

C++ 설명서

C++ 및 C++ 표준 라이브러리를 사용하는 방법을 알아봅니다.

Visual Studio의 C++ 알아보기

다운로드

[Windows용 Visual Studio 다운로드 ↗](#)

[Visual Studio에 C/C++ 지원 설치](#)

[명령줄 빌드 도구만 다운로드 ↗](#)

시작하기

[C++ 지원 Visual Studio의 Hello World](#)

[C++에서 콘솔 계산기 만들기](#)

VIDEO

[C++ 알아보기 - 범용 언어 및 라이브러리](#)

학습

[C++ 시작하기 - 최신 C++](#)

[샘플 및 샘플 보관](#)

Visual Studio의 새로운 C++ 기능

새로운 기능

[Visual Studio의 새로운 C++ 기능](#)

[C++ 규칙 향상](#)

개요

[Visual Studio의 C++ 개발 개요](#)

[지원되는 대상 플랫폼](#)

[도움말 및 커뮤니티 리소스](#)

[문제 보고 또는 제안하기](#)

컴파일러 및 도구 사용

 참조

[C/C++ 빌드 참조](#)

[프로젝트 및 빌드 시스템](#)

[컴파일러 참조](#)

[링커 참조](#)

[추가 빌드 도구](#)

[오류 및 경고](#)

C++ 언어

 참조

[C++ 언어 참조](#)

[C++ 키워드](#)

[C++ 연산자](#)

[C/C++ 전처리기 참조](#)

C++ 표준 라이브러리(STL)

 참조

[C++ 표준 라이브러리 개요](#)

[헤더별 C++ 표준 라이브러리 참조](#)

[C++ 표준 라이브러리 컨테이너](#)

[반복기](#)

[알고리즘](#)

할당자

함수 개체

iostream 프로그래밍

정규식

파일 시스템 탐색

C++ 언어 참조

아티클 • 2024. 11. 21.

이 참조는 Microsoft C++ 컴파일러에서 구현된 C++ 프로그래밍 언어에 대해 설명합니다. 조직은 Margaret Ellis 및 Bjarne Stroustrup의 주석이 추가된 C++ 참조 설명서와 ANSI/ISO C++ International Standard(ISO/IEC FDIS 14882)를 기반으로 합니다. C++ 언어 기능의 Microsoft 전용 구현이 포함되어 있습니다.

최신 C++ 프로그래밍 방법에 대한 개요는 [C++로 돌아가기를 참조하세요](#).

키워드 또는 연산자를 빠르게 찾으려면 다음 표를 참조하십시오.

- [C++ 키워드](#)
- [C++ 연산자](#)

섹션 내용

어휘 규칙

C++ 프로그램의 기본적인 어휘 요소에는 토큰, 주석, 연산자, 키워드, 문장 부호, 리터럴이 있습니다. 또한 파일 변환, 연산자 우선 순위/결합성이 있습니다.

기본 개념

범위, 링크, 프로그램 시작 및 종료, 스토리지 클래스 및 형식입니다.

[기본 제공 형식](#) C++ 컴파일러 및 해당 값 범위에 기본 제공되는 기본 형식입니다.

표준 변환

기본 제공 형식 간의 형식 변환입니다. 또한 산술 변환 및 포인터, 참조 및 멤버 포인터 형식 간의 변환입니다.

[변수, 형식 및 함수를 선언하고 정의하는 선언 및 정의](#) 입니다.

연산자, 우선 순위 및 결합성

C++의 연산자입니다.

식

식 형식 및 식 의미 체계, 연산자에 대한 참조 항목, 캐스팅 및 캐스팅 연산자, 런타임 형식 정보입니다.

람다 식

함수 객체 클래스를 암시적으로 정의하고 해당 클래스 형식의 함수 객체를 생성하는 프로그래밍 기술입니다.

문

식, null, 복합, 선택, 반복, 점프 및 선언문입니다.

클래스 및 구조체

클래스, 구조체 및 공용 구조체에 대한 소개입니다. 또한 멤버 함수, 특수 멤버 함수, 데이터 멤버, 비트 필드, `this` 포인터, 중첩 클래스도 있습니다.

공용 구조체

모든 멤버가 동일한 메모리 위치를 공유하는 사용자 정의 형식입니다.

파생 클래스

단일 및 다중 상속, `virtual` 함수, 여러 기본 클래스, **추상** 클래스, 범위 규칙. 또한 키워드 `super` 및 `interface` 키워드입니다.

멤버 액세스 제어

클래스 멤버에 대한 액세스 제어: `public`, `private` 및 `protected` 키워드. Friend 함수 및 클래스입니다.

오버 로드

오버로드된 연산자, 연산자 오버로드 규칙입니다.

예외 처리

C++ 예외 처리, SEH(구조적 예외 처리), 예외 처리 문을 작성하는 데 사용되는 키워드입니다.

어설션 및 사용자 제공 메시지

`#error` 지시문, `static_assert` 키워드, 매크로입니다 `assert`.

템플릿

템플릿 사양, 함수 템플릿, 클래스 템플릿, `typename` 키워드, 템플릿 및 매크로, 템플릿 및 스마트 포인터

이벤트 처리

이벤트 및 이벤트 처리기 선언입니다.

Microsoft 전용 한정자

Microsoft C++ 전용 한정자입니다. 메모리 주소 지정, 호출 규칙, `naked` 함수, 확장 스토리지 클래스 특성(`_declspec`), `_w64`.

인라인 어셈블러

블록에서 어셈블리 언어 및 C++를 `_asm` 사용합니다.

컴파일러 COM 지원

COM 형식을 지원하는 데 사용되는 Microsoft 전용 클래스 및 전역 함수에 대한 참조입니다.

다.

Microsoft 확장

C++에 대한 Microsoft 확장입니다.

비표준 동작

Microsoft C++ 컴파일러의 비표준 동작에 대한 정보입니다.

C++의 진화

안전하고 정확하며 효율적인 프로그램을 작성하기 위한 최신 C++ 프로그래밍 방법에 대한 개요입니다.

관련 섹션

런타임 플랫폼용 구성 요소 확장

Microsoft C++ 컴파일러를 사용하여 .NET을 대상으로 하는 참조 자료입니다.

C/C++ 빌드 참조

컴파일러 옵션, 링커 옵션 및 기타 빌드 도구입니다.

전처리기 참조

Pragma, 전처리기 지시문, 미리 정의된 매크로 및 전처리기에 대한 참조 자료입니다.

Visual C++ 라이브러리

다양한 Microsoft C++ 라이브러리에 대한 참조 시작 페이지에 대한 링크 목록입니다.

참고 항목

C 언어 참조

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

C++ 시작하기 - 최신 C++

아티클 • 2023. 04. 03.

C++는 만들어진 이후 전 세계에서 가장 널리 사용되는 프로그래밍 언어 중 하나가 되었습니다. 잘 작성된 C++ 프로그램은 빠르고 효율적입니다. C++ 언어는 다른 언어보다 유연성이 높습니다. 추상화 수준이 매우 높은 방식부터, 반대로 하드웨어 제어까지 폭넓게 사용할 수 있습니다. C++는 고도로 최적화된 표준 라이브러리를 제공합니다. 이를 통해 낮은 수준의 하드웨어 기능에 액세스하여 속도를 최대한으로 높이고 메모리 요구 사항을 최소화할 수 있습니다. C++는 게임, 장치 드라이버, HPC, 클라우드, 데스크톱, 임베디드 및 모바일 앱 등 거의 모든 종류의 프로그램을 만들 수 있습니다. 심지어 다른 프로그래밍 언어를 위한 라이브러리와 컴파일러도 C++로 작성됩니다.

C++의 본래 요구 사항 중 하나는 C 언어와의 역 호환성이었습니다. 따라서 C++에서는 원시 포인터, 배열, null 종료 문자열, 기타 기능을 통해 항상 C 스타일 프로그래밍이 가능했습니다. 이로 인해 성능이 향상될 수 있는 반면 버그 및 복잡성이 생성될 수도 있습니다. C++가 진화하면서 C 스타일 관용구를 사용할 필요성을 크게 줄이는 기능이 강조되었습니다. 이전 C 프로그래밍 기능은 필요할 때 여전히 존재합니다. 그러나 최신 C++ 코드에서는 더 적은 수의 해당 기능이 사용됩니다. 최신 C++ 코드는 보다 간단하고 안전하고 명쾌하면서도 속도는 가장 빠릅니다.

다음 섹션에서는 최신 C++의 주요 기능에 대한 개요를 제공합니다. 별도로 언급하지 않는 한 여기에 나열된 기능은 C++11 이상에서 사용할 수 있습니다. Microsoft C++ 컴파일러에서는 `/std` 컴파일러 옵션을 설정하여 프로젝트에 사용할 표준 버전을 지정할 수 있습니다.

리소스 및 스마트 포인터

C 스타일 프로그래밍의 주요 버그 클래스 중 하나는 메모리 누수입니다. 메모리 누수의 흔한 원인은 `new` 를 사용하여 할당된 메모리의 `delete` 호출 오류입니다. 최신 C++는 획득된 자원의 초기화(RAI, Resource Acquisition Is Initialization) 원칙을 강조합니다. 개념은 간단합니다. 리소스(힙 메모리, 파일 핸들, 소켓 등)는 개체에 의해 소유되어야 합니다. 이 개체는 해당 생성자에서 새로 할당된 리소스를 만들거나 받아 해당 소멸자에서 삭제합니다. RAI 원칙은 소유하는 개체가 범위를 벗어나면 모든 리소스가 운영 체제에 제대로 반환되도록 보장합니다.

RAI 원칙을 쉽게 채택할 수 있도록 C++ 표준 라이브러리는 `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`의 세 가지 스마트 포인터 형식을 제공합니다. 스마트 포인터는 소유하고 있는 메모리의 할당 및 삭제를 처리합니다. 다음 예제는 `make_unique()`에 대한 호출에서 힙에 할당된 배열 멤버가 있는 클래스를 보여 줍니다. `new` 및 `delete`에 대

한 호출은 `unique_ptr` 클래스에 의해 캡슐화됩니다. `widget` 개체가 범위를 벗어나면 `unique_ptr` 소멸자가 호출되어 배열을 위해 할당된 메모리를 해제합니다.

C++

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                      // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

가능하면 스마트 포인터를 사용하여 힙 메모리를 관리합니다. 명시적으로 및 `delete` 연산자를 `new` 사용해야 하는 경우 RAII 원칙을 따릅니다. 자세한 내용은 [개체 수명 및 리소스 관리\(RAII\)](#)를 참조하세요.

std::string 및 std::string_view

C 스타일 문자열은 버그의 또 다른 주요 원인입니다. `std::string` 및 `std::wstring`를 사용하면 C 스타일 문자열과 관련된 거의 모든 오류를 제거할 수 있습니다. 또한 검색, 추가, 앞에 추가 등에서 멤버 함수의 이점을 얻을 수 있습니다. 두 가지 모두 속도에 고도로 최적화되어 있습니다. 읽기 전용 액세스만 필요한 함수에 문자열을 전달하는 경우 C++17에서는 `std::string_view`를 사용하여 훨씬 큰 성능상 이점을 얻을 수 있습니다.

std::vector 및 기타 표준 라이브러리 컨테이너

표준 라이브러리 컨테이너는 모두 RAII 원칙을 따르며 요소의 안전한 탐색을 위한 반복기를 제공합니다. 또한 성능에 고도로 최적화되어 있으며, 정확성을 철저하게 테스트했습니다. 이 같은 컨테이너를 사용하면 사용자 지정 데이터 구조에 유입될 수 있는 버그 또는 비효율성의 가능성을 없앨 수 있습니다. C++에서 원시 배열 대신 `vector`를 순차 컨테이너로 사용하세요.

C++

```
vector<string> apples;
apples.push_back("Granny Smith");
```

`map`(`unordered_map` 아님)을 기본 연관 컨테이너로 사용하세요. 중복 제거 및 다중 케이스에는 `set`, `multimap`, `multiset`를 사용하세요.

C++

```
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

성능 최적화가 필요한 경우 다음을 사용하는 것이 좋습니다.

- 예를 들어 클래스 멤버로서 포함이 중요한 경우 `array` 형식.
- `unordered_map`과 같이 순서가 지정되지 않은 연관 컨테이너. 이 컨테이너에는 보다 낮은 요소당 오버헤드와 상수 시간 조회가 있지만 올바르고 효율적으로 사용하기는 더 어렵습니다.
- 정렬 `vector`. 자세한 내용은 [알고리즘](#)을 참조하세요.

C 스타일 배열을 사용하지 마세요. 직접 데이터 액세스가 필요한 이전 API의 경우 `f(vec.data(), vec.size())`; 와 같은 접근자 메서드를 대신 사용합니다. 컨테이너에 대한 자세한 내용은 [C++ 표준 라이브러리 컨테이너](#)를 참조하세요.

표준 라이브러리 알고리즘

프로그램을 위한 사용자 지정 알고리즘 작성이 필요하다고 가정하기 전에 먼저 C++ 표준 라이브러리 [알고리즘](#)을 검토하세요. 표준 라이브러리에는 검색, 정렬, 필터링, 무작위화 등 여러 일반적 작업을 위해 계속 늘어나는 알고리즘 모음이 포함되어 있습니다. 수식 라이브러리는 광범위합니다. C++17 이상에서는 많은 알고리즘의 병렬 버전이 제공됩니다.

몇 가지 중요한 예는 다음과 같습니다.

- 기본 탐색 알고리즘인 `for_each`(범위 기반 `for` 루프와 함께 사용).
- 컨테이너 요소의 not-in-place 수정을 위한 `transform`.
- 기본 검색 알고리즘인 `find_if`.
- `sort`, `lower_bound`, 기타 기본 정렬 및 검색 알고리즘.

비교자를 작성하려면 `strict <` 를 사용하고 가능한 경우 `명명된 람다`를 사용합니다.

C++

```
auto comp = [] (const widget& w1, const widget& w2)
{ return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), widget{0}, comp );
```

명시적 형식 이름 대신 `auto`

C++11에서는 변수, 함수, 템플릿 선언에 사용할 `auto` 키워드가 도입되었습니다. `auto` 가 개체의 형식을 추론하도록 컴파일러에 지시하므로 명시적으로 입력할 필요가 없습니다. `auto` 는 추론된 형식이 중첩된 템플릿인 경우 특히 유용합니다.

C++

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

범위 기반 `for` 루프

배열 및 컨테이너에 대한 C 스타일 반복은 인덱싱 오류가 발생하기 쉬우며 입력하기도 번거롭습니다. 이러한 오류를 제거하고 코드를 더 읽기 쉽게 만들려면 표준 라이브러리 컨테이너 및 원시 배열과 함께 범위 기반 `for` 루프를 사용하세요. 자세한 내용은 [범위 기반 for 문](#)을 참조하세요.

C++

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
```

```
}
```

매크로 대신 `constexpr` 식

C와 C++의 매크로는 컴파일 전에 전처리기에 의해 처리되는 토큰입니다. 매크로 토큰의 각 인스턴스는 파일이 컴파일되기 전에 정의된 값 또는 식으로 교체됩니다. 매크로는 일반적으로 C 스타일 프로그래밍에서 컴파일 시간 상수 값을 정의하는 데 사용됩니다. 그러나 매크로는 오류가 발생하기 쉬우며 디버그하기 어렵습니다. 최신 C++에서는 컴파일 시간 상수에 `constexpr` 변수를 사용하는 것이 좋습니다.

C++

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

균일한 초기화

최신 C++에서는 모든 형식에 중괄호 초기화를 사용할 수 있습니다. 이러한 형태의 초기화는 배열, 벡터 또는 기타 컨테이너를 초기화할 때 특히 편리합니다. 다음 예제에서는 `s` 인스턴스 세 개를 사용하여 `v2` 가 초기화됩니다. `v3` 는 중괄호를 사용하여 초기화되는 `s` 인스턴스 세 개를 사용하여 초기화됩니다. 컴파일러는 `v3` 의 선언된 형식을 기반으로 각 요소의 형식을 추론합니다.

C++

```
#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}

};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
```

```

v.push_back(s3);

// Modern C++:
std::vector<S> v2 {s1, s2, s3};

// or...
std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}

```

자세한 내용은 [중괄호 초기화](#)를 참조하세요.

이동 의미 체계

최신 C++는 불필요한 메모리 복사본을 제거할 수 있도록 이동 의미 체계를 제공합니다. 이전 버전의 C++에서는 특정 상황에서 복사본이 불가피했습니다. 이동 작업은 복사를 수행하지 않고 한 개체에서 다음 개체로 리소스 소유권을 전송합니다. 일부 클래스는 힙 메모리, 파일 핸들 등의 리소스를 소유합니다. 리소스 소유 클래스를 구현할 때 이동 생성자와 해당 이동 대입 연산자를 정의할 수 있습니다. 컴파일러는 복사본이 필요하지 않은 상황에서 오버로드 확인 중에 이러한 특수 멤버를 선택합니다. 표준 라이브러리 컨테이너 형식은 개체에 대해 이동 생성자를 호출합니다(정의된 경우). 자세한 내용은 [이동 생성자 및 이동 대입 연산자\(C++\)](#)를 참조하세요.

람다 식

C 스타일 프로그래밍에서는 함수 포인터를 사용하여 함수를 다른 함수에 전달할 수 있습니다. 함수 포인터는 유지 관리하고 이해하기에 불편합니다. 함수 포인터가 참조하는 함수는 호출되는 지점과 멀리 떨어진 소스 코드 다른 곳에서 정의될 수 있습니다. 또한 형식이 안전하지 않습니다. 최신 C++는 [operator\(\)](#) 연산자를 재정의하는 클래스인 함수 개체를 제공하므로 함수 개체를 함수처럼 호출할 수 있습니다. 함수 개체를 만드는 가장 편리한 방법은 인라인 [람다 식](#)을 사용하는 것입니다. 다음 예제는 람다 식을 사용하여 `find_if` 함수가 벡터의 각 요소에 대해 호출할 함수 개체를 전달하는 방법을 보여 줍니다.

C++

```

std::vector<int> v {1,2,3,4,5};
int x = 2;
int y = 4;
auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i < y; });

```

람다 식 [=](int i) { return i > x && i < y; } 은 "형식 int 의 단일 인수를 사용하고 인수가 보다 x 크고 작은 y 지 여부를 나타내는 부울을 반환하는 함수"로 읽을 수 있습니다. 변수 및 x y 주변 컨텍스트에서 람다에서 사용할 수 있습니다. [=] 는 이러한 변수가 값에 의해 캡처됨을 명시합니다. 즉, 람다 식에는 이러한 값의 고유한 복사본이 있습니다.

예외

최신 C++는 오류 조건을 보고하고 처리하는 가장 좋은 방법으로 오류 코드가 아닌 예외를 강조합니다. 자세한 내용은 [최신 C++ 예외 및 오류 처리 모범 사례](#)를 참조하세요.

std::atomic

스레드 간 통신 메커니즘에 C++ 표준 라이브러리 std::atomic 구조체와 관련 형식을 사용하세요.

std::variant (C++17)

C 스타일 프로그래밍에서는 일반적으로 공용 구조체를 사용하여 서로 다른 형식의 멤버가 동일한 메모리 위치를 점유할 수 있도록 함으로써 메모리를 보존합니다. 하지만 공용 구조체는 형식이 안전하지 않으며 프로그래밍 오류가 발생하기 쉽습니다. C++17에서는 공용 구조체보다 강력하고 안전한 대안으로 std::variant 클래스가 도입되었습니다.

std::visit 함수를 사용하면 variant 형식의 멤버에 형식이 안전한 방식으로 액세스할 수 있습니다.

참조

[C++ 언어 참조](#)

[람다 식](#)

[C++ 표준 라이브러리](#)

[Microsoft C/C++ 언어 규칙](#)

어휘 규칙

아티클 • 2024. 11. 21.

이 단원에서는 C++ 프로그램의 기본 요소를 소개합니다. 문, 정의, 선언 등을 생성하는 "어휘 요소" 또는 "토큰"이라고 하는 요소를 사용하여 완전한 프로그램을 구성할 수 있습니다. 이 단원에서 설명하는 어휘 요소는 다음과 같습니다.

- 토큰 및 문자 집합
- 의견
- 식별자
- 키워드
- 문장 부호
- 숫자, 부울 및 포인터 리터럴
- 문자열 및 문자 리터럴
- 사용자 정의 리터럴

C++ 원본 파일을 구문 분석하는 방법에 대한 자세한 내용은 번역 [단계를 참조하세요](#).

참고 항목

[C++ 언어 참조](#)

[변환 단위 및 링크](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

토큰 및 문자 집합

아티클 • 2023. 04. 03.

C++ 프로그램의 텍스트는 토큰과 공백으로 구성됩니다. 토큰은 컴파일러에 의미 있는 C++ 프로그램의 최소 요소입니다. C++ 파서는 다음과 같은 종류의 토큰을 인식합니다.

- C++ 키워드
- 식별자
- 숫자, 부울 및 포인터 리터럴
- 문자열 및 문자 리터럴
- 사용자 정의 리터럴
- 연산자
- 문장 부호

토큰은 일반적으로 공백으로 구분되며 하나 이상일 수 있습니다.

- 공백
- 가로 또는 세로 탭
- 새로운 줄
- 양식 피드
- 의견

기본 소스 문자 집합

C++ 표준은 원본 파일에서 사용할 수 있는 기본 소스 문자 집합을 지정합니다. 이 집합 외부에 문자를 나타내려면 유니버설 문자 이름을 사용하여 추가 문자를 지정할 수 있습니다. MSVC 구현은 추가 문자를 허용합니다. 기본 소스 문자 집합은 소스 파일에서 사용 할 수 있는 96자로 구성됩니다. 이 집합에는 공백 문자, 가로 탭, 세로 탭, 폼 피드 및 줄 바 꿈 제어 문자가 포함되며 다음 그래픽 문자 집합도 포함됩니다.

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

_ { } [] # () < > % : ; . ? * + - / ^ & | ~ ! = , \ " '

Microsoft 전용

MSVC는 문자를 기본 원본 문자 집합의 멤버로 포함합니다 \$. 또한 MSVC를 사용하면 파일 인코딩에 따라 소스 파일에서 추가 문자 집합을 사용할 수 있습니다. 기본적으로

Visual Studio는 기본 코드 페이지를 사용하여 소스 파일을 저장합니다. 로캘별 코드 페이지 또는 유니코드 코드 페이지를 사용하여 소스 파일을 저장하는 경우 MSVC를 사용하면 기본 소스 문자 집합에서 명시적으로 허용되지 않는 제어 코드를 제외하고 소스 코드에서 해당 코드 페이지의 문자를 사용할 수 있습니다. 예를 들어 일본어 코드 페이지를 사용하여 파일을 저장하는 경우 주석, 식별자 또는 문자열 리터럴에 일본어 문자를 배치할 수 있습니다. MSVC는 유효한 멀티바이트 문자 또는 유니코드 코드 포인트로 변환할 수 없는 문자 시퀀스를 허용하지 않습니다. 컴파일러 옵션에 따라 허용되는 일부 문자가 식별자에 나타나지 않을 수 있습니다. 자세한 내용은 [Identifiers](#)를 참조하세요.

Microsoft 전용 끝

유니버설 문자 이름

C++ 프로그램에서는 기본 소스 문자 집합에 지정된 것보다 훨씬 더 많은 문자를 사용할 수 있으므로 유니버설 문자 이름을 사용하여 이식 가능한 방법으로 이러한 문자를 지정할 수 있습니다. 유니버설 문자 이름은 유니코드 코드 포인트를 나타내는 문자 시퀀스로 구성되며, 두 가지 형식을 사용합니다. `\UNNNNNNNN` 을 사용하여 U+NNNNNNNN 형식의 유니코드 코드 포인터를 나타냅니다. 여기서 NNNNNNNN은 8자리 16진수 코드 포인트 번호입니다. 4자리 `\uNNNN` 을 사용하여 U+0000NNNN 형식의 유니코드 코드 포인트를 나타냅니다.

유니버설 문자 이름은 문자열 및 문자 리터럴과 식별자에서 사용할 수 있습니다. 유니버설 문자 이름은 0xD800-0xDFFF 범위에 있는 서로게이트 코드 포인트를 나타내는 데 사용할 수 없습니다. 대신 원하는 코드 포인트를 사용하면 컴파일러에서 필요한 서로게이트를 자동으로 생성합니다. 식별자에 사용할 수 있는 유니버설 문자 이름에는 추가 제한이 적용됩니다. 자세한 내용은 [Identifiers](#) 및 [String and Character Literals](#)을 참조하세요.

Microsoft 전용

Microsoft C++ 컴파일러는 범용 문자 이름 형식의 문자와 리터럴 형식을 서로 바꿔서 처리합니다. 예를 들어 유니버설 문자 이름 형식을 사용하여 식별자를 선언하고 리터럴 형식에서 사용할 수 있습니다.

C++

```
auto \u30AD = 42; // \u30AD is 'キ'  
if (\u30AD == 42) return true; // \u30AD and \u30AD are the same to the compiler
```

Windows 클립보드의 확장된 문자 형식은 애플리케이션 로캘 설정과 관련이 있습니다. 다른 애플리케이션에서 이러한 문자를 잘라내어 코드에 붙여 넣으면 예기치 않은 문자 인코딩이 발생할 수 있습니다. 그러면 표시되는 원인 없이 코드에서 구문 분석 오류가 발생할 수 있습니다. 따라서 확장된 문자를 붙여 넣기 전에 소스 파일 인코딩을 유니코드 코

드 페이지로 설정하는 것이 좋습니다. 또한 IME 또는 문자표 앱을 사용하여 확장된 문자를 생성하는 것이 좋습니다.

Microsoft 전용 끝

실행 문자 집합

실행 문자 집합은 컴파일된 프로그램에 나타날 수 있는 문자와 문자열을 나타냅니다. 이러한 문자 집합은 원본 파일에 허용되는 모든 문자와 경고, 백스페이스, 캐리지 리턴 및 null 문자를 나타내는 컨트롤 문자로 구성됩니다. 실행 문자 집합에는 로캘별 표현이 있습니다.

주석(C++)

아티클 • 2023. 10. 12.

주석은 컴파일러가 무시하지만 프로그래머에게 유용한 텍스트입니다. 주석은 일반적으로 이후 참조를 위해 코드에 주석을 추가하는 데 사용됩니다. 컴파일러는 이를 공백으로 처리합니다. 테스트에 주석을 사용하여 특정 코드 줄을 비활성 상태로 만들 수 있습니다. 그러나 `#if` / `#endif` 주석이 포함된 코드를 둘러싸는 것은 가능하지만 주석을 중첩할 수는 없으므로 전처리기 지시문이 더 잘 작동합니다.

C++ 주석은 다음 방법 중 하나로 작성됩니다.

- `/*` (슬래시, 별표) 문자 뒤에 문자 시퀀스(새 줄 포함) 뒤에 문자가 있습니다`*/`. 이 구문은 ANSI C와 동일합니다.
- `//` (두 슬래시) 문자 뒤에 문자 시퀀스가 있습니다. 백슬래시 바로 앞에 새 줄이 있으면 이 형식의 주석이 종료됩니다. 따라서 일반적으로 "단일 줄 주석"이라고 합니다.

주석 문자(`/*` 및 `*/`, `//`)는 문자 상수, 문자열 리터럴 또는 주석 내에서 특별한 의미가 없습니다. 따라서 첫 번째 구문을 사용하는 주석은 중첩할 수 없습니다.

참고 항목

[어휘 규칙](#)

식별자(C++)

아티클 • 2023. 10. 12.

식별자는 다음 중 하나를 나타내는 데 사용되는 문자 시퀀스입니다.

- 개체 또는 변수 이름
- 클래스, 구조체 또는 공용 구조체 이름
- 열거 형식 이름
- 클래스, 구조체, 공용 구조체 또는 열거의 멤버
- 함수 또는 클래스 멤버 함수
- typedef 이름
- Label name
- 매크로 이름
- 매크로 매개 변수

다음 문자는 식별자의 모든 문자로 허용됩니다.

```
_ a b c d e f g h i j k l m  
n o p q r s t u v w x y z  
A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z
```

특정 범위의 유니버설 문자 이름도 식별자에서 허용됩니다. 식별자의 유니버설 문자 이름은 제어 문자나 기본 소스 문자 집합의 문자를 지정할 수 없습니다. 자세한 내용은 [Character Sets](#)을 참조하세요. 이러한 유니코드 코드 포인트 숫자 범위는 식별자의 모든 문자에 대해 유니버설 문자 이름으로 허용됩니다.

- 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFFD, 10000-1FFFFD, 20000-2FFFFD, 30000-3FFFFD, 40000-4FFFFD, 50000-5FFFFD, 60000-6FFFFD, 70000-7FFFFD, 80000-8FFFFD, 90000-9FFFFD, A0000-AFFFFD, B0000-BFFFFD, C0000-CFFFFD, D0000-DFFFFD, E0000-EFFFFD

다음 문자는 식별자의 첫 번째 문자를 제외한 모든 문자로 허용됩니다.

```
0 1 2 3 4 5 6 7 8 9
```

이러한 유니코드 코드 포인트 숫자 범위는 식별자의 첫 번째 문자를 제외한 모든 문자에 대해 유니버설 문자 이름으로도 허용됩니다.

- 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F

Microsoft 전용

Microsoft C++ 식별자의 처음 2048 문자만 의미가 있습니다. 사용자 정의 형식의 이름은 컴파일러에서 "데코레이팅"되어 형식 정보를 유지합니다. 형식 정보를 포함하는 결과 이름은 2048자를 초과할 수 없습니다. (참조) [자세한 내용을 보려면 데코레이팅된 이름 입니다.](#)) 데코레이팅된 식별자의 길이에 영향을 줄 수 있는 요소는 다음과 같습니다.

- 식별자가 사용자 정의 형식의 개체 또는 사용자 정의 형식에서 파생된 형식을 나타내는지 여부
- 식별자가 함수 또는 함수에서 파생된 형식을 나타내는지 여부
- 함수에 사용되는 인수의 수

달러 기호 \$ 는 MSVC(Microsoft C++ 컴파일러)의 유효한 식별자 문자입니다. MSVC를 사용하면 식별자에서 허용된 범용 문자 이름 범위로 표시되는 실제 문자를 사용할 수도 있습니다. 이러한 문자를 사용하려면 해당 문자를 포함하는 파일 인코딩 코드 페이지를 사용하여 파일을 저장해야 합니다. 이 예에서는 확장 문자와 유니버설 문자 이름을 코드에서 서로 바꿔 사용할 수 있는 방법을 보여 줍니다.

C++

```
// extended_identifier.cpp
// In Visual Studio, use File, Advanced Save Options to set
// the file encoding to Unicode codepage 1200
struct テスト          // Japanese 'test'
{
    void トスト() {}   // Japanese 'toast'
};

int main() {
    テスト \u30D1\u30F3;    // Japanese パン 'bread' in UCN form
    パン.トスト();        // compiler recognizes UCN or literal form
}
```

식별자에서 허용되는 문자의 범위는 C++/CLI 코드를 컴파일하는 경우보다 덜 제한적입니다. /clr을 사용하여 컴파일된 코드의 식별자는 [Standard ECMA-335: Common Language Infrastructure\(CLI\)](#) 를 준수해야 합니다.

Microsoft 전용 종료

식별자의 첫 번째 문자는 영문자(대문자 또는 소문자) 또는 밑줄(_)이어야 합니다. C++ 식별자가 대/소문자를 구분하기 때문에 `fileName` 은 `FileName`과 다릅니다.

식별자는 키워드와 정확히 동일한 철자와 대/소문자를 사용할 수 없습니다. 키워드가 포함된 식별자를 사용할 수 있습니다. 예를 들어, `Pint` 는 키워드인 `int`가 포함되어 있어도 유효한 식별자입니다.

식별자에서 두 개의 순차적 밑줄 문자(_) 또는 앞에 오는 단일 밑줄과 대문자를 차례로 사용하는 것은 모든 범위에서 C++ 구현을 위해 예약되어 있습니다. 현재 또는 나중에 예약되는 식별자와 충돌할 수 있기 때문에 파일 범위가 있는 이름에 소문자가 뒤에 오는 단일 선행 밑줄을 사용하지 않도록 해야 합니다.

참고 항목

[어휘 규칙](#)

키워드(C++)

아티클 • 2024. 07. 15.

키워드는 특별한 의미가 있는 미리 정의된 예약된 식별자입니다. 이러한 키워드는 프로그램에서 식별자로 사용할 수 없습니다. 다음 키워드는 Microsoft C++에서 예약되었습니다. 선행 밑줄이 있는 이름과 C++/CX 및 C++/CLI에 대해 지정된 이름은 Microsoft 확장 명입니다.

표준 C++ 키워드

alignas

alignof

and^b

and_eq^b

asm^a

auto

bitand^b

bitor^b

bool

break

case

catch

char

char8_t^c

char16_t

char32_t

class

compl^b

concept^c

const

const_cast

consteval^c

constexpr

constinit^c

continue

co_await^c

co_return^c

co_yield^c

decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
export ^c
extern
false
float
for
friend
goto
if
inline

int
long
mutable
namespace
new
noexcept
not ^b
not_eq ^b
nullptr
operator
or ^b
or_eq ^b
private
protected
public
register reinterpret_cast
requires ^c
return
short
signed
sizeof

static
static_assert

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using 선언
using 지시문
virtual
void
volatile
wchar_t
while
xor ^b
xor_eq ^b

^a Microsoft 전용 `__asm` 키워드는 C++ `asm` 구문을 대체합니다. `asm`은 다른 C++ 구현과의 호환성을 위해 예약되었지만 구현되지는 않았으므로 x86 대상의 인라인 어셈블리에는 `__asm`을 사용합니다. Microsoft C++는 다른 대상에 대한 인라인 어셈블리를 지원하지 않습니다.

^b 확장 연산자 동의어는 `/permissive-` 또는 `/Za(언어 확장 사용 안 함)`가 지정된 경우의 키워드입니다. 이는 Microsoft 확장을 사용하는 경우의 키워드가 아닙니다.

^c `/std:c++20` 또는 그 이상(예: `/std:c++latest`)이 지정된 경우 지원됩니다.

Microsoft 전용 C++ 키워드

C++에서 두 개의 연속된 밑줄이 포함된 식별자는 컴파일러 구현용으로 예약되었습니다. Microsoft 규칙은 Microsoft 전용 키워드 앞에 이중 밑줄을 붙이는 것입니다. 이들 단어는

식별자 이름으로 사용할 수 없습니다.

Microsoft 확장은 기본적으로 사용하도록 설정됩니다. 프로그램이 완전하게 이식 가능하도록, 컴파일하는 동안 [/permissive-](#) 또는 [/Za\(언어 확장 사용 안 함\)](#) 옵션을 지정하여 Microsoft 확장을 사용하지 않을 수 있습니다. 이러한 옵션은 일부 Microsoft 전용 키워드를 사용하지 않도록 설정합니다.

Microsoft 확장을 사용하도록 설정한 경우 Microsoft 관련 키워드를 프로그램에서 사용할 수 있습니다. ANSI 규칙의 준수를 위해 이러한 키워드에는 이중 밑줄이 앞에 옵니다. 이전 버전과의 호환성을 위해 많은 이중 밑줄 키워드에 대해 단일 밑줄 버전이 지원됩니다.

`_cdecl` 키워드는 선행 밑줄 없이 사용할 수 있습니다.

`_asm` 키워드는 C++ `asm` 구문을 대체합니다. `asm`은 다른 C++ 구현과의 호환성을 위해 예약되었지만 구현되지는 않았으므로 `_asm`을 사용합니다.

`_based` 키워드는 32비트 및 64비트 대상 컴파일에서 제한적으로 사용됩니다.

`_alignof` ^e

`_asm` ^e

`_assume` ^e

`_based` ^e

`_cdecl` ^e

`_declspec` ^e

`_event`

`_except` ^e

`_fastcall` ^e

`_finally` ^e

`_forceinline` ^e

`_hook` ^d

`_if_exists`

`_if_not_exists`

`_inline` ^e

`_int16` ^e

`_int32` ^e

`_int64` ^e

`_int8` ^e

`_interface`

`_leave` ^e

`_m128`

`_m128d`
`_m128i`
`_m64`
`_multiple_inheritancee`
`_ptr32e`
`_ptr64e`
`_raise`
`_restricte`
`_single_inheritancee`
`_sptre`
`_stdcalle`

`_super`
`_thiscall`
`_unalignede`
`_unhookd`
`_uptre`
`_uuidofe`
`_vectorcalle`
`_virtual_inheritancee`
`_w64e`
`_wchar_t`

^d 이벤트 처리에서 사용되는 내장 함수입니다.

^e 이전 버전과의 호환성을 위해 이러한 키워드는 Microsoft 확장이 사용하도록 설정된 경우(기본값), 두 개의 선행 밑줄 및 한 개의 선행 밑줄 둘 다와 함께 사용할 수 있습니다.

`_declspec` 한정자의 Microsoft 키워드

이러한 식별자는 `_declspec` 한정자에 대한 확장 특성입니다. 이는 해당 컨텍스트 내의 키워드로 간주됩니다.

`align`
`allocate`
`allocator`
`appdomain`
`code_seg`
`deprecated`

dllexport
dllimport
jitintrinsic
naked
noalias
noinline

noreturn
no_sanitize_address
nothrow
novtable
process
property

restrict
safebuffers
selectany
spectre
thread
uuid

C++/CLI 및 C++/CX 키워드

_abstract f
_box f
_delegate f
_gc f
_identifier
_nogc f
_noop
_pin f
_property f
_sealed f

_try_cast f
_value f
abstract g
array g
as_friend
delegate g

enum class
enum struct
event ⁹

finally
for each in
gcnew ⁹
generic ⁹
initonly
interface class ⁹
interface struct ⁹
interior_ptr ⁹
literal ⁹

new ⁹
property ⁹
ref class
ref struct
safecast
sealed ⁹
typeid
value class ⁹
value struct ⁹

^f Managed Extensions for C++에만 적용 가능합니다. 이 구문은 이제 사용되지 않습니다.

자세한 내용은 [Component Extensions for Runtime Platforms](#)을 참조하세요.

⁹ C++/CLI에 적용 가능합니다.

참고 항목

[어휘 규칙](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

문장 부호(C++)

아티클 • 2023. 10. 12.

C++의 문장 부호는 컴파일러에서 구문 및 의미 체계를 의미하지만 스스로 값을 생성하는 연산을 지정하지 않습니다. 또한 일부 단독 또는 조합 문장 기호는 C++ 연산자가 될 수도 있고 전처리기에 중요할 수도 있습니다.

다음 문자는 모두 문장 부호로 간주됩니다.

```
! % ^ & * ( ) - + = { } | ~  
[ ] \ ; ' : " < > ? , . / #
```

문장 부호 [], (), {}는 번역 4단계 [이후](#) 쌍으로 나타나야 합니다.

참고 항목

[어휘 규칙](#)

숫자, 부울 및 포인터 리터럴

아티클 • 2023. 10. 12.

리터럴은 값을 직접 나타내는 프로그램 요소입니다. 이 문서에서는 정수, 부동 소수점, 부울 및 포인터 형식의 리터럴에 대해 설명합니다. 문자열 및 문자 리터럴에 대한 자세한 내용은 [문자열 및 문자 리터럴\(C++\)](#)을 참조하세요. 이러한 범주에 따라 고유한 리터럴을 정의할 수도 있습니다. 자세한 내용은 [사용자 정의 리터럴\(C++\)](#)을 참조하세요.

리터럴은 많은 컨텍스트에서 사용할 수 있지만 가장 일반적으로 명명된 변수를 초기화하고 인수를 함수에 전달하는 데 사용됩니다.

C++

```
const int answer = 42;           // integer literal
double d = sin(108.87);        // floating point literal passed to sin function
bool b = true;                 // boolean literal
MyClass* mc = nullptr;         // pointer literal
```

경우에 따라 리터럴을 해석하는 방법 또는 지정할 특정 형식을 컴파일러에 알려야 합니다. 이 작업은 리터럴에 접두사 또는 접미사를 추가하여 수행됩니다. 예를 들어 접두 `0x` 사는 컴파일러에 뒤에 있는 숫자를 16진수 값 `0x35`으로 해석하도록 지시합니다. `ULL` 접미사는 컴파일러에 값을 다음과 같이 `5894345ULL` 형식으로 `unsigned long long` 처리하도록 지시합니다. 각 리터럴 형식에 대한 접두사 및 접미사의 전체 목록을 다음 섹션을 참조하세요.

정수 리터럴

정수 리터럴은 숫자로 시작하고 소수 부분이나 지수가 없습니다. 정수 리터럴을 10진수, 이진, 8진수 또는 16진수 형식으로 지정할 수 있습니다. 필요에 따라 접미사를 사용하여 정수 리터럴을 부호 없는 형식으로, 길고 긴 형식으로 지정할 수 있습니다.

접두사 또는 접미사가 없으면 컴파일러는 정수 리터럴 값 형식 `int` (32비트)을 제공하고, 값이 맞으면 형식(64비트)을 제공합니다 `long long`.

10진수 정수 계열 리터럴을 지정하려면 0이 아닌 숫자를 사용하여 지정을 시작합니다. 예시:

C++

```
int i = 157;           // Decimal literal
int j = 0198;          // Not a decimal number; erroneous octal literal
```

```
int k = 0365;           // Leading zero specifies octal literal, not decimal
int m = 36'000'000 // digit separators make large values more readable
```

8진수 정수 계열 리터럴을 지정하려면 0으로 시작하여 0부터 7까지의 숫자 시퀀스를 지정합니다. 8진수 리터럴을 지정할 때 숫자 8과 9는 오류입니다. 예시:

C++

```
int i = 0377;    // Octal literal
int j = 0397;    // Error: 9 is not an octal digit
```

16진수 정수 리터럴을 지정하려면 사양을 `0x` 시작하거나 `0X` ("x"의 경우는 중요하지 않음) 범위의 숫자 시퀀스 `0`(또는 `0`)를 통해 `9 f a` (또는 `A F`)를 시작합니다. 16진수 숫자 `a`(또는 `A`)부터 `f`(또는 `F`)는 10부터 15 사이의 값을 나타냅니다. 예시:

C++

```
int i = 0x3fff;    // Hexadecimal literal
int j = 0X3FFF;    // Equal to i
```

부호 없는 형식을 지정하려면 접미사 또는 `u` 접미사를 사용합니다`u`. 긴 형식을 지정하려면 접미사 또는 `l` 접미사를 사용합니다`l`. 64비트 정수 계열 형식을 지정하려면 `LL` 또는 `ll` 접미사를 사용합니다. `i64` 접미사는 여전히 지원되지만 권장하지는 않습니다. Microsoft 전용이며 이식 가능하지 않습니다. 예시:

C++

```
unsigned val_1 = 328u;                  // Unsigned value
long val_2 = 0x7FFFFFFL;                // Long value specified
                                         // as hex literal
unsigned long val_3 = 0776745ul;        // Unsigned long value
auto val_4 = 108LL;                    // signed long long
auto val_4 = 0x800000000000000ULL << 16; // unsigned long long
```

숫자 구분 기호: 작은따옴표 문자(아포스트로피)를 사용하여 값을 더 큰 숫자로 구분하여 사용자가 더 쉽게 읽을 수 있도록 할 수 있습니다. 구분 기호는 컴파일에 영향을 주지 않습니다.

C++

```
long long i = 24'847'458'121;
```

부동 소수점 리터럴

부동 소수점 리터럴은 소수 부분이 있어야 하는 값을 지정합니다. 이러한 값은 소수점(.)을 포함하며 지수를 포함할 수 있습니다.

부동 소수점 리터럴에는 숫자의 값을 지정하는 중요도(가수라고도 함)가 있습니다. 숫자의 크기를 지정하는 지수가 있습니다. 또한 리터럴의 형식을 지정하는 선택적 접미사가 있습니다. significand는 숫자 시퀀스 뒤에 마침표, 숫자의 소수 부분을 나타내는 선택적 숫자 시퀀스로 지정됩니다. 예시:

C++

```
18.46  
38.
```

지수는(있는 경우) 다음 예제와 같이 숫자의 크기를 10의 거듭제곱으로 지정합니다.

C++

```
18.46e0      // 18.46  
18.46e1      // 184.6
```

지수는 의미가 같거나 E 선택적 기호(+ 또는 -) 및 숫자 시퀀스를 사용하여 e 지정할 수 있습니다. 지수가 있는 경우 18E0과 같은 정수에는 뒤에 오는 소수점이 필요하지 않습니다.

부동 소수점 리터럴은 기본적으로 .를 입력 double 합니다. 접미사 f 또는 l 접미사를 사용하거나 F L (접미사가 대/소문자를 구분하지 않음) 리터럴을로 float 지정하거나 long double 지정할 수 있습니다.

표현은 double 동일하지만 long double 형식은 동일하지 않습니다. 예를 들어 다음과 같은 오버로드된 함수가 있을 수 있습니다.

C++

```
void func( double );
```

및

C++

```
void func( long double );
```

부울 리터럴

부울 리터럴은 다음과 `false` 같습니다 `true`.

포인터 리터럴(C++11)

C++는 리터럴을 `nullptr` 도입하여 초기화되지 않은 포인터를 지정합니다. 이식 가능한 코드 `nullptr` 에서는 정수 형식 0 또는 매크로(예: `NULL`) 대신 사용해야 합니다.

이진 리터럴(C++14)

이진 리터럴은 `0B` 또는 `0b` 접두사와 뒤에 오는 1과 0의 시퀀스를 사용하여 지정할 수 있습니다.

C++

```
auto x = 0B001101 ; // int
auto y = 0b000001 ; // int
```

리터럴을 "매직 상수"로 사용하지 마세요.

리터럴은 식과 문에서 직접 사용할 수 있지만 항상 바람직한 프로그래밍 방법은 아닙니다.

C++

```
if (num < 100)
    return "Success";
```

이전 예제에서는 명확한 의미를 전달하는 명명된 상수(예: "MAXIMUM_ERROR_THRESHOLD")를 사용하는 것이 좋습니다. 최종 사용자가 반환 값 "Success"를 볼 경우 명명된 문자열 상수 사용이 더 좋을 수 있습니다. 다른 언어로 지역화 할 수 있는 파일의 단일 위치에 문자열 상수는 유지할 수 있습니다. 명명된 상수는 자신과 다른 사용자 모두 코드의 의도를 이해하는 데 도움이 됩니다.

참고 항목

[어휘 규칙](#)

[C++ 문자열 리터럴](#)

[C++ 사용자 정의 리터럴](#)

문자열 및 문자 리터럴(C++)

아티클 • 2023. 04. 03.

C++를 사용하면 다양한 문자열 및 문자 형식이 지원되며 이러한 각 형식의 리터럴 값을 표현할 수 있습니다. 소스 코드에서는 문자 집합을 사용하여 문자 및 문자열 리터럴의 내용을 표현합니다. 유니버설 문자 이름 및 이스케이프 문자를 사용하면 기본 소스 문자 집합만 사용하여 모든 문자열을 표현할 수 있습니다. 원시 문자열 리터럴을 사용하면 이스케이프 문자를 사용하지 않아도 되며 원시 문자열 리터럴을 사용하여 모든 유형의 문자열 리터럴을 표현할 수 있습니다. 추가 생성 또는 변환 단계를 수행할 필요 없이 리터럴을 만들 `std::string` 수도 있습니다.

C++

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string literals

int main()
{
    // Character literals
    auto c0 = 'A'; // char
    auto c1 = u8'A'; // char
    auto c2 = L'A'; // wchar_t
    auto c3 = u'A'; // char16_t
    auto c4 = U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char* before C++20, encoded as UTF-8,
                         // const char8_t* in C++20
    auto s2 = L"hello"; // const wchar_t*
    auto s3 = u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 = U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 = R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char* before C++20, encoded
    as UTF-8,
                                         // const char8_t* in C++20
    auto R2 = LR"("Hello \ world")"; // const wchar_t*
    auto R3 = uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
    auto R4 = UR"("Hello \ world")"; // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 = "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string before C++20, std::u8string in
```

C++20

```
auto S2 = L"hello"s; // std::wstring
auto S3 = u"hello"s; // std::u16string
auto S4 = U"hello"s; // std::u32string

// Combining raw string literals with standard s-suffix
auto S5 = R"("Hello \ world")"s; // std::string from a raw const char*
auto S6 = u8R"("Hello \ world")"s; // std::string from a raw const char*
before C++20, encoded as UTF-8,
                                                // std::u8string in C++20
auto S7 = LR"("Hello \ world")"s; // std::wstring from a raw const
wchar_t*
auto S8 = uR"("Hello \ world")"s; // std::u16string from a raw const
char16_t*, encoded as UTF-16
auto S9 = UR"("Hello \ world")"s; // std::u32string from a raw const
char32_t*, encoded as UTF-32
}
```

문자열 리터럴에는 각각 좁은 문자(단일 바이트 또는 `u8`다중 바이트), UTF-8, 와이드 문자(UCS-2 또는 UTF-16), UTF-16 및 UTF-32 인코딩을 나타내는 접두사 또는 `L`, `u`, `U`, 및 접두사를 가질 수 없습니다. 원시 문자열 리터럴은 이러한 인코딩에 해당하는 원시 버전에 대한 `u8R`, `LR`, `uR` 및 `UR` 접두사를 가질 수 있습니다. 임시 또는 정적 `std::string` 값을 만들려면 문자열 리터럴 또는 접미사가 있는 원시 문자열 리터럴을 `s` 사용할 수 있습니다. 자세한 내용은 아래 [의 문자열 리터럴 섹션을](#) 참조하세요. 기본 소스 문자 집합, 범용 문자 이름 및 소스 코드의 확장된 코드 페이지의 문자 사용에 대한 자세한 내용은 [문자 집합을](#) 참조하세요.

문자 리터럴

문자 리터럴은 상수 문자로 구성됩니다. 작은따옴표로 둘러싸인 문자로 표시됩니다. 문자 리터럴에는 5가지 종류가 있습니다.

- 형식 `char`의 일반 문자 리터럴(예: `'a'`)
- 예를 들어 형식 `char`의 UTF-8 문자 리터럴(`char8_t` C++20)입니다. `u8'a'`
- `wchar_t` 형식의 와이드 문자 리터럴(예: `L'a'`)
- 형식의 `char16_t` UTF-16 문자 리터럴(예: `u'a'`)
- 형식의 `char32_t` UTF-32 문자 리터럴(예: `U'a'`)

문자 리터럴에 사용되는 문자는 예약된 문자 백슬래시(), 작은따옴표(') 또는 줄 바꿈을 제외한 모든 문자일 수 있습니다. 예약된 문자는 이스케이프 시퀀스를 사용하여 지정할 수 있습니다. 문자는 형식이 문자를 저장할 수 있을 만큼 큰 경우 유니버설 문자 이름을 사용하여 지정할 수 있습니다.

Encoding

문자 리터럴은 접두사를 기반으로 다르게 인코딩됩니다.

- 접두사 없는 문자 리터럴은 일반 문자 리터럴입니다. 실행 문자 집합에 나타낼 수 있는 단일 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 일반 문자 리터럴의 값은 실행 문자 집합에서 인코딩의 숫자 값과 같은 값을 줍니다. 둘 이상의 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 일반 문자 리터럴은 다자간 리터럴입니다. 다중 문자 리터럴 또는 실행 문자 집합에 나타낼 수 없는 일반 문자 리터럴에는 형식 `int`이 있으며 해당 값은 구현에서 정의됩니다. MSVC의 경우 아래 [의 Microsoft 관련 섹션](#)을 참조하세요.
- 접두사로 `L` 시작하는 문자 리터럴은 와이드 문자 리터럴입니다. 단일 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 와이드 문자 리터럴의 값은 실행 와이드 문자 집합에 문자 리터럴이 표현되지 않는 한 실행 와이드 문자 집합에서 인코딩의 숫자 값과 같은 값을 하며, 이 경우 값이 구현 정의됩니다. 여러 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 와이드 문자 리터럴의 값은 구현에서 정의됩니다. MSVC의 경우 아래 [의 Microsoft 관련 섹션](#)을 참조하세요.
- 접두사로 `u8` 시작하는 문자 리터럴은 UTF-8 문자 리터럴입니다. 단일 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-8 문자 리터럴의 값은 단일 UTF-8 코드 단위(C0 컨트롤 및 기본 라틴 유니코드 블록에 해당)로 나타낼 수 있는 경우 해당 ISO 10646 코드 포인트 값과 같은 값을 가집니다. 값을 단일 UTF-8 코드 단위로 나타낼 수 없는 경우 프로그램이 잘못 구성됩니다. 둘 이상의 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-8 문자 리터럴의 형식이 잘못되었습니다.
- 접두사로 `u` 시작하는 문자 리터럴은 UTF-16 문자 리터럴입니다. 단일 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-16 문자 리터럴의 값은 단일 UTF-16 코드 단위(기본 다국어 평면에 해당)로 나타낼 수 있는 경우 해당 ISO 10646 코드 포인트 값과 같은 값을 가집니다. 값을 단일 UTF-16 코드 단위로 나타낼 수 없는 경우 프로그램이 잘못 구성됩니다. 둘 이상의 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-16 문자 리터럴의 형식이 잘못되었습니다.
- 접두사로 `U` 시작하는 문자 리터럴은 UTF-32 문자 리터럴입니다. 단일 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-32 문자 리터럴의 값은 해당 ISO 10646 코드 포인트 값과 같은 값을 가집니다. 둘 이상의 문자, 이스케이프 시퀀스 또는 범용 문자 이름을 포함하는 UTF-32 문자 리터럴의 형식이 잘못되었습니다.

이스케이프 시퀀스

이스케이프 시퀀스는 단순, 8진수 및 16진수의 세 종류가 있습니다. 이스케이프 시퀀스는 다음 값 중 어느 것일 수 있습니다.

값	이스케이프 시퀀스
줄 바꿈	\n
백슬래시	\\
가로 탭	\t
물음표	? 또는 \?
세로 탭	\v
작은따옴표	\'
백스페이스	\b
큰따옴표	\"
캐리지 리턴	\r
null 문자	\0
폼 피드	\f
8진수	\ooo
경고(벨)	\a
16진수	\xhhh

8진수 이스케이프 시퀀스는 백슬래시 뒤에 1~3개의 8진수 시퀀스입니다. 8진수 이스케이프 시퀀스는 8진수가 아닌 첫 번째 문자에서 종료됩니다(세 번째 숫자보다 빨리 발생하는 경우). 가능한 가장 높은 8진수 값은 입니다 [\377](#).

16진수 이스케이프 시퀀스는 백슬래시 뒤에 문자 x가 뒤에 하나 이상의 16진수 숫자 시퀀스입니다. 앞에 오는 0은 무시됩니다. 일반 또는 u8 접두사 문자 리터럴에서 가장 높은 16진수 값은 0xFF. L 접두사 또는 u 접두사가 있는 와이드 문자 리터럴에서 가장 큰 16진수 값은 0xFFFF입니다. U 접두사가 있는 와이드 문자 리터럴에서 가장 큰 16진수 값은 0xFFFFFFFF입니다.

이 샘플 코드는 일반 문자 리터럴을 사용하여 이스케이프된 문자의 몇 가지 예를 보여 줍니다. 동일한 이스케이프 시퀀스 구문은 다른 문자 리터럴 형식에 유효합니다.

C++

```
#include <iostream>
using namespace std;

int main() {
    char newline = '\n';
```

```

char tab = '\t';
char backspace = '\b';
char backslash = '\\';
char nullChar = '\0';

cout << "Newline character: " << newline << "ending" << endl;
cout << "Tab character: " << tab << "ending" << endl;
cout << "Backspace character: " << backspace << "ending" << endl;
cout << "Backslash character: " << backslash << "ending" << endl;
cout << "Null character: " << nullChar << "ending" << endl;
}

/* Output:
Newline character:
ending
Tab character: ending
Backspace character:ending
Backslash character: \ending
Null character: ending
*/

```

백슬래시 문자(\)는 줄 끝에 배치되는 줄 연속 문자입니다. 백슬래시 문자가 문자 리터럴로 표시되도록 하려면 두 개의 백슬래시(\\)를 연속하여 입력해야 합니다. 줄 연속 문자에 대한 자세한 내용은 [변환 단계](#)를 참조하세요.

Microsoft 전용

좁은 다중 문자 리터럴에서 값을 만들기 위해 컴파일러는 작은따옴표 사이의 문자 또는 문자 시퀀스를 32비트 정수 내의 8비트 값으로 변환합니다. 리터럴의 여러 문자는 필요에 따라 상위에서 하위로 해당 바이트를 채웁니다. 그런 다음 컴파일러는 일반적인 규칙에 따라 정수를 대상 형식으로 변환합니다. 예를 들어 값을 만들기 `char` 위해 컴파일러는 낮은 순서의 바이트를 사용합니다. `wchar_t` 또는 `char16_t` 값을 만들기 위해 컴파일러는 하위 단어를 사용합니다. 컴파일러는 비트가 할당된 바이트 또는 단어 이상으로 설정된 경우 결과가 잘린다고 경고합니다.

C++

```

char c0      = 'abcd';      // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';        // C4305, C4309, truncates to '\x6364'
int i0      = 'abcd';        // 0x61626364

```

3자리 이상의 숫자를 포함하는 것처럼 보이는 8진수 이스케이프 시퀀스는 3자리 8진수 시퀀스로 처리된 다음, 이후 숫자가 다자간 리터럴의 문자로 처리되어 놀라운 결과를 얻을 수 있습니다. 예를 들면 다음과 같습니다.

C++

```
char c1 = '\100'; // '@'
char c2 = '\1000'; // C4305, C4309, truncates to '0'
```

8진수가 아닌 문자를 포함하는 것처럼 보이는 이스케이프 시퀀스는 마지막 8진수까지의 8진수 시퀀스로 평가되고 나머지 문자는 다자간 리터럴의 후속 문자로 평가됩니다. 첫 번째 8진수가 아닌 문자가 10진수이면 경고 C4125가 생성됩니다. 예를 들면 다음과 같습니다.

C++

```
char c3 = '\009'; // '9'
char c4 = '\089'; // C4305, C4309, truncates to '9'
char c5 = '\qrs'; // C4129, C4305, C4309, truncates to 's'
```

값이 보다 [\377](#) 높은 8진수 이스케이프 시퀀스로 인해 오류 C2022: '*value-in-decimal*' 문자가 너무 큽니다.

16진수 및 16진수 문자가 있는 것처럼 보이는 이스케이프 시퀀스는 마지막 16진수 문자 까지 16진수 이스케이프 시퀀스를 포함하고 16진수 문자가 아닌 문자를 포함하는 다자문자 리터럴로 평가됩니다. 16진수 숫자가 포함되지 않은 16진수 이스케이프 시퀀스는 컴파일러 오류 C2153을 발생시킵니다. "16진수 리터럴은 16진수 이상이어야 합니다."

C++

```
char c6 = '\x0050'; // 'P'
char c7 = '\x0pqr'; // C4305, C4309, truncates to 'r'
```

접두사로 L 접두사로 된 와이드 문자 리터럴에 다중 문자 시퀀스가 포함된 경우 값은 첫 번째 문자에서 가져온 것이고 컴파일러는 경고 C4066을 발생합니다. 후속 문자는 동일한 일반 다중 문자 리터럴의 동작과 달리 무시됩니다.

C++

```
wchar_t w1 = L'\100'; // L'@
wchar_t w2 = L'\1000'; // C4066 L'@', 0 ignored
wchar_t w3 = L'\009'; // C4066 L'\0', 9 ignored
wchar_t w4 = L'\089'; // C4066 L'\0', 89 ignored
wchar_t w5 = L'\qrs'; // C4129, C4066 L'q' escape, rs ignored
wchar_t w6 = L'\x0050'; // L'P'
wchar_t w7 = L'\x0pqr'; // C4066 L'\0', pqr ignored
```

Microsoft 관련 섹션은 여기서 끝납니다.

유니버설 문자 이름

문자 리터럴 및 네이티브(비 원시) 문자열 리터럴에서는 유니버설 문자 이름으로 모든 문자를 나타낼 수 있습니다. 범용 문자 이름은 접 `\u` 두사 뒤에 8자리 유니코드 코드 포인트 또는 접두 `\u` 사 뒤에 4자리 유니코드 코드 포인트로 구성됩니다. 올바른 형식의 유니버설 문자 이름을 만들려면 모든 8자리 또는 4자리 숫자가 각각 있어야 합니다.

C++

```
char u1 = 'A';           // 'A'  
char u2 = '\101';        // octal, 'A'  
char u3 = '\x41';         // hexadecimal, 'A'  
char u4 = '\u0041';        // \u UCN 'A'  
char u5 = '\U00000041'; // \U UCN 'A'
```

서로게이트 쌍

유니버설 문자 이름은 서로게이트 코드 포인트 범위 D800-DFFF의 값을 인코딩할 수 없습니다. 유니코드 서로게이트 쌍에 대해 `\UNNNNNNNN`을 사용하여 유니버설 문자 이름을 지정합니다. 여기서 NNNNNNNN은 문자에 대한 8자리 코드 포인트입니다. 컴파일러는 필요한 경우 서로게이트 쌍을 생성합니다.

C++03에서 언어는 문자 하위 집합만 해당 범용 문자 이름으로 나타낼 수 있도록 허용했으며 실제로 유효한 유니코드 문자를 나타내지 않는 일부 범용 문자 이름을 허용했습니다. 이 실수는 C++11 표준에서 수정되었습니다. C++11에서는 문자 및 문자열 리터럴과 식별자 모두 유니버설 문자 이름을 사용할 수 있습니다. 유니버설 문자 이름에 대한 자세한 내용은 [Character Sets](#)을 참조하세요. 유니코드에 대한 자세한 내용은 [유니코드](#)(영문)를 참조하세요. 서로게이트 쌍에 대한 자세한 내용은 [서로게이트 쌍 및 보조 문자](#)(영문)를 참조하세요.

문자열 리터럴

문자열 리터럴은 함께 사용되어 `null`로 끝나는 문자열을 형성하는 문자 시퀀스를 나타냅니다. 문자를 큰따옴표로 묶어야 합니다. 다음과 같은 종류의 문자열 리터럴이 있습니다.

좁은 문자열 리터럴

좁은 문자열 리터럴은 접두사로 구분된 큰따옴표로 구분된 `null`로 끝나는 형식 `const char[n]` 배열입니다. 여기서 `n`은 배열의 길이(바이트)입니다. 좁은 문자열 리터럴에는 큰따옴표("), 백슬래시(\) 또는 줄 바꿈 문자를 제외한 모든 그래픽 문자가 포함될 수 있습니다.

니다. 또한 좁은 문자열 리터럴에는 위에 나열된 이스케이프 시퀀스 및 바이트 크기에 맞는 유니버설 문자 이름이 포함될 수 있습니다.

C++

```
const char *narrow = "abcd";  
  
// represents the string: yes\no  
const char *escaped = "yes\\no";
```

UTF-8로 인코딩된 문자열

UTF-8로 인코딩된 문자열은 `u8` 접두사, 큰따옴표로 구분된 `null`로 끝나는 형식 `const char[n]` 배열입니다. 여기서 `n`은 인코딩된 배열의 길이(바이트)입니다. `u8` 접두사가 있는 문자열 리터럴에는 큰따옴표("), 백슬래시(\) 또는 줄 바꿈 문자를 제외한 모든 그래픽 문자가 포함될 수 있습니다. 또한 `u8` 접두사가 있는 문자열 리터럴에는 위에 나열된 이스케이프 시퀀스 및 모든 유니버설 문자 이름이 포함될 수 있습니다.

C++20에서는 이식 가능한 `char8_t` (UTF-8로 인코딩된 8비트 유니코드) 문자 형식이 도입되었습니다. C++20 `u8`에서 리터럴 접두사는 대신 `char`의 `char8_t` 문자 또는 문자열을 지정합니다.

C++

```
// Before C++20  
const char* str1 = u8"Hello World";  
const char* str2 = u8"\U0001F607 is 0:-)";  
// C++20 and later  
const char8_t* u8str1 = u8"Hello World";  
const char8_t* u8str2 = u8"\U0001F607 is 0:-)";
```

와이드 문자열 리터럴

와이드 문자열 리터럴은 `null`로 끝나는 상수 `wchar_t` 배열로, 접두사는 '`L`'이며 큰따옴표(), 백슬래시("\") 또는 줄 바꿈 문자를 제외한 모든 그래픽 문자를 포함합니다. 와이드 문자열 리터럴에는 위에 나열된 이스케이프 시퀀스 및 모든 유니버설 문자 이름이 포함될 수 있습니다.

C++

```
const wchar_t* wide = L"zyxw";  
const wchar_t* newline = L"hello\ngoodbye";
```

char16_t 및 char32_t(C++11)

C++11에서는 이식 가능한 `char16_t` (16비트 유니코드) 및 `char32_t` (32비트 유니코드) 문자 형식이 도입되었습니다.

C++

```
auto s3 = u"hello"; // const char16_t*
auto s4 = U"hello"; // const char32_t*
```

원시 문자열 리터럴(C++11)

원시 문자열 리터럴은 모든 문자 형식의 null로 끝나는 배열로, 큰따옴표(), 백슬래시(" ") 또는 줄 바꿈 문자를 비롯한 그래픽 문자를 포함합니다. 원시 문자열 리터럴은 문자 클래스를 사용하는 정규식과 HTML 문자열 및 XML 문자열에 종종 사용됩니다. 예제를 보려면 Bjarne Stroustrup의 [C++11에 대한 FAQ](#)(영문) 문서를 참조하세요.

C++

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
const char* raw_utf8a = u8R"(An unescaped \ character)"; // Before C++20
const char8_t* raw_utf8b = u8R"(An unescaped \ character)"; // C++20
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character);
```

구분 기호는 원시 문자열 리터럴의 여는 괄호 바로 앞에 닫는 괄호 바로 뒤에 오는 최대 16자의 사용자 정의 시퀀스입니다. 예를 들어 `R"abc(Hello"\()abc"`에서 구분 기호 시퀀스는 `abc`이고 문자열 콘텐츠는 `Hello"\()`입니다. 구분 기호를 사용하여 큰따옴표와 괄호를 모두 포함하는 원시 문자열을 구분할 수 있습니다. 이 문자열 리터럴은 컴파일러 오류를 발생합니다.

C++

```
// meant to represent the string: ")
const char* bad_parens = R"()""; // error C2059
```

그러나 구분 기호를 사용하면 오류가 해결됩니다.

C++

```
const char* good_parens = R"xyz()"xyz";
```

원본에서 줄 바꿈(이스케이프된 문자가 아님)이 포함된 원시 문자열 리터럴을 생성할 수 있습니다.

```
C++  
  
// represents the string: hello  
//goodbye  
const wchar_t* newline = LR"(hello  
goodbye);
```

std::string 리터럴(C++14)

std::string 리터럴은 (접미사가 있는) 로 "xyz"s 표시되는 사용자 정의 리터럴(s 아래 참조)의 표준 라이브러리 구현입니다. 이러한 종류의 문자열 리터럴은 지정된 접두사에 따라, std::wstring, std::u32string 또는 std::u16string 형식 std::string의 임시 개체를 생성합니다. 위와 같이 접두사를 사용하지 않으면 가 std::string 생성됩니다.

L"xyz"s 는 을 std::wstring 생성합니다. u"xyz"s 는 std::u16string을 생성하고

U"xyz"s std::u32string을 생성합니다.

```
C++  
  
//#include <string>  
//using namespace std::string_literals;  
string str{ "hello"s };  
string str2{ u8"Hello World" }; // Before C++20  
u8string u8str2{ u8"Hello World" }; // C++20  
wstring str3{ L"hello"s };  
u16string str4{ u"hello"s };  
u32string str5{ U"hello"s };
```

s 접미사는 원시 문자열 리터럴에서도 사용할 수 있습니다.

```
C++  
  
u32string str6{ UR"(She said \"hello.\")"s };
```

std::string 리터럴은 문자열> 헤더 파일의 네임스페이스 std::literals::string_literals 이<스에 정의됩니다. 및 는 모두 인라인 네임스페이스로 선언되므로 는 네임스페이스 std:: 이<스에 std::literals::string_literals 직접 속한 것처럼 자동으로 처리됩니다. std::literals std::literals::string_literals

문자열 리터럴의 크기

ANSI `char*` 문자열 및 기타 단일 바이트 인코딩(UTF-8 아님)의 경우 문자열 리터럴의 크기(바이트)는 종료되는 null 문자에 1을 더한 문자 수입니다. 다른 모든 문자열 형식의 경우 크기는 문자 수와 엄격하게 관련되지 않습니다. UTF-8은 최대 4개의 `char` 요소를 사용하여 일부 코드 단위를 인코딩하거나 `char16_t` 또는 `wchar_t` UTF-16으로 인코딩하면 두 개의 요소(총 4바이트)를 사용하여 단일 코드 단위를 인코딩할 수 있습니다. 이 예제에서는 와이드 문자열 리터럴의 크기(바이트)를 보여 줍니다.

C++

```
const wchar_t* str = L"Hello!";
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

`strlen()` 및 `wcslen()`에는 종결 null 문자의 크기가 포함되지 않습니다. 그 크기는 문자열 형식의 요소 크기(또는 `char8_t*` 문자열의 바이트 1바이트 `char*`, 또는 문자열의 바이트 2바이트 `char16_t*` `wchar_t*`, 문자열의 `char32_t*` 경우 4바이트)입니다.

Visual Studio 2022 버전 17.0 이전 버전의 Visual Studio에서 문자열 리터럴의 최대 길이는 65,535바이트입니다. 이 제한은 좁은 문자열 리터럴과 와이드 문자열 리터럴 모두에 적용됩니다. Visual Studio 2022 버전 17.0 이상에서는 이 제한이 해제되고 문자열 길이가 사용 가능한 리소스에 의해 제한됩니다.

문자열 리터럴 수정

문자열 리터럴(리터럴 포함 `std::string` 안 됨)은 상수이므로 수정 `str[2] = 'A'` 하려고 하면 컴파일러 오류가 발생합니다.

Microsoft 전용

Microsoft C++에서 문자열 리터럴을 사용하여 비 `const` `char` 또는 `wchar_t`에 대한 포인터를 초기화할 수 있습니다. 이 비구성 초기화는 C99 코드에서 허용되지만 C++98에서는 더 이상 사용되지 않으며 C++11에서 제거됩니다. 문자열을 수정하려고 하면 다음 예제와 같이 액세스 위반이 발생합니다.

C++

```
wchar_t* str = L"hello";
str[2] = L'a'; // run-time error: access violation
```

(문자열 리터럴 형식 변환 사용 안 함) 컴파일러 옵션을 설정할 `/Zc:strictStrings` 때 문자열 리터럴이 `const`가 아닌 문자 포인터로 변환될 때 컴파일러에서 오류가 발생할 수 있습니다. 표준 준수 휴대용 코드에 사용하는 것이 좋습니다. 또한 올바른 (`const`) 형식으로 확인

되므로 키워드를 사용하여 `auto` 문자열 리터럴 초기화된 포인터를 선언하는 것이 좋습니다. 예를 들어 다음 코드 예제는 컴파일 시간에 문자열 리터럴에 쓰려는 시도를 catch합니다.

C++

```
auto str = L"hello";
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

경우에 따라 실행 파일의 공간을 절약하기 위해 동일한 문자열 리터럴이 풀링될 수 있습니다. 문자열 리터럴 풀링에서 컴파일러는 각 참조가 문자열 리터럴의 별도 인스턴스를 가리키는 것이 아니라 특정 문자열 리터럴에 대한 모든 참조가 메모리의 같은 위치를 가리키게 합니다. 문자열 풀링을 사용하도록 설정하려면 컴파일러 옵션을 사용합니다 [/GF](#).

Microsoft 관련 섹션은 여기서 끝납니다.

인접 문자열 리터럴 연결

인접하는 와이드 문자열 리터럴 또는 좁은 문자열 리터럴은 연결됩니다. 다음 선언은

C++

```
char str[] = "12" "34";
```

다음 선언과 동일합니다.

C++

```
char atr[] = "1234";
```

또한 다음 선언과 동일합니다.

C++

```
char atr[] = "12\
34";
```

포함된 16진수 이스케이프 코드를 사용하여 문자열 리터럴을 지정하면 예기치 않은 결과가 발생할 수 있습니다. 다음 예제는 ASCII 5 문자와 f, i, v 및 e 문자가 포함된 문자열 리터럴을 만들려고 합니다.

C++

```
"\x05five"
```

실제 결과는 16진수 5F(ASCII 코드의 밑줄)와 i, v 및 e 문자입니다. 올바른 결과를 얻으려면 다음 이스케이프 시퀀스 중 하나를 사용할 수 있습니다.

C++

```
"\005five"      // Use octal literal.  
"\x05" "five"   // Use string splicing.
```

`std::string` 리터럴(및 관련 `std::u8string`, `std::u16string` 및 `std::u32string`)은 형식에 + 대해 정의된 `basic_string` 연산자와 연결할 수 있습니다. 또한 인접 문자열 리터럴과 동일한 방식으로 연결할 수도 있습니다. 두 경우 모두, 문자열 인코딩과 접미사가 다음과 일치해야 합니다.

C++

```
auto x1 = "hello" " " " world"; // OK  
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix  
auto x3 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

유니버설 문자 이름을 가진 문자열 리터럴

네이티브(비 원시) 문자열 리터럴은 문자열 형식에서 하나 이상의 문자로 유니버설 문자 이름을 인코드할 수 있는 한 유니버설 문자 이름을 사용하여 모든 문자를 나타낼 수 있습니다. 예를 들어 확장 문자를 나타내는 범용 문자 이름은 ANSI 코드 페이지를 사용하여 좁은 문자열에서 인코딩할 수 없지만 일부 다중 바이트 코드 페이지 또는 UTF-8 문자열 또는 와이드 문자열의 좁은 문자열로 인코딩할 수 있습니다. C++11에서 유니코드 지원은 및 `char32_t*` 문자열 형식에 의해 `char16_t*` 확장되고 C++20은 이를 형식으로 `char8_t` 확장합니다.

C++

```
// ASCII smiling face  
const char* s1 = ": - )";  
  
// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)  
const wchar_t* s2 = L": ) = \U0001F609 is ; - )";  
  
// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)  
const char* s3a = u8": ) = \U0001F607 is O:- )"; // Before C++20  
const char8_t* s3b = u8": ) = \U0001F607 is O:- )"; // C++20  
  
// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
```

```
const char16_t* s4 = u"\uD83D\uDC03 is :-D";  
// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)  
const char32_t* s5 = U"\uD83D\uDC03 is B-)";
```

추가 정보

[문자 집합](#)

[숫자, 부울 및 포인터 리터럴](#)

[사용자 정의 리터럴](#)

사용자 정의 리터럴

아티클 • 2023. 10. 12.

C++에는 정수, 문자, 부동 소수점, 문자열, 부울 및 포인터의 6가지 주요 범주가 있습니다. C++ 11부터 이러한 범주에 따라 고유한 리터럴을 정의하여 일반적인 관용구에 대한 구문 바로 가기를 제공하고 형식 안전을 높일 수 있습니다. 예를 들어 클래스가 `Distance` 있다고 가정해 보겠습니다. 킬로미터에 대한 리터럴과 마일에 대한 리터럴을 정의하고 사용자가 다음을 작성 `auto d = 42.0_km` 하여 측정 단위에 대해 명시하도록 장려할 `auto d = 42.0_mi` 수 있습니다. 사용자 정의 리터럴에는 성능 이점이나 단점이 없습니다. 주로 편의를 위해 또는 컴파일 시간 형식 추론을 위한 것입니다. 표준 라이브러리에는 크로노> 헤더의 시간 및 기간 작업 단위에 대한 `std::complex` 사용자 정의 리터럴 `std::string`이 <있습니다.

C++

```
Distance d = 36.0_mi + 42.0_km;           // Custom UDL (see below)
std::string str = "hello"s + "World"s;    // Standard Library <string> UDL
complex<double> num =
  (2.0 + 3.01i) * (5.0 + 4.3i);        // Standard Library <complex> UDL
auto duration = 15ms + 42h;               // Standard Library <chrono> UDLs
```

사용자 정의 리터럴 연산자 서명

다음 양식 중 하나를 사용하여 네임스페이스 범위에서 연산자""를 정의하여 사용자 정의 리터럴을 구현합니다.

C++

```
ReturnType operator "" _a(unsigned long long int); // Literal operator for
user-defined INTEGRAL literal
ReturnType operator "" _b(long double);           // Literal operator for
user-defined FLOATING literal
ReturnType operator "" _c(char);                  // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _d(wchar_t);                // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _e(char16_t);               // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _f(char32_t);               // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _g(const char*, size_t);   // Literal operator for
user-defined STRING literal
ReturnType operator "" _h(const wchar_t*, size_t); // Literal operator for
user-defined STRING literal
ReturnType operator "" _i(const char16_t*, size_t); // Literal operator for
```

```

user-defined STRING literal
ReturnType operator "" _g(const char32_t*, size_t); // Literal operator for
user-defined STRING literal
ReturnType operator "" _r(const char*); // Raw literal operator
template<char...> ReturnType operator "" _t(); // Literal operator
template

```

앞의 예제에서 연산자 이름은 사용자가 제공하는 이름에 대한 자리 표시자이지만 선행 밑줄이 필요합니다. (표준 라이브러리만 밑줄 없이 리터럴을 정의할 수 있습니다.) 반환 형식은 리터럴에서 수행하는 변환 또는 기타 작업을 사용자 지정하는 위치입니다. 또한 이러한 모든 연산자를 `constexpr`로 정의할 수 있습니다.

가공된 리터럴

소스 코드에서 사용자 정의 여부와 관계없이 모든 리터럴은 기본적으로 영숫자 문자(예: `101` 또는 또는 `54.7 "hello" true`)의 시퀀스입니다. 컴파일러는 시퀀스를 정수, `float`, `const char*` 문자열 등으로 해석합니다. 리터럴 값에 할당된 컴파일러가 어떤 형식이든 입력으로 허용하는 사용자 정의 리터럴을 비공식적으로 조리된 리터럴이라고 합니다. `_r` 및 `_t`를 제외한 위의 모든 연산자는 가공된 리터럴입니다. 예를 들어 리터럴 `42.0_km`는 `_b`와 유사한 서명을 가진 `_km`이라는 연산자에 바인딩하고 리터럴 `42_km`은 `_a`와 유사한 서명을 가진 연산자에 바인딩합니다.

다음 예제에서는 사용자 정의 리터럴을 통해 호출자에게 명시적으로 입력을 지정하도록 장려하는 방법을 보여 줍니다. `Distance`를 생성하려면 사용자가 적절한 사용자 정의 리터럴을 사용하여 킬로미터 또는 마일을 명시적으로 지정해야 합니다. 다른 방법으로 동일한 결과를 얻을 수 있지만 사용자 정의 리터럴은 대안보다 자세한 정보가 적습니다.

C++

```

// UDL_Distance.cpp

#include <iostream>
#include <string>

struct Distance
{
private:
    explicit Distance(long double val) : kilometers(val)
    {}

    friend Distance operator"" _km(long double val);
    friend Distance operator"" _mi(long double val);

    long double kilometers{ 0 };
public:
    const static long double km_per_mile;

```

```

long double get_kilometers() { return kilometers; }

Distance operator+(Distance other)
{
    return Distance(get_kilometers() + other.get_kilometers());
}
};

const long double Distance::km_per_mile = 1.609344L;

Distance operator"" _km(long double val)
{
    return Distance(val);
}

Distance operator"" _mi(long double val)
{
    return Distance(val * Distance::km_per_mile);
}

int main()
{
    // Must have a decimal point to bind to the operator we defined!
    Distance d{ 402.0_km }; // construct using kilometers
    std::cout << "Kilometers in d: " << d.get_kilometers() << std::endl; // 402

    Distance d2{ 402.0_mi }; // construct using miles
    std::cout << "Kilometers in d2: " << d2.get_kilometers() << std::endl; // 646.956

    // add distances constructed with different units
    Distance d3 = 36.0_mi + 42.0_km;
    std::cout << "d3 value = " << d3.get_kilometers() << std::endl; // 99.9364

    // Distance d4(90.0); // error constructor not accessible

    std::string s;
    std::getline(std::cin, s);
    return 0;
}

```

리터럴 번호는 10진수를 사용해야 합니다. 그렇지 않으면 숫자가 정수로 해석되고 형식이 연산자와 호환되지 않습니다. 부동 소수점 입력의 경우 형식은 이어야 `long double`이며 정수 계열 형식의 경우 형식이어야 `long long`입니다.

원시 리터럴

원시 사용자 정의 리터럴에서 정의하는 연산자는 리터럴을 문자 값 시퀀스로 허용합니다. 해당 시퀀스를 숫자나 문자열 또는 다른 형식으로 해석해야 합니다. 이 페이지의 앞부분

에 나온 연산자 목록에서 `_r` 및 `_t`는 원시 리터럴을 정의하는 데 사용할 수 있습니다.

C++

```
ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();      // Literal operator
template
```

원시 리터럴을 사용하여 컴파일러의 일반 동작과 다른 입력 시퀀스에 대한 사용자 지정 해석을 제공할 수 있습니다. 예를 들어 4.75987 시퀀스를 IEEE 754 부동 소수점 형식 대신 사용자 지정 10진수 형식으로 변환하는 리터럴을 정의할 수 있습니다. 조리된 리터럴과 같은 원시 리터럴은 입력 시퀀스의 컴파일 시간 유효성 검사에도 사용할 수 있습니다.

예: 원시 리터럴의 제한 사항

원시 리터럴 연산자 및 리터럴 연산자 템플릿은 다음 예제와 같이 정수 계열 및 부동 소수점 사용자 정의 리터럴에 대해서만 작동합니다.

C++

```
#include <cstddef>
#include <cstdio>

// Literal operator for user-defined INTEGRAL literal
void operator "" _dump(unsigned long long int lit)
{
    printf("operator \"\" _dump(unsigned long long int) : ===>%llu<===\n",
lit);
}

// Literal operator for user-defined FLOATING literal
void operator "" _dump(long double lit)
{
    printf("operator \"\" _dump(long double) : ===>%Lf<===\n",
lit);
}

// Literal operator for user-defined CHARACTER literal
void operator "" _dump(char lit)
{
    printf("operator \"\" _dump(char) : ===>%c<===\n",
lit);
}

void operator "" _dump(wchar_t lit)
{
    printf("operator \"\" _dump(wchar_t) : ===>%d<===\n",
lit);
}
```

```
void operator "" _dump(char16_t lit)
{
    printf("operator \"\" _dump(char16_t)           : ===>%d<===\n",
lit);
};

void operator "" _dump(char32_t lit)
{
    printf("operator \"\" _dump(char32_t)           : ===>%d<===\n",
lit);
};

// Literal operator for user-defined STRING literal
void operator "" _dump(const     char* lit, size_t)
{
    printf("operator \"\" _dump(const     char*, size_t): ===>%s<===\n",
lit);
};

void operator "" _dump(const wchar_t* lit, size_t)
{
    printf("operator \"\" _dump(const wchar_t*, size_t): ===>%ls<===\n",
lit);
};

void operator "" _dump(const char16_t* lit, size_t)
{
    printf("operator \"\" _dump(const char16_t*, size_t):\n"
);

};

void operator "" _dump(const char32_t* lit, size_t)
{
    printf("operator \"\" _dump(const char32_t*, size_t):\n"
);

};

// Raw literal operator
void operator "" _dump_raw(const char* lit)
{
    printf("operator \"\" _dump_raw(const char*)      : ===>%s<===\n",
lit);
};

template<char...> void operator "" _dump_template(); // Literal
operator template

int main(int argc, const char* argv[])
{
    42_dump;
    3.1415926_dump;
    3.14e+25_dump;
    'A'_dump;
    L'B'_dump;
```

```

u'C'_dump;
U'D'_dump;
    "Hello World"_dump;
    L"Wide String"_dump;
u8"UTF-8 String"_dump;
u"UTF-16 String"_dump;
U"UTF-32 String"_dump;
42_dump_raw;
3.1415926_dump_raw;
3.14e+25_dump_raw;

// There is no raw literal operator or literal operator template support
on these types:
// 'A'_dump_raw;
// L'B'_dump_raw;
// u'C'_dump_raw;
// U'D'_dump_raw;
// "Hello World"_dump_raw;
// L"Wide String"_dump_raw;
// u8"UTF-8 String"_dump_raw;
// u"UTF-16 String"_dump_raw;
// U"UTF-32 String"_dump_raw;
}

}

```

Output

```

operator "" _dump(unsigned long long int) : ===>42<===
operator "" _dump(long double) : ===>3.141593<===
operator "" _dump(long double) :
==>3139999999999998506827776.000000<===
operator "" _dump(char) : ===>A<===
operator "" _dump(wchar_t) : ===>66<===
operator "" _dump(char16_t) : ===>67<===
operator "" _dump(char32_t) : ===>68<===
operator "" _dump(const char*, size_t): ===>Hello World<===
operator "" _dump(const wchar_t*, size_t): ===>Wide String<===
operator "" _dump(const char*, size_t): ===>UTF-8 String<===
operator "" _dump(const char16_t*, size_t):
operator "" _dump(const char32_t*, size_t):
operator "" _dump_raw(const char*) : ===>42<===
operator "" _dump_raw(const char*) : ===>3.1415926<===
operator "" _dump_raw(const char*) : ===>3.14e+25<===

```

기본 개념(C++)

아티클 • 2024. 11. 21.

이 섹션에서는 C++를 이해하는 데 중요한 개념을 설명합니다. C 프로그래머는 이러한 많은 개념을 잘 알고 있지만 예기치 않은 프로그램 결과를 초래할 수 있는 몇 가지 미묘한 차이점이 있습니다. 주제는 다음과 같습니다.

- C++ 형식 시스템
- 범위
- 변환 단위 및 링크
- main 함수 및 명령줄 인수
- 프로그램 종료
- Lvalue 및 rvalue
- 임시 개체
- 맞춤
- 사소한 표준 레이아웃 및 POD 형식

참고 항목

C++ 언어 참조

피드백

이 페이지가 도움이 되었나요?

Yes

No

제품 사용자 의견 제공 | Microsoft Q&A에서 도움말 보기

C++ 형식 시스템

아티클 • 2023. 10. 12.

형식의 개념은 C++에서 중요합니다. 모든 변수, 함수 인수 및 함수 반환 값은 형식이 있어야 컴파일할 수 있습니다. 또한 모든 식(리터럴 값 포함)은 평가되기 전에 컴파일러에서 암시적으로 형식을 부여합니다. 형식의 몇 가지 예로는 정수 값을 저장하거나 부 `double` 동 소수점 값을 저장하는 것과 같은 `int` 기본 제공 형식 또는 텍스트를 저장할 클래스 `std::basic_string` 와 같은 표준 라이브러리 형식이 있습니다. 또는 `struct`을 정의하여 고유한 형식을 `class` 만들 수 있습니다. 형식은 변수(또는 식 결과)에 할당된 메모리 양을 지정합니다. 또한 이 형식은 저장할 수 있는 값의 종류, 컴파일러가 해당 값의 비트 패턴을 해석하는 방법 및 해당 값에 대해 수행할 수 있는 작업을 지정합니다. 이 문서에는 C++ 형식 시스템의 주요 기능에 대한 비공식적 개요가 들어 있습니다.

용어

스칼라 형식: 정의된 범위의 단일 값을 보유하는 형식입니다. 스칼라에는 산술 형식(정수 또는 부동 소수점 값), 열거형 형식 멤버, 포인터 형식, 포인터-멤버 형식 및 `std::nullptr_t`. 기본 형식은 일반적으로 스칼라 형식입니다.

복합 형식: 스칼라 형식이 아닌 형식입니다. 복합 형식에는 배열 형식, 함수 형식, 클래스 (또는 구조체) 형식, 공용 구조체 형식, 열거형, 참조 및 비정적 클래스 멤버에 대한 포인터가 포함됩니다.

변수: 데이터 수량의 기호화된 이름입니다. 이 이름은 정의된 코드 범위 전체에서 참조하는 데이터에 액세스하는 데 사용할 수 있습니다. C++에서 변수는 스칼라 데이터 형식의 인스턴스를 참조하는 데 자주 사용되는 반면, 다른 형식의 인스턴스는 일반적으로 개체라고 합니다.

개체: 단순성과 일관성을 위해 이 문서에서는 개체라는 용어를 사용하여 클래스 또는 구조체의 인스턴스를 참조합니다. 일반적으로 사용되는 경우 모든 형식, 심지어 스칼라 변수도 포함됩니다.

POD 형식 (일반 데이터): C++의 이 비공식적인 데이터 형식 범주는 스칼라(기본 형식 섹션 참조) 또는 POD 클래스인 형식을 나타냅니다. POD 클래스에는 POD가 아닌 정적 데이터 멤버가 없으며 사용자 정의 생성자, 사용자 정의 소멸자 또는 사용자 정의 할당 연산자가 없습니다. 또한 POD 클래스에는 가상의 함수, 기본 클래스 및 비공개 또는 보호된 비정적 데이터 멤버가 없습니다. POD 유형은 주로 외부 데이터 교환(예: POD 유형만 있는 C 언어로 작성된 모듈)에 사용됩니다.

변수 및 함수 형식 지정

C++는 강력한 형식의 언어와 정적으로 형식화된 언어입니다. 모든 개체에는 형식이 있으며 해당 형식은 변경되지 않습니다. 코드에서 변수를 선언할 때 해당 형식을 명시적으로 지정하거나 키워드(keyword) 사용하여 `auto` 컴파일러에 이니셜라이저에서 형식을 추론하도록 지시해야 합니다. 코드에서 함수를 선언할 때는 해당 반환 값의 형식과 각 인수의 형식을 지정해야 합니다. 함수에서 반환되는 값이 없는 경우 반환 값 형식 `void` 을 사용합니다. 예외는 임의의 형식의 인수를 허용하는 함수 템플릿을 사용하는 경우입니다.

변수를 처음 선언한 후에는 나중에 해당 형식을 변경할 수 없습니다. 그러나 변수의 값 또는 함수의 반환 값을 다른 형식의 다른 변수로 복사할 수 있습니다. 이러한 작업을 형식 변환이라고 하며, 때로는 필요하지만 데이터 손실 또는 올바르지 않은 잠재적 원인이기도 합니다.

POD 형식의 변수를 선언할 때 초기화하는 것이 좋습니다. 즉, 초기 값을 지정합니다. 변수를 초기화할 때까지 해당 메모리 위치에 이전에 있던 비트로 구성된 "가비지" 값을 가지고 있습니다. 특히 초기화를 처리하는 다른 언어에서 온 경우 기억해야 할 C++의 중요한 측면입니다. POD가 아닌 클래스 형식의 변수를 선언하면 생성자가 초기화를 처리합니다.

다음 예제에서는 각각에 대한 일부 설명을 포함한 몇 가지 간단한 변수 선언을 보여 줍니다. 또한 컴파일러가 특정 변수의 후속 작업을 허용하거나 거부하기 위해 형식 정보를 사용하는 방법을 보여줍니다.

C++

```
int result = 0;           // Declare and initialize an integer.
double coefficient = 10.8; // Declare and initialize a floating
                           // point value.
auto name = "Lady G.";   // Declare a variable and let compiler
                           // deduce the type.
auto address;            // error. Compiler cannot deduce a type
                           // without an initializing value.
age = 12;                // error. Variable declaration must
                           // specify a type or use auto!
result = "Kenny G.";    // error. Can't assign text to an int.
string result = "zero";  // error. Can't redefine a variable with
                           // new type.
int maxValue;            // Not recommended! maxValue contains
                           // garbage bits until it is initialized.
```

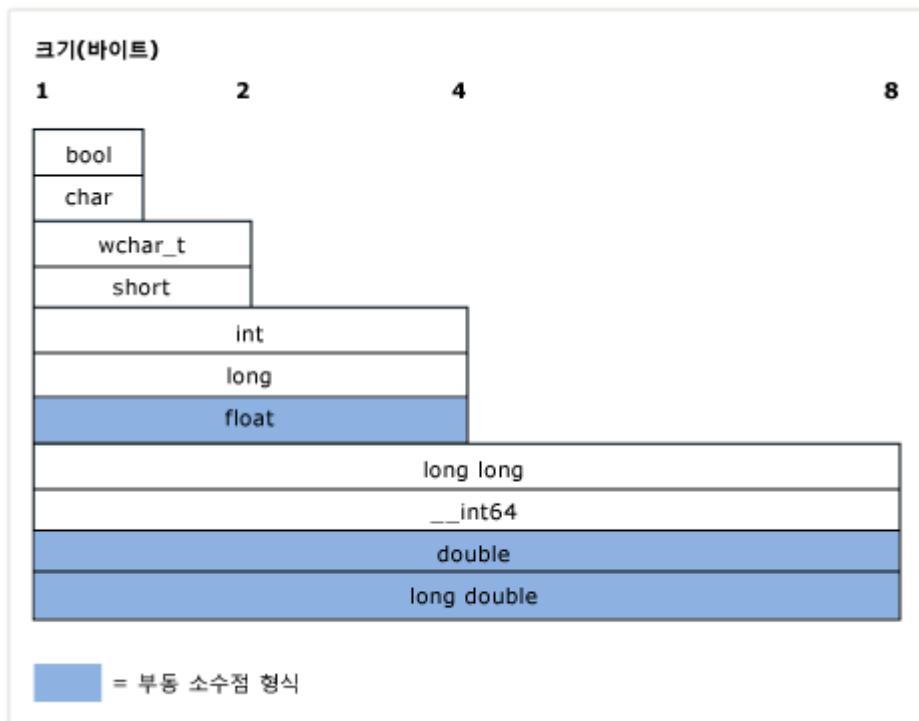
기본(기본 제공) 형식

일부 언어와 달리, C++에는 다른 형식이 파생되는 유니버설 기본 형식이 없습니다. 언어에는 기본 제공 형식이라고도 하는 많은 기본 형식이 포함됩니다. 이러한 형식에는 각각 ASCII 및 `wchar_t` UNICODE 문자에 `char` 대한 숫자 형식(예: `int`, `double` `long`, `bool` 및 형식)이 포함됩니다. 대부분의 정수 계열 기본 형식(`double`, `wchar_t` 및 관련 형식 제외

`bool`)에는 `unsigned` 모두 변수가 저장할 수 있는 값 범위를 수정하는 버전이 있습니다. 예를 들어 `int` 부가된 32비트 정수는 -2,147,483,648에서 2,147,483,647까지의 값을 나타낼 수 있습니다. 32비트로도 저장되는 이 `unsigned int` 값은 0에서 4,294,967,295까지 저장할 수 있습니다. 각 사례에서 사용할 수 있는 값의 총 수는 동일하며, 범위만 다릅니다.

컴파일러는 이러한 기본 제공 형식을 인식하며, 이러한 형식에 대해 수행할 수 있는 작업과 다른 기본 형식으로 변환할 수 있는 방법을 제어하는 기본 제공 규칙이 있습니다. 기본 제공 형식 및 해당 크기 및 숫자 제한의 전체 목록은 기본 제공 형식을 참조하세요.

다음 그림에서는 Microsoft C++ 구현에서 기본 제공 형식의 상대적 크기를 보여 줍니다.



다음 표에서는 Microsoft C++ 구현에서 가장 자주 사용되는 기본 형식 및 해당 크기를 나열합니다.

Type	크기	설명
<code>int</code>	4바이트	정수 값에 대한 기본 선택입니다.
<code>double</code>	8바이트	부동 소수점 값에 대한 기본 선택입니다.
<code>bool</code>	1바이트	true 또는 false가 될 수 있는 값을 나타냅니다.

Type	크기	설명
<code>char</code>	1바이트	UNICODE로 변환되지 않는 이전 C 스타일 문자열 또는 <code>std::string</code> 개체의 ASCII 문자에 사용합니다.
<code>wchar_t</code>	2바이트	UNICODE 형식(Windows의 경우 UTF-16, 운영 체제마다 다를 수 있음)으로 인코딩할 수 있는 "와이드" 문자 값을 나타냅니다. <code>wchar_t</code> 는 형식의 문자열에 사용되는 문자 형식 <code>std::wstring</code> 입니다.
<code>unsigned char</code>	1바이트	C++에는 기본 제공 바이트 형식이 없습니다. 바이트 값을 나타내는 데 사용합니다 <code>unsigned char</code> .
<code>unsigned int</code>	4바이트	비트 플래그에 대한 기본 선택입니다.
<code>long long</code>	8바이트	훨씬 더 큰 정수 값 범위를 나타냅니다.

다른 C++ 구현은 특정 숫자 형식에 서로 다른 크기를 사용할 수 있습니다. C++ 표준에 필요한 크기 및 크기 관계에 대한 자세한 내용은 기본 제공 형식을 참조 [하세요](#).

void 형식

형식은 `void` 특수 형식입니다. 형식 `void`의 변수를 선언할 수는 없지만 형식 변수 `void *` (포인터)를 선언할 `void` 수 있습니다. 이 변수는 원시(형식화되지 않은) 메모리를 할당할 때 필요한 경우가 있습니다. 그러나 포인터는 `void` 형식이 안전하지 않으며 최신 C++에서는 사용하지 않는 것이 좋습니다. 함수 선언 `void`에서 반환 값은 함수가 값을 반환하지 않음을 의미합니다. 반환 형식으로 사용하는 것이 일반적이며 허용 가능한 용도 `void`입니다. 예를 들어 `fn(void)` C 언어에서는 매개 변수 목록에서 선언 `void` 할 매개 변수가 없는 함수가 필요하지만 최신 C++에서는 이 방법을 사용하지 않는 것이 좋습니다. 매개 변수가 없는 함수는 선언 `fn()` 해야 합니다. 자세한 내용은 형식 변환 및 형식 안전을 참조하세요.

const 형식 한정자

모든 기본 제공 또는 사용자 정의 형식은 키워드(keyword) 의해 `const` 정규화될 수 있습니다. 또한 멤버 함수는 정규화되고 심지어 오버로드될 `const` 수도 `const` 있습니다. 형식의 `const` 값은 초기화된 후에 수정할 수 없습니다.

