

C++ 언어 >

C++ 강좌 10회 : namespace와 scope 연산자



김성업 카페매니저

+ 구독

1:1 채팅

2022.05.27. 14:57 조회 306



댓글 10

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 식별자(ex 객체 이름, 함수 이름, ...)의 이름 중복 문제를 해결할 수 있는 namespace 문법과 식별자의 소속을 지정할 수 있는 scope 연산자에 대해 소개하겠습니다.

The screenshot shows a video player interface for a C++ lecture. The title is "C++ 강좌 10회 : namespace와 scope 연산자 (작업 중)". Below the title, there is a warning message in Korean: "[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." The video player includes a progress bar at the bottom showing 00:01 / 42:52. The interface also displays the user's profile information and a list of related videos on the left side.

C++ 강좌 10회 : namespace와 scope 연산자

1. 식별자의 이름 중복 문제

클래스, 함수, 변수 그리고 사용자 정의 자료형은 문법을 구성할 때 개발자가 원하는 이름을 적게 되어 있고 이 이름을 '식별자'라고 합니다. 혼자서 프로그램 개발을 하는 경우에는 자신이 모든 식별자의 이름을 결정하기 때문에 이름 중복 문제가 거의 발생하지 않습니다. 하지만 프로그램의 규모가 커져서 **여러 명의 개발자가 소스를 나누어서 작업하다 보면 개발자들 간에 정보 교환이 잘 안 되어 함수나 전역 변수의 이름이 중복되는 경우가 발생할 수 있습니다.**

아래의 그림처럼 같은 프로그램에 사용될 두 개의 소스 파일이 서로 다른 사람에 의해 작업이 되었는데 **우연하게 이름이 같은 전역 변수가 사용**되었다고 가정하겠습니다. 만약, 두 변수의 역할이 동일하고 자료형도 일치한다면 한 쪽 변수를 extern 처리해서 쉽게 해결이 되겠지만, **다른 의미로 사용되는 변수인데 이름이 같다면 둘 중에 한 변수의 이름을 변경하거나 static 키워드를 사용해서 다른 소스에 있는 동일한 이름의 변수와 중복되지 않게 처리**해야 합니다. 그리고 이때 변수 이름을 수정하는 방법을 선택하면 해당 변수가 사용된 곳에 가서 모두 이름을 수정해야 하기 때문에 의외로 작업량이 많아질 수도 있습니다. 그리고 이 문제는 변수 이름에만 발생하는 문제가 아니라 함수 이름에도 동일하게 발생하기 때문에 생각보다 중복될 확률이 높습니다.



그래도 소스 파일을 다 가지고 있는 경우라면 중복된 이름을 수정하고 작업을 진행하면 됩니다. 하지만 **서로 다른 곳에서 구매한 두 개의 라이브러리에서 이름 중복 문제가 발생하게 되면, 소스가 없어서 수정을 못하기 때문에 두 개의 라이브러리 중 한 개는 사용을 못하게 되는 경우도 있습니다.** 즉, 라이브러리 파일은 이미 컴파일된 상태이기 때문에 라이브러리에 포함된 함수의 이름을 수정하고 싶다면 이 라이브러리 소스에서 수정 작업을 진행해야 합니다. 그런데 라이브러리를 구매한다고 라이브러리 소스까지 주는 것은 아니기 때문에 소스가 없어서 수정이 불가능하다는 뜻입니다.

프로그램의 기능이 점점 복잡해지고 규모가 커질수록 이름 중복에 대한 문제는 심각해질 수밖에 없지만 C 언어는 이 문제에 대한 근본적인 해결책이 없었습니다. 그냥 프로그래머가 잘 계획을 수립해서 이름이 중복되지 않게 최선을 다하는 것이 유일한 해결책이었습니다.

