를 변경하면 프로그램이 메모리를 해제하다가 문제가 발생합니다. 왜냐하면 위 코드에서는 a와 b를 위해 20바이트씩 개별적으로 할당했기 때문에 a.p와 b.p에 저장된 주소가 서로 다릅니다. 그래서 두 메모리는 개별적으로 해제되어야 합니다. 하지만 아래의 예제에서는 b 변수의 내용을 a 변수에 복사했기 때문에 a.p에 저장된 주소는 b.p와 동일합니다. 따라서 a.p에 저장된 주소를 해제하면 b.p에 저장된 주소도 이미 해제된 상태인데 b.p에는 이전 주소가 그대로 주소가 남아있기 때문에 delete[] b.p; 명령을 수행하다가 응용 프로그램에 오류가 발생하게 됩니다. 즉, 이미 해제된 메모리를 해제하려고 하다가 오류가 발생하는 것입니다.

```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

struct Tipware
{
    int count;  // 할당된 정수의 개수를 저장
    int *p;     // 할당된 메모리의 주소를 저장
};

int main()
{
    // Tipware 구조체로 a, b 변수를 만들고 초기화한다.
    Tipware a = { 0, 0 }, b = { 5, 0 };

    // 20바이트 메모리를 동적 할당하고 그 주소를 b의 p 항목에 저장
    b.p = new int[b.count];
    cout << "b : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 20, 21, 22, 23, 24 저장
    for (int i = 0; i < b.count; i++) {
        *(b.p + i) = 20 + i;
        cout << *(b.p + i) << ", ";
    }
    cout << endl;

    // b 변수의 내용을 a 변수에 그대로 복사
    a = b;

    cout << "a : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 10, 11, 12, 13, 14 저장
    for (int i = 0; i < a.count; i++) {
        *(a.p + i) = 10 + i;
        cout << *(a.p + i) << ", ";
    }
    cout << endl;

    // a 또는 b 변수의 p 항목에 주소가 저장되어 있다면 저장된 주소에 해당하는
    // 메모리를 해제한다.
    if (a.p) delete[] a.p;
    if (b.p) delete[] b.p;

    return 0;
}

```

따라서 구조체와 같은 그룹 메모리를 대입 연산자로 복사할 때는 해당 구조체를 구성하는 항목 중에 주소를 저장하는 항목이 있는지 체크해야 하며, **주소를 사용하는 항목이 있다면 구조체 내용 복사로 인해 동적 할당된 메모리가 중복 해제되지 않도록 주의**해야 합니다. 따라서 이런 형식의 프로그램을 해야 한다면 아래와 같이 **구조체에 저장된 주소가 원본인지 아니면 사본 인지를 기억하는 항목을 추가해서 사본인 경우에는 메모리가 해제되지 않도록** 처리해야 합니다.

```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

struct Tipware
{
    int clone; // 복제된 동적 메모리인지 기억 (1이면 복제)
    int count; // 할당된 정수의 개수를 저장
    int *p;    // 할당된 메모리의 주소를 저장
};

int main()
{
    // Tipware 구조체로 a, b 변수를 만들고 초기화한다.
    Tipware a = { 0, }, b = { 0, 5, 0 };

    // 20바이트 메모리를 동적 할당하고 그 주소를 b의 p 항목에 저장
    b.p = new int[b.count];
    cout << "b : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 20, 21, 22, 23, 24 저장
    for (int i = 0; i < b.count; i++) {
        *(b.p + i) = 20 + i;
        cout << *(b.p + i) << ", ";
    }
    cout << endl;

    // b 변수의 내용을 a 변수에 그대로 복사
    a = b;
    // a가 사용하는 동적 메모리는 b에서 할당한 동적 메모리를 사용하는 것임을 명시
    a.clone = 1;

    cout << "a : ";
    // 동적 할당된 20바이트 메모리에 4바이트 단위로 10, 11, 12, 13, 14 저장
    for (int i = 0; i < a.count; i++) {
        *(a.p + i) = 10 + i;
        cout << *(a.p + i) << ", ";
    }
    cout << endl;

    // a 또는 b 변수의 p 항목에 주소가 저장되어 있다면 저장된 주소에 해당하는
    // 메모리를 해제한다. 이때 복제 메모리가 아닌 경우에만 해제한다.
    if (a.p && !a.clone) delete[] a.p;
    if (b.p && !b.clone) delete[] b.p;

    return 0;
}