```
const double PI = 3.1415;
PI = .75; //Error. Cannot modify const variable.
```

한정자는 `const` 함수 및 변수 선언에서 광범위하게 사용되며 "const 정확성"은 C++의 중요한 개념입니다. 기본적으로 컴파일 시 값이 의도치 않게 수정되지 않도록 보장하는 데 사용됩니다 `const`. 자세한 내용은 [const](#)를 참조하세요.

`const` 형식은 버전이 아닌 형식과 다릅니다 `const`. 예를 들어 `const int int`. 변수에서 `const-ness`를 제거해야 하는 드문 경우에서 C++ `const_cast` 연산자를 사용할 수 있습니다. 자세한 내용은 형식 변환 및 형식 안전을 [참조하세요](#).

String 형식

엄밀히 말하면 C++ 언어에는 기본 제공 문자열 형식이 없습니다. `char` 및 `wchar_t` 단일 문자 저장 - 마지막 유효한 문자(C 스타일 문자열이라고도 함)를 지나서 배열 요소에 종결 null 값(예: ASCII '\0')을 추가하여 문자열 근사치로 이러한 형식의 배열을 선언해야 합니다. 훨씬 더 많은 코드를 작성해야하거나 외부 문자열 유ти리티 라이브러리 함수를 사용해야 하는 C 스타일 문자열입니다. 그러나 최신 C++에서는 표준 라이브러리 형식

`std::string` (8비트 `char`-type 문자열의 경우) 또는 `std::wstring` (16비트 `wchar_t`-type 문자열의 경우) 이러한 C++ 표준 라이브러리 컨테이너는 모든 준수 C++ 빌드 환경에 포함된 표준 라이브러리의 일부이기 때문에 네이티브 문자열 형식으로 간주할 수 있습니다. 지시문을 `#include <string>` 사용하여 프로그램에서 이러한 형식을 사용할 수 있도록 합니다. (MFC 또는 ATL `cString` 을 사용하는 경우 클래스도 사용할 수 있지만 C++ 표준에는 속하지 않습니다.) null로 끝나는 문자 배열(이전에 멘션 C 스타일 문자열)을 사용하는 것은 최신 C++에서 권장되지 않습니다.

사용자 정의 형식

또는 `struct` `union` `enum` 해당 구문을 정의 `class` 할 때 해당 구문은 기본 형식인 것처럼 코드의 나머지 부분에 사용됩니다. 메모리 크기는 잘 알려져 있고, 사용법에 관한 규정이 컴파일 시간 검사, 가동 시 프로그램 수명 점검을 위해 적용됩니다. 기본 제공 형식과 사용자 정의 형식 간 기본적 차이는 다음과 같습니다.

- 컴파일러는 기본적으로 사용자 정의 형식을 인식하지 못합니다. 컴파일 프로세스 중에 정의가 처음 발견되면 형식에 대해 알아봅니다.
- 오버로드를 통해 적합한 연산자를 클래스 멤버 또는 비멤버 함수로 정의하여 해당 형식에서 수행할 수 있는 연산자와 다른 형식으로 변환할 수 있는 방법을 지정합니다.

다. 자세한 내용은 함수 오버로드를 참조하세요.

포인터 형식

C 언어의 초기 버전과 마찬가지로 C++는 특수 선언자 `*` (별표)를 사용하여 포인터 형식의 변수를 선언할 수 있도록 계속합니다. 포인터 형식은 메모리에서 실제 데이터 값이 저장되는 위치의 주소를 저장합니다. 최신 C++에서는 이러한 포인터 형식을 원시 포인터라고 하며 특수 연산자 `*` (별표) 또는 `->` (보다 크고 자주 화살표라고도 하는 대시)를 통해 코드에서 액세스합니다. 이 메모리 액세스 작업을 역참조라고 합니다. 사용하는 연산자는 스칼라에 대한 포인터 또는 개체의 멤버에 대한 포인터를 역참조하는지에 따라 달라집니다.

포인터 형식 사용은 C 및 C++ 프로그램 개발 환경에서 가장 까다롭고 복잡한 요소 중 하나였습니다. 이 섹션에서는 원하는 경우 원시 포인터를 사용하는 데 도움이 되는 몇 가지 사실과 사례를 간략하게 설명합니다. 그러나 최신 C++에서는 스마트 포인터의 진화 ([이 섹션의 끝에 자세히 설명됨](#))로 인해 개체 소유권에 원시 포인터를 사용하는 것이 더 이상 필요하지 않습니다(또는 권장). 개체를 관찰하기 위해 원시 포인터를 사용하는 것은 여전히 유용하고 안전합니다. 그러나 개체 소유권에 사용해야 하는 경우 소유하는 개체를 만들고 소멸하는 방법을 신중하게 고려해야 합니다.

가장 먼저 알아야 할 것은 원시 포인터 변수 선언이 주소를 저장할 충분한 메모리만 할당한다는 것입니다. 포인터가 역참조될 때 참조하는 메모리 위치입니다. 포인터 선언은 데이터 값을 저장하는 데 필요한 메모리를 할당하지 않습니다. (해당 메모리를 백업 저장소라고도 합니다.) 즉, 원시 포인터 변수를 선언하여 실제 데이터 변수가 아닌 메모리 주소 변수를 만듭니다. 포인터 변수에 백업 저장소에 대한 유효한 주소가 포함되어 있는지 확인하기 전에 포인터 변수를 역참조하는 경우 프로그램에서 정의되지 않은 동작(일반적으로 심각한 오류)이 발생합니다. 다음 예제에서는 이런 종류의 오류를 보여 줍니다.

C++

이 예제는 실제 정수 데이터 또는 할당된 유효한 메모리 주소를 저장하기 위해 할당된 메모리 없이 포인터를 역참조합니다. 다음은 다음 오류를 해결한 코드입니다.

C++

```

int* pNumber = &number;    // Declare and initialize a local integer
                         // pointer variable to a valid memory
                         // address to that backing store.
...
*pNumber = 41;           // Dereference and store a new value in
                         // the memory pointed to by
                         // pNumber, the integer variable called
                         // "number". Note "number" was changed, not
                         // "pNumber".

```

수정된 코드 예제는 로컬 스택 메모리를 사용하여 `pNumber` 가 가리키는 백업 저장소를 만듭니다. 간단히 기본 형식을 사용합니다. 실제로 포인터에 대한 백업 저장소는 키워드(keyword) 식을 사용하여 `new` 힙(또는 자유 저장소)이라는 메모리 영역에 동적으로 할당되는 사용자 정의 형식입니다(C 스타일 프로그래밍에서는 이전 `malloc()` C 런타임 라이브러리 함수가 사용됨). 할당되면 이러한 변수를 일반적으로 개체라고 하며, 특히 클래스 정의를 기반으로 하는 경우 해당 변수를 개체라고 합니다. 할당된 `new` 메모리는 해당 `delete` 문(또는 함수를 사용하여 `malloc()` 할당한 경우 C 런타임 함수 `free()`)에 의해 삭제되어야 합니다.

그러나 동적으로 할당된 개체를 삭제하는 것은 잊기 쉽습니다. 특히 복잡한 코드에서는 메모리 누수라는 리소스 버그가 발생합니다. 이러한 이유로 원시 포인터의 사용은 최신 C++에서 권장되지 않습니다. 소멸자가 호출될 때 자동으로 메모리를 해제하는 스마트 포인터에 원시 포인터를 래핑하는 것이 거의 항상 좋습니다. 즉, 코드가 스마트 포인터의 범위를 벗어나는 경우입니다. 스마트 포인터를 사용하면 C++ 프로그램에서 전체 버그 클래스를 사실상 제거합니다. 다음 예에서 `MyClass` 는 `DoSomeWork();` 의 공용 메서드를 가진 사용자 정의 형식입니다.

C++

```

void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
// No memory leak. Out-of-scope automatically calls the destructor
// for the unique_ptr, freeing the resource.

```

스마트 포인터에 대한 자세한 내용은 스마트 포인터를 참조하세요.

포인터 변환에 대한 자세한 내용은 형식 변환 및 형식 안전을 참조하세요.

일반적인 포인터에 대한 자세한 내용은 포인터를 참조하세요.

Windows 데이터 형식

C 및 C++용 클래식 Win32 프로그래밍에서 대부분의 함수는 Windows 관련 `typedef` 및 `#define` 매크로(정의 `windef.h` 됨)를 사용하여 매개 변수 형식과 반환 값을 지정합니다. 이러한 Windows 데이터 형식은 주로 C/C++ 기본 제공 형식에 지정된 특수 이름(별칭)입니다. 이러한 `typedef` 및 전처리기 정의의 전체 목록은 Windows 데이터 형식을 참조 [하세요](#). 이러한 `typedef`(예: `HRESULT` 및 `LCID`)는 유용하고 설명적입니다. 같은 `INT` 다른 항목은 특별한 의미가 없으며 기본 C++ 형식에 대한 별칭일 뿐입니다. 그 외 Windows 데이터 유형은 C 프로그래밍 및 16비트 프로세서 시기부터 내려온 이름을 그대로 가지고 있으며 최신 하드웨어 또는 운영 체제에 다른 목적과 의미를 가지고 있지 않습니다. Windows 런타임 라이브러리와 연결된 특수 데이터 형식도 Windows 런타임 기본 데이터 형식[으로](#) 나열됩니다. 최신 C++에서 일반적인 지침은 Windows 형식이 값을 해석하는 방법에 대한 추가적인 의미를 전달하지 않는 한 C++ 기본 형식을 선호하는 것입니다.

자세한 정보

C++ 형식 시스템에 대한 자세한 내용은 다음 문서를 참조하세요.

[값 형식](#)

사용과 관련된 문제와 함께 값 형식에 대해 설명합니다.

[형식 변환 및 형식 안전성](#)

일반적인 형식 변환 문제를 설명하고 이러한 문제를 방지하는 방법을 보여 줍니다.

참고 항목

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

범위 (C++)

아티클 • 2023. 10. 12.

클래스, 함수 또는 변수와 같은 프로그램 요소를 선언하는 경우 해당 이름은 프로그램의 특정 부분에서만 "표시"되고 사용될 수 있습니다. 이름이 표시되는 컨텍스트를 해당 범위라고 합니다. 예를 들어 함수 `x` 내에서 변수 `x` 를 선언하는 경우 해당 함수 본문 내에서만 표시됩니다. 로컬 범위가 있습니다. 프로그램에 같은 이름의 다른 변수가 있을 수 있습니다. 범위가 다르면 하나의 정의 규칙을 위반하지 않으며 오류가 발생하지 않습니다.

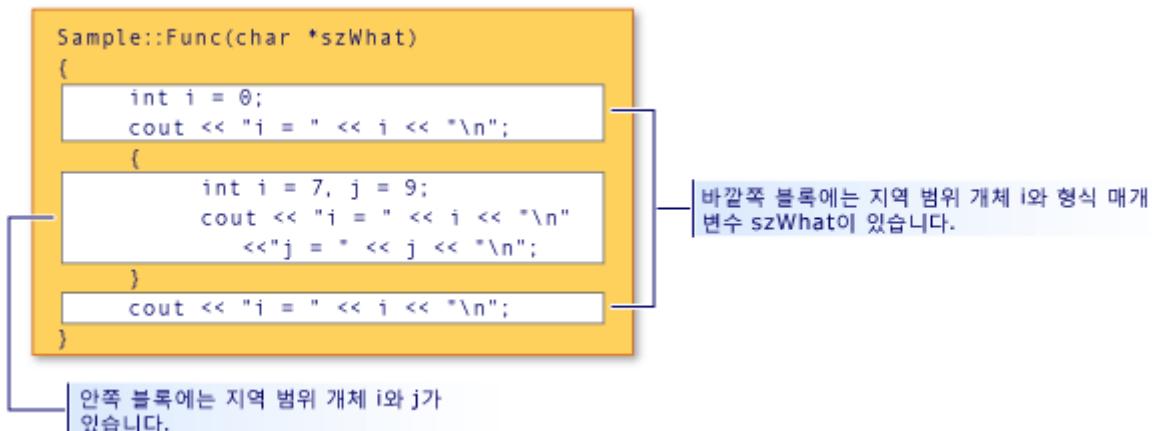
자동 비정적 변수의 경우 범위는 프로그램 메모리에서 생성 및 소멸되는 시기도 결정합니다.

범위는 6가지입니다.

- **전역 범위** A 전역 이름은 클래스, 함수 또는 네임스페이스 외부에서 선언된 이름입니다. 그러나 C++에서도 이러한 이름은 암시적 전역 네임스페이스와 함께 존재합니다. 전역 이름의 범위는 선언 지점에서 선언된 파일의 끝까지 확장됩니다. 전역 이름의 경우 표시 유형은 프로그램의 다른 파일에 이름이 표시되는지 여부를 결정하는 연결 규칙에 의해 제어됩니다.
- **네임스페이스 범위** 클래스 또는 열거형 정의 또는 함수 블록 외부의 네임스페이스 [내에서](#) 선언된 이름은 선언 지점에서 네임스페이스 끝까지 표시됩니다. 네임스페이스는 여러 파일에서 여러 블록으로 정의될 수 있습니다.
- **로컬 범위** 매개 변수 이름을 포함하여 함수 또는 람다 내에 선언된 이름에는 로컬 범위가 있습니다. "지역 주민"이라고도 합니다. 선언 지점에서 함수 또는 람다 본문의 끝까지만 표시됩니다. 로컬 범위는 이 문서의 뒷부분에서 설명하는 일종의 블록 범위입니다.
- **클래스 범위 멤버의 클래스 범위** 이름은 선언 지점에 관계없이 클래스 정의 전체에 걸쳐 확장되는 클래스 범위를 갖습니다. 클래스 멤버 접근성은, `private` 및 `protected` 키워드(keyword) 의해 `public` 추가로 제어됩니다. 공용 또는 보호된 멤버는 멤버 선택 연산자(또는 -) 또는 멤버에 대한 포인터 연산자(*또는 ->*)를 사용해야만 액세스할 수 있습니다.>
- **문 범위** 이름, 또는 `while` `switch` 문에 `for` `if` 선언된 이름은 문 블록이 끝날 때까지 표시됩니다.
- **함수 범위** A [레이블](#) 에는 함수 범위가 있습니다. 즉, 선언 지점 이전에도 함수 본문 전체에 표시됩니다. 함수 범위를 사용하면 레이블이 선언되기 전 `cleanup` 과 같은 `goto cleanup` 문을 작성할 수 있습니다.

이름 숨기기

이름을 포함된 블록에서 선언하여 숨길 수 있습니다. 다음 그림에서는 `i`가 내부 블록 안에서 다시 선언되므로 바깥쪽 블록 범위에서 `i`와 연결된 변수가 숨겨집니다.



차단 범위 및 이름 숨기기

그림에 표시된 프로그램의 출력은 다음과 같습니다.

C++

```
i = 0
i = 7
j = 9
i = 0
```

① 참고

`szWhat` 인수는 함수 범위에 있는 것으로 간주되므로 함수의 가장 바깥쪽 블록에서 선언된 것처럼 취급됩니다.

클래스 이름 숨기기

함수, 개체, 변수 또는 열거자를 동일한 코드에서 선언하여 클래스 이름을 숨길 수 있습니다. 그러나 키워드(keyword) `class` 접두사로 클래스 이름에 계속 액세스할 수 있습니다.

C++

```
// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at global scope.
```

```

class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
private:
    double balance;
};

double Account = 15.37;           // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                         // class name

    cout << "Opening account with a balance of: "
        << Checking.GetBalance() << "\n";
}
//Output: Opening account with a balance of: 15.37

```

① 참고

클래스 이름(Account)이 호출되는 모든 위치에서 키워드(keyword) 클래스를 사용하여 전역 범위 변수 계정과 구분해야 합니다. 이 규칙은 범위 결정 연산자(:)의 왼쪽에 클래스 이름이 나타나는 경우에는 적용되지 않습니다. 범위 결정 연산자의 왼쪽에 있는 이름은 항상 클래스 이름으로 간주됩니다.

다음 예제에서는 키워드(keyword) 사용하여 형식 Account 개체에 대한 포인터를 선언하는 **class** 방법을 보여 줍니다.

C++

```
class Account *Checking = new class Account( Account );
```

앞의 문에 있는 이 Account 니셜라이저의 괄호 안에는 전역 범위가 있습니다. 형식 **double**입니다.

① 참고

이 예제와 같이 식별자 이름을 다시 사용하는 것은 좋지 않은 프로그래밍 스타일로 간주됩니다.

클래스 개체의 선언 및 초기화에 대한 자세한 내용은 클래스, 구조체 및 공용 구조체를 참조 [하세요](#). 무료 저장소 연산자 및 `delete` 자유 저장소 연산자 `new` 사용에 대한 자세한 내용은 새 연산자 및 삭제 연산자를 [참조하세요](#).

전역 범위를 사용하여 이름 숨기기

블록 범위에서 동일한 이름을 명시적으로 선언하여 전역 범위의 이름을 숨길 수 있습니다. 그러나 범위 확인 연산자(`::`)를 사용하여 전역 범위 이름에 액세스할 수 있습니다.

C++

```
#include <iostream>

int i = 7;      // i has global scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
    int i = 5;    // i has block scope, hides i at global scope
    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "Global-scoped i has the value: " << ::i << "\n";
}
```

Output

```
Block-scoped i has the value: 5
Global-scoped i has the value: 7
```

참고 항목

[기본 개념](#)

헤더 파일(C++)

아티클 • 2024. 11. 21.

변수, 함수, 클래스 등과 같은 프로그램 요소의 이름을 선언해야 사용할 수 있습니다. 예를 들어 먼저 'x'를 선언하지 않고는 작성 `x = 42` 할 수 없습니다.

C++

```
int x; // declaration  
x = 42; // use x
```

이 선언은 요소가 요소 `int`인지, 함수 `double`인지, 아니면 다른 요소인지를 컴파일러에 `class` 알려줍니다. 또한 사용되는 모든 .cpp 파일에서 각 이름을 직접 또는 간접적으로 선언해야 합니다. 프로그램을 컴파일할 때 각 .cpp 파일은 컴파일 단위로 독립적으로 컴파일됩니다. 컴파일러는 다른 컴파일 단위에서 선언된 이름을 알지 않습니다. 즉, 클래스 또는 함수 또는 전역 변수를 정의하는 경우 해당 변수를 사용하는 각 추가 .cpp 파일에 해당 항목의 선언을 제공해야 합니다. 해당 항목의 각 선언은 모든 파일에서 정확히 동일해야 합니다. 링커가 모든 컴파일 단위를 단일 프로그램에 병합하려고 할 때 약간의 불일치로 인해 오류 또는 의도하지 않은 동작이 발생합니다.

오류 가능성을 최소화하기 위해 C++는 헤더 파일을 사용하여 선언을 포함하는 규칙을 채택했습니다. 헤더 파일에서 선언을 만든 다음 모든 .cpp 파일 또는 해당 선언이 필요한 다른 헤더 파일에서 `#include` 지시문을 사용합니다. `#include` 지시문은 컴파일하기 전에 헤더 파일의 복사본을 .cpp 파일에 직접 삽입합니다.

① 참고

Visual Studio 2019에서 C++20 모듈 기능은 헤더 파일을 개선하고 최종적으로 대체하는 기능으로 도입되었습니다. 자세한 내용은 [C++의 모듈 개요를 참조하세요](#).

예시

다음 예제에서는 클래스를 선언한 다음 다른 소스 파일에서 사용하는 일반적인 방법을 보여 있습니다. 헤더 파일 `my_class.h`로 시작하겠습니다. 클래스 정의가 포함되어 있지만 정의가 불완전합니다. 멤버 함수 `do_something` 가 정의되지 않았습니다.

C++

```
// my_class.h  
namespace N
```

```
{  
    class my_class  
    {  
        public:  
            void do_something();  
    };  
  
}
```

다음으로, 구현 파일(일반적으로 .cpp 또는 유사한 확장명 포함)을 만듭니다.

my_class.cpp 파일을 호출하고 멤버 선언에 대한 정의를 제공합니다. .cpp 파일의 이 시점에서 my_class 선언을 삽입하기 위해 "my_class.h" 파일에 대한 `std::cout` 지시문을 추가하고 `#include` 선언을 끌어오도록 포함합니다 `<iostream>`. 따옴표는 원본 파일과 동일한 디렉터리의 헤더 파일에 사용되며 꺾쇠 괄호는 표준 라이브러리 헤더에 사용됩니다. 또한 많은 표준 라이브러리 헤더에는 .h 또는 다른 파일 확장명이 없습니다.

구현 파일에서 필요에 따라 문을 사용하여 `using "my_class"` 또는 "cout"에 대한 모든 멤션을 "N::" 또는 "std::"로 한정하지 않아도 됩니다. 헤더 파일에 문을 넣지 `using` 마세요.

C++

```
// my_class.cpp  
#include "my_class.h" // header in local directory  
#include <iostream> // header in standard library  
  
using namespace N;  
using namespace std;  
  
void my_class::do_something()  
{  
    cout << "Doing something!" << endl;  
}
```

이제 다른 .cpp 파일에서 사용할 `my_class` 수 있습니다. 컴파일러가 선언을 끌어오도록 헤더 파일을 `#include`. 컴파일러에서 알아야 할 모든 것은 my_class 공용 멤버 함수가 호출 `do_something()` 된 클래스라는 것입니다.

C++

```
// my_program.cpp  
#include "my_class.h"  
  
using namespace N;  
  
int main()  
{  
    my_class mc;  
    mc.do_something();
```

```
    return 0;  
}
```

컴파일러가 각 .cpp 파일을 .obj 파일로 컴파일한 후 .obj 파일을 링커에 전달합니다. 링커가 개체 파일을 병합하면 my_class 대한 정의가 하나만 발견됩니다. my_class.cpp 대해 생성된 .obj 파일에 있으며 빌드가 성공합니다.

가드 포함

일반적으로 헤더 파일에는 단일 .cpp 파일에 여러 번 삽입되지 않도록 하는 포함 가드 또는 `#pragma once` 지시문이 있습니다.

```
C++  
  
// my_class.h  
#ifndef MY_CLASS_H // include guard  
#define MY_CLASS_H  
  
namespace N  
{  
    class my_class  
    {  
        public:  
            void do_something();  
    };  
}  
  
#endif /* MY_CLASS_H */
```

헤더 파일에 넣을 내용

헤더 파일은 잠재적으로 여러 파일에 포함될 수 있으므로 동일한 이름의 여러 정의를 생성할 수 있는 정의를 포함할 수 없습니다. 다음은 허용되지 않거나 매우 나쁜 사례로 간주됩니다.

- 네임스페이스 또는 전역 범위의 기본 제공 형식 정의
- 인라인이 아닌 함수 정의
- 비 const 변수 정의
- 집계 정의
- 명명되지 않은 네임스페이스
- using 지시문

`using` 지시문을 사용하면 반드시 오류가 발생하는 것은 아니지만 해당 헤더를 직접 또는 간접적으로 포함하는 모든 .cpp 파일의 범위로 네임스페이스를 가져오기 때문에 잠재적

으로 문제가 발생할 수 있습니다.

샘플 헤더 파일

다음 예제에서는 헤더 파일에서 허용되는 다양한 종류의 선언 및 정의를 보여 줍니다.

C++

```
// sample.h
#pragma once
#include <vector> // #include directive
#include <string>

namespace N // namespace declaration
{
    inline namespace P
    {
        //...
    }

    enum class colors : short { red, blue, purple, azure };

    const double PI = 3.14; // const and constexpr definitions
    constexpr int MeaningOfLife{ 42 };
    constexpr int get_meaning()
    {
        static_assert(MeaningOfLife == 42, "unexpected!"); // static_assert
        return MeaningOfLife;
    }
    using vstr = std::vector<int>; // type alias
    extern double d; // extern variable

#define LOG // macro definition

#ifndef LOG // conditional compilation directive
    void print_to_log();
#endif

    class my_class // regular class definition,
    { // but no non-inline function definitions

        friend class other_class;
    public:
        void do_something(); // definition in my_class.cpp
        inline void put_value(int i) { vals.push_back(i); } // inline OK

    private:
        vstr vals;
        int i;
    };

    struct RGB
```

```
{  
    short r{ 0 }; // member initialization  
    short g{ 0 };  
    short b{ 0 };  
};  
  
template <typename T> // template definition  
class value_store  
{  
public:  
    value_store<T>() = default;  
    void write_value(T val)  
    {  
        //... function definition OK in template  
    }  
private:  
    std::vector<T> vals;  
};  
  
template <typename T> // template declaration  
class value_widget;  
}
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

변환 단위 및 링크

아티클 • 2023. 10. 12.

C++ 프로그램에서 는 변수 또는 함수 이름과 같은 기호를 범위 내에서 여러 번 선언할 수 있습니다. 그러나 한 번만 정의할 수 있습니다. 이 규칙은 "ODR(하나의 정의 규칙)"입니다. 선언은 나중에 이름을 정의와 연결할 수 있는 충분한 정보와 함께 이름을 프로그램에 도입(또는 다시 도입)합니다. 정의는 이름을 소개하고 이름을 만드는 데 필요한 모든 정보를 제공합니다. 이름이 변수를 나타내는 경우 정의는 명시적으로 스토리지를 만들고 초기화합니다. 함수 정의는 시그니처와 함수 본문으로 구성됩니다. 클래스 정의는 클래스 이름 뒤에 모든 클래스 멤버를 나열하는 블록으로 구성됩니다. (멤버 함수의 본문은 필요에 따라 다른 파일에서 별도로 정의될 수 있습니다.)

다음 예제에서는 몇 가지 선언을 보여 줍니다.

```
C++  
  
int i;  
int f(int x);  
class C;
```

다음 예제에서는 몇 가지 정의를 보여 줍니다.

```
C++  
  
int i{42};  
int f(int x){ return x * i; }  
class C {  
public:  
    void DoSomething();  
};
```

프로그램은 하나 이상의 번역 단위로 구성됩니다. 번역 단위는 구현 파일과 직접 또는 간접적으로 포함하는 모든 헤더로 구성됩니다. 구현 파일에는 일반적으로 파일 확장명이 `.cpp` 있거나 `.cxx`. 헤더 파일에는 일반적으로 확장명이 `.h` 있거나 `.hpp`. 각 번역 단위는 컴파일러에 의해 독립적으로 컴파일됩니다. 컴파일이 완료되면 링커는 컴파일된 번역 단위를 단일 프로그램에 병합합니다. ODR 규칙 위반은 일반적으로 링커 오류로 표시됩니다. 링커 오류는 동일한 이름이 둘 이상의 번역 단위에 정의되어 있을 때 발생합니다.

일반적으로 여러 파일에 변수를 표시할 수 있는 가장 좋은 방법은 헤더 파일에서 선언하는 것입니다. 그런 다음 선언이 `#include` 필요한 모든 `.cpp` 파일에 지시문을 추가합니다. 헤더 내용 주위에 포함 가드를 추가하면 헤더가 선언하는 이름이 각 번역 단위에 대해 한번만 선언되도록 합니다. 하나의 구현 파일에서만 이름을 정의합니다.

C++20 에서는 모듈이 헤더 파일에 대한 향상된 대안으로 도입되었습니다.

경우에 따라 파일에서 전역 변수 또는 클래스 .cpp 를 선언해야 할 수 있습니다. 이러한 경우 컴파일러와 링커에 이름에 어떤 종류의 링크가 있는지 알려주는 방법이 필요합니다. 링크 유형은 개체 이름이 한 파일에서만 표시되는지 아니면 모든 파일에만 표시되는지 여부를 지정합니다. 링크의 개념은 전역 이름에만 적용됩니다. 링크의 개념은 범위 내에서 선언된 이름에는 적용되지 않습니다. 범위는 함수 또는 클래스 정의와 같은 묶은 중괄호 집합에 의해 지정됩니다.

외부 링크 및 내부 링크

free 함수는 전역 또는 네임스페이스 범위에서 정의된 함수입니다. 비 const 전역 변수 및 무료 함수에는 기본적으로 외부 링크가 있으며 프로그램의 모든 번역 단위에서 볼 수 있습니다. 다른 전역 개체는 해당 이름을 가질 수 없습니다. 내부 링크가 있거나 링크가 없는 기호는 선언된 변환 단위 내에서만 표시됩니다. 이름에 내부 링크가 있는 경우 동일한 이름이 다른 번역 단위에 있을 수 있습니다. 클래스 정의 또는 함수 본문 내에서 선언된 변수에는 연결이 없습니다.

전역 이름은 명시적으로 선언하여 내부 링크가 있도록 강제 적용할 수 있습니다 static. 이 키워드(keyword) 선언된 것과 동일한 번역 단위로 표시 유형을 제한합니다. 이 컨텍스트 static 에서는 지역 변수에 적용할 때와 다른 것을 의미합니다.

다음 개체에는 기본적으로 내부 링크가 있습니다.

- const 개체입니다.
- constexpr 개체입니다.
- typedef 개체입니다.
- static 네임스페이스 범위의 개체

개체 외부 링크를 제공하려면 개체를 const 선언 extern 하고 값을 할당합니다.

C++

```
extern const int value = 42;
```

자세한 내용은 [extern](#)를 참조하세요.

참고 항목

[기본 개념](#)

main 함수와 명령줄 인수

아티클 • 2024. 09. 26.

모든 C++ 프로그램에는 `main` 함수가 있어야 합니다. `main` 함수 없이 C++ 프로그램을 컴파일하려고 하면 컴파일러에서 오류가 발생합니다. (동적 링크 라이브러리 및 static 라이브러리에는 `main` 함수가 없습니다.) 이 `main` 함수는 소스 코드 실행을 시작하지만 프로그램이 `main` 함수에 들어가기 전에 명시적 이니셜라이저가 없는 모든 static 클래스 멤버는 0으로 설정됩니다. Microsoft C++에서는 전역 static 개체도 `main`에 입력되기 전에 초기화됩니다. 다른 모든 C++ 함수에는 적용되지 않는 여러 제한이 `main` 함수에는 적용됩니다. `main` 함수:

- 오버로드할 수 없습니다([함수 오버로드](#) 참조).
- `inline`으로 선언할 수 없습니다.
- `static`으로 선언할 수 없습니다.
- 주소를 사용할 수 없습니다.
- 프로그램에서 호출할 수 없습니다.

main 함수 시그니처

`main` 함수는 언어에 기본 제공되므로 선언이 없습니다. 이 경우 `main`에 대한 선언 구문은 다음과 같습니다.

```
C++

int main();
int main(int argc, char *argv[]);
```

반환 값이 `main`에 지정되지 않은 경우, 컴파일러는 반환 값 0을 제공합니다.

표준 명령줄 인수

인수의 편리한 명령줄 구문 분석을 허용하기 위한 `main`에 대한 인수입니다. `argc` 및 `argv`의 형식은 언어에서 정의됩니다. 이름 `argc` 및 `argv`는 전통적이지만 원하는 대로 이름을 지정할 수 있습니다.

인수 정의는 다음과 같습니다.

`argc`

`argv` 뒤에 오는 인수 개수를 포함하는 정수입니다. `argc` 매개 변수는 항상 1보다 크거나 같습니다.

`argv`

프로그램의 사용자가 입력한 명령줄 인수를 나타내는 `null`로 끝나는 문자열의 배열입니다. 규칙에 따라 `argv[0]`은 프로그램을 호출하는 데 사용하는 명령입니다. `argv[1]`은 첫 번째 명령줄 인수입니다. 명령줄의 마지막 인수는 `argv[argc - 1]`이고, `argv[argc]`는 항상 `NUL`입니다.

명령줄 처리를 표시하지 않는 방법에 대한 자세한 내용은 [C++ 명령줄 처리 사용자 지정](#)을 참조하세요.

① 참고

규칙에 따라 `argv[0]`은 프로그램의 파일 이름입니다. 그러나 Windows에서는 [CreateProcess](#)를 사용하여 프로세스를 생성할 수 있습니다. 첫 번째 인수와 두 번째 인수(`lpApplicationName` 및 `lpCommandLine`)를 모두 사용하는 경우, `argv[0]`은 실행 가능한 이름이 아닐 수 있습니다. 실행 파일 이름 및 정규화된 경로를 검색하는 데 [GetModuleFileName](#)를 사용할 수 있습니다.

Microsoft 관련 확장

다음 섹션에서는 Microsoft 관련 동작에 대해 설명합니다.

`wmain` 함수 및 `_tmain` 매크로

유니코드 와이드 문자를 사용하도록 소스 코드를 디자인하는 경우, `main`의 와이드 문자 버전인 Microsoft 관련 `wmain` 진입점을 사용할 수 있습니다. 다음은 `wmain`의 유효 선언 구문입니다.

C++

```
int wmain();
int wmain(int argc, wchar_t *argv[]);
```

`tchar.h`에 정의된 전처리기 매크로인 Microsoft 전용 `_tmain`을 사용할 수도 있습니다.

`_tmain`은 `_UNICODE`이 정의되지 않는 한 `main`로 해결됩니다. `_UNICODE`가 정의된 경우에는 `_tmain`이 `wmain`으로 확인됩니다. `_t`로 시작하는 `_tmain` 매크로 및 다른 매크로는 코드는 좁은 문자 집합과 와이드 문자 집합 모두에 대해 별도의 버전을 빌드해야 하는 코드에 유용합니다. 자세한 내용은 [일반 텍스트 매크로 사용](#)을 참조하세요.

`main`에서 `void` 반환

Microsoft 확장으로, `main` 및 `wmain` 함수는 `void` 반환(반환 값 없음)으로 선언할 수 있습니다. 이 확장은 다른 컴파일러에서도 사용할 수 있지만 사용하지 않는 것이 좋습니다. `main`이 값을 반환하지 않는 경우 대칭으로 사용할 수 있습니다.

`void`을 반환하는 것으로 `main` 또는 `wmain`을 선언하는 경우, `return`문을 사용하여 부모 프로세스 또는 운영 체제에 `exit` 코드를 반환할 수 없습니다. `main` 또는 `wmain`이 `void`로 선언될 때 `exit` 코드를 반환하려면 `exit` 함수를 사용해야 합니다.

envp 명령줄 인수

`main` 또는 `wmain` 서명은 환경 변수에 액세스하기 위한 선택적 Microsoft 전용 확장을 허용합니다. 이 확장은 Windows 및 UNIX 시스템의 다른 컴파일러에서도 일반적입니다. 이름 `envp`은 기존 이름이지만 원하는 대로 환경 매개 변수의 이름을 지정할 수 있습니다. 환경 매개 변수를 포함하는 인수 목록에 대한 유효 선언은 다음과 같습니다.

C++

```
int main(int argc, char* argv[], char* envp[]);
int wmain(int argc, wchar_t* argv[], wchar_t* envp[]);
```

`envp`

선택적 `envp` 파라미터는 사용자 환경에서 설정된 변수를 나타내는 문자열의 배열입니다. 이 배열은 NULL 항목으로 종료됩니다. `char`에 대한 포인터의 배열(`char *envp[]`) 또는 `char`에 대한 포인터의 포인터(`char **envp`)로 매개 변수를 선언할 수 있습니다. 프로그램에서 `main` 대신 `wmain`을 사용하는 경우 `char` 대신 `wchar_t` 데이터 형식을 사용합니다.

`main` 및 `wmain`으로 전달되는 환경 블록은 현재 환경의 "고정된" 복사본입니다. 이후에 `putenv` 또는 `_wputenv`에 호출을 통해 환경을 변경할 경우, 현재 환경(`getenv` 또는 `_wgetenv` 및 `_environ` 또는 `_wenviron` 변수에서 반환)이 변경되지만 `envp`가 가리키는 블록은 변경되지 않습니다. 환경 처리를 표시하지 않는 방법에 대한 자세한 내용은 [C++ 명령줄 처리 사용자 지정](#)을 참조하세요. `envp` 인수는 C89 표준과 호환되지만 C++ 표준과는 호환되지 않습니다.

main에 대한 예제 인수

다음 예제에서는, `argc`, `argv` 및 `envp` 인수를 `main`에 사용하는 방법을 보여줍니다.

C++

```
// argument_definitions.cpp
// compile with: /EHsc
```

```

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

using namespace std;
int main( int argc, char *argv[], char *envp[] )
{
    bool numberLines = false;      // Default is no line numbers.

    // If /n is passed to the .exe, display numbered listing
    // of environment variables.
    if ( (argc == 2) && _strcmp( argv[1], "/n" ) == 0 )
        numberLines = true;

    // Walk through list of strings until a NULL is encountered.
    for ( int i = 0; envp[i] != NULL; ++i )
    {
        if ( numberLines )
            cout << i << ":"; // Prefix with numbers if /n specified
        cout << envp[i] << "\n";
    }
}

```

C++ 명령줄 인수 구문 분석

Microsoft C/C++ 코드에서 사용하는 명령줄 구문 분석 규칙은 Microsoft 전용입니다. 런타임 시작 코드에서는 운영 체제 명령줄에 지정된 인수를 해석할 때 이러한 규칙을 사용합니다.

- 인수를 공백이나 탭으로 구분합니다.
- 첫 번째 인수(`argv[0]`)는 특별하게 처리됩니다. 이 인수는 프로그램 이름을 나타냅니다. 유효한 경로 이름이어야 하므로 큰따옴표(`"`)로 묶은 파트가 허용됩니다. 큰따옴표는 `argv[0]` 출력에 포함되지 않습니다. 큰따옴표로 묶은 파트는 공백 또는 탭 문자를 인수의 끝으로 해석하는 것을 방지합니다. 이 목록의 이후 규칙은 적용되지 않습니다.
- 큰 따옴표로 묶은 문자열은 공백 문자를 포함할 수 있는 하나의 인수로 해석됩니다. 따옴표로 묶은 문자열은 인수에 포함될 수 있습니다. 캐럿(`^`)은 이스케이프 문자나 구분 기호로 인식되지 않습니다. 따옴표 붙은 문자열 내에서 큰따옴표 쌍은 이스케이프된 단일 큰따옴표로 해석됩니다. 닫는 큰따옴표 전에 명령줄이 끝나면 지금 까지 읽은 모든 문자가 마지막 인수로 출력됩니다.
- 백슬래시 다음의 큰따옴표(`\"`)는 리터럴 큰따옴표(`"`)로 해석됩니다.
- 백슬래시는 큰따옴표 바로 앞에 있지 않으면 리터럴로 해석됩니다.

- 짹수 개의 백슬래시 다음에 큰따옴표가 오는 경우, 백슬래시 쌍(\\)마다 하나의 백슬래시(\\)가 argv 배열에 배치되고, 큰따옴표(")는 문자열 구분 기호로 해석됩니다.
- 홀수 개의 백슬래시 다음에 큰따옴표가 오는 경우, 백슬래시 쌍(\\)마다 argv 배열에 하나의 백슬래시(\\)가 배치됩니다. 큰따옴표는 나머지 백슬래시에 의해 이스케이프 시퀀스로 해석되어 리터럴 큰따옴표(")가 argv에 배치됩니다.

명령줄 인수 구문 분석의 예

다음 프로그램은 명령줄 인수를 전달하는 방법을 보여 줍니다.

```
C++
```

```
// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main( int argc,           // Number of strings in array argv
          char *argv[],     // Array of command-line argument strings
          char *envp[] )    // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
        cout << "  argv[" << count << "]  "
              << argv[count] << "\n";
}
```

명령줄 구문 분석 결과

다음 표에서는 앞의 목록의 규칙을 보여 줌으로써 예제 입력 및 예상 출력을 보여줍니다.

[\[+\] 테이블 확장](#)

명령줄 입력	argv[1]	argv[2]	argv[3]
"abc" d e	abc	d	e
a\\b d"e f"g h	a\\b	de fg	h
a\\\\\"b c d	a\\\"b	c	d

명령줄 입력	argv[1]	argv[2]	argv[3]
a\\\"b c" d e	a\\b c	d	e
a"b"" c d	ab" c d		

와일드카드 식

Microsoft 컴파일러는 필요에 따라 와일드카드 문자, 물음표(?) 및 별표(*)를 사용하여 명령줄에서 파일 이름 및 경로 인수를 지정할 수 있습니다.

명령줄 인수는 런타임 시작 코드의 내부 루틴에 의해 처리되며, 기본적으로 와일드카드는 argv 문자열 배열에서 별도 문자열로 확장하지 않습니다. **/link** 컴파일러 옵션 또는 **LINK** 명령줄에 *setargv.obj* 파일(*wmain*에 대한 *wsetargv.obj* 파일)을 포함하여 와일드 카드 확장을 사용하도록 설정할 수 있습니다.

런타임 시작 링커 옵션에 대한 자세한 내용은 [링크 옵션](#)을 참조하세요.

C++ 명령줄 처리 사용자 지정

프로그램에서 명령줄 인수를 사용하지 않는 경우 명령줄 처리를 수행하는 루틴을 억제하여 약간의 공간을 절약할 수 있습니다. 사용을 억제하려면 **/link** 컴파일러 옵션이나 **LINK** 명령줄에 *noarg.obj* 파일(*main* 및 *wmain* 둘 다에 대해)을 포함합니다.

마찬가지로 *envp* 인수를 통해 환경 테이블에 액세스하지 않는 경우 내부 환경 처리 루틴 또한 억제할 수 있습니다. 사용을 억제하려면 **/link** 컴파일러 옵션이나 **LINK** 명령줄에 *noenv.obj* 파일(*main* 및 *wmain* 둘 다에 대해)을 포함합니다.

프로그램에서 C 런타임 라이브러리의 루틴 중 *spawn* 또는 *exec* 제품군을 호출할 수 있습니다. 이 경우 부모 프로세스에서 자식 프로세스로 환경을 전달하는 데 사용되므로 환경 처리 루틴을 억제하면 안 됩니다.

참고 항목

[기본 개념](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

C++ 프로그램 종료

아티클 • 2024. 08. 02.

C++에서는 다음과 같은 방법으로 프로그램을 종료할 수 있습니다.

- 함수를 호출합니다 `exit`.
- 함수를 호출합니다 `abort`.
- 에서 `return main` 문을 실행합니다.

exit 함수

`stdlib.h`>에 <선언된 함수는 `exit` C++ 프로그램을 종료합니다. 인수 `exit`로 제공된 값은 프로그램의 반환 코드 또는 종료 코드로 운영 체제에 반환됩니다. 규칙에 따라 반환 코드 0은 프로그램이 성공적으로 완료되었음을 의미합니다. 상수 `EXIT_FAILURE` 와 `EXIT_SUCCESS` `stdlib.h`>에 <정의된 상수도 사용하여 프로그램의 성공 또는 실패를 나타낼 수 있습니다.

abort 함수

표준 include file `<stdlib.h>`에도 선언된 함수는 `abort` C++ 프로그램을 종료합니다. 차이점은 `exit` `abort` C++ 런타임 종료 처리가 수행될 수 있다는 `exit` 점입니다(전역 개체 소멸자가 호출됨). `abort` 는 프로그램을 즉시 종료합니다. 이 함수는 `abort` 초기화된 전역 정적 개체에 대한 일반 소멸 프로세스를 무시합니다. 또한 이 함수는 `atexit` 함수를 사용하여 지정된 모든 특수 처리를 건너뜁니다.

Microsoft 관련: Windows 호환성을 위해 Microsoft 구현 `abort`을 통해 특정 상황에서 DLL 종료 코드를 실행할 수 있습니다. 자세한 내용은 [abort](#)를 참조하세요.

atexit 함수

함수를 `atexit` 사용하여 프로그램이 종료되기 전에 실행되는 작업을 지정합니다. 종료 처리 함수를 실행하기 전에 호출 `atexit` 이 제거되기 전에 초기화된 전역 정적 개체가 없습니다.

return statement in main

이 `return` 문을 사용하면 .에서 `main` 반환 값을 지정할 수 있습니다. 첫 번째 문은 `return main` 다른 `return` 문처럼 작동합니다. 모든 자동 변수가 제거됩니다. 그런 다음 반환

`main` 값을 매개 변수로 호출합니다 `exit`. 다음 예제를 참조하세요.

C++

```
// return_statement.cpp
#include <stdlib.h>
struct S
{
    int value;
};
int main()
{
    S s{ 3 };

    exit( 3 );
    // or
    return 3;
}
```

`exit` 이전 예제의 and `return` 문은 비슷한 동작을 갖습니다. 둘 다 프로그램을 종료하고 3 값을 운영 체제에 반환합니다. 차이점은 `exit` 문이 자동 변수 `s` `return` 를 삭제하지 않는다는 점입니다.

일반적으로 C++에서는 값을 반환하지 않는 반환 형식 `void` 이 있는 함수가 필요합니다. 함수는 `main` 예외입니다. 문 없이 `return` 종료할 수 있습니다. 이 경우 호출 프로세스에 구현별 값을 반환합니다. (기본적으로 MSVC는 0을 반환합니다.)

스레드 및 정적 개체의 소멸

직접 호출 `exit` 하거나 문 `main` 이후에 `return` 호출될 때 현재 스레드와 연결된 스레드 개체가 제거됩니다. 다음으로 정적 개체는 초기화의 역순으로 제거됩니다(지정된 함수(있는 경우)에 대한 `atexit` 호출 후). 다음 예제에서는 이러한 초기화 및 정리가 작동하는 방법을 보여 줍니다.

예시

다음 예제에서는 정적 개체 `sd1` 를 `sd2` 만들고 입력 `main`하기 전에 초기화합니다. 이 프로그램이 문을 사용하여 `return` 종료되면 먼저 `sd2` 제거된 다음 `sd1`. `ShowData` 클래스에 대한 소멸자가 이 정적 개체와 연결된 파일을 닫습니다.

C++

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
```

```

public:
    // Constructor opens a file.
    ShowData( const char *szDev ) {
        errno_t err;
        err = fopen_s(&OutputDev, szDev, "w" );
    }

    // Destructor closes the file.
    ~ShowData() { fclose( OutputDev ); }

    // Disp function shows a string on the output device.
    void Disp( char *szData ) {
        fputs( szData, OutputDev );
    }
private:
    FILE *OutputDev;
};

// Define a static object of type ShowData. The output device
// selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

// Define another static object of type ShowData. The output
// is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}

```

이 코드를 작성하는 또 다른 방법은 블록 범위가 있는 개체를 `ShowData` 선언하는 것입니다. 이 개체는 범위를 벗어날 때 암시적으로 삭제됩니다.

C++

```

int main() {
    ShowData sd1( "CON" ), sd2( "hello.dat" );

    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}

```

참고 항목

[main 함수 및 명령줄 인수](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Lvalue 및 Rvalue (C++)

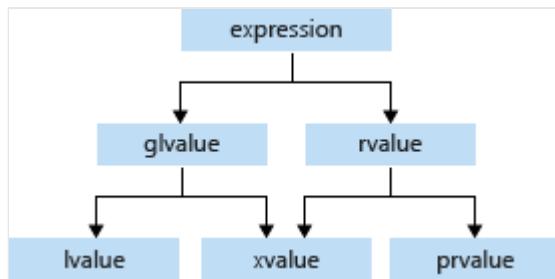
아티클 • 2024. 01. 10.

모든 C++ 식에는 형식이 있으며 값 범주에 속합니다. 값 범주는 식 평가 중에 임시 개체를 만들고 복사하고 이동할 때 컴파일러가 따라야 하는 규칙의 기초입니다.

C++17 표준은 다음과 같이 식 값 범주를 정의합니다.

- *glvalue*는 평가에서 개체, 비트 필드 또는 함수의 ID를 결정하는 식입니다.
- *prvalue*는 계산에서 개체 또는 비트 필드를 초기화하거나 표시되는 컨텍스트에 정된 대로 연산자의 피연산자 값을 계산하는 식입니다.
- *xvalue*는 리소스를 재사용할 수 있는 개체 또는 비트 필드를 나타내는 *glvalue*입니다(일반적으로 수명이 거의 다 되었으므로). 예: rvalue 참조(8.3.2)와 관련된 특정 종류의 식은 반환 형식이 rvalue 참조이거나 rvalue 참조 형식에 대한 캐스트인 함수에 대한 호출과 같이 *xvalue*를 생성합니다.
- *lvalue*는 *xvalue*가 아닌 *glvalue*입니다.
- *rvalue*는 *prvalue* 또는 *xvalue*입니다.

다음 다이어그램은 범주 간의 관계를 보여 줍니다.



*lvalue*에는 프로그램에서 액세스할 수 있는 주소가 있습니다. *lvalue* 식의 예로는 변수, 배열 요소, *lvalue* 참조, 비트 필드, 공용 구조체 및 클래스 멤버를 반환하는 함수 호출을 비롯한 `const` 변수 이름이 있습니다.

prvalue 식에는 프로그램에서 액세스할 수 있는 주소가 없습니다. *prvalue* 식의 예로는 리터럴, 비참조 형식을 반환하는 함수 호출 및 식 평가 중에 생성되지만 컴파일러에서만 액세스할 수 있는 임시 개체가 있습니다.

xvalue 식에는 프로그램에서 더 이상 액세스할 수 없지만 식에 대한 액세스를 제공하는 *rvalue* 참조를 초기화하는 데 사용할 수 있는 주소가 있습니다. 예를 들어 *rvalue* 참조를 반환하는 함수 호출과 배열 또는 개체가 *rvalue* 참조인 멤버 식에 대한 배열 아래 첨자, 멤버 및 포인터가 있습니다.

예시

다음 예제에서는 lvalue 및 rvalue에 대한 여러 가지 올바른 사용과 올바르지 않은 사용의 예를 보여 줍니다.

C++

```
// lvalues_and_rvalues2.cpp
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a
    // prvalue.
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106). `j * 4` is a prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}
```

① 참고

이 항목의 예제는 연산자가 오버로드되지 않을 때 올바른 사용과 올바르지 않은 사용의 예를 보여 줍니다. 연산자를 오버로드하여 `j * 4`와 같은 식을 lvalue로 만들 수 있습니다.

`lvalue` 및 `rvalue`라는 용어는 개체 참조를 참조할 때 자주 사용됩니다. 참조 [에 대한 자세한 내용은 Lvalue 참조 선언자: 및 Rvalue 참조 선언자: &&>](#)를 참조하세요.

참고 항목

[기본 개념](#)

[Lvalue 참조 선언자: &](#)

[Rvalue 참조 선언자: &&>](#)

임시 개체

아티클 • 2024. 07. 12.

임시 개체는 임시 값을 저장하기 위해 컴파일러에서 만든 명명되지 않은 개체입니다.

설명

경우에 따라 컴파일러가 임시 개체를 만들어야 합니다. 다음과 같은 이유로 이 임시 개체를 만들 수 있습니다.

- 초기화 중인 참조의 기본 형식과 다른 형식의 이니셜라이저로 `const` 참조를 초기화 하려는 경우
- 사용자 정의 형식(UDT)을 반환하는 함수의 반환 값을 저장하려는 경우 이러한 임시 개체는 프로그램이 반환 값을 개체에 복사하지 않는 경우에만 만들어집니다. 예시:

```
C++  
  
UDT Func1();    // Declare a function that returns a user-defined  
                  // type.  
  
...  
  
Func1();        // Call Func1, but discard return value.  
                  // A temporary object is created to store the return  
                  // value.
```

반환 값이 다른 개체에 복사되지 않으므로 임시 개체가 만들어집니다. 임시 개체가 만들어지는 보다 일반적인 경우는 오버로드된 연산자 함수를 호출해야 하는 식을 계산하는 경우입니다. 이러한 오버로드된 연산자 함수는 다른 개체에 자주 복사되지 않는 사용자 정의 형식을 반환합니다.

`ComplexResult = Complex1 + Complex2 + Complex3` 식을 참조하십시오. `Complex1 + Complex2` 식이 계산되고 결과가 임시 개체에 저장됩니다. 다음으로, 임시 `+ Complex3` 식이 계산되고 결과가 `ComplexResult`에 복사됩니다(대입 연산자가 오버로드되지 않는다고 가정함).

- 캐스팅 결과를 사용자 정의 형식에 저장하려는 경우. 지정된 형식의 개체를 사용자 정의 형식으로 명시적으로 변환하는 경우 새 개체는 임시 개체로 구성됩니다.

임시 개체는 생성 지점과 소멸 지점으로 정의된 수명을 갖습니다. 둘 이상의 임시 개체를 만드는 식은 결국 생성 순서를 반대로 삭제합니다.

임시 삭제가 발생하는 경우 사용 방법에 따라 달라집니다.

- `const` 참조를 초기화하는 데 사용되는 임시:

이니셜라이저가 초기화되는 참조와 동일한 형식의 `I`-값이 아니면 기본 개체 형식의 임시 형식이 만들어집니다. 초기화 식에 의해 초기화됩니다. 이 임시 개체는 이 개체가 바인딩된 참조 개체가 제거된 후 즉시 제거됩니다. 이 소멸은 임시를 만든 식 이후에 발생할 수 있으므로 **수명 확장**이라고도 합니다.

- 식 평가의 효과로 만든 임시:

첫 번째 범주에 맞지 않고 식 계산의 효과로 생성되는 모든 임시는 식 문(즉, 세미콜론)의 끝이나 `for`, `if`, `while`, `do` 및 `switch` 문에 대한 제어 식의 끝에서 제거됩니다.

참고 항목

Herb Sutter의 블로그, [레퍼런스](#), 단순하게 ↴

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) ↴ | Microsoft Q&A에서 도움말 보기

맞춤

아티클 • 2024. 11. 21.

C++의 하위 수준 기능 중 하나는 특정 하드웨어 아키텍처를 최대한 활용하기 위해 메모리 내 개체의 정확한 맞춤을 지정하는 기능입니다. 기본적으로 컴파일러는 클래스 및 구조체 멤버의 크기 값 `bool` `char` 과 1 바이트 경계, `short` 2 바이트 경계, `long float` `int` 4 바이트 경계 및 `long double` `long long double` 8 바이트 경계에 정렬합니다.

대부분의 시나리오에서는 기본 맞춤이 이미 최적이므로 맞춤에 신경 쓰지 않아도 됩니다. 그러나 경우에 따라 데이터 구조에 대한 사용자 지정 맞춤을 지정하여 상당한 성능 향상 또는 메모리 절감을 달성할 수 있습니다. Visual Studio 2015 이전에는 Microsoft 관련 키워드를 `_alignof` `_declspec(align)` 사용하고 기본값보다 큰 맞춤을 지정할 수 있습니다. Visual Studio 2015부터 최대 코드 이식성을 위해 C++11 표준 키워드를 `alignof` `alignas` 사용해야 합니다. 새 키워드는 Microsoft 관련 확장과 동일한 방식으로 작동합니다. 이러한 확장에 대한 설명서는 새 키워드에도 적용됩니다. 자세한 내용은 연산자, [alignas 지정자](#) 및 [맞춤을 참조 alignof 하세요](#). C++ 표준은 대상 플랫폼의 컴파일러 기본값보다 작은 경계 맞춤에 대한 압축 동작을 지정하지 않으므로 이 경우 Microsoft `#pragma pack` 를 사용해야 합니다.

사용자 지정 맞춤을 사용하여 [데이터 구조의 메모리 할당에 aligned_storage 클래스](#) 를 사용합니다. [aligned_union 클래스](#)는 비휘발성 생성자 또는 소멸자를 사용하는 공용 구조체에 대한 맞춤을 지정하기 위한 것입니다.

맞춤 및 메모리 주소

맞춤은 숫자 주소 모듈로 2의 거듭제곱으로 표현된 메모리 주소의 속성입니다. 예를 들어 주소 0x0001103F 모듈로 4는 3입니다. 해당 주소는 $4n+3$ 에 맞춰진 것으로, 여기서 4는 선택한 2의 힘을 나타냅니다. 주소의 맞춤은 선택한 2의 힘에 따라 달라집니다. 동일한 주소의 모듈로 8은 7입니다. 맞춤이 $Xn+0$ 인 경우 주소가 X에 맞춰집니다.

CPU는 메모리에 저장된 데이터에 대해 작동하는 명령을 실행합니다. 데이터는 메모리의 주소로 식별됩니다. 단일 데이터의 크기도 있습니다. 주소가 크기에 맞춰지면 자연스럽게 정렬된 데이터를 호출합니다. 그렇지 않으면 잘못 정렬되었다고 합니다. 예를 들어 8바이트 부동 소수점 데이터는 식별에 사용되는 주소에 8바이트 맞춤이 있는 경우 자연스럽게 정렬됩니다.

데이터 맞춤의 컴파일러 처리

컴파일러는 데이터 정렬을 방해하는 방식으로 데이터 할당을 시도합니다.

단순 데이터 형식의 경우 컴파일러에서 데이터 형식 크기(바이트)의 배수인 주소를 할당합니다. 예를 들어 컴파일러는 주소의 하위 2 비트를 0으로 설정하여 4의 배수인 형식 `long` 의 변수에 주소를 할당합니다.

또한 컴파일러는 구조체의 각 요소를 자연스럽게 맞추는 방식으로 구조체를 패딩합니다. 다음 코드 예제의 구조를 `struct x_` 고려합니다.

C++

```
struct x_
{
    char a;          // 1 byte
    int b;           // 4 bytes
    short c;         // 2 bytes
    char d;          // 1 byte
} bar[3];
```

컴파일러는 자연스럽게 맞춰지도록 이 구조를 채웁니다.

다음 코드 예제에서는 컴파일러가 패딩된 구조를 메모리에 배치하는 방법을 보여 줍니다.

C++

```
// Shows the actual memory layout
struct x_
{
    char a;          // 1 byte
    char _pad0[3];   // padding to put 'b' on 4-byte boundary
    int b;           // 4 bytes
    short c;         // 2 bytes
    char d;          // 1 byte
    char _pad1[1];   // padding to make sizeof(x_) multiple of 4
} bar[3];
```

두 선언 모두 12바이트로 반환 `sizeof(struct x_)` 됩니다.

두 번째 선언에는 다음 두 개의 패딩 요소가 포함되어 있습니다.

1. `char _pad0[3]` 4 바이트 경계에 멤버를 정렬 `int b` 합니다.
2. `char _pad1[1]` 구조체의 `struct _x bar[3];` 배열 요소를 4 바이트 경계에 맞추려면

안쪽 여백은 자연 액세스를 허용하는 방식으로 요소를 `bar[3]` 정렬합니다.

다음 코드 예제에서는 배열 레이아웃을 `bar[3]` 보여줍니다.

Output

```

adr offset   element
-----
0x0000  char a;           // bar[0]
0x0001  char pad0[3];
0x0004  int b;
0x0008  short c;
0x000a  char d;
0x000b  char _pad1[1];

0x000c  char a;           // bar[1]
0x000d  char _pad0[3];
0x0010  int b;
0x0014  short c;
0x0016  char d;
0x0017  char _pad1[1];

0x0018  char a;           // bar[2]
0x0019  char _pad0[3];
0x001c  int b;
0x0020  short c;
0x0022  char d;
0x0023  char _pad1[1];

```

alignof 및 alignas

`alignas` 지정자는 변수 및 사용자 정의 형식의 사용자 지정 맞춤을 지정하는 이식 가능한 C++ 표준 방법입니다. `alignof` 마찬가지로 연산자는 지정된 형식 또는 변수의 맞춤을 가져오는 표준 이식 가능한 방법입니다.

예시

클래스, 구조체 또는 공용 구조체 또는 개별 멤버에서 사용할 `alignas` 수 있습니다. 여러 `alignas` 지정자가 발견되면 컴파일러는 값이 가장 큰 지정자를 선택합니다.

C++

```

// alignas_alignof.cpp
// compile with: cl /EHsc alignas_alignof.cpp
#include <iostream>

struct alignas(16) Bar
{
    int i;          // 4 bytes
    int n;          // 4 bytes
    alignas(4) char arr[3];
    short s;        // 2 bytes
};

```

```
int main()
{
    std::cout << alignof(Bar) << std::endl; // output: 16
}
```

참고 항목

[데이터 구조 맞춤 ↴](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공 ↴](#) | [Microsoft Q&A에서 도움말 보기](#)

Trivial, 표준 레이아웃, POD 및 리터럴 형식

아티클 • 2023. 04. 03.

용어 **레이아웃**은 클래스, 구조체 또는 공용 형식의 개체 멤버가 메모리에 정렬되는 방식을 가리킵니다. 경우에 따라 레이아웃은 언어 사양에 따라 잘 정의됩니다. 그러나 클래스 또는 구조체에 가상 기본 클래스, 가상 함수, 액세스 제어가 다른 멤버와 같은 특정 C++ 언어 기능이 포함되어 있는 경우, 컴파일러는 레이아웃을 자유롭게 선택할 수 있습니다. 해당 레이아웃은 수행되는 최적화에 따라 달라질 수 있으며, 대부분의 경우 개체가 인접한 메모리의 영역을 차지하지 못할 수도 있습니다. 예를 들어 클래스에 가상 함수가 있는 경우 해당 클래스의 모든 인스턴스는 단일 가상 함수 테이블을 공유할 수 있습니다. 이러한 형식은 매우 유용하지만 제한 사항도 있습니다. 레이아웃이 정의되지 않았기 때문에 C와 같은 다른 언어로 작성된 프로그램에 전달할 수 없으며, 인접하지 않을 수도 있기 때문에 `memcpy`와 같은 빠른 하위 수준 함수로 안정적으로 복사하거나 네트워크를 통해 직렬화할 수 없습니다.

C++ 프로그램 및 메타프로그램뿐만 아니라 컴파일러가 특정 메모리 레이아웃에 종속된 작업에 대해 지정된 유형의 적합성에 대한 추론을 위해 C++14는 간단한 클래스와 구조체인 *trivial*, 표준 레이아웃 및 *POD* 또는 Plain Old Data의 세 가지 범주를 도입했습니다. 표준 라이브러리에는 지정된 유형이 지정된 범주에 속하는지 여부를 결정하는 함수 템플릿 `is_trivial<T>`, `is_standard_layout<T>` 및 `is_pod<T>`가 있습니다.

Trivial 형식

C++의 클래스 또는 구조체에 컴파일러 제공 또는 명시적으로 특수 멤버 함수를 기본값으로 지정하는 경우 이는 *trivial* 형식입니다. 인접한 메모리 영역을 차지합니다. 다른 액세스 지정자가 있는 멤버를 가질 수 있습니다. C++에서 컴파일러는 이 상황에서 멤버를 주문하는 방법을 자유롭게 선택할 수 있습니다. 따라서 이러한 개체를 `memcpy`할 수는 있지만 C 프로그램에서 안정적으로 사용할 수 없습니다. *trivial* 형식 T는 `char` 또는 부호 없는 `char` 배열에 복사하고, 안전하게 T 변수로 다시 복사할 수 있습니다. 맞춤 요구 사항으로 인해 형식 멤버 간에 패딩 바이트가 있을 수 있습니다.

Trivial 형식에는 *trivial* 기본 생성자, *trivial* 복사 생성자, *trivial* 복사 할당 연산자 및 *trivial* 소멸자가 있습니다. 각각의 경우에 *trivial*은 생성자/연산자/소멸자가 사용자 제공이 아니라 다음과 같은 클래스에 속한다는 것을 의미합니다.

- 가상 함수 또는 기본 클래스가 없음
- 해당 비 *trivial* 생성자/연산자/소멸자가 있는 기본 클래스가 없음

- 해당 비 trivial 생성자/연산자/소멸자가 있는 클래스 유형의 데이터 멤버 없음

다음 예제는 trivial 형식을 보여줍니다. Trivial2에서 `Trivial2(int a, int b)` 생성자가 있으면 기본 생성자를 제공해야 합니다. trivial로 형식을 한정하려면 해당 생성자를 명시적으로 기본 설정해야 합니다.

C++

```
struct Trivial
{
    int i;
private:
    int j;
};

struct Trivial2
{
    int i;
    Trivial2(int a, int b) : i(a), j(b) {}
    Trivial2() = default;
private:
    int j; // Different access control
};
```

표준 레이아웃 형식

클래스 또는 구조체에 C 언어에서 찾을 수 없는 가상 함수와 같은 특정 C++ 언어 기능이 없고 모든 멤버가 동일한 액세스 제어를 갖고 있으면 표준 레이아웃 형식입니다. 이는 memcpy가 가능하고 레이아웃이 충분히 정의되어 C 프로그램에서 사용할 수 있습니다. 표준 레이아웃 형식은 사용자 정의 특수 멤버 함수를 사용할 수도 있습니다. 또한 표준 레이아웃 형식에는 다음과 같은 특성이 있습니다.

- 가상 함수 또는 기본 클래스 없음
- 모든 비정적 데이터 멤버는 동일한 액세스 제어를 가지고 있습니다.
- 클래스 형식의 모든 비정적 멤버는 표준 레이아웃입니다.
- 모든 기본 클래스는 표준 레이아웃입니다.
- 첫 번째 비정적 데이터 멤버와 동일한 유형의 기본 클래스가 없습니다.
- 다음 조건 중 하나를 충족합니다.
 - 가장 많이 파생된 클래스에 비정적 데이터 멤버가 없고, 비정적 데이터 멤버가 있는 기본 클래스가 하나 이하인 경우, 또는

- 비정적 데이터 멤버가 있는 기본 클래스가 없음

다음 코드는 표준 레이아웃 형식의 한 가지 예를 보여줍니다.

C++

```
struct SL
{
    // All members have same access:
    int i;
    int j;
    SL(int a, int b) : i(a), j(b) {} // User-defined constructor OK
};
```

마지막 두 가지 요구 사항은 코드로 더 잘 설명될 수 있습니다. 다음 예제에서 기본 사항이 표준 레이아웃임에도 불구하고 `Derived`는 표준 레이아웃이 아닙니다. 가장 많이 파생된 클래스와 `Base`는 모두 비정적 데이터 멤버가 있기 때문입니다.

C++

```
struct Base
{
    int i;
    int j;
};

// std::is_standard_layout<Derived> == false!
struct Derived : public Base
{
    int x;
    int y;
};
```

이 예제에서 `Base`에 비정적 데이터 멤버가 없기 때문에 `Derived`는 표준 레이아웃입니다.

C++

```
struct Base
{
    void Foo() {}
};

// std::is_standard_layout<Derived> == true
struct Derived : public Base
{
    int x;
    int y;
};
```

`Base`에 데이터 멤버가 있고 `Derived`에 멤버 함수만 있는 경우에도 파생된 것이 표준 레이아웃이 될 수 있습니다.

POD 형식

클래스 또는 구조체가 사소하고 표준 레이아웃인 경우 POD(Plain Old Data) 형식입니다. 따라서 POD 형식의 메모리 레이아웃은 연속적이며 각 멤버는 이전에 선언된 멤버보다 더 높은 주소를 가지므로 바이트 복사본 및 이진 I/O를 위한 바이트가 이러한 유형에 대해 수행될 수 있습니다. `int`와 같은 스칼라 형식도 POD 형식입니다. 클래스인 POD 형식은 POD 형식만 비정적 데이터 멤버로 가질 수 있습니다.

예

다음 예제에서는 trivial, 표준 레이아웃 및 POD 형식 간의 차이점을 보여줍니다.

C++

```
#include <type_traits>
#include <iostream>

using namespace std;

struct B
{
protected:
    virtual void Foo() {}
};

// Neither trivial nor standard-layout
struct A : B
{
    int a;
    int b;
    void Foo() override {} // Virtual function
};

// Trivial but not standard-layout
struct C
{
    int a;
private:
    int b;    // Different access control
};

// Standard-layout but not trivial
struct D
{
    int a;
```

```

int b;
D() {} //User-defined constructor
};

struct POD
{
    int a;
    int b;
};

int main()
{
    cout << boolalpha;
    cout << "A is trivial is " << is_trivial<A>() << endl; // false
    cout << "A is standard-layout is " << is_standard_layout<A>() << endl;
// false

    cout << "C is trivial is " << is_trivial<C>() << endl; // true
    cout << "C is standard-layout is " << is_standard_layout<C>() << endl;
// false

    cout << "D is trivial is " << is_trivial<D>() << endl; // false
    cout << "D is standard-layout is " << is_standard_layout<D>() << endl; //
true

    cout << "POD is trivial is " << is_trivial<POD>() << endl; // true
    cout << "POD is standard-layout is " << is_standard_layout<POD>() <<
endl; // true

    return 0;
}

```

리터럴 형식

리터럴 형식은 컴파일 타임에 해당 레이아웃이 결정될 수 있는 형식입니다. 다음은 리터럴 형식입니다.

- void
- 스칼라 형식
- references
- void, 스칼라 형식 또는 참조의 배열
- trivial 소멸자 및 이동 또는 복사 생성자가 아닌 constexpr 생성자를 하나 이상 포함하는 클래스입니다. 또한 해당 비정적 데이터 멤버 및 기본 클래스가 모두 리터럴 형식이고 volatile이 아니어야 합니다.

추가 정보

기본 개념

값 형식으로서의 C++ 클래스

아티클 • 2023. 10. 12.

C++ 클래스는 기본적으로 값 형식입니다. 개체 지향 프로그래밍을 지원하기 위해 다형 동작을 지원하는 참조 형식으로 지정할 수 있습니다. 값 형식은 메모리 및 레이아웃 컨트롤의 관점에서 볼 수 있는 반면 참조 형식은 다형성을 위해 기본 클래스 및 가상 함수에 관한 것입니다. 기본적으로 값 형식은 복사 가능하므로 항상 복사 생성자와 복사 할당 연산자가 있습니다. 참조 형식의 경우 클래스를 복사할 수 없게 만들고(복사 생성자 및 복사 할당 연산자를 사용하지 않도록 설정) 의도한 다형성을 지원하는 가상 소멸자를 사용합니다. 또한 값 형식은 콘텐츠에 관한 것이며, 복사할 때는 항상 별도로 수정할 수 있는 두 개의 독립적인 값을 제공합니다. 참조 형식은 ID에 관한 것입니다. 개체의 종류는 무엇인가요? 이러한 이유로 "참조 형식"을 "다형 형식"이라고도 합니다.

참조와 유사한 형식(기본 클래스, 가상 함수)을 원하는 경우 다음 코드의 클래스에 `MyRefType` 표시된 것처럼 복사를 명시적으로 사용하지 않도록 설정해야 합니다.

```
C++  
  
// cl /EHsc /nologo /W4  
  
class MyRefType {  
private:  
    MyRefType & operator=(const MyRefType &);  
    MyRefType(const MyRefType &);  
public:  
    MyRefType () {}  
};  
  
int main()  
{  
    MyRefType Data1, Data2;  
    // ...  
    Data1 = Data2;  
}
```

위의 코드를 컴파일하면 다음 오류가 발생합니다.

Output

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private  
member declared in class 'MyRefType'  
        meow.cpp(5) : see declaration of 'MyRefType::operator ='  
        meow.cpp(3) : see declaration of 'MyRefType'
```

값 형식 및 이동 효율성

새 복사 최적화로 인해 복사 할당 오버헤드가 방지됩니다. 예를 들어 문자열의 벡터 중간에 문자열을 삽입하는 경우 벡터 자체의 증가가 발생하더라도 복사 재할당 오버헤드는 없으며 이동만 있습니다. 이러한 최적화는 다른 작업(예: 두 개의 거대한 개체에 대한 추가 작업 수행)에도 적용됩니다. 이러한 값 작업 최적화를 사용하도록 설정하려면 어떻게 해야 할까요? 컴파일러를 사용하면 복사 생성자가 컴파일러에서 자동으로 생성될 수 있는 것처럼 암시적으로 사용할 수 있습니다. 그러나 클래스 정의에서 할당을 선언하여 할당을 이동하고 생성자를 이동하려면 클래스가 "옵트인"해야 합니다. Move는 적절한 멤버 함수 선언에서 이중 앤퍼샌드(&&) rvalue 참조를 사용하고 이동 생성자 및 이동 할당 메서드를 정의합니다. 또한 원본 개체에서 "배짱을 도용"하는 올바른 코드를 삽입해야 합니다.

이동 작업을 사용하도록 설정해야 하는지 여부를 어떻게 결정합니까? 복사 생성을 사용하도록 설정해야 한다는 것을 이미 알고 있는 경우 특히 딥 카피보다 저렴할 경우 이동 생성을 사용하도록 설정하려고 할 수 있습니다. 그러나 이동 지원이 필요하다고 해서 반드시 복사 작업을 사용하도록 설정하려는 것은 아닙니다. 이 후자의 경우를 "이동 전용 형식"이라고 합니다. 표준 라이브러리에 이미 있는 예는 .입니다 `unique_ptr`. 참고로 이전 `auto_ptr` 버전은 사용되지 않으며 이전 버전의 C++에서 이동 의미 체계 지원이 부족하여 정확하게 대체 `unique_ptr` 되었습니다.

이동 의미 체계를 사용하여 값으로 반환하거나 중간에 삽입할 수 있습니다. 이동은 복사 최적화입니다. 해결 방법으로 힙 할당이 필요하지 않습니다. 다음 의사 코드를 살펴보겠습니다.

C++

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData();    // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" );    // efficient, no deep copy-
shuffle
v.insert( begin(v)+v.size()/2, "Andrei" );   // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix& , const HugeMatrix& );
```

```

HugeMatrix operator+(const HugeMatrix& ,           HugeMatrix&&);
HugeMatrix operator+(      HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+(      HugeMatrix&&,           HugeMatrix&&);
//...
hm5 = hm1+hm2+hm3+hm4+hm5; // efficient, no extra copies

```

적절한 값 형식에 대해 이동 사용

딥 카피보다 이동이 저렴할 수 있는 가치와 유사한 클래스의 경우 효율성을 위해 이동 생성 및 이동 할당을 사용하도록 설정합니다. 다음 의사 코드를 살펴보겠습니다.

C++

```

#include <memory>
#include <stdexcept>
using namespace std;
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other ) // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other ) // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() { // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient
resources!");
    }
};

```

복사 생성/할당을 사용하도록 설정하는 경우 딥 카피보다 저렴할 수 있는 경우 이동 생성/할당을 사용하도록 설정합니다.

일부 *비값* 형식은 리소스를 복제할 수 없고 소유권만 이전하는 경우와 같이 이동 전용입니다. 예: `unique_ptr`.

참고 항목

[C++ 형식 시스템](#)

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

형식 변환 및 형식 안전성

아티클 • 2023. 10. 12.

이 문서에서는 공용 형식 변환 문제를 식별하고 C++ 코드에서 이 문제를 방지하는 방법에 대해 설명합니다.

C++ 프로그램을 작성하는 경우 형식 안전이 확인되어야 합니다. 즉 모든 변수, 함수 인수 및 함수 반환 값에서 적합한 종류의 데이터를 저장하고 다른 형식의 값을 포함하는 작업이 "합리적"이며 데이터 손실, 비트 패턴의 잘못된 해석 또는 메모리 손상이 발생하지 않음을 의미합니다. 명시적 또는 암시적으로 값을 한 형식에서 다른 형식으로 변환하지 않는 프로그램은 정의에 따라 형식이 안전합니다. 그러나 안전하지 않은 변환인 경우에도 형식 변환이 필요한 경우가 있습니다. 예를 들어 부동 소수점 연산의 결과를 형식 `int` 변수에 저장해야하거나 값을 사용하는 함수 `signed int`에 `unsigned int` 전달해야 할 수 있습니다. 두 예제 모두 값의 데이터 손실 또는 다시 해석을 일으킬 수 있으므로 안전하지 않은 변환을 보여 줍니다.

컴파일러에서 안전하지 않은 변환을 발견하는 경우 오류 또는 경고가 발생합니다. 오류는 컴파일을 중지합니다. 경고는 컴파일을 계속하지만 코드에서 가능한 오류를 나타냅니다. 그러나 프로그램이 경고 없이 컴파일되는 경우에도 암시적 형식 변환이 발생하고 잘못된 결과가 발생하는 코드가 포함될 수 있습니다. 코드에서 형식 오류는 명시적 변환 또는 캐스트에서 나타날 수도 있습니다.

암시적 형식 변환

식에 다른 기본 제공 형식의 피연산자가 포함되어 있고 명시적 캐스트가 없는 경우 컴파일러는 기본 제공 표준 변환을 사용하여 피연산자 중 하나를 변환하여 형식이 일치하도록 합니다. 컴파일러는 하나가 성공할 때까지 잘 정의된 시퀀스의 변환을 시도합니다. 선택한 변환이 승격인 경우 컴파일러는 경고를 발생하지 않습니다. 변환이 축소인 경우 컴파일러에서 가능한 데이터 손실에 대한 경고가 발생됩니다. 실제 데이터 손실이 발생하는지 여부는 관련된 실제 값에 따라 다르지만 이 경고를 오류로 처리하는 것이 좋습니다. 사용자 정의 형식이 포함된 경우 컴파일러에서는 클래스 정의에서 지정한 변환을 사용하고 합니다. 허용되는 변환을 찾을 수 없는 경우 컴파일러는 오류를 발생시키고 프로그램을 컴파일하지 않습니다. 표준 변환을 제어하는 규칙에 대한 자세한 내용은 표준 변환을 참조 [하세요](#). 사용자 정의 변환에 대한 자세한 내용은 사용자 정의 변환 ([C++/CLI](#))을 참조하세요.

확대 변환(승격)

확대 변환에서 작은 변수의 값은 데이터의 손실 없이 큰 변수에 할당됩니다. 확대 변환은 항상 안전하므로 컴파일러는 자동으로 수행하며 경고를 발생하지 않습니다. 다음 변환은

확대 변환입니다.

보낸 사람	수행할 작업
다음을 제외한 <code>long long</code> 모든 <code>signed</code> 또는 <code>unsigned</code> 정수 계열 형식 <code>_int64</code>	<code>double</code>
<code>bool</code> 또는 <code>char</code>	다른 기본 제공 형식
<code>short</code> 또는 <code>wchar_t</code>	<code>int</code> , <code>long</code> , <code>long long</code> <code>long</code>
<code>int</code> , <code>long</code>	<code>long long</code>
<code>float</code>	<code>double</code>

축소 변환(강제 변환)

컴파일러에서 암시적으로 축소 변환을 수행하지만 잠재적인 데이터 손실에 대한 경고가 나타납니다. 이러한 경고를 매우 심각하게 고려하십시오. 큰 변수의 값은 항상 작은 변수에 맞으므로 데이터 손실이 발생하지 않을 것이라고 가정하는 경우 컴파일러에서 더 이상 경고가 발생하지 않도록 명시적 캐스트를 추가합니다. 변환이 안전하다고 확신할 수 없는 경우 코드에 일종의 런타임 검사 추가하여 가능한 데이터 손실을 처리하여 프로그램이 잘못된 결과를 생성하지 않도록 합니다.

부동 소수점 형식에서 정수 계열 형식으로의 모든 변환은 부동 소수점 값의 소수 부분이 삭제 및 손실되므로 축소 변환입니다.

다음 코드 예제에서는 일부 암시적 축소 변환 및 이에 대해 컴파일러에서 발생하는 경고를 보여 줍니다.

C++

```
int i = INT_MAX + 1; //warning C4307:'+' : integral constant overflow
wchar_t wch = 'A'; //OK
char c = wch; // warning C4244:'initializing':conversion from 'wchar_t'
               // to 'char', possible loss of data
unsigned char c2 = 0xffffe; //warning C4305:'initializing': truncation from
                           // 'int' to 'unsigned char'
int j = 1.9f; // warning C4244:'initializing':conversion from 'float' to
               // 'int', possible loss of data
int k = 7.7; // warning C4244:'initializing':conversion from 'double' to
               // 'int', possible loss of data
```

부호 있는 - 부호 없는 변환

부호 있는 정수 계열 형식 및 해당 부호 없는 정수 계열 형식은 항상 크기가 동일하지만 값 변형에 대한 비트 패턴 해석 방법은 서로 다릅니다. 다음 코드 예제에서는 부호 있는 값과 부호 없는 값으로 동일한 비트 패턴을 해석할 때 발생하는 사항을 보여 줍니다. `num` 및 `num2`에 저장된 비트 패턴은 이전 그림에 표시되는 내용에서 변경되지 않습니다.

C++

```
using namespace std;
unsigned short num = numeric_limits<unsigned short>::max(); // #include
<limits>
short num2 = num;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"

// Go the other way.
num2 = -1;
num = num2;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"
```

값은 두 가지 지시 사항으로 다시 해석됩니다. 프로그램에서 값의 부호가 예상되는 결과와 반대인 흔수 결과를 생성하는 경우 부호 있는 정수 계열과 부호 없는 정수 계열 간의 암시적 변환을 찾습니다. 다음 예제에서는 식(0 - 1)의 결과가 저장될 때 암시적으로 변환 `int unsigned int` 됩니다 `num`. 이를 통해 비트 패턴이 다시 해석됩니다.

C++

```
unsigned int u3 = 0 - 1;
cout << u3 << endl; // prints 4294967295
```

컴파일러는 부호 있는 정수 형식과 서명되지 않은 정수 형식 간의 암시적 변환에 대해 경고하지 않습니다. 따라서 서명되지 않은 변환을 모두 피하는 것이 좋습니다. 이를 방지할 수 없는 경우 런타임 검사 추가하여 변환되는 값이 0보다 크거나 같은지, 서명된 형식의 최대값보다 작거나 같은지 감지합니다. 이 범위의 값은 다시 해석되지 않고 부호 있음에서 부호 없음으로 또는 부호 없음에서 부호 있음으로 전송됩니다.

포인터 변환

많은 식에서 C 스타일 배열은 배열의 첫 번째 요소에 대한 포인터로 암시적으로 변환되고 상수 변환이 자동으로 발생할 수 있습니다. 이 방법이 편리하지만 잠재적으로 오류가 발생할 수도 있습니다. 예를 들어 다음과 같이 잘못 디자인된 코드 예제는 무의미해 보이지만 컴파일되어 'p'의 결과를 생성합니다. 먼저 "Help" 문자열 상수 리터럴은 배열의 첫 번째 요소를 가리키는 `char*`로 변환되고 해당 포인터는 이제 마지막 요소 'p'를 가리키도록 세 가지 요소로 증가됩니다.

C++

```
char* s = "Help" + 3;
```

명시적 변환(캐스트)

캐스트 작업을 사용하여 컴파일러가 한 형식의 값을 다른 형식으로 변환하도록 지시할 수 있습니다. 컴파일러는 두 형식이 완전히 관련이 없는 경우에 오류를 발생시키지만, 다른 경우에는 작업이 형식이 안전하지 않더라도 오류가 발생하지 않습니다. 한 형식에서 다른 형식으로의 변환은 프로그램 오류의 잠재적 원인이므로 캐스트를 가능하면 사용하지 마십시오. 그러나 캐스트는 필요한 경우가 있으며 모든 캐스트가 똑같이 위험한 것은 아닙니다. 캐스트의 한 가지 효과적인 사용은 코드가 축소 변환을 수행하고 변환으로 인해 프로그램이 잘못된 결과를 생성하지 않는다는 것을 알고 있는 경우입니다. 실제로 사용자가 해야 할 사항을 알고 있으므로 이에 대해 경고하지 않도록 컴파일러에 지시합니다. 다른 사용은 파생 클래스에 대한 포인터에서 기본 클래스에 대한 포인터로 캐스팅하는 것입니다. 또 다른 용도는 변수의 단점을 캐스팅하여 비 const 인수가 필요한 함수에 전달하는 것입니다. 대부분의 이러한 캐스팅 작업은 일부 위험이 내포됩니다.

C 스타일의 프로그래밍에서 동일한 C 스타일 캐스트 연산자는 모든 종류의 캐스트에 사용됩니다.

C++

```
(int) x; // old-style cast, old-style syntax  
int(x); // old-style cast, functional syntax
```

C 스타일 캐스트 연산자는 호출 연산자 ()와 동일하므로 코드에서 눈에 띄지 않으며 간과하기 쉽습니다. 둘 다 한눈에 인식하거나 검색하기 어렵기 때문에 좋지 않으며, `const` 및 `reinterpret_cast`의 `static` 조합을 호출할 수 있을 만큼 이질적이기 때문입니다. 이전 스타일의 캐스트에서 실제로 수행하는 사항을 알아내는 것은 어려울 수 있으며 오류가 발생할 수 있습니다. 이러한 모든 이유로 캐스팅이 필요할 때 다음 C++ 캐스트 연산자 중 하나를 사용하는 것이 좋습니다. 해당 연산자는 경우에 따라 훨씬 더 형식 안정적이며 더욱 명시적인 프로그래밍 의도를 표시합니다.

- `static_cast` - 컴파일 시간에만 검사 캐스트의 경우 `static_cast` 는 컴파일러가 완전히 호환되지 않는 형식 간에 캐스팅을 시도하는 것을 감지하면 오류를 반환합니다. 포인터에서 기본 클래스에 대한 포인터와 파생 클래스에 대한 포인터 간을 캐스팅하는 데 사용할 수도 있지만 이러한 변환이 런타임에서 안전한지 여부를 컴파일에서 항상 확인할 수 없습니다.

C++

```

double d = 1.58947;
int i = d; // warning C4244 possible loss of data
int j = static_cast<int>(d); // No warning.
string s = static_cast<string>(d); // Error C2440:cannot convert from
// double to std::string

// No error but not necessarily safe.
Base* b = new Base();
Derived* d2 = static_cast<Derived*>(b);

```

자세한 내용은 [static_cast](#)를 참조하세요.

- **dynamic_cast** - 포인터에서 파생된 포인터로의 안전한 런타임 검사 캐스트를 위한 것입니다. A **dynamic_cast** 는 다운캐스트보다 **static_cast** 안전하지만 런타임 검사 약간의 오버헤드가 발생합니다.

C++

```

Base* b = new Base();

// Run-time check to determine whether b is actually a Derived*
Derived* d3 = dynamic_cast<Derived*>(b);

// If b was originally a Derived*, then d3 is a valid pointer.
if(d3)
{
    // Safe to call Derived method.
    cout << d3->DoSomethingMore() << endl;
}
else
{
    // Run-time check failed.
    cout << "d3 is null" << endl;
}

//Output: d3 is null;

```

자세한 내용은 [dynamic_cast](#)를 참조하세요.

- **const_cast** - 변수의 -ness를 **const** 캐스팅하거나 변수가 아닌 변수를 **const**>로 변환하는 경우 **const** 이 연산자를 사용하여 -ness를 캐스팅 **const** 하는 것은 실수로 캐스트를 수행할 가능성이 적다는 점을 **const_cast** 제외하고 C 스타일 캐스트를 사용하는 것과 마찬가지로 오류가 발생하기 쉽습니다. 예를 들어 변수를 매개 변수가 **const** 아닌 **const** 함수에 전달 **const** 하기 위해 변수의 -ness를 캐스팅해야 하는 경우가 있습니다. 다음 예제에 이 작업을 수행하는 방법이 나와 있습니다.

C++

```
void Func(double& d) { ... }
void ConstCast()
{
    const double pi = 3.14;
    Func(const_cast<double&>(pi)); //No error.
}
```

자세한 내용은 [const_cast](#)를 참조하세요.

- [reinterpret_cast](#)- 포인터 형식과 같은 관련 없는 형식 간 캐스트의 [int](#) 경우 .

① 참고

이 캐스트 연산자는 다른 연산자만큼 자주 사용되지 않으며 다른 컴파일러에 이식 가능하도록 보장되지 않습니다.

다음 예제에서는 .와 차이점 [reinterpret_cast](#) 을 보여 줍니다 [static_cast](#).

C++

```
const char* str = "hello";
int i = static_cast<int>(str); //error C2440: 'static_cast' : cannot
                                // convert from 'const char *' to 'int'
int j = (int)str; // C-style cast. Did the programmer really intend
                  // to do this?
int k = reinterpret_cast<int>(str); // Programming intent is clear.
                                    // However, it is not 64-bit safe.
```

자세한 내용은 연산자(Operator)를 참조하세요[reinterpret_cast](#).

참고 항목

[C++ 형식 시스템](#)

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

표준 변환

아티클 • 2023. 10. 12.

C++ 언어에서는 기본 형식 간의 변환을 정의합니다. 또한 포인터, 참조 및 멤버 포인터 파생 형식에 대한 변환도 정의합니다. 이러한 변환을 표준 변환이라고 합니다.

이 단원에서는 다음과 같은 표준 변환에 대해 설명합니다.

- 정수 계열 확장
- 정수 계열 변환
- 부동 변환
- 부동 및 정수 계열 변환
- 산술 변환
- 포인터 변환
- 참조 변환
- 멤버 포인터 변환

① 참고

사용자 정의 형식은 고유한 변환을 지정할 수 없습니다. 사용자 정의 형식의 변환은 생성자 및 변환에서 **다릅니다**.

다음 코드에서는 변환을 발생시킵니다(이 예제의 경우 정수 계열 확장).

C++

```
long long_num1, long_num2;
int int_num;

// int_num promoted to type long prior to assignment.
long_num1 = int_num;

// int_num promoted to type long prior to multiplication.
long_num2 = int_num * long_num2;
```

참조 형식을 생성하는 경우에만 변환의 결과가 l-value입니다. 예를 들어 참조로 선언된 `operator int&()` 사용자 정의 변환은 참조를 반환하고 l-value입니다. 그러나 선언된 변환은 `operator int()` 개체를 반환하며 l-value가 아닙니다.

정수 계열 확장

정수 계열 형식의 개체를 다른 더 넓은 정수 계열 형식, 즉 더 큰 값 집합을 나타낼 수 있는 형식으로 변환할 수 있습니다. 이 확대 변환 유형을 정수 승격이라고 합니다. 정수 계열 승격을 사용하면 다른 정수 계열 형식을 사용할 수 있는 모든 식에서 다음 형식을 사용할 수 있습니다.

- 개체, 리터럴 및 형식 `char` 의 상수 및 `short int`
- 열거형 형식
- `int` 비트 필드
- 열거자

C++ 승격은 승격 후 값이 승격 전 값과 같도록 보장되므로 "값 유지"입니다. 값 유지 승격에서는 원래 형식의 전체 범위를 나타낼 수 있는 경우 `int` 더 짧은 정수 계열 형식(예: 비트 필드 또는 형식 `char` 개체)의 개체가 형식 `int` 으로 승격됩니다. 전체 값 범위를 나타낼 수 없으면 `int` 개체가 형식 `unsigned int`으로 승격됩니다. 이 전략은 표준 C에서 사용하는 전략과 동일하지만 값 유지 변환은 개체의 "서명성"을 유지하지 않습니다.

값을 보존하는 확장과 부호 유부를 보존하는 확장은 보통 동일한 결과를 생성합니다. 그러나 승격된 개체가 다음과 같이 표시되면 다른 결과를 생성할 수 있습니다.

- , „ <= % > /= < %= 또는 피 / 연산자 >=

이 연산자는 부호를 사용하여 결과를 확인합니다. 값 보존 및 부호 유지 승격은 이러한 피연산자에 적용될 때 다른 결과를 생성합니다.

- 의 왼쪽 피연산자 >> >>=

이러한 연산자는 시프트 작업에서 서명된 수량과 서명되지 않은 수량을 다르게 처리합니다. 부호 있는 수량의 경우 오른쪽 시프트 작업은 부호 있는 비트 위치를 비운 비트 위치로 전파하고, 비어 있는 비트 위치는 부호 없는 수량으로 채워지지 않습니다.

- 인수 일치를 위해 피연산자 형식의 부호에 따라 달라지는 오버로드된 함수 또는 오버로드된 연산자의 피연산자에 대한 인수입니다. 오버로드된 연산자를 정의하는 방법에 대한 자세한 내용은 오버로드된 연산자를 참조 [하세요](#).

정수 계열 변환

정수 **변환**은 정수 계열 형식 간의 변환입니다. 정수 계열 형식은 `char`, `short` (또는 `short int`) `int`, `long` 및 `long long`. 이러한 형식은 정규화 `signed` 될 수 있으며 `unsigned`,에 대한 `unsigned int` 약식으로 `unsigned` 사용할 수 있습니다.

서명되지 않은 상태로 서명됨

부호가 있는 정수 계열 형식의 개체를 부호가 없는 해당 형식으로 변환할 수 있습니다. 이러한 변환이 발생하면 실제 비트 패턴이 변경되지 않습니다. 그러나 데이터 해석이 변경되었습니다. 다음과 같은 코드를 생각해 볼 수 있습니다.

C++

```
#include <iostream>

using namespace std;
int main()
{
    short i = -3;
    unsigned short u;

    cout << (u = i) << "\n";
}
// Output: 65533
```

앞의 예제에서, `i`는 `signed short` 정의되고 음수로 초기화됩니다. 이 식 `(u = i)` `i` 은 할당 전에 `unsigned short`로 변환됩니다 `u`.

서명되지 않음

부호 없는 정수 계열 형식의 개체를 부호 있는 해당 형식으로 변환할 수 있습니다. 그러나 부호 없는 값이 부호 있는 형식의 표시 가능한 범위를 벗어나면 다음 예제와 같이 결과에 올바른 값이 없습니다.

C++

```
#include <iostream>

using namespace std;
int main()
{
    short i;
    unsigned short u = 65533;

    cout << (i = u) << "\n";
}
//Output: -3
```

앞의 예제 `u unsigned short` 에서는 식을 (`i = u`) 평가하기 위해 서명된 수량으로 변환해야 하는 정수 개체입니다. 해당 값을 제대로 나타낼 `signed short` 수 없으므로 표시된 대로 데이터가 잘못 해석됩니다.

부동 소수점 변환

부동 형식의 개체를 보다 정확한 부동 형식으로 안전하게 변환할 수 있습니다. 즉, 변환으로 인해 유의성이 손실되지 않습니다. 예를 들어 대/중 `double long double` 변환 `float double` 은 안전하며 값은 변경되지 않습니다.

부동 형식의 개체가 해당 형식으로 나타낼 수 있는 범위에 있는 경우 덜 정확한 형식으로 변환할 수도 있습니다. (참조) [부동 형식의 범위에 대한 부동 제한](#) 입니다.) 원래 값을 정확하게 나타낼 수 없는 경우 다음으로 더 높거나 다음으로 낮은 표현 가능한 값으로 변환할 수 있습니다. 이러한 값이 없으면 결과가 정의되지 않습니다. 다음 예제를 참조하세요.

C++

```
cout << (float)1E300 << endl;
```

형식 `float` 으로 나타낼 수 있는 최대값은 1E300보다 훨씬 작은 3.402823466E38입니다. 따라서 숫자는 무한대로 변환되고 결과는 "inf"입니다.

정수 계열 및 부동 소수점 형식 간의 변환

특정 식은 부동 형식의 개체가 정수 계열 형식으로 변환되도록 하거나 그 반대로 변환되도록 할 수 있습니다. 정수 계열 형식의 개체가 부동 형식으로 변환되고 원래 값을 정확하게 나타낼 수 없는 경우 결과는 다음으로 높거나 다음으로 낮은 표현 가능한 값입니다.

부동 형식의 개체가 정수 형식으로 변환되면 소수 부분이 잘리거나 0으로 반올림됩니다. 1.3과 같은 숫자는 1로 변환되고 -1.3은 -1로 변환됩니다. 잘린 값이 가장 높은 표현 가능한 값보다 높거나 가장 낮은 표현 가능한 값보다 낮으면 결과는 정의되지 않습니다.

산술 변환

많은 이진 연산자([이진 연산자를 사용하여 식에서 설명됨](#))는 피연산자를 변환하고 결과를 동일한 방식으로 생성합니다. 이러한 연산자가 유발하는 변환을 일반적인 산술 변환이라고 합니다. 다음 표와 같이 네이티브 형식이 다른 피연산자의 산술 변환이 수행됩니다. `typedef` 형식은 해당 기본 네이티브 형식에 따라 동작합니다.

형식 변환 조건

조건 충족	전환
피연산자 중 하나가 형식 <code>long double</code> 입니다.	다른 피연산자는 형식 <code>long double</code> 으로 변환됩니다.
이전 조건이 충족되지 않고 피연산자 중 하나가 형식 <code>double</code> 입니다.	다른 피연산자는 형식 <code>double</code> 으로 변환됩니다.
이전 조건이 충족되지 않고 피연산자 중 하나가 형식 <code>float</code> 입니다.	다른 피연산자는 형식 <code>float</code> 으로 변환됩니다.
이전 조건이 충족되지 않았습니다(부동 형식인 피연산자가 없음).	<p>피연산자는 다음과 같이 정수 승격을 가져옵니다.</p> <ul style="list-style-type: none"> - 피연산자 중 하나가 형식 <code>unsigned long</code>인 경우 다른 피연산자는 형식 <code>unsigned long</code>으로 변환됩니다. - 이전 조건이 충족되지 않고 피연산자 중 하나가 형식 <code>long</code>이고 다른 형식 <code>unsigned int</code>인 경우 두 피연산자는 모두 형식 <code>unsigned long</code>으로 변환됩니다. - 위의 두 조건이 충족되지 않고 피연산자 중 하나가 형식 <code>long</code>인 경우 다른 피연산자는 형식 <code>long</code>으로 변환됩니다. - 위의 세 조건이 충족되지 않고 피연산자 중 하나가 형식 <code>unsigned int</code>인 경우 다른 피연산자는 형식 <code>unsigned int</code>으로 변환됩니다. - 위의 조건이 충족되지 않으면 두 피연산자는 모두 형식 <code>int</code>으로 변환됩니다.

다음 코드에서는 표에 설명된 변환 규칙을 보여 줍니다.

C++

```
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main() {
    // iVal converted to unsigned long
    // result of multiplication converted to double
    dVal = iVal * ulVal;

    // ulVal converted to float
    // result of addition converted to double
    dVal = ulVal + fVal;
}
```

위의 예제에서 첫 번째 문은 `iVal`과 `ulVal`이라는 두 정수 계열 형식을 곱한 것입니다. 충족되는 조건은 피연산자 모두 부동 형식이 아니고 하나의 피연산자는 형식 `unsigned int`

입니다. 따라서 다른 피연산자는 `iVal` 형식 `unsigned int` 으로 변환됩니다. 그런 다음, 결과가 `.에 할당됩니다 dVal`. 여기서 충족되는 조건은 하나의 피연산자 형식 `double` 이므로 `unsigned int` 곱하기 결과가 형식 `double` 으로 변환됩니다.

앞의 예제의 두 번째 문은 정 `float` 수 계열 형식과 정수 계열 형식 `fVal` 을 더한 것을 보여 하며, `u1Val` `u1Val` 변수가 형식 `float` (테이블의 세 번째 조건)으로 변환됩니다. 추가의 결과는 형식 `double` (테이블의 두 번째 조건)으로 변환되고 `에 할당됩니다 dVal`.

포인터 변환

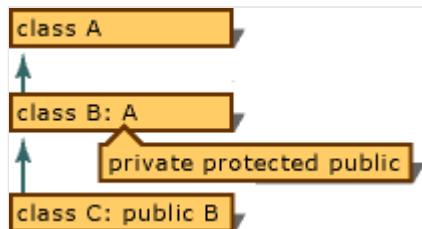
초기화, 할당, 비교 및 기타 식 실행 중에 포인터가 변환될 수 있습니다.

클래스 포인터

클래스에 대한 포인터가 기본 클래스에 대한 포인터로 변환될 수 있는 두 가지 경우가 있습니다.

첫 번째 경우는 지정된 기본 클래스에 액세스할 수 있고 변환이 명확한 경우입니다. 모호한 기본 클래스 참조에 대한 자세한 내용은 여러 기본 클래스를 참조 [하세요](#).

기본 클래스에 액세스할 수 있는지 여부는 파생에서 사용되는 상속의 종류에 따라 달라집니다. 다음 그림에 설명된 상속을 고려합니다.



기본 클래스 접근성을 보여 주는 상속 그래프

다음 표에서는 그림에 나와 있는 상황에 대한 기본 클래스 액세스 가능성을 보여 줍니다.

함수 형식	파생 상품	B*에서 A*로의 변환이
외부(클래스 범위가 아닌) 함수	프라이빗	아니요
	Protected	아니요
	공용	예
B 멤버 함수(B 범위에 있음)	프라이빗	예

함수 형식	파생 상품	B*에서 A*로의 변환이
	B* A* 법적?	
	Protected	예
	공용	예
C 멤버 함수(C 범위에 있음)	프라이빗	아니요
	Protected	예
	공용	예

클래스에 대한 포인터가 기본 클래스에 대한 포인터로 변환될 수 있는 두 번째 경우는 명시적 형식 변환을 사용하는 경우입니다. 명시적 형식 변환에 대한 자세한 내용은 명시적 형식 변환 연산자를 참조 [하세요](#).

이러한 변환의 결과는 하위 개체 `A`에 대한 포인터로, 기본 클래스에서 완전히 설명하는 개체 부분입니다.

다음 코드에서는 두 클래스 `A` 및 `B`를 정의합니다. `B`는 `A`에서 파생됩니다. 상속에 대한 자세한 내용은 [파생 클래스입니다](#).) 그런 다음 개체를 `bObject` 가리키는 두 개의 포인터 (`pA` 및 `pB`)와 형식 `B`의 개체를 정의합니다.

C++

```
// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

int main()
{
    B bObject;
    A *pA = &bObject;
    B *pB = &bObject;

    pA->AMemberFunc(); // OK in class A
    pB->AMemberFunc(); // OK: inherited from class A
```

```
pA->BMemberFunc(); // Error: not in class A  
}
```

포인터 `pA` 는 "형식 `A *` 의 개체에 대한 포인터"를 의미하는 것으로 해석될 수 있는 형식 `A`입니다. (예: `BComponent` 및)의 `bObject` 멤버는 형식 `B`에 고유하므로 을 통해 `pA BMemberFunc` 액세스할 수 없습니다. `pA` 포인터는 `A` 클래스에서 정의된 개체의 특성(멤버 함수 및 데이터)에만 액세스할 수 있습니다.

함수 포인터

형식이 해당 포인터를 보유할 수 있을 만큼 큰 경우 함수에 대한 포인터를 형식 `void * void *` 으로 변환할 수 있습니다.

void 포인터

형식 `void`에 대한 포인터는 다른 형식에 대한 포인터로 변환할 수 있지만 C와 달리 명시적 형식 캐스트를 사용하는 경우에만 가능합니다. 모든 형식에 대한 포인터를 암시적으로 형식 `void`에 대한 포인터로 변환할 수 있습니다. 형식의 불완전한 개체에 대한 포인터는 (암시적으로) 및 뒤로(명시적으로) 포인터 `void`로 변환할 수 있습니다. 이러한 변환 결과는 원래 포인터 값과 같습니다. 개체가 선언된 경우 불완전한 것으로 간주되지만 크기 또는 기본 클래스를 결정하는 데 사용할 수 있는 정보가 부족합니다.

형식 `void *`의 포인터로 암시적으로 변환되지 않거나 `const volatile` 암시적으로 변환할 수 있는 개체에 대한 포인터입니다.

const 및 volatile 포인터

C++는 형식 또는 형식에서 `const` 형식이 아닌 형식으로의 표준 변환을 제공하지 않습니다 `volatile const`, `const volatile` 하지만 모든 종류의 변환은 명시적 형식 캐스트를 사용하여 지정할 수 있습니다(안전하지 않은 변환 포함).

① 참고

정적 멤버에 대한 포인터를 제외한 멤버에 대한 C++ 포인터는 일반 포인터와 다르며 표준 변환이 동일하지 않습니다. 정적 멤버에 대한 포인터는 일반 포인터이며 일반 포인터와 동일한 변환이 적용됩니다.

null 포인터 변환

0으로 계산되는 정수 상수 식 또는 포인터 형식으로 캐스팅된 식은 null 포인터라는 포인터로 변환됩니다. 이 포인터는 항상 포인터와 같지 않은 개체 또는 함수를 비교합니다. 예외는 동일한 오프셋을 가질 수 있고 여전히 다른 개체를 가리킬 수 있는 기반 개체에 대한 포인터입니다.

C++11에서는 C 스타일 null 포인터에 nullptr 형식을 선호해야 합니다.

포인터 식 변환

배열 형식의 식을 동일한 형식의 포인터로 변환할 수 있습니다. 변환 결과는 첫 번째 배열 요소의 포인터입니다. 다음 예제에서는 이러한 변환에 대해 설명합니다.

C++

```
char szPath[_MAX_PATH]; // Array of type char.  
char *pszPath = szPath; // Equals &szPath[0].
```

특정 형식을 반환하는 함수를 도출하는 식은 해당 형식을 반환하는 함수의 포인터로 변환됩니다. 단, 다음의 경우는 제외됩니다.

- 식은 address-of 연산자(&)에 대한 피연산자로 사용됩니다.
- 해당 식이 함수 호출 연산자의 피연산자로 사용됩니다.

참조 변환

다음과 같은 경우 클래스에 대한 참조를 기본 클래스에 대한 참조로 변환할 수 있습니다.

- 지정된 기본 클래스에 액세스할 수 있습니다.
- 해당 변환은 명확합니다. (모호한 기본 클래스 참조에 대한 자세한 내용은 다음을 참조하세요 [여러 기본 클래스](#).)

변환의 결과는 기본 클래스를 나타내는 하위 개체에 대한 포인터입니다.

멤버 포인터

할당, 초기화, 비교 및 기타 식 실행 중에 클래스 멤버에 대한 포인터가 변환될 수 있습니다. 이 단원에서는 다음과 같은 멤버 포인터 변환에 대해 설명합니다.

기본 클래스 멤버 포인터

기본 클래스의 멤버에 대한 포인터는 다음 조건이 충족될 때 파생되는 클래스의 멤버에 대한 포인터로 변환할 수 있습니다.

- 포인터에서 파생 클래스 및 기본 클래스 포인터로의 역 변환에 액세스할 수 있습니다.
- 파생 클래스는 기본 클래스에서 사실상 상속되지 않습니다.

왼쪽 피연산자가 멤버에 대한 포인터인 경우 오른쪽 피연산자가 멤버 포인터 형식이거나 0으로 계산되는 상수 식이어야 합니다. 이 대입은 다음과 같은 경우에만 유효합니다.

- 오른쪽 피연산자가 왼쪽 피연산자와 같은 클래스의 멤버에 대한 포인터입니다.
- 왼쪽 피연산자가 오른쪽 피연산자의 클래스로부터 공개적으로 명확하게 파생된 클래스의 멤버에 대한 포인터입니다.

멤버 변환에 대한 null 포인터

0으로 계산되는 정수 상수 식은 null 포인터로 변환됩니다. 이 포인터는 항상 포인터와 같지 않은 개체 또는 함수를 비교합니다. 예외는 동일한 오프셋을 가질 수 있고 여전히 다른 개체를 가리킬 수 있는 기반 개체에 대한 포인터입니다.

다음 코드에서는 `i` 클래스의 `A` 멤버에 대한 포인터 정의에 대해 설명합니다. `pai` 포인터는 0으로 초기화되어 null 포인터가 됩니다.

C++