예를 들어, 아래와 같이 찬길씨와 민수씨가 함께 협력에서 프로그램을 하는데 두 사람이 별다른 규정을 사용하지 않고 작업하면 나중에 두 사람의 소스를 합쳤을 때 아래와 같이 전역 변수 이름과 함수 이름에 중복 오류가 발생할 수 있습니다.

```
// =====[ 찬길씨가 작업한 내용 ]=====
int g_data;    // 전역 변수

int CheckData()
{
    if(g_data < 0) g_data = 0;  // 음수면 0으로 변경!
    return g_data;
}

// =====[ 민수씨가 작업한 내용 ]=====
int g_data;    // 전역 변수

int CheckData()
{
    if(g_data < 0) g_data = g_data*(-1); // 음수면 양수로 변경!
    return g_data;
}
```

그래서 C 언어는 아래와 같이 작업자들이 함수나 변수 이름 앞에 개발자를 식별할 수 있는 접두어를 붙어서 사용하는 경우도 있었습니다. 여기서 **찬길씨는 Gil_**라는 접두어를 사용하고 **민수씨는 Su_**이라는 식별자를 사용했습니다. 이렇게 하면 찬길씨와 민수씨는 절대 식별자가 중복되는 문제가 발생하지 않습니다.

```
int Gil_g_data;    // 전역 변수

int Gil_CheckData()
{
    if(Gil_g_data < 0) Gil_g_data = 0;  // 음수면 0으로 변경!
    return Gil_g_data;
}

int Su_g_data;    // 전역 변수

int Su_CheckData()
{
    if(Su_g_data < 0) Su_g_data = Su_g_data*(-1); // 음수면 양수로 변경!
    return Su_g_data;
}
```

위 예제는 C 언어 개발자에 사용했던 한 가지의 예시일뿐 꼭 저렇게 작업하라는 뜻이 아닙니다.

2. namespace

식별자 이름에 'Gil_'이나 'Su_'과 같은 식별자를 사용하는 방법은 **식별자 이름 중복 문제는 회피할 수는 있지만 식별자의 이름과 상관없는 접두어가 식별자 앞에 붙기 때문에 소스 보기가 불편해질 것입니다.** 그래서 C++ 언어는 이런 접두어를 사용하는 방법을 문법적으로 재구성해서 아래와 같이 **접두어를 영역으로 표시하게 만들었는데 이 문법이 namespace** 입니다. 즉,

접두어를 식별자 이름 앞에 매번 적는 것이 아니라 중괄호 블록으로 접두어 영역을 표시해 놓고 해당 영역에 식별자를 선언하는 방식을 사용한 것입니다.

```
namespace Gil
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = 0;    // 음수면 0으로 변경!
        return g_data;
    }
}

namespace Su
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = g_data*(-1); // 음수면 양수로 변경!
        return g_data;
    }
}
```

위와 같이 namespace 문법을 사용해서 작업 영역을 구분 짓게 되면 식별자의 이름이 동일하더라도 Gil과 Su 이름으로 구분이 가능하기 때문에 오류가 발생하지 않습니다.

3. scope 연산자

C++ 언어는 식별자의 소유 관계를 지정할 수 있도록 scope 연산자(::)를 제공합니다. 예를 들어, namespace 내에 선언된 g_data 변수와 CheckData 함수는 Gil 또는 Su 소유이기 때문에 이 함수를 사용하기 위해서는 scope 연산자를 사용해서 아래와 같이 적어야 합니다.

```
Gil::g_data = 5;    // Gil 영역에 있는 g_data 변수에 5를 대입한다.
Su::g_data = 5;    // Su 영역에 있는 g_data 변수에 5를 대입한다.

int num = Gil::CheckData();    // Gil 영역에 있는 CheckData 함수를 호출한다.
int temp = Su::CheckData();    // Su 영역에 있는 CheckData 함수를 호출한다.
```