```

위 예제는 아래와 같이 출력됩니다.

```
b : 20, 21, 22, 23, 24,  
a : 10, 11, 12, 13, 14,
```

이 창을 닫으려면 아무 키나 누르세요...

3. 객체도 대입 연산자로 복제가 가능한가?

일단 2번에서 소개한 코드를 클래스를 사용해서 변경하고 [객체도 구조체와 동일하게 동작하는지 확인](#)해 보겠습니다.


```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

class Tipsware
{
private:
    int m_clone;    // 복제된 동적 메모리인지 기억 (1이면 복제)
    int m_count;    // 할당된 정수의 개수를 저장
    int *mp_data;   // 할당된 메모리의 주소를 저장

public:
    Tipsware()    // 기본 생성자
    {
        m_clone = 0;
        m_count = 0;
        mp_data = NULL;
    }

    Tipsware(int a_count)    // 정숫값을 사용하는 생성자
    {
        m_clone = 0;
        m_count = a_count;
        mp_data = new int [m_count];
    }

    ~Tipsware()    // 파괴자
    {
        // 할당된 메모리가 있고 복제된 상태가 아니라면 메모리를 해제한다.
        if (mp_data && !m_clone) delete[] mp_data;
    }

    // 객체에 데이터를 저장하는 함수
    void SetData(int a_start_value)
    {
        if (mp_data) {
            // 동적 할당된 20바이트 메모리에 4바이트 단위로 a_start_value 부터 1씩 증가한 값을 대입
            for (int i = 0; i < m_count; i++) *(mp_data + i) = a_start_value + i;
        }
    }

    // 클론 상태로 설정하는 함수
    void SetClone()
    {
        m_clone = 1;
    }

    // 현재 객체가 가지고 있는 값을 보여주는 함수
    void ShowData()
    {
        if (mp_data) {
            // 동적 할당된 메모리에 저장된 데이터를 출력한다.
            for (int i = 0; i < m_count; i++) cout << *(mp_data + i) << ", ";
        }
    }
};

```

```

        cout << endl;
    }
}

};

int main()
{
    // Tipsware 클래스로 a, b 객체를 만들고 초기화한다.
    Tipsware a, b(5);

    // b 객체에 할당된 메모리에 20, 21, 22, 23, 24 저장한다.
    b.SetData(20);
    cout << "b : ";
    // b 객체에 저장된 값을 출력한다.
    b.ShowData();

    // b 객체의 내용을 a 객체에 그대로 복사
    a = b;
    // a 객체는 클론 상태로 설정
    a.SetClone();
    // a 객체에 할당된 메모리에 10, 11, 12, 13, 14 저장한다.
    a.SetData(10);

    cout << "a : ";
    // a 객체에 저장된 값을 출력한다.
    a.ShowData();

    cout << endl << "공유 상태 확인" << endl << "b : ";
    // 메모리가 공유되는 것을 확인하기 위해 b 객체에 저장된 값을 출력한다.
    b.ShowData();

    return 0;
}

```

위 예제는 아래와 같이 출력됩니다. b 객체의 메모리를 a 객체에 대입해서 사용했기 때문에 **a 객체에 대입한 값이 b 객체를 사용해도 동일하게 출력되는 것을 확인**수 있습니다.

```

b : 20, 21, 22, 23, 24,
a : 10, 11, 12, 13, 14,

공유 상태 확인
b : 10, 11, 12, 13, 14,

이 창을 닫으려면 아무 키나 누르세요...