```
class A
{
public:
    int i;
};

int A::*pai = 0;

int main()
{}
```

참고 항목

[C++ 언어 참조](#)

기본 제공 형식(C++)

아티클 • 2023. 10. 12.

기본 제공 형식(기본 형식이라고도 함)은 C++ 언어 표준에 의해 지정되며 컴파일러에 기본 제공됩니다. 기본 제공 형식은 헤더 파일에 정의되지 않습니다. 기본 제공 형식은 정수, 부동 소수점 및 `void`의 세 가지 기본 범주로 나뉩니다. 정수 형식은 정수를 나타냅니다. 부동 소수점 형식은 소수 부분이 있을 수 있는 값을 지정할 수 있습니다. 대부분의 기본 제공 형식은 컴파일러에서 고유 형식으로 처리됩니다. 그러나 일부 형식은 동의어이거나 컴파일러에서 동등한 형식으로 처리됩니다.

void 형식

이 형식은 `void` 빈 값 집합을 설명합니다. 형식 `void`의 변수를 지정할 수 없습니다. 이 `void` 형식은 주로 값을 반환하지 않는 함수를 선언하거나 형식화되지 않거나 임의로 형식화된 데이터에 대한 제네릭 포인터를 선언하는 데 사용됩니다. 모든 식은 `void` 형식으로 명시적으로 변환되거나 캐스팅될 수 있습니다. 그러나 이러한 식은 사용이 다음으로 제한됩니다.

- 식 문. (자세한 내용은 [를 참조하세요 .식.](#))
- 쉼표 연산자의 왼쪽 피연산자. (자세한 내용은 [를 참조하세요 .쉼표 연산자](#)입니다.)
- 조건 연산자의 두 번째 또는 세 번째 피연산자(`? :`)입니다. (자세한 내용은 [를 참조하세요 .조건부 연산자](#)를 사용하는 식입니다.)

std::nullptr_t

키워드(keyword) `nullptr` 모든 원시 포인터 형식으로 변환할 수 있는 형식 `std::nullptr_t`의 null 포인터 상수입니다. 자세한 내용은 [nullptr](#)를 참조하세요.

부울 형식

형식에는 `bool` 값 `true` 과 `false`. 형식의 `bool` 크기는 구현에 따라 다릅니다. Microsoft 관련 구현 세부 정보는 [기본 제공 형식의 크기를 참조하세요](#).

문자 형식

형식은 `char` 기본 실행 문자 집합의 멤버를 효율적으로 인코딩하는 문자 표현 형식입니다. C++ 컴파일러는 `char`, `signed char` 및 `unsigned char` 형식의 변수를 서로 다른 형식

으로 처리합니다.

Microsoft 관련: 컴파일 옵션을 사용하지 않는 한 /J 형식 `char` 의 변수는 기본적으로 형식 `signed char` 에서 온 것처럼 승격 `int` 됩니다. 이 경우 형식 `unsigned char` 으로 처리되고 서명 확장 없이 승격 `int` 됩니다.

형식 `wchar_t` 의 변수는 와이드 문자 또는 멀티바이트 문자 형식입니다. `L` 문자 또는 문자열 리터럴 앞에 접두사를 사용하여 와이드 문자 형식을 지정합니다.

Microsoft 전용: 기본적으로 `wchar_t` 네이티브 형식이지만, 예 대한 `unsigned short` typedef를 만드는 `wchar_t` 데 사용할 `/Zc:wchar_t-` 수 있습니다. `_wchar_t` 형식은 네이티브 `wchar_t` 형식의 Microsoft 전용 동의어입니다.

이 `char8_t` 형식은 UTF-8 문자 표현에 사용됩니다. 표현은 같지 `unsigned char` 만 컴파일러에서 고유 형식으로 처리됩니다. `char8_t` C++20의 새로운 형식입니다. **Microsoft 관련:** 사용 `char8_t` 하려면 컴파일러 옵션 이상(예: `/std:c++latest`)이 필요합니다 `/std:c++20`.

이 `char16_t` 형식은 UTF-16 문자 표현에 사용됩니다. UTF-16 코드 단위를 나타낼 수 있을 만큼 커야 합니다. 컴파일러에서 고유 형식으로 처리됩니다.

이 `char32_t` 형식은 UTF-32 문자 표현에 사용됩니다. UTF-32 코드 단위를 나타낼 수 있을 만큼 커야 합니다. 컴파일러에서 고유 형식으로 처리됩니다.

부동 소수점 형식

부동 소수점 형식은 IEEE-754 표현을 사용하여 광범위한 크기의 소수 자릿수 값을 근사값으로 제공합니다. 다음 표에서는 C++의 부동 소수점 형식과 부동 소수점 형식 크기에 대한 비교 제한을 나열합니다. 이러한 제한은 C++ 표준에 의해 의무화되며 Microsoft 구현과 독립적입니다. 기본 제공 부동 소수점 형식의 절대 크기는 표준에 지정되지 않습니다.

Type	콘텐츠
<code>float</code>	형식 <code>float</code> 은 C++에서 가장 작은 부동 소수점 형식입니다.
<code>double</code>	<code>double</code> 형식은 <code>float</code> 형식보다 크거나 같지만 <code>long double</code> 형식의 크기보다 짧거나 같은 부동 소수점 형식입니다.
<code>long double</code>	<code>long double</code> 형식은 <code>double</code> 형식보다 크거나 같은 부동 소수점 형식입니다.

Microsoft 관련: 표현 `long double` 과 `double` 동일합니다. 그러나 `long double double` 컴파일러에서 고유 형식으로 처리됩니다. Microsoft C++ 컴파일러는 4 바이트 및 8 바이트

IEEE-754 부동 소수점 표현을 사용합니다. 자세한 내용은 IEEE 부동 소수점 표현[을 참조하세요](#).

정수 형식

`int` 이 형식은 기본 기본 정수 형식입니다. 구현별 범위에 대한 모든 정수를 나타낼 수 있습니다.

부인정수 표현은 양수 값과 음수 값을 모두 보유할 수 있는 정수 표현입니다. 기본적으로 사용되거나 한정자 키워드(keyword) 있는 경우에 `signed` 사용됩니다. `unsigned` 한정자 키워드(keyword) 음수가 아닌 값만 보유할 수 있는 부호 없는 표현을 지정합니다.

크기 한정자는 사용된 정수 표현의 너비를 비트 단위로 지정합니다. 언어는, `long` 및 `long long` 한정자를 지원합니다 `short`. 형식은 `short` 너비가 16비트 이상이어야 합니다. 형식은 `long` 너비가 32비트 이상이어야 합니다. 형식은 `long long` 너비가 64비트 이상이어야 합니다. 표준은 정수 계열 형식 간의 크기 관계를 지정합니다.

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

구현은 기본 각 형식에 대한 최소 크기 요구 사항과 크기 관계를 모두 충족해야 합니다. 그러나 실제 크기는 구현마다 다를 수 있습니다. Microsoft 관련 구현 세부 정보는 기본 제공 형식[의 크기를 참조하세요](#).

`int` 또는 크기 한정자를 지정할 때 `signed unsigned` 키워드(keyword) 생략할 수 있습니다. 한정자와 `int` 형식(있는 경우)이 순서대로 나타날 수 있습니다. 예를 들어 `short unsigned unsigned int short` 동일한 형식을 참조합니다.

정수 형식 동의어

다음 형식 그룹은 컴파일러에서 동의어로 간주됩니다.

- `short, short int, signed short, signed short int`
- `unsigned short, unsigned short int`
- `int, signed, signed int`
- `unsigned, unsigned int`
- `long, long int, signed long, signed long int`
- `unsigned long, unsigned long int`

- `long long`, `long long int`, `signed long long`, `signed long long int`
- `unsigned long long`, `unsigned long long int`

Microsoft 특정 정수 형식에는 특정 너비 `_int8`, `_int16` `_int32` 및 `_int64` 형식이 포함됩니다. 이러한 형식은 `signed` 및 `unsigned` 한정자를 사용할 수 있습니다. `_int8` 데이터 형식은 `char` 형식과 동의어이고, `_int16`은 `short`와 동의어이고, `_int32`는 `int`와 동의어이고, `_int64`는 `long long`과 동의어입니다.

기본 제공 형식의 크기

대부분의 기본 제공 형식에는 구현 정의 크기가 있습니다. 다음 표에서는 Microsoft C++의 기본 제공 형식에 필요한 스토리지 양을 나열합니다. 특히 `long` 64비트 운영 체제에서도 4바이트입니다.

Type	크기
<code>bool</code> , <code>char</code> , <code>char8_t</code> , <code>unsigned char</code> , <code>signed char</code> , <code>_int8</code>	1바이트
<code>char16_t</code> , <code>_int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>_wchar_t</code>	2바이트
<code>char32_t</code> , <code>float</code> , <code>_int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>	4바이트
<code>double</code> , <code>_int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code>	8바이트

각 형식의 값 범위에 대한 요약은 데이터 형식 범위를 참조하세요.

형식 변환에 대한 자세한 내용은 표준 변환을 참조하세요.

참고 항목

[데이터 형식 범위](#)

데이터 형식 범위

아티클 • 2024. 11. 21.

Microsoft C++ 32비트 및 64비트 컴파일러는 이 문서의 뒷부분의 표에 나온 형식을 인식합니다.

C++

```
- int (unsigned int)
- __int8 (unsigned __int8)
- __int16 (unsigned __int16)
- __int32 (unsigned __int32)
- __int64 (unsigned __int64)
- short (unsigned short)
- long (unsigned long)
- long long (unsigned long long)
```

이름이 두 개의 밑줄(____)로 시작하는 경우 데이터 형식은 비표준입니다.

다음 표에 지정된 범위는 포함-포함입니다.

[+] 테이블 확장

형식 이름	바이트	기타 이름	값의 범위
int	4	signed	-2,147,483,648 ~ 2,147,483,647
unsigned int	4	unsigned	0 ~ 4,294,967,295
__int8	1	char	-128 ~ 127
unsigned __int8	1	unsigned char	0 ~ 255
__int16	2		-32,768 ~ 32,767
unsigned __int16	2	unsigned short: unsigned short int	0 ~ 65,535
__int32	4		-2,147,483,648 ~ 2,147,483,647
unsigned __int32	4	unsigned: unsigned int	0 ~ 4,294,967,295
__int64	8	long long: signed long long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

형식 이름	바이트	기타 이름	값의 범위
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	0 ~ 18,446,744,073,709,551,615
<code>bool</code>	1	없음	<code>false</code> 또는 <code>true</code>
<code>char</code>	1	없음	-128~127(기본값) <code>/J</code> 를 사용하여 컴파일된 경우 0~255
<code>signed char</code>	1	없음	-128 ~ 127
<code>unsigned char</code>	1	없음	0 ~ 255
<code>short</code>	2	<code>short int</code> : <code>signed short int</code>	-32,768 ~ 32,767
<code>unsigned short</code>	2	<code>unsigned short int</code>	0 ~ 65,535
<code>long</code>	4	<code>long int</code> : <code>signed long int</code>	-2,147,483,648 ~ 2,147,483,647
<code>unsigned long</code>	4	<code>unsigned long int</code>	0 ~ 4,294,967,295
<code>long long</code>	8	없음(그러나 <code>__int64</code> 와 동일)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>unsigned long long</code>	8	없음(그러나 <code>unsigned __int64</code> 와 동일)	0 ~ 18,446,744,073,709,551,615
<code>enum</code>	다양함	없음	
<code>float</code>	4	없음	3.4E +/- 38(7자리 숫자)
<code>double</code>	8	없음	1.7E +/- 308(15자리 숫자)
<code>long double</code>	<code>double</code> 과 같습니다.	없음	<code>double</code> 과 같음
<code>wchar_t</code>	2	<code>_wchar_t</code>	0 ~ 65,535

`_wchar_t` 변수는 와이드 문자 형식 또는 멀티바이트 문자 형식을 지정합니다. 문자 또는 문자열 상수 앞에 `L` 접두사를 사용하여 와이드 문자 형식 상수를 지정합니다.

`signed` 및 `unsigned` 는 `bool`을 제외한 모든 정수 형식에서 사용할 수 있는 한정자입니다.
`char`, `signed char` 및 `unsigned char`는 오버로드 및 템플릿과 같은 메커니즘에 사용되는

3가지 고유 형식입니다.

`int` 및 `unsigned int` 형식의 크기는 4바이트입니다. 그러나 언어 표준에서 구현 전용으로 허용되므로 이식 가능한 코드는 `int`의 크기에 의존해서는 안 됩니다.

Visual Studio의 C/C++에서도 크기가 지정된 정수 형식을 지원합니다. 자세한 내용은 [_int8, _int16, _int32, _int64](#) 및 [정수 제한](#)을 참조하세요.

각 형식의 크기 제한에 대한 자세한 내용은 [기본 제공 형식](#)을 참조하세요.

열거 형식의 범위는 언어 컨텍스트 및 지정된 컴파일러 플래그에 따라 달라집니다. 자세한 내용은 [C 열거형 선언](#) 및 [열거형](#)을 참조하세요.

참고 항목

[키워드](#)

[기본 제공 형식](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

nullptr

아티클 • 2024. 11. 21.

키워드는 `nullptr` 형식의 null 포인터 상수(원시 포인터 형식 `std::nullptr_t`으로 변환 가능)를 지정합니다. 헤더를 포함하지 않고 키워드 `nullptr`를 사용할 수 있지만 코드에서 형식 `std::nullptr_t`을 사용하는 경우 헤더 `<cstddef>`를 포함하여 정의해야 합니다.

① 참고

`nullptr` 키워드는 관리 코드 애플리케이션에 대한 C++/CLI에서도 정의되며 ISO 표준 C++ 키워드와 교환할 수 없습니다. 코드가 관리 코드를 대상으로 하는 컴파일러 옵션을 사용하여 `/clr` 컴파일될 수 있는 경우 컴파일러에서 네이티브 C++ 해석을 사용하도록 보장해야 하는 모든 코드 줄에서 사용합니다 `_nullptr`. 자세한 내용은 [nullptr \(C++/CLI 및 C++/CX\)](#)를 참조하세요.

설명

null 포인터 상수 `nullptr`로 사용하거나 0(`0`)을 사용하지 `NULL` 마세요. 오용에 덜 취약하며 대부분의 상황에서 더 잘 작동합니다. 예를 들어 `func(std::pair<const char *, double>)` 가 주어진 경우 `func(std::make_pair(NULL, 3.14))` 를 호출하면 컴파일러 오류가 발생합니다. 매크로 `NULL` 가 확장 `0` 되어 호출 `std::make_pair(0, 3.14)` 이 반환 `std::pair<int, double>` 되므로 매개 변수 형식 `func` 으로 변환할 `std::pair<const char *, double>` 수 없습니다. `func(std::make_pair(nullptr, 3.14))` 는 `std::make_pair(nullptr, 3.14)` 로 변환될 수 있는 `std::pair<std::nullptr_t, double>` 를 반환하기 때문에 `std::pair<const char *, double>` 를 호출하면 성공적으로 컴파일됩니다.

참고 항목

키워드

[nullptr \(C++/CLI 및 C++/CX\)](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

void (C++)

아티클 • 2024. 11. 21.

함수 반환 형식으로 사용되는 경우 키워드는 `void` 함수가 값을 반환하지 않도록 지정합니다. 함수의 매개 변수 목록에 `void` 사용되는 경우 함수가 매개 변수를 사용하지 않음을 지정합니다. 포인터 선언에 사용되는 경우 포인터 `void` 가 "범용"임을 지정합니다.

포인터의 형식이면 포인터는 `void*` 키워드로 `volatile` 선언되지 않은 변수를 `const` 가리킬 수 있습니다. 포인터가 `void*` 다른 형식으로 캐스팅되지 않는 한 역참조할 수 없습니다. 포인터는 `void*` 다른 형식의 데이터 포인터로 변환할 수 있습니다.

C++에서 포인터는 `void` 자유 함수(클래스의 멤버가 아닌 함수) 또는 정적 멤버 함수를 가리킬 수 있지만 비정적 멤버 함수는 가리킬 수 없습니다.

형식 `void`의 변수를 선언할 수 없습니다.

스타일에 따라 C++ 핵심 지침에서는 빈 정식 매개 변수 목록을 지정하는 데 사용하지 `void` 않는 것이 좋습니다. 자세한 내용은 [C++ 핵심 지침 NL.25: 인수 형식](#)으로 사용하지 `void` 마세요.

예시

```
C++  
  
// void.cpp  
  
void return_nothing()  
{  
    // A void function can have a return with no argument,  
    // or no return statement.  
}  
  
void vobject;    // C2182  
void *pv;      // okay  
int *pint; int i;  
int main()  
{  
    pv = &i;  
    // Cast is optional in C, required in C++  
    pint = (int *)pv;  
}
```

참고 항목

피드백

이 페이지가 도움이 되었나요?  

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

bool (C++)

아티클 • 2024. 08. 02.

이 키워드는 기본 제공 형식입니다. 이 형식의 변수에는 값 `true` 과 `false`. 조건식에는 형식이 있으므로 형식 `bool` `bool`의 값이 있습니다. 예를 들어 `i != 0` 이제 값이 `i` 있거나 `false` `true` 값에 따라 달라집니다.

Visual Studio 2017 버전 15.3 이상(/std:c++17 이상에서 사용 가능): 접두사 또는 접두사 증가 또는 감소 연산자의 피연산자는 형식 `bool` 이 아닐 수 있습니다. 즉, 형식 `bool`의 변수 `b` 를 지정하면 이러한 식은 더 이상 허용되지 않습니다.

C++

```
b++;
++b;
b--;
--b;
```

값 `true` 과 `false` 다음과 같은 관계가 있습니다.

C++

```
!false == true
!true == false
```

다음 문에서

C++

```
if (condexpr1) statement1;
```

이 `statement1` `true` 면 `condexpr1` 항상 실행됩니다. 이 `statement1` `false` 경우 `condexpr1` 실행되지 않습니다.

후위 또는 접두사 `++` 연산자가 형식 `bool` 변수에 적용되면 변수는 .로 `true` 설정됩니다.

Visual Studio 2017 버전 15.3 이상: `operator++` `bool` 언어에서 제거되었으며 더 이상 지원되지 않습니다.

이 형식의 변수에는 접두사 또는 접두사 `--` 연산자를 적용할 수 없습니다.

이 형식은 `bool` 기본 정수 승격에 참여합니다. 형식 `bool` 의 r-값은 0 `true` 이 되고 1이 되는 형식 `int` 의 r 값으로 `false` 변환할 수 있습니다. 고유한 형식으로 오버 `bool` 로드 확

인에 참여합니다.

참고 항목

키워드

기본 제공 형식

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

false (C++)

아티클 • 2024. 11. 21.

키워드는 부울 형식 변수 또는 조건식의 두 값 중 하나입니다(조건식은 이제 `true` 부울 식입니다). 예를 들어 형식의 변수인 경우 `i` 문은 `i = false;` .에 `i` 할당됩니다

`false`.`bool`

예시

C++

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

true (C++)

아티클 • 2024. 11. 21.

구문

```
bool-identifier = true ;
bool-expression logical-operator true ;
```

설명

이 키워드는 부울 형식 변수 또는 조건식의 두 값 중 하나입니다(조건식은 이제 진정한 부울 식입니다). 형식이면 `i` 문이 `i = true;` .에 `i` 할당됩니다 `true`. `bool`

예시

C++

```
// bool_true.cpp
#include <stdio.h>
int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

char, wchar_t, char8_t, char16_t, char32_t

아티클 • 2024. 05. 10.

형식 `char`, `wchar_t`, `char8_t` `char16_t` 및 `char32_t` 은(는) 영숫자 문자, 영숫자가 아닌 문자 모양 및 인쇄되지 않는 문자를 나타내는 기본 제공 형식입니다.

구문

C++

```
char      ch1{ 'a' }; // or { u8'a' }
wchar_t   ch2{ L'a' };
char16_t  ch3{ u'a' };
char32_t  ch4{ U'a' };
```

설명

`char` 형식은 C와 C++의 독창적인 문자 형식이었습니다. 이 형식은 `char` ASCII 문자 집합 또는 ISO-8859 문자 집합의 문자와 Shift-JIS 또는 유니코드 문자 집합의 UTF-8 인코딩과 같은 멀티바이트 문자의 개별 바이트를 저장합니다. Microsoft 컴파일러에서 `char` 은(는) 8비트 형식입니다. 이는 `signed char` 및 `unsigned char` 모두의 고유한 형식입니다. 기본적으로 `char` 형식의 변수는 `/J` 컴파일러 옵션을 사용하지 않는 한 `signed char` 형식에서 와 같이 `int`(으)로 승격됩니다. `/J`에서는 `unsigned char` 형식으로 취급되며 부호 확장 없이 `int`(으)로 승격됩니다.

`unsigned char` 형식은 종종 C++의 기본 제공 형식이 아닌 `바이트`를 나타내는 데 사용됩니다.

`wchar_t` 형식은 구현에서 정의한 와이드 문자 형식입니다. Microsoft 컴파일러에서는 Windows 운영 체제의 기본 문자 형식인 UTF-16LE로 인코딩된 유니코드를 저장하는 데 사용되는 16비트 와이드 문자를 나타냅니다. UCRT(유니버설 C 런타임) 라이브러리 함수의 와이드 문자 버전은 네이티브 Windows API의 와이드 문자 버전과 마찬가지로 `wchar_t` 과 해당 포인터 및 배열 형식을 매개 변수 및 반환 값으로 사용합니다.

`char8_t`, `char16_t` 및 `char32_t` 형식은 각각 8비트, 16비트 및 32비트 너비 문자를 나타냅니다. (`char8_t` 은(는) C++20의 새로운 기능이며 `/std:c++20` 또는 `/std:c++latest` 컴파일러 옵션이 필요합니다.) UTF-8로 인코딩된 유니코드는 `char8_t` 형식에 저장할 수 있습니다. `char8_t` 및 `char` 형식의 문자열은 유니코드 또는 멀티바이트 문자를 인코딩하는 데 사용되는 경우에도 내로우(좁은) 문자열이라고 합니다. UTF-16으로 인코딩된 유니코

드는 `char16_t` 형식에 저장할 수 있으며 UTF-32로 인코딩된 유니코드는 `char32_t` 형식에 저장할 수 있습니다. 이러한 유형과 `wchar_t`의 문자열은 모두 와이드(넓은) 문자열이라고 하지만 이 용어는 종종 `wchar_t` 유형의 문자열을 구체적으로 나타냅니다.

C++ 표준 라이브러리에서 `basic_string` 형식은 좁은 문자열과 와이드 문자열 모두에 특화되었습니다. 문자가 `char` 유형인 경우 `std::string`을 사용하고, 문자가 `char8_t` 유형인 경우 `std::u8string`을 사용하고, 문자가 `char16_t` 유형인 경우 `std::u16string`을 사용하고, 문자가 `char32_t` 유형인 경우 `std::u32string`을 사용하고, 문자가 `wchar_t` 유형인 경우 `std::wstring`을 사용합니다.

`std::stringstream` 및 `std::cout`을(를) 포함하여 텍스트를 나타내는 다른 형식에는 좁고 넓은 문자열에 대한 특수화가 있습니다.

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__int8, __int16, __int32, __int64

아티클 • 2023. 10. 12.

Microsoft 전용

Microsoft C/C++는 크기가 지정된 정수 형식을 지원합니다. 형식 지정자(8, 16, 32 또는 64)를 사용하여 `__intN` 8, `N` 16, 32 또는 64비트 정수 변수를 선언할 수 있습니다.

다음 예제에서는 이러한 형식의 크기가 지정된 정수 각각에 대해 변수 하나를 선언합니다.

C++

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

형식 `__int8` `__int16` 은 크기가 같고 `__int32` 여러 플랫폼에서 동일하게 동작하는 이식 가능한 코드를 작성하는데 유용한 ANSI 형식의 동의어입니다. 데이터 형식은 `__int8` 형식 `char`과 동의어이고 형식 `__int16` 과 동의어이며 형식 `short int`과 `__int32` 동의어입니다. 형식은 `__int64` 형식 `long long`과 동의어입니다.

이전 버전과의 호환성을 위해, `__int8`, 및 컴파일러 옵션 [/Za \(언어 확장 사용 안 함\)](#)을 지정하지 않는 한, `__int16` `__int32`, 및 `__int64` 동의어 `__int8`입니다. `__int64` `__int32` `__int16`

예시

다음 샘플에서는 매개 변수가 다음으로 `__intN` 승격됨을 보여 주는 예제 `int`입니다.

C++

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
{
    __int8 i8 = 100;
    func(i8); // no void func(__int8 i8) function
```

```
// __int8 will be promoted to int  
}
```

Output

```
func
```

참고 항목

[키워드](#)

[기본 제공 형식](#)

[데이터 형식 범위](#)

__m64

아티클 • 2023. 10. 12.

Microsoft 전용

데이터 형식은 __m64 MMX 및 3DNow! 내장 함수이며 xmmintrin.h>에 정의됩니다.

C++

```
// data_types__m64.cpp
#include <xmmintrin.h>
int main()
{
    __m64 x;
```

설명

필드에 직접 액세스 __m64 해서는 안 됩니다. 그러나 디버거에서 이러한 형식을 볼 수 있습니다. 형식 __m64의 변수는 MM[0-7] 레지스터에 매핑됩니다.

_m64 형식의 변수는 8 바이트 경계에 자동으로 정렬됩니다.

__m64 x64 프로세서에서는 데이터 형식이 지원되지 않습니다. MMX 내장 함수의 일부로 _m64를 사용하는 애플리케이션은 동등한 SSE 및 SSE2 내장 함수를 사용하도록 다시 작성해야 합니다.

Microsoft 전용 종료

참고 항목

[키워드](#)

[기본 제공 형식](#)

[데이터 형식 범위](#)

_m128

아티클 • 2023. 10. 12.

Microsoft 전용

스트리밍 SIMD 확장 및 스트리밍 SIMD 확장 2 명령 내장 함수와 함께 사용할 `_m128` 데이터 형식은 `xmmmintrin.h`에 정의되어 있습니다.

C++

```
// data_types_m128.cpp
#include <xmmmintrin.h>
int main() {
    _m128 x;
}
```

설명

필드에 직접 액세스 `_m128` 해서는 안 됩니다. 그러나 디버거에서 이러한 형식을 볼 수 있습니다. 형식 `_m128`의 변수는 XMM[0-7] 레지스터에 매핑됩니다.

형식 `_m128`의 변수는 16 바이트 경계에 자동으로 정렬됩니다.

`_m128` ARM 프로세서에서는 데이터 형식이 지원되지 않습니다.

Microsoft 전용 종료

참고 항목

[키워드](#)

[기본 제공 형식](#)

[데이터 형식 범위](#)

__m128d

아티클 • 2023. 10. 12.

Microsoft 전용

__m128d 스트리밍 SIMD 확장 2 명령 내장 함수와 함께 사용할 데이터 형식은 `<emmintrin.h>`에 정의되어 있습니다.

C++

```
// data_types__m128d.cpp
#include <emmintrin.h>
int main() {
    __m128d x;
}
```

설명

필드에 직접 액세스 __m128d 해서는 안 됩니다. 그러나 디버거에서 이러한 형식을 볼 수 있습니다. 형식 __m128의 변수는 XMM[0-7] 레지스터에 매핑됩니다.

_m128d 형식의 변수는 16 바이트 경계에 자동으로 정렬됩니다.

__m128d ARM 프로세서에서는 데이터 형식이 지원되지 않습니다.

Microsoft 전용 종료

참고 항목

[키워드](#)

[기본 제공 형식](#)

[데이터 형식 범위](#)

__m128i

아티클 • 2023. 10. 12.

Microsoft 전용

__m128i SSE2(Streaming SIMD Extensions 2) 명령 내장 함수와 함께 사용할 데이터 형식은 `<emmintrin.h>`에 정의되어 있습니다.

C++

```
// data_types__m128i.cpp
#include <emmintrin.h>
int main() {
    __m128i x;
}
```

설명

필드에 직접 액세스 **__m128i** 해서는 안 됩니다. 그러나 디버거에서 이러한 형식을 볼 수 있습니다. 형식 **__m128i**의 변수는 XMM[0-7] 레지스터에 매핑됩니다.

형식 **__m128i**의 변수는 16 바이트 경계에 자동으로 정렬됩니다.

① 참고

형식 **__m128i**의 변수를 사용하면 컴파일러가 SSE2 `movdqa` 명령을 생성합니다. 이 명령은 펜티엄 III 프로세서에 오류를 발생시키지 않지만, 펜티엄 III 프로세서에서 어떤 명령 `movdqa`으로 변환되든 가능한 부작용으로 인해 자동 오류가 발생합니다.

__m128i ARM 프로세서에서는 데이터 형식이 지원되지 않습니다.

Microsoft 전용 종료

참고 항목

키워드

기본 제공 형식

데이터 형식 범위

__ptr32, __ptr64

아티클 • 2024. 11. 21.

Microsoft 전용

__ptr32 는 32비트 시스템의 네이티브 포인터를 나타내고 **__ptr64** 64비트 시스템의 네이티브 포인터를 나타냅니다.

다음 예제에서는 이러한 포인터 형식 각각을 선언하는 방법을 보여 줍니다.

C++

```
int * __ptr32 p32;  
int * __ptr64 p64;
```

32비트 시스템에서 선언된 **__ptr64** 포인터는 32비트 포인터로 잘립니다. 64비트 시스템에서 선언된 **__ptr32** 포인터는 64비트 포인터로 강제 변환됩니다.

① 참고

/clr:pure로 컴파일할 때는 사용할 **__ptr32** **__ptr64** 수 없습니다. 그렇지 않으면 컴파일러 오류 C2472가 생성됩니다. /clr:pure 및 /clr:safe 컴파일러 옵션은 Visual Studio 2015에서 더 이상 사용되지 않으며 Visual Studio 2017에서는 지원되지 않습니다.

이전 버전과의 호환성을 위해 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)를 지정하지 않는 한 **_ptr32** 및 **_ptr64** _ptr64 동의어 **__ptr32**입니다.

예시

다음 예제에서는 및 키워드를 사용하여 포인터를 선언하고 할당하는 **__ptr32** **__ptr64** 방법을 보여 줍니다.

C++

```
#include <cstdlib>  
#include <iostream>  
  
int main()  
{  
    using namespace std;  
  
    int * __ptr32 p32;  
    int * __ptr64 p64;
```

```
p32 = (int * __ptr32)malloc(4);
*p32 = 32;
cout << *p32 << endl;

p64 = (int * __ptr64)malloc(4);
*p64 = 64;
cout << *p64 << endl;
}
```

Output

```
32
64
```

Microsoft 전용 종료

참고 항목

[기본 제공 형식](#)

피드백

이 페이지가 도움이 되었나요? [!\[\]\(d44d8fe394c839707eb6bc14205803a9_img.jpg\) Yes](#) [!\[\]\(e8e2830a8aafd40059ef4db2b5ff2f56_img.jpg\) No](#)

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

숫자 제한 (C++)

아티클 • 2023. 10. 12.

두 표준 포함 파일, <limits.h> 및 <float.h>는 숫자 제한 또는 지정된 형식의 변수가 보유할 수 있는 최소값과 최대값을 정의합니다. 이러한 최소값 및 최대값은 ANSI C와 동일한 데이터 표현을 사용하는 모든 C++ 컴파일러에 이식 가능하도록 보장됩니다. limits.h 포함 파일은 [정수 계열 형식에 대한 숫자 제한을 정의](#)하고 <float.h>는 부동 형식에 대한 숫자 제한을 정의합니다.>

참고 항목

[기본 개념](#)

정수 제한

아티클 • 2023. 10. 12.

Microsoft 전용

다음 표에서는 정수 형식에 대한 제한 사항을 보여 줍니다. 이러한 제한에 대한 전처리기 매크로는 표준 헤더 파일 <클리미트도 포함할 때 정의됩니다>.

정수 상수에 대한 제한

상수	의미	값
CHAR_BIT	비트 필드가 없는 가장 작은 변수의 비트 수입니다.	8
SCHAR_MIN	<code>signed char</code> 형식 변수의 최소값입니다.	-128
SCHAR_MAX	<code>signed char</code> 형식 변수의 최대값입니다.	127
UCHAR_MAX	<code>unsigned char</code> 형식 변수의 최대값입니다.	255(0xff)
CHAR_MIN	<code>char</code> 형식 변수의 최소값입니다.	-128; 0 if /J option used
CHAR_MAX	<code>char</code> 형식 변수의 최대값입니다.	127; 옵션을 사용하는 경우 /J 255
MB_LEN_MAX	여러 문자 상수에서의 최대 바이트 수입니다.	5
SHRT_MIN	<code>short</code> 형식 변수의 최소값입니다.	-32768
SHRT_MAX	<code>short</code> 형식 변수의 최대값입니다.	32767
USHRT_MAX	<code>unsigned short</code> 형식 변수의 최대값입니다.	65535(0xffff)
INT_MIN	<code>int</code> 형식 변수의 최소값입니다.	-2147483648
INT_MAX	<code>int</code> 형식 변수의 최대값입니다.	2147483647
UINT_MAX	<code>unsigned int</code> 형식 변수의 최대값입니다.	4294967295(0xffffffff)
LONG_MIN	<code>long</code> 형식 변수의 최소값입니다.	-2147483648
LONG_MAX	<code>long</code> 형식 변수의 최대값입니다.	2147483647
ULONG_MAX	<code>unsigned long</code> 형식 변수의 최대값입니다.	4294967295(0xffffffff)
LLONG_MIN	형식 변수의 최소값 <code>long long</code>	-9223372036854775808

상수	의미	값
LLONG_MAX	형식 변수의 최대값 <code>long long</code>	9223372036854775807
ULLONG_MAX	형식 변수의 최대값 <code>unsigned long long</code>	18446744073709551615(0xffffffffffffffffffff)

값이 최대 정수 표현을 초과하는 경우 Microsoft 컴파일러에서 오류가 발생합니다.

참고 항목

[부동 한계](#)

부동 한계

아티클 • 2023. 10. 12.

Microsoft 전용

다음 표에는 부동 소수점 상수의 값에 대한 제한이 나와 있습니다. 이러한 제한은 표준 헤더 파일 <float.h>에도 정의됩니다.

부동 소수점 상수에 대한 제한

상수	의미	값
<code>FLT_DIG</code>	소수 자릿수가 q인 부동 소수점 수가 부동 소수점 표현으로 반올림되고 정밀도의 손실 없이 다시 복원될 수 있는 자릿수 q입니다.	6
<code>DBL_DIG</code>		15
<code>LDBL_DIG</code>		15
<code>FLT_EPSILON</code>	$x + 1.0$ 이 1.0과 같지 않은 가장 작은 양수 x 입니다.	1.192092896e-07F
<code>DBL_EPSILON</code>		2.2204460492503131e-016
<code>LDBL_EPSILON</code>		2.2204460492503131e-016
<code>FLT_GUARD</code>		0
<code>FLT_MANT_DIG</code>	부동 소수점에서 지정한 <code>FLT_RADIX</code> radix의 숫자 수입니다. 기수는 2입니다. 따라서 이러한 값이 비트를 지정합니다.	24
<code>DBL_MANT_DIG</code>		53
<code>LDBL_MANT_DIG</code>		53
<code>FLT_MAX</code>	표현 가능한 최대 부동 소수점 수입니다.	3.402823466e+38F
<code>DBL_MAX</code>		1.7976931348623158e+308
<code>LDBL_MAX</code>		1.7976931348623158e+308
<code>FLT_MAX_10_EXP</code>	숫자에 10이 더해진 최대 정수는 표현 가능한 부동 소수점 숫자입니다.	38
<code>DBL_MAX_10_EXP</code>		308
<code>LDBL_MAX_10_EXP</code>		308
<code>FLT_MAX_EXP</code>	해당 숫자로 올라온 최대 정 <code>FLT_RADIX</code> 수는 표시 가능한 부동 소수점 숫자입니다.	128
<code>DBL_MAX_EXP</code>		1024
<code>LDBL_MAX_EXP</code>		1024
<code>FLT_MIN</code>	최소 양수 값입니다.	1.175494351e-38F
<code>DBL_MIN</code>		2.2250738585072014e-308
<code>LDBL_MIN</code>		2.2250738585072014e-308
<code>FLT_MIN_10_EXP</code>	해당 숫자로 10이 표시 가능한 부동 소수점 숫자 인 최소 음수 정수입니다.	-37
<code>DBL_MIN_10_EXP</code>		-307
<code>LDBL_MIN_10_EXP</code>		-307

상수	의미	값
<code>FLT_MIN_EXP</code>	해당 숫자로 올라온 최소 음수 정 <code>FLT_RADIX</code> 수	-125
<code>DBL_MIN_EXP</code>	는 표시 가능한 부동 소수점 숫자입니다.	-1021
<code>LDBL_MIN_EXP</code>		-1021
<code>FLT_NORMALIZE</code>		0
<code>FLT_RADIX</code>	지수를 표현하는 기수입니다.	2
<code>_DBL_RADIX</code>		2
<code>_LDBL_RADIX</code>		2
<code>FLT_ROUNDS</code>	부동 소수점 더하기의 반올림 모드입니다.	1(근사값)
<code>_DBL_ROUNDS</code>		1(근사값)
<code>_LDBL_ROUNDS</code>		1(근사값)

① 참고

표의 정보는 제품의 이후 버전에서 달라질 수 있습니다.

Microsoft 전용 종료

참고 항목

[정수 제한](#)

선언 및 정의(C++)

아티클 • 2024. 07. 12.

C++ 프로그램은 변수, 함수, 형식, 네임스페이스 등 다양한 엔티티로 구성됩니다. 이러한 각 엔티티는 선언된 사용해야만 사용할 수 있습니다. 선언은 엔티티의 유형 및 기타 특성에 대한 정보와 함께 엔티티의 고유 이름을 지정합니다. C++에서 이름이 선언되는 시점은 컴파일러에 표시되는 시점입니다. 컴파일 단위의 나중에 선언된 함수나 클래스는 참조할 수 없습니다. 변수는 사용 시점 이전에 가능한 한 가깝게 선언해야 합니다.

다음 예제에서는 몇 가지 선언을 보여 줍니다.

```
C++  
  
#include <string>  
  
int f(int i); // forward declaration  
  
int main()  
{  
    const double pi = 3.14; //OK  
    int i = f(2); //OK. f is forward-declared  
    C obj; // error! C not yet declared.  
    std::string str; // OK std::string is declared in <string> header  
    j = 0; // error! No type specified.  
    auto k = 0; // OK. type inferred as int by compiler.  
}  
  
int f(int i)  
{  
    return i + 42;  
}  
  
namespace N {  
    class C{/*...*/};  
}
```

줄 5에서 `main` 함수가 선언됩니다. 줄 7에서 `pi`(으)로 이름이 지정된 `const` 변수가 선언되고 초기화되었습니다. 줄 8에서 정수 `i`이(가) `f` 함수에서 생성된 값으로 선언되고 초기화됩니다. `f` 이름은 줄 3의 전달 선언으로 인해 컴파일러에 표시됩니다.

줄 9에서는 `c` 형식의 `obj` 변수가 선언됩니다. 그러나 이 선언은 `c`이(가) 프로그램의 뒷 부분까지 선언되지 않고 전달 선언되지 않으므로 오류가 발생합니다. 오류를 해결하려면 `main` 전에 `c`의 전체 정의를 이동하거나 전달 선언을 추가할 수 있습니다. 이 동작은 C#과 같은 다른 언어와는 다릅니다. 이러한 언어에서는 원본 파일의 선언 지점 이전에 함수 및 클래스를 사용할 수 있습니다.

줄 10에서는 `std::string` 형식의 `str` 변수가 선언됩니다. `std::string` 이름은 줄 1의 원본 파일에 병합되는 `string` 헤더 파일에 도입되었기 때문에 표시됩니다. `std` 은(는) `string` 클래스가 선언되는 네임스페이스입니다.

줄 11에서 `j`(이)라는 이름이 선언되지 않았기 때문에 오류가 발생합니다. 선언은 JavaScript와 같은 다른 언어와 달리 유형을 제공해야 합니다. 줄 12에서는 `auto` 키워드가 사용되며, 이 키워드는 초기화된 값에 따라 `k` 형식을 유추하도록 컴파일러에 지시합니다. 이 경우 컴파일러는 형식으로 `int` 을(를) 선택합니다.

선언 범위

선언에 의해 도입된 이름은 선언이 발생하는 범위 내에서 유효합니다. 이전 예제에서 `main` 함수 내에 선언된 변수는 지역 변수입니다. 주 외부의 전역 범위에서 `i`(이)라는 다른 변수를 선언할 수 있으며, 이 변수는 별도의 엔티티가 됩니다. 그러나 이러한 이름 중복은 프로그래머에게 혼란과 오류를 초래할 수 있으므로 피해야 합니다. 줄 21에서 `c` 클래스는 네임스페이스 `N`의 범위에 선언됩니다. 네임스페이스를 사용하면 이를 충돌을 방지 할 수 있습니다. 대부분의 C++ 표준 라이브러리 이름은 `std` 네임스페이스 내에 선언됩니다. 범위 규칙이 선언에 사용되는 방법에 대한 자세한 내용은 [범위](#)를 참조하세요.

정의

함수, 클래스, 열거형 및 상수 변수를 포함한 일부 엔터티는 정의되고 선언되어야 합니다. 정의는 나중에 프로그램에서 엔티티가 사용될 때 컴파일러에 머신 코드를 생성하는 데 필요한 모든 정보를 제공합니다. 이전 예제에서 줄 3에는 함수 `f`에 대한 선언이 포함되어 있지만 함수에 대한 정의는 줄 15부터 줄 18까지 제공됩니다. 줄 21에서는 클래스 `c`이 (가) 선언되고 정의됩니다(정의된대로 클래스는 아무 작업도 수행하지 않음). 상수 변수는 정의, 즉 값이 선언된 곳과 동일한 문에 값을 할당해야 합니다. 컴파일러가 할당할 공간을 알고 있기 때문에 `int`과(와) 같은 기본 제공 형식의 선언은 자동으로 정의가 됩니다.

다음 예제에서는 정의이기도 한 선언을 보여 줍니다.

C++

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
```

```
public:  
    Boolean IsChecked();  
    virtual int     ChangeState() = 0;  
};
```

정의가 아닌 몇 가지 선언은 다음과 같습니다.

C++

```
extern int i;  
char *strchr( const char *Str, const char Target );
```

TypeDefs 및 using 문

이전 버전의 C++에서 `typedef` 키워드는 다른 이름의 별칭인 새 이름을 선언하는 데 사용됩니다. 예를 들어 `std::string` 형식은 `std::basic_string<char>`의 다른 이름입니다. 프로그래머가 실제 이름이 아닌 `typedef` 이름을 사용하는 이유는 분명합니다. 최신 C++에서는 `typedef` 키워드보다 `using` 키워드가 선호되지만, 이미 선언되고 정의된 엔티티에 대해 새 이름을 선언한다는 개념은 동일합니다.

정적 클래스 멤버

정적 클래스 데이터 멤버는 클래스의 모든 개체에서 공유하는 불연속 변수입니다. 공유되므로 클래스 정의 외부에서 정의하고 초기화해야 합니다. 자세한 내용은 [클래스](#)를 참조하세요.

extern 선언

C++ 프로그램은 둘 이상의 [컴파일 단위](#)를 포함할 수 있습니다. 별도의 컴파일 단위에 정의된 엔티티를 선언하려면 `extern` 키워드를 사용하세요. 선언의 정보는 컴파일러에 충분합니다. 그러나 연결 단계에서 엔티티의 정의를 찾을 수 없는 경우 링커는 오류를 발생시킵니다.

이 섹션의 내용

[스토리지 클래스](#)

`const`

`constexpr`

`extern`

`Initializers`

별칭 및 `typedef`

`using` 선언

`volatile`

`decltype`

C++의 특성

참고 항목

[기본 개념](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

스토리지 클래스

아티클 • 2024. 11. 21.

C++ 변수 선언 컨텍스트의 스토리지 클래스는 개체의 수명, 연결 및 메모리 위치를 제어하는 형식 지정자입니다. 주어진 개체에는 스토리지 클래스가 하나만 있을 수 있습니다. 블록 내에 정의된 변수에는, 또는 `thread_local` 지정자를 사용하여 지정하지 않는 한 자동 스토리지가 `extern static` 있습니다. 자동 개체 및 변수에는 연결이 없습니다. 블록 외부의 코드에는 표시되지 않습니다. 실행이 블록에 들어갈 때 메모리가 자동으로 할당되고 블록이 종료되면 할당이 해제됩니다.

주의

- 키워드는 `mutable` 스토리지 클래스 지정자로 간주될 수 있습니다. 그러나 클래스 정의의 멤버 목록에서만 사용할 수 있습니다.
- **Visual Studio 2010 이상:** 키워드는 `auto` 더 이상 C++ 스토리지 클래스 지정자가 아니고 `register` 키워드는 더 이상 사용되지 않습니다. **Visual Studio 2017 버전 15.7 이상:** (모드 이상에서 `/std:c++17` 사용 가능): `register` 키워드가 C++ 언어에서 제거됩니다. 이 기능을 사용하면 진단 메시지가 발생합니다.

C++

```
// c5033.cpp
// compile by using: cl /c /std:c++17 c5033.cpp
register int value; // warning C5033: 'register' is no longer a
                     supported storage class
```

static

키워드를 `static` 사용하여 전역 범위, 네임스페이스 범위 및 클래스 범위에서 변수 및 함수를 선언할 수 있습니다. 정적 변수는 로컬 범위에서 선언할 수도 있습니다.

정적 생존 기간이란 프로그램이 시작될 때 개체 또는 변수가 할당되고 프로그램이 끝날 때 개체 또는 변수가 할당 취소됨을 의미합니다. 외부 연결은 변수의 이름이 변수가 선언된 파일 외부에서 볼 수 있음을 의미합니다. 반대로 내부 링크는 변수가 선언된 파일 외부에 이름이 표시되지 않음을 의미합니다. 기본적으로 전역 네임스페이스에서 정의된 개체 또는 변수에는 정적 지속 기간 및 외부 링크가 있습니다. 키워드는 `static` 다음과 같은 경우에 사용할 수 있습니다.

- 파일 범위(전역 및/또는 네임스페이스 범위) `static`에서 변수 또는 함수를 선언하는 경우 키워드는 변수 또는 함수에 내부 링크가 있음을 지정합니다. 변수를 선언할 때 변수가 정적 생존 기간을 가지며, 다른 값을 지정하지 않으면 컴파일러가 0으로 초기화합니다.
- 함수에서 변수를 선언할 때 키워드는 해당 함수 `static`에 대한 호출 간에 변수의 상태를 유지하도록 지정합니다.
- 클래스 선언에서 데이터 멤버를 선언 `static` 할 때 키워드는 클래스의 모든 인스턴스에서 하나의 멤버 복사본을 공유한다고 지정합니다. 데이터 멤버는 `static` 파일 범위에서 정의해야 합니다. 이니셜라이저를 가질 수 있는 것으로 `const static` 선언하는 정수 데이터 멤버입니다.
- 클래스 선언에서 멤버 함수를 선언 `static` 할 때 키워드는 함수가 클래스의 모든 인스턴스에서 공유되도록 지정합니다. 함수에 `static` 암시적 `this` 포인터가 없으므로 멤버 함수는 인스턴스 멤버에 액세스할 수 없습니다. 인스턴스 멤버에 액세스하려면 인스턴스 포인터 또는 참조인 매개 변수를 사용하여 함수를 선언합니다.
- 의 멤버 `union`를 선언 `static`할 수 없습니다. 그러나 전역적으로 선언된 익명 `union`을 명시적으로 선언 `static`해야 합니다.

이 예제에서는 함수에 선언된 `static` 변수가 해당 함수에 대한 호출 간에 상태를 유지하는 방법을 보여 줍니다.

```
C++

// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
    static int nStatic;      // Value of nStatic is retained
                            // between each function call
    nStatic += curr;
    cout << "nStatic is " << nStatic << endl;
}

int main() {
    for ( int i = 0; i < 5; i++ )
        showstat( i );
}
```

Output

```
nStatic is 0  
nStatic is 1  
nStatic is 3  
nStatic is 6  
nStatic is 10
```

이 예제에서는 클래스에서의 `static` 사용을 보여줍니다.

C++

```
// static2.cpp  
// compile with: /EHsc  
#include <iostream>  
  
using namespace std;  
class CMyClass {  
public:  
    static int m_i;  
};  
  
int CMyClass::m_i = 0;  
CMyClass myObject1;  
CMyClass myObject2;  
  
int main() {  
    cout << myObject1.m_i << endl;  
    cout << myObject2.m_i << endl;  
  
    myObject1.m_i = 1;  
    cout << myObject1.m_i << endl;  
    cout << myObject2.m_i << endl;  
  
    myObject2.m_i = 2;  
    cout << myObject1.m_i << endl;  
    cout << myObject2.m_i << endl;  
  
    CMyClass::m_i = 3;  
    cout << myObject1.m_i << endl;  
    cout << myObject2.m_i << endl;  
}
```

Output

```
0  
0  
1  
1  
2  
2
```

다음 예제에서는 멤버 함수에 선언된 `static` 지역 변수를 보여줍니다. 변수는 `static` 전체 프로그램에서 사용할 수 있습니다. 형식의 모든 인스턴스는 변수의 동일한 복사본을 `static` 공유합니다.

C++

```
// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
    void Test(int value) {
        static int var = 0;
        if (var == value)
            cout << "var == value" << endl;
        else
            cout << "var != value" << endl;

        var = value;
    }
};

int main() {
    C c1;
    C c2;
    c1.Test(100);
    c2.Test(100);
}
```

Output

```
var != value
var == value
```

C++11 `static` 부터는 로컬 변수 초기화가 스레드로부터 안전합니다. 이 기능을 매직 정적이라고도 합니다. 그러나 다중 스레드 애플리케이션에서는 모든 후속 할당을 동기화해야 합니다. CRT에 대한 종속성을 사용하지 않도록 플래그를 `/Zc:threadSafeInit-` 사용하여 스레드로부터 안전한 정적 초기화 기능을 사용하지 않도록 설정할 수 있습니다.

extern

선언된 `extern` 개체 및 변수는 다른 변환 단위 또는 묶은 범위에 외부 링크가 있는 것으로 정의된 개체를 선언합니다. 자세한 내용은 번역 단위 및 [연결을 참조](#) `extern` 하세요.

thread_local (C++11)

지정자를 사용하여 `thread_local` 선언된 변수는 생성된 스레드에서만 액세스할 수 있습니다. 이 변수는 스레드를 만들 때 만들어지고 스레드가 제거될 때 제거됩니다. 각 스레드에 변수의 자체 복사본이 있습니다. Windows에서는 `thread_local` Microsoft 특정 `_declspec(thread)` 특성과 기능적으로 동일합니다.

C++

```
thread_local float f = 42.0; // Global namespace. Not implicitly static.

struct S // cannot be applied to type definition
{
    thread_local int i; // Illegal. The member must be static.
    thread_local static char buf[10]; // OK
};

void DoSomething()
{
    // Apply thread_local to a local variable.
    // Implicitly "thread_local static S my_struct".
    thread_local S my_struct;
}
```

지정자에 대해 유의 `thread_local` 해야 할 사항:

- DLL에서 동적으로 초기화된 스레드-로컬 변수는 모든 호출 스레드에서 올바르게 초기화되지 않을 수 있습니다. 자세한 내용은 [thread](#)를 참조하세요.
- `thread_local` 지정자를 사용하거나 `extern` 와 결합 `static` 할 수 있습니다.
- 데이터 선언 및 정의 `thread_local` 에만 적용 `thread_local` 할 수 있으며 함수 선언 또는 정의에는 사용할 수 없습니다.
- 전역 데이터 객체(및 `static`), 로컬 정적 객체 및 `extern` 클래스의 정적 데이터 멤버를 포함하는 정적 스토리지 기간이 있는 데이터 항목에서만 지정할 `thread_local` 수 있습니다. 선언된 `thread_local` 모든 지역 변수는 다른 스토리지 클래스가 제공되지 않은 경우 암시적으로 정적입니다. 즉, 블록 범위 `thread_local`에서 해당합니다 `thread_local static`.
- 스레드 로컬 객체의 선언과 정의가 같은 파일에서 발생하는지, 아니면 별도의 파일에서 발생하는지와 관계없이 해당 선언과 정의 둘 다에 대해 `thread_local`을 지정해야 합니다.

- 와 함께 `std::launch::async` 변수를 사용하지 `thread_local` 않는 것이 좋습니다. 자세한 내용은 함수를 참조 <[future](#)> 하세요.

Windows에서는 `thread_local` * 형식 정의에 `_declspec(thread)` 적용할 수 있으며 C 코드에서 유효하다 `*__declspec(thread)`는 점을 제외하면 기능적으로 동일합니다. 가능하면 C++ 표준의 일부이므로 더 이식성이 높기 때문에 사용 `thread_local` 하세요.

register

Visual Studio 2017 버전 15.3 이상 (모드 이상에서 `/std:c++17` 사용 가능): 키워드는 `register` 더 이상 지원되는 스토리지 클래스가 아닙니다. 해당 용도로 인해 진단이 발생합니다. 키워드는 나중에 사용할 수 있는 표준에 계속 예약되어 있습니다.

C++

```
register int val; // warning C5033: 'register' is no longer a supported
storage class
```

예: 자동 및 정적 초기화

로컬 자동 개체나 변수는 컨트롤의 흐름이 정의에 도달할 때마다 초기화됩니다. 로컬 정적 개체 또는 변수는 컨트롤의 흐름이 정의에 처음 도달할 때 초기화됩니다.

다음 예제에서는 개체의 초기화 및 개체의 초기화와 소멸을 기록한 후 `I1`, `I2` 및 `I3` 객체를 정의합니다.

C++

```
// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
```

```

InitDemo::InitDemo( const char *szWhat ) :
    szObjName(NULL), sizeofObjName(0) {
    if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
        // Allocate storage for szObjName, then copy
        // initializer szWhat into szObjName, using
        // secured CRT functions.
        sizeofObjName = strlen( szWhat ) + 1;

        szObjName = new char[ sizeofObjName ];
        strcpy_s( szObjName, sizeofObjName, szWhat );

        cout << "Initializing: " << szObjName << "\n";
    }
    else {
        szObjName = 0;
    }
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
    if( szObjName != 0 ) {
        cout << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" );
    cout << "In block.\n";
    InitDemo I2( "Auto I2" );
    static InitDemo I3( "Static I3" );
}
cout << "Exited block.\n";
}

```

Output

```

Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3

```

이 예제에서는 개체 `I1` `I2` 가 초기화되는 방법과 시기 및 `I3` 제거 시기를 보여 줍니다.

프로그램에 대해 주의해야 할 몇 가지 사항이 있습니다.

- 먼저 제어 `I1` `I2` 흐름이 정의된 블록을 종료하면 자동으로 제거됩니다.

- 둘째, C++에서는 블록의 시작 부분에 개체 또는 변수를 선언할 필요가 없습니다. 또한 제어 흐름이 정의에 도달해야 이 개체가 초기화됩니다. (`I2` 및 `I3` 이러한 정의의 예입니다.) 출력은 초기화되는 시기를 정확하게 표시합니다.
- 마지막으로, 프로그램이 실행되는 동안 해당 값을 유지하지만 프로그램이 종료되면 소멸되는 정 `I3` 적 지역 변수입니다.

참고 항목

[선언 및 정의](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

alignas (C++)

아티클 • 2023. 12. 22.

alignas 지정자는 메모리에 있는 형식 또는 개체의 맞춤을 변경합니다.

구문

C++

```
alignas(expression)
alignas(type-id)
alignas(pack...)
```

설명

,, **union** **class** 또는 변수 선언에서 **struct** 지정자를 사용할 **alignas** 수 있습니다.

식 **alignas(expression)** 은 0 또는 2의 힘(1, 2, 4, 8, 16, ...)인 정수 상수 식이어야 합니다.
다른 모든 식의 형식이 잘못되었습니다.

코드 이식성을 위해 대신 **_declspec(align(#))** 사용합니다 **alignas**.

일반적인 용도 **alignas** 는 다음 예제와 같이 사용자 정의 형식의 맞춤을 제어하는 것입니다.

C++

```
struct alignas(8) S1
{
    int x;
};

static_assert(alignof(S1) == 8, "alignof(S1) should be 8");
```

동일한 선언에 여러 **alignas** 항목이 적용되면 값이 가장 큰 선언이 사용됩니다. **alignas** 값 **0** 은 무시됩니다.

다음 예제에서는 사용자 정의 형식으로 사용하는 **alignas** 방법을 보여줍니다.

C++

```

class alignas(4) alignas(16) C1 {};

// `alignas(0)` ignored
union alignas(0) U1
{
    int i;
    float f;
};

union U2
{
    int i;
    float f;
};

static_assert(alignof(C1) == 16, "alignof(C1) should be 16");
static_assert(alignof(U1) == alignof(U2), "alignof(U1) should be equivalent
to alignof(U2)");

```

형식을 맞춤 값으로 제공할 수 있습니다. 형식의 기본 맞춤은 다음 예제와 같이 맞춤 값으로 사용됩니다.

C++

```

struct alignas(double) S2
{
    int x;
};

static_assert(alignof(S2) == alignof(double), "alignof(S2) should be
equivalent to alignof(double)");

```

맞춤 값에 템플릿 매개 변수 팩(`alignas (pack...)`)을 사용할 수 있습니다. 팩에 있는 모든 요소의 가장 큰 맞춤 값이 사용됩니다.

C++

```

template <typename... Ts>
class alignas(Ts...) C2
{
    char c;
};

static_assert(alignof(C2<>) == 1, "alignof(C2<>) should be 1");
static_assert(alignof(C2<short, int>) == 4, "alignof(C2<short, int>) should
be 4");
static_assert(alignof(C2<int, float, double>) == 8, "alignof(C2<int, float,
double>) should be 8");

```

여러 `개 적용된 경우 결과 맞춤은 모든 값 중에서 가장 크며 적용된 형식의 자연 맞춤보다 작을 수 없습니다.`

사용자 정의 형식의 선언 및 정의에는 동일한 맞춤 값이 있어야 합니다.

C++

```
// Declaration of `C3`
class alignas(16) C3;

// Definition of `C3` with differing alignment value
class alignas(32) C3 {}; // Error: C2023 'C3': Alignment (32) different from
prior declaration (16)

int main()
{
    alignas(2) int x; // ill-formed because the natural alignment of int is
4
}
```

참고 항목

[#pragma pack](#)

[맞춤](#)

[alignof](#)

[컴파일러 오류 C2023](#)

[컴파일러 경고 C4359](#)

auto (C++)

아티클 • 2024. 07. 15.

초기화 식에서 선언된 변수의 형식을 추론합니다.

① 참고

C++ 표준에는 이 키워드의 원래 의미와 수정된 의미가 정의되어 있습니다. Visual Studio 2010 전까지 `auto` 키워드에서는 자동 스토리지 클래스에 있는 변수, 즉 현지 수명이 있는 변수를 선언합니다. Visual Studio 2010부터 `auto` 키워드에서는 선언의 초기화 식에서 형식이 추론되는 변수를 선언합니다. `/Zc:auto[-]` 컴파일러 옵션은 `auto` 키워드의 의미를 제어합니다.

구문

`auto declarator initializer ;`

`[](auto param1 , auto param2) {};`

설명

`auto` 키워드는 선언된 변수 또는 람다 식 매개 변수의 초기화 식을 사용하여 해당 형식을 추론하도록 컴파일러에 지시합니다.

`auto` 키워드는 다음과 같은 이점을 제공하므로 꼭 변환이 필요한 경우가 아니라면 대부분의 경우 이 키워드를 사용하는 것이 좋습니다.

- 강력함:** 함수의 반환 형식이 변경할 때 포함되므로 식의 형식이 변경되는 경우에도 작동합니다.
- 성능:** 변환이 없음을 보증합니다.
- 유용성:** 형식 이름 맞춤법 문제 및 오타를 걱정할 필요가 없습니다.
- 효율성:** 코딩의 효율성이 향상됩니다.

`auto`을 사용하지 않는 변환의 경우:

- 특정 형식을 원한다면, 다른 항목은 사용할 수 없습니다.

- 언어 식 템플릿 도우미 유형 - (예: `(valarray+valarray)`).

`auto` 키워드를 사용하려면 형식 대신 이를 사용하여 변수를 선언하고 초기화 식을 지정합니다. 또한 지정자 및 `const`, `volatile`, 포인터(*), 참조(&) 및 rvalue 참조(&&)를 사용하여 `auto` 키워드를 수정할 수 있습니다. 컴파일러는 초기화 식을 계산하고 해당 정보를 사용하여 변수의 형식을 추론합니다.

`auto` 초기화 식은 다음과 같이 여러 가지 형식을 취할 수 있습니다.

- 유니버설 초기화 구문(예: `auto a { 42 };`).
- 대입 구문(예: `auto b = 0;`).
- 유니버설 할당 구문(예: `auto c = { 3.14159 };`, 두 개의 이전 형식을 결합함).
- 직접 초기화 또는 생성자 스타일 구문(예: `auto d(1.41421f);`).

자세한 내용은 [이니셜라이저](#) 및 이 문서의 뒷부분에 있는 코드 예제를 참조하세요.

`auto`이 범위 기반 `for` 문에서 루프 매개 변수를 선언하는 데 사용되는 경우, 다른 초기화 구문을 사용합니다(예: `for (auto& i : iterable) do_action(i);`). 자세한 내용은 [범위 기반 for 문\(C++\)](#)을 참조하세요.

`auto` 키워드는 형식 그 자체가 아니라 형식을 위한 자리 표시자입니다. 따라서 `auto` 키워드는 `sizeof` 및 (C++/CLI용) `typeid`과 같은 캐스트 또는 연산자에서 사용할 수 없습니다.

유용성

`auto` 키워드는 복잡한 형식의 변수를 선언하는 간단한 방법입니다. 예를 들어 `auto`을 사용하여 초기화 식이 템플릿, 함수에 대한 포인터 또는 멤버에 대한 포인터와 관련된 변수를 선언할 수 있습니다.

`auto`을 사용하여 변수를 람다 식으로 선언 및 초기화할 수 있습니다. 람다 식의 형식은 컴파일러에만 알려져 있기 때문에 직접 변수 유형을 선언할 수 없습니다. 자세한 내용은 [람다 식의 예제](#)를 참조하세요.

후행 반환 형식

`auto`을 `decltype` 형식 지정자와 함께 사용하여 템플릿 라이브러리를 작성할 수 있습니다. 반환 형식이 템플릿 인수 형식에 따라 달라지는 함수 템플릿을 선언하려면 `auto` 및 `decltype`를 사용합니다. 또는 다른 함수에 대한 호출을 래핑한 후 다른 함수의 반환 형식을 반환하는 템플릿 함수를 선언하려면 `auto` 및 `decltype`를 사용합니다. 자세한 내용은 [decltype](#)를 참조하세요.

참조 및 cv 한정자

`auto` 드롭 참조, `const` 한정자 및 `volatile` 한정자를 사용하세요. 다음 예제를 참조하세요.

C++

```
// cl.exe /analyze /EHsc /W4
#include <iostream>

using namespace std;

int main( )
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

이전 예제에서 `myAuto`는 `int`이지 `int` 참조가 아니므로 출력은 11 11이며, `auto`로 인해 참조 한정자가 드롭되었을 경우처럼 11 12가 아닙니다.

중괄호로 묶인 초기화자를 사용한 형식 추론(C++14)

다음 코드 예제에서는 중괄호를 사용하여 `auto` 변수를 초기화하는 방법을 보여줍니다. B와 C의 차이점과 A와 E의 차이점을 잘 살펴보세요.

C++

```
#include <initializer_list>

int main()
{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
```

```

auto C{ 4 };

// C3535: cannot deduce type for 'auto' from initializer list'
auto D = { 5, 6.7 };

// C3518 in a direct-list-initialization context the type for 'auto'
// can only be deduced from a single initializer expression
auto E{ 8, 9 };

return 0;
}

```

제한 사항 및 오류 메시지

다음 표에서는 `auto` 키워드 사용에 대한 제한 사항 및 컴파일러에서 내보내는 해당 진단 오류 메시지를 보여줍니다.

[] 테이블 확장

오류 번호	설명
C3530	<code>auto</code> 키워드는 다른 형식 지정자와 함께 사용할 수 없습니다.
C3531	<code>auto</code> 키워드를 사용하여 선언된 기호에는 이니셜라이저가 있어야 합니다.
C3532	<code>auto</code> 키워드를 잘못 사용하여 형식을 선언했습니다. 예를 들어 메서드 반환 형식 또는 배열을 선언했습니다.
C3533, C3539	매개 변수 또는 템플릿 인수는 <code>auto</code> 키워드를 사용하여 선언할 수 없습니다.
C3535	메서드 또는 템플릿 매개 변수는 <code>auto</code> 키워드를 사용하여 선언할 수 없습니다.
C3536	초기화되기 전에는 기호를 사용할 수 없습니다. 실제로는 변수를 사용하여 자신을 초기화할 수 없음을 의미합니다.
C3537	<code>auto</code> 키워드를 사용하여 선언된 형식으로 캐스팅할 수 없습니다.
C3538	<code>auto</code> 키워드를 사용하여 선언된 선언자 목록의 모든 기호는 동일한 형식으로 확인되어야 합니다. 자세한 내용은 선언 및 정의 를 참조하세요.
C3540, C3541	<code>auto</code> 키워드를 사용하여 선언된 기호에 <code>sizeof</code> 및 <code>typeid</code> 연산자를 적용할 수 없습니다.

예제

이러한 코드 조각은 `auto` 키워드를 사용할 수 있는 몇 가지 방법을 보여줍니다.

다음 선언은 동일합니다. 첫 번째 문에서는 `j` 변수가 `int` 형식으로 선언되었습니다. 두 번째 문에는 초기화 식(0)이 정수이므로 `k` 변수가 `int` 형식으로 추론되었습니다.

C++

```
int j = 0; // Variable j is explicitly type int.  
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

다음 선언은 동일하지만 두 번째 선언이 첫 번째 선언보다 간단합니다. `auto` 키워드를 사용하는 가장 중요한 이유 중 하나는 단순성입니다.

C++

```
map<int,list<string>>::iterator i = m.begin();  
auto i = m.begin();
```

다음 코드 조각은 `for` 및 범위 `for` 루프가 시작될 때 `iter` 및 `elem` 형식의 변수를 선언합니다.

C++

```
// cl /EHsc /nologo /W4  
#include <deque>  
using namespace std;  
  
int main()  
{  
    deque<double> dqDoubleData(10, 0.1);  
  
    for (auto iter = dqDoubleData.begin(); iter != dqDoubleData.end();  
        ++iter)  
    { /* ... */ }  
  
    // prefer range-for loops with the following information in mind  
    // (this applies to any range-for with auto, not just deque)  
  
    for (auto elem : dqDoubleData) // COPIES elements, not much better than  
        the previous examples  
    { /* ... */ }  
  
    for (auto& elem : dqDoubleData) // observes and/or modifies elements IN-  
    PLACE  
    { /* ... */ }  
  
    for (const auto& elem : dqDoubleData) // observes elements IN-PLACE  
    { /* ... */ }  
}
```

다음 코드 조각에서는 `new` 연산자 및 포인터 선언을 사용하여 포인터를 선언합니다.

C++

```
double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);
```

다음 코드 조각은 각 선언문에서 여러 기호를 선언합니다. 각 명령문의 모든 기호는 동일한 형식으로 확인됩니다.

C++

```
auto x = 1, *y = &x, **z = &y; // Resolves to int.
auto a(2.01), *b (&a);           // Resolves to double.
auto c = 'a', *d(&c);           // Resolves to char.
auto m = 1, &n = m;              // Resolves to int.
```

다음 코드 조각은 조건부 연산자(`?:`)를 사용하여 변수 `x`를 정수 값을 갖는 정수로 선언합니다.

C++

```
int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;
```

다음 코드 조각에서는 `x` 변수를 `int` 형식으로 초기화하고, `y` 변수를 `const int` 형식에 대한 참조로 초기화하고, `fp` 변수를 `int` 형식을 반환하는 함수에 대한 포인터로 초기화합니다.

C++

```
int f(int x) { return x; }
int main()
{
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x);
    p = f;
    auto fp = p;
    //...
}
```

참고 항목

키워드

/Zc:auto(Deduce 변수 형식)

sizeof 연산자

typeid

operator new

선언 및 정의

람다 식의 예제

Initializers

decltype

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

const (C++)

아티클 • 2024. 11. 21.

데이터 선언을 수정할 때 `const` 키워드는 개체나 변수를 수정할 수 없음을 지정합니다.

구문

```
:  
ptr-declarator  
noptr-declarator parameters-and-qualifiers trailing-return-type  
:  
noptr-declarator  
ptr-operator ptr-declarator  
:  
declarator-id attribute-specifier-seqopt  
noptr-declarator parameters-and-qualifiers  
noptr-declarator [ constant-expressionopt ] attribute-specifier-seqopt  
( ptr-declarator )  
:  
( parameter-declaration-clause ) cv-qualifier-seqopt  
ref-qualifieropt noexcept-specifieropt attribute-specifier-seqopt  
:  
-> type-id  
:  
* attribute-specifier-seqopt cv-qualifier-seqopt  
& attribute-specifier-seqopt  
&& attribute-specifier-seqopt  
nested-name-specifier * attribute-specifier-seqopt cv-qualifier-seqopt  
:  
cv-qualifier cv-qualifier-seqopt  
:  
const  
volatile  
:  
&  
&&
```

```
:  
... opt id-expression
```

const 값

const 키워드는 변수의 값이 상수임을 지정하고 프로그래머가 이 변수를 수정하지 못하게 하도록 컴파일러에 지시합니다.

C++

```
// constant_values1.cpp  
int main() {  
    const int i = 5;  
    i = 10;    // C3892  
    i++;     // C2105  
}
```

C++에서는 **#define** 전처리기 지시문 대신 **const** 키워드를 사용하여 상수 값을 정의할 수 있습니다. **const**로 정의된 값은 형식 검사를 받으며 상수 식 대신 사용할 수 있습니다. C++에서는 다음과 같이 **const** 변수를 사용하여 배열의 크기를 지정할 수 있습니다.

C++

```
// constant_values2.cpp  
// compile with: /c  
const int maxarray = 255;  
char store_char[maxarray]; // allowed in C++; not allowed in C
```

C에서 상수 같은 기본적으로 외부 링크로 설정되므로 소스 파일에만 나타날 수 있습니다. C++에서 상수 같은 기본적으로 내부 링크로 설정되므로 헤더 파일에 나타날 수 있습니다.

const 키워드는 포인터 선언에도 사용할 수 있습니다.

C++

```
// constant_values3.cpp  
int main() {  
    char this_char{'a'}, that_char{'b'};  
    char *mybuf = &this_char, *yourbuf = &that_char;  
    char *const aptr = mybuf;  
    *aptr = 'c';    // OK  
    aptr = yourbuf; // C3892  
}
```

`const`로 선언된 변수에 대한 포인터는 `const`로 선언된 포인터에만 할당될 수 있습니다.

C++

```
// constant_values4.cpp
#include <stdio.h>
int main() {
    const char *mybuf = "test";
    char *yourbuf = "test2";
    printf_s("%s\n", mybuf);

    const char *bptr = mybuf; // Pointer to constant data
    printf_s("%s\n", bptr);

    /* *bptr = 'a'; // Error
}
```

상수 데이터에 대한 포인터를 함수 매개 변수로 사용하여 함수가 포인터를 통해 전달된 매개 변수를 수정하지 못하게 할 수 있습니다.

`const`로 선언된 개체의 경우 상수 멤버 함수만 호출할 수 있습니다. 컴파일러는 상수 개체가 수정되지 않도록 보장합니다.

C++

```
birthday.getMonth(); // Okay
birthday.setMonth( 4 ); // Error
```

상수가 아닌 개체에 대해 상수 또는 상수가 아닌 멤버 함수를 호출할 수 있습니다. `const` 키워드를 사용하여 멤버 함수를 오버로드할 수도 있으며, 이 기능에 따라 상수 및 비상수 개체에 대해 다른 버전의 함수를 호출할 수 있습니다.

`const` 키워드를 사용하여 생성자나 소멸자를 선언할 수 없습니다.

const 멤버 함수

`const` 키워드로 멤버 함수를 선언하면 함수가 자신이 호출되는 개체를 수정하지 않는 "읽기 전용" 함수로 지정됩니다. 상수 멤버 함수는 비정적 데이터 멤버를 수정하거나 상수가 아닌 멤버 함수를 호출할 수 없습니다. 상수 멤버 함수를 선언하려면 인수 목록의 닫는 괄호 뒤에 `const` 키워드를 배치합니다. 선언과 정의 모두에 `const` 키워드가 필요합니다.

C++

```

// constant_member_function.cpp
class Date
{
public:
    Date( int mn, int dy, int yr );
    int getMonth() const;      // A read-only function
    void setMonth( int mn );  // A write function; can't be const
private:
    int month;
};

int Date::getMonth() const
{
    return month;           // Doesn't modify anything
}
void Date::setMonth( int mn )
{
    month = mn;            // Modifies data member
}
int main()
{
    Date MyDate( 7, 4, 1998 );
    const Date BirthDate( 1, 18, 1953 );
    MyDate.setMonth( 4 );   // Okay
    BirthDate.getMonth();   // Okay
    BirthDate.setMonth( 4 ); // C2662 Error
}

```

C와 C++ `const` 차이점

C 소스 코드 파일에서 `const` 변수를 정의하는 경우 다음과 같이 수행합니다.

C

```
const int i = 2;
```

그런 다음 이 변수를 아래와 같이 다른 모듈에서 사용할 수 있습니다.

C

```
extern const int i;
```

그러나 C++에서 동일한 동작을 가져오려면 `const` 변수를 다음과 같이 정의해야 합니다.

C++

```
extern const int i = 2;
```

C와 마찬가지로 이 변수를 다음과 같이 다른 모듈에서 사용할 수 있습니다.

C++

```
extern const int i;
```

C 소스 코드 파일에서 사용하기 위해 C++ 소스 코드 파일에 `extern` 변수를 정의하려면 다음을 사용합니다.

C++

```
extern "C" const int x=10;
```

C++ 컴파일러에 의한 이름 변환을 방지해야 합니다.

설명

멤버 함수의 매개 변수 목록을 따를 때 `const` 키워드는 함수가 자신이 호출되는 개체를 수정하지 않도록 지정합니다.

`const`에 대한 자세한 내용은 다음 문서를 참조하세요.

- [const 및 volatile 포인터](#)
- [형식 한정자\(C 언어 참조\)](#)
- [volatile](#)
- [#define](#)

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

constexpr(C++)

아티클 • 2024. 09. 26.

`constexpr` 키워드는 C++11에서 도입되고 C++14에서 향상되었습니다. 끊임없는 표현을 의미합니다. `const` 와 마찬가지로, 변수에 적용될 수 있습니다: 코드가 값을 수정하려고 할 때 컴파일러 오류가 발생합니다. `const` 와 달리, 함수와 클래스 생성자에도 `constexpr` 을 적용할 수 있습니다. `constexpr` 는 값 또는 반환 값이 상수이며 가능한 경우 컴파일 타임에 계산됨을 나타냅니다.

템플릿 인수 및 배열 선언과 같이 `const` 정수가 필요한 곳마다 `constexpr` 정수 값을 사용할 수 있습니다. 그리고 런타임이 아닌 컴파일 시간에 값이 계산되면 프로그램이 더 빠르게 실행되고 메모리를 덜 사용하는 데 도움이 됩니다.

컴파일 타임 상수 계산의 복잡성과 컴파일 시간에 대한 잠재적 영향을 제한하기 위해 C++14 표준에서는 상수 식의 형식을 [리터럴 형식](#)이어야 합니다.

구문

```
constexpr 리터럴 형식 식별자 = 상수식;  
constexpr 리터럴 형식 식별자 { 상수식 } ;  
constexpr 리터럴 타입 식별자 ( params ) ;  
constexpr ctor ( params ) ;
```

매개 변수

params

하나 이상의 매개 변수로, 각 매개 변수는 리터럴 형식이어야 하며 그 자체가 상수 식이어야 합니다.

반환 값

`constexpr` 변수 또는 함수는 [리터럴 형식](#)을 반환해야 합니다.

constexpr 변수

`const` 와 `constexpr` 변수의 주요 차이점은 `const` 변수의 초기화가 런타임까지 지연될 수 있다는 것입니다. `constexpr` 변수는 컴파일 시간에 초기화되어야 합니다. 모든 `constexpr` 변수는 `const`입니다.

- 변수가 리터럴 형식을 갖고 초기화되면 `constexpr` 을 사용하여 선언할 수 있습니다. If 초기화는 생성자에 의해 수행되며 생성자는 다음과 같이 `constexpr` 로 선언되어야 합니다.
- 이 두 가지 조건이 모두 충족되면 참조를 `constexpr` 로 선언할 수 있습니다. 참조된 객체가 상수 표현식으로 초기화되고 초기화 중에 호출된 모든 암시적 변환도 상수 표현식입니다.
- `constexpr` 변수 또는 함수의 모든 선언에는 `constexpr` 지정자가 있어야 합니다.

C++

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

constexpr 함수

`constexpr` 함수는 코드 사용에 필요할 때 컴파일 시간에 반환 값을 계산할 수 있는 함수입니다. 코드를 사용하려면 컴파일 시간에 `constexpr` 변수를 초기화하거나 비형식 템플릿 인수를 제공하기 위한 반환 값이 필요합니다. 인수가 `constexpr` 값인 경우 `constexpr` 함수는 컴파일 타임 상수를 생성합니다. `constexpr` 이외의 인수를 사용하여 호출하거나 컴파일 시간에 해당 값이 필요하지 않은 경우 일반 함수처럼 런타임에 값을 생성합니다. 이 이중 동작 덕분에 동일한 함수의 `constexpr` 버전과 `constexpr` 이외 버전을 작성하지 않아도 됩니다.

`constexpr` 함수 또는 생성자는 암시적으로 `inline` 합니다.

`constexpr` 함수에는 다음 규칙이 적용됩니다.

- `constexpr` 함수는 리터럴 형식만 사용하고 반환해야 합니다.
- `constexpr` 함수는 재귀적일 수 있습니다.
- C++20 이전에는 `constexpr` 함수를 가상으로 설정할 수 없으며 바깥쪽 클래스에 가상 기본 클래스가 있는 경우와 같이 생성자를 `constexpr` 로 정의할 수 없습니다. C++20 이상에서는 `constexpr` 함수가 가상일 수 있습니다. Visual Studio 2019 버전 16.10 이상 버전은 `/std:c++20` 이상 컴파일러 옵션을 지정할 때 `constexpr` 가상 함수를 지원합니다.

- 본문은 `= default` 또는 `= delete`로 정의할 수 있습니다.
- 본문에는 `goto` 문이나 `try` 블록이 포함될 수 없습니다.
- `constexpr` 이 아닌 템플릿의 명시적 전문화는 `constexpr`로 선언될 수 있습니다.
- `constexpr` 템플릿의 명시적인 전문화는 `constexpr` 일 필요도 없습니다.

Visual Studio 2017 이상의 `constexpr` 함수에는 다음 규칙이 적용됩니다.

- 여기에는 `if` 및 `switch` 문과 `for`, 범위 기반 `for`, `while` 및 `do-while`를 포함한 모든 반복 문이 포함될 수 있습니다.
- 여기에는 지역 변수 선언이 포함될 수 있지만 변수는 초기화되어야 합니다. 리터럴 형식이어야 하며 `static` 또는 스레드 로컬일 수 없습니다. 로컬로 선언된 변수는 `const` 일 필요가 없으며 변경될 수 있습니다.
- `constexpr` 비정적 멤버 함수는 암시적으로 `const` 일 필요는 없습니다.

C++

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

💡 팁

Visual Studio 디버거에서 최적화되지 않은 디버그 빌드를 디버깅할 때 내부에 중단 점을 넣어 `constexpr` 함수가 컴파일 시간에 평가되고 있는지 여부를 알 수 있습니다. If 중단점에 도달하여 런타임에 함수가 호출되었습니다. If 그렇지 않을 경우 함수가 컴파일 타임에 호출되었습니다.

extern constexpr

[/Zc:externConstexpr](#) 컴파일러 옵션을 사용하면 컴파일러가 `extern constexpr`을 사용하여 선언된 변수에 [외부 연결](#)을 적용합니다. 이전 버전의 Visual Studio에서는 기본적으로 또는 [/Zc:externConstexpr](#)-이 지정된 경우 `extern` 키워드가 사용되는 경우에도 Visual Studio에서 `constexpr` 변수에 내부 링크를 적용합니다. [/Zc:externConstexpr](#) 옵션은 Visual Studio 2017 업데이트 15.6부터 사용할 수 있으며 기본적으로 꺼져 있습니다. [/permissive-](#) 옵션은 [/Zc:externConstexpr](#)을 사용하도록 설정하지 않습니다.

예시

다음 예에서는 `constexpr` 변수, 함수 및 사용자 정의 형식을 보여 줍니다. `main()`의 마지막 문에서 `constexpr` 멤버 함수 `GetValue()`는 값이 컴파일 시간에 알려질 필요가 없기 때문에 런타임 호출입니다.

C++

```
// constexpr.cpp
// Compile with: cl /EHsc /W4 constexpr.cpp
#include <iostream>

using namespace std;

// Pass by value
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}

// Pass by reference
constexpr float exp2(const float& x, const int& n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
}

// Compile-time computation of array length
template<typename T, int N>
constexpr int length(const T(&)[N])
{
    return N;
}

// Recursive constexpr function
constexpr int fac(int n)
{
    return n == 1 ? 1 : n * fac(n - 1);
}

// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const
    {
        return _i;
    }
}
```

```
private:
    int _i;
};

int main()
{
    // foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!

    // Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;
}
```

요구 사항

Visual Studio 2015 이상 -

참고 항목

[선언 및 정의](#)

[const](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

extern (C++)

아티클 • 2024. 09. 26.

`extern` 키워드는 전역 변수, 함수 또는 템플릿 선언에 적용될 수 있습니다. 기호에 외부 링크가 있음을 지정합니다. 연결에 대한 백그라운드 정보와 전역 변수 사용이 권장되지 않는 이유는 [변환 단위 및 연결](#)을 참조하세요.

`extern` 키워드는 상황에 따라 네 가지 의미를 갖습니다.

- `const` 가 아닌 전역 변수 선언에서 `extern`은 변수 또는 함수가 다른 변환 단위에 정의되도록 지정합니다. 변수가 정의된 파일을 제외한 모든 파일에 `extern`을 적용해야 합니다.
- `const` 변수 선언에서는 변수에 외부 링크가 있음을 지정합니다. `extern`은 모든 파일의 모든 선언에 적용되어야 합니다. (전역 `const` 변수에는 기본적으로 내부 연결이 있습니다.)
- `extern "C"` 는 함수가 다른 곳에서 정의되고 C 언어 호출 규칙을 사용함을 지정합니다. `extern "C"` 한정자는 블록의 여러 함수 선언에 적용될 수도 있습니다.
- 템플릿 선언에서 `extern`은 템플릿이 이미 다른 곳에서 인스턴스화되었음을 지정합니다. `extern`은 현재 위치에 새 인스턴스를 만드는 대신 다른 인스턴스화를 다시 사용할 수 있음을 컴파일러에 알립니다. `extern` 사용에 대한 자세한 내용은 [명시적 인스턴스화](#)를 참조하세요.

const 가 아닌 전역에 대한 `extern` 연결

링커가 전역 변수 선언 전에 `extern`을 보면 다른 변환 단위에서 정의를 찾습니다. 전역 범위에서 `const` 변수가 아닌 선언은 기본적으로 외부에 있습니다. 정의를 제공하지 않는 선언에만 `extern`을 적용합니다.

C++

```
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
extern int i; // declaration only. same as i in FileA

//fileC.cpp
extern int i; // declaration only. same as i in FileA

//fileD.cpp
```

```
int i = 43; // LNK2005! 'i' already has a definition.  
extern int i = 43; // same error (extern is ignored on definitions)
```

const 전역에 대한 extern 연결

`const` 전역 변수에는 기본적으로 내부 연결이 있습니다. 변수가 외부 링크를 갖도록 하려면 `extern` 키워드를 정의에 적용하고 다른 파일의 다른 모든 선언에 적용합니다.

C++

```
//fileA.cpp  
extern const int i = 42; // extern const definition  
  
//fileB.cpp  
extern const int i; // declaration only. same as i in FileA
```

extern constexpr 링크

Visual Studio 2017 버전 15.3 이하에서는 변수가 `extern`으로 표시된 경우에도 컴파일러가 항상 `constexpr` 변수에 내부 연결을 제공했습니다. Visual Studio 2017 버전 15.5 이상에서는 `/Zc:externConstexpr` 컴파일러 스위치를 사용하여 올바른 표준 준수 동작을 사용하도록 설정합니다. 결국 옵션이 기본값이 됩니다. `/permissive-` 옵션은 `/Zc:externConstexpr`을 사용하도록 설정하지 않습니다.

C++

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

헤더 파일에 `extern constexpr`로 선언된 변수가 포함된 경우 중복 선언이 올바르게 결합되도록 `__declspec(selectany)`로 표시해야 합니다.

C++

```
extern constexpr __declspec(selectany) int x = 10;
```

extern "C" 및 extern "C++" 함수 선언

C++에서 `extern`을 문자열과 함께 사용하면 다른 언어의 링크 규칙이 선언자에 대해 사용 중임을 지정합니다. C 함수 및 데이터는 이전에 C 링크가 있는 것으로 선언된 경우에만 액세스할 수 있습니다. 단, 별도로 컴파일된 변환 단위로 정의되어야 합니다.

Microsoft C++는 *string-literal* 필드에서 문자열 "c" 및 "c++"를 지원합니다. 모든 표준 include 파일은 `extern "C"` 구문을 사용하여 C++ 프로그램에 사용할 런타임 라이브러리 함수를 허용합니다.

예시

다음 예에서는 C 연결이 있는 이름을 선언하는 방법을 보여 줍니다.

```
C++

// Declare printf with C linkage.
extern "C" int printf(const char *fmt, ...);

// Cause everything in the specified
// header files to have C linkage.
extern "C" {
    // add your #include statements here
#include <stdio.h>
}

// Declare the two functions ShowChar
// and GetChar with C linkage.
extern "C" {
    char ShowChar(char ch);
    char GetChar(void);
}

// Define the two functions
// ShowChar and GetChar with C linkage.
extern "C" char ShowChar(char ch) {
    putchar(ch);
    return ch;
}

extern "C" char GetChar(void) {
    char ch;
    ch = getchar();
    return ch;
}

// Declare a global variable, errno, with C linkage.
extern "C" int errno;
```

함수에 둘 이상의 연결 사양이 있는 경우에는 동의해야 합니다. C와 C++ 링크가 모두 있는 함수를 선언하는 것은 오류입니다. 뿐만 아니라 함수에 대한 두 개의 선언(하나는 연결 사양이 있고 다른 하나는 없음)이 프로그램에서 발생하는 경우 연결 사양이 있는 선언이 먼저 이루어져야 합니다. 이미 링크 사양이 있는 함수의 모든 중복 선언에는 첫 번째 선언에서 지정된 링크가 제공됩니다. 예시:

C++

```
extern "C" int CFunc1();
...
int CFunc1();           // Redeclaration is benign; C linkage is
                        // retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                        // CFunc2;  cannot contain linkage
                        // specifier.
```

Visual Studio 2019부터 `/permissive-` 가 지정되면 컴파일러는 `extern "C"` 함수 매개 변수의 선언도 일치하는지 확인합니다. `extern "C"`로 선언된 함수는 오버로드할 수 없습니다. Visual Studio 2019 버전 16.3부터 `/permissive-` 옵션 뒤에 `/Zc:externC-` 컴파일러 옵션을 사용하여 이 검사를 재정의할 수 있습니다.

참고 항목

키워드

변환 단위 및 링크

[extern C의 스토리지 클래스 지정자](#)

[C의 식별자 동작](#)

[C의 연결](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Initializers

아티클 • 2024. 07. 15.

이니셜라이저는 변수의 초기 값을 지정합니다. 변수를 초기화할 수 있는 컨텍스트는 다음과 같습니다.

- 변수 정의

```
C++
```

```
int i = 3;  
Point p1{ 1, 2 };
```

- 함수 매개 변수 중 하나

```
C++
```

```
set_point(Point{ 5, 6 });
```

- 함수의 반환 값으로서 초기화

```
C++
```

```
Point get_new_point(int x, int y) { return { x, y }; }  
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

이니셜라이저가 사용할 수 있는 폼은 다음과 같습니다.

- 괄호 안에 있는 식(또는 쉼표로 구분된 식 목록)

```
C++
```

```
Point p1(1, 2);
```

- 등호 뒤에 식

```
C++
```

```
string s = "hello";
```

- 중괄호로 묶인 이니셜라이저 목록 목록은 다음 예제와 같이 비어 있을 수도 있고, 목록 집합으로 구성될 수도 있습니다.

C++

```
struct Point{
    int x;
    int y;
};

class PointConsumer{
public:
    void set_point(Point p){};
    void set_points(initializer_list<Point> my_list){};
};

int main() {
    PointConsumer pc{};
    pc.set_point({});
    pc.set_point({ 3, 4 });
    pc.set_points({ { 3, 4 }, { 5, 6 } });
}
```

초기화 종류

초기화는 여러 종류가 있으며 프로그램 실행의 여러 지점에서 발생할 수 있습니다. 다양한 종류의 초기화는 상호 배타적이지 않습니다. 예를 들어, 목록 초기화는 값 초기화를 트리거할 수 있고 다른 상황에서는 집합체 초기화를 트리거할 수 있습니다.

0 초기화

0 초기화는 변수를 암시적으로 형식으로 변환된 0으로 설정하는 것입니다.

- 숫자 변수는 0(또는 0.0 또는 0.0000000000 등)으로 초기화됩니다.
- Char 변수는 '\0'로 초기화됩니다.
- 포인터는 `nullptr`로 초기화됩니다.
- 배열, POD 클래스, 구조체 및 공용 구조체는 멤버 값을 0값으로 초기화합니다.

0 초기화는 다음과 같은 다양한 시간에 수행됩니다.

- 프로그램 시작 시 - 정적 지속 시간이 있는 명명된 모든 변수 이러한 변수는 나중에 다시 초기화할 수 있습니다.
- 값을 초기화하는 동안 - 빈 중괄호를 사용하여 초기화된 스칼라 형식 및 POD 클래스 형식
- 멤버의 하위 집합만 초기화된 어레이

0 초기화의 몇 가지 예제는 다음과 같습니다.

C++

```
struct my_struct{
    int i;
    char c;
};

int i0;                  // zero-initialized to 0
int main() {
    static float f1;  // zero-initialized to 0.000000000
    double d{};       // zero-initialized to 0.0000000000000000
    int* ptr{};        // initialized to nullptr
    char s_array[3]{'a', 'b'}; // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 }; // the fourth and fifth ints are
initialized to 0
    my_struct a_struct{}; // i = 0, c = '\0'
}
```

기본값 초기화

기본 생성자를 사용하는 클래스, 구조체 및 공용 구조체에 대한 기본 초기화입니다. 기본 값 생성자는 초기화 식 없이 또는 `new` 키워드를 사용하여 호출할 수 있습니다.

C++

```
MyClass mc1;  
MyClass* mc3 = new MyClass;
```

클래스, 구조체 또는 공용 구조체에 기본 생성자가 있으면 컴파일러가 오류를 내보냅니다.

스칼라 변수는 초기화 식 없이 정의할 경우, 기본값으로 초기화됩니다. 비활성화 상태 값입니다.

C++

```
int i1;  
float f;  
char c;
```

배열은 초기화 식 없이 정의할 경우, 기본값으로 초기화됩니다. 배열이 기본값으로 초기화되면 다음 예제와 같이 멤버가 기본값으로 초기화되고 비활성화 상태 값을 가집니다.

C++

```
int int_arr[3];
```

배열에 기본 생성자가 없는 경우 컴파일러가 오류를 내보냅니다.

상수 변수의 기본 초기화

상수 변수는 이니셜라이저와 함께 선언해야 합니다. 스칼라 형식인 경우, 컴파일러 오류가 발생하며 기본 생성자가 있는 클래스 형식인 경우에는 경고가 발생합니다.

C++

```
class MyClass{};  
int main() {  
    //const int i2;    // compiler error C2734: const object must be  
    initialized if not extern  
    //const char c2;  // same error  
    const MyClass mc1; // compiler error C4269: 'const automatic data  
    initialized with compiler generated default constructor produces unreliable  
    results  
}
```

정적 변수의 기본 초기화

이니셜라이저 없이 선언된 정적 변수는 0으로 초기화됩니다(형식으로 암시적으로 변환).

C++

```
class MyClass {  
private:  
    int m_int;  
    char m_char;  
};  
  
int main() {  
    static int int1;      // 0  
    static char char1;   // '\0'  
    static bool bool1;   // false  
    static MyClass mc1;  // {0, '\0'}  
}
```

전역 정적 개체의 초기화에 대한 자세한 내용은 [주 함수 및 명령줄 인수](#)를 참조하세요.

값 초기화

값 초기화는 다음과 같은 경우에 발생합니다.

- 명명된 값이 빈 중괄호 초기화를 사용하여 초기화됩니다.

- 익명의 임시 개체가 빈 괄호나 중괄호를 사용하여 초기화됩니다.
- 개체가 `new` 키워드와 함께 빈 괄호 또는 중괄호를 사용하여 초기화됩니다.

값 초기화가 수행하는 작업은 다음과 같습니다.

- 하나 이상의 공용 생성자가 있는 클래스의 경우 기본 생성자가 호출됩니다.
- 선언된 생성자가 없는 공용 구조체가 아닌 클래스의 경우, 개체가 0으로 초기화되고 기본값 생성자가 호출됩니다.
- 배열의 경우 모든 요소의 값이 초기화됩니다.
- 다른 모든 경우에는 변수가 0으로 초기화됩니다.

```
C++
```

```
class BaseClass {
private:
    int m_int;
};

int main() {
    BaseClass bc{};      // class is initialized
    BaseClass* bc2 = new BaseClass(); // class is initialized, m_int value
is 0
    int int_arr[3]{}; // value of all members is 0
    int a{};          // value of a is 0
    double b{};        // value of b is 0.0000000000000000
}
```

복사 초기화

복사 초기화는 하나의 개체를 다른 개체로 초기화하는 것입니다. 다음과 같은 경우에 발생합니다.

- 변수가 등호를 사용하여 초기화됩니다.
- 인수가 함수에 전달됩니다.
- 개체가 함수에서 반환됩니다.
- 예외가 발생하거나 catch됩니다.
- 비정적 데이터 멤버가 등호를 사용하여 초기화됩니다.
- 클래스, 구조체 및 공용 구조체 멤버가 집합체 초기화 중에 복사 초기화로 초기화됩니다. 예제는 [집계 초기화](#)를 참조하세요.

다음 코드는 복사 초기화의 몇 가지 예를 보여 줍니다.

C++

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass(int myInt) {}
    void set_int(int myInt) { m_int = myInt; }
    int get_int() const { return m_int; }
private:
    int m_int = 7; // copy initialization of m_int

};

class MyException : public exception{};

int main() {
    int i = 5;           // copy initialization of i
    MyClass mc1{ i };
    MyClass mc2 = mc1;   // copy initialization of mc2 from mc1
    MyClass mc1.set_int(i); // copy initialization of parameter from i
    int i2 = mc2.get_int(); // copy initialization of i2 from return value
                           // of get_int()

    try{
        throw MyException();
    }
    catch (MyException ex){ // copy initialization of ex
        cout << ex.what();
    }
}
```

복사 초기화는 명시적 생성자를 호출할 수 없습니다.

C++

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440:
                     // can't convert from 'int' to 'std::vector<int,std::allocator<_Ty>>'
regex r = "a.*b"; // the constructor is explicit; same error
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same
                                    // error
```

경우에 따라 클래스의 복사 생성자가 삭제되거나 액세스할 수 없는 경우 복사 초기화로 컴파일러 오류가 발생합니다.

직접 초기화

직접 초기화는 (비어 있지 않은) 중괄호 또는 괄호를 사용한 초기화입니다. 복사 초기화와는 달리 명시적 생성자를 호출할 수 있습니다. 다음과 같은 경우에 발생합니다.

- 변수가 비어 있지 않은 중괄호 또는 괄호를 사용하여 초기화됩니다.
- 변수가 `new` 키워드와 비어 있지 않은 중괄호 또는 괄호를 사용하여 초기화됩니다.
- 변수가 `static_cast`을 사용하여 초기화됩니다.
- 생성자에서 기본 클래스 및 비정적 멤버가 이니셜라이저 목록을 사용하여 초기화됩니다.
- 람다 식 내의 캡처된 변수 복사본에서

다음 코드는 직접 초기화의 몇 가지 예를 보여 줍니다.