그리고 아래와 같이 scope 연산자를 사용하지 않고 CheckData();로 적으면 소유 관계를 적지 않았기 때문에 ::CheckData(); 라고 적은 것으로 처리됩니다. 그런데 이 소스에서는 namespace 영역 밖에 CheckData라는 이름으로 만들어진 함수가 없기 때문에 CheckData(); 라고 적은 코드는 문법 오류로 처리됩니다. 그리고 이 상황은 Gil과 Su에 각각 CheckData 함수가 있어서 그런 것이 아니라 소유 관련 문제이기 때문에 namespace Gil만 선언되어 있어도 동일하게 오류가 발생합니다.

```

namespace Gil
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = 0;    // 음수면 0으로 변경!
        return g_data;
    }
}

namespace Su
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = g_data*(-1); // 음수면 양수로 변경!
        return g_data;
    }
}

int main()
{
    int num = CheckData(); // (오류 발생) int num = ::CheckData(); 라고 적은 것과 동일!
    return 0;
}

```

4. using namespace

namespace 문법은 이름 중복에 대한 확실한 대안이긴 하지만 namespace에 포함된 함수를 호출하기 위해서는 namespace 이름과 :: 연산자를 매번 함께 적어야 하는 불편함이 있습니다. 그래서 **자주 사용하는 namespace를 생략할 수 있도록 해주는 'using namespace'라는 문법이 제공**됩니다. 예를 들어, Gil namespace에 포함된 CheckData 함수를 많이 사용한다면 아래와 같이 'using namespace'를 사용하여 함수 호출 시에 Gil:: 을 생략할 수 있습니다. 이때, Su 영역은 unusing namespace를 사용하지 않았기 때문에 Su 영역에 있는 CheckData 함수를 호출하려면 기존 방식처럼 Su::CheckData()라고 사용해야 합니다.

```

namespace Gil
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = 0;    // 음수면 0으로 변경!
        return g_data;
    }
}

namespace Su
{
    int g_data;    // 전역 변수

    int CheckData()
    {
        if(g_data < 0) g_data = g_data*(-1); // 음수면 양수로 변경!
        return g_data;
    }
}

using namespace Gil;    // Gil 이름을 기본으로 사용하도록 지정

int main()
{
    int num = CheckData();    // Gil::CheckData() 함수가 사용됨!
    int temp = Su::CheckData();    // Su 영역은 직접 지정해야 함!
    return 0;
}

```

그런데 아래와 같이 동일한 이름의 함수를 가진 두 namespace에 대해 모두 'using namespace'를 사용하지 않은 상태에서 namespace::를 생략하게 되면 C++ 컴파일러는 어떤 namespace에 소속된 CheckData 함수를 호출해야 할지 몰라 이 코드를 오류로 처리합니다.

```

using namespace Gil;
using namespace Su;

int main()
{
    // [오류] Gil::CheckData() Su::CheckData() 중에 어떤 것인지 판단할 수 없음!
    int num = CheckData();
    return 0;
}

```

위 코드에 발생한 오류를 해결하고 싶다면 Gil, Su namespace에 using namespace를 사용했다고 해도 아래와 같이 원하는 namespace를 생략하지 않고 그대로 적어야 합니다.

```
using namespace Gil;
using namespace Su;

int main()
{
    int num = Su::CheckData(); // namespace 생략이 불가능 함!
    return 0;
}
```

5. 전역 변수와 scope 연산자

C 언어에서는 아래와 같이 코드를 구성했을 때, 전역 변수나 지역 변수의 이름을 변경하지 않고서는 전역 변수로 선언된 data 변수에 값을 대입할 수 있는 방법이 없었습니다. 왜냐하면 **전역 변수와 지역 변수가 이름이 동일한 경우 함수에 선언된 지역 변수의 사용 우선 순위가 더 높기 때문**입니다.

```
int data; // 전역 변수

int main()
{
    int data; // 지역 변수

    data = 5; // main 함수에 선언된 지역 변수 data에 5가 대입됩니다.
    return 0;
}
```

