# C++ 강좌 21회 : 함수 포인터

김성엽 카페매니저 + 구독 1:1 채팅
2022.08.22. 02:47 조회 542



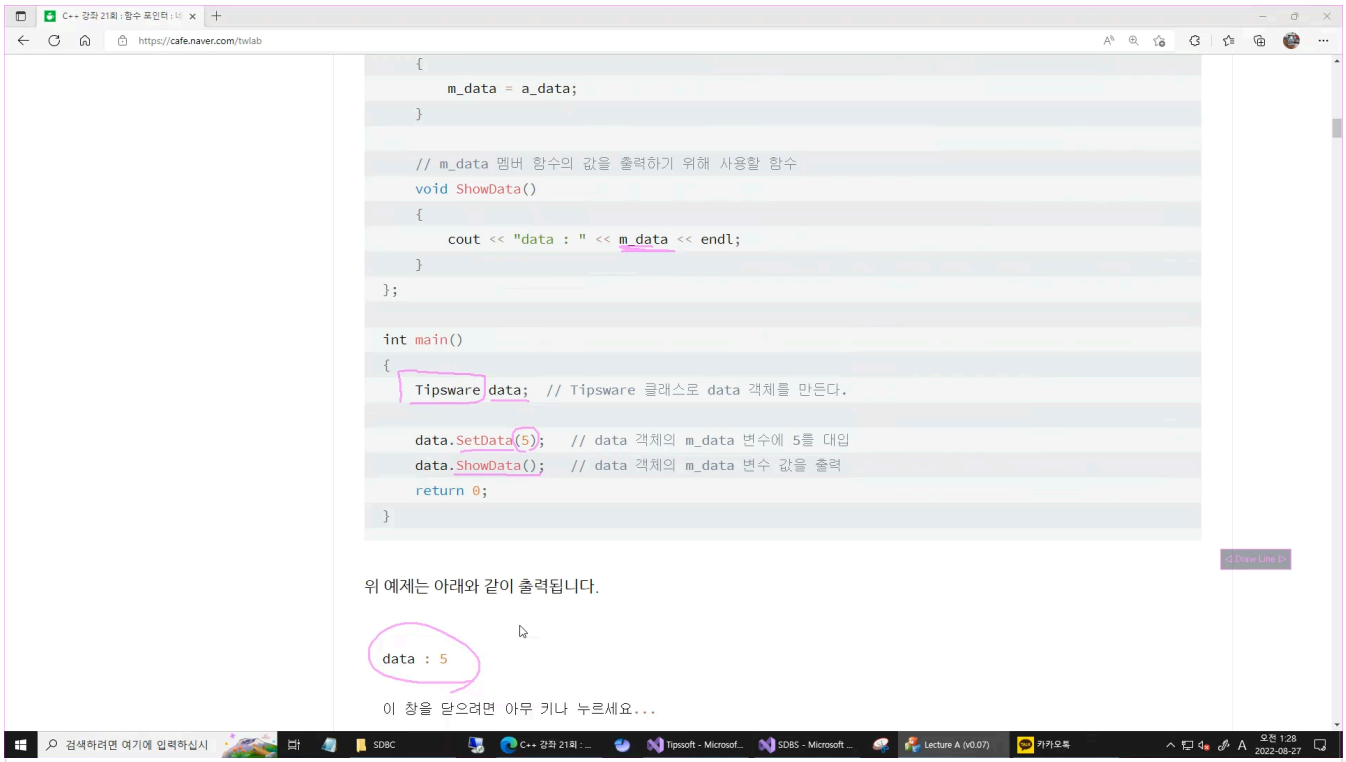

## [주의]

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.

이 강좌를 이해하려면 C 언어 기반의 함수 포인터를 먼저 공부해야 합니다. 따라서 아래에 링크한 강좌를 먼저 보고 충분히 함수 포인터를 공부한 다음 이 강좌를 보기 바랍니다.

**함수 포인터에 대하여**
: C 언어 관련 전체 목차 http://blog.naver.com/tipsware/221010831969 1. 데이터 포인터 C 언어를 배운...
blog.naver.com

이번 강좌에서는 함수를 이름이 아닌 주소를 사용해서 호출하는 함수 포인터(Function Pointer)에 대해 소개하겠습니다. C++ 언어가 제공하는 함수 포인터는 C 언어가 제공하는 함수 포인터와 매우 유사하지만 C++ 언어는 함수를 호출할 때 this 포인터를 위한 주소가 전달되어야 하기 때문에 포인터의 형식과 사용법에 약간 차이가 있습니다.




21회차. 함수 포인터

# 1. C++ 언어에서 제공하는 함수 포인터 형식

함수 포인터를 설명하기 위해 아래와 같이 정숫값을 저장하거나 저장된 값을 출력하는 Tipsware라는 클래스를 만들었습니다. 그리고 Tipsware 클래스를 사용해서 data 객체를 만든 다음 data 객체의 SetData 멤버 함수를 사용해서 5를 대입하고 다시 ShowData 함수를 사용해서 저장된 값을 출력하도록 예제를 구성했습니다.