```
C++  
  
class BaseClass{  
public:  
    BaseClass(int n) :m_int(n){} // m_int is direct initialized  
private:  
    int m_int;  
};  
  
class DerivedClass : public BaseClass{  
public:  
    // BaseClass and m_char are direct initialized  
    DerivedClass(int n, char c) : BaseClass(n), m_char(c) {}  
private:  
    char m_char;  
};  
int main(){  
    BaseClass bc1(5);  
    DerivedClass dc1{ 1, 'c' };  
    BaseClass* bc2 = new BaseClass(7);  
    BaseClass bc3 = static_cast<BaseClass>(dc1);  
  
    int a = 1;  
    function<int()> func = [a](){ return a + 1; }; // a is direct  
initialized  
    int n = func();  
}
```

목록 초기화

목록 초기화는 변수가 중괄호로 묶인 이니셜라이저 목록을 사용하여 초기화될 때 발생합니다. 중괄호로 묶인 이니셜라이저 목록이 사용되는 경우는 다음과 같습니다.

- 변수가 초기화됩니다.
- 클래스가 `new` 키워드를 사용하여 초기화됩니다.
- 개체가 함수에서 반환됩니다.
- 인수가 함수에 전달됩니다.
- 직접 초기화의 인수 중 하나
- 비정적 데이터 멤버 이니셜라이저에서
- 생성자 이니셜라이저 목록에서

다음 코드는 목록 초기화의 몇 가지 예를 보여 줍니다.

C++

```
class MyClass {
public:
    MyClass(int myInt, char myChar) {}
private:
    int m_int[]{ 3 };
    char m_char;
};

class MyClassConsumer{
public:
    void set_class(MyClass c) {}
    MyClass get_class() { return MyClass{ 0, '\0' }; }
};

struct MyStruct{
    int my_int;
    char my_char;
    MyClass my_class;
};

int main() {
    MyClass mc1{ 1, 'a' };
    MyClass* mc2 = new MyClass{ 2, 'b' };
    MyClass mc3 = { 3, 'c' };

    MyClassConsumer mcc;
    mcc.set_class(MyClass{ 3, 'c' });
    mcc.set_class({ 4, 'd' });

    MyStruct ms1{ 1, 'a', { 2, 'b' } };
}
```

집계 초기화

집합체 초기화는 다음과 같은 일종의 배열 또는 클래스 형식(대개 구조체 또는 공용 구조체) 목록 초기화입니다.

- 전용 또는 보호된 멤버 없음
- 명시적으로 기본값으로 설정되었거나 삭제된 생성자를 제외하고 사용자 제공 생성자 없음
- 기본 클래스 없음
- 가상 멤버 함수 없음

① 참고

Visual Studio 2015 및 그 이전에서는 집계에 비정적 멤버에 대한 중괄호 또는 동일한 이니셜라이저를 사용할 수 없습니다. 이 제한은 C++14 표준에서 제거되었으며 Visual Studio 2017에 구현되었습니다.

집합체 이니셜라이저는 다음 예제와 같이 중괄호로 묶은 초기화 목록으로 구성되며, 등호가 있을 수도 있고 없을 수도 있습니다.

C++

```
#include <iostream>
using namespace std;

struct MyAggregate{
    int myInt;
    char myChar;
};

struct MyAggregate2{
    int myInt;
    char myChar = 'Z'; // member-initializer OK in C++14
};

int main() {
    MyAggregate agg1{ 1, 'c' };
    MyAggregate2 agg2{2};
    cout << "agg1: " << agg1.myChar << ":" << agg1.myInt << endl;
    cout << "agg2: " << agg2.myChar << ":" << agg2.myInt << endl;

    int myArr1[] { 1, 2, 3, 4 };
    int myArr2[3] = { 5, 6, 7 };
    int myArr3[5] = { 8, 9, 10 };

    cout << "myArr1: ";
    for (int i : myArr1){
        cout << i << " ";
    }
}
```

```
cout << endl;

cout << "myArr3: ";
for (auto const &i : myArr3) {
    cout << i << " ";
}
cout << endl;
}
```

다음과 같은 출력이 표시됩니다.

Output

```
agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0
```

① 중요

위의 `myArr3`과 같이 집합체 초기화 중에 선언되었지만, 명시적으로 초기화되지 않은 배열 멤버는 0 초기화됩니다.

공용 구조체 및 구조체 초기화

공용 구조체에 생성자가 없는 경우 단일 값을 사용하여(또는 공용 구조체의 다른 인스턴스를 사용하여) 초기화할 수 있습니다. 값을 사용하여 첫 번째 비정적 필드를 초기화합니다. 이는 구조체 초기화와는 다릅니다. 구조체 초기화는 이니셜라이저의 첫 번째 값을 사용하여 첫 번째 필드를 초기화하고, 두 번째 값을 사용하여 두 번째 필드를 초기화하는 식으로 이루어집니다. 다음 예제에서 구조체와 공용 구조체 초기화를 비교해 보세요.

C++

```
struct MyStruct {
    int myInt;
    char myChar;
};

union MyUnion {
    int my_int;
    char my_char;
    bool my_bool;
    MyStruct my_struct;
};

int main() {
    MyUnion mu1{ 'a' }; // my_int = 97, my_char = 'a', my_bool = true,
```

```

{myInt = 97, myChar = '\0'}
    MyUnion mu2{ 1 };      // my_int = 1, my_char = 'x1', my_bool = true,
{myInt = 1, myChar = '\0'}
    MyUnion mu3{};        // my_int = 0, my_char = '\0', my_bool = false,
{myInt = 0, myChar = '\0'}
    MyUnion mu4 = mu3;   // my_int = 0, my_char = '\0', my_bool = false,
{myInt = 0, myChar = '\0'}
    //MyUnion mu5{ 1, 'a', true }; // compiler error: C2078: too many
initializers
    //MyUnion mu6 = 'a';       // compiler error: C2440: cannot convert
from 'char' to 'MyUnion'
    //MyUnion mu7 = 1;        // compiler error: C2440: cannot convert
from 'int' to 'MyUnion'

MyStruct ms1{ 'a' };           // myInt = 97, myChar = '\0'
MyStruct ms2{ 1 };             // myInt = 1, myChar = '\0'
MyStruct ms3{};                // myInt = 0, myChar = '\0'
MyStruct ms4{1, 'a'};          // myInt = 1, myChar = 'a'
MyStruct ms5 = { 2, 'b' };     // myInt = 2, myChar = 'b'
}

```

집합체가 포함된 집합체 초기화

집합체 형식은 배열의 배열, 구조체의 배열 등 다른 집합체 형식을 포함할 수 있습니다. 이러한 형식은 중첩된 중괄호 집합을 사용하여 초기화됩니다. 예:

```
C++

struct MyStruct {
    int myInt;
    char myChar;
};

int main() {
    int intArr1[2][2]{{ 1, 2 }, { 3, 4 }};
    int intArr3[2][2] = {1, 2, 3, 4};
    MyStruct structArr[] { { 1, 'a' }, { 2, 'b' }, { 3, 'c' } };
}
```

참조 초기화

참조 형식의 변수는 참조 형식이 파생된 형식의 개체 또는 참조 형식이 파생된 형식으로 변환될 수 있는 형식의 개체를 사용하여 초기화되어야 합니다. 예시:

```
C++

// initializing_references.cpp
int iVar;
long lVar;
```

```

int main()
{
    long& LongRef1 = lVar;           // No conversion required.
    long& LongRef2 = iVar;          // Error C2440
    const long& LongRef3 = iVar;   // OK
    LongRef1 = 23L;                // Change lVar through a reference.
    LongRef2 = 11L;                // Change iVar through a reference.
    LongRef3 = 11L;                // Error C3892
}

```

임시 개체를 사용하여 참조를 초기화하는 유일한 방법은 상수 임시 개체를 초기화하는 것입니다. 참조 형식 변수는 초기화되면 항상 동일한 개체를 가리키며, 다른 개체를 가리키도록 수정될 수 없습니다.

구문은 동일할 수 있지만 참조 형식 변수의 초기화와 참조 형식 변수에 대한 할당은 의미 체계가 서로 다릅니다. 위의 예제에서 `iVar` 및 `lVar`을 변경하는 할당은 초기화와 유사해 보이지만 결과가 다릅니다. 초기화는 참조 형식 변수가 가리키는 개체를 지정하고, 할당은 참조를 통해 참조된 개체에 할당합니다.

참조 형식의 인수를 함수에 전달하는 것과 함수에서 참조 형식의 값을 반환하는 것은 둘다 초기화이기 때문에 함수에 대한 형식 인수는 참조가 반환됨에 따라 올바르게 초기화됩니다.

참조 형식 변수는 다음에서만 이니셜라이저 없이 선언될 수 있습니다.

- 함수 선언(프로토타입). 예시:

C++

```
int func( int& );
```

- 함수 반환 형식 선언. 예시:

C++

```
int& func( int& );
```

- 참조 형식 클래스 멤버의 선언. 예시:

C++

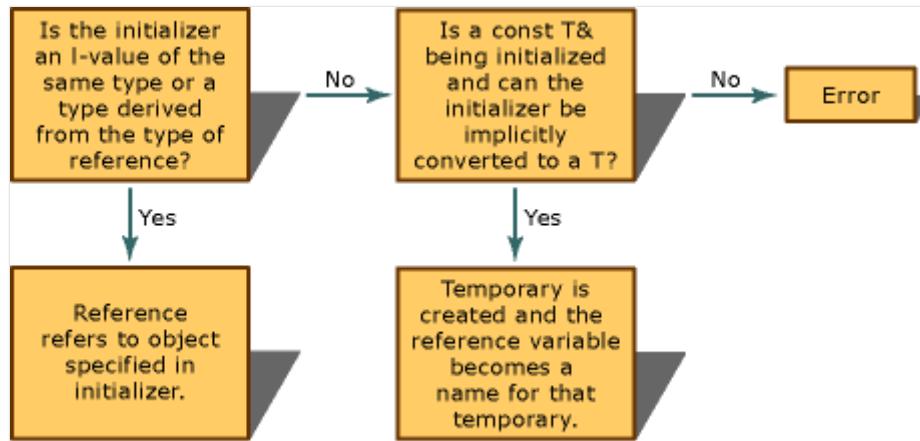
```
class c {public:    int& i;};
```

- `extern`으로 명시적으로 지정된 변수의 선언입니다. 예시:

C++

```
extern int& iVal;
```

참조 형식 변수를 초기화할 때 컴파일러는 다음 그림과 같은 결정 그래프를 사용하여 개체에 대한 참조 만들기 또는 참조가 가리키는 임시 개체 만들기 중에서 하나를 선택합니다.



참조 형식 초기화를 위한 결정 그래프

`volatile` 형식에 대한 참조(`volatile typename& identifier`로 선언됨)는 동일한 형식의 `volatile` 개체 또는 `volatile`으로 선언되지 않은 개체를 사용하여 초기화될 수 있습니다. 그러나 해당 형식의 `const` 개체를 사용하여 초기화될 수는 없습니다. 이와 마찬가지로 `const` 형식에 대한 참조(`const typename& identifier`로 선언됨)는 동일한 형식의 `const` 개체(또는 해당 형식으로 변환되는 임의의 개체나 `const`으로 선언되지 않은 개체)를 사용하여 초기화될 수 있습니다. 그러나 해당 형식의 `volatile` 개체를 사용하여 초기화될 수는 없습니다.

`const` 또는 `volatile` 키워드를 사용하여 한정되지 않은 참조는 `const` 또는 `volatile`로 선언되지 않은 개체를 사용해서만 초기화될 수 있습니다.

외부 변수 초기화

자동, 정적 및 외부 변수의 선언은 이니셜라이저를 포함할 수 있습니다. 하지만 외부 변수 선언은 변수가 `extern`로 선언되지 않은 경우에만 이니셜라이저를 포함할 수 있습니다.

피드백

이 페이지가 도움이 되었나요?

Yes

No

별칭 및 `typedef`(C++)

아티클 • 2023. 10. 12.

별칭 선언을 사용하여 이전에 선언된 형식의 동의어로 사용할 이름을 선언할 수 있습니다. (이 메커니즘은 비공식적으로 형식 별칭이라고도 함). 이 메커니즘을 사용하여 사용자 지정 할당자에 유용할 수 있는 별칭 템플릿을 만들 수도 있습니다.

구문

C++

```
using identifier = type;
```

설명

identifier

별칭 이름입니다.

type

별칭을 만드는 형식 식별자입니다.

별칭은 새 형식을 도입하지 않으며 기존 형식 이름의 의미를 변경할 수 없습니다.

가장 간단한 형태의 별칭은 C++03의 `typedef` 메커니즘과 동일합니다.

C++

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

이러한 두 양식은 모두 형식 `counter`의 변수를 만들 수 있습니다.

`std::ios_base::fmtflags`에 대해서는 이와 같은 형식 별칭이 더 유용합니다.

C++

```
// C++11
using fmtfl = std::ios_base::fmtflags;

// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;
```

```
fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase |
std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

별칭은 함수 포인터에서도 작동하지만 동등한 `typedef`보다 훨씬 더 읽기가 가능합니다.

C++

```
// C++11
using func = void(*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

메커니즘의 `typedef` 제한 사항은 템플릿에서 작동하지 않는다는 것입니다. 그러나 C++ 11의 형식 별칭 구문을 사용하면 별칭 템플릿을 만들 수 있습니다.

C++

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

예시

다음 예제에서는 사용자 지정 할당자(이 경우 정수 벡터 형식)와 별칭 템플릿을 사용하는 방법을 데모로 보여 줍니다. 모든 형식을 `int` 대체하여 편리한 별칭을 만들어 기본 함수 코드에서 복잡한 매개 변수 목록을 숨길 수 있습니다. 코드 전체에서 사용자 지정 할당자를 사용하면 가독성을 개선하고 오타로 인한 버그가 발생할 위험을 줄일 수 있습니다.

C++

```
#include <stdlib.h>
#include <new>

template <typename T> struct MyAlloc {
    typedef T value_type;

    MyAlloc() { }
```

```

template <typename U> MyAlloc(const MyAlloc<U>&) { }

bool operator==(const MyAlloc&) const { return true; }
bool operator!=(const MyAlloc&) const { return false; }

T * allocate(const size_t n) const {
    if (n == 0) {
        return nullptr;
    }

    if (n > static_cast<size_t>(-1) / sizeof(T)) {
        throw std::bad_array_new_length();
    }

    void * const pv = malloc(n * sizeof(T));

    if (!pv) {
        throw std::bad_alloc();
    }

    return static_cast<T *>(pv);
}

void deallocate(T * const p, size_t) const {
    free(p);
}
};

#include <vector>
using MyIntVector = std::vector<int, MyAlloc<int>>;

#include <iostream>

int main ()
{
    MyIntVector foov = { 1701, 1764, 1664 };

    for (auto a: foov) std::cout << a << " ";
    std::cout << "\n";

    return 0;
}

```

Output

1701 1764 1664

Typealias

선언은 `typedef` 해당 범위 내에서 선언의 형식 선언 부분에서 지정된 형식의 동의어가 되는 이름을 소개합니다.

`typedef` 선언을 사용하여 언어로 이미 정의된 형식 또는 선언한 형식에 대해 더 짧거나 더 의미 있는 이름을 생성할 수 있습니다. `typedef` 이름을 사용하면 변경될 수 있는 구현 정보를 캡슐화할 수 있습니다.

선언은 `class`, `struct`, `union` 및 선언 `typedef` 과 `enum` 달리 새 형식을 도입하지 않으며 기존 형식에 대한 새 이름을 도입합니다.

선언 `typedef` 된 이름은 다른 식별자와 동일한 네임스페이스를 차지합니다(문 레이블 제외). 따라서 클래스 형식 선언을 제외하고 이전에 선언된 이름과 동일한 식별자를 사용할 수 없습니다. 다음 예제를 참조하세요.

C++

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                      // C2377: redefined.
```

다른 식별자와 관련된 이름 숨기기 규칙도 사용하여 `typedef` 선언된 이름의 표시 유형을 제어합니다. 따라서 다음 예제는 C++에서 사용할 수 있습니다.

C++

```
// typedef_names2.cpp
typedef unsigned long UL;    // Declare a typedef name, UL
int main()
{
    unsigned int UL;    // Redefinition hides typedef name
}

// typedef UL back in scope
```

이름 숨기기의 또 다른 인스턴스:

C++

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{

void myproc( int )
```

```
{  
    int FlagType;  
}
```

동일한 이름으로 `typedef` 로컬 범위 식별자를 선언하거나 동일한 범위 또는 내부 범위에서 구조체 또는 공용 구조체의 멤버를 선언할 때 형식 지정자를 지정해야 합니다. 예시:

C++

```
typedef char FlagType;  
const FlagType x;
```

식별자, 구조체 멤버 또는 공용 구조체 멤버에 `FlagType` 이름을 다시 사용하려면 형식을 제공해야 합니다.

C++

```
const int FlagType; // Type specifier required
```

말하는 것으로 충분하지 않은 이유는

C++

```
const FlagType; // Incomplete specification
```

다시 `FlagType` 선언되는 식별자가 아니라 형식의 일부로 사용되기 때문입니다. 이 선언은 다음과 유사한 잘못된 선언으로 처리됩니다.

C++

```
int; // Illegal declaration
```

포인터, 함수 및 배열 형식을 비롯한 모든 형식을 `typedef` 를 사용하여 선언할 수 있습니다. 정의의 표시 유형이 선언의 표시 유형과 동일한 경우 구조체 또는 공용 구조체 형식을 정의하기 전에 구조체 또는 공용 구조체 형식에 대한 포인터의 `typedef` 이름을 선언할 수 있습니다.

예제

선언의 `typedef` 한 가지 사용은 선언을 보다 깔끔하고 간결하게 만드는 것입니다. 예시:

C++

```
typedef char CHAR;           // Character type.  
typedef CHAR * PSTR;        // Pointer to a string (char *).  
PSTR strchr( PSTR source, CHAR target );  
typedef unsigned long ulong;  
ulong ul;      // Equivalent to "unsigned long ul;"
```

동일한 선언에서 기본 형식과 파생 형식을 지정하는 데 사용 `typedef` 하려면 선언자를 쉼표로 구분할 수 있습니다. 예시:

C++

```
typedef char CHAR, *PSTR;
```

다음 예제에서는 값을 반환하지 않고 두 개의 int 인수를 사용하는 함수에 대한 `DRAWF` 형식을 제공합니다.

C++

```
typedef void DRAWF( int, int );
```

위의 `typedef` 문 다음에 선언합니다.

C++

```
DRAWF box;
```

다음 선언과 동일합니다.

C++

```
void box( int, int );
```

`typedef` 는 사용자 정의 형식을 선언하고 이름을 지정하기 위해 결합 `struct` 되는 경우가 많습니다.

C++

```
// typedefSpecifier2.cpp  
#include <stdio.h>  
  
typedef struct mystructtag  
{  
    int i;  
    double f;  
} mystruct;
```

```
int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d %f\n", ms.i, ms.f);
}
```

Output

```
10 0.990000
```

typedefs 다시 선언

이 선언은 `typedef` 동일한 형식을 참조하기 위해 동일한 이름을 다시 묶는 데 사용할 수 있습니다. 예시:

원본 파일 `file1.h`:

C++

```
// file1.h
typedef char CHAR;
```

원본 파일 `file2.h`:

C++

```
// file2.h
typedef char CHAR;
```

원본 파일 `prog.cpp`:

C++

```
// prog.cpp
#include "file1.h"
#include "file2.h" // OK
```

파일에 `prog.cpp` 는 두 개의 헤더 파일이 포함되며, 둘 다 이름 `CHAR`에 대한 선언을 포함합니다 `typedef`. 두 선언이 동일한 형식을 참조하는 한 이러한 재선언은 허용됩니다.

A `typedef` 는 이전에 다른 형식으로 선언된 이름을 다시 정의할 수 없습니다. 이 대안은 다음과 같습니다. `file2.h`

```
C++  
  
// file2.h  
typedef int CHAR;      // Error
```

컴파일러는 다른 형식을 참조하기 위해 이름을 `CHAR` 다시 묶으려고 했기 때문에 오류를 `prog.cpp` 발생합니다. 이 정책은 다음과 같은 구문으로 확장됩니다.

```
C++  
  
typedef char CHAR;  
typedef CHAR CHAR;      // OK: redeclared as same type  
  
typedef union REGS      // OK: name REGS redeclared  
{  
    struct wordregs x;   // by typedef name with the  
    structbyteregs h;  
} REGS;
```

C++ 및 C의 `typedef`

선언에서 `typedef` 명명되지 않은 구조를 `typedef` 선언하는 ANSI C 사례로 인해 클래스 형식과 함께 지정자의 사용이 주로 지원됩니다. 예를 들어 많은 C 프로그래머는 다음 관용구를 사용합니다.

```
C++  
  
// typedef_with_class_types1.cpp  
// compile with: /c  
typedef struct {    // Declare an unnamed structure and give it the  
                    // typedef name POINT.  
    unsigned x;  
    unsigned y;  
} POINT;
```

이러한 선언의 장점은 다음과 같은 선언이 가능하다는 것입니다.

```
C++  
  
POINT ptOrigin;
```

위의 선언을 아래의 선언 대신 사용할 수 있습니다.

C++

```
struct point_t ptOrigin;
```

C++에서는 이름과 실제 형식(, , `struct` `union` 및 `enum` 키워드(keyword)으로 선언됨 `class`)의 차이가 `typedef` 더 뚜렷합니다. 문에서 `typedef` 이름 없는 구조를 선언하는 C 사례는 여전히 작동하지만 C에서와 마찬가지로 표기법상의 이점을 제공하지 않습니다.

C++

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

앞의 예제에서는 명명되지 않은 클래스 구문을 사용하여 명명 `POINT` 된 클래스 `typedef`를 선언합니다. `POINT`는 클래스 이름으로 간주되지만 다음과 같은 제한 사항이 이런 방식으로 생성된 이름에 적용됩니다.

- 이름(동의어)은, `struct` 또는 `union` 접두사 뒤를 `class` 표시할 수 없습니다.
- 이 이름은 클래스 선언 내에서 생성자 또는 소멸자 이름으로 사용할 수 없습니다.

요약하자면, 이 구문은 상속, 생성 또는 소멸을 위한 메커니즘을 제공하지 않습니다.

선언 사용

아티클 • 2024. 07. 15.

using 선언은 using 선언이 나타나는 선언 영역에 이름을 도입합니다.

구문

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

매개 변수

nested-name-specifier 범위 확인 연산자로 끝나는 일련의 네임스페이스, 클래스 또는 열거형 이름과 범위 확인 연산자(:)입니다. 단일 범위 확인 연산자를 사용하여 전역 이름 공간에서 이름을 도입할 수 있습니다. **typename** 키워드는 선택 사항이며 기본 클래스에서 클래스 템플릿에 도입될 때 종속 이름을 확인하는 데 사용될 수 있습니다.

unqualified-id 식별자, 오버로드된 연산자 이름, 사용자 정의 리터럴 연산자 또는 변환 함수 이름, 클래스 소멸자 이름 또는 템플릿 이름 및 인수 목록일 수 있는 정규화되지 않은 ID 식입니다.

declarator-list [**typename**] *nested-name-specifier unqualified-id* 선언자의 쉼표로 구분된 목록이며 선택적으로 줄임표가 뒤에옵니다.

설명

using 선언은 다른 곳에서 선언된 엔터티에 대한 동의어로 정규화되지 않은 이름을 도입합니다. 이는 특정 네임스페이스의 단일 이름이 나타나는 선언 영역에서 명시적인 한정 없이 사용될 수 있도록 허용합니다. 이는 네임스페이스의 모든 이름을 한정 없이 사용할 수 있도록 허용하는 [using 지시문](#)과 대조됩니다. **using** 키워드는 [형식 별칭](#)에도 사용됩니다.

예: 클래스 필드의 **using** 선언

클래스 정의에서 using 선언을 사용할 수 있습니다.

```

// using_declaration1.cpp
#include <stdio.h>
class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class D : B {
public:
    using B::f;      // B::f(char) is now visible as D::f(char)
    using B::g;      // B::g(char) is now visible as D::g(char)
    void f(int) {
        printf_s("In D::f()\n");
        f('c');      // Invokes B::f(char) instead of recursing
    }

    void g(int) {
        printf_s("In D::g()\n");
        g('c');      // Invokes B::g(char) instead of recursing
    }
};

int main() {
    D myD;
    myD.f(1);
    myD.g('a');
}

```

Output

```

In D::f()
In B::f()
In B::g()

```

예: 멤버 선언을 위한 `using` 선언

멤버를 선언하는 데 사용되는 경우 `using` 선언은 기본 클래스의 멤버를 참조해야 합니다.

C++

```

// using_declaration2.cpp
#include <stdio.h>

```

```

class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class C {
public:
    int g();
};

class D2 : public B {
public:
    using B::f; // ok: B is a base of D2
    // using C::g; // error: C isn't a base of D2
};

int main() {
    D2 MyD2;
    MyD2.f('a');
}

```

Output

In B::f()

예: 명시적 한정이 있는 `using` 선언

`using` 선언을 사용하여 선언된 멤버는 명시적 한정자를 사용하여 참조할 수 있습니다. `::` 접두사는 전역 네임스페이스를 나타냅니다.

C++

```

// using_declaration3.cpp
#include <stdio.h>

void f() {
    printf_s("In f\n");
}

namespace A {
    void g() {
        printf_s("In A::g\n");
    }
}

```

```

namespace X {
    using ::f;    // global f is also visible as X::f
    using A::g;   // A's g is now visible as X::g
}

void h() {
    printf_s("In h\n");
    X::f();      // calls ::f
    X::g();      // calls A::g
}

int main() {
    h();
}

```

Output

```

In h
In f
In A::g

```

예: `using` 선언 동의어 및 별칭

`using` 선언이 만들어지면 선언에서 만든 동의어는 `using` 선언의 지점에서 유효한 정의만 참조합니다. `using` 선언 뒤에 네임스페이스에 추가된 정의는 유효한 동의어가 아닙니다.

`using` 선언으로 정의된 이름은 원래 이름의 별칭입니다. 원래 선언의 형식, 연결 또는 기타 특성에는 영향을 주지 않습니다.

C++

```

// post_declarator_namespace_additions.cpp
// compile with: /c
namespace A {
    void f(int) {}
}

using A::f;    // f is a synonym for A::f(int) only

namespace A {
    void f(char) {}
}

void f() {
    f('a');    // refers to A::f(int), even though A::f(char) exists
}

void b() {

```

```
    using A::f;    // refers to A::f(int) AND A::f(char)
    f('a');      // calls A::f(char);
}
```

예: 로컬 선언 및 `using` 선언

네임스페이스의 함수와 관련하여 단일 이름에 대한 로컬 선언 집합과 `using` 선언이 선언 영역에 제공되는 경우 모두 동일한 엔터티를 참조하거나 모두 함수를 참조해야 합니다.

C++

```
// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}

void g() {
    int i;
    using B::i;    // error: i declared twice
    void f(char);
    using B::f;    // ok: each f is a function
}
```

위의 예에서 `using B::i` 문은 두 번째 `int i`가 `g()` 함수에서 선언되도록 합니다. `using B::f` 문은 `B::f`에 의해 도입된 함수 이름이 다른 매개 변수 형식을 가지므로 `f(char)` 함수와 충돌하지 않습니다.

예: 로컬 함수 선언 및 `using` 선언

로컬 함수 선언은 `using` 선언으로 도입된 함수와 동일한 이름 및 형식을 가질 수 없습니다. 예시:

C++

```
// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
```

```

    void f(char);
}

void h() {
    using B::f;           // introduces B::f(int) and B::f(double)
    using C::f;           // C::f(int), C::f(double), and C::f(char)
    f('h');              // calls C::f(char)
    f(1);                // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);          // C2883 conflicts with B::f(int) and C::f(int)
}

```

예: using 선언 및 상속

상속과 관련하여 using 선언이 기본 클래스의 이름을 파생 클래스 범위에 도입하는 경우
파생 클래스의 멤버 함수는 기본 클래스의 동일한 이름 및 인수 형식을 사용하여 가상 멤
버 함수를 재정의합니다.

C++

```

// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
    virtual void f(int) {
        printf_s("In B::f(int)\n");
    }

    virtual void f(char) {
        printf_s("In B::f(char)\n");
    }

    void g(int) {
        printf_s("In B::g\n");
    }

    void h(int);
};

struct D : B {
    using B::f;
    void f(int) { // ok: D::f(int) overrides B::f(int)
        printf_s("In D::f(int)\n");
    }

    using B::g;
    void g(char) { // ok: there is no B::g(char)
        printf_s("In D::g(char)\n");
    }

    using B::h;
    void h(int) {} // Note: D::h(int) hides non-virtual B::h(int)
};

```

```

void f(D* pd) {
    pd->f(1);      // calls D::f(int)
    pd->f('a');    // calls B::f(char)
    pd->g(1);      // calls B::g(int)
    pd->g('a');    // calls D::g(char)
}

int main() {
    D * myd = new D();
    f(myd);
}

```

Output

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

예: using 선언 접근성

using 선언에 언급된 이름의 모든 인스턴스에 액세스할 수 있어야 합니다. 특히 파생 클래스가 using 선언을 사용하여 기본 클래스의 멤버에 액세스하는 경우 멤버 이름에 액세스 할 수 있어야 합니다. 이름이 오버로드된 멤버 함수의 이름인 경우 명명된 모든 함수에 액세스할 수 있어야 합니다.

멤버 접근성에 대한 자세한 내용은 [멤버 액세스 제어](#)를 참조하세요.

C++

```

// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f;    // C2876: A::f(char) is inaccessible
public:
    using A::g;    // B::g is a public synonym for A::g
};

```

참고 항목

네임스페이스

키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

volatile (C++)

아티클 • 2024. 11. 21.

하드웨어에 의해 프로그램에서 수정할 수 있는 개체를 선언하는 데 사용할 수 있는 형식 한정자입니다.

구문

```
volatile declarator ;
```

설명

/volatile 컴파일러 스위치를 사용하여 컴파일러가 이 키워드를 해석하는 방법을 수정할 수 있습니다.

Visual Studio는 대상 아키텍처에 `volatile` 따라 키워드를 다르게 해석합니다. ARM의 경우 /volatile 컴파일러 옵션이 지정되지 않은 경우 컴파일러는 /volatile:iso가 지정된 것처럼 수행됩니다. ARM 이외의 아키텍처의 경우 /volatile 컴파일러 옵션이 지정되지 않은 경우 컴파일러는 /volatile:ms가 지정된 것처럼 수행되므로 ARM 이외의 아키텍처의 경우 /volatile:iso를 지정하고 스레드 간에 공유되는 메모리를 처리할 때 명시적 동기화 기본 형식 및 컴파일러 내장 함수를 사용하는 것이 좋습니다.

한정자를 `volatile` 사용하여 인터럽트 처리기와 같은 비동기 프로세스에서 사용되는 메모리 위치에 대한 액세스를 제공할 수 있습니다.

`_restrict` 키워드 `volatile` 도 있는 변수에 사용되는 경우 `volatile` 우선합니다.

멤버가 `struct` 표시된 `volatile volatile` 경우 전체 구조체로 전파됩니다. 하나의 명령을 `volatile` 사용하여 현재 아키텍처에서 복사할 수 있는 길이가 구조체에 없는 경우 해당 구조체에서 완전히 손실될 수 있습니다.

`volatile` 다음 조건 중 하나가 true인 경우 키워드가 필드에 영향을 주지 않을 수 있습니다.

- `volatile` 필드의 길이가 현재 아키텍처에서 하나의 명령을 사용하여 복사할 수 있는 최대 크기를 초과합니다.
- 가장 바깥쪽에 포함되거나 `struct` 중첩된 `struct` 멤버인 경우 하나의 명령을 사용하여 현재 아키텍처에서 복사할 수 있는 최대 크기를 초과합니다.

프로세서는 캐시할 수 없는 메모리 액세스를 다시 정렬하지 않지만 캐시할 수 없는 변수를 표시 `volatile` 하여 컴파일러가 메모리 액세스의 순서를 다시 지정하지 않도록 해야 합니다.

특정 최적화에 사용되지 않는 것으로 `volatile` 선언된 개체는 해당 값이 언제든지 변경될 수 있기 때문입니다. 이전 명령에서 동일한 개체의 값을 요청했더라도 시스템은 요청될 때 항상 휘발성 개체의 현재 값을 읽습니다. 또한 개체의 값은 할당 시 즉시 기록됩니다.

ISO 준수

C# `volatile` 키워드에 익숙하거나 이전 버전의 MICROSOFT C++ 컴파일러(MSVC)의 `volatile` 동작에 익숙한 경우 C++11 ISO Standard `volatile` 키워드는 다르며 `/volatile:iso` 컴파일러 옵션이 지정된 경우 MSVC에서 지원됩니다. ARM의 경우 기본적으로 지정됩니다. `volatile` C++11 ISO 표준 코드의 키워드는 하드웨어 액세스에만 사용되며 스레드 간 통신에는 사용하지 마세요. 스레드 간 통신의 경우 C++ 표준 라이브러리의 `std::atomic<T>`와 같은 메커니즘을 사용합니다.

ISO 규칙의 끝

Microsoft 전용

`/volatile:ms` 컴파일러 옵션을 사용하는 경우(기본적으로 ARM 이외의 아키텍처가 대상으로 지정된 경우) 컴파일러는 다른 전역 개체에 대한 참조 순서를 유지하는 것 외에도 휘발성 개체에 대한 참조 간의 순서를 유지하기 위한 추가 코드를 생성합니다. 특히 다음 사항에 주의하십시오.

- `volatile` 개체 쓰기(`volatile` 쓰기)는 Release 의미 체계를 사용합니다. 즉 명령 시퀀스에서 `volatile` 개체에 쓰기 전에 발생하는 전역 또는 정적 개체에 대한 참조는 컴파일된 이진 파일에서 `volatile` 쓰기 전에 발생합니다.
- `volatile` 개체 읽기(`volatile` 읽기)는 Acquire 의미 체계를 사용합니다. 즉 명령 시퀀스에서 `volatile` 메모리 읽기 다음에 발생하는 전역 또는 정적 개체에 대한 참조는 컴파일된 이진 파일에서 `volatile` 읽기 다음에 발생합니다.

이렇게 하면 다중 스레드 애플리케이션에서 메모리 잠금 및 릴리스에 휘발성 개체를 사용할 수 있습니다.

① 참고

/volatile:ms 컴파일러 옵션을 사용할 때 제공되는 향상된 보장을 사용하는 경우 코드는 이식 가능하지 않습니다.

Microsoft 전용 종료

참고 항목

키워드

const

const 및 volatile 포인터

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

decltype (C++)

아티클 • 2024. 07. 08.

`decltype` 형식 지정자는 지정된 식의 형식을 생성합니다. `decltype` 형식 지정자는 `auto` 키워드와 함께 템플릿 라이브러리를 작성하는 개발자에게 주로 유용합니다. 반환 형식이 템플릿 인수 형식에 따라 달라지는 함수 템플릿을 선언하려면 `auto` 및 `decltype`을 사용합니다. 또는 `auto` 및 `decltype`을 사용하여 다른 함수에 대한 호출을 래핑하는 함수 템플릿을 선언한 다음 래핑된 함수의 반환 형식을 반환합니다.

구문

```
decltype( expression )
```

매개 변수

expression

식입니다. 자세한 내용은 [식](#)을 참조하세요.

반환 값

expression 매개 변수의 형식입니다.

설명

`decltype` 형식 지정자는 Visual Studio 2010 이상 버전에서 지원되며 네이티브 코드 또는 관리 코드와 함께 사용할 수 있습니다. `decltype(auto)` (C++14)는 Visual Studio 2015 이상에서 지원됩니다.

컴파일러는 다음 규칙을 사용하여 *expression* 매개 변수의 형식을 결정합니다.

- *expression* 매개 변수가 식별자 또는 [클래스 멤버 액세스](#)인 경우
`decltype(expression)`은 *expression*에 의해 이름이 지정된 엔터티의 형식입니다.
그러한 엔터티가 없거나 *expression* 매개 변수가 오버로드된 함수 집합에 이름을 지정하는 경우에는 컴파일러가 오류 메시지를 생성합니다.
- *expression* 매개 변수가 함수 또는 오버로드된 연산자 함수에 대한 호출인 경우
`decltype(expression)`은 함수의 반환 형식입니다. 오버로드된 연산자를 묶는 괄호는 무시됩니다.

- `expression` 매개 변수가 `rvalue`인 경우 `decltype(expression)`은 `expression`의 형식입니다. `expression` 매개 변수가 `lvalue`인 경우에는 `decltype(expression)`이 `expression` 형식에 대한 `lvalue` 참조입니다.

다음 코드 예제에서는 `decltype` 형식 지정자의 용례를 보여 줍니다. 먼저 다음 문을 코딩 했다고 가정합니다.

C++

```
int var;
const int&& fx();
struct A { double x; };
const A* a = new A();
```

다음으로 아래 표에서 4개의 `decltype` 문에 의해 반환되는 형식을 검토합니다.

[+] 테이블 확장

문	Type	주의
<code>decltype(fx());</code>	<code>const int&&</code>	<code>const int</code> 에 대한 <code>rvalue</code> 참조입니다.
<code>decltype(var);</code>	<code>int</code>	<code>var</code> 변수의 형식입니다.
<code>decltype(a->x);</code>	<code>double</code>	멤버 액세스의 형식입니다.
<code>decltype((a->x));</code>	<code>const double&</code>	내부 괄호를 사용하면 문이 멤버 액세스가 아니라 식으로 평가됩니다. <code>a</code> 가 <code>const</code> 포인터로 선언되므로 형식은 <code>const double</code> 에 대한 참조입니다.

decltype 및 auto

C++14에서는 후행 반환 형식 없이 `decltype(auto)`를 사용하여 반환 형식이 템플릿 인수 형식에 따라 달라지는 함수 템플릿을 선언할 수 있습니다.

C++11에서는 `auto` 키워드와 함께 후행 반환 형식에 대한 `decltype` 형식 지정자를 사용하여 반환 형식이 해당 템플릿 인수의 형식에 종속되는 함수 템플릿을 선언할 수 있습니다. 예를 들어, 함수 템플릿의 반환 형식이 템플릿 인수의 형식에 종속되는 것을 보여 주는 다음 코드 예제를 살펴보세요. 코드 예제에서 `UNKNOWN` 자리 표시자는 반환 형식을 지정할 수 없음을 나타냅니다.

C++

```
template<typename T, typename U>
UNKNOWN_func(T&& t, U&& u){ return t + u; };
```

`decltype` 형식 지정자의 도입으로 개발자는 함수 템플릿이 반환하는 식의 형식을 가져올 수 있습니다. 컴파일하면 지정되는 반환 형식을 선언하려면 나중에 살펴볼 대체 함수 선언 구문, `auto` 키워드 및 `decltype` 형식 지정자를 사용하세요. 컴파일하면 지정되는 반환 형식은 코딩 시가 아니라 선언이 컴파일될 때 결정됩니다.

다음 프로토타입에서는 대체 함수 선언의 구문을 보여 줍니다. `const` 및 `volatile` 한정자와 `throw` 예외 사양은 선택 사항입니다. `function_body` 자리 표시자는 함수가 수행하는 작업을 지정하는 복합 문을 나타냅니다. 모범 코딩 사례는 `decltype` 문의 `expression` 자리 표시자를 `function_body`의 `return` 문(있는 경우)에 의해 지정된 식과 일치시키는 것입니다.

```
auto function_name ( parameters_opt ) const_opt volatile_opt -> decltype( expression )
) noexcept_opt { function_body };
```

다음 코드 예제에서는 `myFunc` 함수 템플릿의 컴파일하면 지정되는 반환 형식이 `t` 및 `u` 템플릿 인수의 형식에 따라 결정됩니다. 모범 코딩 사례에 따라 코드 예제는 rvalue 참조 및 `forward` 함수 템플릿(완벽한 전달 지원)도 사용합니다. 자세한 내용은 [RValue 참조 선언자: &&](#)를 참조하세요.

C++

```
//C++11
template<typename T, typename U>
auto myFunc(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
{ return forward<T>(t) + forward<U>(u); }

//C++14
template<typename T, typename U>
decltype(auto) myFunc(T&& t, U&& u)
{ return forward<T>(t) + forward<U>(u); }
```

decltype 및 전달 함수(C++11)

전달 함수는 다른 함수에 대한 호출을 래핑합니다. 해당 인수 또는 이러한 인수를 포함하는 식의 결과를 다른 함수로 전달하는 함수 템플릿을 고려해 보십시오. 또한 전달 함수는 다른 함수를 호출한 결과를 반환합니다. 이 시나리오에서 전달 함수의 반환 형식은 래핑 된 함수의 반환 형식과 동일해야 합니다.

이 시나리오에서는 `decltype` 형식 지정자 없이 적절한 형식 식을 쓸 수 없습니다.

`decltype` 형식 지정자는 함수가 참조 형식을 반환하는지 여부에 대한 필수 정보를 잊지 않기 때문에 제네릭 전달 함수를 사용할 수 있게 만듭니다. 전달 함수의 코드 예는 이전 `myFunc` 함수 템플릿 예를 참조하세요.

예제

다음 코드 예에서는 함수 템플릿 `Plus()`의 늦게 지정된 반환 형식을 선언합니다. `Plus` 함수는 해당 두 피연산자를 `operator+` 오버로드로 처리합니다. 따라서 `Plus` 함수의 반환 형식 및 더하기 연산자(`+`)에 대한 해석은 함수 인수의 형식에 따라 달라집니다.

C++

```
// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
    decltype(forward<T1>(t1) + forward<T2>(t2))
{
    return forward<T1>(t1) + forward<T2>(t2);
}

class X
{
    friend X operator+(const X& x1, const X& x2)
    {
        return X(x1.m_data + x2.m_data);
    }

public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data; }
private:
    int m_data;
};

int main()
{
    // Integer
    int i = 4;
    cout <<
        "Plus(i, 9) = " <<
```

```

    Plus(i, 9) << endl;

    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout <<
        setprecision(3) <<
        "Plus(dx, dy) = " <<
        Plus(dx, dy) << endl;

    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;

    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout <<
        "x3.Dump() = " <<
        x3.Dump() << endl;
}

```

Output

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

Visual Studio 2017 이상: 템플릿이 인스턴스화되지 않고 선언될 때 컴파일러는 `decltype` 인수를 구문 분석합니다. 따라서 `decltype` 인수에서 비종속 전문화가 발견되면 인스턴스화 시간으로 연기되지 않습니다. 즉시 처리되며 결과로 발생하는 모든 오류는 그때 진단됩니다.

다음 예제에서는 선언 시점에 발생한 컴파일러 오류를 보여 줍니다.

C++

```

#include <utility>
template <class T, class ReturnT, class... ArgsT> class IsCallable
{
public:
    struct BadType {};
    template <class U>
    static decltype(std::declval<T>()(std::declval<ArgsT>(...)) Test(int);
//C2064. Should be declval<U>
    template <class U>
    static BadType Test(...);

```

```
    static constexpr bool value = std::is_convertible<decltype(Test<T>(0)),  
ReturnT>::value;  
};  
  
constexpr bool test1 = IsCallable<int(), int>::value;  
static_assert(test1, "PASS1");  
constexpr bool test2 = !IsCallable<int*, int>::value;  
static_assert(test2, "PASS2");
```

요구 사항

Visual Studio 2010 이상 버전

`decltype(auto)`의 경우 Visual Studio 2015 이상이 필요합니다.

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

C++의 특성

아티클 • 2024. 10. 16.

C++ 표준은 공통 특성 집합을 정의합니다. 또한 컴파일러 공급업체가 공급업체별 네임스페이스 내에서 자체 특성을 정의할 수 있습니다. 그러나 컴파일러는 표준에 정의된 특성을 인식하기만 해야 합니다.

경우에 따라 표준 특성은 컴파일러별 `_declspec` 매개 변수와 겹칩니다. Microsoft C++에서는 `_declspec(deprecated)` 을(를) 사용하는 대신 `[[deprecated]]` 특성을 사용할 수 있습니다. `[[deprecated]]` 특성은 모든 준수 컴파일러에서 인식됩니다. `dllimport` 및 `dllexport` 등의 다른 모든 `_declspec` 매개변수의 경우 지금까지는 해당하는 특성이 없으므로 `_declspec` 구문을 계속 사용해야 합니다. 특성은 형식 시스템에 영향을 주지 않으며 프로그램의 의미를 변경하지 않습니다. 컴파일러는 인식하지 못하는 특성 값을 무시합니다.

Visual Studio 2017 버전 15.3 이상(사용 가능: `/std:c++17` 이상): 특성 목록의 범위에서 단일 `using` 소개자를 사용하여 모든 이름의 네임스페이스를 지정할 수 있습니다.

```
C++  
  
void g() {  
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel,  
    rpr::target(cpu,gpu) ]]  
    do task();  
}
```

C++ 표준 특성

C++11에서 특성은 C++ 구문(클래스, 함수, 변수 및 블록 포함)에 추가 정보를 주석으로 추가하는 표준화된 방법을 제공합니다. 특성은 공급업체에 따라 다를 수도 있고 그렇지 않을 수도 있습니다. 컴파일러는 이 정보를 사용하여 정보 메시지를 생성하거나 특성 코드를 컴파일할 때 특별한 로직을 적용할 수 있습니다. 컴파일러는 인식하지 못하는 특성은 무시하므로 이 구문을 사용하여 사용자 정의 특성을 정의할 수 없습니다. 특성은 이중 대괄호로 묶입니다.

```
C++  
  
[[deprecated]]  
void Foo(int);
```

특성은 `#pragma` 지시문, `__declspec()` (Visual C++) 또는 `__attribute__` (GNU)와 같은 공급업체별 확장에 대한 표준화된 대안을 나타냅니다. 그러나 대부분의 목적에서는 여전히 공급업체별 구성을 사용해야 합니다. 현재 표준은 호환 컴파일러가 인식해야 하는 다음 특성을 지정하고 있습니다.

[[carries_dependency]]

이 [[carries_dependency]] 특성은 함수가 스레드 동기화에 대한 데이터 종속성 순서를 전파하도록 지정합니다. 이 특성은 하나 이상의 매개변수에 적용하여 전달된 인수가 함수 본문에 종속성을 전달하도록 지정할 수 있습니다. 이 특성을 함수 자체에 적용하여 반환 값이 함수 외부의 종속성을 전달하도록 지정할 수 있습니다. 컴파일러는 이 정보를 사용하여 보다 효율적인 코드를 생성할 수 있습니다.

[[deprecated]]

Visual Studio 2015 이상: [[deprecated]] 특성은 함수가 사용되지 않도록 지정합니다. 또는 향후 버전의 라이브러리 인터페이스에는 존재하지 않을 수도 있습니다.

[[deprecated]] 특성은 클래스, `typedef-name`, 변수, 비정적 데이터 멤버, 함수, 네임스페이스, 열거형, 열거자 또는 템플릿 특수화의 선언에 적용할 수 있습니다. 컴파일러는 클라이언트 코드가 함수 호출을 시도할 때 이 속성을 사용하여 정보 메시지를 생성할 수 있습니다. Microsoft C++ 컴파일러가 [[deprecated]] 항목 사용을 감지하면 컴파일러 경고 C4996이 발생합니다.

[[fallthrough]]

Visual Studio 2017 이상: (사용 가능: `/std:c++17` 이상) 이 [[fallthrough]] 특성은 `switch` 문 컨텍스트에서 대체 동작이 의도된 컴파일러(또는 코드를 읽는 모든 사용자)에 대한 힌트로 사용할 수 있습니다. Microsoft C++ 컴파일러는 현재 fallthrough 동작에 대해 경고하지 않으므로 이 특성은 컴파일러 동작에 영향을 주지 않습니다.

[[likely]]

Visual Studio 2019 버전 16.6 이상: (사용 가능: `/std:c++20` 이상) [[likely]] 특성은 특성 레이블 또는 문의 코드 경로가 대안보다 실행될 가능성이 더 높다는 힌트를 컴파일러에 지정합니다. Microsoft 컴파일러에서 [[likely]] 특성은 블록을 '핫 코드'로 표시하여 내부 최적화 점수를 높입니다. 속도에 최적화할 때는 점수가 더 많이 증가하고 크기에 최적화할 때는 점수가 많이 증가하지 않습니다. 순 점수는 인라인, 루프 언롤링 및 벡터화 최적화 가능성에 영향을 미칩니다. [[likely]] 및 [[unlikely]]의 효과는 [프로필 기반 최적화](#)

화와 유사하지만 범위가 현재 번역 단위로 제한됩니다. 이 속성에 대한 블록 재정렬 최적화는 아직 구현되지 않았습니다.

[[maybe_unused]]

Visual Studio 2017 버전 15.3 이상: (사용 가능: /std:c++17 이상) [[maybe_unused]] 특성은 변수, 함수, 클래스, typedef, 비정적 데이터 멤버, 열거형 또는 템플릿 특수화가 의도적으로 사용되지 않을 수 있음을 지정합니다. 표시된 [[maybe_unused]] 엔터티가 사용되지 않는 경우 컴파일러는 경고하지 않습니다. 속성 없이 선언된 엔티티는 나중에 속성을 사용하여 다시 선언할 수 있으며, 그 반대의 경우도 마찬가지입니다. 엔티티는 [[maybe_unused]](으)로 표시된 첫 번째 선언이 분석된 후 현재 번역 단위의 나머지 부분에 대해 표시됨으로 간주됩니다.

[[nodiscard]]

Visual Studio 2017 버전 15.3 이상: (사용 가능: /std:c++17 이상) 함수의 반환 값이 삭제되지 않도록 지정합니다. 다음 예제와 같이 경고 C4834를 발생합니다.

C++

```
[[nodiscard]]
int foo(int i) { return i * i; }

int main()
{
    foo(42); //warning C4834: discarding return value of function with
    'nodiscard' attribute
    return 0;
}
```

[[noreturn]]

[[noreturn]] 특성은 함수가 반환되지 않도록 지정합니다. 즉 항상 예외를 throw하거나 종료합니다. 컴파일러는 [[noreturn]] 엔터티에 대한 컴파일 규칙을 조정할 수 있습니다.

[[unlikely]]

Visual Studio 2019 버전 16.6 이상: (사용 가능: /std:c++20 이상) [[unlikely]] 특성은 특정 레이블 또는 문의 코드 경로가 대안보다 실행될 가능성이 덜 높다는 힌트를 컴파일러에 지정합니다. Microsoft 컴파일러에서 [[unlikely]] 특성은 블록을 '콜드 코드'로 표시하여 내부 최적화 점수를 낮춥니다. 크기에 최적화할 때는 점수가 더 많이 증가하고 속도에 최적화할 때는 점수가 많이 감소하지 않습니다. 순 점수는 인라이닝, 루프 언롤링 및 벡

터화 최적화 가능성에 영향을 미칩니다. 이 속성에 대한 블록 재정렬 최적화는 아직 구현되지 않았습니다.

Microsoft별 특성

[[gsl::suppress(rules)]]

Microsoft별 [[gsl::suppress(rules)]] 특성은 코드에서 [GSL\(Guidelines Support Library\)](#) 규칙을 적용하는 검사기의 경고를 표시하지 않는 데 사용됩니다. 예를 들어 다음 코드 조각을 고려하세요.

C++

```
int main()
{
    int arr[10]; // GSL warning C26494 will be fired
    int* p = arr; // GSL warning C26485 will be fired
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
    {
        int* q = p + 1; // GSL warning C26481 suppressed
        p = q--; // GSL warning C26481 suppressed
    }
}
```

이 예제에서는 이러한 경고를 제거합니다.

- [C26494](#)(형식 규칙 5: 항상 개체를 초기화합니다.)
- [C26485](#)(경계 규칙 3: 포인터 감쇠에 대한 배열 없음)
- [C26481](#)(경계 규칙 1: 포인터 산술 연산을 사용하지 말 것. 대신 범위를 사용)

처음 두 개의 경고는 CppCoreCheck 코드 분석 도구를 설치 및 활성화한 상태에서 이 코드를 컴파일할 때 발생합니다. 그러나 세 번째 경고는 특성 때문에 실행되지 않습니다. 특정 규칙 번호를 포함하지 않고 [[gsl::suppress(bounds)]] 을(를) 작성하여 전체 경계 프로필을 표시하지 않을 수 있습니다. C++ 핵심 가이드라인은 더 좋고 안전한 코드를 작성하는 데 도움이 되도록 설계되었습니다. suppress 특성을 사용하면 원치 않을 때 경고를 쉽게 끌 수 있습니다.

[[msvc::flatten]]

Microsoft별 특성 [[msvc::flatten]] 은(는) [[msvc::forceinline_calls]] 과(와) 매우 유사하며 동일한 위치와 동일한 방식으로 사용할 수 있습니다. 차이점은 [[msvc::flatten]] 은(는) 호출이 남지 않을 때까지 재귀적으로 적용되는 범위의 모든 호출을

`[[msvc::forceinline_calls]]` 한다는 것입니다. 이는 결과적으로 함수의 코드 크기 증가 또는 컴파일러의 처리량에 영향을 미칠 수 있으며, 이를 수동으로 관리해야 합니다.

[[msvc::forceinline]]

함수 선언 앞에 배치할 때 Microsoft 전용 특성 `[[msvc::forceinline]]` 은(는) `_forceinline` 과(와) 동일한 의미를 갖습니다.

[[msvc::forceinline_calls]]

Microsoft 별 특성 `[[msvc::forceinline_calls]]` 은(는) 문 또는 블록 위 또는 그 앞에 배치 할 수 있습니다. 인라인 추론이 해당 문 또는 블록의 모든 호출을 `[[msvc::forceinline]]` (으)로 시도합니다.

```
C++  
  
void f() {  
    [[msvc::forceinline_calls]]  
    {  
        foo();  
        bar();  
    }  
    ...  
    [[msvc::forceinline_calls]]  
    bar();  
  
    foo();  
}
```

`foo`에 대한 첫 번째 호출과 `bar`에 대한 두 번째 호출은 `_forceinline`(으)로 선언된 것처럼 취급됩니다. `foo`에 대한 두 번째 호출은 `_forceinline`(으)로 취급되지 않습니다.

[[msvc::intrinsic]]

`[[msvc::intrinsic]]` 특성에는 적용되는 함수에 대한 세 가지 제약 조건이 있습니다.

- 함수는 재귀적일 수 없으며, 본문에는 매개변수 유형에서 반환 유형까지 `static_cast`이(가) 포함된 반환문만 있어야 합니다.
- 이 함수는 하나의 매개변수만 사용할 수 있습니다.
- `/permissive-` 컴파일러 옵션이 필요합니다. (`/std:c++20` 이상 옵션은 기본적으로 `/permissive-` 을(를) 의미합니다.)

Microsoft 별 특성 `[[msvc::intrinsic]]` 은(는) 매개 변수 형식에서 반환 형식으로 명명된 캐스트 역할을 하는 메타 함수를 인라인으로 컴파일러에 지시합니다. 함수 정의에 특성이 있으면 컴파일러는 해당 함수에 대한 모든 호출을 간단한 형변환으로 대체합니다. `[[msvc::intrinsic]]` 특성은 Visual Studio 2022 버전 17.5 미리 보기 2 이상 버전에서 사용할 수 있습니다. 이 특성은 뒤에 오는 특정 함수에만 적용됩니다.

예시

이 샘플 코드에서 `my_move` 함수에 적용된 `[[msvc::intrinsic]]` 특성은 컴파일러가 함수에 대한 호출을 본문의 인라인 정적 캐스트로 대체합니다.

C++

```
template <typename T>
[[msvc::intrinsic]] T&& my_move(T&& t) { return static_cast<T&&>(t); }

void f() {
    int i = 0;
    i = my_move(i);
}
```

[[msvc::noinline]]

함수 선언 앞에 배치할 때 Microsoft 전용 특성 `[[msvc::noinline]]` 은(는) `__declspec(noinline)` 과(와) 동일한 의미를 갖습니다.

[[msvc::noinline_calls]]

Microsoft 별 특성 `[[msvc::noinline_calls]]` 은 `[[msvc::forceinline_calls]]` 과(와) 사용 법이 동일합니다. 문이나 블록 앞에 배치할 수 있습니다. 해당 블록의 모든 호출을 강제 인라이닝하는 대신 해당 블록이 적용되는 범위에 대한 인라이닝을 해제하는 효과가 있습니다.

[[msvc::no_tls_guard]]

Microsoft 별 `[[msvc::no_tls_guard]]` 특성은 DLL의 스레드 지역 변수에 대한 첫 번째 액세스에서 초기화에 대한 검사를 사용하지 않도록 설정합니다. 이 검사는 Visual Studio 2019 버전 16.5 이상을 사용하여 빌드한 코드에서 기본적으로 활성화됩니다. 이 특성은 뒤에 오는 특정 변수에만 적용됩니다. 전역적으로 검사를 사용하지 않도록 설정하려면 [/Zc:tlsGuards-](#) 컴파일러 옵션을 사용합니다.

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

C++ 기본 제공 연산자, 우선 순위 및 결합성

아티클 • 2023. 10. 12.

C++ 언어는 모든 C 연산자를 포함하며 몇 가지 새로운 연산자를 추가합니다. 연산자는 둘 이상의 피연산자에 대해 수행할 평가를 지정합니다.

우선 순위 및 결합성

연산자 우선 순위는 둘 이상의 연산자를 포함하는 식의 연산 순서를 지정합니다. 연산자 연결성은 우선 순위가 같은 여러 연산자를 포함하는 식에서 피연산자가 왼쪽에 있는 피연산자 또는 오른쪽에 있는 연산자를 사용하여 그룹화되는지 여부를 지정합니다.

대체 맞춤법

C++는 일부 연산자용 대체 맞춤법을 지정합니다. C에서는 대체 맞춤법이 `iso646.h` 헤더에 <매크로로 제공됩니다. C++에서 이러한 대안은 키워드(keyword) 있으며 `iso646.h` 또는 C++ 동등한 `<ciso646>`의 사용은 더 이상 사용되지 않습니다. Microsoft C++ `/permissive-/Za`에서는 대체 맞춤법을 사용하도록 설정하려면 컴파일러 옵션이 필요합니다.

C++ 연산자 우선 순위 및 결합성 테이블

다음 표에서는 C++ 연산자의 우선 순위와 결합성을 내림차순으로 보여 줍니다. 우선 순위 번호가 같은 연산자는 괄호를 사용하여 다른 관계를 명시적으로 강제하지 않는 한 우선 순위가 같습니다.

연산자 설명	작업	대체
그룹 1 우선 순위, 결합성 없음		
범위 확인	<code>::</code>	
그룹 2 우선 순위(왼쪽에서 오른쪽 연결성)		
멤버 선택(개체 또는 포인터)	<code>. 또는 -></code>	
배열 아래 첨자	<code>[]</code>	
함수 호출	<code>()</code>	
후위 증가	<code>++</code>	

연산자 설명 후위 감소	작업	대체
형식 이름	typeid	--
상수 형식 변환	const_cast	
동적 형식 변환	dynamic_cast	
재해석된 형식 변환	reinterpret_cast	
정적 형식 변환	static_cast	
그룹 3 우선 순위, 오른쪽에서 왼쪽 연결		
개체 또는 형식의 크기	sizeof	
접두사 증가	++	
접두사 감소	--	
하나의 보수	~	compl
논리하지 않음	!	not
단항 부정	-	
단항 더하기	+	
주소-of	&	
간접 참조	*	
개체 만들기	new	
개체 삭제	delete	
캐스트	()	
그룹 4 우선 순위(왼쪽에서 오른쪽 연결성)		
멤버에 대한 포인터(개체 또는 포인터)	.* 또는 ->*	
그룹 5 우선 순위(왼쪽에서 오른쪽 연결성)		
곱하기	*	
나누기	/	
계수	%	
그룹 6 우선 순위(왼쪽에서 오른쪽 결합성)		
더하기	+	

연산자 설명	작업	대체
빼기	-	
그룹 7 우선 순위(왼쪽에서 오른쪽 결합성)		
왼쪽 시프트	<<	
오른쪽 시프트	>>	
그룹 8 우선 순위(왼쪽에서 오른쪽 연결성)		
보다 작음	<	
보다 큼	>	
작거나 같음	<=	
크거나 같음	>=	
그룹 9 우선 순위(왼쪽에서 오른쪽 연결성)		
등호	==	
같지 않음	!=	not_eq
그룹 10 우선 순위 왼쪽에서 오른쪽 결합성		
비트 AND	&	bitand
그룹 11 우선 순위(왼쪽에서 오른쪽 연결성)		
비트 배타적 OR	^	xor
그룹 12 우선 순위(왼쪽에서 오른쪽 결합성)		
비트 포함 OR		bitor
그룹 13 우선 순위(왼쪽에서 오른쪽 결합성)		
논리적 AND	&&	and
그룹 14 우선 순위(왼쪽에서 오른쪽 연결성)		
논리적 OR		or
그룹 15 우선 순위, 오른쪽에서 왼쪽 결합성		
조건부	? :	
양도	=	
곱하기 할당	*=	

연산자 설명	작업	대체
나누기 배정	/=	
모듈러스 할당	%=	
추가 할당	+ =	
빼기 할당	- =	
왼쪽 시프트 할당	<< =	
오른쪽 시프트 할당	>> =	
비트 AND 할당	& =	and_eq
비트 포함 OR 할당	=	or_eq
비트 배타적 OR 할당	^ =	xor_eq
throw 식	throw	
그룹 16 우선 순위(왼쪽에서 오른쪽 결합성)		
Comma	,	

참고 항목

[연산자 오버로드](#)

alignof 연산자

아티클 • 2024. 11. 21.

연산자는 `alignof` 지정된 형식의 맞춤을 형식의 `size_t` 값으로 바이트 단위로 반환합니다.

구문

C++

```
alignof( type )
```

설명

예시:

[+] 테이블 확장

식	값
<code>alignof(char)</code>	1
<code>alignof(short)</code>	2
<code>alignof(int)</code>	4
<code>alignof(long long)</code>	8
<code>alignof(float)</code>	4
<code>alignof(double)</code>	8

`alignof` 값은 기본 형식의 값 `sizeof` 과 동일합니다. 그러나 다음과 같은 예제를 고려해야 합니다.

C++

```
typedef struct { int a; double b; } S;  
// alignof(S) == 8
```

이 경우 값은 `alignof` 구조에서 가장 큰 요소의 맞춤 요구 사항입니다.

마찬가지로

C++

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)`이(가) 32와 같은 경우.

한 가지 용도는 사용자 고유의 메모리 할당 루틴 중 하나에 대한 `alignof` 매개 변수로 사용할 수 있습니다. 예를 들어, 다음의 정의된 구조인 `s`가 지정된 경우 `aligned_malloc`이라는 메모리 할당 루틴을 호출하여 특정한 할당 경계에서 메모리를 할당할 수 있습니다.

C++

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), alignof(S));
```

맞춤 수정에 대한 자세한 내용은 다음을 참조하세요.

- `pack`
- `align`
- `_unaligned`
- `/Zp` (구조체 멤버 맞춤)
- `x64 구조체 맞춤` 예

X86 및 x64 관련 코드에서 맞춤의 차이점에 대한 자세한 내용은 다음을 참조하세요.

- `x86 컴파일러와 충돌`

Microsoft 전용

`alignof` `_alignof` 은 Microsoft 컴파일러의 동의어입니다. C++11 표준의 일부가 되기 전에 Microsoft 관련 `_alignof` 운영자가 이 기능을 제공했습니다. 최대 이식성을 위해 Microsoft 관련 `_alignof` 연산자 `alignof` 대신 연산자를 사용해야 합니다.

이전 버전과의 호환성을 위해 `_alignof`은 `_alignof`의 동의어입니다. 단, 컴파일러 옵션 `/Za(언어 확장 사용 안 함)`가 지정된 경우는 예외입니다.

참고 항목

[단항 연산자가 있는 식](#)
[키워드](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

`_uuidof` 연산자

아티클 • 2024. 07. 15.

Microsoft 전용

식에 연결된 GUID를 검색합니다.

구문

`_uuidof (expression)`

설명

*expression*은 형식 이름, 포인터, 참조 또는 해당 형식의 배열, 이러한 형식에 특화된 템플릿 또는 이러한 형식의 변수가 될 수 있습니다. 컴파일러에서 인수를 사용하여 연결된 GUID를 찾을 수 있으면 해당 인수는 유효합니다.

특징은 0 또는 NULL이 인수로 제공된다는 점입니다. 이 경우 `_uuidof`는 0으로 구성된 GUID를 반환합니다.

이 키워드를 사용하여 연결된 GUID를 다음으로 추출합니다.

- `uuid` 확장 특성에 의한 개체입니다.
- `module` 특성으로 만들어진 라이브러리 블록입니다.

① 참고

디버그 빌드에서 `_uuidof`는 항상 동적으로(런타임으로) 개체를 초기화합니다. 릴리스 빌드에서 `_uuidof`는 정적으로(컴파일 시) 개체를 초기화할 수 있습니다.

이전 버전과의 호환성을 위해 `_uuidof`은 `_uuidof`의 동의어입니다. 단 컴파일러 옵션 `/Za`(언어 확장 사용 안 함)가 지정된 경우는 예외입니다.

예시

`ole32.lib`를 사용하여 컴파일된 다음 코드는 `module` 특성을 사용하여 만든 라이브러리 블록의 `uuid`를 표시합니다.

C++

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include "stdio.h"
#include "windows.h"

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
    int i;
};

int main() {
    LPOLESTR lpolestr;
    StringFromCLSID(__uuidof(MyLib), &lpolestr);
    wprintf_s(L"%s", lpolestr);
    CoTaskMemFree(lpolestr);
}
```

설명

라이브러리 이름이 더 이상 범위에 포함되지 않는 경우 `__uuidof` 대신 `__LIBID_`를 사용할 수 있습니다. 예시:

C++

```
StringFromCLSID(__LIBID_, &lpolestr);
```

Microsoft 전용 종료

참고 항목

[단항 연산자가 있는 식](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

가감 연산자: + 및 -

아티클 • 2024. 03. 06.

구문

```
expression + expression  
expression - expression
```

설명

가감 연산자는 다음과 같습니다.

- 더하기(+)
- 빼기(-)

이 이항 연산자는 왼쪽에서 오른쪽으로 연결됩니다.

가감 연산자는 산술 또는 포인터 형식의 피연산자를 사용합니다. 더하기(+) 연산자의 결과는 피연산자의 합계입니다. 빼기(-) 연산자의 결과는 피연산자 간의 차이입니다. 피연산자 중 하나 이상이 포인터인 경우 함수가 아닌 개체에 대한 포인터여야 합니다. 피연산자가 둘 다 포인터인 경우 둘 다 동일한 배열의 개체에 대한 포인터가 아니면 결과는 의미가 없습니다.

가감 연산자는 산술, 정수 및 스칼라 형식의 피연산자를 사용합니다. 이러한 형식은 다음 표에 정의되어 있습니다.

가감 연산자와 함께 사용되는 형식

[] 테이블 확장

Type	의미
산술	정수 계열 및 부동 형식을 전체적으로 "산술" 형식이라고 합니다.
정수	모든 크기(long, short)의 char 및 int 형식과 열거형은 "정수 계열" 형식입니다.
scalar	스칼라 피연산자는 산술 또는 포인터 형식의 피연산자입니다.

이러한 연산자의 올바른 조합은 다음과 같습니다.

산술 + 산술

스칼라 + 정수

정수 + 스칼라

산술 - 산술

스칼라 - 스칼라

더하기와 빼기는 동등한 연산이 아닙니다.

두 피연산자가 모두 산술 형식인 경우 표준 변환에서 다루는 변환이 피연산자에 적용되고 결과는 변환된 형식이 됩니다.

예시

C++

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
    int i = 5, j = 10;
    int n[SIZE] = { 0, 1, 2, 3, 4 };
    cout << "5 + 10 = " << i + j << endl
        << "5 - 10 = " << i - j << endl;

    // use pointer arithmetic on array

    cout << "n[3] = " << *( n + 3 ) << endl;
}
```

포인터 더하기

더하기 연산에서 피연산자 중 하나가 개체 배열에 대한 포인터인 경우 다른 피연산자는 정수 계열 형식이어야 합니다. 결과는 원래 포인터와 형식이 같으며 다른 배열 요소를 가리키는 포인터입니다. 다음 코드 조각에서는 이 개념을 보여 줍니다.

C++

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
```

```
{  
    *pIntArray = i;  
    cout << *pIntArray << "\n";  
    pIntArray = pIntArray + 1;  
}
```

정수 계열 값 1이 `pIntArray`에 추가되지만 이는 "주소에 1을 추가"하라는 의미가 아니라 "배열의 다음 개체(2바이트 또는 `sizeof(int)` 만큼 떨어져 있음)를 가리키도록 포인터를 조정"하라는 의미입니다.

① 참고

`pIntArray = pIntArray + 1` 형태의 코드는 C++ 프로그램에서 거의 발견되지 않습니다. 증분을 수행하려면 `pIntArray++` 또는 `pIntArray += 1` 형태를 사용하는 것이 좋습니다.

포인터 빼기

두 피연산자가 모두 포인터인 경우 빼기의 결과는 피연산자 간 배열 요소의 차이입니다. 빼기 식은 `ptrdiff_t` 형식(표준 포함 파일 `<stddef.h>`에 정의됨)의 부호 있는 정수 결과를 생성합니다.

피연산자 중 하나가 두 번째 피연산자라면 정수 계열 형식이 될 수 있습니다. 빼기 결과는 원래 포인터와 동일한 형식입니다. 빼기 값은 $(n - i)$ 번째 배열 요소에 대한 포인터입니다. 여기에서 n 은 원래의 포인터가 가리키는 요소이며 i 는 두 번째 피연산자의 정수 값입니다.

참고 항목

[이항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 가감 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Address-of 연산자: &

아티클 • 2024. 07. 15.

구문

`address-of-expression:`

& `cast-expression`

설명

단항 주소 연산자(&)는 해당 피연산자의 주소(즉, 포인터)를 반환합니다. 주소 연산자의 피연산자는 비트 필드가 아닌 개체를 참조하는 함수 지정자 또는 lvalue일 수 있습니다.

주소 연산자는 특정 lvalue 식(기본, 구조, 클래스 또는 공용체 형식의 변수 또는 첨자 배열 참조)에만 적용될 수 있습니다. 이러한 식에서는 상수 식(주소 연산자를 포함하지 않는 식)을 주소 식에 추가하거나 뺄 수 있습니다.

함수 또는 lvalue에 적용된 식의 결과는 피연산자의 형식에서 파생된 포인터 형식(rvalue)입니다. 예를 들어, 피연산자의 형식이 `char`인 경우 식의 결과 형식은 `char`의 포인터 형식입니다. `const` 또는 `volatile` 개체에 적용된 주소 연산자는 `const type *` 또는 `volatile type *`으로 평가됩니다. 여기서 `type`은 원래 개체의 형식입니다.

어떤 버전의 함수가 참조되는지가 분명한 경우에만 오버로드된 함수의 주소를 가져올 수 있습니다. 특정 오버로드된 함수의 주소를 가져오는 방법에 대한 자세한 내용은 [함수 오버로드](#)를 참조하세요.

주소 연산자가 정규화된 이름에 적용되면, 결과는 정규화된 이름이 정적 멤버를 지정하는지 여부에 따라 달라집니다. 그럴 경우 결과는 멤버의 선언에 지정된 형식의 포인터입니다. 정적 멤버가 아닌 경우 결과는 정규화된 클래스 이름에서 표시한 클래스의 멤버 이름에 대한 포인터입니다. 정규화된 클래스 이름에 대한 자세한 내용은 [기본 식](#)을 참조하세요.

예: 정적 멤버의 주소

다음 코드 조각은 클래스 멤버가 정적인지 여부에 따라 주소 연산자 결과가 어떻게 다른지 보여 줍니다.

C++

```

// expre_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
    int iValue;
    static float fValue;
};

int main() {
    int    PTM::*piValue = &PTM::iValue; // OK: non-static
    float PTM::*pfValue = &PTM::fValue; // C2440 error: static
    float *spfValue     = &PTM::fValue; // OK
}

```

이 예제에서 `&PTM::fValue` 식은 `float *` 가 정적 멤버이기 때문에 `float PTM::*` 형식이 아닌 `fValue` 형식이 됩니다.

예: 참조 형식의 주소

주소 연산자를 참조 형식에 적용하면 해당 연산자를 참조가 바인딩된 개체에 적용하는 것과 같은 결과가 도출됩니다. 예시:

C++

```

// expre_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    double d;           // Define an object of type double.
    double& rd = d;   // Define a reference to the object.

    // Obtain and compare their addresses
    if( &d == &rd )
        cout << "&d equals &rd" << endl;
}

```

Output

```
&d equals &rd
```

예: 매개 변수로서의 함수 주소

다음 예제에서는 주소 연산자를 사용하여 포인터 인수를 함수에 전달합니다.

C++

```
// expre_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
    return (*n) * (*n);
}

int main() {
    int mynum = 5;
    cout << square( &mynum ) << endl;    // pass address of int
}
```

Output

25

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[Lvalue 참조 선언자: &](#)

[연산자 주소 및 간접 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

대입 연산자

아티클 • 2024. 03. 11.

구문

`expression assignment-operator expression`

`assignment-operator`: 다음 중 하나

= *= /= %= += -= <<= >>= &= ^= |=

설명

할당 연산자는 왼쪽 피연산자가 지정한 개체에 값을 저장합니다. 할당 작업에는 두 가지 종류가 있습니다.

- 두 번째 피연산자의 값이 첫 번째 피연산자가 지정한 개체에 저장되는 **간단한 할당**.
 - 결과를 저장하기 전에 산술, 시프트 또는 비트 연산이 수행되는 **복합 할당**.
- = 연산자를 제외한 다음 표의 모든 대입 연산자는 복합 할당 연산자입니다.

대입 연산자 테이블

 테이블 확장

연산자	의미
=	첫 번째 피연산자에서 지정한 개체에 두 번째 피연산자의 값을 저장합니다(단순 할당).
*=	첫 번째 피연산자의 값과 두 번째 피연산자의 값을 곱하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
/=	첫 번째 피연산자의 값을 두 번째 피연산자의 값으로 나누어 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
%=	두 번째 피연산자의 값에서 지정한 첫 번째 피연산자의 모듈러스를 가져와서 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
+=	두 번째 연산자의 값과 첫 번째 연산자의 값을 더하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
-=	첫 번째 피연산자의 값에서 두 번째 피연산자의 값을 빼서 첫 번째 피연산자가 지정한 개체

연산자	의미
	에 결과를 저장합니다.
<code><<=</code>	두 번째 피연산자의 값에서 지정한 비트 수만큼 첫 번째 피연산자의 값을 왼쪽으로 이동하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
<code>>>=</code>	두 번째 피연산자의 값에서 지정한 비트 수만큼 첫 번째 피연산자의 값을 오른쪽으로 이동하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
<code>&=</code>	첫 번째 및 두 번째 피연산자의 비트 AND를 구하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
<code>^=</code>	첫 번째 및 두 번째 피연산자의 비트 배타적 OR를 구하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.
<code> =</code>	첫 번째 및 두 번째 피연산자의 비트 포함 OR를 구하여 첫 번째 피연산자가 지정한 개체에 결과를 저장합니다.

연산자 키워드

복합 할당 연산자 중 3개에는 해당하는 키워드가 있습니다. 화면은 다음과 같습니다.

[+] 테이블 확장

연산자	해당
<code>&=</code>	<code>and_eq</code>
<code> =</code>	<code>or_eq</code>
<code>^=</code>	<code>xor_eq</code>

C++는 복합 할당 연산자의 대체 맞춤법으로 이러한 연산자 키워드를 지정합니다. C에서 대체 맞춤법은 `<iso646.h>` 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. `<iso646.h>` 또는 C++ 동등한 `<ciso646>` 사용은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 [/permissive-](#) 또는 [/Za](#) 컴파일러 옵션이 필요합니다.

예시

```
C++

// exprre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
```

```

#include <iostream>
using namespace std;
int main() {
    int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

    a += b;          // a is 9
    b %= a;          // b is 6
    c >>= 1;         // c is 5
    d |= e;          // Bitwise--d is 0xFFFF

    cout << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
        << "a += b yields " << a << endl
        << "b %= a yields " << b << endl
        << "c >>= 1 yields " << c << endl
        << "d |= e yields " << hex << d << endl;
}

```

단순 할당

단순 대입 연산자(=)를 사용하면 두 번째 피연산자의 값이 첫 번째 피연산자가 지정한 개체에 저장됩니다. 두 개체가 모두 산술 형식인 경우 값을 저장하기 전에 오른쪽 피연산자를 왼쪽 형식으로 변환합니다.

`const` 및 `volatile` 형식의 개체는 오직 `volatile` 이거나 `const` 또는 `volatile` 가 아닌 형식의 | 값에 할당할 수 있습니다.

클래스 형식(`struct`, `union` 및 `class` 형식)의 개체에 대한 할당은 `operator=`라고 불리는 함수에 의해 수행됩니다. 이 연산자 함수의 기본 동작은 개체의 비정적 데이터 멤버 및 직접 기본 클래스의 멤버 단위 복사 할당을 수행하는 것입니다. 그러나 이 동작은 오버로드된 연산자를 사용하여 수정할 수 있습니다. 자세한 내용은 [연산자 오버로드](#)를 참조하세요. 클래스 형식에는 복사 할당 및 이동 할당 연산자가 있을 수 있습니다. 자세한 내용은 [복사 생성자 및 복사 할당 연산자](#) 및 [이동 생성자 및 이동 할당 연산자](#)를 참조하세요.

지정된 기본 클래스에서 명확히 파생된 모든 클래스의 개체는 기본 클래스의 개체에 할당될 수 있습니다. 파생 클래스에서 기본 클래스로의 암시적 변환이 있지만 기본 클래스에서 파생 클래스로 변환되지 않으므로 반대의 경우는 `true`가 아닙니다. 예시:

C++

```

// expre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }

```

```

};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}

```

참조 형식에 대한 할당은 참조가 가리키는 개체에 할당이 수행되는 것처럼 작동합니다.

클래스 형식 개체의 경우 할당은 초기화와 다릅니다. 할당과 초기화가 어떻게 다를 수 있는지를 알아보려면 다음 코드를 살펴보세요.

C++

```
UserType1 A;
UserType2 B = A;
```

위의 코드에서는 이니셜라이저를 보여 줍니다. 여기에서는 `UserType2` 형식의 인수를 사용하는 `UserType1`의 생성자를 호출합니다. 다음 코드에서

C++

```
UserType1 A;
UserType2 B;

B = A;
```

아래의 할당 문은

C++

```
B = A;
```

다음과 같은 결과 중 하나를 생성할 수 있습니다.

- `operator=(0)`가 `UserType1` 인수와 함께 제공된 경우 함수 `UserType2`에 대해 `operator=` 함수를 호출합니다.

- 명시적 변환 함수 `UserType1::operator UserType2`를 호출합니다(이러한 함수가 있는 경우).
- `UserType2::UserType2` 인수를 사용하고 결과를 복사하는 생성자 `UserType1`를 호출합니다(이러한 생성자가 있는 경우).

복합 할당

복합 할당 연산자는 [대입 연산자 테이블](#)에 표시됩니다. 이러한 연산자에는 $e1op = e2$ 양식이 있습니다. 여기서 $e1$ 은 수정할 수 `const` 없는 l-value이고 $e2$ 는 다음과 같습니다.

- 산술 형식
- op 가 `+` 또는 `-`인 경우 포인터
- $e1$ 형식에 대해 일치하는 `operator *op*=` 오버로드가 있는 형식입니다.

기본 제공 $e1op = e2$ 양식은 $e1 = e1ope2$ 으로 동작합니다. 그러나 $e1$ (은)는 한 번만 평가됩니다.

열거 형식에 대한 복합 할당은 오류 메시지를 생성합니다. 왼쪽 피연산자의 포인터 형식이면 오른쪽 피연산자는 포인터 형식이거나 0으로 계산되는 상수 식이어야 합니다. 왼쪽 피연산자는 정수 형식인 경우 오른쪽 피연산자는 포인터 형식이 아니어야 합니다.

기본 제공 할당 연산자의 결과

기본 제공 대입 연산자는 할당 후 왼쪽 피연산자가 지정한 개체의 값과 복합 할당 연산자의 경우 산술/논리 연산을 반환합니다. 결과 형식은 왼쪽 피연산자의 형식입니다. 할당 식의 결과는 항상 l-value입니다. 이러한 연산자는 오른쪽에서 왼쪽으로 결합됩니다. 왼쪽 피연산자는 수정 가능한 l-value여야 합니다.

ANSI C에서 할당 식의 결과는 l-value가 아닙니다. 즉, 법적 C++ 식 `(a += b) += c`(은)는 C에서 허용되지 않습니다.

참고 항목

[이항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 대입 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

비트 AND 연산자: &

아티클 • 2024. 07. 08.

구문

and-expression:

equality-expression

and-expression & *equality-expression*

설명

비트 AND 연산자(&)는 첫 번째 피연산자의 각 비트를 두 번째 피연산자의 해당 비트와 비교합니다. 양쪽 비트가 모두 1이면 해당 결과 비트는 1로 설정됩니다. 그렇지 않으면 해당 결과 비트는 0으로 설정됩니다.

비트 AND 연산자에 대한 두 피연산자는 모두 정수 형식을 가져야 합니다. 표준 변환이 적용되는 일반적인 산술 변환이 피연산자에 적용됩니다.

&에 대한 연산자 키워드

C++에서는 &에 대한 대체 맞춤법으로 `bitand`을 지정합니다. C에서는 대체 맞춤법이 `<iso646.h>` 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. `<iso646.h>` 또는 C++에 해당하는 `<ciso646>`은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 `/permissive-` 또는 `/Za` 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;           // pattern 1100 ...
    unsigned short b = 0xAAAA;           // pattern 1010 ...
```

```
    cout << hex << ( a & b ) << endl; // prints "8888", pattern 1000 ...  
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 비트 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

비트 배타적 OR 연산자: ^

아티클 • 2024. 07. 15.

구문

expression ^ *expression*

설명

비트 배타적 OR 연산자(^)는 첫 번째 피연산자의 각 비트를 두 번째 피연산자의 해당 비트와 비교합니다. 피연산자 중 하나의 비트가 0이고 다른 피연산자의 비트가 1이면 해당 결과 비트는 1로 설정됩니다. 그렇지 않으면 해당 결과 비트는 0으로 설정됩니다.

연산자에 대한 두 피연산자 모두 정수 형식이 있어야 합니다. 표준 변환이 적용되는 일반적인 산술 변환이 피연산자에 적용됩니다.

C++/CLI 및 C++/CX에서 ^ 문자의 대체 사용에 대한 자세한 내용은 [Handle to Object Operator\(^\)\(C++/CLI 및 C++/CX\)](#)을 참조하세요.

^ 연산자 키워드

C++에서는 ^에 대한 대체 맞춤법으로 xor을 지정합니다. C에서는 대체 맞춤법이 <iso646.h> 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. <iso646.h> 또는 C++에 해당하는 <ciso646>은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 /permissive- 또는 /Za 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0x5555;      // pattern 0101 ...
    unsigned short b = 0xFFFF;      // pattern 1111 ...
```

```
cout << hex << ( a ^ b ) << endl; // prints "aaaa" pattern 1010 ...  
}
```

참고 항목

C++ 기본 제공 연산자, 우선 순위 및 결합성

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

포괄적 비트 OR 연산자: |

아티클 • 2024. 07. 15.

구문

`expression1 | expression2`

설명

포괄적 비트 OR 연산자(|)는 첫 번째 피연산자의 각 비트를 두 번째 피연산자의 해당 비트와 비교합니다. 어느 한쪽 비트가 1이면 해당 결과 비트는 1로 설정됩니다. 그렇지 않으면 해당 결과 비트는 0으로 설정됩니다.

연산자에 대한 두 피연산자 모두 정수 형식이 있어야 합니다. 표준 변환이 적용되는 일반적인 산술 변환이 피연산자에 적용됩니다.

|에 대한 연산자 키워드

C++에서는 |에 대한 대체 맞춤법으로 `bitor`을 지정합니다. C에서는 대체 맞춤법이 `<iso646.h>` 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. `<iso646.h>` 또는 C++에 해당하는 `<ciso646>`은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 `/permissive-` 또는 `/Za` 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
    unsigned short a = 0x5555;          // pattern 0101 ...
    unsigned short b = 0xFFFF;          // pattern 1111 ...
    cout << hex << ( a | b ) << endl; // prints "ffff" pattern 1111 ...
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 비트 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

캐스트 연산자: ()

아티클 • 2024. 07. 15.

형식 캐스팅은 특정 상황에서의 개체 형식에 대한 명시적 변환을 위한 메서드를 제공합니다.

구문

```
cast-expression:  
    unary-expression  
    ( type-name ) cast-expression
```

설명

모든 단항 식은 캐스트 식으로 간주됩니다.

형식 캐스팅이 만들어지면 컴파일러에서 `cast-expression` 을 `type-name` 형식으로 처리합니다. 모든 스칼라 형식의 개체로 또는 다른 모든 스칼라 형식에서 변환하는 데 캐스트를 사용할 수 있습니다. 명시적 형식 캐스트는 암시적 변환 결과를 확인하는 동일한 규칙으로 제한됩니다. 캐스팅 관련 제한은 특정 형식의 실제 크기 또는 표현에서 발생할 수 있습니다.

예제

기본 제공 형식 간의 표준 캐스트 변환:

C++

```
// expre_CastOperator.cpp  
// compile with: /EHsc  
// Demonstrate cast operator  
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    double x = 3.1;  
    int i;  
    cout << "x = " << x << endl;  
    i = (int)x; // assign i the integer part of x
```

```
    cout << "i = " << i << endl;
}
```

사용자 정의 형식으로 정의된 캐스트 연산자:

C++

```
// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
        m_nSize(nSize)
    {
        m_pStr = new char[sizeOfBuffer];

        strncpy_s(m_pStr, sizeOfBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
        m_pStr[m_nSize] = '\0';
        return(m_pStr);
    }

    enum
    {
        sizeOfBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
```

```
const char *kStr = "Excitinggg";
CountedAnsiString myStr(kStr, 8);

const char *pRaw = myStr.GetRawBytes();
printf_s("RawBytes truncated to 10 chars: %.10s\n", pRaw);

const char *pCast = myStr; // or (const char *)myStr;
printf_s("Casted Bytes: %s\n", pCast);

puts("Note that the cast changed the raw internal string");
printf_s("Raw Bytes after cast: %s\n", pRaw);
}
```

Output

```
RawBytes truncated to 10 chars: Exciting!!
Casted Bytes: Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast: Exciting
```

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[명시적 형식 변환 연산자: \(\)](#)

[캐스팅 연산자 \(C++\)](#)

[캐스트 연산자\(C\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

쉼표 연산자: ,

아티클 • 2024. 03. 11.

하나의 문이 예상되는 곳에서 두 문을 그룹화할 수 있도록 합니다.

구문

```
expression , expression
```

설명

쉼표 연산자는 왼쪽에서 오른쪽으로 결합됩니다. 두 식은 쉼표로 구분되며 왼쪽에서 오른쪽으로 계산됩니다. 오른쪽 피연산자는 항상 왼쪽 피연산자가 실행되고, 모든 파생 작용이 완료된 후에 실행됩니다.

쉼표는 함수 인수 목록과 같은 일부 컨텍스트에서 구분 기호로 사용될 수 있습니다. 쉼표를 구분 기호로 사용하는 경우와 연산자로 사용하는 경우는 완전히 다르기 때문에 둘을 혼동하지 않도록 주의해야 합니다.

`e1, e2` 식을 참조하십시오. 식의 형식과 값은 `e2`의 형식과 값이며, `e1`의 계산 결과는 폐기합니다. 결과는 오른쪽 피연산자가 l-value인 경우 l-value입니다.

쉼표가 일반적으로 구분 기호로 사용되는 경우(예: 함수에 대한 실제 인수 또는 집합체 인니셜라이저), 쉼표 연산자와 피연산자는 괄호로 묶어야 합니다. 예시:

C++

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

위 `func_one`에 대한 함수 호출에서 쉼표로 구분된 세 인수, `x`, `y + 2`, `z`가 전달됩니다. `func_two`에 대한 함수 호출에서 괄호는 컴파일러가 첫 번째 쉼표를 순차적 계산 연산자로 해석하도록 합니다. 이 함수 호출은 두 인수를 `func_two`에 전달합니다. 첫 번째 인수는 `(x--, y + 2)` 식의 값과 형식을 가진 순차적 계산 연산 `y + 2`의 결과이며, 두 번째 인수는 `z`입니다.

예시

C++

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
    int i = 10, b = 20, c= 30;
    i = b, c;
    printf("%i\n", i);

    i = (b, c);
    printf("%i\n", i);
}
```

Output

```
20
30
```

참고 항목

이항 연산자가 있는 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

순차적 확인 연산자

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

조건부 연산자: ?:

아티클 • 2023. 10. 12.

구문

```
expression ? expression : expression
```

설명

조건부 연산자(?)는 3개의 피연산자를 사용하는 3개의 연산자입니다. 조건 연산자는 다음과 같이 사용됩니다.

- 첫 번째 피연산자는 암시적으로 .로 **bool** 변환됩니다. 계속하기 전에 피연산자가 계산되고 의도하지 않은 모든 결과가 완료됩니다.
- 첫 번째 피연산자가 **true** (1)로 계산되면 두 번째 피연산자가 계산됩니다.
- 첫 번째 피연산자가 **false** (0)로 계산되면 세 번째 피연산자가 계산됩니다.

조건 연산자의 결과는 두 번째 또는 세 번째 피연산자가 계산된 결과입니다. 마지막 피연산자 2개 중 1개만 조건식에서 계산됩니다.

조건식은 오른쪽과 왼쪽이 연결되어 있습니다. 첫 번째 피연산자는 정수 계열 또는 포인터 형식이어야 합니다. 다음 규칙이 두 번째 피연산자와 세 번째 피연산자에 적용됩니다.

- 두 피연산자의 형식이 동일할 경우 결과도 해당 형식입니다.
- 두 피연산자가 산술 또는 열거형 형식인 경우 표준 [변환에서](#) 다루는 일반적인 산술 변환을 수행하여 공통 형식으로 변환합니다.
- 두 피연산자가 모두 포인터 형식이거나 하나는 포인터 형식이고 나머지 하나는 0으로 계산된 상수 식일 경우 공용 형식으로 변환하기 위해 포인터 변환이 수행됩니다.
- 두 피연산자가 모두 참조 형식일 경우 공용 형식으로 변환하기 위해 참조 변환이 수행됩니다.
- 두 피연산자가 모두 void 형식일 경우 공용 형식이 void 형식입니다.
- 두 피연산자의 형식이 동일한 사용자 정의 형식인 경우 공통 형식이 해당 형식입니다.

- 피연산자의 형식이 다르고 피연산자 중 하나 이상의 형식이 사용자 지정 형식인 경우 공통 형식을 결정하기 위한 언어 규칙이 사용됩니다. 아래의 경고를 참조하세요.

위의 목록에 없는 두 번째 피연산자와 세 번째 피연산자의 조합은 부적합합니다. 결과의 형식은 공용 형식이며, 두 번째 피연산자와 세 번째 피연산자가 같은 형식이고 둘 다 l-value일 경우 l-value입니다.

⚠ 경고

두 번째와 세 번째 피연산자의 형식이 같지 않으면 C++ 표준에 지정된 복합 형식 변환 규칙이 호출됩니다. 이러한 변환이 수행되면 임시 개체 생성 및 제거를 비롯한 예기치 않은 동작이 발생할 수 있습니다. 그러므로 조건부 연산자와 함께 사용자 정의 형식을 피연산자로 사용하지 않거나, 사용자 정의 형식을 사용하는 경우에는 명시적으로 각 피연산자를 공용 형식으로 캐스팅하는 것이 좋습니다.

예시

C++

```
// expr_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
    int i = 1, j = 2;
    cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)
[조건식 연산자](#)

delete 연산자 (C++)

아티클 • 2024. 07. 08.

메모리 블록을 할당 취소합니다.

구문

```
[::] delete cast-expression  
[::] delete [] cast-expression
```

설명

cast-expression 인수는 [new 연산자](#)를 사용하여 만든 개체에 이전에 할당된 메모리 블록에 대한 포인터여야 합니다. **delete** 연산자는 **void** 형식의 결과를 가지므로 값을 반환하지 않습니다. 예시:

C++

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

new로 할당되지 않은 개체에 대한 포인터에 **delete**를 사용하면 예측할 수 없는 결과가 발생합니다. 그러나 값이 0인 포인터에는 **delete**를 사용할 수 있습니다. 이 프로비전은 실패 시 **new**가 0을 반환할 때 실패한 **new** 작업의 결과를 삭제해도 해가 없음을 의미합니다. 자세한 내용은 [new 및 delete 연산자](#)를 참조하세요.

new 및 **delete** 연산자는 배열을 포함한 기본 제공 형식에도 사용할 수 있습니다. **pointer**가 배열을 가리키는 경우 **pointer** 앞에 빈 대괄호([])를 넣습니다.

C++

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

개체에 **delete** 연산자를 사용하면 메모리 할당이 해제됩니다. 개체가 삭제된 후에 포인터를 역참조하는 프로그램에서는 예기치 않은 결과나 충돌이 발생할 수 있습니다.

delete를 사용하여 C++ 클래스 개체에 대한 메모리 할당을 취소하는 경우 개체의 메모리 할당이 취소되기 전에 개체의 소멸자가 호출됩니다(개체에 소멸자가 있는 경우).

`delete` 연산자에 대한 피연산자가 수정 가능한 l-value인 경우 개체가 삭제된 후 해당 값이 정의되지 않습니다.

/sdl(추가 보안 확인 사용) 컴파일러 옵션이 지정된 경우 개체가 삭제된 후 `delete` 연산자의 피연산자가 잘못된 값으로 설정됩니다.

삭제 사용

단일 개체와 개체의 배열에 각각 사용되는 `delete` 연산자의 두 가지 구문 변형이 있습니다. 다음 코드 조각은 이들의 차이점을 보여 줍니다.

C++

```
// expre_Using_delete.cpp
struct UDTtype
{
};

int main()
{
    // Allocate a user-defined object, UDObject, and an object
    // of type double on the free store using the
    // new operator.
    UDTtype *UDObject = new UDTtype;
    double *dObject = new double;
    // Delete the two objects.
    delete UDObject;
    delete dObject;
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDTtype (*UDArr)[7] = new UDTtype[5][7];
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;
}
```

개체에 대해 `delete`의 배열 형태(`delete []`)를 사용하는 경우와 배열에 대해 `delete`의 비 배열 형태를 사용하는 경우에는 정의되지 않은 결과가 생성됩니다.

예시

`delete` 사용 예는 [new 연산자](#)를 참조하세요.

delete 작동 방식

`delete` 연산자는 `operator delete` 함수를 호출합니다.

클래스 형식이 아닌 개체의 경우([class](#), [struct](#) 또는 [union](#)), 전역 delete 연산자가 호출됩니다. 클래스 형식 개체의 경우에는 삭제 식이 단항 범위 확인 연산자(::)로 시작되면 할당 해제 함수의 이름이 전역 범위에서 확인됩니다. 그렇지 않으면 delete 연산자가 메모리 할당을 해제하기 전에 개체에 대한 소멸자를 호출합니다(포인터가 null이 아닌 경우). delete 연산자는 클래스별로 정의될 수 있습니다. 지정된 클래스에 이러한 정의가 없는 경우 전역 delete 연산자가 호출됩니다. 삭제 식을 사용하여 정적 형식이 가상 소멸자인 클래스 개체를 할당 해제하는 경우, 할당 해제 함수는 개체의 동적 형식에 대한 가상 소멸자를 통해 확인됩니다.

참고 항목

[단항 연산자가 있는 식](#)

[키워드](#)

[new 및 delete 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

같음 연산자: `==` 및 `!=`

아티클 • 2024. 08. 13.

구문

expression `==` expression

expression `!=` expression

설명

같은 연산자(`==`)는 두 피연산자의 값이 같으면 반환하고, 그렇지 않으면 `false` 반환 `true` 됩니다.

피연산자에 같은 값이 없으면 같지 않은 연산자(`!=`)가 반환됩니다. 그렇지 않으면 `false` 반환 `true` 됩니다.

C 및 C++ `not_eq`에서는 대신 사용할 `!=` 수 있습니다. 자세한 내용은 [not-eq](#)를 참조하세요.

예시

C++

```
#include <iostream>

int main()
{
    int x = 1, y = 1, z = 2;

    if (x == y)
    {
        std::cout << "Equal\n";
    }

    if (x != z)
    {
        std::cout << "Not equal\n";
    }
}
```

출력

Equal
Not equal

참고 항목

[not-eq](#)

[연산자 오버로드](#)

[이항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 관계 및 같음 연산자](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

명시적 형식 변환 연산자: ()

아티클 • 2024. 03. 06.

C++에서는 함수 호출 구문과 유사한 구문을 사용하여 명시적인 형식 변환을 수행할 수 있습니다.

구문

```
simple-type-name ( expression-list )
```

설명

simple-type-name 뒤에 괄호로 묶인 *expression-list*이 지정된 식을 사용하여 지정된 형식의 개체를 생성합니다. 다음 예제에서는 int 형식으로의 명시적 형식 변환을 보여 줍니다.

C++

```
int i = int( d );
```

다음 예제에서는 `Point` 클래스를 보여줍니다.

예시

C++

```
// expe_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
```

```

        unsigned& y() { return _y; }
    void Show() { cout << "x = " << _x << ", "
                  << "y = " << _y << "\n"; }

private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    // of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    // conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}

```

출력

Output

```

x = 20, y = 10
x = 0, y = 0

```

위의 예제에서는 상수를 사용한 명시적 형식 변환을 보여 주지만 개체에서 이러한 변환을 수행할 때도 동일한 기술을 사용할 수 있습니다. 다음 코드 조각에서 이를 보여 줍니다.

C++

```

int i = 7;
float d;

d = float( i );

```

"캐스트" 구문을 사용하여 명시적 형식 변환을 지정할 수도 있습니다. 캐스트 구문을 사용하여 다시 작성한 이전 예제는 다음과 같습니다.

C++

```
d = (float)i;
```

캐스트 스타일 변환과 함수 스타일 변환 모두 단일 값에서 변환할 때 동일한 결과를 생성합니다. 그러나 함수 스타일 구문에서는 변환을 위해 인수를 둘 이상 지정할 수 있습니다. 이 차이점은 사용자 정의 형식의 경우 중요합니다. `Point` 클래스 및 해당 변환을 고려해 보십시오.

C++

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};

...
Point pt = Point( 3, 10 );
```

함수 스타일 변환을 사용하는 앞의 예제에서는 두 값(x 값과 y 값)을 사용자 정의 형식 `Point`(을)를 변환하는 방법을 보여 줍니다.

⊗ 주의

명시적 형식 변환은 C++ 컴파일러의 기본 제공 형식 검사를 재정의하므로 신중하게 사용하십시오.

[캐스트](#) 표기법은 *simple-type-name* 없는 형식으로 변환하는 데 사용해야 합니다(예: 포인터 또는 참조 형식). *simple-type-name*을 사용하여 표현할 수 있는 형식으로 변환할 수 있습니다.

캐스트 내의 형식 정의는 올바르지 않습니다.

참고 항목

[후위 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

함수 호출 연산자: ()

아티클 • 2024. 03. 11.

함수 호출은 함수 또는 호출 가능한 개체로 계산되는 식과 함수 호출 연산자, ()에 따라 형성되는 일종의 *postfix-expression*입니다. 개체는 개체에 대한 함수 호출 의미 체계를 제공하는 *operator ()* 함수를 선언할 수 있습니다.

구문

postfix-expression:

postfix-expression (*argument-expression-list* opt)

설명

함수 호출 연산자에 대한 인수는 쉼표로 구분된 식 목록인 *argument-expression-list*에서 나옵니다. 이러한 식의 값은 함수에 인수로 전달됩니다. 인수 식 목록은 비워 둘 수 있습니다. C++17 이전에는 함수 식과 인수 식의 계산 순서가 지정되지 않았으며 어떤 순서로든 발생할 수 있습니다. C++17 이상에서는 함수 식이 인수 식이나 기본 인수 앞에 평가됩니다. 인수 식은 확정되지 않은 시퀀스로 계산됩니다.

postfix-expression(은)는 호출할 함수로 평가됩니다. 다음과 같은 여러 가지 형식을 사용할 수 있습니다.

- 현재 범위 또는 제공된 함수 인수의 범위에 표시되는 함수 식별자
- 함수, 함수 포인터, 호출 가능한 개체 또는 함수에 대한 참조로 계산되는 식입니다.
- 명시적 또는 묵시적 멤버 함수 접근자
- 멤버 함수에 대한 역참조 포인터입니다.

postfix-expression(은)는 오버로드된 함수 식별자 또는 오버로드된 멤버 함수 접근자일 수 있습니다. 오버로드 확인 규칙은 호출할 실제 함수를 결정합니다. 멤버 함수가 가상인 경우 호출 할 함수는 런타임에 결정됩니다.

몇 가지 선언 예제는 다음과 같습니다.

- T 형식을 반환하는 함수. 선언 예제:

C++

```
T func( int i );
```

- `T` 형식을 반환하는 함수의 포인터. 선언 예제:

```
C++
```

```
T (*func)( int i );
```

- `T` 형식을 반환하는 함수의 참조. 선언 예제:

```
C++
```

```
T (&func)(int i);
```

- `T` 형식을 반환하는 멤버 포인터 함수 역참조. 함수 호출 예제:

```
C++
```

```
(pObject->*pmf)();  
(Object.*pmf)();
```

예시

다음 예제에서는 3개의 인수로 표준 라이브러리 함수 `strcat_s`를 호출합니다.

```
C++
```

```
// expre_Function_Call_Operator.cpp  
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library name space
using namespace std;

int main()
{
    enum
    {
        sizeOfBuffer = 20
    };

    char s1[ sizeOfBuffer ] = "Welcome to ";
    char s2[ ] = "C++";

    strcat_s( s1, sizeOfBuffer, s2 );
}
```

```
    cout << s1 << endl;
}
```

Output

```
Welcome to C++
```

함수 호출 결과

함수가 참조 형식으로 선언되지 않는 한 함수 호출은 rvalue로 평가됩니다. 참조 반환 형식이 있는 함수는 lvalues로 평가됩니다. 이러한 함수는 다음과 같이 대입 문의 왼쪽에서 사용할 수 있습니다.

C++

```
// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y(); // Use y() as an r-value.

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
        << "y = " << ThePoint.y() << "\n";
}
```

앞의 코드는 `x` 및 `y`좌표를 나타내는 프라이빗 데이터 개체를 포함하는 `Point`라고 불리는 클래스를 정의합니다. 이러한 데이터 개체를 수정하고 해당 값을 검색해야 합니다. 이 프로그램은 이러한 클래스를 위한 여러 디자인 중 하나이며, `GetX`와 `SetX` 또는 `GetY`와 `SetY` 함수는 사용할 수 있는 디자인입니다.

클래스 형식, 클래스 형식에 대한 포인터 또는 클래스 형식에 대한 참조를 반환하는 함수는 멤버 선택 연산자에 대한 왼쪽 피연산자로 사용할 수 있습니다. 다음 코드는 유효합니다.

C++

```
// expre_Function_Results2.cpp
class A {
public:
    A() {}
    A(int i) {}
    int SetA( int i ) {
        return (I = i);
    }

    int GetA() {
        return I;
    }

private:
    int I;
};

A func1() {
    A a = 0;
    return a;
}

A* func2() {
    A *a = new A();
    return a;
}

A& func3() {
    A *a = new A();
    A &b = *a;
    return b;
}

int main() {
    int iResult = func1().GetA();
    func2()->SetA( 3 );
    func3().SetA( 7 );
}
```

함수를 재귀적으로 호출할 수 있습니다. 함수 선언에 대한 자세한 내용은 [함수](#)를 참조하세요. 관련 자료는 [번역 단위 및 연결](#)에 있습니다.

참고 항목

후위 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

함수 호출

피드백

이 페이지가 도움이 되었나요?

Yes

No

제품 사용자 의견 제공 | Microsoft Q&A에서 도움말 보기

간접 참조 연산자: *

아티클 • 2024. 03. 06.

구문

```
* cast-expression
```

설명

단항 간접 참조 연산자(*)는 포인터를 역참조합니다. 즉, 포인터 값을 l-value로 변환합니다. 간접 연산자의 피연산자는 형식에 대한 포인터여야 합니다. 간접 참조의 결과는 포인터 형식이 파생된 형식입니다. 이 문맥에서 사용되는 * 연산자의 의미는 이항 연산자로 사용될 때의 의미(곱셈)와 다릅니다.

피연산자가 함수를 가리키는 경우 결과는 함수 지정자입니다. 스토리지 위치를 가리키는 경우 결과는 스토리지 위치를 지정하는 l-value입니다.

포인터에 대한 포인터를 역참조하기 위해 간접 참조 연산자를 누적하여 사용할 수도 있습니다. 예시:

C++

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
    int n = 5;
    int *pn = &n;
    int **ppn = &pn;

    cout << "Value of n:\n"
        << "direct value: " << n << endl
        << "indirect value: " << *pn << endl
        << "doubly indirect value: " << **ppn << endl
        << "address of n: " << pn << endl
        << "address of n via indirection: " << *ppn << endl;
}
```

포인터 값이 잘못된 경우 결과가 정의되지 않습니다. 다음 목록은 포인터 값을 무효화하는 가장 일반적인 조건의 일부입니다.