하지만 C++ 언어에서는 scope 연산자가 제공되어 아래와 같이 **namespace 이름 없이 scope 연산자만 사용하면 특정 namespace에 소속되지 않고 전역으로 선언된 변수를 의미**하기 때문에 전역 변수 data에 5가 저장됩니다.

```
int data; // 전역 변수

int main()
{
    int data; // 지역 변수

    ::data = 5; // 전역 변수 data에 5가 대입됩니다.
    return 0;
}
```

6. class와 scope 연산자

class 문법도 함수나 멤버 변수를 내부에 가지고 있기 때문에 **class 이름은 namespace 이름과 유사**하다고 생각해도 됩니다. 따라서 class 이름에도 scope 연산자를 사용해서 소유 관계를 표시할 수 있습니다. 예를 들어, 지금까지 우리는 class 문법으로 객체를 정의하면서 아래와 같이 class 문법에 직접 변수나 함수를 포함시켜서 적었습니다. 그런데 이렇게 **모든 정보를 class 문법 내부에 적으면 class 구현 코드가 너무 길어져서 클래스를 파악하기가 어려워 집니다.**

```
class Person
{
private:
    int m_age;    // 멤버 변수 선언

public:
    Person()    // 기본 객체 생성자
    {
        m_age = 1;
    }

    Person(int a_age)    // 추가로 선언된 객체 생성자
    {
        m_age = a_age;
    }

    int SetAge(int a_age)    // 멤버 함수. 객체 외부에서 m_age에 값을 설정 할때 사용
    {
        if(a_age >= 0 && a_age <= 150) m_age = a_age;
        else {
            printf("잘못된 나이를 사용했습니다!\n");
            return 0;
        }
        return 1;
    }

    int GetAge()    // 멤버 함수, 객체 외부에서 m_age 값을 얻고 싶을 때 사용
    {
        return m_age;
    }
};
```

그래서 class 내용을 쉽게 파악할 수 있도록 아래와 같이 **class 선언 코드와 class 구현 코드를 구분**해서 적습니다. 이때 class의 멤버 함수 구현부를 class 문법 밖에 추가하고 싶다면, 해당 **멤버 함수 이름 앞에 Person 클래스에 소속되었음을 의미하는 Person::을 붙여주면 됩니다.** 좀더 정확하게 이야기 하면 함수의 반환 자료형과 함수 이름 사이에 Person::을 적으면 됩니다.


```

class Person
{
private:
    int m_age;    // 멤버 변수 선언

public:
    Person();    // 기본 객체 생성자
    Person(int a_age);    // 추가로 선언된 객체 생성자

    int SetAge(int a_age);    // 멤버 함수. 객체 외부에서 m_age에 값을 설정 할때 사용
    int GetAge();    // 멤버 함수, 객체 외부에서 m_age 값을 얻고 싶을 때 사용
};

Person::Person()    // 기본 객체 생성자
{
    m_age = 1;
}

Person::Person(int a_age)    // 추가로 선언된 객체 생성자
{
    m_age = a_age;
}

int Person::SetAge(int a_age)    // 멤버 함수. 객체 외부에서 m_age에 값을 설정 할때 사용
{
    if(a_age >= 0 && a_age <= 150) m_age = a_age;
    else {
        printf("잘못된 나이를 사용했습니다!\n");
        return 0;
    }
    return 1;
}

int Person::GetAge()    // 멤버 함수, 객체 외부에서 m_age 값을 얻고 싶을 때 사용
{
    return m_age;
}

```

그런데 위와 같이 작업하는 본질적인 이유는 단순히 class를 파악하기 편하게 하려는 목적보다 **class 정의를 헤더 파일과 소스 파일로 나누어서 관리**하기 위한 것입니다. 예를 들어, class 선언부는 아래와 같이 person.h 헤더 파일을 만들고 아래와 같이 추가합니다.

```
// person.h : Person 클래스의 선언부 입니다.

class Person
{
private:
    int m_age;    // 멤버 변수 선언

public:
    Person();    // 기본 객체 생성자
    Person(int a_age);    // 추가로 선언된 객체 생성자

    int SetAge(int a_age);    // 멤버 함수. 객체 외부에서 m_age에 값을 설정 할때 사용
    int GetAge();    // 멤버 함수, 객체 외부에서 m_age 값을 얻고 싶을 때 사용
};
```