```

만약, 의도적으로 객체들 간에 메모리를 공유하게 만들었다면 위와 같이 코드를 구성해도 됩니다. 하지만 객체가 고유하게 데이터를 관리해야 한다면 위 코드는 잘못된 방법입니다. 예를 들어, **a 객체가 값을 10, 11, 12, 13, 14로 변경해도 b 객체는 기존에 자신이 저장하고 있던 값을 그대로 유지해야 한다면 위 방법은 잘못되었다는 뜻**입니다. 그래서 보통의 개발자라면 이 문제를 아래와 같이 대입 연산자 대신 **CopyData 함수를 추가해서 해결**할 것입니다.


```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

class Tipsware
{
private:
    int m_count;    // 할당된 정수의 개수를 저장
    int *mp_data;   // 할당된 메모리의 주소를 저장

public:
    Tipsware()    // 기본 생성자
    {
        m_count = 0;
        mp_data = NULL;
    }

    Tipsware(int a_count)    // 정숫값을 사용하는 생성자
    {
        m_count = a_count;
        mp_data = new int [m_count];
    }

    ~Tipsware()    // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 객체에 데이터를 저장하는 함수
    void SetData(int a_start_value)
    {
        if (mp_data) {
            // 동적 할당된 20바이트 메모리에 4바이트 단위로 a_start_value 부터 1씩 증가한 값을 대입
            for (int i = 0; i < m_count; i++) *(mp_data + i) = a_start_value + i;
        }
    }

    // 다른 객체의 값을 복사하는 함수
    void CopyData(const Tipsware *ap_data)
    {
        // 기존에 할당된 메모리가 있으면 해제한다.
        if (mp_data) delete[] mp_data;

        if (ap_data->mp_data) {
            // 동일한 클래스로 만들어진 객체가 인자로 전달되었을 때는
            // 해당 객체의 private 멤버라고 해도 항목 지정 연산자로 바로 사용이 가능하다.

            // 전달된 객체에서 사용되는 데이터의 수를 저장한다.
            m_count = ap_data->m_count;
            // 데이터를 저장할 메모리를 할당한다.
            mp_data = new int[m_count];
            // 전달된 객체에 저장된 데이터를 할당된 메모리에 복사한다.

```

```

        for (int i = 0; i < m_count; i++) *(mp_data + i) = *(ap_data->mp_data + i);
    } else mp_data = NULL;
}

// 현재 객체가 가지고 있는 값을 보여주는 함수
void ShowData()
{
    if (mp_data) {
        // 동적 할당된 메모리에 저장된 데이터를 출력한다.
        for (int i = 0; i < m_count; i++) cout << *(mp_data + i) << ", ";
        cout << endl;
    }
}

};

int main()
{
    // Tipsware 클래스로 a, b 객체를 만들고 초기화한다.
    Tipsware a, b(5);

    // b 객체에 할당된 메모리에 20, 21, 22, 23, 24 저장한다.
    b.SetData(20);
    cout << "b : ";
    // b 객체에 저장된 값을 출력한다.
    b.ShowData();

    // b 객체의 내용을 a 객체에 그대로 복사
    a.CopyData(&b);
    // a 객체에 할당된 메모리에 10, 11, 12, 13, 14 저장한다.
    a.SetData(10);

    cout << "a : ";
    // a 객체에 저장된 값을 출력한다.
    a.ShowData();

    cout << endl << "공유 상태 확인" << endl << "b : ";
    // 메모리가 공유되는 것을 확인하기 위해 b 객체에 저장된 값을 출력한다.
    b.ShowData();

    return 0;
}

```

위 예제는 아래와 같이 출력됩니다. 위 코드는 복제 개념을 제거하고 CopyData 함수를 사용해서 객체의 값을 복사하기 때문에 a 객체에 메모리와 b 객체의 메모리가 공유되지 않습니다. 따라서 a 객체의 값을 변경하더라도 b 객체의 값은 그대로 유지됩니다.

```
b : 20, 21, 22, 23, 24,  
a : 10, 11, 12, 13, 14,
```

공유 상태 확인

```
b : 20, 21, 22, 23, 24,
```

이 창을 닫으려면 아무 키나 누르세요...