- 포인터가 null 포인터입니다.
- 포인터가 참조 시 표시되지 않는 로컬 항목의 주소를 지정합니다.
- 포인터가 가리키는 개체의 형식에 대해 부적절하게 정렬된 주소를 지정합니다.
- 포인터가 실행 중인 프로그램에서 사용되지 않는 주소를 지정합니다.

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[주소 연산자: &](#)

[연산자 주소 및 간접 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

왼쪽 시프트 및 오른쪽 시프트 연산자: << 및 >>

아티클 • 2024. 03. 12.

비트 시프트 연산자는 정수 또는 열거형 형식 식의 비트를 오른쪽으로 이동하는 오른쪽 시프트 연산자(>>)와 비트를 왼쪽으로 이동하는 왼쪽 시프트 연산자(<<)입니다.¹

구문

```
shift-expression:  
    additive-expression  
    shift-expression << additive-expression  
    shift-expression >> additive-expression
```

설명

① 중요

다음 설명 및 예제는 x86 및 x64 아키텍처용 Windows에서 유효합니다. ARM 디바이스용 Windows에서는 왼쪽 시프트 및 오른쪽 시프트 연산자의 구현이 크게 다릅니다. 자세한 내용은 [Hello ARM](#) 블로그 게시물의 "Shift Operators" 섹션을 참조하세요.

왼쪽 시프트

왼쪽 시프트 연산자를 사용하면 `shift-expression`에 비트가 `additive-expression`에 의해 지정된 위치 수만큼 왼쪽으로 이동됩니다. 시프트 연산으로 비워진 비트 위치는 0으로 채워집니다. 왼쪽 시프트는 논리 시프트입니다(부호 비트를 포함해 끝에서 벗어나 이동한 비트는 무시됨). 비트 시프트 종류에 대한 자세한 내용은 [Bitwise 시프트](#)를 참조하세요.

다음 예제에서는 부호 없는 숫자를 사용하는 왼쪽 시프트 연산을 보여 줍니다. 예제에서는 값을 비트 집합으로 나타내어 비트에서 어떤 일이 발생하는지 보여 줍니다. 자세한 내용은 [비트 세트 클래스](#)를 참조하세요.

C++

```
#include <iostream>  
#include <set>
```

```

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};    // the bitset representation of 4
    cout << bitset1 << endl;    // 0b00000000'00001000

    unsigned short short2 = short1 << 1;      // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;    // 0b00000000'00001000

    unsigned short short3 = short1 << 2;      // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;    // 0b00000000'00010000
}

```

부호 비트가 영향을 받도록 부호 있는 숫자를 왼쪽 시프트하면 결과가 정의되지 않습니다. 다음 예제에서는 1비트를 왼쪽에서 부호 비트 위치로 이동하면 어떻게 되는지 보여집니다.

C++

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;    // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3);    // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl;    // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4);    // 4 left-shifted by 14 = 0
    cout << bitset4 << endl;    // 0b00000000'00000000
}

```

오른쪽 시프트

오른쪽 시프트 연산자를 사용하면 *shift-expression*에 비트 패턴이 *additive-expression*에 의해 지정된 수만큼 오른쪽으로 이동됩니다. 부호 없는 숫자의 경우 시프트 연산으로 비워진 비트 위치는 0으로 채워집니다. 부호 있는 숫자의 경우 부호 비트는 비워진 비트

위치를 채우는 데 사용됩니다. 즉, 숫자가 양수이면 0이 사용되고 숫자가 음수이면 1이 사용됩니다.

① 중요

부호 있는 음수의 오른쪽 시프트 결과는 구현에 따라 다릅니다. Microsoft C++ 컴파일러는 부호 비트를 사용하여 비워진 비트 위치를 채우지만 다른 구현에서도 그렇게 한다는 보장은 없습니다.

이 예제에서는 부호 없는 숫자를 사용하는 오른쪽 시프트 연산을 보여 줍니다.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl;      // 0b00000100'00000000

    unsigned short short12 = short11 >> 1;   // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl;      // 0b00000010'00000000

    unsigned short short13 = short11 >> 10;  // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl;      // 0b00000000'00000001

    unsigned short short14 = short11 >> 11;  // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl;      // 0b00000000'00000000
}
```

다음 예제에서는 부호 있는 양수를 사용하는 오른쪽 시프트 연산을 보여 줍니다.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;      // 0b00000100'00000000
```

```

short short2 = short1 >> 1; // 512
bitset<16> bitset2(short2);
cout << bitset2 << endl; // 0b00000010'00000000

short short3 = short1 >> 11; // 0
bitset<16> bitset3(short3);
cout << bitset3 << endl; // 0b00000000'00000000
}

```

다음 예제에서는 부호 있는 음수를 사용하는 오른쪽 시프트 연산을 보여 줍니다.

C++

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl; // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl; // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl; // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
    cout << bn4 << endl; // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl; // 0b11111111'11111111
}

```

시프트 및 승격

시프트 연산자 양쪽에 있는 식은 정수 계열 형식이어야 합니다. 정수 승격은 표준 변환에 설명된 규칙에 따라 수행됩니다. 결과의 형식은 승격된 *shift-expression* 형식과 동일합니다.

다음 예제에서는 `char` 형식의 변수가 `int`(으)로 승격됩니다.

C++

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1;    // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10;   // 99328
    cout << typeid(promoted2).name() << endl; // int
}
```

세부 정보

시프트 작업의 결과는 *additive-expression(0|)* 음수이거나 만약 *additive-expression(0|)*가 (승격된) *shift-expression* 비트 수보다 크거나 같은 경우 정의되지 않습니다. 만약 *additive-expression(0|)*가 0이면 교대 근무 작업이 수행되지 않습니다.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl;    // 0b00000000'00000000'00000000'00000100

    unsigned int int2 = int1 << -3; // C4293: '<<' : shift count negative
or too big, undefined behavior
    unsigned int int3 = int1 >> -3; // C4293: '>>' : shift count negative
or too big, undefined behavior
    unsigned int int4 = int1 << 32; // C4293: '<<' : shift count negative
or too big, undefined behavior
    unsigned int int5 = int1 >> 32; // C4293: '>>' : shift count negative
or too big, undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl;    // 0b00000000'00000000'00000000'00000100 (no
change)
}
```

각주

¹ 다음은 C++11 ISO 사양(INCITS/ISO/IEC 14882-2011[2012]), 섹션 5.8.2 및 5.8.3의 시프트 연산자 설명입니다.

$E1 \ll E2$ 의 값은 $E1$ 왼쪽 이동된 $E2$ 비트 위치입니다. 비워진 비트는 0으로 채워집니다. 만약 $E1$ 에 부호 없는 형식이 있는 경우 결과 값은 $E1 \times 2^{E2}$ 입니다. 결과 형식에서 나타낼 수 있는 최대 값 보다 한 개 더 줄어드는 모듈입니다. 그렇지 않으면 만약 $E1$ 에 부호 있는 형식과 음수가 아닌 값이 있고 $E1 \times 2^{E2}$ 가 결과 형식의 해당 부호 없는 형식으로 나타낼 수 있는 경우 결과 형식으로 변환된 해당 값이 결과 값입니다. 그렇지 않으면 동작이 정의되지 않았습니다.

$E1 \gg E2$ 의 값은 $E1$ 오른쪽 이동된 $E2$ 비트 위치입니다. 만약 $E1$ 에 부호 없는 형식이 있거나 $E1$ 에 부호 있는 형식과 음수가 아닌 값이 있는 경우 결과 값은 $E1/2^{E2}$ 몫의 정수 부분입니다. $E1$ 이 부호 있는 형식이고 음수인 경우 결과 값은 구현 시 정의됩니다.

참고 항목

이항 연산자가 있는 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

논리 AND 연산자: `&&`

아티클 • 2024. 11. 21.

구문

:

```
equality-expression
logical-and-expression && equality-expression
```

설명

논리 AND 연산자(`&&`)는 두 피연산자가 모두 `true`이면 `true`를 반환하고 그렇지 않으면 `false`를 반환합니다. 피연산자는 평가 전에 암시적으로 `bool` 형식으로 변환되며 결과는 `bool` 형식입니다. 논리 AND에는 왼쪽에서 오른쪽으로의 결합성이 있습니다.

논리 AND 연산자에 대한 피연산자의 형식은 동일할 필요는 없지만 부울, 정수 또는 포인터 형식이어야 합니다. 피연산자는 일반적으로 관계형 또는 동등 식입니다.

첫째 피연산자가 완전히 계산되고 의도하지 않은 모든 결과가 논리 AND 식의 계산을 계속하기 전에 완료됩니다.

두 번째 피연산자는 첫 번째 피연산자가 `true`(0이 아님)으로 평가되는 경우에만 평가됩니다. 이 계산으로 인해 논리 AND 식이 `false` 일 때 둘째 피연산자의 불필요한 계산이 배제됩니다. 다음 예제와 같이 이 단락(short-circuit) 계산을 사용하여 null 포인터 역참조를 방지할 수 있습니다.

C++

```
char *pch = 0;
// ...
(pch) && (*pch = 'a');
```

`pch` 가 `null(0)`인 경우 식의 오른쪽이 계산되지 않습니다. 이 단락 평가는 널 포인터를 통한 할당을 불가능하게 만듭니다.

`&&`에 대한 연산자 키워드

C++에서는 `&&`에 대한 대체 맞춤법으로 `and`을 지정합니다. C에서는 대체 맞춤법이 `<iso646.h>` 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다.

<iostream> 또는 C++에 해당하는 <ciso646>은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 /permissive- 또는 /Za 컴파일러 옵션이 필요합니다.

예시

```
C++  
  
// expre_Logical_AND_Operator.cpp  
// compile with: /EHsc  
// Demonstrate logical AND  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int a = 5, b = 10, c = 15;  
    cout << boolalpha  
        << "The true expression "  
        << "a < b && b < c yields "  
        << (a < b && b < c) << endl  
        << "The false expression "  
        << "a > b && b < c yields "  
        << (a > b && b < c) << endl;  
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 논리 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

논리 부정 연산자: !

아티클 • 2024. 03. 06.

구문

`!cast-expression`

설명

논리 부정 연산자(`!`)는 해당 피연산자의 의미를 반대로 바꿉니다. 피연산자는 산술 형식, 포인터 형식 또는 산술/포인터 형식으로 계산되는 식이어야 합니다. 피연산자는 암시적으로 `bool` 형식으로 변환됩니다. 변환된 피연산자가 `false`이면 결과는 `true`입니다. 변환된 피연산자가 `true`이면 결과는 `false`입니다. 결과는 `bool` 형식입니다.

식 `e`의 경우 단항 식 `!e`는 오버로드된 연산자가 관련된 경우를 제외하고 식 `(e == 0)`과 동등합니다.

!에 대한 연산자 키워드

C++에서는 `!`에 대한 대체 맞춤법으로 `not`을 지정합니다. C에서는 대체 맞춤법이 `<iso646.h>` 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. `<iso646.h>` 또는 C++에 해당하는 `<ciso646>`은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 `/permissive-` 또는 `/Za` 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

참고 항목

단항 연산자가 있는 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

단항 산술 연산자

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공 [↗](#) | Microsoft Q&A에서 도움말 보기

논리적 OR 연산자: ||

아티클 • 2024. 07. 15.

구문

logical-or-expression || *logical-and-expression*

설명

논리적 OR 연산자(||)는 피연산자 중 하나 또는 둘 모두가 `true`이면 부울 값 `true`를 반환하고 그렇지 않으면 `false`를 반환합니다. 피연산자는 평가 전에 암시적으로 `bool` 형식으로 변환되며 결과는 `bool` 형식입니다. 논리 OR은 왼쪽에서 오른쪽으로의 결합성을 가집니다.

논리적 OR 연산자의 피연산자는 동일한 형식일 필요는 없지만 부울, 정수 또는 포인터 형식이어야 합니다. 피연산자는 일반적으로 관계형 또는 동등 식입니다.

첫 번째 피연산자가 완전히 계산되고 모든 파생 작업이 완료된 후에 논리 OR 계산이 시작됩니다.

두 번째 피연산자는 첫 번째 피연산자가 `false`로 평가되는 경우에만 평가됩니다. 논리적 OR 식이 `true`이면 평가가 필요하지 않기 때문입니다. 이를 단락 평가라고 합니다.

C++

```
printf( "%d" , (x == w || x == y || x == z) );
```

위의 예에서 `x`가 `w`, `y` 또는 `z`와 같은 경우 `printf` 함수에 대한 두 번째 인수는 `true`로 평가된 다음 정수로 승격됩니다. 값 1이 인쇄됩니다. 그렇지 않으면 `false`로 평가되고 값 0이 인쇄됩니다. 조건 중 하나가 `true`로 평가되는 즉시 평가가 중지됩니다.

||에 대한 연산자 키워드

C++에서는 ||에 대한 대체 맞춤법으로 `or`을 지정합니다. C에서는 대체 맞춤법이 <iso646.h> 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. <iso646.h> 또는 C++에 해당하는 <ciso646>은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 `/permissive-` 또는 `/Za` 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
        << "The true expression "
        << "a < b || b > c yields "
        << (a < b || b > c) << endl
        << "The false expression "
        << "a > b || b > c yields "
        << (a > b || b > c) << endl;
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 논리 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

멤버 액세스 연산자: . 및 ->

아티클 • 2024. 11. 21.

구문

:

```
postfix-expression . templateopt id-expression  
postfix-expression -> templateopt id-expression
```

설명

- . 및 -> 멤버 액세스 연산자는 `struct`, `union` 및 `class` 형식의 멤버를 참조하는 데 사용됩니다. 멤버 액세스 식에는 선택한 멤버의 값과 형식이 있습니다.

다음 두 가지 형태의 멤버 액세스 식이 있습니다.

- 첫 번째 모양에서 `postfix-expression`은 `struct`, `class` 또는 `union` 형식의 값을 나타내고, `id-expression`은 지정된 `struct`, `union` 또는 `class` 멤버의 이름을 지정합니다. 연산값은 `id-expression`의 값이며 `postfix-expression`이 l-value인 경우 l-value입니다.
- 두 번째 모양에서 `postfix-expression`은 `struct`, `union` 또는 `class`에 대한 포인터를 나타내고 `id-expression`은 지정된 `struct`, `union` 또는 `class` 멤버의 이름을 지정합니다. 값은 `id-expression`의 값이며 l-value입니다. -> 연산자는 포인터를 역참조합니다. `e->member` 및 `(*(e)).member` 식(`e`가 포인터를 나타냄)은 동일한 결과를 생성합니다(연산자 -> 또는 * 가 오버로드되는 경우 제외).

예시

다음 예제에서는 두 가지 형태의 멤버 액세스 연산자를 보여 줍니다.

C++

```
// expre_Selection_Operator.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
struct Date {  
    Date(int i, int j, int k) : day(i), month(j), year(k){}  
    int month;
```

```
int day;
int year;
};

int main() {
    Date mydate(1,1,1900);
    mydate.month = 2;
    cout << mydate.month << "/" << mydate.day
        << "/" << mydate.year << endl;

    Date *mydate2 = new Date(1,1,2000);
    mydate2->month = 2;
    cout << mydate2->month << "/" << mydate2->day
        << "/" << mydate2->year << endl;
    delete mydate2;
}
```

Output

```
2/1/1900
2/1/2000
```

참고 항목

후위 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

클래스 및 구조체

구조체 및 공용 구조체 멤버

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

곱하기 연산자 및 나머지 연산자

아티클 • 2024. 03. 06.

구문

```
expression * expression  
expression / expression  
expression % expression
```

설명

곱하기 연산자는 다음과 같습니다.

- 곱하기(*)
- 나누기(/)
- 모듈러스(나누기에서 나머지) (%)

이 이항 연산자는 왼쪽에서 오른쪽으로 연결됩니다.

곱하기 연산자는 산술 형식의 피연산자를 사용합니다. 모듈러스 연산자(%)는 피연산자가 정수 형식이어야 한다는 엄격한 요구 사항을 가지고 있습니다. (부동 소수점 나누기의 나머지 부분을 얻으려면 런타임 함수를 사용하여 [fmod\(\)](#)) 표준 변환이 적용되는 변환은 피연산자에 적용되고 결과는 변환된 형식입니다.

곱하기 연산자는 첫 번째 피연산자와 두 번째 피연산자를 곱한 결과를 구합니다.

나누기 연산자는 첫 번째 피연산자를 두 번째 피연산자로 나눈 결과를 구합니다.

모듈러스 연산자는 다음 식에서 지정된 나머지를 생성합니다. 여기서 $e1$ 은 첫 번째 피연산자이고 $e2$ 는 두 번째 피연산자입니다. $e1 - (e1 / e2) * e2$ 두 피연산자는 모두 정수 형식입니다.

0으로 나누기는 나누기 또는 모듈러스 식에 정의되지 않았으며 런타임 오류를 생성합니다. 따라서 다음 식에서는 정의되지 않은 잘못된 결과가 생성됩니다.

C++

```
i % 0
```

```
f / 0.0
```

곱하기, 나누기 또는 모듈러스 식의 두 피연산자 모두 부호가 같고 결과는 양수입니다. 그렇지 않으면 결과는 음수입니다. 모듈러스 연산 부호의 결과는 구현에서 정의됩니다.

① 참고

곱셈 연산자로 수행된 변환은 오버플로 또는 언더플로 조건을 제공하지 않으므로 변환 후 곱셈 연산 결과가 피연산자 형식으로 표현되지 않는 경우 정보가 손실될 수 있습니다.

Microsoft 전용

Microsoft C++에서 모듈러스 식의 결과가 항상 첫 번째 피연산자의 부호와 같습니다.

Microsoft 전용 종료

두 정수의 나누기 계산이 정확하지 않고 피연산자가 한 개만 음수일 경우 나누기 연산에서 구하는 정확한 값보다 작은 최대 정수(부호에 관계 없는 크기)가 결과가 됩니다. 예를 들어 $-11/3$ 의 계산 값은 -3.666666666 입니다. 해당 정수 분할의 결과는 -3 입니다.

곱하기 연산자 간의 관계는 $ID(e1 / e2) * e2 + e1 \% e2 == e1$ 로 지정됩니다.

예시

다음 프로그램은 곱셈 연산자를 보여 줍니다. 두 피연산자 모두 나누기 전에 `float` 형식이 되도록 잘림을 방지하려면 `10 / 3` 피연산자 중 하나를 형식 `float` 명시적으로 캐스팅해야 합니다.

C++

```
// expr_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    int x = 3, y = 6, z = 10;
    cout << "3 * 6 is " << x * y << endl
        << "6 / 3 is " << y / x << endl
        << "10 % 3 is " << z % x << endl
        << "10 / 3 is " << (float) z / x << endl;
}
```

참고 항목

[이항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 곱하기 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

new 연산자(C++)

아티클 • 2024. 07. 15.

지정된 형식이나 자리 표시자 형식의 개체 또는 개체의 배열에 대한 할당과 초기화를 시도하고, 개체(또는 배열의 첫 번째 개체)에 대한 적절한 형식의 0이 아닌 포인터를 반환합니다.

구문

new-expression:

```
:: opt new new-placement opt new-type-id new-initializer opt  
:: opt new new-placement opt ( type-id ) new-initializer opt
```

new-placement:

```
( expression-list )
```

new-type-id:

```
type-specifier-seq new-declarator opt
```

new-declarator:

```
ptr-operator new-declarator opt  
noptr-new-declarator
```

noptr-new-declarator:

```
[ expression ] attribute-specifier-seq opt  
noptr-new-declarator [ constant-expression ] attribute-specifier-seq opt
```

new-initializer:

```
( opt expression-list )  
braced-init-list
```

설명

실패하면 `new`는 0을 반환하거나 예외를 `throw`합니다. 자세한 내용은 [new 및 delete 연산자를 참조하세요](#). 사용자 지정 예외 처리 루틴을 작성하고 함수 이름을 인수로 사용하여 `_set_new_handler` 런타임 라이브러리 함수를 호출하면 이 기본 동작을 변경할 수 있습니다.

C++/CLI 및 C++/CX에서 관리되는 힙에 개체를 만드는 방법에 대한 자세한 내용은 [gcnew](#)를 참조하세요.

① 참고

Microsoft C++ 구성 요소 확장(C++/CX)은 vtable 슬롯 항목을 추가하기 위해 `new` 키워드에 대한 지원을 제공합니다. 자세한 내용은 [new\(vtable의 new 슬롯\)](#)를 참조하세요.

`new` 가 C++ 클래스 개체에 대한 메모리를 할당하는 데 사용되는 경우, 메모리가 할당된 후 개체의 생성자가 호출됩니다.

`new` 연산자로 할당한 메모리에 대한 할당을 취소하려면 `delete` 연산자를 사용하세요.

`new` 연산자로 할당한 배열을 삭제하려면 `delete[]` 연산자를 사용하세요.

다음 예제에서는 크기가 `dim`의 10배인 2차원 문자 배열을 할당한 다음 해제합니다. 다차원 배열을 할당할 때 첫 번째를 제외한 모든 차원은 양수 값으로 계산되는 상수 식이어야 합니다. 가장 왼쪽 배열의 차원은 양수 값으로 계산되는 임의의 식일 수 있습니다. `new` 연산자를 사용하여 배열을 할당할 때 첫 번째 차원은 0이 될 수 있으며, `new` 연산자는 고유 포인터를 반환합니다.

C++

```
char (*pchar)[10] = new char[dim][10];
delete [] pchar;
```

`type-id`은 `const`, `volatile`, 클래스 선언 또는 열거형 선언을 포함할 수 없습니다. 다음 식의 형식은 잘못되었습니다.

C++

```
volatile char *vch = new volatile char[20];
```

참조 형식은 개체가 아니므로 `new` 연산자는 참조 형식을 할당하지 않습니다.

`new` 연산자를 사용하여 함수를 할당할 수는 없지만, 함수에 대한 포인터를 할당할 수는 있습니다. 다음 예제에서는 정수를 반환하는 함수에 대한 7개의 포인터 배열을 할당한 다음 해제합니다.

C++

```
int (**p) () = new (int (*[7]) ());
delete p;
```

추가 인수 없이 `new` 연산자를 사용하고 `/GX`, `/EHs`, 또는 `/EHsc` 옵션을 사용하여 컴파일하는 경우, 생성자가 예외를 `throw`하면 컴파일러는 `delete` 연산자를 호출하는 코드를 생성합니다.

다음 목록에서는 `new`의 문법 요소에 대해 설명합니다.

`new-placement`

`new`를 오버로드하는 경우 추가 인수를 전달하는 방법을 제공합니다.

`type-id`

기본 제공 또는 사용자 정의 형식 중에서 할당할 형식을 지정합니다. 형식 사양이 복잡한 경우 괄호로 묶어 바인딩 순서를 강제로 지정할 수 있습니다. 형식은 컴파일러에 의해 형식이 결정되는 자리 표시자(`auto`)일 수 있습니다.

`new-initializer`

초기화된 개체의 값을 제공합니다. 배열에 대해 이니셜라이저는 지정할 수 없습니다. `new` 연산자는 클래스에 기본 생성자가 있는 경우에만 개체의 배열을 만듭니다.

`noptr-new-declarator`

배열의 범위를 지정합니다. 다차원 배열을 할당할 때 첫 번째를 제외한 모든 차원은 `std::size_t`로 변환 가능한 양수 값으로 계산되는 상수 식이어야 합니다. 가장 왼쪽 배열의 차원은 양수 값으로 계산되는 임의의 식일 수 있습니다. `attribute-specifier-seq`는 연결된 배열 형식에 적용됩니다.

예제: 문자 배열의 할당 및 해제

다음 코드 예제는 문자 배열 및 `CName` 클래스 개체를 할당한 다음 해제합니다.

C++

```
// expre_new_Operator.cpp
// compile with: /EHsc
#include <string.h>

class CName {
public:
    enum {
        sizeOfBuffer = 256
    };

    char m_szFirst[sizeOfBuffer];
    char m_szLast[sizeOfBuffer];

public:
    void SetName(char* pszFirst, char* pszLast) {
        strcpy_s(m_szFirst, sizeOfBuffer, pszFirst);
    }
}
```

```

        strcpy_s(m_szLast, sizeOfBuffer, pszLast);
    }

};

int main() {
    // Allocate memory for the array
    char* pCharArray = new char[CName::sizeOfBuffer];
    strcpy_s(pCharArray, CName::sizeOfBuffer, "Array of characters");

    // Deallocate memory for the array
    delete [] pCharArray;
    pCharArray = NULL;

    // Allocate memory for the object
    CName* pName = new CName;
    pName->SetName("Firstname", "Lastname");

    // Deallocate memory for the object
    delete pName;
    pName = NULL;
}

```

예제: `new` 연산자

`new` 연산자의 배치 형식(크기보다 더 많은 인수가 있는 형식)을 사용하는 경우, 생성자가 예외를 `throw`하면 컴파일러는 `delete` 연산자의 배치 형식을 지원하지 않습니다. 예시:

C++

```

// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
    A(int) { throw "Fail!"; }
};

void F(void) {
    try {
        // heap memory pointed to by pa1 will be deallocated
        // by calling ::operator delete(void*).
        A* pa1 = new A(10);
    } catch (...) {
    }
    try {
        // This will call ::operator new(size_t, char*, int).
        // When A::A(int) does a throw, we should call
        // ::operator delete(void*, char*, int) to deallocate
        // the memory pointed to by pa2. Since
        // ::operator delete(void*, char*, int) has not been implemented,
        // memory will be leaked when the deallocation can't occur.
    }
}

```

```

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

new로 할당된 개체의 초기화

선택적 `new-initializer` 필드는 `new` 연산자의 문법에 포함됩니다. 이 필드를 사용하면 사용자 정의 생성자로 새 개체를 초기화할 수 있습니다. 초기화 수행 방법에 대한 자세한 내용은 [이니셜라이저](#)를 참조하세요. 다음 예제에서는 `new` 연산자로 초기화 식을 사용하는 방법을 보여 줍니다.

C++

```

// expt_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    // an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }

private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double { 43.0 };
    // ...
}

```

이 예제에서 `CheckingAcct` 개체는 `new` 연산자를 사용하여 할당되지만 기본 초기화는 지정되지 않습니다. 그래서 클래스의 기본 생성자인 `Acct()`가 호출됩니다. 그런 다음, `SavingsAcct` 개체는 명시적으로 34.98로 초기화된다는 점을 제외하고 동일한 방식으로 할당됩니다. 34.98은 `double` 형식이므로, 해당 형식의 인수를 사용하는 생성자가 초기화를 처리하기 위해 호출됩니다. 마지막으로 비클래스 형식 `HowMuch`가 43.0으로 초기화됩니다.

개체가 클래스 형식이고 (앞의 예제에서처럼) 해당 클래스에 생성자가 있으면, 다음 조건 중 하나가 충족될 경우에만 `new` 연산자로 개체를 초기화할 수 있습니다.

- 이니셜라이저에서 제공된 인수가 생성자의 인수와 일치합니다.
- 클래스에 기본 생성자(인수 없이 호출할 수 있는 생성자)가 있습니다.

`new` 연산자를 사용하여 배열을 할당할 때 요소마다 명시적으로 초기화할 수 없으며, 기본 생성자(있을 경우)만 호출됩니다. 자세한 내용은 [기본 인수](#)를 참조하세요.

메모리 할당이 실패할 경우(`operator new`는 0 값 반환) 초기화가 수행되지 않습니다. 이러한 동작은 존재하지 않는 데이터를 초기화하려는 시도를 막습니다.

함수 호출과 마찬가지로, 초기화된 식이 계산되는 순서는 정의되지 않습니다. 또한, 메모리 할당이 이루어지기 전에 이러한 식이 완전히 계산될 것이라고 믿어서는 안 됩니다. 메모리 할당이 실패하고 `new` 연산자가 0을 반환하면, 이니셜라이저의 일부 식이 완전히 계산되지 않을 수도 있습니다.

new로 할당된 개체의 수명

`new` 연산자로 할당된 개체는 정의된 범위가 종료되어도 소멸되지 않습니다. `new` 연산자는 할당한 개체에 대한 포인터를 반환하기 때문에, 해당 개체에 대한 액세스 및 삭제를 위해 프로그램은 알맞은 범위의 포인터를 정의해야 합니다. 예시:

C++

```
// expr_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        // another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
    }

    delete [] AnotherArray; // Error: pointer out of scope.
    delete [] AnArray;     // OK: pointer still in scope.
}
```

`AnotherArray` 포인터가 예제 범위를 벗어나면 개체를 더 이상 삭제할 수 없습니다.

new 작동 방법

`new-expression (new)` 연산자가 포함된 식)은 다음 세 가지 작업을 수행합니다.

- 할당할 개체에 대한 스토리지를 찾고 예약합니다. 이 단계가 완료되면 정확한 양의 스토리지가 할당되지만, 해당 스토리지는 아직 개체가 아닙니다.
- 개체를 초기화합니다. 초기화가 완료되면 할당된 스토리지가 개체가 되는 데 충분한 정보가 있습니다.
- `new-type-id` 또는 `type-id`에서 파생된 포인터 형식의 개체(들)에 대한 포인터를 반환합니다. 프로그램에서는 이 포인터를 사용하여 새로 할당된 개체에 액세스합니다.

`new` 연산자는 `operator new` 함수를 호출합니다. 모든 형식의 배열과 `class`, `struct`, 또는 `union` 형식이 아닌 개체의 경우, 전역 함수인 `::operator new`가 스토리지를 할당하기 위해 호출됩니다. 클래스 형식 개체는 고유한 `operator new` 정적 멤버 함수를 클래스별로 정의할 수 있습니다.

컴파일러는 `T` 형식의 개체를 할당할 `new` 연산자를 발견하면 `T::operator new(sizeof(T))`를 호출하거나, 사용자 정의 `operator new`가 정의되지 않은 경우 `::operator new(sizeof(T))`를 호출합니다. 이러한 방식을 통해 `new` 연산자는 개체에 대해 정확한 양의 메모리를 할당할 수 있습니다.

① 참고

`operator new`에 대한 인수는 `std::size_t` 형식입니다. 이 형식은 `<direct.h>`, `<malloc.h>`, `<memory.h>`, `<search.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, 및 `<time.h>`에서 정의됩니다.

문법의 옵션을 사용하여 `new-placement`를 지정할 수 있습니다(`new` 연산자에 대한 문법 참조). `new-placement` 매개 변수는 `operator new`의 사용자 정의 구현에만 사용할 수 있으며, 이 매개 변수를 사용하면 `operator new`에 추가 정보를 전달할 수 있습니다. `T *TObject = new (0x0040) T;`와 같은 `new-placement` 필드가 포함된 식은 `T` 클래스에 `operator new` 멤버가 있으면 `T *TObject = T::operator new(sizeof(T), 0x0040);`로 변환되고, 그렇지 않으면 `T *TObject = ::operator new(sizeof(T), 0x0040);`로 변환됩니다.

`new-placement` 필드의 원래 용도는 하드웨어 종속 개체가 사용자 지정 주소에서 할당될 수 있도록 하는 것이었습니다.

① 참고

앞의 예제에서는 *new-placement* 필드에서 인수를 하나만 보여 주지만, 이런 식으로 **operator new**에 전달할 수 있는 추가 인수의 개수에는 제한이 없습니다.

operator new가 클래스 형식 `T`에 대해 정의된 경우에도, 다음 예제에서처럼 전역 연산자 **new**를 명시적으로 사용할 수 있습니다.

C++

```
T *TObject = ::new TObject;
```

범위 확인 연산자(::)는 강제로 전역 **new** 연산자를 사용하도록 합니다.

참고 항목

[단항 연산자가 있는 식](#)

[키워드](#)

[new 및 delete 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

보수 연산자: ~

아티클 • 2024. 03. 06.

구문

C++

~ cast-expression

설명

비트 보수 연산자라고도 하는 보수 연산자(~)는 피연산자의 비트 보수를 생성합니다. 즉, 피연산자의 1인 모든 비트는 결과적으로 0입니다. 반대로 피연산자의 0인 모든 비트는 결과적으로 1입니다. 1의 보수 연산자의 피연산자는 정수 계열 형식이어야 합니다.

~에 대한 연산자 키워드

C++에서는 ~에 대한 대체 맞춤법으로 **compl**을 지정합니다. C에서는 대체 맞춤법이 <iso646.h> 헤더에 매크로로 제공됩니다. C++에서 대체 맞춤법은 키워드입니다. <iso646.h> 또는 C++에 해당하는 <ciso646>은 더 이상 사용되지 않습니다. Microsoft C++에서는 대체 맞춤법을 사용하도록 설정하려면 /permissive- 또는 /Za 컴파일러 옵션이 필요합니다.

예시

C++

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
    unsigned short y = 0xFFFF;
    cout << hex << y << endl;
    y = ~y;      // Take one's complement
    cout << hex << y << endl;
}
```

이 예제에서 `y`에 할당된 새 값은 부호 없는 값인 0xFFFF의 1의 보수이거나 0x0000입니다.

정수 계열 확장은 정수 계열 피연산자를 대상으로 수행됩니다. 피연산자가 승격되는 형식은 결과 형식입니다. 정수 계열 승격에 대한 자세한 내용은 [표준 변환](#)을 참조하세요.

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[단항 산술 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

멤버 포인터 연산자: `.*` 및 `->*`

아티클 • 2024. 07. 15.

구문

pm-expression:

cast-expression

pm-expression `.*` *cast-expression*

pm-expression `->*` *cast-expression*

설명

멤버 포인터 연산자 `.*` 및 `->*`는 식의 왼쪽에 지정된 개체에 대한 특정 클래스 멤버의 값을 반환합니다. 오른쪽에는 클래스 멤버를 지정해야 합니다. 다음 예제에서는 이러한 연산자를 사용하는 방법을 보여 줍니다.

C++

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
    void m_func1() { cout << "m_func1\n"; }
    int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
    Testpm ATestpm;
    Testpm *pTestpm = new Testpm;

    // Access the member function
    (ATestpm.*pmfn)();
    (pTestpm->*pmfn)();    // Parentheses required since * binds
                           // less tightly than the function call.

    // Access the member data
```

```

ATestpm.*pmd = 1;
pTestpm->*pmd = 2;

cout << ATestpm.*pmd << endl
    << pTestpm->*pmd << endl;
delete pTestpm;
}

```

출력

Output

```

m_func1
m_func1
1
2

```

앞의 예제에서는 멤버 함수 `pmfn`을 호출하기 위해 `m_func1` 멤버에 대한 포인터가 사용되었습니다. 또 다른 멤버 포인터, `pmd`는 `m_num` 멤버에 액세스하는 데 사용됩니다.

`.*` 이항 연산자는 첫 번째 피연산자(반드시 클래스 형식의 개체)와 두 번째 피연산자(반드시 멤버 포인터 형식)를 결합합니다.

`->*` 이항 연산자는 첫 번째 피연산자(반드시 클래스 형식의 개체 포인터)와 두 번째 피연산자(반드시 멤버 포인터 형식)를 결합합니다.

`.*` 연산자가 포함된 식에서 첫 번째 피연산자는 두 번째 피연산자에 지정된 멤버 포인터의 클래스 형식이면서 해당 멤버 포인터에 액세스할 수 있거나 해당 클래스에서 명확히 파생되고 해당 클래스에 액세스할 수 있는 형식이어야 합니다.

`->*` 연산자가 포함된 식에서 첫 번째 피연산자는 두 번째 피연산자에 지정된 형식의 "클래스 형식 포인터" 형식이거나 해당 클래스에서 명확히 파생된 형식이어야 합니다.

예시

다음 클래스와 프로그램 부분을 살펴보십시오.

C++

```

// expre_Expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
    BaseClass(); // Base class constructor.
    void Func1();
}

```

```

};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
    Derived(); // Derived class constructor.
    void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)(); // OK: defined for BaseClass.
    (ABase.*pmfnFunc2)(); // Error: cannot use base class to
                          // access pointers to members of
                          // derived classes.

    (ADerived.*pmfnFunc1)(); // OK: Derived is unambiguously
                            // derived from BaseClass.
    (ADerived.*pmfnFunc2)(); // OK: defined for Derived.
}

```

.* 또는 ->* 멤버 포인터 연산자의 결과는 멤버 포인터 선언에 지정된 형식의 개체 또는 함수입니다. 따라서 앞의 예제에서 ADerived.*pmfnFunc1() 식의 결과는 void를 반환하는 함수 포인터입니다. 두 번째 피연산자가 l-value인 경우 이 결과는 l-value입니다.

① 참고

멤버 포인터 연산자 중 하나의 결과가 함수인 경우 결과는 함수 호출 연산자에 대한 피연산자로만 사용할 수 있습니다.

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

후위 증가 및 감소 연산자: `++` 및 `--`

아티클 • 2024. 03. 11.

구문

```
postfix-expression ++
postfix-expression --
```

설명

C++는 전위/후위 증가 및 감소 연산자를 제공합니다. 이 단원에서는 후위 증가 및 감소 연산자에 대해서만 설명합니다. (자세한 내용은 [전위 증가 및 감소 연산자](#)를 참조하십시오.) 후위 표기법과 전위 표기법의 차이는 후위 표기법에서는 연산자가 후위 식 뒤에 표시되지만 전위 표기법에서는 연산자가 식 앞에 표시된다는 점입니다. 다음 예제에서는 후위 증가 연산자를 보여 줍니다.

```
C++
```

```
i++;
```

후위 증가 연산자(`++`)를 적용하면 피연산자 값이 적절한 형식의 단위로 한 단위씩 증가하게 됩니다. 마찬가지로 후위 감소 연산자(`--`)를 적용하면 피연산자의 값은 적절한 형식의 단위로 한 단위씩 감소하게 됩니다.

주의해야 할 점은 후위 증가 또는 감소 식은 각 연산자를 적용하기 전에 식의 값으로 계산된다는 것입니다. 증가 또는 감소 연산은 피연산자가 계산된 후에 진행됩니다. 이 문제는 후위 증가나 감소 연산이 더 큰 수식의 컨텍스트에서 진행할 경우에만 발생합니다.

후위 연산자가 함수 인수에 적용될 때 해당 인수의 값은 함수에 전달된 후에만 증가하거나 감소됩니다. 자세한 내용은 C++ 표준의 1.9.17 단원을 참조하십시오.

후위 증가 연산자를 `long` 형식 개체의 배열 포인터에 적용하면 포인터의 내부 표현에 4가 실제로 추가됩니다. 이 동작은 이전에 n 번째 배열 요소를 참조한 포인터가 $(n+1)$ 번째 요소를 참조하도록 합니다.

후위 증가 및 감소 연산자의 피연산자는 `const` 형식이 아닌 산술 또는 포인터 형식의 변경 가능한 l-value여야 합니다. 결과의 형식은 후위 식 형식과 동일하지만 l-value가 될 수는 없습니다.

Visual Studio 2017 버전 15.3 이상([/std:c++17](#) 모드 이상에서 사용 가능): 후위 증가 또는 감소 연산자의 피연산자는 **bool** 형식이 아닐 수 있습니다.

다음 코드에서는 후위 증가 연산자에 대해 설명합니다.

C++

```
// expr_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    cout << i++ << endl;
    cout << i << endl;
}
```

후위 증가 및 감소 연산은 열거형 형식에서 지원되지 않습니다.

C++

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

참고 항목

[후위 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[C 후위 증가 및 감소 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

접두사 증가 및 감소 연산자: **++** 및 **--**

아티클 • 2024. 03. 06.

구문

```
++ unary-expression  
-- unary-expression
```

설명

전위 증가 연산자(**++**)는 피연산자를 1씩 증가시킵니다. 연산 결과는 증가된 값입니다. 피연산자는 **const** 형식이 아닌 l-value여야 합니다. 결과는 피연산자와 동일한 형식의 l-value입니다.

전위 감소 연산자(**--**)는 피연산자를 1씩 감소시켜 결과 값이 작아지는 것을 제외하고 전위 증가 연산자와 유사합니다.

Visual Studio 2017 버전 15.3 이상([/std:c++17](#) 모드 이상에서 사용 가능): 증가 또는 감소 연산자의 피연산자는 **bool** 형식이 아닐 수 있습니다.

전위 및 후위 증가 및 감소 연산자는 피연산자에 영향을 줍니다. 두 처리자의 주요 차이점은 식의 계산에서 증가 또는 감소가 수행되는 순서입니다. (자세한 내용은 [후위 증가 및 감소 연산자를 참조하세요](#).) 전위 형식에서는 식 계산에서 값이 사용되기 전에 증가 또는 감소가 발생하므로 식 같은 피연산자 값과 다릅니다. 후위 형식에서는 식 계산에서 값이 사용된 후에 증가 또는 감소가 발생하므로 식 같은 피연산자 값과 동일합니다. 예를 들어 다음 프로그램은 "**++i = 6**"을 인쇄합니다.

C++

```
// expre_Increment_and_Decrement_Operators.cpp  
// compile with: /EHsc  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int i = 5;  
    cout << "++i = " << ++i << endl;  
}
```

정수 계열 또는 부동 형식의 피연산자는 정수 값 1만큼 증가하거나 감소합니다. 결과 형식은 피연산자 형식과 동일합니다. 포인터 형식의 피연산자는 자신이 처리하는 개체의 크기만큼 증가하거나 감소합니다. 증가한 포인터는 다음 개체를 가리키고, 감소한 포인터는 이전 개체를 가리킵니다.

증가 및 감소 연산자는 부작용이 있기 때문에 [전처리기 매크로](#)에서 증가 또는 감소 연산자가 있는 식을 사용하면 원하지 않는 결과를 얻을 수 있습니다. 다음 예제를 고려해 보세요.

C++

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
    int i = 0, j = 0, k;
    k = max( ++i, j );
}
```

매크로는 다음과 같이 확장됩니다.

C++

```
k = ((++i)<(j))?(j):(++i);
```

`i`가 `j`보다 크거나 같거나 `j`보다 1만큼 작은 경우 두 번 증가합니다.

① 참고

이곳에서 설명한 것과 같은 부작용을 제거하고 언어를 사용하여 보다 완벽한 형식 검사를 수행할 수 있기 때문에 대부분 매크로보다는 C++ 인라인 함수를 사용하는 것이 좋습니다.

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[접두사 증가 및 감소 연산자](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

관계형 연산자: <, >, <=, 및 >=

아티클 • 2024. 03. 11.

구문

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

설명

이항 관계형 연산자는 다음과 같은 관계를 확인합니다.

- 미만(<)
- 보다 큼(>)
- 작거나 같음(<=)
- 크거나 같음(>=)

관계형 연산자는 왼쪽에서 오른쪽으로 결합됩니다. 관계형 연산자의 두 피연산자는 산술 또는 포인터 형식이어야 하며, 형식 `bool`의 값을 생성합니다. 식의 관계가 `false`이면 반환 값은 `false(0)`이고, 그렇지 않으면 반환 값은 `true(1)`입니다.

예시

C++

```
// expre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << "The true expression 3 > 2 yields: "
        << (3 > 2) << endl
        << "The false expression 20 < 10 yields: "
```

```
<< (20 < 10) << endl;  
}
```

스트림 삽입 연산자(`<<`)가 관계형 연산자보다 우선순위가 높기 때문에 위의 예제에 있는 식은 괄호로 묶여야 합니다. 따라서 괄호가 없는 첫 번째 식은 다음과 같이 평가됩니다.

C++

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

[표준 변환](#)에서 다루는 일반적인 산술 변환은 산술 형식의 피연산자에 적용됩니다.

포인터 비교

형식이 같은 개체의 포인터 2개를 비교할 때 프로그램의 주소 공간에서 가리키는 개체 위치에 따라 결과가 결정됩니다. 0 또는 `void *` 형식의 포인터로 계산되는 상수 식과 포인터를 비교할 수도 있습니다. `void *` 형식의 포인터에 대해 포인터를 비교할 경우 암시적으로 다른 포인터가 `void *` 형식으로 변환됩니다. 그런 다음 비교가 수행됩니다.

다음과 같은 경우에만 형식이 다른 두 포인터를 비교할 수 있습니다.

- 한 형식이 다른 형식에서 파생된 클래스 형식입니다.
- 하나 이상의 포인터가 `void *` 형식으로 명시적으로 변환됩니다. (다른 포인터는 변환을 위해 `void *` 형식으로 암시적으로 변환됩니다.)

형식이 같은 두 포인터가 같은 개체를 가리킬 경우 동일하다고 간주합니다. 개체의 비정적 멤버에 대한 두 포인터를 비교할 경우 다음 규칙이 적용됩니다.

- 클래스 형식이 `union` 이(가) 아닐 경우 그리고 `public`, `protected` 또는 `private` 등의 *access-specifier*로 두 멤버가 구분되지 않은 경우 마지막에 선언된 멤버의 포인터가 먼저 선언된 멤버의 포인터보다 크다고 간주합니다.
- 두 멤버가 *access-specifier*로 구분되면 결과가 정의되지 않습니다.
- 클래스 형식이 `union` 일 경우 해당 `union`의 여러 데이터 멤버에 대한 포인터가 같다고 간주합니다.

포인터 2개가 배열이 같은 요소를 가리키거나 배열의 끝을 벗어난 요소를 가리킬 경우 첨자가 높은 개체의 포인터가 더 높다고 간주합니다. 포인터가 같은 배열의 개체를 참조하거나 배열의 끝을 벗어난 위치를 참조할 경우에만 포인터 비교가 유효합니다.

참고 항목

이항 연산자가 있는 식

C++ 기본 제공 연산자, 우선 순위 및 결합성

C 관계 및 같음 연산자

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공 [↗](#) | Microsoft Q&A에서 도움말 보기

범위 확인 연산자: ::

아티클 • 2024. 11. 21.

범위 확인 연산자 `::`은 여러 범위에 사용된 식별자를 확인하고 구분하는 데 사용됩니다.
범위에 대한 자세한 내용은 [범위](#)를 참조하세요.

구문

```
:  
    nested-name-specifier opt template unqualified-id  
  
:  
    ::  
    type-name ::  
    namespace-name ::  
    decltype-specifier ::  
    nested-name-specifier identifier ::  
    nested-name-specifier template opt simple-template-id ::  
  
:  
    identifier  
    operator-function-id  
    conversion-function-id  
    literal-operator-id  
    ~ type-name  
    ~ decltype-specifier  
    template-id
```

설명

`identifier`에는 변수, 함수 또는 열거형 값이 해당됩니다.

클래스 및 네임스페이스에 :: 사용

다음 예제에서는 범위 확인 연산자가 네임스페이스 및 클래스에 사용되는 방법을 보여줍니다.

C++

```
namespace NamespaceA{
    int x;
    class ClassA {
        public:
            int x;
    };
}

int main() {

    // A namespace name used to disambiguate
    NamespaceA::x = 1;

    // A class name used to disambiguate
    NamespaceA::ClassA a1;
    a1.x = 2;
}
```

범위 한정자가 없는 범위 확인 연산자는 전역 네임스페이스를 참조합니다.

C++

```
namespace NamespaceA{
    int x;
}

int x;

int main() {
    int x;

    // the x in main()
    x = 0;
    // The x in the global namespace
    ::x = 1;

    // The x in the A namespace
    NamespaceA::x = 2;
}
```

범위 확인 연산자를 사용하여 `namespace`의 멤버를 식별하거나 `using` 지시문으로 멤버의 네임스페이스를 지명하는 네임스페이스를 식별할 수 있습니다. 아래 예에서는 `ClassB`가 `NamespaceB` 네임스페이스에서 선언되었더라도 `using` 지시문에 의해 `NamespaceB`가 `NamespaceC`에서 지명되었기 때문에 `NamespaceC`를 사용하여 `ClassB`에 자격을 줄 수 있습니다.

C++

```

namespace NamespaceB {
    class ClassB {
        public:
            int x;
    };
}

namespace NamespaceC{
    using namespace NamespaceB;
}

int main() {
    NamespaceB::ClassB b_b;
    NamespaceC::ClassB c_b;

    b_b.x = 3;
    c_b.x = 4;
}

```

범위 확인 연산자를 연쇄적으로 사용할 수 있습니다. 다음 예제에서
`NamespaceD::NamespaceD1`은 중첩된 네임스페이스 `NamespaceD1`을 식별하고
`NamespaceE::ClassE::ClassE1`은 중첩된 클래스 `ClassE1`을 식별합니다.

C++

```

namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{
        public:
            class ClassE1{
                public:
                    int x;
            };
    };
}

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7 ;
}

```

정적 멤버에 :: 사용

클래스의 정적 멤버를 호출하려면 범위 확인 연산자를 사용해야 합니다.

C++

```
class ClassG {
public:
    static int get_x() { return x; }
    static int x;
};

int ClassG::x = 6;

int main() {

    int gx1 = ClassG::x;
    int gx2 = ClassG::get_x();
}
```

범위가 지정된 열거형에 :: 사용

다음 예와 같이, 범위 확인 연산자를 범위가 지정된 열거형 [열거형 선언](#)의 값과 함께 사용할 수도 있습니다.

C++

```
enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}
```

참고 항목

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)
[네임스페이스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

sizeof 연산자

아티클 • 2024. 03. 06.

형식 `char`의 크기에 따라 피연산자의 크기를 결과로 생성합니다.

① 참고

`sizeof ...` 연산자에 대한 자세한 내용은 [줄임표 및 가변 템플릿](#)을 참조하세요.

구문

```
sizeof unary-expression  
sizeof ( type-name )
```

설명

`sizeof` 연산자의 결과는 포함 파일 <stddef.h>에 정의된 정수 형식인 `size_t` 형식에 속합니다. 이 연산자를 사용하면 프로그램에서 컴퓨터 종속 데이터 크기를 지정하지 않아도 됩니다.

`sizeof`의 피연산자는 다음 중 하나가 될 수 있습니다.

- 형식 이름. 형식 이름과 함께 `sizeof`를 사용하려면 이름이 괄호로 묶여 있어야 합니다.
- 식입니다. 식과 함께 사용하는 경우 `sizeof`는 괄호를 포함하거나 포함하지 않고 지정할 수 있습니다. 식은 계산되지 않습니다.

`sizeof` 연산자가 `char` 형식의 개체에 적용되면 1이 결과로 생성됩니다. `sizeof` 연산자가 배열에 적용되면 배열 식별자가 나타내는 포인터의 크기가 아닌 해당 배열의 전체 바이트 수가 결과로 생성됩니다. 배열 식별자가 나타내는 포인터의 크기를 구하려면 `sizeof`가 사용되는 함수에 매개 변수로 전달합니다. 예시:

예시

C++

```

#include <iostream>
using namespace std;

size_t getPtrSize( char *ptr )
{
    return sizeof( ptr );
}

int main()
{
    char szHello[] = "Hello, world!";

    cout << "The size of a char is: "
        << sizeof( char )
        << "\nThe length of " << szHello << " is: "
        << sizeof szHello
        << "\nThe size of the pointer is "
        << getPtrSize( szHello ) << endl;
}

```

샘플 출력

Output

```

The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4

```

sizeof 연산자가 **class**, **struct** 또는 **union** 형식에 적용되면 해당 형식 개체의 바이트 수와 함께 단어 경계에 멤버를 맞추기 위해 추가된 안쪽 여백이 결과로 생성됩니다. 결과는 개별 멤버의 스토리지 요구 사항을 추가하여 계산된 크기와 일치하지 않을 수도 있습니다. **/Zp** 컴파일러 옵션과 **pack** pragma는 멤버 맞춤 경계에 영향을 줍니다.

빈 클래스인 경우라도 **sizeof** 연산자는 0을 결과로 생성하지 않습니다.

sizeof 연산자는 다음의 피연산자와 함께 사용할 수 없습니다.

- 함수 (하지만, 함수의 포인터에는 **sizeof**를 적용할 수 없음)
- 비트 필드
- 정의되지 않은 클래스
- **void** 형식입니다.
- 동적으로 할당된 배열

- 외부 배열
- 불완전한 형식
- 괄호로 묶인 불완전한 형식의 이름

sizeof 연산자가 참조에 적용되는 경우 그 결과는 **sizeof** 가 개체 자체에 적용되었을 때와 같습니다.

크기가 지정되지 않은 배열이 구조체의 마지막 요소인 경우 **sizeof** 연산자는 배열 없는 구조체의 크기를 반환합니다.

sizeof 연산자는 대개 다음과 같은 형태의 식을 통해 배열 내 요소의 개수를 계산하는 데 사용됩니다.

C++

```
sizeof array / sizeof array[0]
```

참고 항목

[단항 연산자가 있는 식](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

아래 첨자 연산자: []

아티클 • 2024. 03. 06.

구문

```
postfix-expression [ expression ]
```

설명

후위 식(기본 식이 될 수도 있음) 뒤에 첨자 연산자 []가 오면 배열 인덱싱이 지정됩니다.

C++/CLI의 관리 배열에 대한 자세한 내용은 [배열](#)을 참조하세요.

일반적으로 *postfix-expression*으로 표현되는 값은 배열 식별자와 같은 포인터 값이며 *expression*은 열거 형식을 포함한 정수 값입니다. 그러나 구문적 요구 사항은 식 중 하나가 포인터 형식이고 다른 하나는 정수 계열 형식이어야 한다는 것 밖에 없습니다. 따라서 정수 값은 *postfix-expression* 위치에 있을 수 있으며 포인터 값은 *expression*의 괄호 안 또는 첨자 위치에 있을 수 있습니다. 다음과 같은 코드 조각을 생각해 봅시다.

C++

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;         // prints "2"
```

위의 예제에서 `nArray[2]` 식은 `2[nArray]` 와 동일합니다. 그 이유는 아래 첨자 식 `e1[e2]`의 결과가 다음과 같이 제공되기 때문입니다.

```
*((e2) + (e1))
```

식에서 생성된 주소는 `e1` 주소의 `e2` 바이트가 아닙니다. 대신 `e2` 배열의 다음 개체가 생성되도록 주소의 크기가 조정됩니다. 예시:

C++

```
double aDbl[2];
```

`aDb[0]` 및 `aDb[1]`의 주소는 8바이트 떨어져 있습니다(`double` 형식의 개체 크기). 개체 형식에 따른 이 크기 조정은 C++ 언어에 의해 자동으로 수행되며 포인터 형식 피연산자의 덧셈과 뺄셈에 대해 설명하는 [덧셈 연산자](#)에 정의되어 있습니다.

첨자 식에는 다음과 같이 여러 첨자가 있을 수도 있습니다.

expression1[expression2] [expression3] ...

첨자 식은 왼쪽에서 오른쪽으로 연결합니다. 맨 왼쪽 첨자 식인 *expression1[expression2]*가 먼저 계산됩니다. *expression1* 및 *expression2*를 추가한 결과인 주소는 포인터 식을 형성합니다. 그런 다음 이 포인터 식에 *expression3*이 추가되어 새 포인터 식을 형성하고 마지막 첨자 식이 추가될 때까지 계속됩니다. 최종 포인터 값이 배열 형식을 해결하지 않으면 [간접 참조 연산자\(*\)](#)는 마지막 첨자 식이 계산된 후 적용됩니다.

여러 첨자가 포함된 식은 다차원 배열의 요소를 참조합니다. 다차원 배열은 요소가 배열인 배열입니다. 예를 들어 3차원 배열의 첫 번째 요소는 2차원 배열입니다. 다음 예제에서는 간단한 2차원 문자 배열을 선언 및 초기화합니다.

C++

```
// exre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
    char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
    for ( int i = 0; i < MAX_ROWS; i++ )
        for ( int j = 0; j < MAX_COLS; j++ )
            cout << c[ i ][ j ] << endl;
}
```

양수 및 음수 아래 첨자

배열의 첫 번째 요소는 요소 0입니다. C++ 배열의 범위는 `배열[0]`에서 `배열[크기 - 1]` 사이입니다. 하지만 C++는 양수 및 음수 첨자를 지원합니다. 음수 첨자는 배열 경계 내에 속해야 하며, 그렇지 않으면 결과를 예측할 수 없습니다. 다음 코드에서는 양수 및 음수 배열 첨자를 보여줍니다.

C++

```
#include <iostream>
using namespace std;
```

```

int main() {
    int intArray[1024];
    for (int i = 0, j = 0; i < 1024; i++)
    {
        intArray[i] = j++;
    }

    cout << intArray[512] << endl; // 512

    cout << 257[intArray] << endl; // 257

    int *midArray = &intArray[512]; // pointer to the middle of the array

    cout << midArray[-256] << endl; // 256

    cout << intArray[-256] << endl; // unpredictable, may crash
}

```

마지막 줄의 음수 첨자는 배열 원본보다 메모리에서 낮은 위치의 256 `int` 주소를 가리키기 때문에 런타임 오류가 발생할 수 있습니다. 포인터 `midArray`는 `intArray`의 중간으로 초기화되므로 양수 배열 인덱스와 음수 배열 인덱스를 모두 사용할 수 있지만 위험합니다. 배열 첨자 오류는 컴파일 시간 오류를 발생시키지 않지만 예측할 수 없는 결과를 생성합니다.

첨자 연산자는 가환적입니다. 따라서 `배열[인덱스]`와 `인덱스[배열]`은 첨자 연산자가 오버로드되지 않는 한(오버로드된 연산자 참조) 동일한 것으로 보장됩니다. 첫 번째 형태가 가장 일반적인 코딩 방법이지만, 둘 다 작동합니다.

참고 항목

[후위 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

[배열](#)

[1차원 배열](#)

[다차원 배열](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

typeid 연산자

아티클 • 2024. 03. 11.

구문

```
typeid(type-id)
typeid(expression)
```

설명

typeid 연산자를 사용하면 런타임에 개체 형식이 결정됩니다.

typeid의 결과는 `const type_info&`입니다. 사용하는 **typeid** 형식에 따라 *type-id* 또는 *expression*의 형식을 나타내는 `type_info` 개체에 대한 참조가 값입니다. 자세한 내용은 [type_info 클래스](#)를 참조하세요.

typeid 연산자는 관리되는 형식(추상 선언자 또는 인스턴스)에서 작동하지 않습니다. 지정된 형식의 [Type](#)을(를) 가져오는 방법에 대한 자세한 내용은 [typeid](#)를 참조하세요.

typeid 연산자는 다형 클래스 형식의 l-value에 적용될 때 런타임 검사를 하지 않습니다. 제공된 정적 정보로 개체의 true 형식을 확인할 수 없습니다. 다음과 같은 경우가 여기에 해당합니다.

- 클래스에 대한 참조
- *으로 역참조되는 포인터
- 아래 첨자 포인터([])입니다. (다형 형식의 포인터에 첨자를 사용하는 것은 안전하지 않습니다.)

*expression*이 기본 클래스를 가리지만 개체 형식이 실제로 기본 클래스에서 파생된 경우 결과는 파생된 클래스의 `type_info` 참조입니다. *expression*은 다형 형식(가상 함수가 있는 클래스)을 가리켜야 합니다. 그렇지 않으면 결과는 *expression*에서 참조되는 정적 클래스의 `type_info`입니다. 또한 포인터가 가리키는 객체가 사용된 객체가 되도록 포인터를 역참조해야 합니다. 포인터를 역참조하지 않으면 포인터가 가리키는 것이 아니라 포인터의 결과는 `type_info`이(가) 됩니다. 예시:

C++

```

// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual void vfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
    Derived* pd = new Derived;
    Base* pb = pd;
    cout << typeid( pb ).name() << endl;    //prints "class Base *"
    cout << typeid( *pb ).name() << endl;    //prints "class Derived"
    cout << typeid( pd ).name() << endl;    //prints "class Derived *"
    cout << typeid( *pd ).name() << endl;    //prints "class Derived"
    delete pd;
}

```

*expression*이 포인터를 역참조하고 포인터의 값이 0이면 `typeid`이(가) `bad_typeid exception`을 `throw`합니다. 포인터가 유효한 개체를 가리키지 않으면 `_non_rtti_object` 예외가 `throw`됩니다. 개체가 어떻게든 유효하지 않으므로 오류를 트리거한 RTTI를 분석하려는 시도를 나타냅니다. (예를 들어 잘못된 포인터이거나 코드가 `/GR`을 사용하여 컴파일되지 않았습니다).

*expression*이 개체의 기본 클래스에 대한 포인터도 아니고 참조도 아닐 경우 *expression*의 정적 형식을 나타내는 `type_info` 참조가 결과입니다. 식의 정적 형식은 컴파일 타임에 알려지는 식의 형식을 참조합니다. 식의 정적 형식을 평가할 때 식의 의미 체계가 무시됩니다. 또한 식의 정적 형식을 결정할 때 가능할 경우 참조가 무시됩니다.

C++

```

// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}

```

`typeid`을(를) 템플릿에 사용하여 템플릿 매개 변수의 형식을 확인할 수도 있습니다.

C++

```
// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
    cout << typeid( T ).name() << "s compared." << endl;
    return ( arg1 > arg2 ? arg1 : arg2 );
}
```

참고 항목

[런타임 형식 정보](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

단항 더하기 및 부정 연산자: + 및 -

아티클 • 2024. 03. 06.

구문

- + cast-expression
- cast-expression

+ 연산자

단항 더하기 연산자(+)의 결과는 피연산자의 값입니다. 단항 더하기 연산자의 피연산자는 산술 형식이어야 합니다.

정수 계열 확장은 정수 계열 피연산자를 대상으로 수행됩니다. 결과 형식은 피연산자가 승격될 형식입니다. 따라서 `ch`(이)가 `char` 형식인 `+ch` 식이 `int` 유형으로 나타납니다. 값은 수정되지 않았습니다. 프로모션이 수행되는 방법에 대한 자세한 내용은 [표준 변환](#)을 참조하세요.

- 연산자

단항 부정 연산자(-)는 피연산자의 음수입니다. 단항 부정 연산자의 피연산자는 산술 형식이어야 합니다.

정수 계열 확장은 정수 계열 피연산자에서 수행되며, 결과 형식은 피연산자가 확장되는 형식입니다. 승격이 수행되는 방법에 대한 자세한 내용은 [표준 변환](#)을 참조하세요.

Microsoft 전용

부호 없는 수량의 단항 부정은 2^n 에서 피연산자의 값을 빼서 수행됩니다. 여기서 n은 지정된 부호 없는 형식의 개체에 있는 비트 수입니다.

Microsoft 전용 종료

참고 항목

[단항 연산자가 있는 식](#)

[C++ 기본 제공 연산자, 우선 순위 및 결합성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

식 (C++)

아티클 • 2024. 11. 21.

이 단원에서는 C++ 식에 대해 설명합니다. 식은 다음과 같은 용도 중 하나 이상으로 사용되는 일련의 연산자와 피연산자입니다.

- 피연산자에서 값을 계산합니다.
- 개체 또는 함수를 지정합니다.
- "부작용"을 생성합니다. (부작용은 식의 계산 이외의 작업(예: 개체 값 수정)입니다.)

C++에서는 연산자를 오버로드할 수 있으며 해당 의미는 사용자 정의할 수 있습니다. 하지만 우선 순위와 피연산자의 수를 수정할 수 없습니다. 이 섹션에서는 오버로드가 아니라 언어로 제공되기 때문에 연산자의 문법과 의미론을 설명합니다. 식 형식 및 식 의미 체계 외에도 다음 항목에 대해 설명합니다.

- 기본 식
- 범위 확인 연산자
- 후위 식
- 단항 연산자가 있는 식
- 이항 연산자가 있는 식
- 조건 연산자
- 상수 식
- 캐스팅 연산자
- 런타임 형식 정보

다른 단원에서 연산자에 대한 항목:

- C++ 기본 제공 연산자, 우선 순위 및 결합성
- 오버로드된 연산자
- typeid (C++/CLI)

① 참고

기본 제공 형식에 대한 연산자를 오버 로드할 수 없으며 해당 동작은 미리 정의되어 있습니다.

참고 항목

[C++ 언어 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

식의 형식

아티클 • 2023. 06. 16.

C++ 식은 여러 범주로 나뉩니다.

- **기본 식입니다.** 이 식은 다른 모든 식을 형성하는 빌딩 블록입니다.
- **후위 식입니다.** 이 식은 연산자가 뒤에 오는 기본 식으로, 그 예로 배열 첨자나 후위 증가 연산자 등이 있습니다.
- **단항 연산자를 사용하여 형성된 식입니다.** 단항 연산자는 하나의 식에서 하나의 피연산자에 대해서만 작동합니다.
- **이진 연산자를 사용하여 형성된 식입니다.** 이항 연산자는 한 식에서 두 개의 피연산자에 대해 작동합니다.
- **조건부 연산자가 있는 식입니다.** 조건 연산자는 C++ 언어에서 유일한 삼항 연산자이며 피연산자 세 개를 사용합니다.
- **상수 식입니다.** 상수 식은 상수 데이터로만 구성됩니다.
- **명시적 형식 변환이 있는 식입니다.** 명시적 형식 변환, 즉 "캐스트"를 식에서 사용할 수 있습니다.
- **포인터-멤버 연산자가 있는 식입니다.**
- **캐스팅.** 형식 안전 "캐스트"를 식에 사용할 수 있습니다.
- **런타임 형식 정보입니다.** 프로그램 실행 중 개체의 형식을 확인합니다.

추가 정보

식

기본 식

아티클 • 2024. 07. 08.

기본 식은 더 복잡한 식의 구성 요소이며 범위 결정 연산자(::)로 한정된 리터럴, 이름 및 이름일 수 있습니다. 기본 식의 형식은 다음 중 하나입니다.

primary-expression

literal

this

name

:: *name* (*expression*)

*literal*은 상수 기본 식입니다. 지정의 형태에 따라 형식이 결정됩니다. 리터럴 지정에 대한 자세한 내용은 [리터럴](#)을 참조하세요.

this 키워드는 클래스 인스턴스에 대한 포인터입니다. 비정적 멤버 함수 내에서 사용할 수 있으며 함수가 호출된 클래스의 인스턴스를 가리킵니다. **this** 키워드는 클래스 멤버 함수 본문 외부에서 사용할 수 없습니다.

this 포인터 형식에 대한 자세한 내용은 [this 포인터](#)를 참조하세요.

범위 결정 연산자(::) 뒤에 이름이 오는 것이 기본 식입니다. 이러한 이름은 멤버 이름이 아니라 전역 범위의 이름이어야 합니다. 이름 선언에 따라 식 형식이 결정됩니다. 선언 이름이 l-value인 경우 이는 l-value입니다(즉, 할당 식의 왼쪽에 나타날 수 있음). 범위 결정 연산자를 사용하면 전역 이름이 현재 범위에서 숨겨지더라도 해당 이름이 참조됩니다. 범위 결정 연산자를 사용하는 방법의 예는 [범위](#)를 참조하세요.

괄호로 묶인 식이 기본 식입니다. 해당 형식 및 값은 괄호로 묶이지 않은 식의 형식 및 값과 동일합니다. 괄호로 묶지 않은 식이 l-value일 경우 l-value입니다.

기본 식의 예제는 다음과 같습니다.

C++

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // a global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

이러한 예는 모두 *이름*으로 간주되므로 다양한 형식의 기본 식입니다.

C++

```
 MyClass // an identifier  
 MyClass::f // a qualified name  
 operator = // an operator function name  
 operator char* // a conversion operator function name  
 ~MyClass // a destructor name  
 A::B // a qualified name  
 A<int> // a template id
```

참고 항목

[식의 형식](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Ellipsis 및 Variadic 템플릿

아티클 • 2023. 10. 12.

이 문서에서는 C++ variadic 템플릿에서 줄임표(ellipsis)를 사용하는 방법을 보여 줍니다. 줄임표는 C 및 C++에서 많은 용도를 사용했습니다. 여기에는 함수에 대한 변수 인수 목록이 포함됩니다. `printf()` C 런타임 라이브러리의 함수는 가장 잘 알려진 예제 중 하나입니다.

variadic 템플릿은 임의의 수의 인수를 지원하는 클래스 또는 함수 템플릿입니다. 이 메커니즘은 C++ 라이브러리 개발자에게 특히 유용합니다. 클래스 템플릿과 함수 템플릿 모두에 적용할 수 있으므로 다양한 형식 안전 및 사소한 기능과 유연성을 제공할 수 있습니다.

구문

줄임표는 variadic 템플릿에서 두 가지 방법으로 사용됩니다. 매개 변수 이름의 왼쪽에는 매개 변수 팩 `Args` 표시되고 매개 변수 이름의 오른쪽에는 매개 변수 팩이 별도의 이름으로 확장됩니다.

다음은 variadic 클래스 템플릿 정의 구문의 기본 예제입니다.

C++

```
template<typename... Arguments> class classname;
```

매개 변수 팩과 확장 모두에 대해 다음 예제와 같이 기본 설정에 따라 줄임표 주위에 공백을 추가할 수 있습니다.

C++

```
template<typename ...Arguments> class classname;
```

또는 다음 예제를 수행합니다.

C++

```
template<typename ... Arguments> class classname;
```

이 문서에서는 첫 번째 예제에 표시된 규칙(줄임표가 연결됨)을 `typename` 사용합니다.

앞의 예제 `Arguments` 에서는 매개 변수 팩입니다. 클래스 `classname` 는 다음 예제와 같이 가변 개수의 인수를 수락할 수 있습니다.

C++

```
template<typename... Arguments> class vtclass;

vtclass<> vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

variadic 클래스 템플릿 정의를 사용하면 하나 이상의 매개 변수가 필요할 수도 있습니다.

C++

```
template <typename First, typename... Rest> class classname;
```

다음은 variadic 함수 템플릿 구문의 기본 예제입니다.

C++

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

Arguments 그런 다음 다음 섹션에 표시된 대로 매개 변수 팩을 확장하여 사용할 수 있습니다.

다음 예제를 포함하지만 이에 국한되지 않는 다른 형태의 variadic 함수 템플릿 구문이 가능합니다.

C++

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

다음과 같은 **const** 지정자도 허용됩니다.

C++

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

variadic 템플릿 클래스 정의와 마찬가지로 매개 변수가 하나 이상 필요한 함수를 만들 수 있습니다.

C++

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

Variadic 템플릿은 연산자 `sizeof...()` (이전 `sizeof()` 연산자와 관련이 없는)를 사용합니다.

C++

```
template<typename... Arguments>
void tfunc(const Arguments&... args)
{
    constexpr auto numargs{ sizeof...(Arguments) };

    X xobj[numargs]; // array of some previously defined type X

    helper_func(xobj, args...);
}
```

줄임표 배치에 대한 자세한 내용

이전에 이 문서에서는 매개 변수 팩 및 확장을 정의하는 줄임표 배치에 대해 설명했습니다. "매개 변수 이름의 왼쪽에는 매개 변수 팩을 의미하고 매개 변수 이름의 오른쪽에는 매개 변수 팩을 별도의 이름으로 확장합니다." 기술적으로는 사실이지만 코드로 변환할 때 혼동될 수 있습니다. 고려할 사항은 다음과 같습니다.

- `template-parameter-list(template <parameter-list>)` `typename...`에서 템플릿 매개 변수 팩을 소개합니다.
- 매개 변수 선언 절(`func(parameter-list)`)에서 "최상위" 줄임표는 함수 매개 변수 팩을 도입하며 줄임표 위치 지정이 중요합니다.

C++

```
// v1 is NOT a function parameter pack:
template <typename... Types> void func1(std::vector<Types...> v1);

// v2 IS a function parameter pack:
template <typename... Types> void func2(std::vector<Types>... v2);
```

- 매개 변수 이름 바로 뒤에 줄임표가 표시된 경우 매개 변수 팩 확장이 있는 것입니다.

예시

variadic 함수 템플릿 메커니즘을 설명하는 좋은 방법은 다음과 같은 기능 중 일부를 다시 작성하는 것입니다.

C++

```
#include <iostream>

using namespace std;

void print() {
    cout << endl;
}

template <typename T> void print(const T& t) {
    cout << t << endl;
}

template <typename First, typename... Rest> void print(const First& first,
const Rest&... rest) {
    cout << first << ", ";
    print(rest...); // recursive call using pack expansion syntax
}

int main()
{
    print(); // calls first overload, outputting only a newline
    print(1); // calls second overload

    // these call the third overload, the variadic template,
    // which uses recursion as needed.
    print(10, 20);
    print(100, 200, 300);
    print("first", 2, "third", 3.14159);
}
```

출력

Output

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

① 참고

variadic 함수 템플릿을 통합하는 대부분의 구현은 일부 형식의 재귀를 사용하지만 기존 재귀와는 약간 다릅니다. 기존 재귀에는 동일한 서명을 사용하여 자신을 호출하는 함수가 포함됩니다. (오버로드되거나 템플릿으로 작성될 수 있지만 매번 동일한 서명이 선택됩니다.) Variadic 재귀에는 서로 다른(거의 항상 감소하는) 인수 수를 사용하여 variadic 함수 템플릿을 호출하여 매번 다른 서명을 스탬핑하는 작업이 포함됩니다. "기본 사례"는 여전히 필요하지만 재귀의 특성은 다릅니다.

후위 식

아티클 • 2024. 07. 08.

후위 식은 기본 식 또는 후위 연산자가 기본 식 뒤에 오는 식으로 구성됩니다. 다음 표에서는 후위 연산자를 보여 줍니다.

후위 연산자

 테이블 확장

연산자 이름	연산자 표기법
아래 첨자 연산자	[]
함수 호출 연산자	()
명시적 형식 변환 연산자	<i>type-name</i> ()
멤버 액세스 연산자	. 또는 ->
후위 증가 연산자	++
후위 감소 연산자	--

다음 구문에서는 가능한 후위 식을 설명합니다.

```
primary-expression
postfix-expression[expression]postfix-expression(expression-list)simple-
type-name(expression-list)postfix-expression.namepostfix-expression-
>namepostfix-expression++postfix-expression--cast-keyword < typename >
(expression )typeid ( typename )
```

위에서 *postfix-expression*은 기본 식 또는 다른 후위 식일 수 있습니다. 후위 식은 왼쪽에 서 오른쪽까지 그룹화되므로 식이 다음과 같이 연결될 수 있습니다.

C++

```
func(1)->GetValue()++
```

위 식에서 `func`은 기본 식, `func(1)`은 함수 후위 식, `func(1)->GetValue()`는 클래스 멤버를 지정하는 후위 식, `func(1)->GetValue()`는 또 다른 함수 후위 식이며, 전체 식은 `GetValue`의 반환 값을 증가시키는 후위 식입니다. 이 식은 1을 인수로 전달하는 함수를 호출하거

나 클래스의 포인터를 반환 값으로 가져오라는 의미입니다. 그다음에 해당 클래스에서 `GetValue()`를 호출하고 반환되는 값을 증가시킵니다.

위에 나열된 식은 할당 식입니다. 이는 이 식의 결과가 r-value여야 한다는 의미입니다.

후위 식 형식

C++

```
simple-type-name ( expression-list )
```

생성자의 호출을 나타냅니다. `simple-type-name`이 기본 형식인 경우 식 목록은 단일 식이어야 하며 이 식은 식 값의 기본 형식으로의 캐스트를 나타냅니다. 이 형식의 캐스트 식은 생성자를 모방합니다. 이 형식을 사용하면 기본 형식 및 클래스를 같은 구문을 사용하여 생성할 수 있기 때문에 이 형식은 특히 템플릿 클래스를 정의할 때 유용합니다.

`cast-keyword`는 `dynamic_cast`, `static_cast` 또는 `reinterpret_cast` 중 하나입니다. 자세한 내용은 `dynamic_cast`, `static_cast` 및 `reinterpret_cast`에서 확인할 수 있습니다.

`typeid` 연산자는 후위 식으로 간주됩니다. `typeid` 연산자를 참조하세요.

형식 및 실제 인수

호출하는 프로그램은 "실제 인수"에서 호출되는 함수에 정보를 전달합니다. 호출되는 함수는 해당하는 "형식 인수"를 사용하여 정보에 액세스합니다.

함수가 호출되면 다음 작업이 수행됩니다.

- 모든 실제 인수(호출자가 제공한 인수)가 계산됩니다. 이 인수가 계산되는 암시적 순서는 없지만 함수에 진입하기 전에 인수가 모두 계산되고 의도하지 않은 결과가 모두 완료됩니다.
- 식 목록에서 해당하는 실제 인수를 사용하여 각 형식 인수가 초기화됩니다. 형식 인수는 함수 헤더에 선언되며 함수 본문에 사용되는 인수입니다. 실제 인수를 올바른 형식으로 변환할 때 초기화에 의해 표준 변환과 사용자 정의 변환이 둘 다 수행되는 것처럼 변환이 수행됩니다. 다음 코드는 수행된 초기화를 개념적으로 보여 줍니다.

C++

```
void Func( int i ); // Function prototype
...
Func( 7 );           // Execute function call
```

호출 전의 개념적 초기화는 다음과 같습니다.

C++

```
int Temp_i = 7;  
Func( Temp_i );
```

괄호 구문 대신 등호 구문을 사용하는 것처럼 초기화가 수행됩니다. 함수에 값을 전달하기 전에 `i`의 복사본을 만듭니다. (자세한 내용은 [이니셜라이저](#) 및 [변환](#)을 참조하세요.)

따라서 함수 프로토타입(선언)이 `long` 형식의 인수를 호출하고 호출하는 프로그램이 `int` 형식의 실제 인수를 제공하는 경우 `long` 형식에 대한 표준 형식 변환을 사용하여 실제 인수가 승격됩니다([표준 변환](#) 참조).

형식 인수의 형식에 대한 표준 또는 사용자 정의 변환이 없는 실제 인수를 제공하는 것은 오류입니다.

클래스 형식의 실제 인수의 경우 클래스의 생성자를 호출하여 형식 인수가 초기화됩니다. (이 특수 클래스 멤버 함수에 대한 자세한 내용은 [생성자](#)를 참조하세요.)

- 함수 호출이 실행됩니다.

다음 프로그램 조각은 함수 호출을 보여 줍니다.

C++

```
// expre_Formal_and_Actual_Arguments.cpp  
void func( long param1, double param2 );  
  
int main()  
{  
    long i = 1;  
    double j = 2;  
  
    // Call func with actual arguments i and j.  
    func( i, j );  
}  
  
// Define func with formal parameters param1 and param2.  
void func( long param1, double param2 )  
{  
}
```

`func` 가 `main`에서 호출되면 정식 매개 변수 `param1`이 `i` 값으로 초기화되고(`i`는 표준 변환을 사용하여 올바른 형식에 해당하도록 `long` 형식으로 변환됨), 공식 매개 변수 `param2`는 `j` 값으로 초기화됩니다(`j`는 표준 변환을 사용하여 `double` 형식으로 변환됨).

인수 형식 처리

`const` 형식으로 선언된 형식 인수는 함수 본문 내에서 변경할 수 없습니다. 함수는 `const` 형식이 아닌 모든 인수를 변경할 수 있습니다. 그러나 이러한 변경은 함수에 대해 로컬이며 실제 인수가 `const` 형식이 아닌 개체에 대한 참조가 아닌 경우 실제 인수의 값에 영향을 주지 않습니다.

다음 함수는 이러한 개념을 몇 가지 보여 줍니다.

```
C++  
  
// expre_Treatment_of_Argument_Types.cpp  
int func1( const int i, int j, char *c ) {  
    i = 7;      // C3892 i is const.  
    j = i;      // value of j is lost at return  
    *c = 'a' + j; // changes value of c in calling function  
    return i;  
}  
  
double& func2( double& d, const char *c ) {  
    d = 14.387;    // changes value of d in calling function.  
    *c = 'a';      // C3892 c is a pointer to a const object.  
    return d;  
}
```

줄임표 및 기본 인수

함수는 줄임표(...) 또는 기본 인수 메서드 중 하나를 사용하여 함수 정의에 지정된 것보다 적은 인수를 받도록 선언할 수 있습니다.

줄임표는 선언에 인수가 필요할 수 있지만 숫자와 형식은 지정되지 않음을 나타냅니다. 이는 C++의 장점인 형식 안전성을 없애기 때문에 일반적으로 좋지 않은 C++ 프로그래밍 관행입니다. 형식 및 실제 인수 형식을 보여 주는 함수보다는 줄임표로 선언된 함수에 다양한 변환이 적용됩니다.

- 실제 인수가 `float` 형식인 경우 해당 인수는 함수 호출 전에 `double` 형식으로 승격됩니다.
- 모든 `signed char` 또는 `unsigned char`, `signed short` 또는 `unsigned short`, 열거형 형식 또는 비트 필드는 정수 계열 승격을 사용하여 `signed int` 또는 `unsigned int`로 변환됩니다.
- 클래스 형식의 인수는 데이터 구조 값에 의해 전달됩니다. 사본은 클래스의 복사 생성자(있을 경우) 호출을 통해서가 아닌 이진 복사를 통해 만들어집니다.

줄임표를 사용할 경우 인수 목록 마지막에 선언해야 합니다. 인수의 가변 수 전달에 대한 자세한 내용은 [런타임 라이브러리](#) 참조에서 `va_arg`, `va_start` 및 `va_list`의 설명을 참조하세요.

CLR 프로그래밍의 기본 인수에 대한 자세한 내용은 [가변 인수 목록\(...\)\(C++/CLI\)](#)을 참조하세요.

기본 인수를 사용하면 함수 호출에 아무것도 제공되지 않은 경우 인수가 가정하는 값을 지정할 수 있습니다. 다음 코드는 기본 인수의 작동 방법을 보여 줍니다. 기본 인수 지정 제한에 대한 자세한 내용은 [기본 인수](#)를 참조하세요.

C++

```
// expe_Ellipsis_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", ", " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

앞의 프로그램은 두 개의 인수를 사용하는 `print` 함수를 선언합니다. 그러나 두 번째 인수 `terminator`의 기본값은 `"\n"`입니다. `main`에서 `print`에 대한 처음 두 개의 호출은 기본 두 번째 인수가 새 줄을 제공하여 인쇄된 문자열을 종료할 수 있도록 합니다. 세 번째 호출은 두 번째 인수에 대해 명시적 값을 지정합니다. 프로그램의 출력:

Output

hello,
world!
good morning, sunshine.

참고 항목

[식의 형식](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

단항 연산자가 있는 식

아티클 • 2024. 07. 20.

단항 연산자는 하나의 식에서 하나의 피연산자에 대해서만 작동합니다. 단항 연산자의 종류는 다음과 같습니다.

- 간접 참조 연산자(*)
- Address-of 연산자(&)
- 단항 더하기 연산자(+)
- 단항 부정 연산자(-)
- 논리 부정 연산자(!)
- 1의 보수 연산자(~)
- 전위 증가 연산자(++)
- 전위 감소 연산자(--)
- 캐스트 연산자()
- sizeof 연산자
- alignof 연산자
- noexcept 식
- new 연산자
- delete 연산자

이러한 연산자는 오른쪽에서 왼쪽으로 결합됩니다. 단항 식은 일반적으로 후위 식 또는 주 식 앞에 오는 구문을 포함합니다.

구문

```
unary-expression:  
    postfix-expression  
    ++ cast-expression  
    -- cast-expression  
    unary-operator cast-expression
```

```
sizeof unary-expression
sizeof ( type-id )
sizeof ... ( identifier )
alignof ( type-id )
noexcept-expression
new-expression
delete-expression
```

unary-operator: 다음 중 하나

* & + - ! ~

설명

모든 *postfix-expression* 항목은 *unary-expression*으로 간주되며, 모든 *primary-expression* 항목이 *postfix-expression*으로 간주되기 때문에 모든 *primary-expression* 항목 또한 *unary-expression*으로 간주됩니다. 자세한 내용은 [후위 식 및 기본 식](#)을 참조하십시오.

*cast-expression*은 형식을 변경하는 선택적 캐스트를 가진 *unary-expression*입니다. 자세한 내용은 [Cast 연산자 \(\)](#)를 참조하세요.

*noexcept-expression*은 *constant-expression* 인수가 있는 *noexcept-specifier*입니다. 자세한 내용은 [noexcept](#)를 참조하세요.

*new-expression*은 [new](#) 연산자를 말합니다. *delete-expression*은 [delete](#) 연산자를 말합니다. 자세한 내용은 [new 연산자](#) 및 [delete 연산자](#)를 참조하세요.

참고 항목

[식 형식](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

이항 연산자로 구성된 식

아티클 • 2023. 10. 12.

이항 연산자는 한 식에서 두 개의 피연산자에 대해 작동합니다. 이항 연산자는 다음과 같습니다.

- 곱하기 연산자

- 곱하기(*)
- 나누기(/)
- 모듈러스(%)

- 가산 연산자

- 더하기(+)
- 빼기(-)

- 시프트 연산자

- 오른쪽 시프트(>>)
- 왼쪽 시프트(<<)

- 관계형 및 같음 연산자

- 미만(<)
- 보다 큼(>)
- 작거나 같음(<=)
- 보다 크거나 같음(>=)
- 같음(==)
- (!=)와 같지 않음

- 비트 연산자

- 비트 AND(&)
- 비트 배타적 OR(^)
- 비트 포함 OR(|)

- 논리 연산자
 - 논리 AND(&&)
 - 논리 OR(||)
- 할당 연산자
 - 대입(=)
 - 더하기 대입(+=)
 - 빼기 할당(-=)
 - 곱하기 대입(*=)
 - 나누기 대입(/=)
 - 모듈러스 대입(%=)
 - 왼쪽 시프트 할당(<<=)
 - 오른쪽 시프트 할당(>>=)
 - 비트 AND 할당(&=)
 - 배타적 비트 OR 대입(^=)
 - 비트 포괄 OR 할당(|=)
- 쉼표 연산자(,)

참고 항목

식의 형식

C++ 상수 식

아티클 • 2023. 10. 12.

상수 값은 변경되지 않는 값입니다. C++에서는 개체를 수정할 수 없음을 나타내는 의도를 표현하고 해당 의도를 적용할 수 있는 두 가지 키워드를 제공합니다.

C++에서는 다음 선언에 대해 상수 식(상수로 계산되는 식)이 필요합니다.

- 배열 범위
- Case 문의 선택기
- 비트 필드 길이 사양
- 열거형 이니셜라이저

상수 식에는 다음 피연산자만 사용할 수 있습니다.

- 리터럴
- 열거형 상수
- 상수 식을 사용하여 초기화되며 const로 선언된 값
- `sizeof` 식

비정수 상수는 명시적이거나 암시적으로 상수 식에 사용할 수 있는 정수 계열 형식으로 변환해야 합니다. 따라서 다음 코드를 사용할 수 있습니다.

C++

```
const double Size = 11.0;
char chArray[(int)Size];
```

정수 형식으로의 명시적 변환은 상수 식에서 유효합니다. 연산자에 피연산자로 사용되는 경우를 제외하고 다른 모든 형식 및 파생 형식은 `sizeof` 불법입니다.

쉼표 연산자와 할당 연산자를 상수 식에 사용할 수 없습니다.

참고 항목

[식의 형식](#)

식의 의미

아티클 • 2024. 07. 08.

식은 해당 연산자의 그룹화 및 우선 순위에 따라 계산됩니다. ([어휘 규칙](#)의 연산자 우선 순위 및 결합성에서 C++ 연산자가 식에 적용하는 관계를 보여줍니다.)

계산 순서

다음 예제를 고려해 보세요.

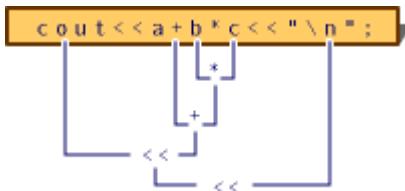
C++

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

Output

```
38
38
54
```



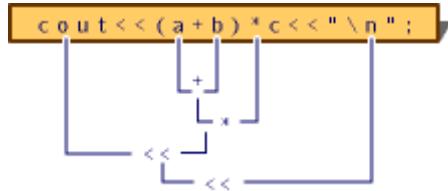
식 평가 순서

위의 그림에 표시된 식이 계산되는 순서는 연산자의 우선 순위 및 결합성에 따라 결정됩니다.

1. 이 식에서는 곱셈(*)의 우선 순위가 가장 높습니다. 따라서 하위 식 `b * c` 가 먼저 계산됩니다.
2. 다음으로 덧셈(+)의 우선 순위가 높으므로 `a`와 `b`를 곱한 값에 `c` 가 더해집니다.

3. 왼쪽 시프트(<<)는 식에서 우선 순위가 가장 낮지만 두 가지가 발생합니다. 왼쪽 시프트 연산자는 왼쪽에서 오른쪽으로 그룹화하므로 왼쪽 하위 식이 먼저 계산된 다음 오른쪽 하위 식이 계산됩니다.

괄호를 사용하여 하위 식을 그룹화할 경우 다음 그림에 표시된 것처럼 우선 순위 및 식이 계산되는 순서도 변경됩니다.



괄호가 있는 식 평가 순서

위의 그림에 있는 것과 같은 식은 전적으로 의도하지 않은 결과를 위해 계산되며, 이 경우에는 표준 출력 디바이스로 정보를 전송하기 위해 계산됩니다.

식에서의 표기법

C++ 언어는 피연산자를 지정할 때 특정 호환성을 지정합니다. 다음 표에서는 형식 형식의 피연산자가 필요한 연산자에 사용할 수 있는 피연산자 형식을 보여줍니다.

연산자에 사용할 수 있는 피연산자 형식

[+] 테이블 확장

필요한 형식	허용되는 형식
type	<code>const</code> 형식 <code>volatile</code> 형식 형식& <code>const</code> 형식& <code>volatile</code> 형식& <code>volatile const</code> 형식 <code>volatile const</code> 형식&
type *	type * <code>const</code> 형식 * <code>volatile</code> 형식 * <code>volatile const</code> 형식 *
<code>const</code> 형식	type <code>const</code> 형식 <code>const</code> 형식&

필요한 형식	허용되는 형식
<code>volatile</code> 형식	<code>type</code> <code>volatile</code> 형식 <code>volatile</code> 형식&

항상 이전 규칙을 조합하여 사용할 수 있으므로 포인터가 필요한 지점에 `volatile` 개체에 대한 `const` 포인터를 제공할 수 있습니다.

모호한 식

특정 식의 의미가 모호합니다. 이러한 식은 개체의 값이 동일한 식에서 두 번 이상 변경될 때 가장 자주 발생합니다. 이러한 식은 언어에서 하나로 정의되지 않는 특정 계산 순서에 의존합니다. 다음 예제를 참조하세요.

```
int i = 7;
func( i, ++i );
```

C++ 언어에서는 함수 호출에 대한 인수가 계산되는 순서를 보장하지 않습니다. 따라서 위의 예제에서 매개 변수의 계산 방향이 왼쪽에서 오른쪽인지, 아니면 오른쪽에서 왼쪽인지에 따라 `func`는 매개 변수의 값으로 7과 8을 받거나 8과 8을 받을 수 있습니다.

C++ 시퀀스 위치(Microsoft 전용)

식은 연속적인 "시퀀스 위치" 사이에서 개체의 값을 한 번만 수정할 수 있습니다.

현재 C++ 언어 정의는 시퀀스 위치를 지정하지 않습니다. Microsoft C++는 C 연산자를 사용하고 오버로드된 연산자는 사용하지 않는 모든 식에 대해 ANSI C와 동일한 시퀀스 위치를 사용합니다. 연산자가 오버로드되면 연산자 시퀀스에서 함수 호출 시퀀스로 의미 체계가 변경됩니다. Microsoft C++는 다음 시퀀스 위치를 사용합니다.

- 논리 AND 연산자의 왼쪽 피연산자(`&&`). 계속하기 전에 논리 AND 연산자의 왼쪽 피연산자가 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다. 논리 AND 연산자의 오른쪽 피연산자가 평가된다는 보장은 없습니다.
- 논리 OR 연산자의 왼쪽 피연산자(`||`). 계속하기 전에 논리 OR 연산자의 왼쪽 피연산자가 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다. 논리 OR 연산자의 오른쪽 피연산자가 평가된다는 보장은 없습니다.

- 쉼표 연산자의 왼쪽 피연산자. 계속하기 전에 쉼표 연산자의 왼쪽 피연산자가 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다. 쉼표 연산자의 두 피연산자는 항상 계산됩니다.
- 함수 호출 연산자. 함수에 들어가기 전에 함수 호출 식 및 기본 인수를 포함한 함수의 모든 인수가 계산되고 의도하지 않은 모든 결과가 완료됩니다. 인수 또는 함수 호출 식에 대해 지정된 계산 순서는 없습니다.
- 조건 연산자의 첫째 피연산자. 계속하기 전에 조건 연산자의 첫째 피연산자가 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다.
- 전체 초기화 식의 끝(예: 선언 문에서 초기화의 끝)
- 식 문의 식. 식 문은 선택적 식과 세미콜론(;)으로 구성됩니다. 식의 의도하지 않은 결과가 완전히 계산됩니다.
- 선택(if 또는 switch) 문의 제어 식. 선택에 종속된 코드가 실행되기 전에 식이 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다.
- while 또는 do 문의 제어 식. while 또는 do 루프의 다음 반복에 있는 문이 실행되기 전에 식이 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다.
- for 문의 세 가지 식 각각. 다음 식으로 이동하기 전에 각 식이 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다.
- return 문의 식. 호출 함수로 제어가 반환되기 전에 식이 완전히 계산되고 의도하지 않은 모든 결과가 완료됩니다.

참고 항목

식

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

캐스팅

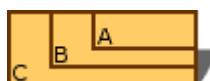
아티클 • 2024. 06. 16.

C++에서 클래스가 하나 이상의 가상 함수를 포함하는 기본 클래스에서 파생된 경우 해당 기본 클래스 형식에 대한 포인터를 사용하여 파생 클래스 개체에서 가상 함수를 호출할 수 있습니다. 가상 함수를 포함하는 클래스를 "다형 클래스"라고도 합니다.



클래스 계층 구조

c 형식의 개체는 다음과 같이 시각화할 수 있습니다.



하위 개체 B와 A가 있는 클래스 C

c 클래스의 인스턴스를 제공하면 b 하위 개체 및 a 하위 개체가 있습니다. c 및 a 하위 개체를 포함한 b의 인스턴스는 "완전한 개체"입니다.

파생 클래스에는 자신이 파생된 모든 기본 클래스의 정의가 완전히 포함되어 있으므로 해당 기본 클래스에 포인터를 캐스팅하는 것이 안전합니다(업캐스트라고도 함). 기본 클래스에 대한 포인터가 주어지면 포인터를 파생 클래스의 인스턴스로 캐스팅하는 것이 안전할 수 있습니다(다운캐스트라고도 함).

런타임 형식 정보를 사용하면 실제로 포인터가 완전한 개체를 가리키며 해당 계층 구조에서 다른 개체를 가리키도록 안전하게 캐스팅될 수 있는지 여부를 확인할 수 있습니다. `dynamic_cast` 연산자는 런타임 검사를 수행하여 작업이 안전한지 확인합니다. 다운캐스팅의 필요성을 피하기 위해 가상 함수를 사용할 수 있도록 클래스 계층 구조를 설계하는 것이 좋습니다. 그러나 다운캐스트해야 하는 경우에는 `dynamic_cast`를 사용하여 작업이 안전한지 확인합니다.

비다형 형식의 변환에서는 `static_cast` 연산자를 사용할 수 있습니다. 이 항목에서는 정적 캐스팅 변환과 동적 캐스팅 변환 간의 차이점과 각 항목을 적절하게 사용하는 경우에 대해 설명합니다.

다음 예제에서는 `dynamic_cast` 및 `static_cast`의 사용을 보여 줍니다.

C++

```
#include <iostream>

class Base {
public:
```

```

        virtual void print() { std::cout << "Base\n"; }

};

class Derived1 : public Base {
public:
    void print() override { std::cout << "Derived1\n"; }
};

class Derived2 : public Base {
public:
    void print() override { std::cout << "Derived2\n"; }
};

class MostDerived : public Derived1, public Derived2 {
public:
    void print() override { std::cout << "MostDerived\n"; }
};

int main() {
    MostDerived md;
    Base* b1 = static_cast<Derived1*>(&md); // Upcast to Derived1 is safe
    Base* b2 = static_cast<Derived2*>(&md); // Upcast to Derived2 is safe

    // Downcast to MostDerived is ambiguous and unsafe
    // MostDerived* md1 = static_cast<MostDerived*>(b1); // This won't
    compile
    // MostDerived* md2 = static_cast<MostDerived*>(b2); // This won't
    compile

    // Correct way to downcast in this situation
    MostDerived* md1 = dynamic_cast<MostDerived*>(b1); // This is safe
    MostDerived* md2 = dynamic_cast<MostDerived*>(b2); // This is safe

    md1->print(); // Prints "MostDerived"
    md2->print(); // Prints "MostDerived"

    return 0;
}

```

이 섹션에서는 다음 항목을 다룹니다.

- 캐스팅 연산자\
- 런타임 형식 정보

참고 항목

식

① 참고: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. [자세한 정보](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

캐스팅 연산자

아티클 • 2023. 10. 12.

캐스트 연산자에는 C++ 언어 전용 연산자가 몇 가지 있습니다. 이 연산자는 예전 스타일의 C 언어 캐스트에 있는 일부 모호함과 위험성을 제거하는 데 목적이 있습니다. 그 종류는 다음과 같습니다.

- `dynamic_cast` 다형 형식의 변환에 사용됩니다.
- `static_cast` 비포형 형식 변환에 사용됩니다.
- `const_cast`, `volatile` 및 `__unaligned` 특성을 제거하는 `const` 데 사용됩니다.
- `reinterpret_cast` 비트의 간단한 재해석에 사용됩니다.
- `safe_cast` C++/CLI에서 확인 가능한 MSIL을 생성하는 데 사용됩니다.

`const_cast` 이러한 연산자는 이전 스타일 캐스트와 `reinterpret_cast` 동일한 위험을 제시하기 때문에 사용 및 최후의 수단으로. 하지만 이 두 캐스트는 이전 스타일 캐스트를 완전히 바꾸기 위해 여전히 필요합니다.

참고 항목

[캐스팅](#)

dynamic_cast 연산자

아티클 • 2023. 10. 12.

피연산 `expression` 자를 형식 `type-id`의 개체로 변환합니다.

구문

```
dynamic_cast < type-id > ( expression )
```

설명

포인터 `type-id` 또는 이전에 정의한 클래스 형식에 대한 참조 또는 "void에 대한 포인터"여야 합니다. `type-id`이 포인터인 경우 `expression`의 형식은 포인터여야 하며 `type-id`이 참조인 경우에는 `I` 값이어야 합니다.

정적 및 동적 캐스팅 변환 간의 차이점과 각각을 사용하는 것이 적절한 경우의 설명은 `static_cast` 참조하세요.

관리 코드의 `dynamic_cast` 동작에는 다음과 같은 두 가지 주요 변경 내용이 있습니다.

- `dynamic_cast` 상자 열거형의 기본 형식에 대한 포인터에 대한 포인터는 런타임에 실패하고 변환된 포인터 대신 0을 반환합니다.
- `dynamic_cast` 는 값 형식에 대한 내부 포인터인 경우 `type-id` 더 이상 예외를 throw하지 않습니다. 대신 런타임에 캐스트가 실패합니다. 캐스트는 throw하는 대신 0 포인터 값을 반환합니다.

명확하게 액세스할 수 있는 직접 또는 간접 기본 클래스에 대한 포인터인 경우 `type-id` 형식 `type-id`의 `expression` 고유 하위 개체에 대한 포인터가 결과입니다. 예시:

C++

```
// dynamic_cast_1.cpp
// compile with: /c
class B { };
class C : public B { };
class D : public C { };

void f(D* pd) {
    C* pc = dynamic_cast<C*>(pd); // ok: C is a direct base class
```

```

        // pc points to C subobject of pd
B* pb = dynamic_cast<B*>(pd); // ok: B is an indirect base class
                                // pb points to B subobject of pd
}

```

이 유형의 변환은 파생 클래스에서 파생된 클래스로 포인터를 클래스 계층 구조 위로 이동하기 때문에 "upcast"라고 합니다. 업캐스트는 암시적 변환입니다.

`void*`이면 `type-id` 실제 형식 `expression`을 확인하기 위해 런타임 검사 만들어집니다. 결과는 `.에서` 가리키는 `expression` 전체 개체에 대한 포인터입니다. 예시:

C++

```

// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = new B;
    void* pv = dynamic_cast<void*>(pa);
    // pv now points to an object of type A

    pv = dynamic_cast<void*>(pb);
    // pv now points to an object of type B
}

```

그렇지 않은 경우 `type-id` 가리키는 개체를 가리키는 `type-id` 형식으로 변환할 수 있는지 확인하기 위해 `expression` 런타임 검사 만들어 `void*` 집니다.

형식이 형식 `type-id`의 `expression` 기본 클래스인 경우 런타임 검사 실제로 형식 `type-id`의 전체 개체를 가리키는지 `expression` 확인합니다. `true`이면 결과는 형식 `type-id`의 전체 개체에 대한 포인터입니다. 예시:

C++

```

// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
    B* pb = new D;    // unclear but ok
    B* pb2 = new B;

    D* pd = dynamic_cast<D*>(pb);    // ok: pb actually points to a D
}

```

```
D* pd2 = dynamic_cast<D*>(pb2); // pb2 points to a B not a D
}
```

이 유형의 변환은 지정된 클래스에서 파생된 클래스로 클래스 계층 구조 아래로 포인터를 이동하기 때문에 "downcast"라고 합니다.