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

class Tipsware
{
private:
    int m_data;  // 정숫값을 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_data = 0;
    }

    // m_data 멤버 함수에 값을 대입하기 위해 사용할 함수
    void SetData(int a_data)
    {
        m_data = a_data;
    }

    // m_data 멤버 함수의 값을 출력하기 위해 사용할 함수
    void ShowData()
    {
        cout << "data : " << m_data << endl;
    }
};

int main()
{
    Tipsware data;  // Tipsware 클래스로 data 객체를 만든다.

    data.SetData(5);    // data 객체의 m_data 변수에 5를 대입
    data.ShowData();    // data 객체의 m_data 변수 값을 출력
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
data : 5

이 창을 닫으려면 아무 키나 누르세요...
```

위 예제처럼 객체의 멤버 변수(m_data)의 값을 변경하거나 출력하기 위해서는 해당 객체가 제공하는 멤버 함수(SetData, GetData)를 호출해서 작업을 진행합니다. 예를 들어, data 객체에 5 값을 저장하고 싶다면 위와 같이 멤버 함수의 이름을 사용해서 data.SetData(5); 라고 호출합니다.

그런데 이렇게 객체가 제공하는 멤버 함수는 꼭 이름으로만 호출할 수 있는 것은 아닙니다. **해당 멤버 함수의 주소를 알고 있다면 이 주소를 사용해서도 얼마든지 멤버 함수를 호출**할 수 있습니다. 예를 들어, 위 예제에서 SetData 멤버 함수의 주소를

얻고 싶다면 아래와 같이 멤버 함수의 이름 앞에 & 연산자를 사용하면 됩니다. C++ 언어는 클래스 외부에서 특정 클래스의 멤버 함수를 지칭하려면 SetData가 아닌 '클래스이름::SetData' 형식을 사용해야 합니다. 따라서 Tipsware 클래스의 SetData 멤버 함수의 주소를 얻으려면 &Tipsware::SetData라고 해야 합니다.

```
&Tipsware::SetData
```

위 표현이 Tipsware 클래스의 SetData 멤버 함수의 주소를 의미하기 때문에, 이 주소에 * 연산자를 사용하면 해당 함수를 호출하는 것이 가능할 것이라고 생각할 수 있습니다. C 언어는 호출할 함수의 주소만 알고 있다면 해당 함수를 호출하는 것이 가능하지만 C++ 언어는 아래와 같이 코드를 사용하면 오류 처리됩니다. 왜냐하면 C++ 언어는 함수를 호출할 때 this 포인터에 주소를 전달하기 위해 해당 함수를 소유하고 있는 객체의 주소가 필요한데 아래의 코드에서는 특정 객체의 주소를 알아 낼수 있는 코드가 포함되어 있지 않습니다.

```
(*(&Tipsware::SetData))(3);  // 오류 처리 됨
```

따라서 위 코드의 오류를 해결하려면 아래와 같이 SetData 멤버 함수를 호출할 객체를 앞쪽에 추가해야 합니다.

```
Tipsware data;  // Tipsware 클래스로 data 객체를 만든다.
(data.*(&Tipsware::SetData))(3);  // 정상적으로 SetData 함수가 호출되어 객체에 3이 저장됩니다.
```

그런데 매번 함수의 주소를 사용할 때마다 &Tipsware::SetData라고 사용할 수는 없기 때문에, 이 주소를 포인터 변수에 저장해서 사용하는데 이 때 사용하는 포인터를 함수 포인터(function pointer)라고 합니다. 함수 포인터를 선언하기 위해서는 해당 포인터 변수에 저장될 멤버 함수의 원형을 알아야 하는데 SetData 함수의 원형은

```
void SetData(int a_data);
```

가 아니라 Tipsware 클래스의 멤버 함수이기 때문에 아래와 같습니다.

```
void Tipsware::SetData(int a_data);
```

따라서 이 함수의 주소에 대한 자료형은 다음과 같습니다. 즉, 함수 원형에서 함수 이름은 빼고 * 만 추가해 주면 됩니다.

```
void (Tipsware::*)(int);  // 매개 변수 이름인 a_data는 생략 가능합니다.
```

그리고 위 자료형을 사용해서 SetData 함수의 주소를 저장할 포인터 변수 p는 아래와 같이 선언하면 됩니다.

```
void (Tipsware::*p)(int);
```

그리고 위와같이 선언된 포인터 변수 p에 SetData 함수 주소를 대입하려면 아래와 같이 코드를 구성하면 됩니다.

```
void (Tipsware::*p)(int);
p = &Tipsware::SetData;
```

위 코드는 아래와 같이 초기화 문법을 사용해 한 줄로 사용하는 것도 가능합니다.

```
void (Tipsware::*p)(int) = &Tipsware::SetData;
```

그리고 포인터 변수 p에 저장된 주소를 사용해 SetData 함수를 호출하는 코드는 아래와 같이 구성하면 됩니다. C++ 언어는 멤버 함수를 호출할 때 해당 함수를 소유한 객체의 주소가 필요하기 때문에 'data.' 코드가 반드시 추가되어야 합니다.