하지만 위와 같이 작업하면 `a.CopyData(&b);` 라고 사용해야 하는 상황에 `a = b;`라고 적어서 버그를 만드는 개발자가 발생하기 마련입니다. 그리고 이 코드는 컴파일러가 오류로 처리하지 않기 때문에 초보 개발자들은 이 문제를 해결하는데 시간이 많이 걸리게 됩니다. 그래서 클래스를 제공하는 개발자들은 이런 문제가 발생하지 않도록 `CopyData` 함수 대신에 대입 연산자 오버로딩을 사용해서 작업합니다. 예를 들어, 아래와 같이 `CopyData` 함수를 대입 연산자 오버로딩 함수로 변경하면 대입 연산자를 사용해도 값이 공유되게 복사하지 않고 `CopyData` 함수를 사용한 것처럼 복사되도록 구성할 수 있습니다.


```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

class Tipsware
{
private:
    int m_count;    // 할당된 정수의 개수를 저장
    int *mp_data;   // 할당된 메모리의 주소를 저장

public:
    Tipsware()    // 기본 생성자
    {
        m_count = 0;
        mp_data = NULL;
    }

    Tipsware(int a_count)    // 정숫값을 사용하는 생성자
    {
        m_count = a_count;
        mp_data = new int [m_count];
    }

    ~Tipsware()    // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 객체에 데이터를 저장하는 함수
    void SetData(int a_start_value)
    {
        if (mp_data) {
            // 동적 할당된 20바이트 메모리에 4바이트 단위로 a_start_value 부터 1씩 증가한 값을 대입
            for (int i = 0; i < m_count; i++) *(mp_data + i) = a_start_value + i;
        }
    }

    // 다른 객체의 값을 복사하기 위해 대입 연산자를 오버로딩 함
    void operator=(const Tipsware &ar_object)
    {
        // 기존에 할당된 메모리가 있으면 해제한다.
        if (mp_data) delete[] mp_data;

        if (ar_object.mp_data) {
            // 동일한 클래스로 만들어진 객체가 인자로 전달되었을 때는
            // 해당 객체의 private 멤버라고 해도 항목 지정 연산자로 바로 사용이 가능하다.

            // 전달된 객체에서 사용되는 데이터의 수를 저장한다.
            m_count = ar_object.m_count;
            // 데이터를 저장할 메모리를 할당한다.
            mp_data = new int[m_count];
            // 전달된 객체에 저장된 데이터를 할당된 메모리에 복사한다.

```

```

        for (int i = 0; i < m_count; i++) *(mp_data + i) = *(ar_object.mp_data + i);
    } else mp_data = NULL;
}

// 현재 객체가 가지고 있는 값을 보여주는 함수
void ShowData()
{
    if (mp_data) {
        // 동적 할당된 메모리에 저장된 데이터를 출력한다.
        for (int i = 0; i < m_count; i++) cout << *(mp_data + i) << ", ";
        cout << endl;
    }
}

};

int main()
{
    // Tipsware 클래스로 a, b 객체를 만들고 초기화한다.
    Tipsware a, b(5);

    // b 객체에 할당된 메모리에 20, 21, 22, 23, 24 저장한다.
    b.SetData(20);
    cout << "b : ";
    // b 객체에 저장된 값을 출력한다.
    b.ShowData();

    // b 객체의 내용을 a 객체에 복사
    a = b;
    // a 객체에 할당된 메모리에 10, 11, 12, 13, 14 저장한다.
    a.SetData(10);

    cout << "a : ";
    // a 객체에 저장된 값을 출력한다.
    a.ShowData();

    cout << endl << "공유 상태 확인" << endl << "b : ";
    // 메모리가 공유되는 것을 확인하기 위해 b 객체에 저장된 값을 출력한다.
    b.ShowData();

    return 0;
}