여러 상속의 경우 모호성에 대한 가능성이 도입됩니다. 다음 그림에 표시된 클래스 계층 구조를 고려합니다.

CLR 형식 `dynamic_cast` 의 경우 변환을 암시적으로 수행할 수 있는 경우 no-op 또는 동적 검사 수행하고 변환이 실패하면 반환 `nullptr` 하는 MSIL `isinst` 명령이 발생합니다.

다음 샘플에서는 클래스가 특정 형식의 인스턴스인지 확인하는 데 사용합니다

```
dynamic_cast .
```

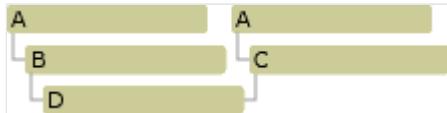
C++

```
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

void PrintObjectType( Object^o ) {
    if( dynamic_cast<String^>(o) )
        Console::WriteLine("Object is a String");
    else if( dynamic_cast<int^>(o) )
        Console::WriteLine("Object is an int");
}

int main() {
    Object^o1 = "hello";
    Object^o2 = 10;

    PrintObjectType(o1);
    PrintObjectType(o2);
}
```



형식 `D` 의 개체에 대한 포인터를 안전하게 캐스팅 `B` 할 수 있습니다 `C`. 그러나 개체를 가리키 `A` 도록 캐스팅되는 경우 `D` 어떤 인스턴스가 `A` 발생합니까? 이로 인해 캐스팅 오류가 모호합니다. 이 문제를 해결하려면 두 개의 명확한 캐스트를 수행할 수 있습니다. 예시:

C++

```
// dynamic_cast_4.cpp
// compile with: /c /GR
```

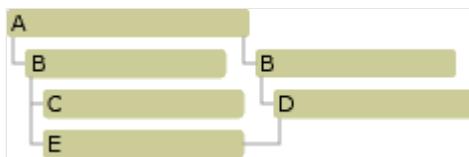
```

class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {virtual void f();};
class D : public B, public C {virtual void f();};

void f() {
    D* pd = new D;
    A* pa = dynamic_cast<A*>(pd);      // C4540, ambiguous cast fails at runtime
    B* pb = dynamic_cast<B*>(pd);      // first cast to B
    A* pa2 = dynamic_cast<A*>(pb);      // ok: unambiguous
}

```

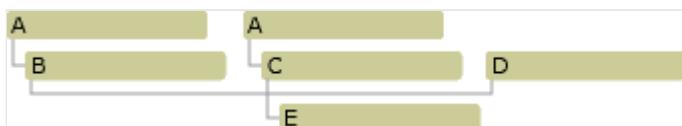
가상 기본 클래스를 사용할 때 추가 모호성을 도입할 수 있습니다. 다음 그림에 표시된 클래스 계층 구조를 고려합니다.



가상 기본 클래스를 보여 주는 클래스 계층 구조

이 계층 구조 A에서는 가상 기본 클래스입니다. 클래스 E 인스턴스와 하위 개체 `dynamic_cast`에 대한 포인터가 A 지정된 경우 모호성으로 인해 실패할 B 포인터에 대한 포인터입니다. 먼저 전체 E 개체로 다시 캐스팅한 다음, 명확한 방식으로 계층 구조를 백업하여 올바른 B 개체에 도달해야 합니다.

다음 그림에 표시된 클래스 계층 구조를 고려합니다.



중복된 기본 클래스를 보여 주는 클래스 계층 구조

하위 개체의 개체 E와 하위 개체에 D 대한 포인터를 사용하여 하위 개체에서 D 가장 왼쪽 A 하위 개체로 이동하려면 세 가지 변환을 수행할 수 있습니다. 포인터에서 D 포인터로 `dynamic_cast` E 변환한 다음 변환(또는 `dynamic_cast` 암시적 변환)을 대 E로 B 변환하고 마지막으로 암시적 변환을 B 수행할 수 있습니다. 예시:

C++

```

// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};

```

```

class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    E* pe = dynamic_cast<E*>(pd);
    B* pb = pe;    // upcast, implicit conversion
    A* pa = pb;    // upcast, implicit conversion
}

```

연산자를 `dynamic_cast` 사용하여 "교차 캐스트"를 수행할 수도 있습니다. 동일한 클래스 계층 구조를 사용하면 전체 개체가 형식인 한 포인터를 하위 개체에서 `B` 하위 개체로 캐스팅할 `D` 수 있습니다 `E`.

교차 캐스트를 고려할 때 포인터에서 가장 왼쪽 `A` 하위 개체에 대한 포인터 `D`로의 변환을 단 두 단계로 수행할 수 있습니다. 크로스 캐스트 `D`를 수행할 수 있습니다. `B` 그런 다음 암시적 변환을 `B` 수행합니다 `A`. 예시:

C++

```

// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    B* pb = dynamic_cast<B*>(pd);    // cross cast
    A* pa = pb;    // upcast, implicit conversion
}

```

null 포인터 값은 대상 형식의 null 포인터 값으로 변환됩니다 `dynamic_cast`.

사용할 `dynamic_cast < type-id > (expression)` 때 안전하게 형식 `type-id`으로 변환할 수 없는 경우 `expression` 런타임 검사 캐스팅이 실패합니다. 예시:

C++

```

// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = dynamic_cast<B*>(pa);    // fails at runtime, not safe;
    // B not derived from A
}

```

포인터 형식으로 캐스팅하지 못한 값은 null 포인터입니다. 참조 형식에 실패한 캐스트는 bad_cast 예외를 throw합니다. 유효한 개체를 가리키거나 참조하지 않으면 `expression` 예외가 `__non_rtti_object` throw됩니다.

예외에 대한 설명은 typeid를 참조하세요. `__non_rtti_object`

예시

다음 샘플에서는 개체(구조체 C)에 대한 기본 클래스(구조체 A) 포인터를 만듭니다. 이는 가상 함수가 있다는 사실과 함께 런타임 다형성을 가능하게 합니다.

또한 이 샘플은 계층 구조에서 비가상 함수를 호출합니다.

C++

```
// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
    }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
}
```

```

    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

    // fails because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

Output

```

in C
test2 in B
in GlobalTest
Can't cast to C

```

참고 항목

[캐스팅 연산자](#)

[키워드](#)

bad_cast 예외

아티클 • 2024. 11. 21.

bad_cast 예외는 참조 형식에 대한 캐스팅 실패의 결과로 연산자에 의해 `dynamic_cast` throw됩니다.

구문

```
catch (bad_cast)
    statement
```

설명

bad_cast 인터페이스는 다음과 같습니다.

C++

```
class bad_cast : public exception
```

다음 코드에는 bad_cast 예외를 throw하는 실패한 `dynamic_cast` 예제가 포함되어 있습니다.

C++

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class Shape {
public:
    virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
    virtual void virtualfunc() const {}
};

using namespace std;
int main() {
    Shape shape_instance;
```

```

Shape& ref_shape = shape_instance;
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}

```

캐스팅되는 개체(세이프)가 지정된 캐스트 형식(원)에서 파생되지 않으므로 예외가 throw 됩니다. 예외를 방지하려면 `main`에 다음과 같은 선언을 추가합니다.

C++

```

Circle circle_instance;
Circle& ref_circle = circle_instance;

```

그런 다음 블록의 캐스트 `try` 감각을 다음과 같이 반대로 바꿉니다.

C++

```

Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);

```

멤버

생성자

[\[\] 테이블 확장](#)

생성자	Description
<code>bad_cast</code>	<code>bad_cast</code> 형식의 개체에 대한 생성자입니다.

함수

[\[\] 테이블 확장](#)

함수	설명
<code>무엇</code>	미정

연산자

연산자	설명
operator=	한 <code>bad_cast</code> 개체를 다른 개체에 할당하는 할당 연산자입니다.

bad_cast

`bad_cast` 형식의 개체에 대한 생성자입니다.

C++

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

operator=

한 `bad_cast` 개체를 다른 개체에 할당하는 할당 연산자입니다.

C++

```
bad_cast& operator=(const bad_cast&) noexcept;
```

대상

C++

```
const char* what() const noexcept override;
```

참고 항목

[dynamic_cast 연산자](#)

[키워드](#)

[최신 C++ 예외 및 오류 처리 모범 사례](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

static_cast 연산자

아티클 • 2023. 10. 12.

식에 있는 형식에만 따라 식을 type-id 형식으로 변환합니다.

구문

```
static_cast <type-id> ( expression )
```

설명

표준 C++에서는 변환의 안전성을 보장하기 위해 런타임 형식 검사가 수행되지 않습니다. C++/CX에서는 컴파일 시간 및 런타임 검사가 수행됩니다. 자세한 내용은 [캐스팅](#)에 정의된 인터페이스의 private C++ 관련 구현입니다.

연산자는 `static_cast` 포인터를 기본 클래스로 변환하여 파생 클래스에 대한 포인터로 변환하는 등의 작업에 사용할 수 있습니다. 이러한 변환이 항상 안전한 것은 아닙니다.

일반적으로 열거형과 같은 숫자 데이터 형식을 ints 또는 ints와 같은 숫자 데이터 형식을 부동 소수점으로 변환하려는 경우에 사용 `static_cast` 하며 변환과 관련된 데이터 형식이 확실합니다. `static_cast` 변환은 검사 런타임 형식 `dynamic_cast` 이 없기 때문에 `static_cast` 변환만큼 `dynamic_cast` 안전하지 않습니다. 모호한 포인터에 대한 A `dynamic_cast` 는 실패하고 `static_cast` 아무 것도 잘못되지 않은 것처럼 반환됩니다. 이것은 위험할 수 있습니다. 변환은 더 안전 `dynamic_cast` 하지만 `dynamic_cast` 포인터 또는 참조에서만 작동하며 런타임 형식 검사 오버헤드입니다. 자세한 내용은 `dynamic_cast` Operator를 참조 [하세요](#).

다음에 나오는 예제에서 `D* pd2 = static_cast<D*>(pb);` 줄은 D에서 B에 있지 않은 필드와 메서드를 가질 수 있기 때문에 안전하지 않습니다. 그러나 `B* pb2 = static_cast<B*>(pd);` 는 항상 모든 D를 포함하기 때문에 B 줄의 변환은 안전합니다.

C++

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};
```

```

void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*>(pb);      // Not safe, D can have fields
                                            // and methods that are not in B.

    B* pb2 = static_cast<B*>(pd);      // Safe conversion, D always
                                            // contains all of B.
}

```

`dynamic_cast` 달리 변환 시 런타임 검사 없습니다. `pb static_cast`. `pb` 가 가리키는 개체는 유형 `D`의 개체가 아닐 수 있으며, 이 경우 `*pd2` 를 사용하는 것은 매우 위험합니다. 예를 들어 `D` 클래스가 아닌 `B` 클래스의 멤버인 함수를 호출하면 액세스 위반이 발생할 수 있습니다.

`dynamic_cast` 및 `static_cast` 연산자는 클래스 계층 전체에서 포인터를 이동합니다. 그러나 `static_cast` 캐스트 문에 제공된 정보에만 의존하므로 안전하지 않을 수 있습니다. 예시:

C++

```

// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
    virtual void Test(){}
};

class D : public B {};

void f(B* pb) {
    D* pd1 = dynamic_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}

```

`pb` 가 `D` 형식의 개체를 가리킬 경우 `pd1` 및 `pd2` 는 동일한 값을 얻습니다. 또한 `pb == 0` 일 경우 같은 값을 얻습니다.

전체 `D` 클래스 `dynamic_cast` 가 아닌 형식 `B` 의 개체를 가리키는 경우 `pb` 0 을 반환할 만큼 충분히 알 수 있습니다. 그러나 `static_cast` 형식 `D` 의 개체를 가리키고 단순히 해당 개체 `D` 에 대한 포인터를 반환하는 `pb` 프로그래머의 어설션에 의존합니다.

따라서 `static_cast` 암시적 변환의 역방향을 수행할 수 있으며, 이 경우 결과가 정의되지 않습니다. 변환 결과가 안전한지 확인하기 위해 프로그래머에게 `static_cast` 맡깁니다.

이 동작은 클래스 형식 이외의 형식에도 적용됩니다. 예를 들어 `int static_cast` 에서 .로 변환하는 `char` 데 사용할 수 있습니다. 그러나 결과 `char` 에서 전체 `int` 값을 보유하기에 충분한 비트가 없을 수 있습니다. 다시 말하지만, 변환 결과가 안전한지 확인하기 위해 프로그래머에게 `static_cast` 맡깁니다.

연산자를 `static_cast` 사용하여 표준 변환 및 사용자 정의 변환을 비롯한 암시적 변환을 수행할 수도 있습니다. 예시:

C++

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
    char ch;
    int i = 65;
    float f = 2.5;
    double dbl;

    ch = static_cast<char>(i);    // int to char
    dbl = static_cast<double>(f);  // float to double
    i = static_cast<BYTE>(ch);
}
```

연산자는 `static_cast` 정수 값을 열거형 형식으로 명시적으로 변환할 수 있습니다. 정수 계열 형식의 값이 열거형 값 범위에 속하지 않으면 결과 열거형 값이 정의되지 않습니다.

`static_cast` 연산자는 null 포인터 값을 대상 형식의 null 포인터 값으로 변환합니다.

모든 식은 연산자가 void `static_cast` 형식으로 명시적으로 변환할 수 있습니다. 대상 void 형식은 선택적으로, `volatile` 또는 `_unaligned` 특성을 포함할 `const` 수 있습니다.

연산자는 `static_cast`, `volatile` 또는 `_unaligned` 특성을 캐스팅 `const` 할 수 없습니다. 이러한 특성을 제거하는 방법에 대한 자세한 내용은 `const_cast` Operator를 [참조하세요](#).

C++/CLI: 재배치된 가비지 수집 `static_cast` 기 위에 검사 없는 캐스트를 수행할 위험이 있으므로 올바르게 작동한다고 확신하는 경우에만 성능이 중요한 코드에 사용해야 합니다. 릴리스 모드에서 사용해야 `static_cast` 하는 경우 성공을 보장하기 위해 디버그 빌드의 `safe_cast` 대체합니다.

참고 항목

[캐스팅 연산자](#)

[키워드](#)

const_cast 연산자

아티클 • 2023. 10. 12.

클래스에서 `const`, `volatile` 및 `_unaligned` 특성을 제거합니다.

구문

```
const_cast <type-id> (expression)
```

설명

개체 형식에 대한 포인터 또는 데이터 멤버에 대한 포인터는, `volatile` 및 `_unaligned` 한정자를 제외하고 `const` 동일한 형식으로 명시적으로 변환할 수 있습니다. 포인터 및 참조의 경우 결과는 원래 개체를 참조합니다. 데이터 멤버에 대한 포인터의 경우 결과는 데이터 멤버에 대한 원래(캐스팅 해제) 포인터와 동일한 멤버를 참조합니다. 참조 개체의 형식에 따라 결과 포인터, 참조 또는 데이터 멤버에 대한 포인터를 통한 쓰기 작업으로 인해 정의되지 않은 동작이 발생할 수 있습니다.

연산자를 `const_cast` 사용하여 상수 변수의 상수 상태 직접 재정의할 수 없습니다.

`const_cast` 연산자는 null 포인터 값을 대상 형식의 null 포인터 값으로 변환합니다.

예시

C++

```
// expre_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
    void setNumber( int );
    void printNumber() const;
private:
    int number;
};

void CCTest::setNumber( int num ) { number = num; }
```

```
void CCTest::printNumber() const {
    cout << "\nBefore: " << number;
    const_cast< CCTest * >( this )->number--;
    cout << "\nAfter: " << number;
}

int main() {
    CCTest X;
    X.setNumber( 8 );
    X.printNumber();
}
```

포인터가 포함된 줄에서 `const_cast` 포인터의 `this` 데이터 형식은 .입니다 `const CCTest` *. 연산자는 `const_cast` 포인터의 데이터 형식을 `this CCTest *` 변경하여 멤버 `number` 를 수정할 수 있도록 합니다. 캐스팅은 그것이 표시되는 문의 나머지 부분에서만 지속됩니다.

참고 항목

[캐스팅 연산자](#)

[키워드](#)

reinterpret_cast 연산자

아티클 • 2024. 08. 02.

포인터가 다른 포인터 형식으로 변환될 수 있도록 합니다. 또한 정수 계열 형식이 포인터 형식으로 변환될 수 있도록 하고 그 반대로도 변환될 수 있도록 합니다.

구문

```
reinterpret_cast < type-id > ( expression )
```

설명

운영자의 오용은 `reinterpret_cast` 쉽게 안전하지 않을 수 있습니다. 원하는 변환이 본질적으로 낮은 수준이 아닌 한 다른 캐스트 연산자 중 하나를 사용해야 합니다.

연산자는 `reinterpret_cast One_class*` `char*` `int*` `Unrelated_class*` 본질적으로 안전하지 않은 변환에 사용할 수 있습니다.

원래 형식으로 `reinterpret_cast` 다시 캐스팅되는 것 이외의 다른 용도로는 결과를 안전하게 사용할 수 없습니다. 다른 용도로 사용하는 경우에는 기껏해야 이식할 수 없는 결과가 생성됩니다.

연산자는 `reinterpret_cast`, `volatile` 또는 `_unaligned` 특성을 캐스팅 `const` 할 수 없습니다. 이러한 특성을 제거하는 방법에 대한 자세한 내용은 `const_cast Operator`를 참조하세요.

`reinterpret_cast` 연산자는 null 포인터 값을 대상 형식의 null 포인터 값으로 변환합니다.

실제로 사용하는 `reinterpret_cast` 것은 두 개의 고유 값이 거의 동일한 인덱스로 끝나지 않는 방식으로 값을 인덱스로 매핑하는 해시 함수입니다.

C++

```
#include <iostream>
using namespace std;

// Returns a hash code based on an address
unsigned short Hash( void *p ) {
    unsigned int val = reinterpret_cast<unsigned int>( p );
```

```
        return ( unsigned short )( val ^ (val >> 16));
    }

using namespace std;
int main() {
    int a[20];
    for ( int i = 0; i < 20; i++ )
        cout << Hash( a + i ) << endl;
}

Output:
64641
64645
64889
64893
64881
64885
64873
64877
64865
64869
64857
64861
64849
64853
64841
64845
64833
64837
64825
64829
```

포인터 `reinterpret_cast` 를 정수 형식으로 처리할 수 있습니다. 결과는 비트 이동되고 자신과 XOR 연산이 수행되어 고유한 인덱스(높은 확률로 고유함)를 생성합니다. 이 인덱스는 표준 C 스타일 캐스트를 통해 함수의 반환 형식으로 잘립니다.

참고 항목

[캐스팅 연산자](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

런타임 형식 정보

아티클 • 2023. 10. 12.

RTTI(런타임 형식 정보)는 프로그램 실행 중에 개체의 형식이 결정될 수 있도록 하는 메커니즘입니다. 많은 클래스 라이브러리 공급업체가 이 기능을 자체적으로 구현하고 있었기 때문에 RTTI가 C++ 언어에 추가되었습니다. 이 때문에 라이브러리 간에 호환되지 않는 문제가 발생하게 되었으므로 언어 수준에서 런타임 형식 정보에 대한 지원이 필요하다는 사실이 명백해졌습니다.

명확성을 위해 여기에서 RTTI에 대한 설명은 거의 전적으로 포인터에 국한됩니다. 하지만 설명된 개념은 참조에도 적용됩니다.

런타임 형식 정보에는 다음 세 가지 기본 C++ 언어 요소가 있습니다.

- [dynamic_cast](#) 연산자입니다.

다형 형식을 변환하는 데 사용됩니다.

- [typeid](#) 연산자입니다.

개체의 정확한 형식을 식별하는 데 사용됩니다.

- [type_info](#) [클래스입니다.](#)

연산자가 반환한 형식 정보를 보관하는 [typeid](#) 데 사용됩니다.

참고 항목

[캐스팅](#)

bad_typeid 예외

아티클 • 2023. 10. 12.

피연산자가 typeid NULL 포인터인 경우 typeid 연산자에 의해 bad_typeid 예외가 throw 됩니다.

구문

```
catch (bad_typeid)
    statement
```

설명

bad_typeid 인터페이스는 다음과 같습니다.

C++

```
class bad_typeid : public exception
{
public:
    bad_typeid();
    bad_typeid(const char * _Message = "bad typeid");
    bad_typeid(const bad_typeid &);
    virtual ~bad_typeid();

    bad_typeid& operator=(const bad_typeid&);

    const char* what() const;
};
```

다음 예제에서는 bad_typeid 예외를 throw하는 연산자를 보여줍니다 typeid.

C++

```
// expre_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class A{
public:
    // object for class needs vtable
    // for RTTI
    virtual ~A();
```

```
};

using namespace std;
int main() {
A* a = NULL;

try {
    cout << typeid(*a).name() << endl; // Error condition
}
catch (bad_typeid){
    cout << "Object is NULL" << endl;
}
}
```

출력

Output

Object is NULL

참고 항목

[런타임 형식 정보](#)

[키워드](#)

type_info 클래스

아티클 • 2023. 10. 12.

type_info 클래스는 컴파일러가 프로그램 내에서 생성된 형식 정보를 설명합니다. 이 클래스의 개체는 형식의 이름에 대한 포인터를 효과적으로 저장합니다. 또한 type_info 클래스는 같음 또는 정렬 순서에 대해 두 형식을 비교하는 데 적합한 인코딩된 값을 저장합니다. 형식의 인코딩 규칙 및 정렬 시퀀스는 지정되지 않으며 프로그램 간에 다를 수 있습니다.

<typeinfo> type_info 클래스를 사용하려면 헤더 파일을 포함해야 합니다. type_info 클래스의 인터페이스는 다음과 같습니다.

C++

```
class type_info {
public:
    type_info(const type_info& rhs) = delete; // cannot be copied
    virtual ~type_info();
    size_t hash_code() const;
    _CRTIMP_PURE bool operator==(const type_info& rhs) const;
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    _CRTIMP_PURE bool operator!=(const type_info& rhs) const;
    _CRTIMP_PURE int before(const type_info& rhs) const;
    size_t hash_code() const noexcept;
    _CRTIMP_PURE const char* name() const;
    _CRTIMP_PURE const char* raw_name() const;
};
```

클래스에 프라이빗 복사 생성자만 있으므로 type_info 클래스의 개체를 직접 인스턴스화 할 수 없습니다. (임시) type_info 개체를 생성하는 유일한 방법은 typeid 연산자를 사용하는 것입니다. 대입 연산자도 프라이빗으로 클래스 type_info 개체를 복사하거나 할당할 수 없습니다.

type_info::hash_code 는 typeinfo 형식의 값을 인덱스 값의 분포에 매핑하는 데 적합한 해시 함수를 정의합니다.

연산 == 자를 != 사용하여 각각 다른 type_info 개체와 같음 및 같지 않음을 비교할 수 있습니다.

형식의 정렬 순서와 상속 관계 간에 링크가 없습니다. 멤버 함수를 type_info::before 사용하여 형식의 정렬 시퀀스를 확인합니다. 서로 다른 프로그램이나 동일한 프로그램의 다른 실행에서 동일한 결과를 얻을 것이라는 보장 type_info::before 은 없습니다. 이러한 방식으로 type_info::before 주소 (&) 연산자는 비슷합니다.

멤버 함수는 `type_info::name` 사람이 읽을 수 있는 형식 이름을 나타내는 null로 끝나는 문자열을 반환 `const char*` 합니다. 가리키는 메모리는 캐시되며 직접 할당 해지되면 안 됩니다.

`type_info::raw_name` 멤버 함수는 Microsoft 전용입니다. 개체 형식의 `const char*` 데코레이팅된 이름을 나타내는 null로 끝나는 문자열을 반환합니다. 이름은 공간을 절약하기 위해 데코레이팅된 형식으로 저장됩니다. 따라서 이 함수는 이름을 취소할 필요가 없기 때문에 더 `type_info::name` 빠릅니다. 함수에서 반환된 `type_info::raw_name` 문자열은 비교 작업에 유용하지만 읽을 수 없습니다. 사람이 읽을 수 있는 문자열이 필요한 경우 대신 사용합니다 `type_info::name`.

형식 정보는 /GR(런타임 형식 정보 사용) [컴파일러 옵션이 지정된 경우에만](#) 다형 클래스에 대해 생성됩니다.

참고 항목

[런타임 형식 정보](#)

문 (C++)

아티클 • 2023. 10. 12.

C++ 명령문은 개체 조작 방법과 순서를 제어하는 프로그램 요소입니다. 이 단원에는 다음과이 포함됩니다.

- [개요](#)
- [레이블 문](#)
- [명령문 범주](#)
 - [식 문입니다.](#) 이 명령문은 식의 부작용 또는 해당 반환 값을 계산합니다.
 - [Null 문입니다.](#) 이 명령문은 C++ 구문에 필요하지만 별도의 작업이 필요하지 않은 명령문에 제공할 수 있습니다.
 - [복합 문입니다.](#) 이 명령문은 중괄호({})로 묶인 명령문의 그룹입니다. 단일 문이 사용되는 모든 경우에 사용할 수 있습니다.
 - [선택 문입니다.](#) 이 명령문은 테스트를 수행하고 테스트 결과가 true(0이 아닌)인 경우 코드의 한 섹션을 실행합니다. 테스트 결과가 false인 경우 다른 코드의 섹션을 실행할 수 있습니다.
 - [반복 문입니다.](#) 이 명령문은 지정된 종료 조건이 충족될 때까지 코드 블록을 반복 실행합니다.
 - [Jump 문입니다.](#) 이 명령문은 함수의 다른 위치로 제어를 즉시 전달하거나 함수로부터의 제어를 반환합니다.
 - [선언문입니다.](#) 선언은 프로그램에 이름을 제공합니다.

예외 처리 문에 대한 자세한 내용은 예외 처리를 참조 [하세요](#).

참고 항목

[C++ 언어 참조](#)

C++ 문 개요

아티클 • 2023. 10. 12.

C++ 문은 식 문, 선택 문, 반복 문 또는 점프 문이 해당 시퀀스를 특정하게 수정하는 경우를 제외하고 순차적으로 실행됩니다.

문은 다음과 같은 형식일 수 있습니다.

```
Labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch
```

대부분의 경우 C++ 문 구문은 ANSI C89의 구문과 동일합니다. 둘 사이의 주요 차이점은 C89에서 선언은 블록의 시작 부분에만 허용된다는 것입니다. C++는 이 제한을 효과적으로 제거하는 데 추가 *declaration-statement* 합니다. 이에 따라 미리 계산된 초기화 값이 계산될 수 있는 프로그램의 지점에 변수를 도입할 수 있습니다.

블록 내에 변수를 선언하면 해당 변수의 범위와 수명을 정확하게 제어할 수 있습니다.

문에 대한 문서에서는 다음 C++ 키워드(keyword) 설명합니다.

```
break
case
catch
continue
default
do

else
__except
__finally
for
goto

if
__if_exists
```

`_if_not_exists`

`_leave`

`return`

`switch`

`throw`

`_try`

`try`

`while`

참고 항목

[문](#)

레이블 문

아티클 • 2024. 11. 21.

레이블은 프로그램 제어를 지정된 문에 직접 전송하는 데 사용됩니다.

구문

Labeled-statement:

```
identifier : statement
case constant-expression : statement
default : statement
```

레이블의 범위는 선언된 전체 함수입니다.

설명

세 가지 형식의 레이블 문이 있습니다. 모두 콜론(:)을 사용하여 문과 일부 형식의 레이블을 구분합니다. 및 **default** 레이블은 **case** 사례 문과 관련이 있습니다.

C++

```
#include <iostream>
using namespace std;

void test_label(int x) {

    if (x == 1){
        goto label1;
    }
    goto label2;

label1:
    cout << "in label1" << endl;
    return;

label2:
    cout << "in label2" << endl;
    return;
}

int main() {
    test_label(1); // in label1
    test_label(2); // in label2
}
```

레이블 및 goto 문

소스 프로그램에서 레이블의 *identifier* 모양은 레이블을 선언합니다. 문만 컨트롤을 **goto** 레이블로 *identifier* 전송할 수 있습니다. 다음 코드 조각에서는 문 및 *identifier* 레이블의 **goto** 사용을 보여 줍니다.

레이블은 단독으로 표시할 수 없지만 항상 문에 연결해야 합니다. 레이블이 단독으로 필요한 경우 레이블 뒤에 null 문을 배치하세요.

레이블에는 함수 범위가 있으며 함수 내에서 다시 선언할 수 없습니다. 그러나 다른 함수에서 동일한 이름을 레이블로 사용할 수 있습니다.

C++

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

Test2:
    cerr << "At Test2 label." << endl;
}

//Output: At Test2 label.
```

case 문의 레이블

키워드 뒤의 레이블은 **case** 문 외부 **switch** 에도 나타날 수 없습니다. (이 제한은 키워드에도 적용됩니다 **default**.) 다음 코드 조각은 레이블의 **case** 올바른 사용을 보여 줍니다.

C++

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
```

```
    memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
    hDC = BeginPaint( hWnd, &ps );
    EndPaint( hWnd, &ps );
    break;

case WM_CLOSE:
    KillTimer( hWnd, TIMER1 );
    DestroyWindow( hWnd );
    if ( hWnd == hWndMain )
        PostQuitMessage( 0 ); // Quit the application.
    break;

default:
    // This choice is taken for all messages not specifically
    // covered by a case statement.
    return DefWindowProc( hWnd, Message, wParam, lParam );
break;
}
```

참고 항목

[C++ 문 개요](#)

[switch 문\(C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

식 문

아티클 • 2023. 10. 12.

식 문을 사용하면 식이 계산됩니다. 식 문의 결과로 어떠한 제어 전송 또는 반복도 발생하지 않습니다.

식 문의 구문은 간단하게 다음과 같습니다.

구문

```
[expression] ;
```

설명

다음 문이 실행되기 전에 식 문의 모든 식이 계산되고 의도하지 않은 모든 결과가 완료됩니다. 가장 일반적인 식 문은 대입 및 함수 호출입니다. 식은 선택 사항이므로 세미콜론만으로는 [null 문이라고 하는 빈 식 문으로](#) 간주됩니다.

참고 항목

[C++ 문 개요](#)

Null 문

아티클 • 2023. 10. 12.

"null 문"은 식이 누락된 식 문입니다. 언어의 구문에 문이 필요하지만 식 계산은 필요하지 않은 경우에 이 문이 유용합니다. 이 문은 세미콜론으로 구성됩니다.

null 문은 일반적으로 반복 문에서 자리 표시자로 사용되거나 복합 문 또는 함수의 끝에 레이블을 배치할 문으로 사용됩니다.

다음 코드 조각에서는 한 문자열을 다른 문자열로 복사하여 null 문을 통합하는 방법을 보여 줍니다.

C++

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ;    // Null statement.

    return DestStart;
}

int main()
{
}
```

참고 항목

[식 문](#)

복합 문(블록)

아티클 • 2023. 10. 12.

복합 문은 중괄호({ })로 묶인 0개 이상의 문으로 구성됩니다. 복합 문은 원하는 어느 곳에 서든 사용할 수 있습니다. 복합 문은 일반적으로 "블록"이라고 합니다.

구문

```
{ [ statement-list ] }
```

설명

다음 예제에서는 복합 문을 문의 문 부분으로 `if` 사용합니다(구문에 대한 자세한 내용은 [The if 문 참조](#)).

C++

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

① 참고

선언은 문이므로 선언은 문 목록의 문 중 하나일 수 있습니다. 결과적으로 복합 문 내에 선언되었지만 정적으로 명시적 선언되지 않은 이름은 지역 범위와 개체의 수명을 갖습니다. 로컬 범위를 사용하여 이를 처리에 대한 자세한 내용은 범위를 참조하세요.

참고 항목

[C++ 문 개요](#)

선택문 (C++)

아티클 • 2023. 10. 12.

C++ 선택 문 ([if](#) 및 [switch](#))은 코드 섹션을 조건부로 실행하는 방법을 제공합니다.

[_if_exists](#) 및 [_if_not_exists](#) 문을 사용하면 기호의 존재 여부에 따라 코드를 조건부로 포함할 수 있습니다.

각 문에 대한 개별 구문 항목을 참조하십시오.

참고 항목

[C++ 문 개요](#)

if-else 문(C++)

아티클 • 2024. 07. 08.

if-else 문은 조건부 분기를 제어합니다. *if-branch*의 문은 *condition*이 0이 아닌 값(또는 `true`)으로 계산되는 경우에만 실행됩니다. *condition*의 값이 0이 아닌 경우 다음 문이 실행되고 선택적 `else` 뒤의 문은 건너뛰됩니다. 그렇지 않으면 다음 문을 건너뛰고, `else`가 있으면 `else` 다음 문이 실행됩니다.

0이 아닌 것으로 계산되는 *condition* 식은 다음과 같습니다.

- `true`
- null이 아닌 포인터,
- 0이 아닌 산술 값, 또는
- 산술, 부울 또는 포인터 형식으로의 명확한 변환을 정의하는 클래스 형식입니다. (변환에 대한 자세한 내용은 [표준 변환](#)을 참조하세요.)

구문

init-statement:

`expression-statement`
`simple-declaration`

condition:

`expression`
`attribute-specifier-seq` *opt* `decl-specifier-seq` `declarator` `brace-or-equal-initializer`

statement:

`expression-statement`
`compound-statement`

expression-statement:

`expression` *opt* ;

compound-statement:

{ *opt* `statement-seq` }

statement-seq:

`statement`
`statement-seq` `statement`

```

if-branch:
    statement

else-branch:
    statement

selection-statement:
    if constexpropt17 ( init-statementopt17 condition ) if-branch
    if constexpropt17 ( init-statementopt17 condition ) if-branch else else-branch

```

¹⁷ 이 선택 요소는 C++17부터 사용할 수 있습니다.

if-else 문

모든 형태의 `if` 문의 경우 구조를 제외한 모든 값을 가질 수 있는 `condition`이 모든 파생 작업을 포함하여 계산됩니다. 실행된 `if-branch` 또는 `else-branch`에 `break`, `continue` 또는 `goto`가 포함되지 않는 한, `if` 문에서 프로그램의 다음 문으로 제어가 전달됩니다.

`if...else` 문의 `else` 절은 해당 `else` 문이 없는 동일한 범위에서 가장 가까운 이전 `if` 문과 연결되어 있습니다.

예시

이 샘플 코드는 `else` 유무에 관계없이 사용 중인 여러 `if` 문을 보여 줍니다.

```
C++

// if_else_statement.cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 10;

    if (x < 11)
    {
        cout << "x < 11 is true!\n"; // executed
    }
    else
    {
        cout << "x < 11 is false!\n"; // not executed
    }

    // no else statement
}
```

```

bool flag = false;
if (flag == true)
{
    x = 100; // not executed
}

int *p = new int(25);
if (p)
{
    cout << *p << "\n"; // outputs 25
}
else
{
    cout << "p is null!\n"; // executed if memory allocation fails
}
}

```

출력

출력

```

x < 11 is true!
25

```

이니셜라이저가 있는 if 문

C++17부터 `if` 문에는 명명된 변수를 선언하고 초기화하는 `init-statement` 식이 포함될 수도 있습니다. `if` 문 범위 내에서만 변수가 필요한 경우 이 형식의 `if` 문을 사용합니다.
Microsoft 전용: 이 양식은 Visual Studio 2017 버전 15.3부터 사용할 수 있으며 최소한 `/std:c++17` 컴파일러 옵션이 필요합니다.

예

C++

```

// Compile with /std:c++17

#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m{ {1, "one"}, {2, "two"}, {10,"ten"} };
mutex mx;
bool shared_flag = true; // guarded by mx

```

```

int getValue() { return 42; }

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second << "\n";
    }

    if (int x = getValue(); x == 42)
    {
        cout << "x is 42\n";
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        cout << "setting shared_flag to false\n";
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(),
keywords.end(), [&s](const char* kw) { return s == kw; }))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}

```

출력:

Output

```

ten
x is 42
setting shared_flag to false
Error! Token must not be a keyword

```

if constexpr 문

C++17부터 함수 템플릿에서 `if constexpr` 문을 사용하여 여러 함수 오버로드에 의존하지 않고도 컴파일 시간 분기 결정을 내릴 수 있습니다. **Microsoft 전용**: 이 양식은 Visual Studio 2017 버전 15.3부터 사용할 수 있으며 최소한 `/std:c++17` 컴파일러 옵션이 필요합니다.

예시

이 예에서는 전송된 형식에 따라 템플릿을 조건부로 컴파일하는 방법을 보여 줍니다.

C++

```
// Compile with /std:c++17
#include <iostream>

template<typename T>
auto Show(T t)
{
    //if (std::is_pointer_v<T>) // Show(a) results in compiler error for
    return *t. Show(b) results in compiler error for return t.
    if constexpr (std::is_pointer_v<T>) // This statement goes away for
Show(a)
{
    return *t;
}
else
{
    return t;
}
}

int main()
{
    int a = 42;
    int* pB = &a;

    std::cout << Show(a) << "\n"; // prints "42"
    std::cout << Show(pB) << "\n"; // prints "42"
}
```

`if constexpr` 문은 컴파일 시간에 계산되며 컴파일러는 함수 템플릿에 전송된 인수 형식과 일치하는 `if` 분기에 대한 코드만 생성합니다. `if constexpr` 문을 주석으로 처리하고 `if` 문의 주석 처리를 제거하면 컴파일러는 두 분기 모두에 대한 코드를 생성합니다. 즉, 오류가 발생합니다.

- `ShowValue(a);` 를 호출하면 `if` 문이 false이고 코드가 실행되지 않더라도 `t` 가 포인터가 아니기 때문에 `return *t` 에서 오류가 발생합니다.
- `ShowValue(pB);` 를 호출하면 `if` 문이 true이고 코드가 실행되지 않더라도 `t` 가 포인터이기 때문에 `return t` 에서 오류가 발생합니다.

함수 템플릿에 전송된 인수 형식과 일치하는 문만 컴파일되기 때문에 `if constexpr` 를 사용하면 이 문제가 해결됩니다.

출력

출력

참고 항목

[선택 문](#)

[키워드](#)

[switch Statement \(C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`__if_exists` 문

아티클 • 2023. 10. 12.

이 문은 `__if_exists` 지정된 식별자가 있는지 여부를 테스트합니다. 식별자가 있는 경우 지정된 문 블록이 실행됩니다.

구문

```
__if_exists ( identifier ) {  
    statements  
};
```

매개 변수

identifier

존재 여부를 테스트할 식별자입니다.

문을

식별자가 있는 경우 실행할 하나 이상의 문입니다.

설명

⊗ 주의

가장 신뢰할 수 있는 결과를 얻으려면 다음 제약 조건 아래의 문을 사용합니다

`__if_exists`.

- 템플릿이 `__if_exists` 아닌 단순 형식에만 문을 적용합니다.
- `__if_exists` 클래스 내부 또는 외부의 식별자에 문을 적용합니다. 문을 지역 변수에 `__if_exists` 적용하지 마세요.
- 함수 본 `__if_exists` 문에만 문을 사용합니다. 함수 본문 외부에서 문은 `__if_exists` 완전히 정의된 형식만 테스트할 수 있습니다.
- 오버로드된 함수에 대해 테스트할 때 특정 양식의 오버로드를 테스트할 수 없습니다.

문에 대한 `__if_exists` 보완은 `__if_not_exists` 문입니다.

예시

다음 예제에서는 템플릿을 사용하지만, 이는 권장되는 방법은 아닙니다.

C++

```
// the__if_exists_statement.cpp
// compile with: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
    void Dump() {
        std::cout << "In X<T>::Dump()" << std::endl;

        __if_exists(T::Dump) {
            T::Dump();
        }

        __if_not_exists(T::Dump) {
            std::cout << "T::Dump does not exist" << std::endl;
        }
    }
};

class A {
public:
    void Dump() {
        std::cout << "In A::Dump()" << std::endl;
    }
};

class B {};

bool g_bFlag = true;

class C {
public:
    void f(int);
    void f(double);
};

int main() {
    X<A> x1;
    X<B> x2;

    x1.Dump();
    x2.Dump();
```

```
__if_exists(::g_bFlag) {
    std::cout << "g_bFlag = " << g_bFlag << std::endl;
}

__if_exists(C::f) {
    std::cout << "C::f exists" << std::endl;
}

return 0;
}
```

출력

Output

```
In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists
```

참고 항목

[선택 문](#)

[키워드](#)

[__if_not_exists 문](#)

`__if_not_exists` 문

아티클 • 2023. 10. 12.

이 문은 `__if_not_exists` 지정된 식별자가 있는지 여부를 테스트합니다. 식별자가 없는 경우 지정된 문 블록이 실행됩니다.

구문

```
__if_not_exists ( identifier ) {  
    statements  
};
```

매개 변수

identifier

존재 여부를 테스트할 식별자입니다.

문을

식별자가 없는 경우 실행할 하나 이상의 문입니다.

설명

⊗ 주의

가장 신뢰할 수 있는 결과를 얻으려면 다음 제약 조건 아래의 문을 사용합니다

`__if_not_exists`.

- 템플릿이 `__if_not_exists` 아닌 단순 형식에만 문을 적용합니다.
- `__if_not_exists` 클래스 내부 또는 외부의 식별자에 문을 적용합니다. 문을 지역 변수에 `__if_not_exists` 적용하지 마세요.
- 함수 본 `__if_not_exists` 문에만 문을 사용합니다. 함수 본문 외부에서 문은 `__if_not_exists` 완전히 정의된 형식만 테스트할 수 있습니다.
- 오버로드된 함수에 대해 테스트할 때 특정 양식의 오버로드를 테스트할 수 없습니다.

문에 대한 `__if_not_exists` 보완은 `__if_exists` 문입니다.

예시

사용 `__if_not_exists` 방법에 대한 예제는 `__if_exists` 문을 참조 [하세요](#).

참고 항목

[선택 문](#)

[키워드](#)

[__if_exists 문](#)

switch 문(C++)

아티클 • 2024. 11. 21.

정수 계열 식의 값에 따라 코드의 여러 섹션 중에서 선택할 수 있습니다.

구문

:

`switch (init-statementoptC++17 condition) statement`

:

`expression-statement`

`simple-declaration`

:

`expression`

`attribute-specifier-seqopt decl-specifier-seq declarator brace-or-equal-initializer`

:

`case constant-expression : statement`

`default : statement`

설명

`switch` 문을 통해 컨트롤은 의 값에 따라 문 본문에서 하나의 `condition Labeled-statement`로 전송됩니다.

`condition`은 정수 형식이거나, 정수 형식으로 명확하게 변환되는 클래스 형식이어야 합니다. 정수 승격은 [표준 변환](#)에서 설명한 대로 이루어집니다.

`switch` 명령문 본문은 일련의 `case` 레이블과 선택적 `default` 레이블로 구성됩니다.

`Labeled-statement`는 다음 레이블 중 하나로, 뒤에 따르는 문입니다. 레이블 문은 구문적 요구 사항이 아니지만 `switch` 문은 레이블 문이 없으면 의미가 없습니다. `case` 문의 두 `constant-expression` 값은 동일한 값으로 계산할 수 없습니다. `default` 레이블은 한 번만 나타날 수 있습니다. `default` 문은 종종 끝에 배치되지만 `switch` 문 본문의 아무 곳에나 나타날 수 있습니다. `case` 또는 `default` 레이블은 `switch` 문 안에만 나타날 수 있습니다.

각 `case` 레이블의 `constant-expression`은 `condition`과 동일한 형식인 상수 값으로 변환됩니다. 그런 다음, `condition`과 같은지 비교됩니다. 컨트롤은 `condition` 값과 일치하는 `case` `constant-expression` 값 뒤의 첫 번째 문으로 전달됩니다. 결과적으로 발생하는 동작은 다음 표에 나와 있습니다.

switch 문 동작

테이블 확장

조건	작업
변환된 값이 승격된 제어 식의 값과 일치합니다.	제어가 해당 레이블 뒤에 오는 문으로 전송됩니다.
어떤 상수도 <code>case</code> 레이블의 상수와 일치하지 않으면, <code>default</code> 레이블이 있습니다.	컨트롤이 <code>default</code> 레이블로 전송됩니다.
어떤 상수도 <code>case</code> 레이블의 상수와 일치하지 않으며, <code>default</code> 레이블이 없습니다.	컨트롤이 <code>switch</code> 문 뒤의 문으로 전송됩니다.

일치하는 식을 찾으면 나중에 `case` 또는 `default` 레이블을 통해 실행을 계속할 수 있습니다. 실행을 중지하고 `switch` 문 뒤의 문으로 컨트롤을 전송하는 데 `break` 문이 사용됩니다. `break` 문이 없으면 `default`를 포함하여, 일치하는 `case` 레이블부터 `switch` 끝까지의 모든 문이 실행됩니다. 예시:

C++

```
// switch_statement1.cpp
#include <stdio.h>

int main() {
    const char *buffer = "Any character stream";
    int uppercase_A, lowercase_a, other;
    char c;
    uppercase_A = lowercase_a = other = 0;

    while ( c = *buffer++ ) // Walks buffer until NULL
    {
        switch ( c )
        {
            case 'A':
                uppercase_A++;
                break;
            case 'a':
                lowercase_a++;
                break;
            default:
                other++;
        }
    }
}
```

```

    }
}

printf_s( "\nUppercase A: %d\nLowercase a: %d\nTotal: %d\n",
    uppercase_A, lowercase_a, (uppercase_A + lowercase_a + other) );
}

```

위의 예제에서 `uppercase_A`는 `c`가 대문자 '`A`'인 경우 증가됩니다. `uppercase_A++` 뒤의 `break` 문이 `switch` 문 본문의 실행을 종료하고 컨트롤을 `while` 루프로 전달합니다. `break` 문이 없으면 실행이 레이블이 지정된 다음 문으로 "fall-through"되므로 `lowercase_a` 및 `other` 또한 증가하게 됩니다. `case 'a'`에 대한 `break` 문도 비슷한 역할을 합니다. `c`이 소문자 '`a`'인 경우 `lowercase_a`이 증가하며 `break` 명령문이 `switch` 명령문 본문을 종료합니다. `c`가 '`a`' 또는 '`A`'가 아닌 경우 `default` 문이 실행됩니다.

Visual Studio 2017 이상(`/std:c++17` 모드 이상에서 사용 가능): `[[fallthrough]]` 특성이 C++17 표준에서 지정됩니다. 이는 `switch` 문에서만 사용할 수 있습니다. fall-through 동작이 의도적이라는 점은 컴파일러 또는 코드를 읽는 모든 사람에게 힌트가 됩니다. Microsoft C++ 컴파일러는 현재 fallthrough 동작에 대해 경고하지 않으므로 이 특성은 컴파일러 동작에 영향을 주지 않습니다. 이 예에서 특성은 종료되지 않은 레이블이 있는 문 내의 빈 문에 적용됩니다. 즉, 세미콜론이 필요합니다.

```
C++

int main()
{
    int n = 5;
    switch (n)
    {

        case 1:
            a();
            break;
        case 2:
            b();
            d();
            [[fallthrough]]; // I meant to do this!
        case 3:
            c();
            break;
        default:
            d();
            break;
    }

    return 0;
}
```

Visual Studio 2017 버전 15.3 이상([/std:c++17](#) 모드 이상에서 사용 가능): `switch` 문에 세 미콜론으로 끝나는 `init-statement` 절이 있을 수 있습니다. 이는 범위가 `switch` 문의 블록으로 제한되는 변수를 도입하고 초기화합니다.

C++

```
switch (Gadget gadget(args); auto s = gadget.get_status())
{
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
}
```

`switch` 문의 내부 블록은 도달 가능한 한, 즉 가능한 모든 실행 경로에서 무시되지 않는 한 이니셜라이저가 있는 정의를 포함할 수 있습니다. 이러한 선언을 사용하여 정의된 이름에는 로컬 범위가 있습니다. 예시:

C++

```
// switch_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    switch( tolower( *argv[1] ) )
    {
        // Error. Unreachable declaration.
        char szChEntered[] = "Character entered was: ";

        case 'a' :
        {
            // Declaration of szChEntered OK. Local scope.
            char szChEntered[] = "Character entered was: ";
            cout << szChEntered << "a\n";
        }
        break;

        case 'b' :
        // Value of szChEntered undefined.
        cout << szChEntered << "b\n";
        break;

    default:
        // Value of szChEntered undefined.
        cout << szChEntered << "neither a nor b\n";
        break;
}
```

```
}
```

`switch` 문은 중첩될 수 있습니다. 중첩되는 경우 `case` 또는 `default` 레이블은 이들을 묶는 가장 가까운 `switch` 문에 연결됩니다.

Microsoft 전용 동작

Microsoft C++은 `switch` 문의 `case` 값의 수를 제한하지 않습니다. 이 수는 사용 가능한 메모리에 의해서만 제한됩니다.

참고 항목

[선택 문](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

반복 문 (C++)

아티클 • 2023. 10. 12.

반복 문을 사용하면 루프 종료 조건에 따라 문(또는 복합 문)이 0회 이상 실행됩니다. 이러한 문이 복합 문인 경우 [break 문 또는 continue 문](#)이 발생하는 경우를 제외하고 순서대로 실행됩니다.

C++는 4개의 반복 문을 제공합니다. 이러한 각 항목은 종료 식이 0(false)으로 평가되거나 루프 종료가 문으로 [break](#) 강제 적용될 때까지 반복됩니다. 다음 표에는 이러한 문과 해당 작업이 요약되어 있습니다. 각 문에 대해서는 뒤에 나오는 단원에서 자세히 설명합니다.

반복 문

문	평가 위치	초기화	증가
<code>while</code>	루프의 맨 위	아니요	아니요
<code>do</code>	루프의 맨 아래	아니요	아니요
<code>for</code>	루프의 맨 위	예	예
범위 기반	루프의 맨 위	예	예

반복 문의 문 부분은 선언일 수 없지만, 선언을 포함하는 복합 문일 수 있습니다.

참고 항목

[C++ 문 개요](#)

while 문 (C++)

아티클 • 2023. 10. 12.

식이 0으로 계산될 때까지 문을 반복적으로 실행합니다.

구문

```
while ( expression )
    statement
```

설명

식 테스트는 루프의 각 실행 전에 발생하므로 루프는 `while` 0회 이상 실행됩니다. 식은 정수 형식, 포인터 형식 또는 정수 또는 포인터 형식으로 명확하게 변환되는 클래스 형식이어야 합니다.

`while` 문 본문 내에서 중단, `goto` 또는 `반환`이 실행될 때 루프가 종료될 수도 있습니다. 루프를 종료하지 않고 현재 반복을 계속 종료합니다. `while`, `continue` 는 루프의 다음 반복에 컨트롤을 전달합니다 `while`.

다음 코드는 루프를 `while` 사용하여 문자열에서 후행 밑줄을 트리밍합니다.

C++

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    // Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    // iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';

    return szSource;
}
int main()
```

```
{  
    char szbuf[] = "12345____";  
  
    printf_s("\nBefore trim: %s", szbuf);  
    printf_s("\nAfter trim: %s\n", trim(szbuf));  
}
```

종료 조건은 루프의 맨 위에서 평가됩니다. 후행 밑줄이 없으면 루프가 실행되지 않습니다.

참고 항목

[반복 문](#)

[키워드](#)

[do-while 문\(C++\)](#)

[for 문\(C++\)](#)

[범위 기반 for 문\(C++\)](#)

do-while 문(C++)

아티클 • 2023. 10. 12.

지정된 종료 조건(식)이 0으로 평가될 때까지 문을 반복적으로 실행합니다.

구문

```
do
    statement
  while ( expression ) ;
```

설명

종료 조건의 테스트는 루프를 실행할 때마다 이루어집니다. 따라서 **do-while 루프는** 종료 식의 값에 따라 한 번 이상 실행됩니다. **do-while** 문은 문 본문 내에서 [break](#), [goto](#) 또는 [return](#) 문이 실행되는 경우에도 종료될 수 있습니다.

*expression*은 산술 형식이나 포인터 형식이어야 합니다. 다음과 같이 실행됩니다.

1. 문 본문이 실행됩니다.
2. 다음으로, *expression*이 평가됩니다. *expression*이 false인 경우 **do-while** 문이 종료되고 프로그램의 다음 문으로 제어가 전달됩니다. *expression*이 true(0이 아님)인 경우에는 프로세스가 1단계부터 반복됩니다.

예시

다음 샘플에서는 do-while 문을 보여 줍니다.

C++

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d", i++);
    }
```

```
    } while (i < 3);  
}
```

참고 항목

[반복 문](#)

[키워드](#)

[while 문\(C++\)](#)

[for 문\(C++\)](#)

[범위 기반 for 문\(C++\)](#)

for 문(C++)

아티클 • 2024. 07. 12.

조건이 `false`가 될 때까지 문을 반복적으로 실행합니다. 범위 기반 `for` 문에 대한 자세한 내용은 [범위 기반 for 문\(C++\)](#)을 참조하세요. C++/CLI `for each` 문에 대한 자세한 내용은 [for each, in](#)을(를) 참조하세요.

구문

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

설명

`for` 문을 사용하여 지정된 횟수만큼 실행해야 하는 루프를 생성합니다.

`for` 문은 다음 표와 같이 세 가지 선택적 부분으로 구성됩니다.

for 루프 요소

[+] 테이블 확장

구문 이름	실행 시	설명
<code>init-expression</code>	<code>for</code> 문의 다른 요소 앞에서는 <code>init-expression</code> 이 한 번만 실행됩니다. 그런 다음 <code>cond-expression</code> 으로 제어가 전달됩니다.	루프 인덱스를 초기화하는 데 자주 사용됩니다. 식 또는 선언을 포함할 수 있습니다.
<code>cond-expression</code>	첫 번째 반복을 포함한 <code>statement</code> 의 각 반복을 실행하기 전에 <code>statement</code> 은 <code>cond-expression</code> 이 <code>true</code> (0이 아님)로 평가될 때만 실행됩니다.	정수 형식으로 명확한 변환을 하는 정수 형식 또는 클래스 형식으로 평가되는 식입니다. 일반적으로 루프 종료 기준을 테스트하는 데 사용됩니다.
<code>Loop-expression</code>	<code>statement</code> 의 각 반복 끝에서 <code>Loop-expression</code> 이 실행된 후, <code>cond-expression</code> 이 평가됩니다.	일반적으로 루프 인덱스 증가에 사용됩니다.

다음 예제에서는 `for` 문을 사용하는 여러 가지 방법을 보여 줍니다.

```

#include <iostream>
using namespace std;

int main() {
    // The counter variable can be declared in the init-expression.
    for (int i = 0; i < 2; i++ ){
        cout << i;
    }
    // Output: 01
    // The counter variable can be declared outside the for loop.
    int i;
    for (i = 0; i < 2; i++){
        cout << i;
    }
    // Output: 01
    // These for loops are the equivalent of a while loop.
    i = 0;
    while (i < 2){
        cout << i++;
    }
    // Output: 01
}

```

init-expression 및 *Loop-expression*은 쉼표로 구분된 여러 개의 문을 포함할 수 있습니다. 예시:

C++

```

#include <iostream>
using namespace std;

int main(){
    int i, j;
    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {
        cout << "i + j = " << (i + j) << '\n';
    }
}
/* Output:
   i + j = 15
   i + j = 17
   i + j = 19
*/

```

*Loop-expression*은 증가 또는 감소하거나 다른 방식으로 수정될 수 있습니다.

C++

```

#include <iostream>
using namespace std;

```

```

int main(){
    for (int i = 10; i > 0; i--) {
        cout << i << ' ';
    }
    // Output: 10 9 8 7 6 5 4 3 2 1
    for (int i = 10; i < 20; i = i+2) {
        cout << i << ' ';
    }
}
// Output: 10 12 14 16 18

```

`for` 루프는 `statement` 내에서 `break`, `반환` 또는 `goto`(`for` 루프 외부의 레이블이 지정된 문으로)가 실행될 때 종료됩니다. `for` 루프의 `continue` 문은 현재 반복만 종료합니다.

`cond-expression`을 생략하면 `true`(으)로 간주되며 `for` 루프는 `statement` 내에서 `break`, `return` 또는 `goto` 없이 종료되지 않습니다.

`for` 문의 세 필드는 일반적으로 초기화, 종료 테스트, 증가에 사용되지만 이러한 용도에 제한된 것은 아닙니다. 예를 들어, 다음 코드는 0부터 4까지의 숫자를 인쇄합니다. 이 경우 `statement`은 null 문입니다.

C++

```

#include <iostream>
using namespace std;

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}

```

for 루프와 C++ 표준에 대해

C++ 표준에 따르면 `for` 루프에서 선언된 변수는 `for` 루프가 종료된 후 범위를 벗어납니다. 예시:

C++

```

for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope

```

기본적으로 /Ze에서 `for` 루프에서 선언된 변수는 `for` 루프의 바깥쪽 범위가 종료될 때까지 범위에 남아 있습니다.

/Zc:forScope에서는 /za를 지정하지 않아도 루프에 대해 선언된 표준 변수 동작을 사용할 수 있습니다.

`for` 루프의 범위 차이를 사용하여 /ze에서 다음과 같이 변수를 다시 선언하는 것도 가능합니다.

```
C++  
  
// for_statement5.cpp  
int main(){  
    int i = 0; // hidden by var with same name declared in for loop  
    for ( int i = 0 ; i < 3; i++ ) {}  
  
    for ( int i = 0 ; i < 3; i++ ) {}  
}
```

이 동작은 `for` 루프에서 선언된 변수의 표준 동작을 더 가깝게 모방합니다. 이 동작은 `for` 루프에서 필요한 변수가 루프가 모두 수행되면 범위를 벗어나야 합니다. 변수가 `for` 루프에서 선언되는 경우 컴파일러는 내부적으로 `for` 루프의 안에 있는 바깥쪽 범위에서 내부적으로 승격합니다. 이름이 같은 지역 변수가 이미 있는 경우에도 승격됩니다.

참고 항목

[반복 문](#)

[키워드](#)

[while 문\(C++\)](#)

[do-while 문\(C++\)](#)

[범위 기반 for 문\(C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

범위 기반 for 문(C++)

아티클 • 2024. 07. 08.

`statement`의 각 요소에 대해 `expression`를 반복적 및 순차적으로 실행합니다.

구문

```
for ( for-range-declaration : expression )  
    statement
```

설명

범위 기반의 `for` 문을 사용하여 범위에 대해 실행할 루프를 생성합니다. 범위는 `std::vector`와 같이 반복할 수 있는 모든 항목 또는 `begin()` 및 `end()`에 의해 범위가 정의된 기타 C++ 표준 라이브러리 시퀀스로 정의됩니다. `for-range-declaration` 부분에서 정의된 이름은 `for` 문에 대해 로컬이며 `expression` 또는 `statement`에서 재정의될 수 없습니다. 문의 `for-range-declaration` 부분에서는 `auto` 키워드가 기본 설정됩니다.

Visual Studio 2017의 새로운 기능: 범위 기반 `for` 루프에서는 더 이상 `begin()` 및 `end()` 가 동일한 형식의 개체를 반환하지 않아도 됩니다. 이 기능을 사용하면 `end()` 가 Ranges-v3 제안에 정의된 대로 범위에서 사용되는 sentinel 개체를 반환할 수 있습니다. 자세한 내용은 [Generalizing the Range-Based For Loop](#) (범위 기반 for 루프 일반화) 및 [range-v3 library on GitHub](#) (GitHub의 range-v3 라이브러리)를 참조하세요.

이 코드는 범위 기반 `for` 루프를 사용하여 배열과 벡터를 반복하는 방법을 보여 줍니다.

C++

```
// range-based-for.cpp  
// compile by using: cl /EHsc /nologo /W4  
#include <iostream>  
#include <vector>  
using namespace std;  
  
int main()  
{  
    // Basic 10-element integer array.  
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
    // Range-based for loop to iterate through the array.  
    for( int y : x ) { // Access by value using a copy declared as a  
                      // specific type.  
        // Not preferred.
```

```

        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is
        // needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-
    // place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}

```

다음과 같이 출력됩니다.

Output

```

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

```

```
0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159
```

9.14159
end of vector test

범위 기반 `for` 루프는 `statement`의 다음 중 하나(범위 기반 `for` 루프 외부의 레이블이 지정된 문에 대한 `break`, `return` 또는 `goto`)가 실행될 때 종료됩니다. 범위 기반 `for` 루프의 `continue` 문은 현재 반복만 종료합니다.

범위 기반 `for`의 경우 다음을 명심하세요.

- 배열을 자동으로 인식합니다.
- `.begin()` 및 `.end()` 가 포함된 컨테이너를 인식합니다.
- 기타 항목에 대해 `begin()` 및 `end()` 인수 종속성 조회를 사용합니다.

참고 항목

[auto](#)

[반복 문](#)

[키워드](#)

[while Statement \(C++\)](#)

[do-while Statement \(C++\)](#)

[for Statement \(C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

점프 문 (C++)

아티클 • 2023. 10. 12.

C++ 점프 문은 제어의 즉시 로컬 전송을 수행합니다.

구문

```
break;  
continue;  
return [expression];  
goto identifier;
```

설명

C++ 점프 문에 대한 설명은 다음 항목을 참조하십시오.

- [break 문](#)
- [continue 문](#)
- [return 문](#)
- [goto 문](#)

참고 항목

[C++ 문 개요](#)

break 문(C++)

아티클 • 2024. 02. 01.

문은 `break` 표시되는 가장 가까운 바깥쪽 루프 또는 조건문의 실행을 종료합니다. 제어는 종료된 문 뒤의 문이 있는 경우 전달됩니다.

구문

C++

```
break;
```

설명

문 `break` 은 조건 `switch` 문 및, `for` 및 루프 문과 `while` 함께 `do` 사용됩니다.

`switch` 문에서 이 문은 `break` 프로그램이 문 외부에서 `switch` 다음 문을 실행하도록 합니다. `break` 문이 없으면 절을 포함하여 `default` 일치하는 `case` 레이블에서 문 끝까지의 `switch` 모든 문이 실행됩니다.

루프에서 문은 `break` 가장 가까운 바깥쪽 `do` 또는 `for while` 문의 실행을 종료합니다. 종료된 문 뒤에 문이 있는 경우 제어가 해당 문으로 전달됩니다.

중첩된 문 내에서 문은 `break` 즉시 끝는, `for` 또는 `switch while` 문만 `do` 종료합니다. 또는 `goto` 문을 사용하여 `return` 더 깊이 중첩된 구조체에서 제어를 전송할 수 있습니다.

예시

다음 코드는 루프에서 `for` 문을 사용하는 `break` 방법을 보여줍니다.

C++

```
#include <iostream>
using namespace std;

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
```

```

    }
    cout << i << '\n';
}

// An example of a range-based for loop
int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int i : nums) {
    if (i == 4) {
        break;
    }
    cout << i << '\n';
}
}

```

Output

```

1
2
3
1
2
3

```

다음 코드는 루프 및 **do** 루프에서 **while** 사용하는 **break** 방법을 보여줍니다.

C++

```

#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }

    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}

```

Output

```
0  
1  
2  
3  
0  
1  
2  
3
```

다음 코드는 switch 문에서 사용하는 `break` 방법을 보여줍니다. 각 사례를 개별적으로 처리하려는 경우 모든 경우에 사용해야 `break` 합니다. 사용하지 `break` 않으면 코드 실행이 다음 사례로 넘어가게 됩니다.

C++

```
#include <iostream>  
using namespace std;  
  
enum Suit{ Diamonds, Hearts, Clubs, Spades };  
  
int main() {  
  
    Suit hand;  
    . . .  
    // Assume that some enum value is set for hand  
    // In this example, each case is handled separately  
    switch (hand)  
    {  
        case Diamonds:  
            cout << "got Diamonds \n";  
            break;  
        case Hearts:  
            cout << "got Hearts \n";  
            break;  
        case Clubs:  
            cout << "got Clubs \n";  
            break;  
        case Spades:  
            cout << "got Spades \n";  
            break;  
        default:  
            cout << "didn't get card \n";  
    }  
    // In this example, Diamonds and Hearts are handled one way, and  
    // Clubs, Spades, and the default value are handled another way  
    switch (hand)  
    {  
        case Diamonds:  
        case Hearts:  
            cout << "got a red card \n";
```

```
        break;
    case Clubs:
    case Spades:
    default:
        cout << "didn't get a red card \n";
    }
}
```

참고 항목

[선프 문](#)

[키워드](#)

[continue 문](#)

continue 문 (C++)

아티클 • 2023. 10. 12.

제어를 가장 작은 바깥쪽의 제어 식(`for` 또는 `while` 루프)으로 강제로 전송합니다.

구문

```
continue;
```

설명

현재 반복에서 나머지 모든 문은 실행되지 않습니다. 루프의 다음 반복은 다음과 같이 결정됩니다.

- `do` 또는 `while` 루프에서 다음 반복은 or `while` 문의 제어 식을 `do` 다시 평가하여 시작합니다.
- `for` 루프에서(구문을 `for(<init-expr> ; <cond-expr> ; <loop-expr>)` 사용하여 절이 `<loop-expr>` 실행됩니다. 그런 다음 `<cond-expr>` 절이 다시 계산되고 해당 결과에 따라 루프가 종료되거나 다른 반복이 발생합니다.

다음 예제에서는 문을 사용하여 코드 섹션을バイ패스하고 루프의 다음 반복을 시작하는 방법을 `continue` 보여줍니다.

예시

C++

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);
```

```
    printf_s("after the do loop\n");
}
```

Output

```
before the continue
before the continue
before the continue
after the do loop
```

참고 항목

[점프 문](#)

[키워드](#)

return 문 (C++)

아티클 • 2023. 10. 12.

함수 실행을 종료하고 컨트롤을 호출 함수(또는 `main` 함수에서 컨트롤을 이전하는 경우 운영 체제로)로 반환합니다. 호출 바로 다음 지점의 호출 함수에서 실행을 다시 시작합니다.

구문

```
return [expression];
```

설명

`expression` 절(있는 경우)은 초기화가 수행되고 있었던 것처럼 함수 선언에 지정된 형식으로 변환됩니다. 식 형식에서 함수 형식으로 변환하면 `return` 임시 개체를 만들 수 있습니다. 임시를 만드는 방법과 시기에 대한 자세한 내용은 임시 개체를 참조 [하세요](#).

`expression` 절 값이 호출 함수에 반환됩니다. 식을 생략하면 함수의 반환 값이 정의되지 않습니다. 생성자 및 소멸자 및 형식 `void`의 함수는 문에 `return` 식을 지정할 수 없습니다. 다른 모든 형식의 함수는 문에 `return` 식을 지정해야 합니다.

제어 흐름이 함수 정의를 둘러싸는 블록을 종료하면 식이 없는 문이 실행된 경우 `return` 와 결과가 동일합니다. 값을 반환할 때 선언되는 함수에는 올바르지 않습니다.

함수에는 임의의 개수의 `return` 문이 있을 수 있습니다.

다음 예제에서는 문과 함께 `return` 식을 사용하여 두 정수 중 가장 큰 값을 가져옵니다.

예시

C++

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
    return ( a > b ? a : b );
}
```

```
int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ));
}
```

참고 항목

[점프 문](#)

[키워드](#)

goto 문 (C++)

아티클 • 2023. 10. 12.

이 문은 `goto` 지정된 식별자가 레이블이 지정된 문으로 컨트롤을 무조건 전송합니다.

구문

```
goto identifier;
```

설명

`identifier`에 의해 지정된 레이블 문은 현재 함수에 있어야 합니다. 모든 `identifier` 이름은 내부 네임스페이스의 멤버이므로 다른 식별자를 방해하지 않습니다.

문 레이블은 문에 `goto` 만 의미가 있으며, 그렇지 않으면 문 레이블이 무시됩니다. 레이블을 다시 선언할 수 없습니다.

`goto` 문은 해당 위치의 범위에 있는 변수의 초기화를 건너뛰는 위치로 제어를 전송할 수 없습니다. 다음 예제에서는 C2362를 발생합니다.

C++

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

가능하면 문 대신 `goto`, `continue` 및 `return` 문을 사용하는 `break` 것이 좋은 프로그래밍 스타일입니다. 그러나 `break` 문이 루프의 한 수준에서만 종료되므로 문을 사용하여 `goto` 깊이 중첩된 루프를 종료해야 할 수 있습니다.

레이블 및 **goto** 문에 대한 자세한 내용은 레이블이 지정된 문을 참조 [하세요](#).

예시

이 예제에서 문은 **goto** 3이면 레이블이 지정된 지점으로 **stop i** 컨트롤을 전송합니다.

C++

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

Output

```
Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3
```

참고 항목

점프 문
키워드

컨트롤 전송

아티클 • 2023. 10. 12.

문이나 레이블 `switch` 을 `goto case` 사용하여 이니셜라이저를 지나 분기하는 프로그램을 지정할 수 있습니다. 이 같은 코드는 이니셜라이저를 포함한 선언이 점프 명령문이 있는 블록이 둘러싼 블록에 없는 경우 올바르지 않습니다.

다음 예제에서는 `total`, `ch` 및 `i` 개체를 선언하고 초기화하는 루프를 보여 줍니다. 이니셜라이저를 지나 컨트롤을 전송하는 잘못된 `goto` 문도 있습니다.

C++

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
    char MyArray[5] = {'2', '2', 'a', 'c'};
    int i = 0;
    while( 1 )
    {
        int total = 0;

        char ch = MyArray[i++];

        if ( ch >= '0' && ch <= '9' )
        {
            goto Label1;

            int i = ch - '0';
        Label1:
            total += i;    // C4700: transfers past initialization of i.
        } // i would be destroyed here if goto error were not present
    else
        // Break statement transfers control out of loop,
        // destroying total and ch.
        break;
    }
}
```

앞의 예제에서 문은 `goto` 초기화를 지나서 제어를 전송하려고 시도합니다 `i`. 단, `i`가 선언되었지만 초기화되지 않은 경우 전송은 유효합니다.

문 문 역할을 `while` 하는 블록에 선언된 개체 `total` 와 `ch` 해당 블록은 문을 사용하여 `break` 종료될 때 제거됩니다.

네임스페이스 (C++)

아티클 • 2023. 06. 16.

네임스페이스는 내부 식별자(형식, 함수, 변수 등의 이름)에 범위를 제공하는 선언적 영역입니다. 네임스페이스는 코드를 논리 그룹으로 구성하고 특히 코드베이스에 여러 라이브러리가 포함된 경우 발생할 수 있는 이름 충돌을 방지하는 데 사용됩니다. 네임스페이스 범위에 있는 모든 식별자는 한정 없이 서로에게 표시됩니다. 네임스페이스 외부의 식별자는 각 식별자에 대해 정규화된 이름을 사용하여 멤버에 액세스할 수 있습니다(예

`std::vector<std::string> vec;) 또는 단일 식별자에 대해 선언 사용 (using std::string) 또는 네임스페이스의 모든 식별자에 대한 using 지시문 (using namespace std;)을 사용하여 멤버에 액세스할 수 있습니다. 헤더 파일의 코드는 항상 정규화된 네임스페이스 이름을 사용해야 합니다.`

다음 예제에서는 네임스페이스 선언 및 네임스페이스 외부 코드가 해당 멤버에 액세스할 수 있는 세 가지 방법을 보여 줍니다.

C++

```
namespace ContosoData
{
    class ObjectManager
    {
        public:
            void DoSomething() {}
        };
        void Func(ObjectManager) {}
    }
}
```

정규화된 이름 사용:

C++

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

using 선언을 사용하여 하나의 식별자를 범위로 가져오기:

C++

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

using 지시문을 사용하여 네임스페이스의 모든 식별자를 범위로 가져오기:

C++

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

using 지시문

using 지시문을 사용하면 의 모든 이름을 네임스페이스 이름 namespace 없이 명시적 한정자로 사용할 수 있습니다. 네임스페이스에서 여러 식별자를 사용하는 경우 구현 파일(예: *.cpp)에서 using 지시문을 사용합니다. 식별자를 하나 또는 두 개만 사용하는 경우 네임스페이스의 모든 식별자가 아닌 scope 해당 식별자만 가져오는 using 선언을 고려합니다. 지역 변수의 이름이 네임스페이스 변수와 동일한 경우 네임스페이스 변수가 숨겨집니다. 전역 변수와 동일한 이름을 가진 네임스페이스 변수를 사용하면 오류가 발생합니다.

① 참고

using 지시문은 .cpp 파일 맨 위나(파일 범위) 클래스 또는 함수 정의 내에 배치할 수 있습니다.

일반적으로 헤더 파일(*.h)에는 using 지시문을 넣지 마세요. 해당 헤더를 포함하는 모든 파일이 네임스페이스의 모든 식별자를 범위로 가져오기 때문에 이름 충돌 및 이름 충돌 문제가 발생할 수 있으며, 디버그하기 매우 어렵습니다. 헤더 파일에는 항상 정규화된 이름을 사용하세요. 이름이 너무 길면 네임스페이스 별칭을 사용하여 축약할 수 있습니다. 다음을 참조하세요.

네임스페이스 및 네임스페이스 멤버 선언

일반적으로 헤더 파일에서 네임스페이스를 선언합니다. 함수 구현이 별도 파일에 있는 경우 이 예제와 같이 함수 이름을 한정합니다.

C++

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
```

```
    int Bar();  
}
```

contosodata.cpp의 함수 구현은 파일 맨 위에 지시문을 배치 `using` 하는 경우에도 정규화된 이름을 사용해야 합니다.

C++

```
#include "contosodata.h"  
using namespace ContosoDataServer;  
  
void ContosoDataServer::Foo() // use fully-qualified name here  
{  
    // no qualification needed for Bar()  
    Bar();  
}  
  
int ContosoDataServer::Bar(){return 0;}
```

단일 파일의 여러 블록과 여러 파일에서 네임스페이스를 선언할 수 있습니다. 컴파일러가 전처리 중에 파트를 결합하며, 결과로 생성된 네임스페이스에는 모든 파트에서 선언된 멤버가 모두 포함됩니다. 이러한 예로 표준 라이브러리의 각 헤더 파일에서 선언된 std 네임스페이스가 있습니다.

정의된 이름의 명시적 정규화로 선언된 명명된 네임스페이스의 멤버는 해당 네임스페이스의 외부에서 정의될 수 있습니다. 그러나 정의는 선언의 네임스페이스를 포함하는 네임스페이스의 선언 위치 다음에 표시되어야 합니다. 예:

C++

```
// defining_namespace_members.cpp  
// C2039 expected  
namespace V {  
    void f();  
}  
  
void V::f() { }           // ok  
void V::g() { }           // C2039, g() is not yet a member of V  
  
namespace V {  
    void g();  
}
```

이 오류는 여러 헤더 파일에서 네임스페이스 멤버가 선언되었으며 해당 헤더를 올바른 순서로 포함하지 않은 경우에 발생할 수 있습니다.

전역 네임스페이스

식별자가 명시적 네임스페이스에서 선언되지 않은 경우 암시적 전역 네임스페이스에 포함됩니다. 일반적으로 전역 네임스페이스에 있어야 하는 진입점 기본 Function을 제외하고 가능한 경우 전역 scope 선언하지 않도록 합니다. 전역 식별자를 명시적으로 한정하려면 `::SomeFunction(x);`과 같이 범위 확인 연산자를 이름 없이 사용합니다. 이렇게 하면 다른 네임스페이스에 있는 동일한 이름의 식별자와 해당 식별자가 구분되며, 다른 사용자가 코드를 더 쉽게 이해하는 데에도 도움이 됩니다.

std 네임스페이스

모든 C++ 표준 라이브러리 형식 및 함수는 예 `std` 중첩된 `std` 네임스페이스 또는 네임스페이스에 선언됩니다.

중첩된 네임스페이스

네임스페이스를 중첩할 수 있습니다. 일반 중첩 네임스페이스는 부모 멤버에 대한 정규화되지 않은 액세스 권한을 갖지만 부모 멤버는 다음 예제와 같이 중첩된 네임스페이스에 대한 정규화되지 않은 액세스 권한이 없습니다(인라인으로 선언되지 않는 한).

C++

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Ban() { return Foo(); }
    }

    int Bar(){...};
    int Baz(int i) { return Details::CountImpl; }
}
```

일반적인 중첩된 네임스페이스를 사용하여 부모 네임스페이스의 공용 인터페이스에 포함되지 않는 내부 구현 세부 정보를 캡슐화할 수 있습니다.

인라인 네임스페이스(C++ 11)

일반적인 중첩된 네임스페이스와 달리, 인라인 네임스페이스의 멤버는 부모 네임스페이스의 멤버로 처리됩니다. 이러한 특징 때문에 오버로드된 함수의 인수 종속 조회가 부모

및 중첩된 인라인 네임스페이스에 오버로드가 있는 함수에서 작동할 수 있습니다. 또한 인라인 네임스페이스에서 선언된 템플릿의 부모 네임스페이스에서 특수화를 선언할 수 있습니다. 다음 예제에서는 외부 코드가 기본적으로 인라인 네임스페이스에 바인딩하는 방법을 보여 줍니다.

C++

```
//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}
```

다음 예제에서는 인라인 네임스페이스에서 선언된 템플릿의 부모에서 특수화를 선언할 수 있는 방법을 보여 줍니다.

C++

```
namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
}
```

```

    }
    template<>
    class C<int> {};
}

```

인라인 네임스페이스를 버전 관리 메커니즘으로 사용하여 라이브러리의 공용 인터페이스에 대한 변경 내용을 관리할 수 있습니다. 예를 들어 단일 부모 네임스페이스를 만들고 인터페이스의 각 버전을 부모 안에 중첩된 자체 네임스페이스에 캡슐화할 수 있습니다. 가장 최근 버전이나 기본 설정 버전을 보유한 네임스페이스는 인라인으로 한정되므로 부모 네임스페이스의 직접 멤버인 것처럼 노출됩니다. Parent::Class를 호출하는 클라이언트 코드는 자동으로 새 코드에 바인딩합니다. 이전 버전을 사용하려는 클라이언트는 해당 코드가 있는 중첩된 네임스페이스의 정규화된 경로를 사용하여 계속 액세스할 수 있습니다.

컴파일 단위에서 네임스페이스의 첫 번째 선언에 `Inline` 키워드를 적용해야 합니다.

다음 예제에서는 각각 중첩된 네임스페이스에 있는 각 인터페이스의 두 버전을 보여 줍니다. `v_20` 네임스페이스는 `v_10` 인터페이스에서 약간 수정되었으며 인라인으로 표시됩니다. 새 라이브러리를 사용하고 `Contoso::Funcs::Add`를 호출하는 클라이언트 코드는 `v_20` 버전을 호출합니다. 코드에서 `Contoso::Funcs::Divide`를 호출하려고 하면 이제 컴파일 타임 오류가 발생합니다. 해당 함수가 실제로 필요한 경우 명시적으로 `Contoso::v_10::Funcs::Divide`를 호출하여 `v_10` 버전에 계속 액세스할 수 있습니다.

C++

```

namespace Contoso
{
    namespace v_10
    {
        template <typename T>
        class Funcs
        {
            public:
                Funcs(void);
                T Add(T a, T b);
                T Subtract(T a, T b);
                T Multiply(T a, T b);
                T Divide(T a, T b);
        };
    }

    inline namespace v_20
    {
        template <typename T>
        class Funcs
        {
            public:
                Funcs(void);
                T Add(T a, T b);
                T Subtract(T a, T b);
        };
    }
}

```

```
    T Multiply(T a, T b);
    std::vector<double> Log(double);
    T Accumulate(std::vector<T> nums);
}
}
```

네임스페이스 별칭

네임스페이스 이름은 고유해야 하며, 이는 대체로 이름이 너무 짧지 않아야 함을 의미합니다. 이름 길이로 인해 코드를 읽기 어렵거나 using 지시문을 사용할 수 없는 헤더 파일을 입력하는 것이 지루한 경우 실제 이름의 약어 역할을 하는 네임스페이스 별칭을 만들 수 있습니다. 예:

C++

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

익명 또는 명명되지 않은 네임스페이스

명시적 네임스페이스를 만들고 이름을 지정하지 않을 수 있습니다.

C++

```
namespace
{
    int MyFunc(){}
}
```

이 네임스페이스는 명명되지 않은 네임스페이스 또는 익명 네임스페이스라고 하며, 명명된 네임스페이스를 만들지 않고도 다른 파일의 코드에 변수 선언(즉, 내부 링크 제공)을 표시하지 않도록 하려는 경우에 유용합니다. 동일한 파일의 모든 코드에서 명명되지 않은 네임스페이스의 식별자를 볼 수 있지만, 해당 파일 외부 또는 보다 정확히 말해 변환 단위 외부에서는 네임스페이스 자체와 함께 식별자를 볼 수 없습니다.

추가 정보

[선언 및 정의](#)

열거형(C++)

아티클 • 2024. 07. 08.

열거형은 열거자라는 명명된 정수 상수 집합으로 구성된 사용자 정의 형식입니다.

① 참고

이 문서에서는 ISO 표준 C++ 언어 `enum` 형식과 C++11에 도입된 범위가 지정된(또는 강력한 형식의) `enum class` 형식을 다룹니다. C++/CLI 및 C++/CX의 `public enum class` 또는 `private enum class` 형식에 대한 자세한 내용은 [enum class\(C++/CLI 및 C++/CX\)](#)를 참조하세요.

구문

`enum-name`:

`identifier`

`enum-specifier`:

`enum-head { enumerator-listopt }`
`enum-head { enumerator-list , }`

`enum-head`:

`enum-key attribute-specifier-seqopt enum-head-nameopt enum-baseopt`

`enum-head-name`:

`nested-name-specifieropt identifier`

`opaque-enum-declaration`:

`enum-key attribute-specifier-seqopt enum-head-name enum-baseopt ;`

`enum-key`:

`enum`
`enum class`
`enum struct`

`enum-base`:

`: type-specifier-seq`

`enumerator-list`:

`enumerator-definition`

`enumerator-list` , `enumerator-definition`

`enumerator-definition:`

`enumerator`

`enumerator` = `constant-expression`

`enumerator:`

`identifier` `attribute-specifier-seq` opt

사용

C++

```
// unscoped enum:  
// enum [identifier] [: type] {enum-list};  
  
// scoped enum:  
// enum [class|struct] [identifier] [: type] {enum-list};  
  
// Forward declaration of enumerations (C++11):  
enum A : int;           // non-scoped enum must have type specified  
enum class B;           // scoped enum defaults to int but ...  
enum class C : short;    // ... may have any integral underlying type
```

매개 변수

`identifier`

열거형에 지정된 형식 이름입니다.

`type`

열거자의 기본 형식이며, 모든 열거자는 동일한 기본 형식을 갖습니다. 모든 정수 계열 형식이 가능합니다.

`enum-list`

열거형에서 열거자의 쉼표로 구분된 목록입니다. 범위에 있는 모든 열거자 또는 변수 이름은 고유해야 합니다. 하지만 값이 중복될 수 있습니다. 범위가 정해지지 않은 열거형에서 범위는 주변 범위이며 범위가 지정된 열거형에서 범위는 `enum-list` 자체입니다. 범위가 지정된 열거형에서는 목록이 비어 있을 수 있으며 이는 사실상 새 정수 형식을 정의합니다.

`class`

이 키워드를 선언에 사용하여, 열거형의 범위가 지정되고 `identifier`가 제공되도록 지정

합니다. `struct` 키워드를 `class` 대신 사용할 수도 있습니다. 이 컨텍스트에서는 이러한 키워드의 의미 체계가 같기 때문입니다.

열거자 범위

열거형은 명명된 상수로 표시되는 값 범위를 설명하는 컨텍스트를 제공합니다. 이러한 명명된 상수를 열거자라고도 합니다. 원래 C 및 C++ `enum` 형식에서는 `enum`이 선언된 범위 전체에서 한정되지 않은 열거자가 표시됩니다. 범위가 지정된 열거형에서 열거자 이름은 `enum` 형식 이름으로 한정되어야 합니다. 다음 예제에서는 두 가지 열거형의 이러한 기본적인 차이점을 보여 줍니다.

C++

```
namespace CardGame_Scoped
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Suit::Clubs) // Enumerator must be qualified by enum type
        { /*...*/}
    }
}

namespace CardGame_NonScoped
{
    enum Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Clubs) // Enumerator is visible without qualification
        { /*...*/}
    }
}
```

열거형의 각 이름은 열거형 값의 순서대로 해당 위치에 해당하는 정수 값이 할당됩니다. 기본적으로 첫 번째 값은 0이 할당되고 다음 값은 1이 할당되는 식이지만 여기에 나와 있는 대로 열거자의 값을 명시적으로 설정할 수 있습니다.

C++

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

`Diamonds` 열거자에 값 1이 할당됩니다. 명시적인 값이 지정되지 않은 경우 다음 열거자는 이전 열거자에 1을 더한 값을 받습니다. 앞의 예제에서 `Hearts`의 값은 2, `Clubs`의 값은 3 등이 될 수 있습니다.

모든 열거자는 상수로 처리되며 `enum`이 정의되는 범위 내에(범위가 지정되지 않은 열거형의 경우) 또는 `enum` 자체에(범위가 지정된 열거형의 경우) 고유한 이름이 있어야 합니다. 이름에 지정된 값은 고유할 필요가 없습니다. 예를 들어, 범위가 지정되지 않은 열거형 `Suit`의 다음 선언을 고려해보세요.

C++

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

`Diamonds`, `Hearts`, `Clubs`, `Spades`의 값은 각각 5, 6, 4, 5입니다. 5가 한 번 이상 사용되며 이는 의도한 것이 아니더라도 허용됩니다. 이러한 규칙은 범위가 지정된 열거형의 경우와 동일합니다.

캐스팅 규칙

범위가 지정되지 않은 열거형 상수를 `int`로 암시적으로 변환할 수 있지만, `int`는 열거형 값으로 암시적으로 변환할 수 없습니다. 다음 예제에서는 `hand`에 `Suit`가 아닌 값을 할당하면 어떻게 되는지를 보여 줍니다.

C++

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to
                     'Suit'
```

`int`를 범위가 지정된 열거자 또는 범위가 지정되지 않은 열거자로 변환하려면 캐스팅해야 합니다. 캐스팅하지 않고 범위가 지정된 열거자를 정수 값으로 승격할 수 있습니다.

C++

```
int account_num = Hearts; //OK if Hearts is in an unscoped enum
```

이러한 방식으로 암시적 변환을 사용할 경우 의도하지 않은 문제가 발생할 수 있습니다. 범위가 지정되지 않은 열거형과 관련된 프로그래밍 오류를 제거하기 위해 범위가 지정된 열거형 값은 강력한 형식입니다. 다음 예제에 표시된 것처럼 범위가 지정된 열거자는 열거형 형식 이름(식별자)으로 한정되어야 하며 암시적으로 변환할 수 없습니다.

C++

```
namespace ScopedEnumConversions
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void AttemptConversions()
    {
        Suit hand;
        hand = Clubs; // error C2065: 'Clubs' : undeclared identifier
        hand = Suit::Clubs; //Correct.
        int account_num = 135692;
        hand = account_num; // error C2440: '=' : cannot convert from 'int'
        to 'Suit'
        hand = static_cast<Suit>(account_num); // OK, but probably a bug!!!

        account_num = Suit::Hearts; // error C2440: '=' : cannot convert
        from 'Suit' to 'int'
        account_num = static_cast<int>(Suit::Hearts); // OK
    }
}
```

선 `hand = account_num;` 은 앞에서 보았듯이 범위가 지정되지 않은 열거형에 발생하는 오류를 초래합니다. 명시적 캐스트를 사용하면 허용됩니다. 그러나 범위가 지정된 열거형을 사용하여 다음 문 `account_num = Suit::Hearts;` 에서 시도된 변환은 명시적 캐스트 없이는 더 이상 허용되지 않습니다.

열거자가 없는 열거형

Visual Studio 2017 버전 15.3 이상([/std:c++17](#) 이상에서 사용 가능): 열거자 없이 명시적 기본 형식을 사용하여 열거형(일반 또는 범위가 지정된)을 정의하면 사실상 다른 형식으로의 암시적 변환이 없는 새로운 정수 형식을 도입할 수 있습니다. 기본 제공 기본 형식 대신 이 형식을 사용하면 의도치 않은 암시적 변환으로 인해 발생할 수 있는 미묘한 오류를 제거할 수 있습니다.

C++

```
enum class byte : unsigned char { };
```

새 형식은 기본 형식의 정확한 복사본이므로 동일한 호출 규칙을 갖습니다. 즉, 성능 저하 없이 ABI 전체에서 사용할 수 있습니다. 직접 목록 초기화를 사용하여 형식의 변수를 초기화하는 경우 캐스트가 필요하지 않습니다. 다음 예에서는 다양한 컨텍스트에서 열거자 없이 열거형을 초기화하는 방법을 보여 줍니다.

C++

```
enum class byte : unsigned char { };

enum class E : int { };
E e1{ 0 };
E e2 = E{ 0 };

struct X
{
    E e{ 0 };
    X() : e{ 0 } { }
};

E* p = new E{ 0 };

void f(E e) {}

int main()
{
    f(E{ 0 });
    byte i{ 42 };
    byte j = byte{ 42 };

    // unsigned char c = j; // C2440: 'initializing': cannot convert from
    // 'byte' to 'unsigned char'
    return 0;
}
```

참고 항목

[C 열거형 선언](#)
[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

union

아티클 • 2024. 07. 15.

① 참고

C++17 이상에서는 `std::variant` class가 union에 대해 형식이 안전한 대안이 됩니다.

`union`은 모든 멤버가 동일한 메모리 위치를 공유하는 사용자 정의 형식입니다. 이 정의는 어느 때라도 union에 멤버 목록의 개체가 둘 이상 포함될 수 없음을 의미합니다. 또한 union은 몇 개의 멤버가 있든 항상 가장 큰 멤버를 저장하는 데 충분한 메모리만 사용한다는 것을 의미합니다.

union은 개체는 많고 메모리는 제한된 경우 메모리를 보존하는 데 유용할 수 있습니다. 하지만 union을 올바르게 사용하려면 주의가 필요합니다. 항상 자신이 할당한 동일한 멤버에 액세스할 수 있도록 해야 합니다. 멤버 형식에 중요한 생성자가 있으면 코드를 작성하여 해당 멤버를 명시적으로 생성하고 삭제해야 합니다. struct union을 사용하기 전에 해결하려는 문제를 기본 class와 파생된 class 형식을 사용하여 더 잘 표현할 수 있는지 고려합니다.

구문

```
union tag opt{ member-list };
```

매개 변수

`tag`
union에 지정된 형식 이름입니다.

`member-list`
union에 포함할 수 있는 멤버입니다.

union 선언

`union` 키워드를 사용하여 union 선언을 시작하고 중괄호로 멤버 목록을 묶습니다.

C++

```

// declaring_a_union.cpp
union RecordType // Declare a simple union type
{
    char ch;
    int i;
    long l;
    float f;
    double d;
    int *int_ptr;
};

int main()
{
    RecordType t;
    t.i = 5; // t holds an int
    t.f = 7.25; // t now holds a float
}

```

union 사용

이전 예제에서 union에 액세스하는 모든 코드는 어떤 멤버가 데이터를 보유하는지 알아야 합니다. 이 문제에 대한 가장 일반적인 해결 방법은 차별된 union으로 불립니다. 이는 union을 struct로 묶고 현재 union에 저장된 멤버 형식을 나타내는 enum 멤버를 포함합니다. 다음 예제에서는 기본 패턴을 보여 줍니다.

C++

```

#include <queue>

using namespace std;

enum class WeatherDataType
{
    Temperature, Wind
};

struct TempData
{
    int StationId;
    time_t time;
    double current;
    double max;
    double min;
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
}

```

```

        short direction;
    };

    struct Input
    {
        WeatherDataType type;
        union
        {
            TempData temp;
            WindData wind;
        };
    };

// Functions that are specific to data types
void Process_Temp(TempData t) {}
void Process_Wind(WindData w) {}

void Initialize(std::queue<Input>& inputs)
{
    Input first;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    Input second;
    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

int main(int argc, char* argv[])
{
    // Container for all the data records
    queue<Input> inputs;
    Initialize(inputs);
    while (!inputs.empty())
    {
        Input const i = inputs.front();
        switch (i.type)
        {
            case WeatherDataType::Temperature:
                Process_Temp(i.temp);
                break;
            case WeatherDataType::Wind:
                Process_Wind(i.wind);
                break;
            default:
                break;
        }
        inputs.pop();
    }
    return 0;
}

```

이전 예에서 `Input struct`의 union은 이름이 없으므로 익명union으로 불립니다. 해당 멤버는 struct의 멤버인 것처럼 직접 액세스할 수 있습니다. 익명 union을 사용하는 방법에 관한 자세한 내용은 [익명 union 섹션](#)을 참조하세요.

이전 예에서는 공통 기본 class에서 파생되는 class 형식을 사용하여 해결할 수도 있는 문제를 보여줍니다. 컨테이너에 있는 각 개체의 런타임 형식에 따라 코드를 분기할 수 있습니다. 코드를 유지 관리하고 이해하기가 더 쉬울 수 있지만 union을 사용하는 것보다 느릴 수도 있습니다. 또한 union을 사용하면 관련 없는 형식을 저장할 수 있습니다. union은 union 변수 자체의 형식을 변경하지 않고도 저장된 값의 형식을 동적으로 변경할 수 있습니다. 예를 들어 요소가 서로 다른 형식의 다양한 값을 저장하는 다른 유형의 `MyUnionType` 배열을 만들 수 있습니다.

예에서 `Input struct`를 쉽게 오용할 수 있습니다. 데이터를 보유하는 멤버에 액세스하는 판별자를 올바르게 사용하는 것은 사용자에게 달려 있습니다. 다음 예와 같이 union을 `private`으로 설정하고 특별한 액세스 함수를 제공하면 잘못 사용되지 않게 보호할 수 있습니다.

무제한 union(C++11)

C++03 및 이전 버전에서는 사용자가 제공한 생성자, 제거자 또는 할당 연산자가 없는 한 class 형식이 있는 비static 데이터 멤버를 union에 포함할 수 있습니다. C++ 11에서는 이러한 제한이 제거됩니다. 이러한 멤버를 union에 포함하면 컴파일러는 사용자 제공이 아닌 특수 멤버 함수를 자동으로 `deleted`으로 표시합니다. union이 class 또는 struct 내의 익명 union인 경우 사용자 제공이 아닌 class 또는 struct의 특수 멤버 함수는 자동으로 `deleted`으로 표시됩니다. 다음 예에서는 이 사례를 처리하는 방법을 보여줍니다. union 멤버 중 하나에 이 특별한 처리가 필요한 멤버가 있습니다.

C++

```
// for MyVariant
#include <crtdbg.h>
#include <new>
#include <utility>

// for sample objects and output
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct A
{
    A() = default;
    A(int i, const string& str) : num(i), name(str) {}
```

```
int num;
string name;
//...
};

struct B
{
    B() = default;
    B(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    vector<int> vec;
    // ...
};

enum class Kind { None, A, B, Integer };

#pragma warning (push)
#pragma warning(disable:4624)
class MyVariant
{
public:
    MyVariant()
        : kind_(Kind::None)
    {
    }

    MyVariant(Kind kind)
        : kind_(kind)
    {
        switch (kind_)
        {
            case Kind::None:
                break;
            case Kind::A:
                new (&a_) A();
                break;
            case Kind::B:
                new (&b_) B();
                break;
            case Kind::Integer:
                i_ = 0;
                break;
            default:
                _ASSERT(false);
                break;
        }
    }

    ~MyVariant()
    {
        switch (kind_)
        {
    }
```

```

    case Kind::None:
        break;
    case Kind::A:
        a_.~A();
        break;
    case Kind::B:
        b_.~B();
        break;
    case Kind::Integer:
        break;
    default:
        _ASSERT(false);
        break;
    }
    kind_ = Kind::None;
}

MyVariant(const MyVariant& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(other.a_);
            break;
        case Kind::B:
            new (&b_) B(other.b_);
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
}

MyVariant(MyVariant&& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(move(other.a_));
            break;
        case Kind::B:
            new (&b_) B(move(other.b_));
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
    }
}

```

```

    default:
        _ASSERT(false);
        break;
    }
    other.kind_ = Kind::None;
}

MyVariant& operator=(const MyVariant& other)
{
    if (&other != this)
    {
        switch (other.kind_)
        {
        case Kind::None:
            this->~MyVariant();
            break;
        case Kind::A:
            *this = other.a_;
            break;
        case Kind::B:
            *this = other.b_;
            break;
        case Kind::Integer:
            *this = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
        }
    }
    return *this;
}

MyVariant& operator=(MyVariant&& other)
{
    _ASSERT(this != &other);
    switch (other.kind_)
    {
    case Kind::None:
        this->~MyVariant();
        break;
    case Kind::A:
        *this = move(other.a_);
        break;
    case Kind::B:
        *this = move(other.b_);
        break;
    case Kind::Integer:
        *this = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
    other.kind_ = Kind::None;
}

```

```
    return *this;
}

MyVariant(const A& a)
    : kind_(Kind::A), a_(a)
{
}

MyVariant(A&& a)
    : kind_(Kind::A), a_(move(a))
{
}

MyVariant& operator=(const A& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(a);
    }
    else
    {
        a_ = a;
    }
    return *this;
}

MyVariant& operator=(A&& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(move(a));
    }
    else
    {
        a_ = move(a);
    }
    return *this;
}

MyVariant(const B& b)
    : kind_(Kind::B), b_(b)
{
}

MyVariant(B&& b)
    : kind_(Kind::B), b_(move(b))
{
}

MyVariant& operator=(const B& b)
{
    if (kind_ != Kind::B)
    {
```

```

        this->~MyVariant();
        new (this) MyVariant(b);
    }
    else
    {
        b_ = b;
    }
    return *this;
}

MyVariant& operator=(B&& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(move(b));
    }
    else
    {
        b_ = move(b);
    }
    return *this;
}

MyVariant(int i)
: kind_(Kind::Integer), i_(i)
{
}

MyVariant& operator=(int i)
{
    if (kind_ != Kind::Integer)
    {
        this->~MyVariant();
        new (this) MyVariant(i);
    }
    else
    {
        i_ = i;
    }
    return *this;
}

Kind GetKind() const
{
    return kind_;
}

A& GetA()
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

const A& GetA() const

```

```

    {
        _ASSERT(kind_ == Kind::A);
        return a_;
    }

B& GetB()
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

const B& GetB() const
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

int& GetInteger()
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

const int& GetInteger() const
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

private:
    Kind kind_;
    union
    {
        A a_;
        B b_;
        int i_;
    };
};

#pragma warning (pop)

int main()
{
    A a(1, "Hello from A");
    B b(2, "Hello from B");

    MyVariant mv_1 = a;

    cout << "mv_1 = a: " << mv_1.GetA().name << endl;
    mv_1 = b;
    cout << "mv_1 = b: " << mv_1.GetB().name << endl;
    mv_1 = A(3, "hello again from A");
    cout << R"aaa(mv_1 = A(3, "hello again from A"): )aaa" <<
    mv_1.GetA().name << endl;
    mv_1 = 42;
    cout << "mv_1 = 42: " << mv_1.GetInteger() << endl;
}

```

```

b.vec = { 10,20,30,40,50 };

mv_1 = move(b);
cout << "After move, mv_1 = b: vec.size = " << mv_1.GetB().vec.size() <<
endl;

cout << endl << "Press a letter" << endl;
char c;
cin >> c;
}

```

union은 참조를 저장할 수 없습니다. union은 상속도 지원하지 않습니다. 이는 union을 기본 class로 사용하거나 다른 class에서 상속하거나 가상 함수를 가질 수 없음을 의미합니다.

union 초기화

중괄호 안에 식을 배치하여 동일한 문에서 union을 선언하고 초기화할 수 있습니다. 식이 계산되고 union의 첫 번째 필드에 할당됩니다.

C++

```

#include <iostream>
using namespace std;

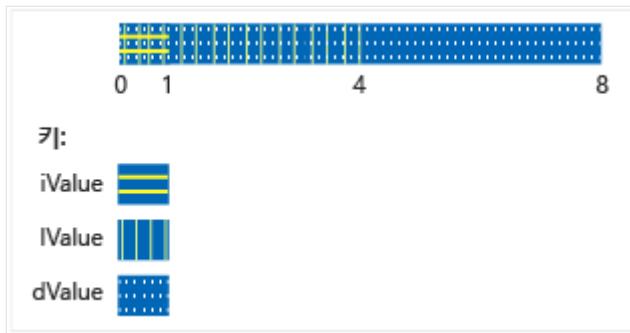
union NumericType
{
    short      iValue;
    long       lValue;
    double     dValue;
};

int main()
{
    union NumericType Values = { 10 };    // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}

/* Output:
10
3.141600
*/

```

NumericType union은 다음 그림과 같이 개념적으로 메모리에 배열됩니다.



익명 union

익명 union은 `class-name` 또는 `declarator-list` 없이 선언된 것입니다.