```
Tipsware data;  // data 객체를 만든다.
void (Tipsware::*p)(int) = &Tipsware::SetData;  // 포인터 변수 p를 선언하고 SetData 함수의 주소를 저장한다.
(data.*p)(3);  // 포인터 변수 p를 사용해서 SetData 함수를 호출하여 data 객체에 3을 저장한다.
```

이제 위에서 설명한 내용을 실습할 수 있도록 아래와 같이 예제 코드를 구성했습니다.

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

class Tipsware
{
private:
    int m_data;  // 정숫값을 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_data = 0;
    }

    // m_data 멤버 함수에 값을 대입하기 위해 사용할 함수
    void SetData(int a_data)
    {
        m_data = a_data;
    }

    // m_data 멤버 함수의 값을 출력하기 위해 사용할 함수
    void ShowData()
    {
        cout << "data : " << m_data << endl;
    }
};

int main()
{
    Tipsware data;  // Tipsware 클래스로 data 객체를 만든다.

    data.SetData(5);    // data 객체의 m_data 변수에 5를 대입
    data.ShowData();    // data 객체의 m_data 변수 값을 출력

    // C++ 형식의 함수 포인터 p 변수에 SetData 함수의 주소를 저장한다.
    void (Tipsware::*p)(int) = &Tipsware::SetData;

    // 함수 포인터 p 변수에 저장된 주소를 사용해서 SetData 함수를 간접 호출하여
    // data 객체의 m_data 변수에 3을 대입한다.
    (data.*p)(3);
    data.ShowData();    // data 객체의 m_data 변수 값을 출력

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다. 5는 SetData 함수를 이름으로 호출해서 저장한 값이고 3은 SetData 함수를 주소로 호출해서 저장한 값입니다.

```
data : 5
data : 3

이 창을 닫으려면 아무 키나 누르세요...
```

그리고 C 언어에서는 함수 포인터를 사용할 때 (*p)(5);를 p(5); 와 같이 편하게 사용할 수 있었지만 C++ 언어에서는 'data.' 형식이 함께 사용하기 때문에 *를 생략하면 의미가 완전히 달라지게 됩니다. 예를 들어, (data.*p)(3)은 data.p(3); 라고 *를 생략해서 적으면 data 객체에서 p라는 이름을 가진 함수를 호출하겠다는 의미로 컴파일러가 해석해서 오류가 발생하기 때문에 주의해야 합니다.

## 2. 함수 포인터를 사용하는 간단한 예제를 만들어 봅시다.

아래와 같이 사칙 연산을 제공하는 OpManager 클래스를 정의하고 이 클래스로 ops 객체를 만듭니다. 그리고 7과 2값을 ops 객체가 제공하는 Add, Sub, Mul, Div 함수를 사용해서 연산한 결과를 출력하도록 예제를 구성했습니다.

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

// OpManager 클래스가 제공하는 연산의 개수
#define MAX_OP_COUNT    4

class OpManager
{
private:
    // 연산 목록을 문자로 구성한다.
    char m_op_table[MAX_OP_COUNT] = { '+', '-', '*', '/' };

public:
    // a_index 번째의 연산의 명칭을 문자로 반환하는 함수
    char GetOpName(int a_index)
    {
        return m_op_table[a_index];
    }

    int Add(int a, int b)  // 덧셈을 수행하는 함수
    {
        return a + b;
    }

    int Sub(int a, int b)  // 뺄셈을 수행하는 함수
    {
        return a - b;
    }

    int Mul(int a, int b)  // 곱셈을 수행하는 함수
    {
        return a * b;
    }

    int Div(int a, int b)  // 나눗셈을 수행하는 함수
    {
        return b ? a / b : 0;
    }
};

int main()
{
    OpManager ops;

    cout << "7과 2에 대한 연산 결과를 출력" << endl;

    // 현재 연산을 문자로 출력한다.
    cout << ops.GetOpName(0) << " : ";
    // 7과 2를 더한 결과를 출력한다.
    cout << ops.Add(7, 2) << endl;

    // 현재 연산을 문자로 출력한다.
```

```
    cout << ops.GetOpName(1) << " : ";
    // 7에서 2를 뺀 결과를 출력한다.
    cout << ops.Sub(7, 2) << endl;

    // 현재 연산을 문자로 출력한다.
    cout << ops.GetOpName(2) << " : ";
    // 7과 2를 곱한 결과를 출력한다.
    cout << ops.Mul(7, 2) << endl;

    // 현재 연산을 문자로 출력한다.
    cout << ops.GetOpName(3) << " : ";
    // 7을 2로 나눈 몫을 출력한다.
    cout << ops.Div(7, 2) << endl;

    cout << endl;
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
7과 2에 대한 연산 결과를 출력
+ : 9
- : 5
* : 14
/ : 3

이 창을 닫으려면 아무 키나 누르세요...
```

그런데 위 예제에서 main 함수의 코드를 보면 ops 객체가 제공하는 함수들이 유사한 형식으로 제공되었지만, 함수 이름이 모두 달라서 코드를 그룹지을수 없기 때문에 비슷한 코드가 나열되는 현상이 발생합니다. 따라서 이렇게 코드가 나열되기 시작하면 나중에 코드를 유지보수 할 때 관리가 어려워질 것입니다.