```

위와 같이 코드를 구성하면 **대입 연산자를 사용해서 객체의 내용을 그대로 복사할 수 있기 때문에 CopyData 함수를 사용하는 것보다 버그 발생 확률이 낮아집니다.**

4. 복사 생성자

3번으로 작업이 모두 완료되었다고 생각하는 개발자도 있겠지만 아직 처리하지 못한 상황이 남아있습니다. 예를 들어, 같은 대입 연산자를 사용하더라도 **a = b;** 라고 **사용했을 때와 Tipsware a = b;** 라고 **사용한 상황은 다르기 때문**입니다. a = b; 코

드는 대입 연산자 오버로딩이 적용되어 객체의 내용이 정상적으로 복사되지만, **Tipsware a = b;** 코드는 **a 객체를 선언하는 상황에서 = 을 사용했기 때문에 대입 연산자 오버로딩이 적용되지 않고 기존 대입 연산자를 사용한 것과 동일하게 동작**합니다. 즉, b 객체가 가지고 있는 멤버 변수값이 그대로 적용되어 메모리가 공유되기 때문에 메모리를 해제할 때 오류가 발생하는 문제가 생깁니다.

이 상황을 테스트 하기 위해 main 함수 코드만 아래와 같이 수정했습니다. **b 객체를 먼저 선언하여 값을 구성하고 a 객체는 생성하면서 = 을 사용해서 초기값을 주는 형식으로 b 객체를 대입하도록 수정**했습니다.

```
int main()
{
    // Tipsware 클래스로 b 객체를 만들고 초기화한다.
    Tipsware b(5);

    // b 객체에 할당된 메모리에 20, 21, 22, 23, 24 저장한다.
    b.SetData(20);
    cout << "b : ";
    // b 객체에 저장된 값을 출력한다.
    b.ShowData();

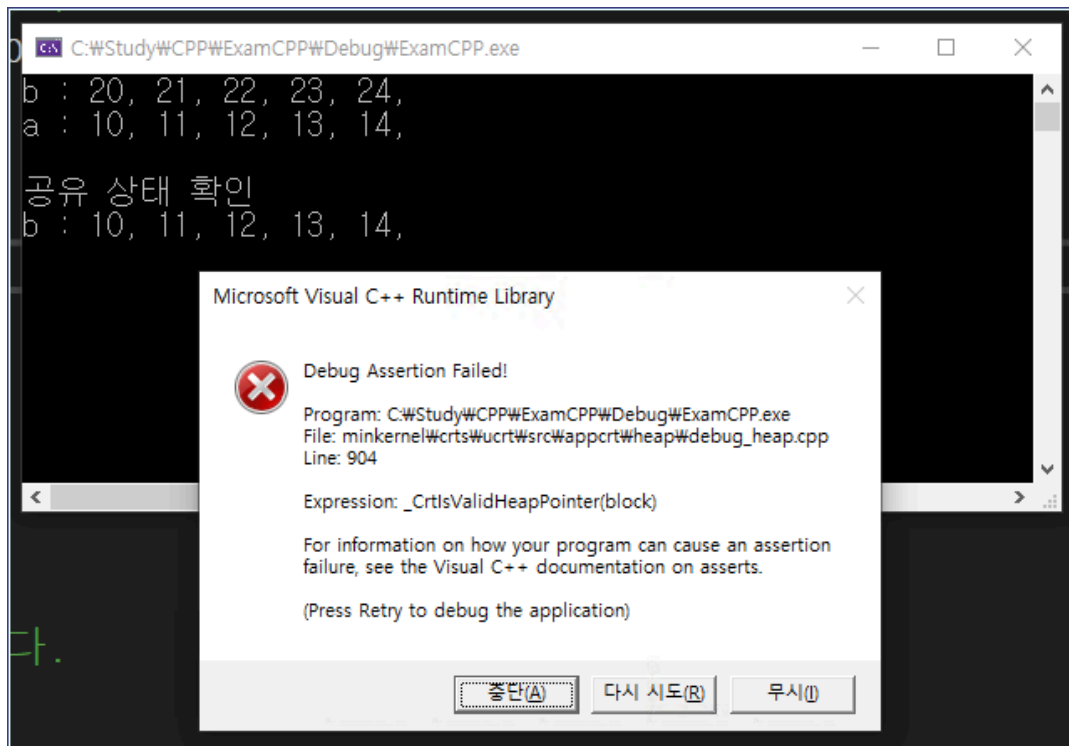
    // a 객체를 생성하면 b 객체의 내용을 a 객체에 복사
    Tipsware a = b;
    // a 객체에 할당된 메모리에 10, 11, 12, 13, 14 저장한다.
    a.SetData(10);

    cout << "a : ";
    // a 객체에 저장된 값을 출력한다.
    a.ShowData();

    cout << endl << "공유 상태 확인" << endl << "b : ";
    // 메모리가 공유되는 것을 확인하기 위해 b 객체에 저장된 값을 출력한다.
    b.ShowData();

    return 0;
}
```