```
union { member-list }
```

익명 union에 선언된 이름은 비멤버 변수처럼 직접 사용됩니다. 이는 익명 union에 선언된 이름이 주변 범위에서 고유해야 함을 의미합니다.

익명 union은 다음과 같은 제한 사항이 적용됩니다.

- 파일 또는 네임스페이스 범위에서 선언된 경우 또한 `static`으로 선언되어야 합니다.
- `public` 멤버만 있을 수 있습니다. 익명 union에 `private` 멤버와 `protected` 멤버가 있으면 오류가 발생합니다.
- 멤버 함수를 가질 수 없습니다.

참고 항목

[클래스 및 구조체](#)

[키워드](#)

[class](#)

[struct](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

함수(C++)

아티클 • 2023. 10. 12.

함수는 일부 작업을 수행하는 코드 블록입니다. 함수는 호출자가 함수에 인수를 전달할 수 있도록 하는 입력 매개 변수를 필요에 따라 정의할 수 있습니다. 함수는 필요에 따라 출력으로 값을 반환할 수 있습니다. 함수는 이상적으로 함수의 기능을 명확하게 설명하는 이름을 사용하여 재사용 가능한 단일 블록에서 일반 작업을 캡슐화하는 데 유용합니다. 다음 함수는 호출자에서 두 개의 정수를 허용하고 해당 합계를 반환합니다. *a* 및 *b*는 형식 `int`의 매개 변수입니다.

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

함수는 프로그램의 여러 위치에서 호출하거나 호출할 수 있습니다. 함수에 전달되는 값은 함수 정의의 **매개 변수 형식과 호환되어야 하는** 인수입니다.

C++

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

함수 길이에 대한 실질적인 제한은 없지만 잘 정의된 단일 작업을 수행하는 함수를 위한 좋은 디자인이 목표입니다. 복잡한 알고리즘은 가능하면 이해하기 쉬운 더 간단한 함수로 세분화해야 합니다.

클래스 범위에서 정의되는 함수는 멤버 함수라고 합니다. 다른 언어와 달리 C++에서는 네임스페이스 범위(암시적 전역 네임스페이스 포함)에서 함수를 정의할 수도 있습니다. 이러한 함수는 자유 함수 또는 **비 멤버 함수**라고 하며 표준 라이브러리에서 광범위하게 사용됩니다.

함수가 오버로드될 수 있습니다. 즉, 형식 매개 변수의 수 및/또는 형식이 다른 경우 함수의 다른 버전이 동일한 이름을 공유할 수 있습니다. 자세한 내용은 [함수 오버로드](#)를 참조하세요.

함수 선언의 요소

최소 함수 선언은 컴파일러에 더 많은 지침을 제공하는 선택적 키워드(keyword)와 함께 반환 형식, 함수 이름 및 매개 변수 목록(비어 있을 수 있음)으로 구성됩니다. 다음 예제는 함수 선언입니다.

C++

```
int sum(int a, int b);
```

함수 정의는 선언과 중괄호 사이의 모든 코드인 본문으로 구성됩니다.

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

함수 선언과 뒤에 오는 세미콜론은 프로그램의 여러 위치에 나타날 수 있으며, 각 반환 단위에서 해당 함수를 호출하기 전에 나타나야 합니다. 함수 정의는 ODR(단일 정의 규칙)에 따라 프로그램에 한 번만 나타나야 합니다.

함수 선언의 필수 요소는 다음과 같습니다.

- 함수가 반환하는 값의 형식을 지정하거나 `void` 값이 반환되지 않는 경우를 지정하는 반환 형식입니다. C++11 `auto`에서는 컴파일러가 반환 문에서 형식을 유추하도록 지시하는 유효한 반환 형식입니다. C++14에서도 `decltype(auto)` 허용됩니다. 자세한 내용은 아래의 반환 형식에서 형식 추론을 참조하세요.
- 문자 또는 밑줄로 시작해야 하며 공백을 포함할 수 없는 함수 이름입니다. 일반적으로 표준 라이브러리 함수 이름의 선형 밑줄은 프라이빗 멤버 함수 또는 코드에서 사용할 수 없는 비 멤버 함수를 나타냅니다.
- 매개 변수 목록 - 중괄호로 구분되거나 쉼표로 구분된 0개 이상의 매개 변수 집합으로, 함수 본문 내에서 값에 액세스할 수 있는 형식 및 로컬 이름(선택 사항)을 지정합니다.

함수 선언의 선택적 요소는 다음과 같습니다.

- `constexpr` - 함수의 반환 값이 컴파일 시간에 계산할 수 있는 상수 값임을 나타냅니다.

C++

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

2. 링크 사양 `extern` 또는 `static`.

C++

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

자세한 내용은 번역 단위 및 연결을 참조 [하세요](#).

3. `inline`- 함수에 대한 모든 호출을 함수 코드 자체로 바꾸도록 컴파일러에 지시합니다. 인라인 처리는 성능이 중요한 코드 섹션에서 함수가 신속하게 실행되고 반복적으로 호출되는 시나리오의 성능 향상에 도움이 됩니다.

C++

```
inline double Account::GetBalance()
{
    return balance;
}
```

자세한 내용은 인라인 함수를 참조 [하세요](#).

4. `noexcept` 함수가 예외를 `throw`할 수 있는지 여부를 지정하는 식입니다. 다음 예제에서는 식이 계산되는 경우 함수에서 예외를 `is_pod true throw`하지 않습니다.

C++

```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

자세한 내용은 `noexcept`을 참조하세요.

5. (멤버 함수만 해당) 함수인지 여부를 지정하는 `cv` 한정자입니다 `const volatile`.

6. (멤버 함수에만 해당) `virtual` 또는 `override final`. `virtual` 는 함수를 파생 클래스에서 재정의할 수 있도록 지정합니다. `override`는 파생 클래스의 함수가 가상 함수를 재정의하고 있음을 의미합니다. `final` 는 함수를 더 이상 파생된 클래스에서 재정의할 수 없음을 의미합니다. 자세한 내용은 Virtual Functions를 참조 [하세요](#).

7. (멤버 함수만 해당) `static` 멤버 함수에 적용된다는 것은 함수가 클래스의 개체 인스턴스와 연결되지 않음을 의미합니다.

8. (비정적 멤버 함수만 해당) 암시적 개체 매개 변수(`*this`)가 rvalue 참조일 때 선택할 함수의 오버로드를 컴파일러에 지정하는 ref-qualifier입니다. 자세한 내용은 함수 오버로드를 참조 [하세요](#).

함수 정의

함수 정의는 변수 선언, 문 및 식을 포함하는 중괄호로 묶인 선언과 함수 본문으로 구성됩니다. 다음 예제에서는 전체 함수 정의를 보여줍니다.

C++

```
int foo(int i, std::string s)
{
    int value {i};
    MyClass mc;
    if(strcmp(s, "default") != 0)
    {
        value = mc.do_something(i);
    }
    return value;
}
```

본문 내에 선언된 변수는 지역 변수 또는 지역이라고 합니다. 이러한 변수는 함수가 종료될 때 범위를 벗어나므로 함수는 지역에 대한 참조를 반환할 수 없습니다.

C++

```
MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

const 및 constexpr 함수

멤버 함수 `const` 를 선언하여 함수가 클래스의 데이터 멤버 값을 변경할 수 없도록 지정 할 수 있습니다. 멤버 함수를 선언 `const` 하면 컴파일러가 `const-correctness`를 적용할 수 있습니다. 누군가가 실수로 선언된 `const` 함수를 사용하여 개체를 수정하려고 하면 컴파일러 오류가 발생합니다. 자세한 내용은 [const를 참조하세요.](#)

함수가 생성하는 값이 컴파일 시간에 결정될 수 있는 경우로 `constexpr` 함수를 선언합니다. `constexpr` 함수는 일반적으로 일반 함수보다 빠르게 실행됩니다. 자세한 내용은 [constexpr를 참조하세요.](#)

함수 템플릿

함수 템플릿은 클래스 템플릿과 유사하며 템플릿 인수에 따라 구체적인 함수를 생성합니다. 대부분의 경우 템플릿은 형식 인수를 유추할 수 있으므로 명시적으로 지정할 필요가 없습니다.

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895); // a is a double
auto b = Add2(string{ "Hello" }, string{ " World" }); // b is a std::string
```

자세한 내용은 [함수 템플릿을 참조하세요.](#)

함수 매개 변수 및 인수

함수에는 0개 이상 형식의 쉼표로 구분된 매개 변수 목록이 있으며, 각각에는 함수 본문 내에서 액세스할 수 있는 이름이 있습니다. 함수 템플릿은 더 많은 형식 또는 값 매개 변수를 지정할 수 있습니다. 호출자는 형식이 매개 변수 목록과 호환되는 구체적인 값인 인수를 전달합니다.

기본적으로 인수는 값으로 함수에 전달됩니다. 즉, 함수는 전달할 개체의 복사본을 받는다는 의미입니다. 큰 개체의 경우 복사본을 만드는데 비용이 많이 들 수 있으며 항상 필요한 것은 아닙니다. 인수가 참조(특히 lvalue 참조)로 전달되도록 하려면 매개 변수에 참조 수량자를 추가합니다.

C++

```
void DoSomething(std::string& input){...}
```

함수에서 참조로 전달된 인수를 수정하면 로컬 복사본이 아니라 원래 개체가 수정됩니다. 함수가 이러한 인수를 수정하지 못하도록 하려면 매개 변수를 const&로 한정합니다.

C++

```
void DoSomething(const std::string& input){...}
```

C++ 11: rvalue-reference 또는 lvalue-reference로 전달되는 인수를 명시적으로 처리하려면 매개 변수에 이중 앤퍼샌드를 사용하여 범용 참조를 나타냅니다.

C++

```
void DoSomething(const std::string&& input){...}
```

키워드(keyword) 인수 선언 목록의 첫 번째이자 유일한 멤버인 경우 매개 변수 선언 목록에서 단일 `void` 키워드(keyword) `void` 선언된 함수는 인수를 사용하지 않습니다. 목록의 다른 위치에 있는 형식 `void`의 인수는 오류를 생성합니다. 예시:

C++

```
// OK same as GetTickCount()
long GetTickCount( void );
```

여기서 설명된 경우를 제외하고 인수를 지정 `void` 하는 것은 불법이지만 형식에서 `void` 파생된 형식(예: 포인터 `void` 및 배열)은 인수 선언 목록의 `void` 아무 곳에나 나타날 수 있습니다.

기본 인수

함수 서명에서 마지막 매개 변수에 기본 인수를 할당할 수 있습니다. 즉, 일부 다른 값을 지정하려는 경우가 아니면 함수를 호출할 때 호출자가 인수를 제외할 수 있습니다.

C++

```
int DoSomething(int num,
    string str,
    Allocator& alloc = defaultAllocator)
{ ... }

// OK both parameters are at end
int DoSomethingElse(int num,
```

```

    string str = string{ "Working" },
    Allocator& alloc = defaultAllocator)
{ ... }

// C2548: 'DoMore': missing default parameter for parameter 2
int DoMore(int num = 5, // Not a trailing parameter!
           string str,
           Allocator& = defaultAllocator)
{...}

```

자세한 내용은 기본 인수를 참조 [하세요](#).

함수 반환 형식

함수는 다른 함수 또는 기본 제공 배열을 반환할 수 없습니다. 그러나 이러한 형식 또는 [함수 자체를 생성하는 람다](#)에 대한 포인터를 반환할 수 있습니다. 이러한 경우를 제외하고 함수는 범위에 있는 모든 형식의 값을 반환하거나 값을 반환하지 않을 수 있습니다. 이 경우 반환 형식이 됩니다 `void`.

후행 반환 형식

"일반" 반환 형식은 함수 서명의 왼쪽에 있습니다. [후행](#) 반환 형식은 서명의 오른쪽에 있으며 연산자가 앞에 옵니다 `->`. 후행 반환 형식은 반환 값의 형식이 템플릿 매개 변수에 따라 달라지는 경우 함수 템플릿에서 특히 유용합니다.

C++

```

template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}

```

후행 반환 형식과 함께 사용되는 경우 `auto` decltype 식이 생성하는 모든 항목에 대한 자리 표시자로만 사용되며 그 자체가 형식 추론을 수행하지 않습니다.

함수 지역 변수

함수 본문 내에서 선언된 변수를 지역 변수 또는 단순히 로컬 변수라고 부릅니다. 비정적 로컬은 함수 본문 내부에만 표시되며, 스택에 선언된 경우 함수가 종료될 때 벗어집니다. 지역 변수를 생성하고 값으로 반환하는 경우 컴파일러는 일반적으로 명명된 반환 값 최적화를 수행하여 불필요한 복사 작업을 방지할 수 있습니다. 지역 변수를 참조로 반

환하는 경우 해당 참조를 사용하려는 호출자의 모든 시도가 지역이 제거된 후 수행되므로 컴파일러에서 경고가 발생합니다.

C++에서는 지역 변수를 정적으로 선언할 수 있습니다. 이 변수는 함수 본문 내에만 표시되지만 함수의 모든 인스턴스에 대해 변수의 단일 복사본이 존재합니다. 로컬 정적 개체는 `atexit`로 지정된 종료 중에 소멸됩니다. 프로그램의 제어 흐름이 해당 선언을 무시했기 때문에 정적 개체가 생성되지 않은 경우 해당 개체를 삭제하려고 시도하지 않습니다.

반환 형식의 형식 추론(C++14)

C++14에서는 후행 반환 형식을 제공하지 않고도 함수 본문에서 반환 형식을 유추하도록 컴파일러에 지시하는 데 사용할 `auto` 수 있습니다. `auto` 항상 값별 반환을 추론합니다. `auto&&`를 사용하여 참조를 추론하도록 컴파일러에 지시합니다.

이 예제 `auto` 에서는 `lhs` 및 `rhs` 합계의 비-`const` 값 복사본으로 추론됩니다.

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

추론하는 `auto` 형식의 `const-ness`는 유지되지 않습니다. 반환 값이 인수의 `const-ness` 또는 `ref-ness`를 유지해야 하는 전달 함수의 경우 형식 유추 규칙을 사용하고 `decltype` 모든 형식 정보를 유지하는 키워드(keyword) 사용할 `decltype(auto)` 수 있습니다.

`decltype(auto)` 는 왼쪽의 일반 반환 값 또는 후행 반환 값으로 사용될 수 있습니다.

다음 예제(N3493의 코드 기반)는 템플릿이 인스턴스화될 때까지 알려지지 않은 반환 형식에서 함수 인수를 완벽하게 전달하도록 설정하는 데 사용되는 방법을 보여 `decltype(auto)` 줍니다.

C++

```
template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);
}

template<typename F, typename Tuple = tuple<T...>,
         typename Indices = make_index_sequence<tuple_size<Tuple>::value >>
decltype( auto )
apply(F&& f, Tuple&& args)
```

```
{  
    return apply_(std::forward<F>(f), std::forward<Tuple>(args), Indices());  
}
```

함수에서 여러 값 반환

함수에서 두 개 이상의 값을 반환하는 다양한 방법이 있습니다.

1. 명명된 클래스 또는 구조체 개체의 값을 캡슐화합니다. 호출자에게 클래스 또는 구조체 정의를 표시해야 합니다.

C++

```
#include <string>  
#include <iostream>  
  
using namespace std;  
  
struct S  
{  
    string name;  
    int num;  
};  
  
S g()  
{  
    string t{ "hello" };  
    int u{ 42 };  
    return { t, u };  
}  
  
int main()  
{  
    S s = g();  
    cout << s.name << " " << s.num << endl;  
    return 0;  
}
```

2. std::tuple 또는 std::pair 개체를 반환합니다.

C++

```
#include <tuple>  
#include <string>  
#include <iostream>  
  
using namespace std;  
  
tuple<int, string, double> f()
```

```

{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

int main()
{
    auto t = f();
    cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << endl;

    // --or--

    int myval;
    string myname;
    double mydecimal;
    tie(myval, myname, mydecimal) = f();
    cout << myval << " " << myname << " " << mydecimal << endl;

    return 0;
}

```

3. Visual Studio 2017 버전 15.3 이상 (모드 이상에서 `/std:c++17` 사용 가능): 구조적 바인딩을 사용합니다. 구조화된 바인딩의 장점은 반환 값을 저장하는 변수가 선언되는 동시에 초기화되므로 경우에 따라 훨씬 더 효율적일 수 있습니다. 문 `auto[x, y, z] = f();`에서 대괄호는 전체 함수 블록의 범위에 있는 이름을 도입하고 초기화합니다.

```

C++

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

struct S
{
    string name;
    int num;
};

S g()
{

```

```

        string t{ "hello" };
        int u{ 42 };
        return { t, u };
    }

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
    return 0;
}

```

4. 반환 값 자체를 사용하는 것 외에도 함수가 호출자가 제공하는 개체의 값을 수정하거나 초기화할 수 있도록 참조를 통해 사용할 매개 변수 수를 정의하여 값을 "반환"할 수 있습니다. 자세한 내용은 참조 형식 함수 인수를 참조하세요.

함수 포인터

C++은 C 언어와 동일한 방식으로 함수 포인터를 지원합니다. 그러나 일반적으로 함수 개체를 사용하면 형식이 보다 더 안전합니다.

함수 포인터 형식을 반환하는 함수를 선언하는 경우 함수 포인터 형식에 대한 별칭을 선언하는 데 사용하는 `typedef` 것이 좋습니다. 예를 들면 다음과 같습니다.

C++

```

typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer

```

이 작업이 수행되지 않으면 다음과 같이 식별자(`fp` 위 예제의 경우)를 함수 이름 및 인수 목록으로 바꿔 함수 포인터의 선언자 구문에서 함수 선언에 대한 적절한 구문을 추론할 수 있습니다.

C++

```

int (*myFunction(char* s))(int);

```

앞의 선언은 앞의 선언과 `typedef` 동일합니다.

참고 항목

함수 오버로드

가변 인수 목록을 사용하는 함수

명시적으로 기본 설정 및 삭제된 함수

함수에 대한 인수 종속 이름(Koenig) 조회

기본 인수

인라인 함수

가변 인수 목록을 사용하는 함수(C++)

아티클 • 2023. 10. 12.

마지막 멤버가 줄임표(...)인 함수 선언에서는 여러 가지 인수를 사용할 수 있습니다. 이러한 경우 C++에서는 명시적으로 선언된 인수에만 형식 검사를 제공합니다. 인수의 수와 형식까지 변경될 수 있는 정도의 일반적인 수준으로 함수를 만들어야 하는 경우 가변 인수 목록을 사용할 수 있습니다. 함수 패밀리는 변수 인수 목록을 사용하는 함수의 예입니다. `printf argument-declaration-list`

가변 인수를 사용하는 함수

선언된 후 인수에 액세스하려면 아래 설명된 대로 표준 include file <stdarg.h> 에 포함된 매크로를 사용합니다.

Microsoft 전용

Microsoft C++에서는 줄임표가 마지막 인수이고 줄임표 앞에 쉼표가 있는 경우 줄임표를 인수로 지정할 수 있습니다. 따라서 `int Func(int i, ...);` 선언은 올바르지만 `int Func(int i ...);`는 올바르지 않습니다.

Microsoft 전용 종료

일정하지 않은 수의 인수를 사용하는 함수의 선언에는 사용하지 않더라도 최소한 하나의 자리 표시자 인수가 있어야 합니다. 이 자리 표시자 인수가 제공되지 않은 경우 나머지 인수에 액세스할 수 있는 방법은 없습니다.

형식 `char` 의 인수가 변수 인수로 전달되면 형식 `int`으로 변환됩니다. 마찬가지로 형식 `float` 인수가 변수 인수로 전달되면 형식 `double`으로 변환됩니다. 다른 형식의 인수에는 일반적인 정수 계열 및 부동 소수점 확장이 적용됩니다. 자세한 내용은 표준 변환을 [참조하세요](#).

변수 목록이 필요한 함수는 인수 목록에서 줄임표(...)를 사용하여 선언합니다. `<stdarg.h>` 포함 파일에 설명된 형식 및 매크로를 사용하여 변수 목록에서 전달되는 인수에 액세스합니다. 이러한 매크로에 대한 자세한 내용은 `va_arg`, `va_copy`, `va_end`, `va_start` 참조하세요. 참조하세요.

다음 예제에서는 매크로가 형식(`<stdarg.h>`)에 <선언됨>과 함께 작동하는 방법을 보여 줍니다.

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
// "ifcs", where each character specifies the
// type of the argument in that position.
//
// i = int
// f = float
// c = char
// s = string (char *)
//
// Following the format specification is a variable
// list of arguments. Each argument corresponds to
// a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;

    // szTypes is the last argument specified; you must access
    // all others using the variable-argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i ) {
        union Printable_t {
            int     i;
            float   f;
            char    c;
            char   *s;
        } Printable;

        switch( szTypes[i] ) {    // Type to expect.
            case 'i':
                Printable.i = va_arg( vl, int );
                printf_s( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = va_arg( vl, double );
                printf_s( "%f\n", Printable.f );
                break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
        }
    }
}
```

```
break;

case 's':
    Printable.s = va_arg( vl, char * );
    printf_s( "%s\n", Printable.s );
break;

default:
break;
}
}

va_end( vl );
}

//Output:
// 32.400002
// a
// Test string
```

앞의 예제는 다음과 같은 중요한 개념을 설명합니다.

1. 가변 인수에 액세스하기 전에 `va_list` 형식의 변수로 목록 표시를 설정해야 합니다.
앞의 예제에서 표식 이름은 `vl`입니다.
2. `va_arg` 매크로를 사용하여 각각의 인수에 액세스합니다. 스택에서 올바른 바이트
수가 전송할 수 있도록 `va_arg` 매크로에 검색 인수의 형식을 설명해야 합니다. 호출
프로그램에서 제공한 것과 다른 잘못된 크기 형식을 `va_arg`로 지정하는 경우 결과
를 예측할 수 없습니다.
3. `va_arg` 매크로를 원하는 형식에 사용하여 얻은 결과를 명시적으로 캐스팅해야 합니
다.

가변 인수 처리를 종료하려면 매크로를 호출해야 합니다. `va_end`

함수 오버로드

아티클 • 2023. 10. 12.

C++를 사용하면 동일한 범위에서 동일한 이름의 함수를 둘 이상 지정할 수 있습니다. 이러한 함수를 오버로드된 함수 또는 오버로드라고 합니다. 오버로드된 함수를 사용하면 인수의 형식과 수에 따라 함수에 대해 서로 다른 의미 체계를 제공할 수 있습니다.

예를 들어 인수를 `print` 사용하는 함수를 고려해 `std::string` 보세요. 이 함수는 형식 `double`의 인수를 사용하는 함수와는 매우 다른 작업을 수행할 수 있습니다. 오버로드를 사용하면 이름(예: `print_string` 또는 `print_double`)을 사용할 필요가 없도록 합니다. 컴파일 시 컴파일러는 호출자가 전달한 인수의 형식 및 수에 따라 사용할 오버로드를 선택합니다. 호출 `print(42.0)` 하면 함수가 `void print(double d)` 호출됩니다. 호출 `print("hello world")` 하면 오버로드가 `void print(std::string)` 호출됩니다.

멤버 함수와 자유 함수를 모두 오버로드할 수 있습니다. 다음 표에서는 C++가 동일한 범위에서 이름이 같은 함수 그룹을 구분하는 데 사용하는 함수 선언 부분을 보여 줍니다.

오버로드 고려 사항

함수 선언 요소	오버로드에 사용하나요?
함수 반환 형식	아니요
인수의 수	예
인수 형식	예
줄임표의 존재 여부	예
<code>typedef</code> 이름 사용	아니요
지정하지 않은 배열 범위	아니요
<code>const</code> 또는 <code>volatile</code>	예, 전체 함수에 적용된 경우
참조 한정자(& 및 &&)	예

예시

다음 예제에서는 함수 오버로드를 사용하는 방법을 보여 줍니다.

C++

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>
#include <string>

// Prototype three print functions.
int print(std::string s);           // Print a string.
int print(double dvalue);          // Print a double.
int print(double dvalue, int prec); // Print a double with a
                                  // given precision.

using namespace std;
int main(int argc, char *argv[])
{
    const double d = 893094.2987;
    if (argc < 2)
    {
        // These calls to print invoke print( char *s ).
        print("This program requires one argument.");
        print("The argument specifies the number of");
        print("digits precision for the second number");
        print("printed.");
        exit(0);
    }

    // Invoke print( double dvalue ).
    print(d);

    // Invoke print( double dvalue, int prec ).
    print(d, atoi(argv[1]));
}

// Print a string.
int print(string s)
{
    cout << s << endl;
    return cout.good();
}

// Print a double in default precision.
int print(double dvalue)
{
    cout << dvalue << endl;
    return cout.good();
}

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.
int print(double dvalue, int prec)
{
    // Use table-lookup for rounding/truncation.
```

```

static const double rgPow10[] = {
    10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1,
    10E0, 10E1, 10E2, 10E3, 10E4, 10E5, 10E6 };
const int iPowZero = 6;

// If precision out of range, just print the number.
if (prec < -6 || prec > 7)
{
    return print(dvalue);
}
// Scale, truncate, then rescale.
dvalue = floor(dvalue / rgPow10[iPowZero - prec]) *
    rgPow10[iPowZero - prec];
cout << dvalue << endl;
return cout.good();
}

```

앞의 코드는 파일 범위에서 함수의 오버로드를 `print` 보여 줍니다.

기본 인수는 함수 형식의 일부로 간주되지 않습니다. 따라서 오버로드된 함수를 선택하는 데 사용되지 않습니다. 해당 기본 인수에만 다른 두 개의 함수는 오버로드된 함수 대신 여러 정의로 간주됩니다.

오버로드된 연산자는 기본 인수를 제공할 수 없습니다.

인수 일치

컴파일러는 현재 범위의 함수 선언과 함수 호출에 제공된 인수 중에서 가장 일치하는 값을 기준으로 호출할 오버로드된 함수를 선택합니다. 적절한 함수를 찾으면 함수가 호출됩니다. 이 컨텍스트에서 "적합"은 다음 중 하나를 의미합니다.

- 정확히 일치하는 항목을 찾았습니다.
- 간단한 변환을 수행했습니다.
- 정수 계열 확장이 수행되었습니다.
- 원하는 인수 형식에 대한 표준 변환이 존재합니다.
- 원하는 인수 형식으로의 사용자 정의 변환(변환 연산자 또는 생성자)이 있습니다.
- 줄임표로 나타낸 인수를 찾았습니다.

컴파일러가 각 인수의 후보 함수 집합을 만듭니다. 후보 함수는 해당 위치의 실제 인수를 형식 인수의 형식으로 변환할 수 있는 함수입니다.

"가장 일치하는 함수" 집합이 각 인수에 대해 빌드되고 모든 집합에 공통된 함수가 선택됩니다. 공통된 함수가 2개 이상일 경우 오버로드가 모호해지고 오류를 생성합니다. 최종

적으로 선택된 함수는 하나 이상의 인수에 대해 그룹의 다른 모든 함수보다 항상 더 나은 일치입니다. 명확한 승자가 없으면 함수 호출에서 컴파일러 오류가 생성됩니다.

다음과 같은 선언이 있습니다. 알아보기 쉽게 함수가 Variant 1, Variant 2 및 Variant 3으로 표시되었습니다.

C++

```
Fraction &Add( Fraction &f, long l );           // Variant 1
Fraction &Add( long l, Fraction &f );           // Variant 2
Fraction &Add( Fraction &f, Fraction &f );       // Variant 3

Fraction F1, F2;
```

다음과 같은 문이 있습니다.

C++

```
F1 = Add( F2, 23 );
```

위의 문에서는 두 집합을 빌드합니다.

집합 1: 형식의 첫 번째 인수가 있는 후보 함수 Fraction

변형 1

Set 2: 두 번째 인수를 형식으로 변환할 수 있는 후보 함수 int

Variant 1(int 표준 변환을 사용하여 변환할 long 수 있습니다).

변형 3

Set 2의 함수는 실제 매개 변수 형식에서 정식 매개 변수 형식으로 암시적으로 변환되는 함수입니다. 이러한 함수 중 하나에는 실제 매개 변수 형식을 해당 형식 매개 변수 형식으로 변환하는 가장 작은 "비용"이 있습니다.

두 집합의 공통된 함수는 변형 1입니다. 다음은 모호한 함수 호출의 예입니다.

C++

```
F1 = Add( 3, 6 );
```

앞의 함수 호출에서 다음 집합을 빌드합니다.

집합 1: 형식의 첫 번째 인수가 있는 후보 함수

`int`

Variant 2(`int` 표준 변환을 사용하여 변환할
`long` 수 있습니다.)

집합 2: 형식의 두 번째 인수가 있는 후보 함수

`int`

Variant 1(`int` 표준 변환을 사용하여 변환할
`long` 수 있습니다.).

이 두 집합의 교집합이 비어 있으므로 컴파일러는 오류 메시지를 생성합니다.

인수 일치의 경우 n 개의 기본 인수가 있는 함수는 각각 인수 수가 다른 $n+1$ 개별 함수로 처리됩니다.

줄임표(...)는 야생으로 작동하며 카드 실제 인수와 일치합니다. 오버로드된 함수 집합을 매우 주의하여 디자인하지 않으면 많은 모호한 집합이 발생할 수 있습니다.

① 참고

오버로드된 함수의 모호성은 함수 호출이 발생할 때까지 확인할 수 없습니다. 이때 함수 호출의 각 인수에 대해 집합이 빌드되고 명확한 오버로드가 존재하는지 여부를 확인할 수 있습니다. 즉, 특정 함수 호출에 의해 호출될 때까지 코드에서 모호성이 다시 기본 수 있습니다.

인수 형식 차이

오버로드된 함수는 다른 이니셜라이저를 사용하는 인수 형식을 구분합니다. 따라서, 주어진 형식의 인수와 그 형식에 대한 참조는 오버로드 목적의 형식과 동일한 것으로 간주됩니다. 동일한 이니셜라이저를 사용하므로 동일한 것으로 간주됩니다. 예를 들어, `max(double, double)` 는 `max(double &, double &)` 와 동일한 것으로 간주됩니다. 이러한 두 함수를 선언하면 오류가 발생합니다.

같은 이유로, 오버로드를 위해 기본 형식과 `const` 다르게 수정되거나 `volatile` 수정되지 않은 형식의 함수 인수입니다.

그러나 함수 오버로드 메커니즘은 정규화된 참조와 `volatile` 기본 형식에 `const` 대한 참조를 구분할 수 있습니다. 다음과 같은 코드를 사용할 수 있습니다.

C++

```
// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
```

```

public:
    Over() { cout << "Over default constructor\n"; }
    Over( Over &o ) { cout << "Over&\n"; }
    Over( const Over &co ) { cout << "const Over&\n"; }
    Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

int main() {
    Over o1;                      // Calls default constructor.
    Over o2( o1 );                // Calls Over( Over& ).
    const Over o3;                 // Calls default constructor.
    Over o4( o3 );                // Calls Over( const Over& ).
    volatile Over o5;              // Calls default constructor.
    Over o6( o5 );                // Calls Over( volatile Over& ).  

}

```

출력

Output

```

Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&

```

또한 포인터 및 `const volatile` 개체는 오버로드를 위해 기본 형식에 대한 포인터와 다른 것으로 간주됩니다.

인수 일치 및 변환

컴파일러가 실제 인수를 함수 선언의 인수와 일치시키려고 할 때 정확히 일치하는 항목이 없을 경우 표준 또는 사용자 정의 변환을 통해 올바른 형식을 가져오도록 할 수 있습니다. 변환 애플리케이션에는 다음의 규칙이 적용됩니다.

- 둘 이상의 사용자 정의 변환을 포함하는 변환 시퀀스는 고려되지 않습니다.
- 중간 변환을 제거하여 단축할 수 있는 변환 시퀀스는 고려되지 않습니다.

변환의 결과 시퀀스(있는 경우)를 가장 일치하는 시퀀스라고 합니다. 표준 변환(표준 변환에 설명)을 사용하여 형식 `int` 개체를 형식 `unsigned long` 으로 변환하는 방법에는 여러 가지가 있습니다.

- 에서 `int` 다음으로 `long long unsigned long` 변환합니다.

- 에서 `int` .로 `unsigned long` 변환

첫 번째 시퀀스는 원하는 목표를 달성하지만 더 짧은 시퀀스가 존재하기 때문에 가장 일치하는 시퀀스는 아닙니다.

다음 표에서는 간단한 변환이라는 **변환 그룹**을 보여 줍니다. 사소한 변환은 컴파일러가 가장 적합한 일치 항목으로 선택하는 시퀀스에 제한적인 영향을 줍니다. 간단한 변환의 효과는 테이블 다음에 설명되어 있습니다.

사소한 변환

인수 형식	변환된 형식
<code>type-name</code>	<code>type-name&</code>
<code>type-name&</code>	<code>type-name</code>
<code>type-name[]</code>	<code>type-name*</code>
<code>type-name(argument-list)</code>	<code>(*type-name)(argument-list)</code>
<code>type-name</code>	<code>const type-name</code>
<code>type-name</code>	<code>volatile type-name</code>
<code>type-name*</code>	<code>const type-name*</code>
<code>type-name*</code>	<code>volatile type-name*</code>

변환이 시도되는 시퀀스는 다음과 같습니다.

1. 정확한 일치. 함수가 호출되는 형식과 함수 프로토타입에서 선언된 형식 간 정확한 일치는 항상 가장 좋은 일치입니다. trivial 변환 시퀀스는 정확히 일치하는 항목으로 분류됩니다. 그러나 이러한 변환을 수행하지 않는 시퀀스는 변환하는 시퀀스보다 더 나은 것으로 간주됩니다.
 - 포인터에서 포인터로(`type-name*` to `const`)로 이동합니다 `const type-name*`.
 - 포인터에서 포인터로(`type-name*` to `volatile`)로 이동합니다 `volatile type-name*`.
 - 참조에서 참조로(`type-name&` to `const`). `const type-name&`
 - 참조에서 참조로(`type-name&` to `volatile`). `volatile type&`
2. 승격을 통한 일치. 정수 승격만 포함하는 정확한 일치 항목으로 분류되지 않은 시퀀스, 변환 순서 `float double` 및 간단한 변환은 승격을 사용하여 일치 항목으로 분류

됩니다. 승격을 통한 일치는 정확한 일치만큼 양호하지는 않지만 표준 변환을 통한 일치에 비해 좋습니다.

3. 표준 변환을 통한 일치. 표준 변환과 trivial 변환만 포함 포함하는, 정확한 일치 또는 승격을 통한 일치로 분류되지 않은 시퀀스는 표준 변환을 통한 일치로 분류됩니다. 이 범주에는 다음 규칙이 적용됩니다.

- 포인터에서 파생 클래스로 변환하고 직접 또는 간접 기본 클래스에 대한 포인터로 변환하는 `void *` 것이 변환하는 `const void *` 것이 좋습니다.
- 파생 클래스에 대한 포인터에서 기본 클래스에 대한 포인터로 변환할 경우 기본 클래스가 직접 기본 클래스에 가까울수록 더 잘 일치합니다. 클래스 계층 구조가 다음 그림과 같이 있다고 가정합니다.



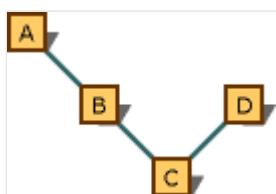
기본 변환을 보여 주는 그래프입니다.

`D*` 형식에서 `C*` 형식으로 변환하는 것이 `D*` 형식에서 `B*` 형식으로 변환하는 것보다 좋습니다. 마찬가지로 `D*` 형식에서 `B*` 형식으로 변환하는 것이 `D*` 형식에서 `A*` 형식으로 변환하는 것보다 좋습니다.

이 규칙은 참조 변환에 동일하게 적용됩니다. `D&` 형식에서 `C&` 형식으로 변환하는 것이 `D&` 형식에서 `B&` 형식 등으로 변환하는 것보다 좋습니다.

이 규칙은 멤버 포인터 변환에 동일하게 적용됩니다. `T D::*` 형식에서 `T C::*` 형식으로 변환하는 것이 `T D::*` 형식에서 `T B::*` 형식 등으로 변환하는 것보다 좋습니다. 여기서 `T`는 멤버 형식입니다.

앞의 규칙은 지정된 파생 경로에만 적용됩니다. 다음 그림에 표시된 그래프를 살펴보세요.



기본 변환을 보여 주는 다중 상속 그래프입니다.

`C*` 형식에서 `B*` 형식으로 변환하는 것이 `C*` 형식에서 `A*` 형식으로 변환하는 것보다 좋습니다. 이유는 이들이 동일한 경로에 있고 `B*` 가 더 가깝기 때문입니다. 그러나 형식에서

형식 `D*` 으로의 변환은 형식 `C* A*` 으로 변환하는 것이 바람직하지 않습니다. 변환이 다른 경로를 따르기 때문에 기본 설정이 없습니다.

1. 사용자 정의 변환을 통한 일치. 이 시퀀스는 정확한 일치, 승격을 사용한 일치 또는 표준 변환을 사용하는 일치 항목으로 분류할 수 없습니다. 사용자 정의 변환과 일치 항목으로 분류하려면 시퀀스에 사용자 정의 변환, 표준 변환 또는 사소한 변환만 포함되어야 합니다. 사용자 정의 변환과의 일치는 줄임표(`...`)를 사용한 일치보다 더 나은 일치로 간주되지만 표준 변환과의 일치만큼 일치하는 항목은 아닙니다.
2. 줄임표를 통한 일치. 선언에서 줄임표와 일치하는 모든 시퀀스는 줄임표를 통한 일치로 분류됩니다. 가장 약한 경기로 간주됩니다.

기본 승격 또는 변환이 존재하지 않는 경우 사용자 정의 변환이 적용됩니다. 이러한 변환은 일치하는 인수의 형식에 따라 선택됩니다. 다음 코드를 생각해 봅시다.

C++

```
// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}
```

클래스 `UDC` 에 사용할 수 있는 사용자 정의 변환은 형식 및 형식 `int long`입니다. 따라서 컴파일러가 일치하는 개체 형식을 위한 변환을 고려합니다(`UDC`). 변환 `int`이 존재하고 선택됩니다.

일치하는 인수를 처리하는 동안 인수 및 사용자 정의 변환 결과 모두에 표준 변환을 적용할 수 있습니다. 따라서 다음 코드가 작동합니다.

C++

```
void LogToFile( long l );
...
UDC udc;
LogFile( udc );
```

이 예제에서 컴파일러는 사용자 정의 변환 `operator long`을 호출하여 형식 `long`으로 변환 `udc` 합니다. 사용자 정의 형식 `long` 변환이 정의되지 않은 경우 컴파일러는 먼저 사용자 정의 `operator int` 변환을 사용하여 형식 `UDC`을 형식 `int`으로 변환합니다. 그런 다음 형식에서 형식 `int long`으로 표준 변환을 적용하여 선언의 인수와 일치합니다.

인수와 일치하기 위해 사용자 정의 변환이 필요한 경우 가장 일치하는 항목을 평가할 때 표준 변환이 사용되지 않습니다. 둘 이상의 후보 함수에 사용자 정의 변환이 필요한 경우에도 함수는 동일한 것으로 간주됩니다. 예시:

C++

```
// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
    UDC1( int ); // User-defined conversion from int.
};

class UDC2
{
public:
    UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
    Func( 1 );
}
```

두 버전 `Func` 모두 형식을 클래스 형식 `int` 인수로 변환하려면 사용자 정의 변환이 필요합니다. 가능한 변환은 다음과 같습니다.

- 형식에서 형식 `int UDC1`으로 변환(사용자 정의 변환).
- 형식에서 형식 `int long`으로 변환한 다음 형식 `UDC2`으로 변환합니다(2단계 변환).

두 번째 변환에는 표준 변환과 사용자 정의 변환이 모두 필요하지만 두 변환은 여전히 동일한 것으로 간주됩니다.

① 참고

사용자 정의 변환은 초기화에 의한 생성 또는 변환에 의한 변환으로 간주됩니다. 컴파일러는 최상의 일치를 결정할 때 두 메서드를 동일하게 간주합니다.

인수 일치 및 `this` 포인터

클래스 멤버 함수는 로 선언되었는지 여부에 따라 다르게 처리됩니다 `static`. `static` 함수에는 포인터를 제공하는 `this` 암시적 인수가 없으므로 일반 멤버 함수보다 인수가 적은 것으로 간주됩니다. 그렇지 않으면 동일하게 선언됩니다.

함수가 호출되는 개체 형식과 일치하기 위해 암시적 `this` 포인터가 필요하지 않은 `static` 멤버 함수입니다. 또는 오버로드된 연산자의 경우 연산자가 적용되는 개체와 일치하도록 첫 번째 인수가 필요합니다. 오버로드된 연산자에 대한 자세한 내용은 오버로드된 연산자를 참조 [하세요](#).

오버로드된 함수의 다른 인수와 달리 컴파일러는 임시 개체를 도입하지 않으며 포인터 인수와 일치 `this` 하려고 할 때 변환을 시도하지 않습니다.

멤버 선택 연산자를 `->` 사용하여 클래스 `class_name` `this` 의 멤버 함수에 액세스하는 경우 포인터 인수에는 형식이 `class_name * const` 있습니다. 멤버로 선언되거나 선언된 `const` 경우 형식은 각각 및 `volatile class_name * const` 그 형식입니다 `const class_name * const volatile`

명시적 `.` 주소 연산자가 개체 이름 앞에 추가된다는 점을 제외하면 `&` 멤버 선택 연산자는 동일하게 작동합니다. 다음 예제에서는 그 작동 방식을 보여 줍니다.

C++

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

`->*`의 왼쪽 피연산자와 `.*`(멤버에 대한 포인터) 연산자는 인수 일치와 관련하여 `.` 및 `->` (멤버 선택) 연산자와 동일한 방식으로 처리됩니다.

멤버 함수의 참조 한정자

참조 한정자를 사용하면 가리키는 개체가 rvalue인지 lvalue인지 여부에 따라 멤버 함수를 오버로드할 `this` 수 있습니다. 데이터에 대한 포인터 액세스를 제공하지 않도록 선택하는 시나리오에서 불필요한 복사 작업을 방지하려면 이 기능을 사용합니다. 예를 들어 클래스 `C` 는 생성자의 일부 데이터를 초기화하고 멤버 함수 `get_data()` 에서 해당 데이터의 복사본을 반환한다고 가정합니다. 형식 `C` 의 개체가 소멸될 rvalue인 경우 컴파일러는 오버로드를 선택하여 `get_data() &&` 데이터를 복사하는 대신 이동합니다.

C++

```
#include <iostream>
#include <vector>

using namespace std;

class C
{
public:
    C() {/*expensive initialization*/}
    vector<unsigned> get_data() &
    {
        cout << "lvalue\n";
        return _data;
    }
    vector<unsigned> get_data() &&
    {
        cout << "rvalue\n";
        return std::move(_data);
    }

private:
    vector<unsigned> _data;
};

int main()
{
    C c;
    auto v = c.get_data(); // get a copy. prints "lvalue".
    auto v2 = C().get_data(); // get the original. prints "rvalue"
    return 0;
}
```

오버로드에 대한 제한 사항

여러 제한은 사용할 수 있는 오버로드된 함수 집합을 관리합니다.

- 오버로드된 함수 집합의 임의의 두 함수에는 서로 다른 인수 목록이 있어야 합니다.

- 반환 형식만을 기준으로 동일한 형식의 인수 목록이 있는 함수를 오버로드하는 것은 오류입니다.

Microsoft 전용

특히 지정된 메모리 모델 한정자에 따라 반환 형식에 따라 오버로드 `operator new` 할 수 있습니다.

Microsoft 전용 종료

- 멤버 함수는 하나만 오버로드할 수 없습니다. 하나는 다른 함수이고 다른 함수는 `static` 오버로드되지 않기 `static` 때문입니다.
- `typedef` 선언은 새 형식을 정의하지 않습니다. 기존 형식의 동의어를 소개합니다. 오버로드 메커니즘에는 영향을 주지 않습니다. 다음 코드를 생각해 봅시다.

C++

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

위의 두 함수에는 동일한 인수 목록이 있습니다. `PSTR` 는 형식 `char *`의 동의어입니다. 멤버 범위에서 이 코드를 사용하면 오류가 발생합니다.

- 열거 형식은 고유한 형식이며 오버로드된 함수 사이를 구분하는 데 사용할 수 있습니다.
- "array of" 및 "pointer to" 형식은 오버로드된 함수를 구분하기 위해 동일하지만 1차원 배열에 대해서만 동일한 것으로 간주됩니다. 이러한 오버로드된 함수는 충돌하고 오류 메시지를 생성합니다.

C++

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

더 높은 차원 배열의 경우 두 번째 및 이후 차원은 형식의 일부로 간주됩니다. 오버로드된 함수를 구분하는 데 사용됩니다.

C++

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
```

```
void Print( char szToPrint[][9][42] );
```

오버로드, 재정의 및 숨기기

동일한 범위에서 동일한 이름의 두 함수 선언은 동일한 함수 또는 두 개의 개별 오버로드된 함수를 참조할 수 있습니다. 선언의 인수 목록에 동일한 형식의 인수가 포함되어 있으면(이전 단원 참조) 함수 선언은 같은 함수를 참조하고, 그렇지 않으면 오버로드를 사용하여 선택된 서로 다른 두 함수를 참조합니다.

클래스 범위는 엄격하게 관찰됩니다. 기본 클래스에서 선언된 함수는 파생 클래스에서 선언된 함수와 동일한 범위에 있지 않습니다. 파생 클래스의 함수가 기본 클래스의 함수와 동일한 이름으로 `virtual` 선언되면 파생 클래스 함수는 기본 클래스 함수를 재정의합니다. 자세한 내용은 Virtual Functions를 참조 [하세요](#).

기본 클래스 함수가 선언 `virtual` 되지 않은 경우 파생 클래스 함수는 이를 숨깁니다. 재정의와 숨기기는 모두 오버로드와 구별됩니다.

블록 범위는 엄격하게 관찰됩니다. 파일 범위에서 선언된 함수는 로컬로 선언된 함수와 동일한 범위에 있지 않습니다. 로컬로 선언된 함수의 이름이 파일 범위에서 선언된 함수의 이름과 같을 경우 로컬로 선언된 함수는 오버로드를 유발하는 대신 파일 범위의 함수를 숨깁니다. 예시:

C++

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally declared func : " << sz << endl;
}

int main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );    // C2664 Error. func( int ) is hidden.
```

```
    func( "s" );
}
```

위의 코드에서는 `func` 함수의 두 정의를 보여 줍니다. 형식 `char *` 의 인수를 사용하는 정의는 문 때문에 로컬 `main` 입니다 `extern`. 따라서 형식 `int` 의 인수를 사용하는 정의는 숨겨지고 첫 번째 호출 `func` 은 오류입니다.

오버로드된 멤버 함수의 경우 함수의 버전마다 서로 다른 액세스 권한을 부여할 수 있습니다. 여전히 바깥쪽 클래스의 범위에 있는 것으로 간주되므로 오버로드된 함수입니다. 멤버 함수 `Deposit` 가 오버로드되는 다음 코드를 살펴보겠습니다. 한 버전은 `public`이고 다른 버전은 `private`입니다.

이 샘플의 목적은 입금을 하려면 올바른 암호가 필요한 `Account` 클래스를 제공하는 것입니다. 오버로드를 사용하여 수행됩니다.

in `Account::Deposit` 호출 `Deposit` 은 프라이빗 멤버 함수를 호출합니다. 이 호출은 멤버 함수이며 클래스의 프라이빗 멤버에 액세스할 수 있기 때문에 `Account::Deposit` 정확합니다.

C++

```
// declaration_matching2.cpp
class Account
{
public:
    Account()
    {
    }
    double Deposit( double dAmount, char *szPassword );

private:
    double Deposit( double dAmount )
    {
        return 0.0;
    }
    int Validate( char *szPassword )
    {
        return 0;
    }

};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );
```

```
// Deposit $57.22 and supply a password. OK: calls a
// public function.
pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
    if ( Validate( szPassword ) )
        return Deposit( dAmount );
    else
        return 0.0;
}
```

참고 항목

[함수\(C++\)](#)

명시적으로 기본 설정 및 삭제된 함수

아티클 • 2023. 11. 16.

C++ 11에서 기본 설정 및 삭제된 함수를 사용하면 특수 멤버 함수가 자동으로 생성되는지 여부를 명시적으로 제어할 수 있습니다. 또한 삭제된 함수는 모든 형식의 함수(특수 멤버 함수 및 일반 멤버 함수 및 비회원 함수)에 대한 인수에서 문제가 있는 형식 승격이 발생하지 않도록 하는 간단한 언어를 제공합니다. 그렇지 않으면 원치 않는 함수 호출이 발생합니다.

명시적으로 기본 설정 및 삭제된 함수의 이점

C++에서 컴파일러는 자체 생성자를 선언하지 않는 경우 형식에 대한 기본 생성자, 복사 생성자, 복사 할당 연산자 및 소멸자를 자동으로 생성합니다. 이러한 함수는 특수 멤버 함수라고 하며 C++의 간단한 사용자 정의 형식이 C에서 구조체처럼 동작하도록 합니다. 즉, 추가 코딩 작업 없이 생성, 복사 및 삭제할 수 있습니다. C++11은 언어에 이동 의미 체계를 가져오고 이동 생성자와 이동 할당 연산자를 컴파일러가 자동으로 생성할 수 있는 특수 멤버 함수 목록에 추가합니다.

따라서 단순 형식에는 편리하지만 복합 형식은 종종 하나 이상의 특수 멤버 함수 자체를 정의하므로 다른 특수 멤버 함수가 자동으로 생성되지 않도록 할 수 있습니다. 실제로는 다음과 같습니다.

- 생성자가 명시적으로 선언된 경우 기본 생성자가 자동으로 생성되지 않습니다.
- 가상 소멸자가 명시적으로 선언된 경우 기본 소멸자가 자동으로 생성되지 않습니다.
- 이동 생성자 혹은 이동 할당 연산자가 명시적으로 선언된 경우 다음과 같습니다.
 - 복사 생성자가 자동으로 생성되지 않습니다.
 - 복사 할당 연산자가 자동으로 생성되지 않습니다.
- 복사 생성자, 복사 할당 연산자, 이동 생성자, 이동 할당 연산자 또는 소멸자가 명시적으로 선언된 경우 다음과 같습니다.
 - 이동 생성자가 자동으로 생성되지 않습니다.
 - 이동 할당 연산자가 자동으로 생성되지 않습니다.

① 참고

또한 C++ 11 표준은 다음 추가 규칙을 지정합니다.

- 복사 생성자나 소멸자가 명시적으로 선언된 경우 복사 할당 연산자가 자동으로 생성되지 않습니다.
- 복사 할당 연산자나 소멸자가 명시적으로 선언된 경우 복사 생성자가 자동으로 생성되지 않습니다.

두 경우 모두 Visual Studio는 계속해서 필요한 함수를 암시적으로 자동으로 생성하며 기본적으로 경고를 내보내지 않습니다. Visual Studio 2022 버전 17.7 **부터 C5267**을 사용하도록 설정하여 경고를 내보낼 수 있습니다.

이러한 규칙으로 인해 개체 계층 구조에 누수가 발생할 수도 있습니다. 예를 들어 어떤 이유로든 기본 클래스에 파생 클래스에서 호출할 수 있는 기본 생성자(즉, `public` 매개 변수를 사용하지 않는 생성자) `protected` 가 없는 경우 해당 클래스에서 파생된 클래스는 자체 기본 생성자를 자동으로 생성할 수 없습니다.

이러한 규칙은 직접 전달되어야 하는 사용자 정의 형식 및 일반적인 C++ 관용구의 구현을 복잡하게 만들 수 있습니다. 예를 들어 복사 생성자 및 복사 할당 연산자를 비공개로 선언하고 정의하지 않음으로써 사용자 정의 형식을 지정할 수 없게 만들 수 있습니다.

C++

```
struct noncopyable
{
    noncopyable() {};

private:
    noncopyable(const noncopyable&);
    noncopyable& operator=(const noncopyable&);
};
```

C++11 이전에는 이 코드 조각이 범위가 지정 불가능한 형식의 idiomatic 형식이었습니다. 그러나 다음과 같은 몇 가지 문제가 있습니다.

- 복사 생성자를 숨기려면 복사 생성자를 비공개로 선언해야 하지만, 복사 생성자가 전혀 선언되어 있으므로 기본 생성자의 자동 생성이 방지됩니다. 기본 생성자가 아무 작업도 수행하지 않는 경우에도 원하면 명시적으로 정의해야 합니다.
- 명시적으로 정의된 기본 생성자가 아무 것도 수행하지 않더라도 컴파일러는 이 생성자를 사소한 것으로 간주합니다. 기본 생성자를 자동으로 생성하는 것보다 덜 효율적이며 `noncopyable` 이 진정한 POD 형식이 되지 않습니다.
- 복사 생성자 및 복사 할당 연산자는 외부 코드로부터 숨겨지지만 멤버 함수와 `noncopyable` 의 friend에서는 보고 호출할 수 있습니다. 선언되었지만 정의되지 않은 경우 호출하면 링커 오류가 발생합니다.

- 일반적으로 허용되는 관용구이지만 특수 멤버 함수의 자동 생성에 대한 모든 규칙을 이해하지 않는 한 의도는 명확하지 않습니다.

C++11에서는 보다 간단한 방식으로 범위를 지정할 수 없는 관용구를 구현할 수 있습니다.

C++

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

C++11 이전 관용구 문제를 해결하는 방법을 확인합니다.

- 기본 생성자의 생성은 복사 생성자를 선언하여 예방되지만 명시적으로 기본값으로 지정하여 되돌릴 수 있습니다.
- 명시적으로 기본값으로 지정된 특수 멤버 함수는 여전히 사소한 것으로 간주되므로 성능 저하 `noncopyable` 가 없으면 실제 POD 형식이 되는 것을 방지하지 않습니다.
- 복사 생성자 및 복사 할당 연산자는 `public`이지만 삭제됩니다. 삭제된 함수를 정의하거나 호출하는 것은 컴파일 시간 오류입니다.
- 이 의도는 `=default` 및 `=delete` 를 이해하는 사용자에게 명확하게 전달됩니다. 특수 멤버 함수의 자동 생성 규칙을 이해할 필요가 없습니다.

이동 불가능하거나 동적으로만 할당할 수 있거나 동적으로 할당할 수 없는 사용자 정의 형식을 만들기 위한 유사한 관용구가 있습니다. 이러한 각 관용구에는 유사한 문제가 발생하는 C++11 이전의 구현이 있으며, C++11에서 기본 설정 및 삭제된 특수 멤버 함수 측면에서 구현하여 유사하게 해결할 수 있습니다.

명시적으로 기본 설정된 함수

특수 멤버 함수를 기본값으로 지정하여 특수 멤버 함수가 기본 구현을 사용한다고 명시적으로 명시하거나, 비공개 액세스 한정자를 사용하여 특수 멤버 함수를 정의하거나, 다른 상황에서 자동 생성이 금지된 특수 멤버 함수를 복원할 수 있습니다.

다음 예제와 같이 선언하여 특수 멤버 함수를 기본 설정합니다.

C++

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);

    inline widget& widget::operator=(const widget&) =default;
```

인라인 가능하면 클래스 본문 외부에서 특수 멤버 함수를 기본값으로 사용할 수 있습니다.

trivial 특수 멤버 함수의 성능 이점으로 인해 기본 동작을 원하는 경우 비어 있는 함수 본문보다 자동으로 생성된 특수 멤버 함수를 사용하는 것이 좋습니다. 이렇게 하려면 특수 멤버 함수를 명시적으로 기본 설정하거나 특수 멤버 함수를 선언하지 않고 자동으로 생성될 수 없도록 하는 다른 특수 멤버 함수도 선언하지 않을 수 있습니다.

삭제된 함수

특수 멤버 함수와 일반 멤버 함수 및 비회원 함수를 삭제하여 정의되거나 호출되지 않도록 할 수 있습니다. 특수 멤버 함수를 삭제하면 컴파일러가 원하지 않는 특수 멤버 함수를 생성하지 못하도록 방지하는 클린 방법을 제공합니다. 함수는 선언된 대로 삭제해야 합니다. 나중에 함수를 선언한 다음 나중에 기본값으로 설정할 수 있는 방식으로 삭제할 수 없습니다.

C++

```
struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};
```

일반 멤버 함수 또는 비회원 함수를 삭제하면 문제가 있는 형식 승격으로 인해 의도하지 않은 함수가 호출되지 않습니다. 이 기능은 삭제된 함수가 오버로드 확인에 계속 참여하고 형식이 승격된 후 호출될 수 있는 함수보다 더 우수한 일치를 제공하기 때문에 작동합니다. 함수 호출은 보다 구체적이지만 삭제된 함수가 되고 컴파일러 오류를 발생시킵니다.

C++

```
// deleted overload prevents call through type promotion of float to double
// from succeeding.
```

```
void call_with_true_double_only(float) =delete;
void call_with_true_double_only(double param) { return; }
```

앞의 샘플에서 인수를 사용하여 호출 `call_with_true_double_only` 하면 컴파일러 오류가 발생하지만 인수를 사용하여 `int` 호출 `call_with_true_double_only` 하는 것은 그렇지 않습니다 `int`. 이 경우 의도한 것이 아니더라도 인수가 함수 버전으로 `int double` 승격되고 함수 버전을 성공적으로 호출 `double` 합니다. `float` 이중이 아닌 인수를 사용하여 이 함수를 호출하면 컴파일러 오류가 발생하도록 하려면 삭제된 함수의 템플릿 버전을 선언할 수 있습니다.

C++

```
template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type
promotion of any T to double from succeeding.

void call_with_true_double_only(double param) { return; } // also define for
const double, double&, etc. as needed.
```

함수에 대한 인수 종속 이름(Koenig) 조회

아티클 • 2023. 10. 12.

컴파일러에서는 인수 종속 이름 조회를 사용하여 정규화되지 않은 함수 호출의 정의를 찾을 수 있습니다. 인수 종속 이름 조회를 Koenig 조회라고도 합니다. 함수 호출에 있는 모든 인수의 형식은 네임스페이스, 클래스, 구조체, 공용 구조체 또는 템플릿의 계층 구조 내에서 정의됩니다. 정규 [화되지 않은 후위](#) 함수 호출을 지정하면 컴파일러는 각 인수 형식과 연결된 계층 구조에서 함수 정의를 검색합니다.

예시

이 샘플에서 컴파일러는 `f()` 함수가 `x` 인수를 사용하는 것을 확인합니다. `x` 인수는 `A::x` 형식이며, 이 형식은 `A` 네임스페이스에서 정의됩니다. 컴파일러는 `A` 네임스페이스를 검색하고 `f()` 형식의 인수를 사용하는 `A::x` 함수에 대한 정의를 찾습니다.

C++

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
    struct X
    {
    };
    void f(const X&)
    {
    }
}
int main()
{
    // The compiler finds A::f() in namespace A, which is where
    // the type of argument x is defined. The type of x is A::X.
    A::X x;
    f(x);
}
```

기본 인수

아티클 • 2023. 10. 12.

많은 경우에 함수에는 기본값이면 충분할 정도로 가끔 사용되는 인수가 있습니다. 이러한 경우 기본 인수 기능을 사용하면 지정된 호출에서 의미가 있는 인수만 함수에 지정할 수 있습니다. 이 개념을 설명하기 위해 함수 오버로드에 [제시된 예제를 고려해 보세요](#).

C++

```
// Prototype three print functions.  
int print( char *s ); // Print a string.  
int print( double dvalue ); // Print a double.  
int print( double dvalue, int prec ); // Print a double with a  
// given precision.
```

많은 애플리케이션에서 `prec`에 대한 적절한 기본값을 제공하여 두 함수의 필요성을 없앨 수 있습니다.

C++

```
// Prototype two print functions.  
int print( char *s ); // Print a string.  
int print( double dvalue, int prec=2 ); // Print a double with a  
// given precision.
```

함수의 `print` 구현은 형식 `double`에 대해 이러한 함수가 하나만 존재한다는 사실을 반영하도록 약간 변경됩니다.

C++

```
// default_arguments.cpp  
// compile with: /EHsc /c  
  
// Print a double in specified precision.  
// Positive numbers for precision indicate how many digits  
// precision after the decimal point to show. Negative  
// numbers for precision indicate where to round the number  
// to the left of the decimal point.  
  
#include <iostream>  
#include <math.h>  
using namespace std;  
  
int print( double dvalue, int prec ) {  
    // Use table-lookup for rounding/truncation.  
    static const double rgPow10[] = {  
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
```

```

    10E1,  10E2,  10E3,  10E4, 10E5,  10E6
};

const int iPowZero = 6;
// If precision out of range, just print the number.
if( prec >= -6 && prec <= 7 )
    // Scale, truncate, then rescale.
    dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
        rgPow10[iPowZero - prec];
cout << dvalue << endl;
return cout.good();
}

```

새 `print` 함수를 호출하려면 다음과 같은 코드를 사용합니다.

C++

```

print( d );      // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
// results.

```

기본 인수를 사용하는 경우 다음 사항에 유의하십시오.

- 기본 인수는 후행 인수가 생략되는 함수 호출에서만 사용되며 마지막 인수여야 합니다. 따라서 다음 코드는 올바르지 않습니다.

C++

```
int print( double dvalue = 0.0, int prec );
```

- 기본 인수는 다시 지정한 정의가 원래 정의와 동일한 경우에도 이후 선언에서 다시 정의할 수 없습니다. 따라서 다음 코드는 오류를 생성합니다.

C++

```

// Prototype for print function.
int print( double dvalue, int prec = 2 );

...
// Definition for print function.
int print( double dvalue, int prec = 2 )
{
    ...
}

```

이 코드의 문제는 정의에 있는 함수 선언이 `prec`의 기본 인수를 다시 정의하는 것입니다.

- 이후 선언에서 기본 인수를 더 추가할 수 있습니다.
- 함수에 대한 포인터에 기본 인수를 제공할 수 있습니다. 예시:

C++

```
int (*pShowIntVal)( int i = 0 );
```

인라인 함수(C++)

아티클 • 2024. 07. 12.

`inline` 키워드는 컴파일러가 해당 함수에 대한 각 호출 대신 함수 정의 내의 코드를 대체하도록 제안합니다.

이론적으로 인라인 함수를 사용하면 함수 호출과 연관된 오버헤드가 제거되어 프로그램 속도가 더 빨라질 수 있습니다. 함수를 호출하려면 스택에 반환 주소를 푸시하고, 인수를 스택에 푸시하고, 함수 본문으로 이동한 다음, 함수가 완료되면 반환 명령을 실행해야 합니다. 이 프로세스는 함수를 인라인 처리하여 제거됩니다. 또한 컴파일러는 인라인으로 확장된 함수와 그렇지 않은 함수를 최적화하는 다른 기회도 있습니다. 인라인 함수의 단점은 프로그램의 전체 크기가 증가할 수 있다는 것입니다.

인라인 코드 대체는 컴파일러의 재량에 따라 수행됩니다. 예를 들어 컴파일러는 함수 주소를 가져온 경우나 함수가 너무 크다고 판단하는 경우 함수를 인라인 처리하지 않습니다.

클래스 선언의 본문에 정의된 함수는 암시적으로 인라인 함수입니다.

예시

다음 클래스 선언에서 `Account` 생성자는 클래스 선언의 본문에 정의되어 있으므로 인라인 함수입니다. 멤버 함수 `GetBalance`, `Deposit` 및 `Withdraw`는 정의에서 `inline`으로 지정됩니다. `inline` 키워드는 클래스 선언의 함수 선언에서 선택 사항입니다.

```
C++  
  
// account.h  
class Account  
{  
public:  
    Account(double initial_balance)  
    {  
        balance = initial_balance;  
    }  
  
    double GetBalance() const;  
    double Deposit(double amount);  
    double Withdraw(double amount);  
  
private:  
    double balance;  
};  
  
inline double Account::GetBalance() const
```

```
{  
    return balance;  
}  
  
inline double Account::Deposit(double amount)  
{  
    balance += amount;  
    return balance;  
}  
  
inline double Account::Withdraw(double amount)  
{  
    balance -= amount;  
    return balance;  
}
```

① 참고

클래스 선언에서는 함수가 `inline` 키워드 없이 선언되었습니다. `inline` 키워드는 클래스 선언에서 지정할 수 있으며 결과는 동일합니다.

주어진 인라인 멤버 함수는 모든 컴파일 단위에서 동일한 방식으로 선언되어야 합니다. 인라인 함수의 정의는 정확히 하나만 있어야 합니다.

해당 함수에 대한 정의에 `inline` 지정자가 포함되지 않는 한 클래스 멤버 함수는 기본적으로 외부 링크로 설정됩니다. 앞의 예에서는 `inline` 지정자를 사용하여 이러한 함수를 명시적으로 선언할 필요가 없음을 보여줍니다. 함수 정의에서 `inline`을 사용하면 컴파일러에 함수를 인라인 함수로 처리하라고 제안합니다. 그러나 함수를 호출한 후에는 해당 함수를 `inline`으로 다시 지정할 수 없습니다.

inline, __inline 및 __forceinline

`inline` 및 `__inline` 지정자는 함수가 호출되는 각 위치에 함수 본문의 복사본을 삽입하도록 컴파일러에 제안합니다.

삽입(인라인 확장 또는 인라인 처리라고 함)은 컴파일러의 비용/이익 분석에서 가치가 있다고 보이는 경우에만 일어납니다. 인라인 확장으로 코드 크기가 커질 수 있지만 함수 호출 오버헤드는 최소화할 수 있습니다.

`__forceinline` 키워드는 비용/이익 분석을 재정의하고 대신 프로그래머의 판단에 의존합니다. `__forceinline`을 사용할 때는 주의해야 합니다. `__forceinline`을 무분별하게 사용하면 코드가 더 커져 성능이 조금밖에 향상되지 않거나, 경우에 따라 더 큰 실행 파일의 페이지 증가와 같은 이유로 성능이 저하될 수도 있습니다.

컴파일러는 인라인 확장 옵션과 키워드를 제안으로 처리합니다. 함수가 인라인 처리된다 는 보장은 없습니다. `_forceinline` 키워드를 사용해도 컴파일러에 특정 함수를 강제로 인라인 처리하게 할 수 없습니다. `/clr`로 컴파일할 때 컴파일러는 함수에 적용된 보안 특성이 있으면 함수를 인라인 처리하지 않습니다.

이전 버전과의 호환성을 위해 `_inline` 및 `_forceinline`은 각각 `_inline` 및 `_forceinline`의 동의어입니다. 단 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

`inline` 키워드는 인라인 확장이 선호된다는 것을 컴파일러에 알려줍니다. 하지만 컴파일러는 이를 무시할 수 있습니다. 이 동작이 발생할 수 있는 두 가지 경우는 다음과 같습니다.

- 재귀 함수.
- 변환 단위의 다른 위치에서 포인터를 통해 참조되는 함수.

이러한 이유는 다른 이유와 마찬가지로 컴파일러에 의해 결정된 대로 인라인 처리에 방해가 될 수 있습니다. 함수가 인라인 처리되게 하는 데 `inline` 지정자에 의존하지 마세요.

컴파일러는 헤더 파일에 정의된 인라인 함수를 확장하는 대신 둘 이상의 변환 단위에서 호출 가능한 함수로 만들 수 있습니다. 컴파일러는 링커에 대해 생성된 함수를 표시하여 ODR(One-definition-rule) 위반을 방지합니다.

일반 함수와 마찬가지로 인라인 함수에서 인수 평가에 정의된 순서는 없습니다. 실제로 인수 평가 순서가 일반 함수 호출 프로토콜을 사용하여 전달될 때와 다를 수 있습니다.

인라인 함수 확장이 실제로 발생하는지 여부에 영향을 주려면 `/Ob` 컴파일러 최적화 옵션을 사용합니다.

`/LTCG`는 소스 코드에서의 요청 여부에 관계없이 모듈 간 인라인 처리를 수행합니다.

예 1

C++

```
// inline_keyword1.cpp
// compile with: /c
inline int max(int a, int b)
{
    return a < b ? b : a;
}
```

클래스의 멤버 함수는 `inline` 키워드를 사용하거나 클래스 정의 내 함수 정의를 배치하여 인라인으로 선언할 수 있습니다.

예제 2

```
C++  
  
// inline_keyword2.cpp  
// compile with: /EHsc /c  
#include <iostream>  
  
class MyClass  
{  
public:  
    void print() { std::cout << i; } // Implicitly inline  
  
private:  
    int i;  
};
```

Microsoft 전용

`_inline` 키워드는 `inline`과 동일합니다.

`_forceinline` 경우에도 다음과 같은 경우 컴파일러는 함수를 인라인 처리할 수 없습니다.

- 함수 또는 해당 호출자가 `/Ob0`(디버그 빌드에 대한 기본 옵션)으로 컴파일됩니다.
- 함수 및 호출자가 다양한 형식의 예외 처리(한 경우 C++ 예외 처리, 다른 경우 구조적 예외 처리)를 사용합니다.
- 함수에 가변 인수 목록이 있습니다.
- `/Ox`, `/O1` 또는 `/O2`로 컴파일되지 않은 한 함수는 인라인 어셈블리를 사용합니다.
- 함수는 재귀적이며 `#pragma inline_recursion(on)`이 설정되지 않습니다. pragma를 사용하면 재귀 함수가 기본 깊이 16번 호출로 인라인 처리됩니다. 인라인 깊이를 줄 이려면 `inline_depth` pragma를 사용합니다.
- 가상 함수이며 실제로 호출됩니다. 가상 함수에 대한 직접 호출은 인라인 처리할 수 있습니다.
- 프로그램에서 함수의 주소를 사용하고 함수에 대한 포인터를 통해 호출합니다. 주소가 사용된 함수에 대한 직접 호출은 인라인 처리할 수 있습니다.
- 또한 함수는 `naked __declspec` 한정자를 사용하여 표시됩니다.

컴파일러가 `_forceinline`으로 선언된 함수를 인라인 처리할 수 없으면 다음 경우를 제외하고 수준 1 경고가 생성됩니다.

- 함수가 `/Od` 또는 `/Ob0`을 사용하여 컴파일됩니다. 이러한 경우에는 인라인 처리가 필요하지 않습니다.
- 함수는 포함된 라이브러리나 다른 변환 단위에서 외부적으로 정의되거나, 가상 호출 대상 또는 간접 호출 대상입니다. 컴파일러는 현재 변환 단위에서 찾을 수 없는,

인라인이 아닌 코드를 식별할 수 없습니다.

재귀 함수는 `inline_depth` pragma에서 지정한 깊이까지 최대 16개의 호출까지 인라인 코드로 대체될 수 있습니다. 해당 깊이 이후 재귀 함수 호출은 함수의 인스턴스 호출로 처리됩니다. 인라인 추론에서 재귀 함수를 검사하는 깊이는 16을 초과할 수 없습니다.

`inline_recursion` pragma는 현재 확장 중인 함수의 인라인 확장을 제어합니다. 관련 정보는 [인라인 함수 확장\(/Ob\)](#) 컴파일러 옵션을 참조하세요.

Microsoft 전용 종료

`inline` 지정자 사용에 대한 자세한 내용은 다음을 참조하세요.

- [인라인 클래스 멤버 함수](#)
- `dllexport` 및 `dllimport`로 인라인 C++ 함수 정의

인라인 함수 사용 시기

인라인 함수는 데이터 멤버에 대한 액세스를 제공하는 함수와 같은 작은 함수에 가장 적합합니다. 짧은 함수는 함수 호출의 오버헤드에 민감합니다. 긴 함수는 호출 및 반환 시퀀스에서 비례적으로 적은 시간이 들고 인라인 처리로 인한 이점이 크지 않습니다.

`Point` 클래스는 다음과 같이 정의할 수 있습니다.

C++

```
// when_to_use_inline_functions.cpp
// compile with: /c
class Point
{
public:
    // Define "accessor" functions
    // as reference types.
    unsigned& x();
    unsigned& y();

private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}

inline unsigned& Point::y()
{
```

```
    return _y;  
}
```

좌표 조작이 이러한 클래스의 클라이언트에서 상대적으로 일반적인 작업이라고 가정하면 두 접근자 함수(앞의 예에서 `x` 및 `y`)를 `inline`으로 지정하여 일반적으로 다음에서 오버헤드를 줄입니다.

- 함수 호출(개체의 주소를 스택에 전달 및 배치하는 매개 변수 포함)
- 호출자의 스택 프레임의 보존
- 새 스택 프레임 설정
- 반환 값 전달
- 이전 스택 프레임 복원
- Return

인라인 함수와 매크로 비교

매크로는 `inline` 함수와 몇 가지 공통점이 있습니다. 하지만 두 가지 중요한 차이점이 있습니다. 다음 예제를 참조하세요.

C++

```
#include <iostream>  
  
#define mult1(a, b) a * b  
#define mult2(a, b) (a) * (b)  
#define mult3(a, b) ((a) * (b))  
  
inline int multiply(int a, int b)  
{  
    return a * b;  
}  
  
int main()  
{  
    std::cout << (48 / mult1(2 + 2, 3 + 3)) << std::endl; // outputs 33  
    std::cout << (48 / mult2(2 + 2, 3 + 3)) << std::endl; // outputs 72  
    std::cout << (48 / mult3(2 + 2, 3 + 3)) << std::endl; // outputs 2  
    std::cout << (48 / multiply(2 + 2, 3 + 3)) << std::endl; // outputs 2  
  
    std::cout << mult3(2, 2.2) << std::endl; // no warning  
    std::cout << multiply(2, 2.2); // Warning C4244 'argument': conversion  
from 'double' to 'int', possible loss of data  
}
```

Output

```
33  
72  
2  
2  
4.4  
4
```

다음은 매크로와 인라인 함수 간의 몇 가지 차이점입니다.

- 매크로는 항상 인라인으로 확장됩니다. 그러나 인라인 함수는 컴파일러에 의해 최적이라고 판단될 때만 인라인 처리가 이루어집니다.
- 매크로는 예기치 않은 동작을 발생시킬 수 있고, 이로 인해 미묘한 버그가 발생할 수 있습니다. 예를 들어 식 `mult1(2 + 2, 3 + 3)` 이 11로 계산되는 `2 + 2 * 3 + 3` 으로 확장되지만, 예상 결과는 24입니다. 유효해 보이는 수정 방법은 함수 매크로의 두 인수 주위에 괄호를 추가하여 `#define mult2(a, b) (a) * (b)` 의 결과를 가져오는 것입니다. 이렇게 하면 당면 문제는 해결되지만, 더 큰 식의 일부인 경우에는 여전히 놀라운 동작이 발생할 수 있습니다. 이는 앞의 예에서 설명했으며, 매크로를 `#define mult3(a, b) ((a) * (b))` 로 정의하여 문제가 해결될 수 있습니다.
- 인라인 함수는 컴파일러의 의미 체계 처리의 대상이 되지만, 전처리기는 이와 동일한 이점 없이 매크로를 확장합니다. 매크로는 형식이 안전하지 않지만 함수는 형식이 안전합니다.
- 인라인 함수에 인수로 전달된 식은 한 번 계산됩니다. 매크로에 인수로 전달된 식은 경우에 따라 여러 번 계산할 수 있습니다. 다음 예를 살펴보세요.

C++

```
#include <iostream>

#define sqr(a) ((a) * (a))

int increment(int& number)
{
    return number++;
}

inline int square(int a)
{
    return a * a;
}

int main()
{
    int c = 5;
    std::cout << sqr(increment(c)) << std::endl; // outputs 30
    std::cout << c << std::endl; // outputs 7

    c = 5;
```

```
    std::cout << square(increment(c)) << std::endl; // outputs 25
    std::cout << c; // outputs 6
}
```

Output

```
30
7
25
6
```

이 예에서는 식 `sqr(increment(c))` 가 `((increment(c)) * (increment(c)))`로 확장할 때 함수 `increment`가 두 번 호출됩니다. 이로 인해 `increment`의 두 번째 호출이 6을 반환하고, 따라서 식은 30으로 계산됩니다. 부작용이 포함된 식은 매크로에 사용될 때 결과에 영향을 줄 수 있습니다. 완전히 확장된 매크로를 검사하여 동작이 의도된 것인지 확인하세요. 대신 인라인 함수 `square`가 사용된 경우 `increment` 함수는 한 번만 호출되고 25라는 올바른 결과를 얻게 됩니다.

참고 항목

[noinline](#)

[auto_inline](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

연산자 오버로드

아티클 • 2024. 11. 21.

operator 키워드는 이 클래스 인스턴스에 적용될 때의 *operator-symbol* 의미를 지정하는 함수를 선언합니다. 이 키워드는 연산자에게 둘 이상의 의미를 제공 즉, 오버로드합니다. 컴파일러는 피연산자의 형식을 검사하여 연산자의 여러 가지 의미 간을 구분합니다.

구문

type **operator** *operator-symbol* (*parameter-list*)

설명

대부분의 기본 제공 연산자의 함수는 전역적으로 또는 클래스 단위로 다시 정의할 수 있습니다. 오버로드된 연산자는 함수로 구현됩니다.

오버로드된 연산자의 이름은 **operator** *x*이며 여기서 *x*는 다음 테이블에 나와 있는 연산자입니다. 예를 들어 더하기 연산자를 오버로드하려면 **operator+**라는 함수를 정의합니다. 마찬가지로, 더하기/할당 연산자 **+ =**를 오버로드하려면 **operator+=**라는 함수를 정의합니다.

다시 정의할 수 있는 연산자

[+] 테이블 확장

연산자	속성	Type
=	Comma	이진
=	논리 NOT	단항
()	같지 않음	이진
%	모듈러스	이진
%=	모듈러스 대입	이진
=	비트 AND	이진
=	Address-of	단항
()	논리적 AND	이진

연산자	속성	Type
()	비트 AND 대입	이진
..	함수 호출	=
..	캐스트 연산자	단항
*	곱하기	이진
*	포인터 역참조	단항
*=	곱하기 할당	이진
+	더하기	이진
+	단항 더하기	단항
++	증가 ¹	단항
+=	더하기 할당	이진
-	빼기	이진
-	단항 부정 연산자	단항
--	감소 ¹	단항
-=	빼기 할당	이진
=>	멤버 선택	이진
->*	멤버 포인터 선택	이진
/	나누기	이진
/=	나누기 할당	이진
<	보다 작음	이진
<<	왼쪽 시프트	이진
<<=	왼쪽 시프트 할당	이진
<=	보다 작거나 같음	이진
=	양도	이진
==	Equality	이진
>	보다 큼	이진
>=	크거나 같음	이진

연산자	속성	Type
>>	오른쪽 시프트	이진
>>=	오른쪽 시프트 할당	이진
..	배열 첨자	=
^	배타적 OR	이진
^=	배타적 OR 할당	이진
	포괄적 비트 OR	이진
=	포괄적 비트 OR 대입	이진
	논리적 OR	이진
~	1의 보수	단항
<code>delete</code>	삭제	=
<code>new</code>	새로 만들기	=
conversion operators	conversion operators	단항

¹ 단항 증가 및 감소 연산자에는 두 가지 버전 즉, 사전 증가 및 사후 증가가 있습니다.

자세한 내용은 [연산자 오버로드에 대한 일반 규칙](#)을 참조하세요. 다양한 범주의 오버로드된 연산자에 대한 제약 조건이 다음 항목에 설명되어 있습니다.

- [단항 연산자](#)
- [이항 연산자](#)
- [양도](#)
- [함수 호출](#)
- [첨자](#)
- [클래스 멤버 액세스](#)
- [증가 및 감소](#)
- [사용자 정의 형식 변환](#)

다음 테이블의 연산자는 오버로드할 수 없습니다. 이 테이블에는 전처리기 기호인 # 및 ## 기호가 포함됩니다.

다시 정의할 수 없는 연산자

[+] 테이블 확장

연산자	속성
.	멤버 선택
.*	멤버 포인터 선택
:\ = :	범위 확인 조건부
#	문자열로 전처리기 변환
##	전처리기 연결

오버로드된 연산자는 코드에서 발견되었을 때 일반적으로 컴파일러에 의해 암시적으로 호출되지만 다음과 같이 멤버 또는 비멤버 함수가 호출될 때처럼 명시적으로 호출할 수 있습니다.

C++

```
Point pt;  
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

예시

다음 예제에서는 + 연산자를 오버로드하여 두 개의 복소수를 추가하고 그 결과를 반환합니다.

C++

```
// operator_overloading.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
struct Complex {  
    Complex( double r, double i ) : re(r), im(i) {}  
    Complex operator+( Complex &other );  
    void Display( ) { cout << re << ", " << im << endl; }  
private:  
    double re, im;  
};  
  
// Operator overloaded using a member function
```

```
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

Output

6.8, 11.2

0| 섹션의 내용

- 연산자 오버로드에 대한 일반 규칙
- 단항 연산자 오버로드
- 이항 연산자
- 양도
- 함수 호출
- 첨자
- 멤버 액세스

참고 항목

C++ 기본 제공 연산자, 우선 순위 및 결합성
키워드

피드백

이 페이지가 도움이 되었나요?

Yes

No

연산자 오버로드에 대한 일반 규칙

아티클 • 2023. 10. 12.

다음 규칙은 오버로드된 연산자를 구현하는 방법을 제한합니다. 그러나 별도로 적용되는 새 및 삭제 연산자는 적용되지 않습니다.

- .와 같은 새 연산자는 정의할 수 없습니다.
- 기본 제공 데이터 형식에 적용할 때 연산자의 의미를 다시 정의할 수 없습니다.
- 오버로드된 연산자는 비정적 클래스 멤버 함수 또는 전역 함수여야 합니다. private 또는 protected 클래스 멤버에 액세스해야 하는 전역 함수는 해당 클래스의 friend로 선언해야 합니다. 전역 함수는 클래스 또는 열거형 형식이거나 클래스 또는 열거형 형식에 대한 참조인 인수를 하나 이상 사용해야 합니다. 예시:

C++

```
// rules_for_operator_overloading.cpp
class Point
{
public:
    Point operator<( Point & ); // Declare a member operator
                                // overload.
    // Declare addition operators.
    friend Point operator+( Point&, int );
    friend Point operator+( int, Point& );
};

int main()
{}
```

위의 코드 예제에서는 작은 연산자를 멤버 함수로 선언합니다. 그러나 더하기 연산자는 friend 액세스 권한이 있는 전역 함수로 선언됩니다. 지정된 연산자에 대해 둘 이상의 구현이 제공될 수 있습니다. 위에 나오는 더하기 연산자의 경우 원활한 가환성을 위해 두 가지 구현이 제공됩니다. 예, `int Point` 등에 추가하는 `Point Point` 연산자가 구현될 가능성이 높습니다.

- 연산자는 기본 제공 형식과 함께 연산자를 사용하는 일반적인 경우에 적용되는 피 연산자의 수, 우선 순위 및 그룹화를 준수합니다. 따라서 x 좌표에 2를 추가하고 3을 y 좌표에 추가해야 하므로 "형식 `Point` 의 개체에 2와 3을 추가"라는 개념을 표현할 방법이 없습니다.
- 멤버 함수로 선언된 단항 연산자는 인수를 사용하지 않습니다. 전역 함수로 선언된 경우 인수를 한 개 사용합니다.

- 멤버 함수로 선언된 이항 연산자는 인수를 한 개 사용합니다. 전역 함수로 선언된 경우 인수를 두 개 사용합니다.
- 연산자를 단항 또는 이진 연산자(&, *, + 및 -)로 사용할 수 있는 경우 각 사용을 별도로 오버로드할 수 있습니다.
- 오버로드된 연산자는 기본 인수를 사용할 수 없습니다.
- 할당(operator=)을 제외한 모든 오버로드된 연산자는 파생 클래스에서 상속됩니다.
- 멤버 함수 오버로드된 연산자의 첫 번째 인수는 항상 연산자가 호출되는 객체의 클래스 형식입니다(연산자가 선언된 클래스 또는 해당 클래스에서 파생된 클래스). 첫 번째 인수에 대한 변환은 제공되지 않습니다.

모든 연산자의 의미는 완전히 변경될 수 있습니다. 여기에는 주소(), 할당(&=) 및 함수 호출 연산자의 의미가 포함됩니다. 또한 기본 제공 형식에 의존할 수 있는 ID는 연산자 오버로드를 사용하여 변경할 수 있습니다. 예를 들어 다음 네 개의 문은 완전히 평가되면 일반적으로 동일합니다.

C++

```
var = var + 1;
var += 1;
var++;
++var;
```

연산자를 오버로드하는 클래스 형식의 경우 이 ID에 의존할 수 없습니다. 또한 기본 형식에 대한 이러한 연산자의 사용에서 암시적인 일부 요구 사항은 오버로드된 연산자의 경우 완화됩니다. 예를 들어 더하기/대입 연산 +=자는 기본 형식에 적용할 때 왼쪽 피연산자가 l-value여야 합니다. 연산자가 오버로드될 때는 이러한 요구 사항이 없습니다.

① 참고

일관성을 위해 오버로드된 연산자를 정의하는 경우 기본 제공 형식의 모델을 따르는 것이 가장 좋습니다. 오버로드된 연산자의 의미 체계가 다른 컨텍스트에서의 해당 의미와 크게 다른 경우 유용하기보다 혼동을 줄 수 있습니다.

참고 항목

[연산자 오버로드](#)

단항 연산자 오버로드

아티클 • 2024. 07. 15.

단항 연산자는 단일 피연산자로부터 결과를 생성합니다. 사용자 정의 형식에 대해 작업하기 위해 표준 단항 연산자 집합의 오버로드를 정의할 수 있습니다.

오버로드 가능한 단항 연산자

사용자 정의 형식에 대해 다음 단항 연산자를 오버로드할 수 있습니다.

- `!(논리적 NOT)`
- `&(address-of)`
- `~(보완)`
- `*(포인터 역참조)`
- `+(단항 더하기)`
- `-(단항 부정 연산자)`
- `++(접두사 증분) 또는(후위 증분)`
- `--(접두사 감소) 또는(후위 감소)`
- `변환 연산자`

단항 연산자 오버로드 선언

오버로드된 단항 연산자를 비정적 멤버 함수 또는 비멤버 함수로 선언할 수 있습니다. 오버로드된 단항 멤버 함수는 암시적으로 `this`에서 작동하므로 인수를 사용하지 않습니다. 비멤버 함수는 하나의 인수로 선언됩니다. 두 형식이 모두 선언되면 컴파일러는 오버로드 해결 규칙에 따라 사용할 함수(있는 경우)를 결정합니다.

다음 규칙은 모든 접두사 단항 연산자에 적용됩니다. 단항 연산자 함수를 비정적 멤버 함수로 선언하려면 다음 선언 형식을 사용합니다.

```
return-type operator op ();
```

이 형식에서 `return-type`은 반환 형식이고 `op`는 이전 표에 나열된 연산자 중 하나입니다.

단항 연산자 함수를 비멤버 함수로 선언하려면 다음 선언 형식을 사용합니다.

```
return-type operator op ( class-type );
```

이 형식에서 `return-type`은 반환 형식이고, `op`는 이전 표에 나열된 연산자 중 하나이며, `class-type`은 연산할 인수의 클래스 형식입니다.

`++` 및 `--`의 접미사 형식은 접두사 형식과 구분하기 위해 추가 `int` 인수를 사용합니다.
`++` 및 `--`의 접두사 및 접미어 형식에 대한 자세한 내용은 [증가 및 감소 연산자 오버로드](#)를 참조하세요.

① 참고

단항 연산자의 반환 형식에 대한 제한은 없습니다. 예를 들어, 논리적 `NOT(!)`이 `bool` 값을 반환하는 것이 합리적이지만 이 동작은 적용되지 않습니다.

참고 항목

[연산자 오버로드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

증가 및 감소 연산자 오버로드(C++)

아티클 • 2023. 10. 12.

증가 및 감소 연산자의 경우 각각 두 가지 변형이 있으므로 특수 범주에 속합니다.

- 사전 증가 및 사후 증가
- 사전 감소 및 사후 감소

오버로드된 연산자 함수를 작성하는 경우 이러한 연산자의 전위 버전과 후위 버전에 대해 별도의 버전을 구현하는 것이 유용할 수 있습니다. 둘을 구분하기 위해 다음 규칙이 관찰됩니다. 연산자의 접두사 형식은 다른 단항 연산자처럼 정확히 동일한 방식으로 선언됩니다. 후위 품은 형식 `int`의 추가 인수를 허용합니다.

① 참고

증분 또는 감소 연산자의 후위 형식에 대해 오버로드된 연산자를 지정하는 경우 추가 인수는 형식 `int`이어야 합니다. 다른 형식을 지정하면 오류가 발생합니다.

다음 예제에서는 `Point` 클래스에 대해 전위 및 후위 증가 연산자와 감소 연산자를 정의하는 방법을 보여 줍니다.

C++

```
// increment_and_decrement1.cpp
class Point
{
public:
    // Declare prefix and postfix increment operators.
    Point& operator++();           // Prefix increment operator.
    Point operator++(int);         // Postfix increment operator.

    // Declare prefix and postfix decrement operators.
    Point& operator--();           // Prefix decrement operator.
    Point operator--(int);         // Postfix decrement operator.

    // Define default constructor.
    Point() { _x = _y = 0; }

    // Define accessor functions.
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x, _y;
};

// Define prefix increment operator.
```

```

Point& Point::operator++()
{
    _x++;
    _y++;
    return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
    Point temp = *this;
    ++*this;
    return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
    Point temp = *this;
    --*this;
    return temp;
}

int main()
{
}

```

다음 함수 프로토타입을 사용하여 파일 범위(전역)에서 동일한 연산자를 정의할 수 있습니다.

C++

```

friend Point& operator++( Point& );      // Prefix increment
friend Point operator++( Point&, int );   // Postfix increment
friend Point& operator--( Point& );      // Prefix decrement
friend Point operator--( Point&, int );   // Postfix decrement

```

증가 또는 감소 연산자의 후위 형식을 나타내는 형식 `int` 의 인수는 일반적으로 인수를 전달하는 데 사용되지 않습니다. 일반적으로 0 값을 포함합니다. 그러나 이 인수는 다음과 같이 사용할 수 있습니다.

C++

```

// increment_and_decrement2.cpp
class Int
{
public:
    Int operator++( int n ); // Postfix increment operator
private:
    int _i;
};

Int Int::operator++( int n )
{
    Int result = *this;
    if( n != 0 )      // Handle case where an argument is passed.
        _i += n;
    else
        _i++;         // Handle case where no argument is passed.
    return result;
}

int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}

```

앞의 코드와 같이 증분 또는 감소 연산자를 사용하여 명시적 호출 이외의 값을 전달하는 구문은 없습니다. 이 기능을 구현하는 보다 간단한 방법은 더하기/할당 연산자($+=$)를 오버로드하는 것입니다.

참고 항목

[연산자 오버로드](#)

이항 연산자

아티클 • 2024. 11. 21.

다음 표에서는 오버로드될 수 있는 연산자 목록을 보여 줍니다.

다시 정의 가능 이항 연산자

[+] 테이블 확장

연산자	속성
=	Comma
()	같지 않음
%	모듈러스
%=	모듈러스/할당
=	비트 AND
()	논리적 AND
()	비트 AND/할당
*	곱하기
*=	곱하기/할당
+	더하기
+ =	더하기/할당
-	빼기
- =	빼기/할당
=>	멤버 선택
->*	멤버 포인터 선택
/	나누기
/ =	나누기/할당
<	보다 작음
<<	왼쪽 시프트

연산자	속성
<<=	왼쪽 시프트/할당
<=	보다 작거나 같음
=	양도
==	Equality
>	보다 큼
>=	크거나 같음
>>	오른쪽 시프트
>>=	오른쪽 시프트/할당
^	배타적 OR
^=	배타적 OR/할당
	포괄적 비트 OR
=	포괄적 비트 OR/할당
	논리적 OR

이항 연산자 함수를 비정적 멤버로 선언하려면 해당 함수를 다음과 같은 형태로 선언해야 합니다.

`ret-type operator op (arg)`

여기서 `ret-type`은 반환 형식이고, `op`는 위의 표에 나와 있는 연산자 중 하나이며, `arg`는 임의 형식의 인수입니다.

이항 연산자 함수를 전역 함수로 선언하려면 해당 함수를 다음과 같은 형태로 선언해야 합니다.

`ret-type operator op (arg1, arg2)`

여기서 `ret-type` 및 `op`은 멤버 연산자 함수에 대해 설명된 것과 동일하고, `arg1` 및 `arg2`는 인수입니다. 인수 중 하나 이상이 클래스 형식이어야 합니다.

① 참고

이항 연산자의 반환 형식에 대한 제한은 없지만 대부분의 사용자 정의 이항 연산자는 클래스 형식이나 클래스 형식에 대한 참조를 반환합니다.

참고 항목

[연산자 오버로드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

양도

아티클 • 2023. 10. 12.

대입 연산자(=)는 엄밀히 말하면 이진 연산자입니다. 할당 연산자의 선언은 다음을 제외하고 다른 모든 이항 연산자와 동일합니다.

- 비정적 멤버 함수여야 합니다. 연산자=는 nonmember 함수로 선언할 수 없습니다.
- 파생 클래스가 상속하지 않습니다.
- 클래스 형식이 없는 경우 컴파일러에서 기본 연산자= 함수를 생성할 수 있습니다.

다음 예제에서는 할당 연산자를 선언하는 방법을 보여 줍니다.

C++

```
class Point
{
public:
    int _x, _y;

    // Right side of copy assignment is the argument.
    Point& operator=(const Point&);

};

// Define copy assignment operator.
Point& Point::operator=(const Point& otherPoint)
{
    _x = otherPoint._x;
    _y = otherPoint._y;

    // Assignment operator returns left side of assignment.
    return *this;
}

int main()
{
    Point pt1, pt2;
    pt1 = pt2;
}
```

제공된 인수는 식의 오른쪽입니다. 연산자는 개체를 반환하여 할당 연산자의 동작을 보존하고, 할당 연산자는 할당이 완료된 후 왼쪽의 값을 반환합니다. 이렇게 하면 다음과 같은 할당을 연결할 수 있습니다.

C++

```
pt1 = pt2 = pt3;
```

복사 할당 연산자는 복사 생성자와 혼동해서는 안 됩니다. 후자는 기존 개체에서 새 개체를 만드는 동안 호출됩니다.

C++

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```

① 참고

복사 할당 연산자를 정의하는 클래스가 복사 생성자, 소멸자 및 C++11부터 생성자 이동 및 이동 할당 연산자를 명시적으로 정의해야 한다는 세 가지 규칙을 따르는  것이 좋습니다.

참고 항목

- [연산자 오버로드](#)
- [복사 생성자 및 복사 할당 연산자\(C++\)](#)

함수 호출 (C++)

아티클 • 2023. 10. 12.

괄호를 사용하여 호출된 함수 호출 연산자는 이항 연산자입니다.

구문

```
primary-expression ( expression-list )
```

설명

이 컨텍스트에서 `primary-expression`이 첫 번째 피연산자이며, 빈 인수 목록일 수 있는 `expression-list`가 두 번째 피연산자입니다. 함수 호출 연산자는 여러 매개 변수가 필요 한 연산에 사용됩니다. 이는 `expression-list`가 단일 피연산자가 아닌 목록이기 때문에 가능합니다. 함수 호출 연산자는 비정적 멤버 함수여야 합니다.

오버로드된 경우 함수 호출 연산자는 함수가 호출되는 방법을 수정하는 것이 아니라 지 정된 클래스 형식의 개체에 연산자가 적용될 때 연산자가 해석되는 방법을 수정합니다. 예를 들어 다음 코드는 대개 무의미합니다.

C++

```
Point pt;
pt( 3, 2 );
```

하지만 적절하게 오버로드된 함수 호출 연산자가 제공된 경우에는 이 구문을 사용하여 `x` 좌표 3 단위 및 `y` 좌표 2 단위를 오프셋할 수 있습니다. 다음 코드에 이러한 정의가 나와 있습니다.

C++

```
// function_call.cpp
class Point
{
public:
    Point() { _x = _y = 0; }
    Point &operator()( int dx, int dy )
        { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
```

```
};

int main()
{
    Point pt;
    pt( 3, 2 );
}
```

함수 호출 연산자는 함수 이름이 아닌 개체 이름에 적용됩니다.

함수 자체를 사용하기 보다는 함수에 대한 포인터를 사용하여 함수 호출 연산자를 오버로드할 수도 있습니다.

C++

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf }()
}
```

참고 항목

[연산자 오버로드](#)

첨자

아티클 • 2023. 10. 12.

함수 호출 연산자처럼 아래 첨자 연산자([])는 이진 연산자로 간주됩니다. 첨자 연산자는 단일 인수를 사용하는 비정적 멤버 함수여야 합니다. 이 인수는 어떠한 형식도 될 수 있으며 원하는 배열 첨자를 지정합니다.

예시

다음 예제에서는 경계 검사 구현하는 형식 `int` 의 벡터를 만드는 방법을 보여 줍니다.

C++

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
    IntVector( int cElements );
    ~IntVector() { delete [] _iElements; }
    int& operator[](int nSubscript);
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}

// Test the IntVector class.
int main() {
    IntVector v( 10 );
```

```
int i;

for( i = 0; i <= 10; ++i )
    v[i] = i;

v[3] = v[9];

for ( i = 0; i <= 10; ++i )
    cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

Output

```
Array bounds violation.
Element: [0] = 0
Element: [1] = 1
Element: [2] = 2
Element: [3] = 9
Element: [4] = 4
Element: [5] = 5
Element: [6] = 6
Element: [7] = 7
Element: [8] = 8
Element: [9] = 9
Array bounds violation.
Element: [10] = 10
```

설명

이전 프로그램에서 10에 도달하면 `i` 연산자[]는 범위를 벗어난 아래 첨자가 사용되고 있음을 감지하고 오류 메시지를 실행합니다.

함수 `연산자[]`는 참조 형식을 반환합니다. 이에 따라 l-value가 되므로 할당 연산자의 양 쪽에서 첨자 식을 사용할 수 있습니다.

참고 항목

[연산자 오버로드](#)

멤버 액세스

아티클 • 2023. 10. 12.

클래스 멤버 액세스는 멤버 액세스 연산자(->)를 오버로드하여 제어할 수 있습니다. 이 연산자는 이 사용법에서 단항 연산자로 간주되며 오버로드된 연산자 함수가 클래스 멤버 함수여야 합니다. 따라서 이러한 함수의 선언은 다음과 같습니다.

구문

```
class-type *operator->()
```

설명

여기서 **클래스 형식**은 이 연산자가 속한 클래스의 이름입니다. 멤버 액세스 연산자 함수는 비정적 멤버 함수여야 합니다.

이 연산자는 역참조나 개수 사용법에 앞서 포인터의 유효성을 검사하는 "스마트 포인터"를 구현하는 데 사용되며 종종 포인터 역참조 연산자와 함께 사용됩니다.

. 멤버 액세스 연산자는 오버로드할 수 없습니다.

참고 항목

[연산자 오버로드](#)

클래스 및 구조체(C++)

아티클 • 2023. 10. 12.

이 섹션에서는 C++ 클래스 및 구조체를 소개합니다. 구조체에서는 기본 접근성이 공용이고, 클래스에서는 기본값이 개인이라는 점을 제외하면 C++에서 두 구문이 동일합니다.

클래스와 구조체는 고유한 형식을 정의하는 데 사용되는 구문입니다. 클래스와 구조체 모두 형식의 상태 및 동작을 설명하는 데 사용되는 데이터 멤버와 멤버 함수를 포함합니다.

주제는 다음과 같습니다.

- [class](#)
- [struct](#)
- [클래스 멤버 개요](#)
- [멤버 액세스 제어](#)
- [상속](#)
- [정적 멤버](#)
- [사용자 정의 형식 변환](#)
- [변경 가능한 데이터 멤버\(변경 가능한 지정자\)](#)
- [중첩 클래스 선언](#)
- [익명 클래스 형식](#)
- [멤버에 대한 포인터](#)
- [this 포인터](#)
- [C++ 비트 필드](#)

세 가지 클래스 형식은 구조체, 클래스 및 공용 구조체입니다. 구조체, 클래스 및 공용 구조체 키워드(keyword) 사용하여 선언됩니다. 다음 표에서는 세 가지 클래스 형식 간의 차이점을 보여 줍니다.

노조에 대한 자세한 내용은 공용 구조체를 참조 [하세요](#). C++/CLI 및 C++/CX의 클래스 및 구조체에 대한 자세한 내용은 클래스 및 구조체를 참조 [하세요](#).

구조체, 클래스 및 공용 구조체의 Access Control 및 제약 조건

구조	클래스	Unions
클래스 키는 <code>struct</code>	클래스 키는 <code>class</code>	클래스 키는 <code>union</code>
기본 액세스가 공용임	기본 액세스가 개인임	기본 액세스가 공용임
사용 제약 조건 없음	사용 제약 조건 없음	한 번에 한 멤버만 사용

참고 항목

[C++ 언어 참조](#)

클래스 (C++)

아티클 • 2024. 11. 21.

키워드는 **class** 클래스 형식을 선언하거나 클래스 형식의 개체를 정의합니다.

구문

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

매개 변수

template-spec

선택적 템플릿 지정입니다. 자세한 내용은 템플릿을 [참조하세요](#).

class

class 키워드.

ms-decl-spec

선택적 스토리지 클래스 지정입니다. 자세한 내용은 [_declspec](#) 키워드를 참조하세요.

tag

클래스에 지정된 형식 이름입니다. 태그는 클래스 범위 내에서 예약된 단어가 됩니다. 태그는 선택 사항입니다. 생략하면 익명 클래스가 정의됩니다. 자세한 내용은 익명 클래스 형식을 참조 [하세요](#).

base-list

이 클래스가 해당 멤버를 파생하는 클래스 또는 구조의 선택적 목록입니다. 자세한 내용은 기본 클래스를 [참조하세요](#). 각 기본 클래스 또는 구조체 이름 앞에 액세스 지정자 ([public](#), [private](#), [protected](#)) 및 가상 키워드가 있을 수 있습니다. 자세한 내용은 클래스 멤버에 대한 액세스 제어의 [멤버](#) 액세스 테이블을 참조하세요.

member-list

클래스 멤버 목록입니다. 자세한 내용은 [클래스 멤버 개요](#)를 참조하세요.

declarators

클래스 형식의 하나 이상의 인스턴스 이름을 지정하는 선언자 목록입니다. 클래스의 모든 데이터 멤버가 있는 경우 선언자에 이니셜라이저 목록이 포함될 수 있습니다 `public`. 이는 클래스보다 기본적으로 데이터 멤버가 있는 구조체에서 더 일반적입니다 `public`. 자세한 내용은 선언자 [개요를 참조하세요](#).

설명

일반적인 클래스에 대한 자세한 내용은 다음 항목 중 하나를 참조하세요.

- [struct](#)
- [union](#)
- [_multiple_inheritance](#)
- [_single_inheritance](#)
- [_virtual_inheritance](#)

C++/CLI 및 C++/CX의 관리되는 클래스 및 구조체에 대한 자세한 내용은 [클래스 및 구조체를 참조하세요](#).

예시

C++

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
using namespace std;

class dog
{
public:
    dog()
    {
        _legs = 4;
        _bark = true;
    }

    void setDogSize(string dogSize)
    {
```

```

        _dogSize = dogSize;
    }
    virtual void setEars(string type)      // virtual function
    {
        _earType = type;
    }

private:
    string _dogSize, _earType;
    int _legs;
    bool _bark;

};

class breed : public dog
{
public:
    breed( string color, string size)
    {
        _color = color;
        setDogSize(size);
    }

    string getColor()
    {
        return _color;
    }

    // virtual function redefined
    void setEars(string length, string type)
    {
        _earLength = length;
        _earType = type;
    }

protected:
    string _color, _earLength, _earType;
};

int main()
{
    dog mongrel;
    breed labrador("yellow", "large");
    mongrel.setEars("pointy");
    labrador.setEars("long", "floppy");
    cout << "Cody is a " << labrador.getColor() << " labrador" << endl;
}

```

참고 항목

[키워드](#)

[클래스 및 구조체](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

struct (C++)

아티클 • 2024. 11. 21.