하지만 위 예제에서 Add, Sub, Mul, Div 함수가 모두 함수 원형이 동일하기 때문에 이 함수들의 주소에 대한 자료형도 동일합니다. 그리고 그 자료형은 아래와 같습니다.

```
int (OpManager::*)(int, int)
```

만약, 사칙 연산 함수 중에 한 개의 함수 주소만 저장한다면 아래와 같이 포인터 변수를 선언하면 됩니다.

```
int (OpManager::*p_func)(int, int);
```

그런데 여기서는 자료형이 동일한 함수의 주소 4개를 그룹 형태로 저장해야 해서 아래와 같이 배열 문법을 사용해서 선언해야 합니다. 왜냐하면 이렇게 그룹 메모리로 주소를 저장해야지 반복문을 사용해서 나열된 코드를 그룹지을 수 있습니다.

```
// OpManager 클래스가 제공하는 함수의 주소를 저장하기 위한 배열을 선언한다.
int (OpManager::*p_func_list[MAX_OP_COUNT])(int, int);
```

배열을 선언하면서 OpManager 클래스가 제공하는 4가지 함수에 대한 주소를 p_func_list 배열에 저장하고 싶다면 아래와 같이 코드를 구성하면 됩니다.

```
// OpManager 클래스가 제공하는 함수의 주소를 배열을 선언하고 저장한다.
int (OpManager::*p_func_list[MAX_OP_COUNT])(int, int) = {
    &OpManager::Add, &OpManager::Sub, &OpManager::Mul, &OpManager::Div
};
```

이렇게 배열로 함수의 주소를 구성하면 아래와 같이 main 함수에 나열되었던 코드가 반복문으로 정리됩니다.

```
cout << "7과 2에 대한 연산 결과를 출력" << endl;
// 4개의 연산을 순서대로 실행한다.
for (int i = 0; i < MAX_OP_COUNT; i++) {
    // 현재 연산을 문자로 출력한다.
    cout << ops.GetOpName(i) << " : ";
    // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
    cout << (ops.*p_func_list[i])(7, 2) << endl;
}
```

위에서 소개한 코드를 기존 예제에 반영해서 전체 코드를 보여드리면 다음과 같습니다.

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

// OpManager 클래스가 제공하는 연산의 개수
#define MAX_OP_COUNT    4

class OpManager
{
private:
    // 연산 목록을 문자로 구성한다.
    char m_op_table[MAX_OP_COUNT] = { '+', '-', '*', '/' };

public:
    // a_index 번째의 연산의 명칭을 문자로 반환하는 함수
    char GetOpName(int a_index)
    {
        return m_op_table[a_index];
    }

    int Add(int a, int b)  // 덧셈을 수행하는 함수
    {
        return a + b;
    }

    int Sub(int a, int b)  // 뺄셈을 수행하는 함수
    {
        return a - b;
    }

    int Mul(int a, int b)  // 곱셈을 수행하는 함수
    {
        return a * b;
    }

    int Div(int a, int b)  // 나눗셈을 수행하는 함수
    {
        return b ? a / b : 0;
    }
};

int main()
{
    OpManager ops;

    // OpManager 클래스가 제공하는 함수의 주소를 배열을 선언하고 저장한다.
    int (OpManager::*p_func_list[MAX_OP_COUNT])(int, int) = {
        &OpManager::Add, &OpManager::Sub, &OpManager::Mul, &OpManager::Div
    };

    cout << "7과 2에 대한 연산 결과를 출력" << endl;
    // 4개의 연산을 순서대로 실행한다.
    for (int i = 0; i < MAX_OP_COUNT; i++) {
```

```
        // 현재 연산을 문자로 출력한다.
        cout << ops.GetOpName(i) << " : ";
        // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
        cout << (ops.*p_func_list[i])(7, 2) << endl;
    }

    cout << endl;
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
7과 2에 대한 연산 결과를 출력
+ : 9
- : 5
* : 14
/ : 3