위 코드를 실행하면 아래와 같이 a 객체가 값을 변경해서 b 객체가 가지고 있는 값이 수정된 것도 확인할 수 있고 프로그램이 종료될 때 메모리 오류가 발생하는 것도 확인할 수 있습니다.



동일한 = 연산자를 사용했는데 왜 Tipsware a = b; 코드는 대입 연산자 오버로딩이 적용되지 않는 이유는 생성자와 관련이 있습니다. 객체는 생성될 때 생성자가 호출되기 때문에 초깃값 대입 표현을 사용하면 대입 연산자가 아닌 복사 생성자(copy constructor)가 자동으로 호출되는데 Tipsware 클래스는 복사 생성자가 정의되지 않아서 기본 대입 연산자가 사용된 것입니다. 따라서 이 문제는 Tipsware 객체와 아래와 같은 형식의 복사 생성자만 추가해주면 해결 됩니다.

```
Tipsware(const Tipsware &ar_object) // 복사 생성자
{
    // 대입 연산자 오버로딩에 사용한 코드와 동일한 코드 추가
}
```

그런데 이제 대입 연산자 오버로딩 코드와 복사 생성자의 코드가 중복되는 문제가 있습니다. 그래서 다시 CopyData라는 함수를 추가하고 이 함수를 대입 연산자 오버로딩과 복사 생성자에서 호출해서 사용하도록 아래와 같이 코드를 수정했습니다.


```

#include <iostream>    // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;  // std::를 생략하기 위해서

class Tipsware
{
private:
    int m_count;    // 할당된 정수의 개수를 저장
    int *mp_data;   // 할당된 메모리의 주소를 저장

public:
    Tipsware()    // 기본 생성자
    {
        m_count = 0;
        mp_data = NULL;
    }

    Tipsware(int a_count)    // 정숫값을 사용하는 생성자
    {
        m_count = a_count;
        mp_data = new int [m_count];
    }

    Tipsware(const Tipsware &ar_object)    // 복사 생성자
    {
        mp_data = NULL;
        CopyData(&ar_object);
    }

    ~Tipsware()    // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 객체에 데이터를 저장하는 함수
    void SetData(int a_start_value)
    {
        if (mp_data) {
            // 동적 할당된 20바이트 메모리에 4바이트 단위로 a_start_value 부터 1씩 증가한 값을 대입
            for (int i = 0; i < m_count; i++) *(mp_data + i) = a_start_value + i;
        }
    }

    // 다른 객체의 값을 복사하기 위해 대입 연산자를 오버로딩 함
    void operator=(const Tipsware &ar_object)
    {
        CopyData(&ar_object);
    }

    // 다른 객체의 값을 복사하는 함수
    void CopyData(const Tipsware *ap_object)
    {