그리고 class 구현부는 아래와 같이 person.cpp 파일을 만들고 아래와 같이 추가합니다.

```
// person.cpp : Person 클래스의 구현부 입니다.

Person::Person()    // 기본 객체 생성자
{
    m_age = 1;
}


Person::Person(int age)    // 추가로 선언된 객체 생성자
{
    m_age = age;
}

int Person::SetAge(int age)    // 멤버 함수. 객체 외부에서 m_age에 값을 설정 할때 사용
{
    if(age >= 0 && age <= 150) m_age = age;
    else {
        printf("잘못된 나이를 사용했습니다!\n");
        return 0;
    }
    return 1;
}


int Person::GetAge()    // 멤버 함수, 객체 외부에서 m_age 값을 얻고 싶을 때 사용
{
    return m_age;
}
```


이렇게 클래스를 헤더 파일과 소스 파일로 나누어서 관리하면 좋은 이유는 목적 파일(*.obj) 개념을 사용해서 [소스 컴파일에 사용되는 시간도 줄일 수도 있고](#) 이 목적 파일을 라이브러리로 변환해서 사용하면 [구현부를 다른 개발자에게 숨길 수 있어 중요한 소스를 보호](#)할 수도 있습니다. 즉, 다른 개발자와 함께 프로그램을 할 때, 다른 개발자에게 헤더 파일과 중요 구현 파일

을 모두 주는 것이 아니라, 헤더 파일과 라이브러리 파일(클래스 구현부 소스를 컴파일해서 만든)만 전달해도 이 클래스를 사용하는데는 문제가 없습니다. 따라서 함께 작업하면서도 중요 소스를 감추어 자신의 소스를 보호할 수 있습니다.

 클린봇이 악성 댓글을 감지합니다.

 설정

댓글 등록순 최신순 

 관심글 댓글 알림 ☐



박찬길 

앗제이름이 ㅎㅎ

2022.05.28. 06:59 답글쓰기

⋮



김성엽  작성자



2022.05.28. 11:32 답글쓰기



Zermi 

와-
업게 포상 ㄷㄷ

2022.09.11. 14:39 답글쓰기

⋮



김성엽  작성자

Zermi ㅎㅎㅎ



2022.09.11. 16:37 답글쓰기

⋮



티모 

엇 음 그렇군 하고 넘기려고했는데.. ㅎㅎ

2022.11.16. 21:20 답글쓰기

⋮




카일 

강의 잘 들었습니다. 복습하고 갑니다~

2022.11.25. 20:50 답글쓰기

⋮



모자꾸기 

헤더파일과 lib 파일을 주면 개발은 가능하되 내부 소스를 볼 수 없다는 말이 잘 이해가 안갑니다~ 혹시 자세하게 참고할 만한 다른 자료 있으실까요 ㅠㅠ

2024.02.04. 00:29 답글쓰기

⋮



김성엽  작성자

제 책(Do it! C언어 입문) 6장을 보시거나 동영상 강좌를 보시면 라이브러리가 왜 필요한지 설명이 나옵니다.
<https://blog.naver.com/tipsware/220884214074>

2024.02.04. 01:20 답글쓰기

⋮



김성엽  작성자

lib 파일이나 obj 파일은 C 언어 소스를 빌드해서 만들어지기 때문에 원본 소스가 없어도 해당 lib 파일이나 obj 파일이 있으면 원본 소스에 있던 함수들은 그대로 이용할 수 있습니다.

2024.02.04. 01:22 답글쓰기

⋮



김성엽  작성자

⋮



제가 링크한 자료에서 6장 첫번째 동영상 강좌 꼭 보세요~

2024.02.04. 01:23 답글쓰기

dh221009

댓글을 남겨보세요



답글
작성