이 창을 닫으려면 아무 키나 누르세요...
```

그런데 함수 포인터를 OpManager 클래스를 사용하는 개발자가 직접 구성하는 것은 힘들고 불편한 일이기 때문에 OpManager 클래스를 제공하는 개발자가 아래와 같이 p_func_list 배열은 OpManager 클래스에 포함시켜서 제공하는 것이 더 좋습니다.

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

// OpManager 클래스가 제공하는 연산의 개수
#define MAX_OP_COUNT    4

class OpManager
{
private:
    // 합, 차, 곱, 그리고 나눗셈 함수의 주소를 저장할 배열을 선언한다.
    int (OpManager::*mp_func_list[MAX_OP_COUNT])(int, int);
    // 연산 목록을 문자로 구성한다.
    char m_op_table[MAX_OP_COUNT] = { '+', '-', '*', '/' };

public:
    OpManager()  // 생성자
    {
        // 함수의 주소를 배열에 순서대로 저장한다.
        mp_func_list[0] = &OpManager::Add;
        mp_func_list[1] = &OpManager::Sub;
        mp_func_list[2] = &OpManager::Mul;
        mp_func_list[3] = &OpManager::Div;
    }

    // a_index 번째의 연산의 명칭을 문자로 반환하는 함수
    char GetOpName(int a_index)
    {
        return m_op_table[a_index];
    }

    int Add(int a, int b)  // 덧셈을 수행하는 함수
    {
        return a + b;
    }

    int Sub(int a, int b)  // 뺄셈을 수행하는 함수
    {
        return a - b;
    }

    int Mul(int a, int b)  // 곱셈을 수행하는 함수
    {
        return a * b;
    }

    int Div(int a, int b)  // 나눗셈을 수행하는 함수
    {
        return b ? a / b : 0;
    }

    int ExecFunc(int a_index, int a_num1, int a_num2)  // 지정한 위치의 함수(주소)를 실행하는 함수
    {
```

```
        // a_index 위치에 저장된 함수의 주소를 사용해서 해당 함수를 호출한다.
        return (this->*mp_func_list[a_index])(a_num1, a_num2);
    }
};

int main()
{
    OpManager ops;

    cout << "7과 2에 대한 연산 결과를 출력" << endl;
    // 4개의 연산을 순서대로 실행한다.
    for (int i = 0; i < MAX_OP_COUNT; i++) {
        // 현재 연산을 문자로 출력한다.
        cout << ops.GetOpName(i) << " : ";
        // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
        cout << ops.ExecFunc(i, 7, 2) << endl;
    }

    cout << endl;
    return 0;
}
```

만약, ExecFunc 함수를 호출해서 mp_func_list 변수의 i번째 저장된 주소에 해당하는 함수를 호출하는 방식이 아닌 mp_func_list의 주소를 넘겨받아서 해당 함수를 실행하고 싶다면 아래와 같이 OpManager 클래스에 GetFuncList  함수를 추가하면 됩니다.

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

// OpManager 클래스가 제공하는 연산의 개수
#define MAX_OP_COUNT    4

class OpManager
{
private:
    // 합, 차, 곱, 그리고 나눗셈 함수의 주소를 저장할 배열을 선언한다.
    int (OpManager::*mp_func_list[MAX_OP_COUNT])(int, int);
    // 연산 목록을 문자로 구성한다.
    char m_op_table[MAX_OP_COUNT] = { '+', '-', '*', '/' };

public:
    OpManager()  // 생성자
    {
        // 함수의 주소를 배열에 순서대로 저장한다.
        mp_func_list[0] = &OpManager::Add;
        mp_func_list[1] = &OpManager::Sub;
        mp_func_list[2] = &OpManager::Mul;
        mp_func_list[3] = &OpManager::Div;
    }

    // a_index 번째의 연산의 명칭을 문자로 반환하는 함수
    char GetOpName(int a_index)
    {
        return m_op_table[a_index];
    }

    int Add(int a, int b)  // 덧셈을 수행하는 함수
    {
        return a + b;
    }

    int Sub(int a, int b)  // 뺄셈을 수행하는 함수
    {
        return a - b;
    }

    int Mul(int a, int b)  // 곱셈을 수행하는 함수
    {
        return a * b;
    }

    int Div(int a, int b)  // 나눗셈을 수행하는 함수
    {
        return b ? a / b : 0;
    }

    int ExecFunc(int a_index, int a_num1, int a_num2)  // 지정한 위치의 함수(주소)를 실행하는 함수
    {
```

```
        // a_index 위치에 저장된 함수의 주소를 사용해서 해당 함수를 호출한다.
        return (this->*mp_func_list[a_index])(a_num1, a_num2);
    }

    int (OpManager::**GetFuncList())(int, int) // 함수의 주소가 저장된 배열의 주소를 반환하는 함수
    {
        return mp_func_list;
    }
};

int main()
{
    OpManager ops;

    cout << "7과 2에 대한 연산 결과를 출력" << endl;
    // 4개의 연산을 순서대로 실행한다.
    for (int i = 0; i < MAX_OP_COUNT; i++) {
        // 현재 연산을 문자로 출력한다.
        cout << ops.GetOpName(i) << " : ";
        // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
        cout << (ops.**(ops.GetFuncList() + i))(7, 2) << endl;
    }

    cout << endl;
    return 0;
}
```

위 예제에서 GetFuncList 함수는 mp_func_list 배열의 주소를 반환하는데, 이 배열의 한 항목의 자료형이 int (OpManager::*)(int, int)이기 때문에 배열의 시작 주소는 int (OpManager::**)(int, int) 자료형을 가집니다. 따라서 GetFuncList 함수는 int (OpManager::**)(int, int) 형식의 값을 반환해야 하기 때문에 아래와 같이 함수를 구성해야 한다고 생각하는 분들도 있겠지만 이렇게 적으면 오류가 발생합니다.

```
int (OpManager::**)(int, int) GetFuncList()  // 오류 발생
{
    return mp_func_list;
}
```

그런데 이 형식은 조금만 생각해보면 이해할 수 있습니다. 왜냐하면 우리가 int (OpManager::**)(int, int) 자료형을 가진 포인터 변수를 int (OpManager::**)(int, int) ptr; 라고 선언하지 않고 int (OpManager::**ptr)(int, int); 라고 선언합니다. 따라서 GetFuncList 함수는 int (OpManager::**)(int, int) GetFuncList()가 아니라 아래와 같이 선언해야 합니다.