```

```

// 기존에 할당된 메모리가 있으면 해제한다.
if (mp_data) delete[] mp_data;

if (ap_object->mp_data) {
    // 동일한 클래스로 만들어진 객체가 인자로 전달되었을 때는
    // 해당 객체의 private 멤버라고 해도 항목 지정 연산자로 바로 사용이 가능하다.

    // 전달된 객체에서 사용되는 데이터의 수를 저장한다.
    m_count = ap_object->m_count;
    // 데이터를 저장할 메모리를 할당한다.
    mp_data = new int[m_count];
    // 전달된 객체에 저장된 데이터를 할당된 메모리에 복사한다.
    for (int i = 0; i < m_count; i++) *(mp_data + i) = *(ap_object->mp_data + i);
} else mp_data = NULL;
}

// 현재 객체가 가지고 있는 값을 보여주는 함수
void ShowData()
{
    if (mp_data) {
        // 동적 할당된 메모리에 저장된 데이터를 출력한다.
        for (int i = 0; i < m_count; i++) cout << *(mp_data + i) << ", ";
        cout << endl;
    }
}

};

int main()
{
    // Tipsware 클래스로 b 객체를 만들고 초기화한다.
    Tipsware b(5);

    // b 객체에 할당된 메모리에 20, 21, 22, 23, 24 저장한다.
    b.SetData(20);
    cout << "b : ";
    // b 객체에 저장된 값을 출력한다.
    b.ShowData();

    // a 객체를 생성하면 b 객체의 내용을 a 객체에 복사
    Tipsware a = b;
    // a 객체에 할당된 메모리에 10, 11, 12, 13, 14 저장한다.
    a.SetData(10);

    cout << "a : ";
    // a 객체에 저장된 값을 출력한다.
    a.ShowData();

    cout << endl << "공유 상태 확인" << endl << "b : ";
    // 메모리가 공유되는 것을 확인하기 위해 b 객체에 저장된 값을 출력한다.
    b.ShowData();

    return 0;
}

```

이제 복사 생성자가 추가되었기 때문에 Tipware a = b;와 같은 표현을 사용해도 값이 분리되어 관리되고 객체가 제거될 때 메모리 오류도 발생하지 않습니다.

```
b : 20, 21, 22, 23, 24,  
a : 10, 11, 12, 13, 14,
```


공유 상태 확인

```
b : 20, 21, 22, 23, 24,
```

이 창을 닫으려면 아무 키나 누르세요...



5. 결론



객체의 데이터를 직접 복사하는 개념을 사용할 때는 **대입 연산자 오버로딩 뿐만 아니라 복사 생성자까지 함께 사용**해야지 자신이 만든 클래스를 사용하는 코드에서 버그 발생 확률을 줄일 수 있습니다. 많은 개발자들이 이 문제를 알면서도 복사 생성자를 등록하지 않는 경우가 많은데, 객체를 복사할 수 있는 상황이 예상된다면 자신과 함께 작업하는 동료들을 위해 무조건 복사 생성자를 추가하는 습관을 가지는 것이 좋습니다.



 클린봇이 악성 댓글을 감지합니다.