```
int (OpManager::**GetFuncList())(int, int)  // 올바르게 선언된 형태
{
    return mp_func_list;
}
```

만약, 위와 같은 표현이 어렵다면 아래와 같이 typedef 문법을 사용해서 FuncList 자료형을 추가하는 것도 방법입니다.

```
// FuncList 자료형을 선언할 때 사용하기 위해 OpManager 클래스가 아래에 있음을 알린다.
class OpManager;
// int (OpManager::*)(int, int) 자료형을 FuncList 자료형으로 치환한다.
// 이렇게 치환하면 FuncList p; 라고 적으면 int (OpManager::*p)(int, int); 라고 적은 것과 동일하다.
typedef int (OpManager::*FuncList)(int, int);
```

이렇게 하면 int (OpManager::*)(int, int) 라고 적어야 하는 표현을 FuncList라고 단순하게 적을 수 있기 때문에 GetFuncList 함수의 형식이 아래와 같이 단순해 집니다.

```
FuncList *GetFuncList()
{
    return mp_func_list;
}
```

그리고 위와 FuncList 자료형을 추가해서 사용하면 mp_func_list 배열 변수를 선언하는 형식도 아래와 같이 단순해 집니다.

```
// 합, 차, 곱, 그리고 나눗셈 함수의 주소를 저장할 배열을 선언한다.
// int (OpManager::*mp_func_list[MAX_OP_COUNT])(int, int); 라고 적은 것과 동일
FuncList mp_func_list[MAX_OP_COUNT];
```

이제 typedef 문법을 사용해서 정리된 전체 코드를 보면 다음과 같습니다.

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

// OpManager 클래스가 제공하는 연산의 개수
#define MAX_OP_COUNT    4

// FuncList 자료형을 선언할 때 사용하기 위해 OpManager 클래스가 아래에 있음을 알린다.
class OpManager;
// int (OpManager::*)(int, int) 자료형을 FuncList 자료형으로 치환한다.
// 이렇게 치환하면 FuncList p; 라고 적으면 int (OpManager::*p)(int, int); 라고 적은 것과 동일하다.
typedef int (OpManager::*FuncList)(int, int);

class OpManager
{
private:
    // 합, 차, 곱, 그리고 나눗셈 함수의 주소를 저장할 배열을 선언한다.
    // int (OpManager::*mp_func_list[MAX_OP_COUNT])(int, int); 라고 적은 것과 동일
    FuncList mp_func_list[MAX_OP_COUNT];
    // 연산 목록을 문자로 구성한다.
    char m_op_table[MAX_OP_COUNT] = { '+', '-', '*', '/' };

public:
    OpManager()  // 생성자
    {
        // 함수의 주소를 배열에 순서대로 저장한다.
        mp_func_list[0] = &OpManager::Add;
        mp_func_list[1] = &OpManager::Sub;
        mp_func_list[2] = &OpManager::Mul;
        mp_func_list[3] = &OpManager::Div;
    }

    // a_index 번째의 연산의 명칭을 문자로 반환하는 함수
    char GetOpName(int a_index)
    {
        return m_op_table[a_index];
    }

    int Add(int a, int b)  // 덧셈을 수행하는 함수
    {
        return a + b;
    }

    int Sub(int a, int b)  // 뺄셈을 수행하는 함수
    {
        return a - b;
    }

    int Mul(int a, int b)  // 곱셈을 수행하는 함수
    {
        return a * b;
    }
```

```
    int Div(int a, int b)  // 나눗셈을 수행하는 함수
    {
        return b ? a / b : 0;
    }

    int ExecFunc(int a_index, int a_num1, int a_num2)  // 지정한 위치의 함수(주소)를 실행하는 함수
    {
        // a_index 위치에 저장된 함수의 주소를 사용해서 해당 함수를 호출한다.
        return (this->*mp_func_list[a_index])(a_num1, a_num2);
    }

    FuncList *GetFuncList() // 함수의 주소가 저장된 배열의 주소를 반환하는 함수
    {
        return mp_func_list;
    }
};

int main()
{
    OpManager ops;

    cout << "7과 2에 대한 연산 결과를 출력" << endl;
    // 4개의 연산을 순서대로 실행한다.
    for (int i = 0; i < MAX_OP_COUNT; i++) {
        // 현재 연산을 문자로 출력한다.
        cout << ops.GetOpName(i) << " : ";
        // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
        cout << (ops.**(ops.GetFuncList() + i))(7, 2) << endl;
    }

    cout << endl;
    return 0;
}
```

위 예제에서 사용된 (ops.**(ops.GetFuncList() + i))(7, 2) 코드는 아래와 같이 배열 표현을 사용해서 적어도 됩니다.

```
cout << (ops.*ops.GetFuncList()[i])(7, 2) << endl;
```

그런데 위와 같이 적어보면 알겠지만 코드는 단순해졌지만 'ops.' 코드가 두 번 연속으로 나와서 코드를 이해하기가 더 어려워질 수도 있으니 잘 선택해서 사용해야 합니다. 그리고 'ops.' 코드가 두 번 연속으로 나오는 것도 마음에 걸리지만 GetFuncList 함수를 매번 호출하는 것도 비효율적이기 때문에 사실 아래와 같이 포인터 변수를 추가로 선언해서 사용하는 것이 더 좋은 방법입니다. 이렇게 하면 함수 포인터를 사용하는 코드가 단순해져서 이해하기가 더 편합니다.

```
// ... 나머지 소스는 동일해서 main 함수만 다시 적었습니다 ...
int main()
{
    OpManager ops;

    // 객체가 제공하는 사칙 연산 함수를 순서대로 저장한 배열의 주소를 얻는다.
    FuncList *p = ops.GetFuncList();
    cout << "7과 2에 대한 연산 결과를 출력" << endl;
    // 4개의 연산을 순서대로 실행한다.
    for (int i = 0; i < MAX_OP_COUNT; i++, p++) {
        // 현재 연산을 문자로 출력한다.
        cout << ops.GetOpName(i) << " : ";
        // 함수 포인터로 함수를 호출해서 결과 값을 출력한다.
        cout << (ops.**p)(7, 2) << endl;
    }

    cout << endl;
    return 0;
}
```

## 3. 결론

C++ 언어가 제공하는 함수 포인터는 this 포인터 구성은 위해 객체의 자료형과 주소가 함께 사용되기 때문에 C 언어가 제공하는 함수 포인터보다 좀더 복잡한 형식을 가집니다. 그래서 C, C++ 언어를 모두 사용하는 개발자들은 C 언어가 제공하는 함수의 포인터를 더 선호하는 경향이 있습니다. 물론 다형성으로 클래스 구조가 설계된 경우에는 C 언어 형식의 함수 포인터를 사용하면 매개 변수 표현이 복잡해지기 때문에 C++ 언어 형식의 함수 포인터를 사용하는 것이 좋습니다.

**김성엽** 작성자

ㅎㅎㅎ 감사합니다!


2022.08.23. 18:14 답글쓰기

**a12345**

안녕하세요 선생님 늘 좋은 강의 감사드립니다.

C언어에서처럼 배열의 이름은 배열의 (시작) 주소와 같듯이

함수의 이름은 함수의 주소를 나타내는 것 처럼

함수의 주소를 쓰고 싶을 때

&연산자 없이 함수이름만 쓰는것은 불가능한가요?

2022.08.24. 12:17 답글쓰기

**김성엽** 작성자

C++ 언어는 C 언어보다 문법 구조가 복잡해서 불필요한 모호성이 나오는 경우를 막기 위해서 & 생략을 허용하지 않습니다.

2022.08.24. 12:31 답글쓰기

**a12345**

**김성엽** 아아 그렇군요 감사합니다!!

2022.08.24. 12:34 답글쓰기

**김성엽** 작성자

**a12345** 이제 동영상 강좌 녹화도 완료되어 등록되었습니다. 이미 글로 보셨겠지만 그래도 시간날때 동영상 강좌 꼭 보세요 ㅎㅎ


2022.08.27. 02:50 답글쓰기

**a12345**

**김성엽** 넵 감사드립니다!!

2022.08.27. 09:25 답글쓰기

**조민희**

좋은 C++ 강의 감사드립니다 :)
계속 반복해서 들어야겠습니다 !

2022.09.04. 19:07 답글쓰기

**김성엽** 작성자

오~ 완강하셨군요!
볼만하죠? 시간날 때마다 반복해서 여러번 보세요~ :)


2022.09.04. 19:34 답글쓰기

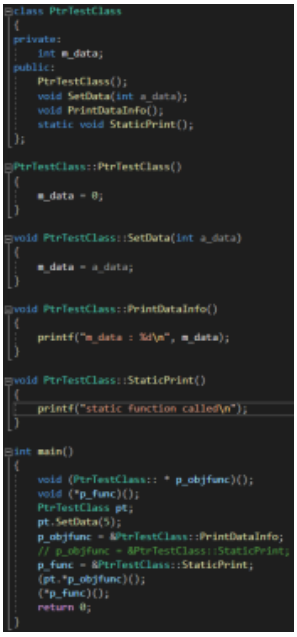
**bac**

질문 있습니다.

1. static 멤버 함수의 주소를 함수 포인터에 저장할 때 C++ 형식의 함수 포인터에는 함수의 주소를 저장할 수 없고 C 언어 형식의 함수 포인터에만 함수의 주소를 저장할 수 있었는데 그 이유는 static 멤버 함수의 함수 원형에는 클래스 네임스페이스가 없기 때문인가요?

(객체를 생성하지 않고 static 멤버 함수를 호출할 때 static 멤버 함수의 이름 앞에 클래스 네임스페이스를 명시하고 첨부한 이미지에서처럼 static 멤버 함수를 클래스의 선언부 밖에 작성할 때 일반적인 멤버 함수들처럼 클래스 네임스페이스를 명시하는데다가 static 멤버 함수의 주소를 얻을 때에도 클래스 네임스페이스를 명시하기 때문에 static 멤버 함수의 함수 원형에도 클래스 네임스페이스가 있을 거라고 생각했는데, static 멤버 함수의 함수 원형에 클래스 네임스페이스가 없다면 일반적인 상황에서 static 멤버 함수에 붙이는 클래스 네임스페이스에는 다른 멤버 함수들 같은 클래스 네임스페이스로써의 의미가 아닌 다른 의미가 있는 건가요?)