 설정

댓글 등록순 최신순 



 관심글 댓글 알림 



 **a12345** 
드디어 영상올라왔군요..! 감사합니다!!
2022.08.15. 10:57 [답글쓰기](#)



 **김성엽**  작성자
이 내용 잘 몰라서 버그에 시달리는 분들 종종 있으니, 꼭 마스터하세요 ㅎㅎ



2022.08.15. 10:58 [답글쓰기](#)

 **a12345** 
김성엽 감사합니다 선생님 혹시 여유관참으시면
16회차 static const 강좌에 질문 답변좀 부탁드립니다 될까요?
2022.08.15. 11:05 [답글쓰기](#)

 **김성엽**  작성자
a12345 네~ 답변했습니다.
2022.08.15. 11:10 [답글쓰기](#)

 **해피파파** 
기본생성자, 내가 정의한 생성자, 복사생성자, 연산자오버로딩까지 장착.... 와~~ 시니어 개발자의 수고가 느껴집니다.
CopData() 함수내에 for문대신 memcpy(mp_data, ap_data->mp_data, sizeof(int) * m_count); 써도 될까요?



김성엽 ✎ 작성자

네~ memcpy 사용하셔도 됩니다~ :)



2022.11.10. 09:20 답글쓰기



황금잉어가물치 3

C언어 구조체 a에 동일한 구조체 형식을 가진 b에 대해 복사는 불가능한게 맞죠? 해당 복사 생성자는 C++에서만 작동되는 것이 맞는지요?

2022.11.24. 21:05 답글쓰기



김성엽 ✎ 작성자

ㅎㅎ 다시 한번 강의를 들어보시는걸 추천합니다. C 언어에서도 동일한 구조체에 복사가 가능한데, 동적할당된 메모리의 주소의 경우, 주소가 그대로 복사되어 할당된 메모리를 두 개의 변수가 함께 참조하는 문제가 발생하기 때문에 C++에서는 복사 생성자를 사용해서 이런 작업을 보완하려고 하는 것입니다~ :)

2022.11.24. 21:17 답글쓰기



황금잉어가물치 3

김성엽 감사합니다.

2022.11.24. 21:30 답글쓰기



카일 3

복사생성자 강의 잘 들었습니다~~~

2022.11.26. 18:02 답글쓰기



김성엽 ✎ 작성자

수고하셨습니다 ㅎㅎ



2022.11.26. 18:56 답글쓰기



황금잉어가물치 3

위에 복사생성자를 확인하기 위해 int clone함수를 사용하는데 clone=1이 대입된 상태에서 !clone을 하면 0으로 되어하는데 비주얼스튜디오 2022에서는 ~clone을 사용하라고 예러가 뜹니다. ~clone을 할 시 비트연산자로 인식해 1에서 -2로 변경되는 것인지요? !사용법과 ! 대신에 ~을 사용하는 것이 맞는지도 궁금합니다.

2022.12.21. 22:33 답글쓰기



김성엽 ✎ 작성자

녹색줄은 틀렸다가 보다는 의견을 제하셔는 것인데 지금 이야기하신 내용은 무시하시면 됩니다. 해당 상황에서는 ! 연산을 사용하는 것이 맞습니다.

2022.12.21. 22:45 답글쓰기



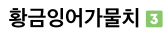
황금잉어가물치 3

여기서 궁금한점이 있습니다. b.p에 new로 메모리 할당 후 a=b로 같은 공간을 공유하게 만든 후 마지막에 new로 할당된 메모리 해제시 if (a.p != NULL) {delete[] a.p; a.p = NULL;}와 if (b.p != NULL) {delete[] b.p; b.p = NULL;}로 하면 예러가 발생하지 않아야 하는데 중복 해제가 뜨네요. a=b로 같은 공간을 공유하는데 a.p를 해제하고 a.p=NULL값을 입력해주면 b.p에 NULL값이 있어서 중복해제가 일어나지 않아야 하는데 일어나니 이상합니다. 어떤 부분이 잘못 됐는지 궁금합니다.

2022.12.31. 15:36 답글쓰기



2022.12.31. 15:59 답글쓰기



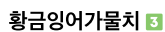
2022.12.31. 16:28 답글쓰기



2023.01.01. 01:07 답글쓰기



2023.01.01. 01:10 답글쓰기



2023.01.01. 01:11 답글쓰기

15/17