2. static 멤버 함수를 함수 포인터로 호출할 때 클래스 네임스페이스를 붙이니까 오류가 발생하는데 이 상황에서 클래스 네임스페이스를 붙이면 안 되는 이유는 static 멤버 함수의 주소를 함수 포인터에 저장할 때 static 멤버 함수의 클래스 네임스페이스를 명시했기 때문인가요? 아니면 다른 이유가 있나요?



```
class PtrTestClass
{
private:
    int m_data;
public:
    PtrTestClass();
    void SetData(int a_data);
    void PrintDataInfo();
    static void StaticPrint();
};

PtrTestClass::PtrTestClass()
{
    m_data = 0;
}

void PtrTestClass::SetData(int a_data)
{
    m_data = a_data;
}

void PtrTestClass::PrintDataInfo()
{
    printf("m_data : %d\n", m_data);
}

void PtrTestClass::StaticPrint()
{
    printf("static function called\n");
}

int main()
{
    void (PtrTestClass:: * p_objfunc)();
    void (*p_func)();
    PtrTestClass pt;
    pt.SetData(5);
    p_objfunc = &PtrTestClass::PrintDataInfo;
    // p_objfunc = &PtrTestClass::StaticPrint;
    p_func = &PtrTestClass::StaticPrint;
    (pt.*p_objfunc)();
    (*p_func)();
    return 0;
}
```

2022.09.06. 17:19 답글쓰기

---

**김성엽** M 작성자

C++ 형식의 함수 포인터는 this 포인터를 전달하는 형식을 사용하는데 static 멤버 함수는 this 포인터를 처리하는 코드가 없어서 C++ 형식의 포인터를사용할 수 없는 것입니다. 그래서 static 멤버 함수는 this 포인터를 사용하지 않는 함수 포인터 형식을 사용해야하기 때문에 C 언어 형식의 함수 포인터를 사용하는 것입니다.

2022.09.06. 22:45 답글쓰기

---

**bac** 2

**김성엽** static 멤버 함수를 일반적인 방법으로 호출할 때에는 PtrTestClass::StaticPrint();처럼 클래스 네임스페이스를 명시하지만 static 멤버 함수를 함수 포인터로 호출할 때((*p_func)(); 부분)에는 클래스 네임스페이스를 명시하지 않는 이유가 무엇인가요?

2022.09.06. 23:48 답글쓰기

---

**김성엽** M 작성자

**bac** 위와 동일한 이유죠. this 포인터를 사용하지 않기 때문에 네임 스페이스가 필요없습니다. 즉, 해당 클래스와 관련된 멤버들을 사용하지 못하기 때문에 네임 스페이스가 있어도 의미가 없거든요.

2022.09.07. 00:04 답글쓰기

---

**bac** 2

**김성엽** 감사합니다

2022.09.07. 01:20 답글쓰기

---

**카일** 3

강의 잘 들었습니다^^

2022.11.28. 10:32 답글쓰기

---

**김성엽** M 작성자

수고하셨습니다

2022.11.27. 22:36 답글쓰기

**티모**

와 반복해야겠어요.. 강좌들을때는 끄덕하다가 혼자 하려니 이게 안되네요..

2022.11.29. 15:21 답글쓰기

**김성엽** 작성자

함수 포인터는 반복해서 많이 보셔야 합니다~ :)


2022.11.29. 15:27 답글쓰기

**황금잉어가물치**

C++강의를 다보고 어느 정도 이해했으며, 이 블로그의 win32강의를 보면 되는지요? 현재 기존 교수님의 유튜브에 "windows 데스크탑 프로그래밍"을 보는데 둘다 같은 내용인지요? 알려주시기 바랍니다.

2022.12.10. 15:50 답글쓰기

**김성엽** 작성자

다릅니다. 이 카페에 있는걸 보세요. 이 카페에 있는거 다보면 MFC로 넘어가시면 됩니다. 블로그에 있는건 심심할때 조금씩 교양처럼 보시면 됩니다. MFC 하면 다 중복되어서 나옵니다.

2022.12.10. 15:52 답글쓰기

**황금잉어가물치**

**김성엽** 감사합니다

2022.12.10. 17:24 답글쓰기

**조민희**

함수포인터와 다단계 포인터가 합쳐지니까 처음에는 이해가 안됐는데 반복하니까 이해가 되네요 ㅎㅎ
계속 반복해서 보겠습니다 강의 감사드립니다 !

2022.12.25. 22:57 답글쓰기

**김성엽** 작성자

반복해서 공부하는 것이 쉽지 않지만 그것을 실천하면 실력을 가장 빠르게 성장합니다!


2022.12.26. 00:49 답글쓰기

dh221009

댓글을 남겨보세요

등록