키워드는 **struct** 구조체 형식 및/또는 구조체 형식의 변수를 정의합니다.

구문

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[struct] tag declarators;
```

매개 변수

template-spec

선택적 템플릿 지정입니다. 자세한 내용은 템플릿 사양을 [참조하세요](#).

struct

struct 키워드.

ms-decl-spec

선택적 스토리지 클래스 지정입니다. 자세한 내용은 [_declspec 키워드](#)를 참조하세요.

tag

구조체에 지정된 형식 이름입니다. 태그는 구조체의 범위 내에서 예약어가 됩니다. 태그는 선택 사항입니다. 생략할 경우 익명 구조체가 정의됩니다. 자세한 내용은 익명 클래스 형식을 참조 [하세요](#).

base-list

이 구조체가 해당 멤버를 파생할 클래스 또는 구조체의 선택적 목록입니다. 자세한 내용은 기본 클래스를 [참조하세요](#). 각 기본 클래스 또는 구조체 이름 앞에 액세스 지정자 ([public](#), [private](#), [protected](#)) 및 가상 키워드가 있을 수 있습니다. 자세한 내용은 클래스 멤버에 대한 액세스 제어의 [멤버 액세스 테이블](#)을 참조하세요.

member-list

구조체 멤버 목록입니다. 자세한 내용은 [클래스 멤버 개요](#)를 참조하세요. 여기서 유일한 차이점은 대신 사용되는 **class** 것입니다 **struct**.

declarators

구조체의 이름을 지정하는 선언자 목록입니다. 선언자 목록은 구조체 형식의 하나 이상의 인스턴스를 선언합니다. 구조체의 모든 데이터 멤버가 이니셜라이저 목록인 경우 선언자에 이니셜라이저 목록이 포함될 수 있습니다 **public**. 데이터 멤버는 기본적으로 이니셜라이저 목록이기 때문에 구조체에서 일반적입니다 **public**. 자세한 내용은 선언자 [개요](#)를 참조하세요.

설명

구조체 형식은 사용자 정의 복합 형식입니다. 이 형식은 다른 형식을 가질 수 있는 필드 또는 멤버로 구성됩니다.

C++에서 구조체는 멤버가 기본적으로 있다는 점을 제외하고 클래스와 동일합니다

public.

C++/CLI의 관리되는 클래스 및 구조체에 대한 자세한 내용은 [클래스 및 구조체를 참조하세요](#).

구조체 사용

C에서는 키워드를 **struct** 명시적으로 사용하여 구조를 선언해야 합니다. C++에서는 형식이 정의된 후에 키워드를 **struct** 사용할 필요가 없습니다.

닫는 중괄호와 세미콜론 사이에 쉼표로 구분된 변수 이름을 하나 이상 넣어 구조체 형식이 정의될 때 변수를 선언하는 옵션이 있습니다.

구조체 변수를 초기화할 수 있습니다. 각 변수의 초기화는 중괄호로 묶어야 합니다.

관련 정보는 [클래스](#), [공용 구조체](#) 및 [열거형을 참조하세요](#).

예제

C++

```
#include <iostream>
using namespace std;

struct PERSON { // Declare PERSON struct type
    int age; // Declare member types
    long ss;
    float weight;
    char name[25];
} family_member; // Define object of type PERSON
```

```
struct CELL {    // Declare CELL bit field
    unsigned short character : 8;    // 00000000 ???????
    unsigned short foreground : 3;   // 0000??? 00000000
    unsigned short intensity : 1;   // 0000?000 00000000
    unsigned short background : 3;  // 0???0000 00000000
    unsigned short blink : 1;       // ?0000000 00000000
} screen[25][80];      // Array of bit fields

int main() {
    struct PERSON sister;    // C style structure declaration
    PERSON brother;    // C++ style structure declaration
    sister.age = 13;    // assign values to members
    brother.age = 7;
    cout << "sister.age = " << sister.age << '\n';
    cout << "brother.age = " << brother.age << '\n';

    CELL my_cell;
    my_cell.character = 1;
    cout << "my_cell.character = " << my_cell.character;
}
// Output:
// sister.age = 13
// brother.age = 7
// my_cell.character = 1
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

클래스 멤버 개요

아티클 • 2023. 10. 12.

A `class` 또는 `struct` 해당 멤버로 구성됩니다. 클래스가 수행하는 작업은 해당 멤버 함수에 의해 수행됩니다. 클래스가 유지하는 상태는 해당 데이터 멤버에 저장됩니다. 멤버 초기화는 생성자에 의해 수행되며, 클린 메모리 해제 및 리소스 해제와 같은 업 작업은 소멸자에 의해 수행됩니다. C++11 이상에서는 선언 지점에 데이터 멤버를 초기화할 수 있으며 일반적으로 초기화해야 합니다.

클래스 멤버 종류

멤버 범주 전체 목록은 다음과 같습니다.

- 특수 멤버 함수입니다.
- 멤버 함수 개요입니다.
- 기본 제공 형식 및 기타 사용자 정의 형식을 포함하여 변경 가능한 정적 데이터 멤버입니다.
- 연산자
- 중첩 클래스 선언 및.)
- 공용 구조체
- 열거형입니다.
- 비트 필드입니다.
- 친구.
- 별칭 및 `typedefs`입니다.

① 참고

Friends는 클래스 선언에 포함되므로 앞의 목록에 포함됩니다. 그러나 클래스 범위에 속하지 않으므로 true 클래스 멤버가 아닙니다.

예제 클래스 선언

다음 예제에서는 간단한 구조체 선언을 보여 줍니다.

C++

```
// TestRun.h

class TestRun
{
    // Start member list.

    // The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
    TestRun(const TestRun&) = delete;
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun();
    enum class State { Active, Suspended };

    // Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

    // Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };
```

멤버 접근성

클래스의 멤버는 멤버 목록에 선언됩니다. 클래스의 멤버 목록은 액세스 지정자라고 하는 키워드(keyword) 사용하여 여러 `private` `protected` 개 및 `public` 섹션으로 나눌 수 있습니다. 콜론은 `:` 액세스 지정자를 따라야 합니다. 이러한 섹션은 연속될 필요가 없습니다. 즉, 이러한 키워드(keyword) 구성원 목록에 여러 번 나타날 수 있습니다. 다음 액세스 지정자나 닫는 중괄호가 나올 때까지 키워드가 모든 멤버의 액세스를 지정합니다. 자세한 내용은 [멤버 액세스 제어\(C++\)](#)를 참조하세요.

정적 멤버

데이터 멤버는 정적 멤버로 선언할 수 있습니다. 이 경우 클래스의 모든 개체가 동일한 복사본에 액세스할 수 있게 됩니다. 멤버 함수는 정적 함수로 선언될 수 있으며, 이 경우 클래스의 정적 데이터 멤버에만 액세스할 수 있으며 포인터가 없습니다 `this`. 자세한 내용은 정적 데이터 멤버를 참조 [하세요](#).

특수 멤버 함수

특수 멤버 함수는 소스 코드에서 지정하지 않으면 컴파일러에서 자동으로 제공하는 함수입니다.

- 기본 생성자
- 복사 생성자
- (C++11) 이동 생성자
- 복사 할당 연산자
- (C++11) 이동 대입 연산자
- Destructor

자세한 내용은 특수 멤버 함수를 참조 [하세요](#).

멤버별 초기화

C++11 이상에서는 비정적 멤버 선언자에 이니셜라이저를 포함할 수 있습니다.

C++

```
class CanInit
{
public:
    long num {7};           // OK in C++11
    int k = 9;              // OK in C++11
    static int i = 9;        // Error: must be defined and initialized
                           // outside of class declaration.

    // initializes num to 7 and k to 9
    CanInit(){}
    // overwrites original initialized value of num:
    CanInit(int val) : num(val) {}

};

int main()
```

```
{  
}
```

멤버에 생성자에 값이 할당된 경우 해당 값은 선언 시 할당된 값을 덮어씁니다.

지정된 클래스 형식의 모든 개체에 대해 정적 데이터 멤버의 공유 복사본은 하나뿐입니다. 정적 데이터 멤버를 정의해야 하며 파일 범위에서 초기화할 수 있습니다. 정적 데이터 멤버에 대한 자세한 내용은 정적 데이터 멤버를 참조 [하세요](#). 다음 예제에서는 정적 데이터 멤버를 초기화하는 방법을 보여줍니다.

C++

```
// class_members2.cpp  
class CanInit2  
{  
public:  
    CanInit2() {} // Initializes num to 7 when new objects of type  
                  // CanInit are created.  
    long      num {7};  
    static int i;  
    static int j;  
};  
  
// At file scope:  
  
// i is defined at file scope and initialized to 15.  
// The initializer is evaluated in the scope of CanInit.  
int CanInit2::i = 15;  
  
// The right side of the initializer is in the scope  
// of the object being initialized  
int CanInit2::j = i;
```

① 참고

정의할 `CanInit2` 가 `i` 클래스의 멤버가 되도록 지정하려면 클래스 이름 `i` 가 `CanInit2` 앞에 와야 합니다.

참고 항목

[클래스 및 구조체](#)

멤버 액세스 제어(C++)

아티클 • 2024. 09. 26.

액세스 제어를 사용하면 클래스의 `public` 인터페이스를 `private` 구현 세부 정보 및 파생 클래스에서만 사용할 `protected` 멤버와 구분할 수 있습니다. 액세스 지정자는 다음 액세스 지정자가 나타날 때까지 해당 액세스 지정자 뒤에 선언된 모든 멤버에 적용됩니다.

C++

```
class Point
{
public:
    Point( int, int ) // Declare public constructor.;
    Point(); // Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:           // Declare private state variables.
    int _x;
    int _y;

protected:        // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

기본 액세스는 `private` 클래스와 구조체 또는 공용 구조체의 `public`에 있습니다. 클래스의 액세스 지정자는 순서에 관계없이 여러 번 사용할 수 있습니다. 클래스 형식의 개체에 대한 스토리지 할당은 구현에 따라 달라집니다. 그러나 컴파일러는 액세스 지정자 간에 연속적으로 더 높은 메모리 주소에 멤버 할당을 보장해야 합니다.

멤버 Access Control

[+] 테이블 확장

액세스 형식	의미
<code>private</code>	<code>private</code> 로 선언된 클래스 멤버는 클래스의 멤버 함수 및 친구(클래스 또는 함수)에서만 사용할 수 있습니다.
<code>protected</code>	<code>protected</code> 로 선언된 클래스 멤버는 클래스의 멤버 함수 및 친구(클래스 또는 함수)에서 사용할 수 있습니다. 또한 클래스에서 파생 클래스에서 사용할 수 있습니다.
<code>public</code>	<code>public</code> 로 선언된 클래스 멤버는 모든 함수에서 사용할 수 있습니다.

액세스 제어는 의도하지 않은 방식으로 개체를 사용하는 것을 방지하는 데 도움이 됩니다. 이러한 보호는 명시적 형식 변환(캐스트)을 수행할 때 손실됩니다.

① 참고

액세스 제어는 모든 이름(멤버 함수, 멤버 데이터, 중첩 클래스 및 열거자)에 동일하게 적용 가능합니다.

파생 클래스의 Access Control

파생 클래스에서 액세스할 수 있는 기본 클래스의 멤버는 두 가지 요인으로 인해 결정됩니다. 이와 동일한 요인이 파생 클래스의 상속된 멤버에 대한 액세스도 결정합니다.

- 파생 클래스가 `public` 액세스 지정자를 사용하여 기본 클래스를 선언하는지 여부입니다.
- 기본 클래스의 멤버 액세스

다음 표에서는 이러한 요인과 기본 클래스 멤버 액세스를 확인하는 방법 간의 상호 작용을 보여 줍니다.

기본 클래스의 멤버 액세스

[+] 테이블 확장

<code>private</code>	<code>protected</code>	<code>public</code>
모든 파생 액세스로 항상 액세스할 수 없음	<code>private</code> 파생을 사용하는 경우 파생 클래스에서 <code>private</code>	<code>private</code> 파생을 사용하는 경우 파생 클래스에서 <code>private</code>
	<code>protected</code> 파생을 사용하는 경우 파생 클래스에서 <code>protected</code>	<code>protected</code> 파생을 사용하는 경우 파생 클래스에서 <code>protected</code>
	<code>public</code> 파생을 사용하는 경우 파생 클래스에서 <code>protected</code>	<code>public</code> 파생을 사용하는 경우 파생 클래스에서 <code>public</code>

다음 예제에서는 액세스 파생을 보여 줍니다.

C++

```
// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
```

```

        int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
}

```

DerivedClass1에서는 멤버 함수 PublicFunc 가 **public** 멤버이고 ProtectedFunc 가 **protected** 멤버입니다. BaseClass 가 **public** 기본 클래스이기 때문입니다. PrivateFunc 는 BaseClass 에 **private**이며 모든 파생 클래스에서 액세스할 수 없습니다.

DerivedClass2에서는 함수 PublicFunc 및 ProtectedFunc 가 **private** 멤버로 간주된다. 왜냐면 BaseClass 가 **private** 기본 클래스이기 때문이다. 다시 한 번, PrivateFunc 는 BaseClass 에 **private**이며 모든 파생 클래스에서 액세스할 수 없습니다.

파생 클래스는 기본 클래스 액세스 지정자 없이 선언할 수 있습니다. 이러한 경우 파생 클래스 선언에서 **class** 키워드를 사용하는 경우 파생이 **private**로 고려됩니다. 파생 클래스 선언에서 **struct** 키워드를 사용하는 경우 파생이 **public**로 고려됩니다. 예를 들어, 다음 코드는

```
class Derived : Base
```

```
...
```

다음과 동일합니다.

C++

```
class Derived : private Base
```

```
...
```

마찬가지로, 다음 코드는

C++

```
struct Derived : Base
```

```
...
```

다음과 동일합니다.

C++

```
struct Derived : public Base
```

```
...
```

private 액세스 권한이 있다고 선언된 멤버는 해당 함수 또는 클래스가 기본 클래스의 **friend** 선언을 사용하여 선언되지 않는 한 함수 또는 파생 클래스에 액세스할 수 없습니다.

union 형식에는 기본 클래스가 있을 수 없습니다.

① 참고

private 기본 클래스를 지정할 때, 파생 클래스의 사용자가 멤버 액세스를 이해할 수 있도록 **private** 키워드를 명시적으로 사용하는 것이 좋습니다.

액세스 제어 및 정적 멤버

기본 클래스를 **private**로 지정하면 비정적 멤버에만 영향을 줍니다. **public** 파생 클래스에서 계속 정적 멤버에 액세스할 수 있습니다. 그러나 포인터, 참조 또는 개체를 사용하여 기본 클래스의 멤버에 액세스하는 경우, 액세스 제어가 다시 적용되는 변환이 필요할 수 있습니다. 다음 예시를 참조하세요.

C++

```
// access_control.cpp
class Base
{
public:
    int Print();           // Nonstatic member.
    static int CountOf(); // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};

// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount(); // Nonstatic member.
};

// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
    // Call static member function CountOf explicitly.
    int cCount = ::Base::CountOf(); // OK.

    // Call static member function CountOf using pointer.
    cCount = this->CountOf(); // C2247: 'Base::CountOf'
                           // not accessible because
                           // 'Derived1' uses 'private'
                           // to inherit from 'Base'

    return cCount;
}
```

위 코드에서는 액세스 제어가 `Derived2`에 대한 포인터를 `Base`에 대한 포인터로 변환하지 못하도록 합니다. `this` 포인터는 암시적으로 형식 `Derived2 *`입니다. `CountOf` 함수를 선택하려면, 형식 `Base *`으로 `this`를 변환해야 합니다. `Base`가 `Derived2`에 대한 private 간접 기본 클래스이므로 이러한 변환이 허용되지 않습니다. 직접 파생 클래스에 대한 포인터에 대해서만 private 기본 클래스 형식으로 변환할 수 있습니다. 따라서 `Derived1 * 형식의 포인터를 Base *` 형식으로 변환할 수 있습니다.

포인터, 참조 또는 개체를 사용하지 않고 `CountOf` 함수를 명시적으로 호출하면 변환이 수행되지 않습니다. 이것이 바로 호출이 허용되는 이유입니다.

파생 클래스인 의 멤버 및 friend는 `T`가 `T`에 대한 포인터를 `T`의 private 직접 기본 클래스에 대한 포인터로 변환할 수 있습니다.

가상 함수에 대한 액세스

`virtual` 함수에 적용된 액세스 제어는 함수를 호출하는 데 사용되는 형식에 따라 결정됩니다. 함수의 선언 재정의는 지정된 형식에 대한 액세스 제어에 영향을 주지 않습니다. 예시:

C++

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
    VFuncDerived vfd;                      // Object of derived type.
    VFuncBase *pvfb = &vfd;                // Pointer to base type.
    VFuncDerived *pvfd = &vfd;              // Pointer to derived type.
    int State;

    State = pvfb->GetState();            // GetState is public.
    State = pvfd->GetState();            // C2248 error expected; GetState is
private;
}
```

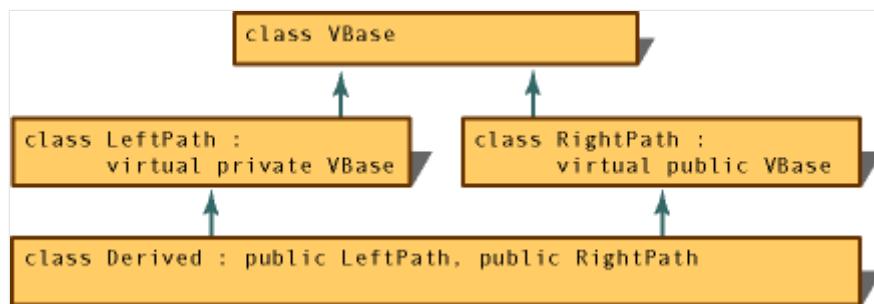
앞의 예제에서 `VFuncBase` 형식에 대한 포인터를 사용하여 가상 함수 `GetState`를 호출하면 `VFuncDerived::GetState`가 호출되며, `GetState`는 `public`으로 취급됩니다. 그러나 `GetState`가 `VFuncDerived` 클래스에서 `private`로 선언되기 때문에 `VFuncDerived` 형식에 대한 포인터를 사용하여 `GetState`를 호출하는 것은 액세스 제어 위반이 됩니다.

⊗ 주의

가상 함수 `GetState`는 기본 클래스 `VFuncBase`에 대한 포인터를 사용하여 호출할 수 있습니다. 이는 호출된 함수가 해당 함수의 기본 클래스 버전임을 의미하지는 않습니다.

다중 상속으로 액세스 제어

가상 기본 클래스를 사용하는 다중 상속 격자의 경우 지정된 이름은 둘 이상의 경로를 통해 도달할 수 있습니다. 이렇게 다양한 경로에는 다양한 액세스 제어가 적용될 수 있기 때문에 컴파일러는 최적의 액세스를 제공하는 경로를 선택합니다. 다음 그림을 참조하세요.



상속 그래프의 경로를 따라 액세스

그림에서 클래스 `VBase`에 선언된 이름은 언제나 클래스 `RightPath`를 통해 도달됩니다. `RightPath`는 `VBase`를 `public` 기본 클래스로 선언하지만, `LeftPath`는 `VBase`를 `private`로 선언합니다. 따라서, 오른쪽 경로를 보다 쉽게 액세스할 수 있습니다.

참고 항목

[C++ 언어 참조](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

friend (C++)

아티클 • 2024. 07. 12.

경우에 따라 클래스의 멤버가 아닌 함수 또는 별도의 클래스에 있는 모든 멤버에게 멤버 수준의 액세스 권한을 부여하는 것이 유용할 수 있습니다. 이러한 무료 함수 및 클래스는 **friend** 키워드로 표시된 *friends*라고 합니다. 클래스 구현자만 이 클래스의 friend를 선언할 수 있습니다. 함수 또는 클래스는 자신을 클래스의 friend로 선언할 수 없습니다. 클래스 정의에서 **friend** 키워드 및 비멤버 함수 또는 기타 클래스의 이름을 사용하여 클래스의 전용 멤버 및 보호된 멤버에 대한 액세스 권한을 부여합니다. 템플릿 정의에서 형식 매개 변수는 **friend**로 선언할 수 있습니다.

구문

friend-declaration:

```
friend function-declaration  
friend function-definition  
friend elaborated-type-specifier ;  
friend simple-type-specifier ;  
friend typename-specifier ;
```

friend 선언

이전에 선언되지 않은 **friend** 함수를 선언하는 경우 해당 함수가 바깥쪽 비클래스 범위로 내보내집니다.

friend 선언에서 선언된 함수는 **extern** 키워드를 사용하여 선언된 것처럼 취급됩니다. 자세한 내용은 [extern](#)를 참조하세요.

전역 범위를 갖는 함수는 프로토타입 이전에 **friend** 함수로 선언될 수 있지만, 멤버 함수는 전체 클래스 선언이 나타나기 전에 **friend** 함수로 선언될 수 없습니다. 다음 코드는 이러한 선언이 어떻게 실패하는지 보여줍니다.

C++

```
class ForwardDeclared; // Class name is known.  
class HasFriends  
{  
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected  
};
```

위의 예제에서는 클래스 이름 `ForwardDeclared` 를 범위에 입력하지만, 전체 선언(특히 `IsAFriend` 함수를 선언하는 부분)은 알 수 없습니다. 따라서 `HasFriends` 클래스의 `friend` 선언은 오류를 생성합니다.

C++11에서는 클래스에 대한 두 가지 형태의 `friend` 선언이 있습니다:

```
C++  
  
friend class F;  
friend F;
```

첫 번째 양식은 가장 안쪽 네임스페이스에 해당 이름의 기존 클래스가 없는 경우 새 클래스 `F`를 도입합니다. C++11: 두 번째 형식은 새로운 클래스를 도입하는 것이 아니라 클래스가 이미 선언된 경우에 사용할 수 있으며, 템플릿 유형 매개 변수 또는 `typedef` 을(를) `friend`로 선언할 때 사용해야 합니다.

참조된 형식이 아직 선언되지 않은 경우 `friend class F`를 사용합니다.

```
C++  
  
namespace NS  
{  
    class M  
    {  
        friend class F; // Introduces F but doesn't define it  
    };  
}
```

선언되지 않은 클래스 형식으로 `friend`를 사용하는 경우 오류가 발생합니다.

```
C++  
  
namespace NS  
{  
    class M  
    {  
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data  
declarations  
    };  
}
```

다음 예제에서 `friend F` 는 NS 범위 외부에서 선언된 `F` 클래스를 참조합니다.

```
C++
```

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

`friend F`를 사용하여 템플릿 매개 변수를 friend로 선언하세요.

C++

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

`friend F`를 사용하여 `typedef`을 friend로 선언하세요.

C++

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

서로 friend인 두 클래스를 선언하려면 두 번째 클래스 전체가 첫 번째 클래스의 friend로 지정되어야 합니다. 이러한 제한은 컴파일러가 두 번째 클래스가 선언된 지점에서만 개별 friend 함수를 선언하는 데 충분한 정보를 갖기 때문에 발생합니다.

① 참고

두 번째 클래스 전체가 첫 번째 클래스의 friend여야 하지만, 두 번째 클래스의 friend 가 될 첫 번째 클래스의 함수를 선택할 수 있습니다.

friend 함수

`friend` 함수는 클래스의 멤버가 아니지만 클래스의 전용 및 보호된 멤버에 액세스할 수 있는 함수입니다. `friend` 함수는 클래스 멤버로 간주되지 않으며, 특수 액세스 권한이 부여된 일반 외부 함수입니다. `friend`는 클래스 범위에 포함되지 않으며, 다른 클래스의 멤버가 아닌 경우 멤버 선택 연산자(. 및 ->)를 사용하여 호출되지 않습니다. `friend` 함수는 액세스 권한을 부여하는 클래스에서 선언됩니다. `friend` 선언은 클래스 선언의 어느 곳에나 배치될 수 있습니다. 액세스 제어 키워드의 영향을 받지 않습니다.

다음 예제에서는 `Point` 클래스와 `changePrivate`라는 `friend` 함수를 보여 줍니다. `friend` 함수는 매개 변수로 받는 `Point` 객체의 전용 데이터 멤버에 액세스할 수 있습니다.

C++

```
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
    // Output: 0
    //          1
}
```

Friend 클래스 멤버

클래스 멤버 함수는 다른 클래스에서 `friend`로 선언될 수 있습니다. 다음 예제를 참조하세요.

C++

```

// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }      // OK
int A::Func2( B& b ) { return b._b; }      // C2248

```

위의 예제에서는 `A::Func1(B&)` 함수에만 `B` 클래스에 대한 `friend` 액세스 권한이 부여됩니다. 따라서 전용 멤버 `_b`에 대한 액세스는 클래스 `A`의 `Func1`에서 올바르지만 `Func2`에서는 올바르지 않습니다.

`friend` 클래스는 모든 멤버 함수가 클래스의 `friend` 함수인 클래스입니다. 즉, 멤버 함수가 다른 클래스의 전용 및 보호된 멤버에 액세스할 수 있습니다. `friend` 클래스의 `B` 선언이 다음과 같다고 가정합니다.

C++

```
friend class A;
```

이 경우 `A` 클래스의 모든 멤버 함수에 `B` 클래스에 대한 `friend` 액세스 권한이 부여되었습니다. 다음 코드는 `friend` 클래스의 예입니다.

C++

```

// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
    friend class YourOtherClass; // Declare a friend class

```

```

public:
    YourClass() : topSecret(0){}
    void printMember() { cout << topSecret << endl; }
private:
    int topSecret;
};

class YourOtherClass {
public:
    void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}

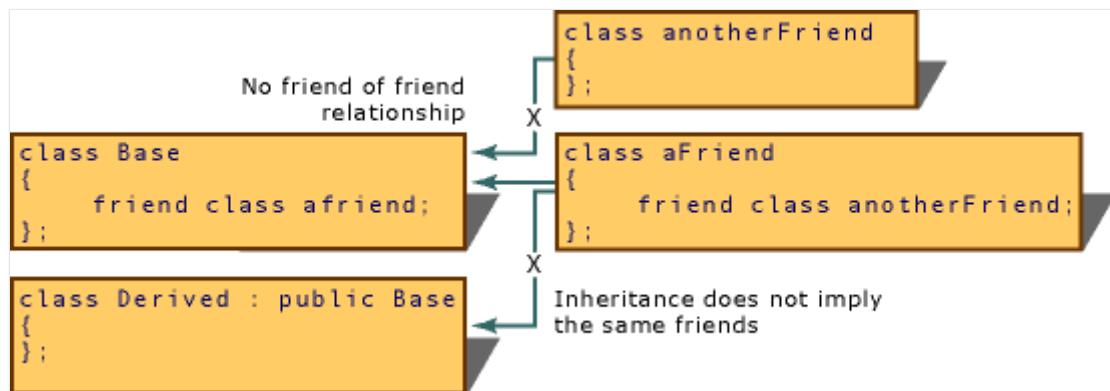
```

friendship은 이와 같이 명시적으로 지정되지 않은 경우 상호적이 아닙니다. 위의 예제에서 `YourClass`의 멤버 함수는 `YourOtherClass`의 전용 멤버에 액세스할 수 없습니다.

관리되는 형식(C++/CLI)에는 `friend` 함수, `friend` 클래스 또는 `friend` 인터페이스가 있을 수 없습니다.

friendship은 상속되지 않습니다. 즉, `YourOtherClass`에서 파생된 클래스는 `YourClass`의 전용 멤버에 액세스할 수 없습니다. friendship은 전이적이 아니므로 `YourOtherClass`의 `friend`인 클래스는 `YourClass`의 전용 멤버에 액세스할 수 없습니다.

다음 그림에서는 `Base`, `Derived`, `aFriend` 및 `anotherFriend`라는 네 가지 클래스 선언을 보여 줍니다. `aFriend` 클래스만 `Base` 전용 멤버(및 `Base`에서 상속했을 수 있는 모든 멤버)에 직접 액세스할 수 있습니다.



인라인 `friend` 정의

클래스 선언 안에 friend 함수를 정의할 수 있습니다(함수 본문 제공). 이러한 함수는 인라인 함수입니다. 멤버 인라인 함수와 같이 클래스 범위가 닫히기 전에(클래스 선언의 종료) 모든 클래스 멤버가 표시된 이후 즉시 정의되도록 동작합니다. 클래스 선언 안에 정의된 friend 함수는 바깥쪽 클래스의 범위 안에 있습니다.

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

private (C++)

아티클 • 2024. 11. 21.

구문

```
private:  
    [member-list]  
private base-class
```

설명

클래스 멤버 `private` 목록 앞에 오는 경우 키워드는 해당 멤버가 클래스의 멤버 함수 및 친구에서만 액세스할 수 있도록 지정합니다. 이 설정은 다음 액세스 지정자 또는 클래스 끝까지 선언된 모든 멤버에 적용됩니다.

기본 클래스의 이름 앞에 오는 경우 키워드는 기본 클래스 `private` 의 `public` 및 `protected` 멤버가 파생 클래스의 프라이빗 멤버임을 지정합니다.

클래스에서 멤버의 기본 액세스는 전용입니다. 구조체나 공용 구조체에서 멤버의 기본 액세스는 공용입니다.

기본 클래스의 기본 액세스는 클래스에 대해 전용이고 구조체에 대해 공용입니다. 공용 구조체에 기본 클래스를 사용할 수 없습니다.

관련 정보는 클래스 멤버에 대한 액세스 제어에서 [friend](#), [public](#), [protected](#) 및 `member-access` 테이블을 참조하세요.

/clr 관련

CLR 형식에서 C++ 액세스 지정자 키워드(`public`, `private` 및 `protected`)는 어셈블리와 관련된 형식 및 메서드의 표시 유형에 영향을 줄 수 있습니다. 자세한 내용은 [멤버 액세스 제어](#)를 참조하세요.

① 참고

[LN](#)으로 컴파일된 파일은 이 동작의 영향을 받지 않습니다. 이 경우 관리되는 클래스(공용 또는 전용)가 모두 표시됩니다.

END /clr 관련

예시

```
C++  
  
// keyword_private.cpp  
class BaseClass {  
public:  
    // privMem accessible from member function  
    int pubFunc() { return privMem; }  
private:  
    void privMem;  
};  
  
class DerivedClass : public BaseClass {  
public:  
    void usePrivate( int i )  
    { privMem = i; }    // C2248: privMem not accessible  
                      // from derived class  
};  
  
class DerivedClass2 : private BaseClass {  
public:  
    // pubFunc() accessible from derived class  
    int usePublic() { return pubFunc(); }  
};  
  
int main() {  
    BaseClass aBase;  
    DerivedClass aDerived;  
    DerivedClass2 aDerived2;  
    aBase.privMem = 1;      // C2248: privMem not accessible  
    aDerived.privMem = 1;  // C2248: privMem not accessible  
                      //     in derived class  
    aDerived2.pubFunc();  // C2247: pubFunc() is private in  
                      //     derived class  
}
```

참고 항목

[클래스 멤버에 대한 액세스 제어](#)
[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

protected (C++)

아티클 • 2024. 03. 20.

구문

```
protected:  
    [member-list]  
protected base-class
```

설명

`protected` 키워드는 *member-list*의 클래스 멤버에 대한 액세스를 다음 액세스 지정자 (`public` 또는 `private`) 또는 클래스 정의의 끝까지 지정합니다. `protected`로 선언된 클래스 멤버는 다음에만 사용할 수 있습니다.

- 원래 이 멤버를 선언한 클래스의 멤버 함수
- 원래 이 멤버를 선언한 클래스의 friend
- 원래 이 멤버를 선언한 클래스에서 공용 또는 보호된 액세스로 파생된 클래스
- 보호된 멤버에 대해 전용 액세스 권한이 있으며 전용으로 직접 파생된 클래스

기본 클래스 이름 앞에 오는 `protected` 키워드는 기본 클래스의 공용 및 보호된 멤버가 파생된 해당 클래스의 보호된 멤버라고 지정합니다.

보호된 멤버는 선언된 클래스의 멤버에만 액세스할 수 있는 `private` 멤버만큼 전용은 아니지만 어느 함수에서나 액세스할 수 있는 `public` 멤버만큼 공용도 아닙니다.

`static`으로도 선언된 보호된 멤버는 파생된 클래스의 friend나 멤버 함수에 액세스할 수 있습니다. `static`으로 선언되지 않은 보호된 멤버는 파생된 클래스의 포인터, 참조 또는 개체를 통해서만 파생된 클래스의 friend와 멤버 함수에 액세스할 수 있습니다.

관련 정보는 [friend](#), [public](#), [private](#) 및 [클래스 멤버에 대한 액세스 제어](#)의 멤버 액세스 테이블을 참조하세요.

/clr 관련

CLR 형식에서 C++ 액세스 지정자 키워드(`public`, `private` 및 `protected`)는 어셈블리와 관련된 형식 및 메서드의 표시 유형에 영향을 줄 수 있습니다. 자세한 내용은 [멤버 액세스 제어](#)를 참조하세요.

① 참고

/LN으로 컴파일된 파일은 이 동작의 영향을 받지 않습니다. 이 경우 관리되는 클래스(공용 또는 전용)가 모두 표시됩니다.

END /clr 관련

예시

C++

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
    void setProtMemb( int i ) { m_protMemb = i; }
    void Display() { cout << m_protMemb << endl; }
protected:
    int m_protMemb;
    void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
    void useProtfunc() { Protfunc(); }
} y;

int main() {
    // x.m_protMemb;           error, m_protMemb is protected
    x.setProtMemb( 0 );      // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );      // OK, uses public access function
    y.Display();
    // x.Protfunc();          error, Protfunc() is protected
    y.useProtfunc();         // OK, uses public access function
                            // in the derived class
}
```

참고 항목

클래스 멤버에 대한 액세스 제어
키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

public (C++)

아티클 • 2024. 11. 21.

구문

```
public:  
    [member-list]  
public base-class
```

설명

클래스 멤버 목록 앞에 오는 경우 키워드는 `public` 해당 멤버가 모든 함수에서 액세스할 수 있도록 지정합니다. 이 설정은 다음 액세스 지정자 또는 클래스 끝까지 선언된 모든 멤버에 적용됩니다.

기본 클래스의 이름 앞에 오는 경우 키워드는 기본 클래스 `public` 의 `public` 및 `protected` 멤버가 각각 파생 클래스의 `public` 및 `protected` 멤버임을 지정합니다.

클래스에서 멤버의 기본 액세스는 전용입니다. 구조체나 공용 구조체에서 멤버의 기본 액세스는 공용입니다.

기본 클래스의 기본 액세스는 클래스에 대해 전용이고 구조체에 대해 공용입니다. 공용 구조체에 기본 클래스를 사용할 수 없습니다.

자세한 내용은 클래스 멤버에 대한 액세스 제어의 [비공개](#), [보호된](#) 친구 및 멤버 액세스 테이블을 참조하세요.

/clr 관련

CLR 형식에서 C++ 액세스 지정자 키워드(`public`, `private` 및 `protected`)는 어셈블리와 관련된 형식 및 메서드의 표시 유형에 영향을 줄 수 있습니다. 자세한 내용은 [멤버 액세스 제어](#)를 참조하세요.

① 참고

[LN](#)으로 컴파일된 파일은 이 동작의 영향을 받지 않습니다. 이 경우 관리되는 클래스(공용 또는 전용)가 모두 표시됩니다.

END /clr 관련

예시

```
C++  
  
// keyword_public.cpp  
class BaseClass {  
public:  
    int pubFunc() { return 0; }  
};  
  
class DerivedClass : public BaseClass {};  
  
int main() {  
    BaseClass aBase;  
    DerivedClass aDerived;  
    aBase.pubFunc();           // pubFunc() is accessible  
                            // from any function  
    aDerived.pubFunc();        // pubFunc() is still public in  
                            // derived class  
}
```

참고 항목

클래스 멤버에 대한 액세스 제어
키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

중괄호 초기화

아티클 • 2023. 10. 12.

특히 비교적 간단한 생성자에 대한 `class` 생성자를 항상 정의할 필요는 없습니다. 사용자는 다음 예제와 같이 균일한 초기화를 사용하거나 `struct` 개체를 초기화 `class` 할 수 있습니다.

C++

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t)
    :
        stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum},
        minTemp{minimum} {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // When there's no constructor, an empty brace initializer does
    // value initialization = {0,0,0,0,0}
    TempData td_emptyInit{};

    // Uninitialized = if used, emits warning C4700 uninitialized local
    // variable
    TempData td_noInit;
```

```

// Member declaration (in order of ctor parameters)
TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

return 0;
}

```

`class` 생성자가 있거나 `struct` 없는 경우 멤버가 애 선언되는 `class` 순서대로 목록 요소를 제공합니다. `class` 생성자가 있는 경우 매개 변수 순서대로 요소를 제공합니다. 형식에 암시적으로 또는 명시적으로 선언된 기본 생성자가 있는 경우 중괄호 초기화를 빈 중괄호와 함께 사용하여 호출할 수 있습니다. 예를 들어 빈 중괄호 초기화와 비어있지 않은 중괄호 초기화를 모두 사용하여 다음 `class` 을 초기화할 수 있습니다.

C++

```

#include <string>
using namespace std;

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : m_string{ str }, m_double{ dbl } {}
double m_double;
string m_string;
};

int main()
{
    class_a c1{};
    class_a c1_1;

    class_a c2{ "ww" };
    class_a c2_1("xx");

    // order of parameters is the same as the constructor
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}

```

클래스에 기본값이 아닌 생성자가 있는 경우 클래스 멤버가 중괄호 이니셜라이저에 나타나는 순서는 해당 매개 변수가 생성자에 나타나는 순서이며, 멤버가 선언되는 순서는 아닙니다(이전 예제와 `class_a` 같이). 그렇지 않으면 형식에 선언된 생성자가 없으면 멤버 이니셜라이저가 선언된 순서와 동일한 순서로 중괄호 이니셜라이저에 표시되어야 합니다. 이 경우 원하는 만큼 공용 멤버를 초기화할 수 있지만 멤버를 건너뛸 수는 없습니다. 다음 예제에서는 선언된 생성자가 없을 때 중괄호 초기화에 사용되는 순서를 보여줍니다.

C++

```

class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // compiler error
    class_d d5{ "string", 'c', 2.0 }; // compiler error
}

```

기본 생성자가 명시적으로 선언되었지만 삭제된 것으로 표시된 경우 빈 중괄호 초기화를 사용할 수 없습니다.

C++

```

class class_f {
public:
    class_f() = delete;
    class_f(string x): m_string { x } {}
    string m_string;
};
int main()
{
    class_f cf{ "hello" };
    class_f cf1{}; // compiler error C2280: attempting to reference a
deleted function
}

```

일반적으로 초기화를 수행하는 모든 위치에서 중괄호 초기화를 사용할 수 있습니다(예: 함수 매개 변수 또는 반환 값으로 또는 키워드(keyword) 사용 `new`).

C++

```

class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };

```

모든 이상에서는 `/std:c++17` 빈 중괄호 초기화에 대한 규칙이 약간 더 제한적입니다. 파생 생성자 및 확장 집계 초기화를 참조 [하세요](#).

initializer_list 생성자

initializer_list 클래스는 생성자 및 다른 컨텍스트에서 사용할 수 있는 지정된 형식의 개체 목록을 나타냅니다. 중괄호 초기화를 사용하여 initializer_list 생성할 수 있습니다.

C++

```
initializer_list<int> int_list{5, 6, 7};
```

① 중요

이 클래스를 사용하려면 `initializer_list` 헤더를 <포함해야 합니다.

복사 `initializer_list` 할 수 있습니다. 이 경우 새 목록의 멤버는 원래 목록의 멤버에 대한 참조입니다.

C++

```
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"
```

표준 라이브러리 컨테이너 클래스 및 `string` `wstring` `regex` 또한 생성자가 있습니다. 다음 예제에서는 이러한 생성자를 사용하여 중괄호 초기화를 수행하는 방법을 보여 줍니다.

C++

```
vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };
```

참고 항목

[클래스 및 구조체](#)

[생성자](#)

개체 수명 및 리소스 관리(RAII)

아티클 • 2023. 10. 12.

관리되는 언어와 달리 C++에는 프로그램이 실행될 때 힙 메모리 및 기타 리소스를 해제하는 내부 프로세스인 자동 가비지 수집이 없습니다. C++ 프로그램은 획득한 모든 리소스를 운영 체제로 반환하는 역할을 담당합니다. 사용하지 않는 리소스를 해제하지 못한 경우 누수라고 합니다. 프로세스가 종료될 때까지 유출된 리소스는 다른 프로그램에서 사용할 수 없습니다. 특히 메모리 누수는 C 스타일 프로그래밍에서 버그의 일반적인 원인입니다.

최신 C++는 스택에 개체를 선언하여 힙 메모리를 최대한 사용하지 않도록 방지합니다. 리소스가 스택에 비해 너무 크면 개체가 소유해야 합니다. 개체가 초기화되면 소유하는 리소스를 획득합니다. 그런 다음, 개체는 소멸자에서 리소스를 해제합니다. 소유 개체 자체는 스택에 선언됩니다. 개체가 리소스를 소유한다는 원칙을 "리소스 획득은 초기화입니다" 또는 RAII라고도 합니다.

리소스 소유 스택 개체가 벗어나면 소멸자가 자동으로 호출됩니다. 이러한 방식으로 C++의 가비지 수집은 개체 수명과 밀접한 관련이 있으며 결정적입니다. 리소스는 항상 제어할 수 있는 프로그램의 알려진 지점에서 해제됩니다. C++와 같은 결정적 소멸자만 메모리 및 비 메모리 리소스를 동일하게 처리할 수 있습니다.

다음 예제에서는 간단한 개체 `w`를 보여줍니다. 함수 범위의 스택에 선언되고 함수 블록의 끝에서 제거됩니다. 개체 `w`는 리소스(예: 힙 할당 메모리)를 소유하지 않습니다. 유일한 멤버 `g`는 자체적으로 스택에 선언되며, 단순히 함께 `w` 범위를 벗어나갑니다. 소멸자에서 특별한 코드가 `widget` 필요하지 않습니다.

C++

```
class widget {
private:
    gadget g;    // lifetime automatically tied to enclosing object
public:
    void draw();
};

void functionUsingWidget () {
    widget w;    // lifetime automatically tied to enclosing scope
                 // constructs w, including the w.g gadget member
    // ...
    w.draw();
    // ...
} // automatic destruction and deallocation for w and w.g
// automatic exception safety,
// as if "finally { w.dispose(); w.g.dispose(); }"
```

다음 예제 `w` 에서는 메모리 리소스를 소유하므로 해당 소멸자의 코드가 있어야 메모리를 삭제할 수 있습니다.

C++

```
class widget
{
private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                       // constructs w, including the w.data member
    w.do_something();

} // automatic destruction and deallocation for w and w.data
```

C++11부터 표준 라이브러리의 스마트 포인터를 사용하여 이전 예제를 작성하는 더 좋은 방법이 있습니다. 스마트 포인터는 소유하는 메모리의 할당 및 삭제를 처리합니다. 스마트 포인터를 사용하면 클래스에서 명시적 소멸자가 `widget` 필요하지 않습니다.

C++

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                       // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

메모리 할당에 스마트 포인터를 사용하면 메모리 누수 가능성이 제거될 수 있습니다. 이 모델은 파일 핸들 또는 소켓과 같은 다른 리소스에 대해 작동합니다. 클래스에서 비슷한

방식으로 고유한 리소스를 관리할 수 있습니다. 자세한 내용은 스마트 포인터를 참조 [하세요](#).

C++의 디자인은 개체가 범위를 벗어날 때 제거되도록 합니다. 즉, 블록이 종료되면 역순으로 생성됩니다. 개체가 제거되면 해당 베이스와 멤버가 특정 순서로 제거됩니다. 전역 범위에서 블록 외부에서 선언된 개체는 문제가 발생할 수 있습니다. 전역 개체의 생성자가 예외를 throw하는 경우 디버그하기 어려울 수 있습니다.

참고 항목

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

컴파일 시간 캡슐화에 대한 Pimpl(모던 C++)

아티클 • 2023. 10. 12.

여드름 관용구는 구현을 숨기고, 결합을 최소화하고, 인터페이스를 분리하는 최신 C++ 기술입니다. 여드름은 "구현에 대한 포인터"에 대한 짧은입니다. 이미 개념에 익숙하지만 Cheshire Cat 또는 컴파일러 방화벽 관용구와 같은 다른 이름으로 알고 있을 수 있습니다.

왜 여드름을 사용합니까?

여드름 관용구가 소프트웨어 개발 수명 주기를 개선하는 방법은 다음과 같습니다.

- 컴파일 종속성 최소화
- 인터페이스와 구현의 분리입니다.
- 이식성.

포클 헤더

C++

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

여드름 관용구는 계단식 및 부서지기 쉬운 개체 레이아웃을 다시 작성하지 않습니다. (전 이적으로) 인기 있는 유형에 적합합니다.

Pimpl 구현

impl.cpp 파일에서 클래스를 정의합니다.

C++

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
```

```
};

my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

모범 사례

throw하지 않는 스왑 특수화에 대한 지원을 추가할지 여부를 고려합니다.

참고 항목

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

ABI 경계의 이식성

아티클 • 2023. 10. 12.

이진 인터페이스 경계에서 충분히 이식 가능한 형식 및 규칙을 사용합니다. "이식 가능한 형식"은 C 기본 제공 형식 또는 C 기본 제공 형식만 포함하는 구조체입니다. 클래스 형식은 호출자와 호출자가 레이아웃, 호출 규칙 등에 동의하는 경우에만 사용할 수 있습니다. 이는 둘 다 동일한 컴파일러 및 컴파일러 설정으로 컴파일되는 경우에만 가능합니다.

C 이식성을 위해 클래스를 평면화하는 방법

호출자가 다른 컴파일러/언어로 컴파일될 수 있는 경우 특정 호출 규칙을 사용하여 `extern "C"` API로 "평면화"합니다.

C++

```
// class widget {
//     widget();
//     ~widget();
//     double method( int, gadget& );
// };
extern "C" {           // functions using explicit "this"
    struct widget;      // opaque type (forward declaration only)
    widget* STDCALL widget_create();          // constructor creates new "this"
    void STDCALL widget_destroy(widget*); // destructor consumes "this"
    double STDCALL widget_method(widget*, int, gadget*); // method uses
    "this"
}
```

참고 항목

[C++ 시작하기](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

생성자 (C++)

아티클 • 2023. 04. 03.

클래스가 멤버를 초기화하는 방법을 사용자 지정하거나 클래스의 개체를 만들 때 함수를 호출하려면 생성자를 정의합니다. 생성자는 클래스와 같은 이름을 사용하며 반환 값이 없습니다. 다양한 방법으로 초기화를 사용자 지정하는 데 필요한 만큼 오버로드된 생성자를 정의할 수 있습니다. 일반적으로 생성자는 클래스 정의 또는 상속 계층 구조 외부의 코드가 클래스의 개체를 만들 수 있도록 공용 접근성을 갖습니다. 그러나 생성자를 또는 `private`로 `protected` 선언할 수도 있습니다.

생성자는 필요에 따라 멤버 이니셜라이저 목록을 사용할 수 있습니다. 생성자 본문에 값을 할당하는 것보다 클래스 멤버를 초기화하는 더 효율적인 방법입니다. 다음 예제에서는 세 개의 오버로드된 생성자가 있는 클래스 `Box`를 보여줍니다. 마지막 두 개의 `use` 멤버 `init` 목록은 다음과 같습니다.

C++

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    explicit Box(int i) : m_width(i), m_length(i), m_height(i) // member
    init list
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}

    int Volume() { return m_width * m_length * m_height; }

private:
    // Will have value of 0 when default constructor is called.
    // If we didn't zero-init here, default constructor would
    // leave them uninitialized with garbage values.
    int m_width{ 0 };
    int m_length{ 0 };
    int m_height{ 0 };
};
```

클래스의 인스턴스를 선언하면 컴파일러는 오버로드 확인 규칙에 따라 호출할 생성자를 선택합니다.

C++

```

int main()
{
    Box b; // Calls Box()

    // Using uniform initialization (preferred):
    Box b2 {5}; // Calls Box(int)
    Box b3 {5, 8, 12}; // Calls Box(int, int, int)

    // Using function-style notation:
    Box b4(2, 4, 6); // Calls Box(int, int, int)
}

```

- 생성자는 `, , explicitfriend` 또는 `constexpr`로 `inline` 선언될 수 있습니다.
- 생성자는 또는 `volatile const volatile`로 `const` 선언된 개체를 초기화할 수 있습니다. 생성자가 완료된 `const` 후 개체가 됩니다.
- 구현 파일에서 생성자를 정의하려면 다른 멤버 함수 `Box::Box(){...}`와 마찬가지로 정규화된 이름을 지정합니다.

멤버 이니셜라이저 목록

생성자에는 필요에 따라 생성자 본문이 실행되기 전에 클래스 멤버를 초기화하는 멤버 이니셜라이저 목록이 있을 수 있습니다. (멤버 이니셜라이저 목록은 형식 `std::initializer_list<T>`의 이니셜라이저 목록과 동일하지 않습니다.)

생성자 본문에 값을 할당하는 경우보다 멤버 이니셜라이저 목록을 선호합니다. 멤버 이니셜라이저 목록은 멤버를 직접 초기화합니다. 다음 예제에서는 콜론 뒤의 모든 `identifier(argument)` 식으로 구성된 멤버 이니셜라이저 목록을 보여 줍니다.

C++

```

Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{}

```

식별자는 클래스 멤버를 참조해야 합니다. 인수 값을 사용하여 초기화됩니다. 인수는 생성자 매개 변수, 함수 호출 또는 `std::initializer_list<T>` 중 하나일 수 있습니다.

`const` 멤버 이니셜라이저 목록에서 참조 형식의 멤버와 멤버를 초기화해야 합니다.

파생 생성자가 실행되기 전에 기본 클래스가 완전히 초기화되도록 하려면 이니셜라이저 목록에서 매개 변수가 있는 기본 클래스 생성자를 호출합니다.

기본 생성자

기본 생성자에는 일반적으로 매개 변수가 없지만 기본값이 있는 매개 변수가 있을 수 있습니다.

C++

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/}

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h),
m_length(l){}
...
}
```

기본 생성자는 **특수 멤버 함수** 중 하나입니다. 클래스에 선언된 생성자가 없는 경우 컴파일러는 암시적 **inline** 기본 생성자를 제공합니다.

C++

```
#include <iostream>
using namespace std;

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}
```

암시적 기본 생성자를 사용하는 경우 이전 예제와 같이 클래스 정의에서 멤버를 초기화해야 합니다. 이러한 이니셜라이저가 없으면 멤버가 초기화되지 않고 Volume() 호출이 가비지 값을 생성합니다. 일반적으로 암시적 기본 생성자를 사용하지 않는 경우에도 이러한 방식으로 멤버를 초기화하는 것이 좋습니다.

컴파일러가 암시적 기본 생성자를 **삭제**된 것으로 정의하여 생성하지 못하도록 방지할 수 있습니다.

C++

```
// Default constructor  
Box() = delete;
```

클래스 멤버를 기본 생성할 수 없는 경우 컴파일러에서 생성된 기본 생성자는 삭제된 것으로 정의됩니다. 예를 들어 클래스 형식의 모든 멤버와 해당 클래스 형식 멤버에는 액세스할 수 있는 기본 생성자 및 소멸자가 있어야 합니다. 참조 형식의 모든 데이터 멤버와 모든 `const` 멤버에는 기본 멤버 이니셜라이저가 있어야 합니다.

컴파일러에서 생성된 기본 생성자를 호출하고 괄호를 사용하려고 하면 다음과 같은 경고가 발생합니다.

C++

```
class myclass{};  
int main(){  
    myclass mc();      // warning C4930: prototyped function not called (was a  
                      // variable definition intended?)  
}
```

이 문은 "가장 벡싱 구문 분석" 문제의 예입니다. 함수 선언 또는 기본 생성자의 호출로 해석 `myclass md();` 할 수 있습니다. C++ 파서는 다른 항목보다 선언을 선호하므로 식은 함수 선언으로 처리됩니다. 자세한 내용은 [대부분의 Vexing 구문 분석을 참조하세요 ↴](#).

기본이 아닌 생성자가 선언된 경우 컴파일러는 기본 생성자를 제공하지 않습니다.

C++

```
class Box {  
public:  
    Box(int width, int length, int height)  
        : m_width(width), m_length(length), m_height(height){}  
private:  
    int m_width;  
    int m_length;  
    int m_height;  
  
};  
  
int main(){  
  
    Box box1(1, 2, 3);  
    Box box2{ 2, 3, 4 };  
    Box box3; // C2512: no appropriate default constructor available  
}
```

클래스에 기본 생성자가 없는 경우 대괄호 구문만 사용하여 해당 클래스의 개체 배열을 생성할 수 없습니다. 예를 들어 이전 코드 블록을 고려할 때 `Box` 배열은 다음과 같이 선언

할 수 없습니다.

C++

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

그러나 이니셜라이저 목록 집합을 사용하여 Box 개체의 배열을 초기화할 수 있습니다.

C++

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

자세한 내용은 [이니셜라이저를 참조하세요.](#)

복사 생성자

복사 생성자는 동일한 형식의 개체에서 멤버 값을 복사하여 개체를 초기화합니다. 클래스 멤버가 스칼라 값과 같은 모든 간단한 형식인 경우 컴파일러에서 생성된 복사 생성자로 충분하며 사용자 고유의 형식을 정의할 필요가 없습니다. 클래스에 더 복잡한 초기화가 필요한 경우 사용자 지정 복사 생성자를 구현해야 합니다. 예를 들어 클래스 멤버가 포인터인 경우 복사 생성자를 정의하여 새 메모리를 할당하고 다른 클래스의 뾰족한 개체에서 값을 복사해야 합니다. 컴파일러에서 생성된 복사 생성자는 포인터를 복사하기만 하면 새 포인터가 다른 포인터의 메모리 위치를 계속 가리킵니다.

복사 생성자에는 다음 서명 중 하나가 있을 수 있습니다.

C++

```
Box(Box& other); // Avoid if possible--allows modification of other.  
Box(const Box& other);  
Box(volatile Box& other);  
Box(volatile const Box& other);  
  
// Additional parameters OK if they have default values  
Box(Box& other, int i = 42, string label = "Box");
```

복사 생성자를 정의할 때 복사 할당 연산자(=)도 정의해야 합니다. 자세한 내용은 [할당 및 복사 생성자 및 복사 할당 연산자를 참조하세요.](#)

복사 생성자를 삭제된 것으로 정의하여 개체가 복사되지 않도록 방지할 수 있습니다.

C++

```
Box (const Box& other) = delete;
```

개체를 복사하려고 시도하면 C2280 오류가 발생합니다. 삭제된 함수를 참조하려고 시도합니다.

생성자 이동

이동 생성자는 원래 데이터를 복사하지 않고 기존 개체의 데이터 소유권을 새 변수로 이동하는 특수 멤버 함수입니다. rvalue 참조를 첫 번째 매개 변수로 사용하고 이후 매개 변수에는 기본값이 있어야 합니다. 이동 생성자는 큰 개체를 전달할 때 프로그램의 효율성을 크게 높일 수 있습니다.

C++

```
Box(Box&& other);
```

컴파일러는 개체가 동일한 형식의 다른 개체에 의해 초기화될 때 다른 개체가 제거될 예정이며 더 이상 리소스가 필요하지 않은 경우 이동 생성자를 선택합니다. 다음 예제에서는 오버로드 확인으로 이동 생성자를 선택한 경우를 보여 주는 예제입니다. 를 호출 `get_Box()` 하는 생성자에서 반환된 값은 `xvalue` (eXpiring 값)입니다. 변수에 할당되지 않으므로 범위를 벗어나려고 합니다. 이 예제에 대한 동기를 부여하기 위해 `Box`에 해당 내용을 나타내는 큰 문자열 벡터를 제공해 보겠습니다. 이동 생성자는 벡터와 해당 문자열을 복사하는 대신 만료되는 값 "box"에서 "도용"하여 이제 벡터가 새 개체에 속하도록 합니다. 및 `string` 클래스가 모두 `vector` 고유한 이동 생성자를 구현하기 `std::move` 때문에 에 대한 호출이 모두 필요합니다.

C++

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Box {
public:
    Box() { std::cout << "default" << std::endl; }
    Box(int width, int height, int length)
        : m_width(width), m_height(height), m_length(length)
    {
        std::cout << "int,int,int" << std::endl;
    }
    Box(Box& other)
        : m_width(other.m_width), m_height(other.m_height),
        m_length(other.m_length)
    {
        std::cout << "copy" << std::endl;
    }
}
```

```

    Box(Box&& other) : m_width(other.m_width), m_height(other.m_height),
m_length(other.m_length)
{
    m_contents = std::move(other.m_contents);
    std::cout << "move" << std::endl;
}
int Volume() { return m_width * m_height * m_length; }
void Add_Item(string item) { m_contents.push_back(item); }
void Print_Contents()
{
    for (const auto& item : m_contents)
    {
        cout << item << " ";
    }
}
private:
    int m_width{ 0 };
    int m_height{ 0 };
    int m_length{ 0 };
    vector<string> m_contents;
};

Box get_Box()
{
    Box b(5, 10, 18); // "int,int,int"
    b.Add_Item("Toupee");
    b.Add_Item("Megaphone");
    b.Add_Item("Suit");

    return b;
}

int main()
{
    Box b; // "default"
    Box b1(b); // "copy"
    Box b2(get_Box()); // "move"
    cout << "b2 contents: ";
    b2.Print_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open
    return 0;
}

```

클래스가 이동 생성자를 정의하지 않으면 컴파일러는 사용자가 선언한 복사 생성자, 복사 할당 연산자, 이동 할당 연산자 또는 소멸자가 없는 경우 암시적 생성자를 생성합니다. 명시적 또는 암시적 이동 생성자가 정의되지 않은 경우 이동 생성자를 사용하지 않는 작업은 복사 생성자를 대신 사용합니다. 클래스가 이동 생성자 또는 이동 할당 연산자를 선언하는 경우 암시적으로 선언된 복사 생성자는 삭제됨으로 정의됩니다.

클래스 형식의 멤버에 소멸자가 없거나 컴파일러가 이동 작업에 사용할 생성자를 확인할 수 없는 경우 암시적으로 선언된 이동 생성자는 삭제됨으로 정의됩니다.

사소한 이동 생성자를 작성하는 방법에 대한 자세한 내용은 [이동 생성자 및 이동 할당 연산자\(C++\)](#)를 참조하세요.

명시적으로 기본 설정 및 삭제된 생성자

명시적으로 복사 생성자, 기본 생성자, 이동 생성자, 복사 할당 연산자, 이동 할당 연산자 및 소멸자를 기본값으로 지정할 수 있습니다. 모든 특수 멤버 함수를 명시적으로 삭제할 수 있습니다.

C++

```
class Box2
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

자세한 내용은 [명시적으로 기본값 및 삭제된 함수를 참조하세요](#).

constexpr 생성자

경우 생성자를 [constexpr](#)로 선언할 수 있습니다.

- 기본값으로 선언되거나 일반적으로 [constexpr](#) 함수에 대한 모든 조건을 충족합니다.
- 클래스에 가상 기본 클래스가 없습니다.
- 각 매개 변수는 [리터럴 형식](#)입니다.
- 본문은 함수 try-block이 아닙니다.
- 모든 비정적 데이터 멤버 및 기본 클래스 하위 개체가 초기화됩니다.
- 클래스가 (a) 변형 멤버가 있는 공용 구조체이거나 (b)에 익명 공용 구조체가 있는 경우 공용 구조체 멤버 중 하나만 초기화됩니다.
- 클래스 형식의 모든 비정적 데이터 멤버 및 모든 기본 클래스 하위 개체에는 [constexpr](#) 생성자가 있습니다.

이니셜라이저 목록 생성자

생성자가 를 매개 변수로 사용하고 `std::initializer_list<T>` 다른 매개 변수에 기본 인수가 있는 경우 직접 초기화를 통해 클래스가 인스턴스화될 때 해당 생성자가 오버로드 확인에서 선택됩니다. `initializer_list` 사용하여 수락할 수 있는 멤버를 초기화할 수 있습니다. 예를 들어 `Box` 클래스(이전에 표시됨)에 멤버 `m_contents` 가 있다고 가정합니다 `std::vector<string>`. 다음과 같은 생성자를 제공할 수 있습니다.

C++

```
Box(initializer_list<string> list, int w = 0, int h = 0, int l = 0)
    : m_contents(list), m_width(w), m_height(h), m_length(l)
{}
```

그런 다음 다음과 같이 `Box` 개체를 만듭니다.

C++

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

명시적 생성자

클래스에 단일 매개 변수를 사용하는 생성자가 있거나 하나를 제외한 모든 매개 변수에 기본값을 사용하는 경우 이 매개 변수 형식은 클래스 형식으로 암시적으로 변환할 수 있습니다. 예를 들어 `Box` 클래스에 다음과 같은 생성자가 있는 경우입니다.

C++

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

다음과 같이 `Box`를 초기화할 수 있습니다.

C++

```
Box b = 42;
```

또는 `Box`를 사용하는 함수에 `int`를 전달합니다.

C++

```
class ShippingOrder
{
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage){}
```

```
private:  
    Box m_box;  
    double m_postage;  
}  
//elsewhere...  
ShippingOrder so(42, 10.8);
```

경우에 따라 이러한 변환은 유용할 수 있지만 코드에서 미세하지만 심각한 오류를 발생시키는 경우가 더 자주 있습니다. 일반적으로 이러한 종류의 암시적 형식 변환을 방지하려면 생성자(및 사용자 정의 연산자)에서 키워드를 사용해야 `explicit` 합니다.

C++

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

생성자가 명시적인 경우 `ShippingOrder so(42, 10.8);` 줄에서 컴파일 오류가 발생합니다. 자세한 내용은 [사용자 정의 형식 변환을 참조하세요](#).

건설 순서

생성자는 다음과 같은 순서로 작업을 수행합니다.

1. 생성자는 선언 순서대로 기본 클래스 및 멤버 생성자를 호출합니다.
2. 클래스가 가상 기본 클래스에서 파생된 경우 해당 클래스는 개체의 가상 기본 포인터를 초기화합니다.
3. 클래스가 가상 함수를 포함하거나 상속하는 경우 해당 클래스는 개체의 가상 함수 포인터를 초기화합니다. 가상 함수 포인터는 클래스의 가상 함수 테이블을 가리켜서 가상 함수 호출의 올바른 바인딩이 코딩되도록 합니다.
4. 이러한 포인터는 함수 본문의 모든 코드를 실행합니다.

다음 예제에서는 파생 클래스의 생성자에서 기본 클래스 및 멤버 생성자가 호출되는 순서를 보여 줍니다. 먼저 기본 생성자가 호출됩니다. 그런 다음 기본 클래스 멤버가 클래스 선언에 표시되는 순서대로 초기화됩니다. 마지막으로 파생 생성자가 호출됩니다.

C++

```
#include <iostream>  
  
using namespace std;  
  
class Contained1 {  
public:  
    Contained1() { cout << "Contained1 ctor\n"; }
```

```

};

class Contained2 {
public:
    Contained2() { cout << "Contained2 ctor\n"; }
};

class Contained3 {
public:
    Contained3() { cout << "Contained3 ctor\n"; }
};

class BaseContainer {
public:
    BaseContainer() { cout << "BaseContainer ctor\n"; }
private:
    Contained1 c1;
    Contained2 c2;
};

class DerivedContainer : public BaseContainer {
public:
    DerivedContainer() : BaseContainer() { cout << "DerivedContainer
ctor\n"; }
private:
    Contained3 c3;
};

int main() {
    DerivedContainer dc;
}

```

출력은 다음과 같습니다.

Output

```

Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor

```

파생 클래스 생성자는 항상 기본 클래스 생성자를 호출하므로, 완전히 생성된 기본 클래스를 통해서만 다른 작업을 수행할 수 있습니다. 기본 클래스 생성자는 파생 순서대로 호출됩니다. 예를 들어 예제에서 파생된 경우 `ClassA` 생성자가 먼저 호출된다. `ClassB` 다음 `ClassC` `ClassA` 생성자, 생성자가 호출됩니다.

기본 클래스에 기본 생성자가 없는 경우 파생 클래스 생성자에서 기본 클래스 생성자 매개 변수를 제공해야 합니다.

C++

```
class Box {
public:
    Box(int width, int length, int height){
        m_width = width;
        m_length = length;
        m_height = height;
    }

private:
    int m_width;
    int m_length;
    int m_height;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, const string label&)
        : Box(width, length, height){
        m_label = label;
    }

private:
    string m_label;
};

int main(){

    const string aLabel = "aLabel";
    StorageBox sb(1, 2, 3, aLabel);
}
```

생성자가 예외를 throw하는 경우 소멸 순서는 생성의 역순입니다.

1. 생성자 함수 본문의 코드가 해제됩니다.
2. 기본 클래스 및 멤버 개체가 선언의 역순으로 제거됩니다.
3. 생성자가 위임되지 않으면 완전히 생성된 모든 기본 클래스 개체와 멤버가 제거됩니다. 그러나 개체 자체가 완전히 생성되지 않으므로 소멸자를 실행하지 않습니다.

파생 생성자 및 확장 초기화

기본 클래스의 생성자가 공용이 아니지만 파생 클래스에 액세스할 수 있는 경우 빈 중괄호를 사용하여 Visual Studio 2017 이상에서 모드에서 `/std:c++17` 파생된 형식의 개체를 초기화할 수 없습니다.

다음 예제에서는 C++14 준수 동작을 보여 줍니다.

C++

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};
```

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
 // which can call Base ctor.

C++17에서 `Derived`는 이제 집계 형식으로 간주되므로 프라이빗 기본 생성자를 통한 `Base`의 초기화가 확장된 집계 초기화 규칙의 일부로 직접 발생합니다. `Base` 이전에는 프라이빗 생성자가 생성자를 통해 `Derived` 호출되었고 선언으로 인해 `friend` 성공했습니다.

다음 예제에서는 Visual Studio 2017 이상 모드의 C++17 동작을 `/std:c++17` 보여줍니다.

C++

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': can't access
               // private member declared in class 'Base'
```

여러 상속이 있는 클래스에 대한 생성자

클래스가 여러 기본 클래스에서 파생된 경우 기본 클래스 생성자는 파생 클래스의 선언에 나열된 순서대로 호출됩니다.

C++

```
#include <iostream>
using namespace std;

class BaseClass1 {
public:
    BaseClass1() { cout << "BaseClass1 ctor\n"; }
};

class BaseClass2 {
public:
    BaseClass2() { cout << "BaseClass2 ctor\n"; }
};

class BaseClass3 {
public:
    BaseClass3() { cout << "BaseClass3 ctor\n"; }
};

class DerivedClass : public BaseClass1,
                     public BaseClass2,
                     public BaseClass3
{
public:
    DerivedClass() { cout << "DerivedClass ctor\n"; }
};

int main() {
    DerivedClass dc;
}
```

다음과 같이 출력됩니다.

Output

```
BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor
```

위임 생성자

위임된 생성자는 동일한 클래스에서 다른 생성자를 호출하여 초기화 작업 중 일부를 수행합니다. 이 기능은 모두 비슷한 작업을 수행해야 하는 여러 생성자가 있는 경우에 유용합니다. 한 생성자에 기본 논리를 작성하고 다른 생성자에서 호출할 수 있습니다. 다음 간단한 예제에서 Box(int)는 해당 작업을 Box(int,int,int,int)에 위임합니다.

C++

```
class Box {
public:
```

```

// Default constructor
Box() {}

// Initialize a Box with equal dimensions (i.e. a cube)
Box(int i) : Box(i, i, i) // delegating constructor
{}

// Initialize a Box with custom dimensions
Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{}
//... rest of class as before
};

```

생성자에 의해 만들어지는 개체는 생성자가 완료되는 즉시 완전하게 초기화됩니다. 자세한 내용은 [생성자 위임을 참조하세요](#).

상속 생성자(C++11)

파생 클래스는 다음 예제와 같이 선언을 사용하여 `using` 직접 기본 클래스에서 생성자를 상속할 수 있습니다.

C++

```

#include <iostream>
using namespace std;

class Base
{
public:
    Base() { cout << "Base()" << endl; }
    Base(const Base& other) { cout << "Base(Base&)" << endl; }
    explicit Base(int i) : num(i) { cout << "Base(int)" << endl; }
    explicit Base(char c) : letter(c) { cout << "Base(char)" << endl; }

private:
    int num;
    char letter;
};

class Derived : Base
{
public:
    // Inherit all constructors from Base
    using Base::Base;

private:
    // Can't initialize newMember from Base constructors.
    int newMember{ 0 };
};

```

```

int main()
{
    cout << "Derived d1(5) calls: ";
    Derived d1(5);
    cout << "Derived d1('c') calls: ";
    Derived d2('c');
    cout << "Derived d3 = d2 calls: " ;
    Derived d3 = d2;
    cout << "Derived d4 calls: ";
    Derived d4;
}

/* Output:
Derived d1(5) calls: Base(int)
Derived d1('c') calls: Base(char)
Derived d3 = d2 calls: Base(Base&)
Derived d4 calls: Base()*/

```

Visual Studio 2017 이상: 모드 이상의 문 `/std:c++17` 은 `using` 파생 클래스의 생성자와 동일한 시그니처가 있는 생성자를 제외하고 기본 클래스의 모든 생성자를 범위로 가져옵니다. 일반적으로 파생 클래스가 새 데이터 멤버 또는 생성자를 선언하지 않을 때 상속 생성자를 사용하는 것이 가장 좋습니다.

클래스 템플릿은 해당 형식이 기본 클래스를 지정하는 경우 형식 인수에서 모든 생성자를 상속할 수 있습니다.

C++

```

template< typename T >
class Derived : T {
    using T::T; // declare the constructors from T
    // ...
};

```

이러한 기본 클래스에 동일한 시그니처가 있는 생성자가 있는 경우 파생 클래스는 여러 기본 클래스에서 상속할 수 없습니다.

생성자 및 복합 클래스

클래스 형식 멤버를 포함하는 클래스를 **복합 클래스**라고 합니다. 복합 클래스의 클래스 형식 멤버를 만들 때 생성자가 클래스의 자체 생성자보다 먼저 호출됩니다. 포함된 클래스에 기본 생성자가 없는 경우 복합 클래스의 생성자에서 초기화 목록을 사용해야 합니다. 이전 `StorageBox` 예제에서 `m_label` 멤버 변수의 형식을 새 `Label` 클래스로 변경할 경우 기본 클래스 생성자를 호출하고 `m_label` 생성자에서 `StorageBox` 변수를 초기화해야 합니다.

C++

```
class Label {
public:
    Label(const string& name, const string& address) { m_name = name;
m_address = address; }
    string m_name;
    string m_address;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, Label label)
        : Box(width, length, height), m_label(label){}
private:
    Label m_label;
};

int main(){
// passing a named Label
Label label1{ "some_name", "some_address" };
StorageBox sb1(1, 2, 3, label1);

// passing a temporary label
StorageBox sb2(3, 4, 5, Label{ "another name", "another address" });

// passing a temporary label as an initializer list
StorageBox sb3(1, 2, 3, {"myname", "myaddress"});
}
```

섹션 내용

- 복사 생성자 및 복사 할당 연산자
- 이동 생성자 및 이동 할당 연산자
- 위임 생성자

추가 정보

[클래스 및 구조체](#)

복사 생성자 및 복사 대입 연산자(C++)

아티클 • 2024. 07. 08.

① 참고

C++11부터 언어에서 복사 할당 및 이동 할당이라는 두 가지 할당이 지원됩니다. 이 문서에서 별도로 명시하지 않는 한 "할당"은 복사 할당을 의미합니다. 이동 할당에 대한 자세한 내용은 [이동 생성자 및 이동 대입 연산자\(C++\)](#)를 참조하세요.

할당 작업과 초기화 작업은 모두 개체를 복사합니다.

- **할당**: 한 개체의 값이 다른 개체에 할당하면 첫 번째 개체를 두 번째 개체에 복사합니다. 따라서 이 코드는 `b`의 값을 `a`에 복사합니다.

C++

```
Point a, b;  
...  
a = b;
```

- **초기화**: 새 개체를 선언할 때, 값으로 함수 인수를 전달하거나 함수에서 값으로 반환할 때 초기화가 발생합니다.

클래스 형식의 개체에 대해 "복사"의 의미를 정의할 수 있습니다. 예를 들어 다음 코드를 고려합니다.

C++

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

앞의 코드는 "FILE1.DAT의 내용을 FILE2.DAT에 복사"를 의미하거나, "FILE2.DAT를 무시하고 `b`를 FILE1.DAT의 두 번째 핸들로 만들기"를 의미할 수도 있습니다. 다음과 같이 각 클래스에 적절한 복사 의미 체계를 연결해야 합니다.

- 클래스 형식에 대한 참조를 반환하고 `const` 참조로 전달된 하나의 매개 변수를 사용하는 대입 연산자 `operator=`를 사용합니다(예: `ClassName& operator=(const ClassName& x);`).
- 복사 생성자를 사용합니다.

복사 생성자를 선언하지 않으면 컴파일러는 자동으로 멤버 단위 복사 생성자를 생성합니다. 마찬가지로 복사 대입 연산자를 선언하지 않으면 컴파일러는 멤버별 복사 대입 연산자를 생성합니다. 복사 생성자를 선언해도 컴파일러에서 생성된 복사 대입 연산자가 억제되지 않으며 그 반대의 경우도 마찬가지입니다. 둘 중 하나를 구현하는 경우 다른 것도 구현하는 것이 좋습니다. 두 가지를 모두 구현하면 코드의 의미가 명확해집니다.

복사 생성자는 `ClassName&` 형식의 인수를 사용합니다. 여기서 `ClassName`은 클래스 이름입니다. 예시:

C++

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& );           // Declare copy constructor.
    Window& operator=( const Window& x ); // Declare copy assignment.
    // ...
};

int main()
{
}
```

① 참고

가능할 때마다 복사 생성자의 인수 `const ClassName&`의 형식을 만듭니다. 이렇게 하면 복사 생성자가 복사된 개체를 실수로 변경하는 것을 방지할 수 있습니다. 또한 `const` 개체에서 복사할 수도 있습니다.

컴파일러에서 생성된 복사 생성자

사용자 정의 복사 생성자와 같이 컴파일러에서 생성된 복사 생성자에는 "class-name에 대한 형식 참조"의 단일 인수가 있습니다. 한 가지 예외는 형식 `const class-name&`의 단일 인수로 사용할 때 모든 기본 클래스와 멤버 클래스에서 복사 생성자를 선언하는 경우입니다. 이런 경우 컴파일러에서 생성된 복사 생성자의 인수는 `const`이기도 합니다.

복사 생성자의 인수 형식이 `const`가 아닌 경우 `const` 개체를 복사하여 초기화하면 오류가 발생합니다. 인수가 `const`이면 `const`가 아닌 개체를 복사하여 초기화할 수 있지만 반대의 경우는 불가능합니다.

컴파일러에서 생성된 대입 연산자는 `const`에 대해 동일한 패턴을 따릅니다. 모든 기본 클래스와 멤버 클래스의 대입 연산자가 `const ClassName&` 형식의 인수를 취하지 않는 한,

이는 `ClassName&` 형식의 단일 인수를 사용합니다. 이 경우 클래스에 대해 생성된 대입 연산자는 `const` 인수를 사용합니다.

① 참고

가상 기본 클래스가 복사 생성자에 의해 초기화되면(컴파일러 생성이든 사용자 정의든) 해당 클래스는 생성되는 시점에 한 번만 초기화됩니다.

의미는 복사 생성자의 의미와 유사합니다. 인수 형식이 `const` 가 아닌 경우 `const` 객체의 할당은 오류를 생성합니다. `const` 값이 `const` 가 아닌 값에 할당될 수 있지만 반대의 경우는 불가능합니다.

오버로드된 대입 연산자에 대한 자세한 내용은 [할당을 참조하세요](#).

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

이동 생성자 및 이동 할당 연산자(C++)

아티클 • 2023. 04. 03.

이 항목에서는 C++ 클래스에 대한 이동 생성자 및 이동 할당 연산자를 작성하는 방법을 설명합니다. 이동 생성자를 사용하면 rvalue 개체가 소유한 리소스를 복사하지 않고 lvalue로 이동할 수 있습니다. 이동 의미 체계에 대한 자세한 내용은 [Rvalue 참조 선언자: &&](#)를 참조하세요.

이 항목은 메모리 버퍼를 관리하는 다음 C++ 클래스 `MemoryBlock`을 기반으로 합니다.

C++

```
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

    // Simple constructor that initializes the resource.
    explicit MemoryBlock(size_t length)
        : _length(length)
        , _data(new int[length])
    {
        std::cout << "In MemoryBlock(size_t). length = "
              << _length << "." << std::endl;
    }

    // Destructor.
    ~MemoryBlock()
    {
        std::cout << "In ~MemoryBlock(). length = "
              << _length << ".";

        if (_data != nullptr)
        {
            std::cout << " Deleting resource.";
            // Delete the resource.
            delete[] _data;
        }

        std::cout << std::endl;
    }

    // Copy constructor.
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length)
        , _data(new int[other._length])
```

```

{
    std::cout << "In MemoryBlock(const MemoryBlock&). length = "
        << other._length << ". Copying resource." << std::endl;

    std::copy(other._data, other._data + _length, _data);
}

// Copy assignment operator.
MemoryBlock& operator=(const MemoryBlock& other)
{
    std::cout << "In operator=(const MemoryBlock&). length = "
        << other._length << ". Copying resource." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        _length = other._length;
        _data = new int[_length];
        std::copy(other._data, other._data + _length, _data);
    }
    return *this;
}

// Retrieves the length of the data resource.
size_t Length() const
{
    return _length;
}

private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};

```

다음 절차에서는 예제 C++ 클래스에 대한 이동 생성자와 이동 할당 연산자를 작성하는 방법을 설명합니다.

C++ 클래스에 대한 이동 생성자를 만들려면

1. 다음 예제와 같이 클래스 형식에 대한 rvalue 참조를 매개 변수로 사용하는 빈 생성자 메서드를 정의합니다.

C++

```

MemoryBlock(MemoryBlock&& other)
    : _data(nullptr)
    , _length(0)
{
}

```

2. 이동 생성자에서 소스 개체의 클래스 데이터 멤버를 생성될 개체에 할당합니다.

```
C++
```

```
_data = other._data;  
_length = other._length;
```

3. 소스 개체의 데이터 멤버를 기본 값에 할당합니다. 이에 따라 소멸자가 리소스(예: 메모리)를 여러 번 해제하는 것이 방지됩니다.

```
C++
```

```
other._data = nullptr;  
other._length = 0;
```

C++ 클래스에 대한 이동 할당 연산자를 만들려면

1. 다음 예제와 같이 클래스 형식에 대한 rvalue 참조를 매개 변수로 사용하고 클래스 형식에 대한 참조를 반환하는 빈 할당 연산자를 정의합니다.

```
C++
```

```
MemoryBlock& operator=(MemoryBlock&& other)  
{  
}
```

2. 이동 할당 연산자에서 개체를 자체에 할당하려는 경우 작업을 수행하지 않는 조건문을 추가합니다.

```
C++
```

```
if (this != &other)  
{  
}
```

3. 조건문에서 할당될 개체로부터 모든 리소스(예: 메모리)를 해제합니다.

다음 예제에서는 할당될 개체로부터 `_data` 멤버를 해제합니다.

```
C++
```

```
// Free the existing resource.  
delete[] _data;
```

첫 번째 절차의 2-3단계에 따라 소스 개체의 데이터 멤버를 생성할 개체로 전송합니다.

```
C++  
  
// Copy the data pointer and its length from the  
// source object.  
_data = other._data;  
_length = other._length;  
  
// Release the data pointer from the source object so that  
// the destructor does not free the memory multiple times.  
other._data = nullptr;  
other._length = 0;
```

4. 다음 예제와 같이 현재 개체에 대한 참조를 반환합니다.

```
C++  
  
return *this;
```

예: 전체 이동 생성자 및 할당 연산자

다음 예제에서는 `MemoryBlock` 클래스에 대한 완전한 이동 생성자와 이동 할당 연산자를 보여 줍니다.

```
C++  
  
// Move constructor.  
MemoryBlock(MemoryBlock&& other) noexcept  
: _data(nullptr)  
, _length(0)  
{  
    std::cout << "In MemoryBlock(MemoryBlock&&). length = "  
        << other._length << ". Moving resource." << std::endl;  
  
    // Copy the data pointer and its length from the  
    // source object.  
    _data = other._data;  
    _length = other._length;  
  
    // Release the data pointer from the source object so that  
    // the destructor does not free the memory multiple times.  
    other._data = nullptr;  
    other._length = 0;  
}  
  
// Move assignment operator.
```

```

MemoryBlock& operator=(MemoryBlock&& other) noexcept
{
    std::cout << "In operator=(MemoryBlock&&). length = "
        << other._length << "." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        // Copy the data pointer and its length from the
        // source object.
        _data = other._data;
        _length = other._length;

        // Release the data pointer from the source object so that
        // the destructor does not free the memory multiple times.
        other._data = nullptr;
        other._length = 0;
    }
    return *this;
}

```

예제 이동 의미 체계를 사용하여 성능 향상

다음 예제에서는 이동 의미 체계를 통해 애플리케이션의 성능을 향상시키는 방법을 보여줍니다. 이 예제에서는 벡터 개체에 두 요소를 추가한 다음 기존의 두 요소 사이에 새 요소를 삽입합니다. 클래스는 `vector` 이동 의미 체계를 사용하여 벡터의 요소를 복사하는 대신 이동하여 삽입 작업을 효율적으로 수행합니다.

C++

```

// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
    // Create a vector object and add a few elements to it.
    vector<MemoryBlock> v;
    v.push_back(MemoryBlock(25));
    v.push_back(MemoryBlock(75));

    // Insert a new element into the second position of the vector.
    v.insert(v.begin() + 1, MemoryBlock(50));
}

```

이 예제는 다음과 같은 출력을 생성합니다.

Output

```
In MemoryBlock(size_t). length = 25.  
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.  
In ~MemoryBlock(). length = 0.  
In MemoryBlock(size_t). length = 75.  
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.  
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.  
In ~MemoryBlock(). length = 0.  
In ~MemoryBlock(). length = 0.  
In MemoryBlock(size_t). length = 50.  
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.  
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.  
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.  
In ~MemoryBlock(). length = 0.  
In ~MemoryBlock(). length = 0.  
In ~MemoryBlock(). length = 0.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 75. Deleting resource.
```

Visual Studio 2010 이전에는 다음 출력이 생성되었습니다.

Output

```
In MemoryBlock(size_t). length = 25.  
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In MemoryBlock(size_t). length = 75.  
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.  
In ~MemoryBlock(). length = 75. Deleting resource.  
In MemoryBlock(size_t). length = 50.  
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.  
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.  
In operator=(const MemoryBlock&). length = 75. Copying resource.  
In operator=(const MemoryBlock&). length = 50. Copying resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 75. Deleting resource.
```

이동 의미 체계를 사용하는 이 예제의 버전은 이동 의미 체계를 사용하지 않는 버전보다 적은 복사, 메모리 할당 및 메모리 할당 취소 작업을 수행하기 때문에 효율적입니다.

강력한 프로그래밍

리소스 누수를 방지하려면 항상 이동 할당 연산자에서 메모리, 파일 핸들 및 소켓과 같은 리소스를 해제합니다.

리소스의 복구할 수 없는 소멸을 방지하려면 이동 할당 연산자에서 자체 할당을 적절하게 처리합니다.

사용자 클래스에 이동 생성자와 이동 할당 연산자를 둘 다 제공하는 경우 이동 할당 연산자를 호출하는 이동 생성자를 작성하여 중복 코드를 제거할 수 있습니다. 다음 예제에서는 이동 할당 연산자를 호출하는 이동 생성자의 수정된 버전을 보여 줍니다.

C++

```
// Move constructor.  
MemoryBlock(MemoryBlock&& other) noexcept  
    : _data(nullptr)  
    , _length(0)  
{  
    *this = std::move(other);  
}
```

`std::move` 함수는 lvalue를 rvalue `other`로 변환합니다.

추가 정보

[Rvalue 참조 선언자: &&](#)

[std::move](#)

위임 생성자

아티클 • 2024. 11. 21.

많은 클래스에는 비슷한 작업을 수행하는 여러 생성자가 있습니다. 예를 들어 매개 변수의 유효성을 검사합니다.

C++

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

모든 유효성 검사를 수행하는 함수를 추가하여 반복 코드를 줄일 수 있지만 `class_c`, 한 생성자가 일부 작업을 다른 생성자에게 위임할 수 있는지를 보다 쉽게 이해하고 유지 관리할 수 있습니다. 위임 생성자를 추가하려면 다음 구문을 사용합니다

```
.) : constructor (. . .).
```

C++

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
};
```

```

        class_c(int my_max, int my_min, int my_middle) : class_c (my_max,
my_min){
            middle = my_middle < max && my_middle > min ? my_middle : 5;
}
};

int main() {

    class_c c1{ 1, 3, 2 };
}

```

이전 예제를 단계별로 진행하면서 생성자가 먼저 생성자를 `class_c(int, int, int)` `class_c(int, int)` 호출하고, 다시 호출 `class_c(int)` 합니다. 각 생성자는 다른 생성자가 수행하지 않는 작업만 수행합니다.

호출되는 첫 번째 생성자는 해당 시점에서 모든 멤버가 초기화되도록 개체를 초기화합니다. 다음과 같이 다른 생성자에 위임하는 생성자에서는 멤버 초기화를 수행할 수 없습니다.

C++

```

class class_a {
public:
    class_a() {}
    // member initialization here, no delegate
    class_a(string str) : m_string{ str } {}

    //can't do member initialization here
    // error C3511: a call to a delegating constructor shall be the only
    member-initializer
    class_a(string str, double dbl) : class_a(str) , m_double{ dbl } {}

    // only member assignment
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string;
};

```

다음 예제에서는 비정적 데이터 멤버 이니셜라이저의 사용을 보여 줍니다. 생성자도 지정된 데이터 멤버를 초기화하면 멤버 이니셜라이저가 재정의됩니다.

C++

```

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string{ m_double < 10.0 ? "alpha" : "beta" };
};

```

```
int main() {
    class_a a{ "hello", 2.0 }; //expect a.m_double == 2.0, a.m_string ==
    "hello"
    int y = 4;
}
```

생성자 위임 구문은 생성자 재귀를 실수로 만드는 것을 방지하지 않습니다. Constructor1은 Constructor1을 호출하는 Constructor2를 호출하며 스택 오버플로가 있을 때까지 오류가 throw되지 않습니다. 주기를 피하는 것은 사용자의 책임입니다.

C++

```
class class_f{
public:
    int max;
    int min;

    // don't do this
    class_f() : class_f(6, 3){ }
    class_f(int my_max, int my_min) : class_f() { }
};
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

소멸자 (C++)

아티클 • 2023. 12. 05.

소멸자가 개체가 범위를 벗어나거나 호출에 `delete[]` 의해 명시적으로 제거될 때 자동으로 호출 `delete` 되는 멤버 함수입니다. 소멸자의 이름은 클래스와 같으며 그 앞에는 타일 (~)이 있습니다. 예를 들어 클래스 `String`의 소멸자는 `~String()`으로 선언됩니다.

소멸자를 정의하지 않으면 컴파일러가 기본값을 제공하며 일부 클래스의 경우 이것으로 충분합니다. 클래스가 시스템 리소스에 대한 핸들 또는 클래스 인스턴스가 제거될 때 해제해야 하는 메모리에 대한 포인터와 같이 명시적으로 해제해야 하는 리소스를 기본 때 사용자 지정 소멸자를 정의해야 합니다.

다음 `String` 클래스 선언을 참조하십시오.

C++

```
// spec1_destructors.cpp
#include <string>

class String {
public:
    String( char *ch ); // Declare constructor
    ~String();           // and destructor.
private:
    char      *_text;
    size_t    sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
    sizeOfText = strlen( ch ) + 1;

    // Dynamically allocate the correct amount of memory.
    _text = new char[ sizeOfText ];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
        strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
    // Deallocate the memory that was previously reserved
    // for this string.
    delete[] _text;
}

int main() {
```

```
String str("The piper in the glen...");  
}
```

앞의 예제에서 소멸자는 `String::~String` 연산자를 `delete[]` 사용하여 텍스트 스토리지에 동적으로 할당된 공간을 할당 취소합니다.

소멸자 선언

소멸자는 클래스와 이름이 같지만 물결표(~)가 앞에 붙어 있는 함수입니다.

몇 가지 규칙이 소멸자의 선언에 적용됩니다. 소멸자는 다음과 같습니다.

- 인수를 수락하지 마세요.
- 값(또는 `void`)을 반환하지 마세요.
- 또는 `volatile static`로 선언 `const` 할 수 없습니다. 그러나, 또는 `static`.로 `const volatile` 선언된 개체의 소멸을 위해 호출할 수 있습니다.
- 로 선언 `virtual` 할 수 있습니다. 가상 소멸자를 사용하면 해당 형식을 모르고 개체를 삭제할 수 있습니다. 가상 함수 메커니즘을 사용하여 개체에 대한 올바른 소멸자가 호출됩니다. 소멸자는 추상 클래스에 대한 순수 가상 함수로 선언할 수도 있습니다.

소멸자 사용

다음 이벤트 중 하나가 발생하면 소멸자가 호출됩니다.

- 블록 범위가 있는 로컬(자동) 개체는 범위를 벗어납니다.
- 를 사용하여 할당된 개체의 할당을 취소하는 데 사용합니다 `delete new`. 정의되지 않은 동작의 결과 사용 `delete[]`.
- 를 사용하여 할당된 개체의 할당을 취소하는 데 사용합니다 `delete[] new[]`. 정의되지 않은 동작의 결과 사용 `delete`.
- 임시 개체의 수명이 종료됩니다.
- 프로그램이 종료되고 전역 또는 정적 개체가 존재합니다.
- 소멸자는 소멸자 함수의 정규화된 이름을 사용하여 명시적으로 호출됩니다.

소멸자는 클래스 멤버 함수를 자유롭게 호출하고 클래스 멤버 데이터에 액세스할 수 있습니다.

소멸자 사용에는 두 가지 제한 사항이 있습니다.

- 주소를 사용할 수 없습니다.
- 파생 클래스는 기본 클래스의 소멸자를 상속하지 않습니다.

소멸 순서

개체가 범위에서 벗어나거나 삭제될 때 완전한 소멸에서 발생하는 이벤트 시퀀스는 다음과 같습니다.

1. 클래스의 소멸자가 호출되고 소멸자 함수의 본문이 실행됩니다.
2. 비정적 멤버 개체의 소멸자가 클래스 선언에 나타나는 순서와 반대로 호출됩니다.
이러한 멤버의 생성에 사용되는 선택적 멤버 초기화 목록은 생성 또는 소멸 순서에 영향을 주지 않습니다.
3. 가상이 아닌 기본 클래스에 대한 소멸자는 선언의 역순으로 호출됩니다.
4. 가상 기본 클래스의 소멸자가 선언과 반대 순서로 호출됩니다.

C++

```
// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
    A1 * a = new A3;
    delete a;
    printf("\n");

    B1 * b = new B3;
    delete b;
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}
```

출력

```
A3 dtor
A2 dtor
A1 dtor

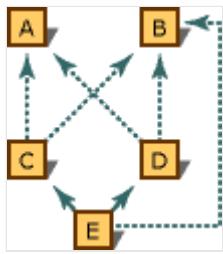
B1 dtor

B3 dtor
```

B2 dtor
B1 dtor

가상 기본 클래스

가상 기본 클래스의 소멸자는 방향이 있는 비순환 그래프(깊이 우선, 왼쪽에서 오른쪽으로, 후위 실행법)에서 표시되는 역순으로 호출됩니다. 다음 그림은 상속 그래프를 보여 줍니다.



다음은 그림에 표시된 클래스에 대한 클래스 정의를 나열합니다.

C++

```
class A {};
class B {};
class C : virtual public A, virtual public B {};
class D : virtual public A, virtual public B {};
class E : public C, public D, virtual public B {};
```

E 형식 개체의 가상 기본 클래스를 파괴하는 순서를 결정하기 위해 컴파일러는 다음 알고리즘을 적용하여 목록을 빌드합니다.

1. 그래프에서 가장 깊은 지점에서 시작하여 왼쪽으로 그래프를 이동합니다(이 경우 E).
2. 모든 노드를 방문할 때까지 왼쪽으로 이동을 수행합니다. 현재 노드의 이름입니다.
3. 이전 노드(아래 및 오른쪽으로)를 다시 방문하여 기억된 노드가 가상 기본 클래스인지 확인합니다.
4. 기억된 노드가 가상 기본 클래스인 경우 목록을 검색하여 이미 입력되었는지를 확인합니다. 가상 기본 클래스가 아닌 경우 무시합니다.
5. 기억된 노드가 목록에 아직 없으면 목록 맨 아래에 추가합니다.
6. 그래프를 위로 이동하고 다음 경로를 따라 오른쪽으로 이동합니다.
7. 2단계로 이동합니다.
8. 마지막 상향 경로가 모두 사용되면 현재 노드의 이름을 참고합니다.
9. 3단계로 이동합니다.
10. 아래쪽 노드가 다시 현재 노드가 될 때까지 이 프로세스를 계속합니다.

따라서 E 클래스의 경우 소멸의 순서는 다음과 같습니다.

1. 가상이 아닌 기본 클래스입니다 E.
2. 가상이 아닌 기본 클래스입니다 D.
3. 가상이 아닌 기본 클래스입니다 C.
4. 가상 기본 클래스 B.
5. 가상 기본 클래스 A.

이 프로세스는 고유 항목의 순서 지정된 목록을 만듭니다. 클래스 이름은 두 번 표시되지 않습니다. 목록이 생성되면 역순으로 안내되고 마지막부터 첫 번째 클래스까지 목록의 각 클래스에 대한 소멸자가 호출됩니다.

생성 또는 소멸 순서는 한 클래스의 생성자 또는 소멸자가 먼저 생성되는 다른 구성 요소에 의존하거나 더 오래 지속되는 경우 주로 중요합니다. 예를 들어 코드 B 가 실행될 때 소멸자가 A 계속 존재하거나 그 반대의 경우도 마찬가지입니다.

나중에 파생된 클래스가 생성과 소멸의 순서를 변경할 수 있는 가장 왼쪽 경로를 변경할 수 있기 때문에 상속 그래프에서 클래스 사이의 이러한 상호 의존성은 본질적으로 위험합니다.

가상이 아닌 기본 클래스

가상이 아닌 기본 클래스에 대한 소멸자는 기본 클래스 이름이 선언되는 역순으로 호출됩니다. 다음과 같은 클래스 선언을 생각해 보세요.

C++

```
class MultInherit : public Base1, public Base2  
...
```

위의 예제에서 Base2의 소멸자가 Base1의 소멸자보다 먼저 호출됩니다.

명시적 소멸자 호출

대부분의 경우 소멸자를 명시적으로 호출할 필요가 없지만 절대 주소에 있는 개체를 정리할 때는 도움이 됩니다. 이러한 개체는 일반적으로 배치 인수를 사용하는 사용자 정의 new 연산자를 사용하여 할당됩니다. 이 연산자는 delete 무료 저장소에서 할당되지 않으므로 이 메모리의 할당을 취소할 수 없습니다(자세한 내용은 새 연산자 및 삭제 연산자 참조). 그러나 소멸자를 호출하면 적절한 정리를 수행할 수 있습니다. s 클래스의 String 개체에 대해 소멸자를 명시적으로 호출하려면 다음 문 중 하나를 사용하세요.

C++

```
s.String::~String();      // non-virtual call  
ps->String::~String();   // non-virtual call  
  
s.~String();              // Virtual call  
ps->~String();           // Virtual call
```

앞에 나온 대로 소멸자를 정의하는 형식에 관계없이 소멸자를 명시적으로 호출하기 위한 표기법을 사용할 수 있습니다. 그러면 형식에 대해 소멸자가 정의되었는지 여부를 몰라도 명시적인 호출이 가능합니다. 소멸자가 정의되지 않은 경우 명시적 호출은 아무 효과가 없습니다.

강력한 프로그래밍

클래스는 리소스를 획득하는 경우 소멸자가 필요하며 리소스를 안전하게 관리하려면 복사 생성자 및 복사 할당을 구현해야 할 수 있습니다.

이러한 특수 함수가 사용자가 정의하지 않은 경우 컴파일러에서 암시적으로 정의됩니다. 암시적으로 생성된 생성자 및 할당 연산자는 단순 멤버 복사를 수행하며 개체가 리소스를 관리하는 경우 거의 틀립니다.

다음 예제에서 암시적으로 생성된 복사 생성자는 포인터 `str1.text str2.text` 를 만들고 동일한 메모리를 참조하며, 반환 `copy_strings()` 할 때 해당 메모리는 정의되지 않은 동작인 두 번 삭제됩니다.

C++

```
void copy_strings()  
{  
    String str1("I have a sense of impending disaster...");  
    String str2 = str1; // str1.text and str2.text now refer to the same  
    object  
} // delete[] _text; deallocates the same memory twice  
// undefined behavior
```

소멸자, 복사 생성자 또는 복사 할당 연산자를 명시적으로 정의하면 이동 생성자 및 이동 할당 연산자의 암시적 정의를 방지할 수 있습니다. 이 경우 일반적으로 이동 작업을 제공하지 못하는 경우 복사 비용이 많이 드는 경우 최적화 기회를 놓칠 수 있습니다.

참고 항목

[복사 생성자 및 복사 할당 연산자](#)
[이동 생성자 및 이동 할당 연산자](#)

멤버 함수 개요

아티클 • 2023. 10. 12.

멤버 함수는 정적이거나 비정적입니다. 정적 멤버 함수에는 암시적 `this` 인수가 없기 때문에 정적 멤버 함수의 동작은 다른 멤버 함수와 다릅니다. 비정적 멤버 함수에는 포인터가 있습니다 `this`. 정적이든 비정적이든 클래스 선언 안이나 밖에서 멤버 함수를 정의할 수 있습니다.

멤버 함수가 클래스 선언 안에 정의될 경우 인라인 함수로 처리되며 함수 이름을 클래스 이름으로 정규화할 필요가 없습니다. 클래스 선언 내에 정의된 함수는 이미 인라인 함수로 처리되지만 키워드(keyword) 사용하여 `inline` 코드를 문서화할 수 있습니다.

클래스 선언 안에 함수를 선언하는 예제는 다음과 같습니다.

C++

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    // of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{}
```

멤버 함수의 정의가 클래스 선언 외부에 있는 경우 명시적으로 선언된 경우에만 인라인 함수로 `inline` 처리됩니다. 또한 범위 결정 연산자(`::`)를 사용하여 클래스 이름으로 정의 안의 함수 이름을 정규화해야 합니다.

다음 예제는 `Account` 함수가 클래스 선언 밖에 정의된 점을 제외하고 위의 `Deposit` 클래스 선언과 동일합니다.

C++

```
// overview_of_member_functions2.cpp
class Account
```

```
{  
public:  
    // Declare the member function Deposit but do not define it.  
    double Deposit( double HowMuch );  
private:  
    double balance;  
};  
  
inline double Account::Deposit( double HowMuch )  
{  
    balance += HowMuch;  
    return balance;  
}  
  
int main()  
{  
}
```

① 참고

클래스 선언 안에 또는 별도로 멤버 함수를 정의할 수 있지만 클래스가 정의된 후에는 클래스에 멤버 함수를 추가할 수 없습니다.

멤버 함수가 포함된 클래스는 많은 선언을 포함할 수 있지만 멤버 함수 자체는 프로그램에 정의를 하나만 포함해야 합니다. 정의가 여러 개 있으면 링크 타임에 오류 메시지가 표시됩니다. 클래스에 인라인 함수 정의가 포함된 경우 함수 정의가 동일해야 이 "단일 정의" 규칙을 지킬 수 있습니다.

virtual 지정자

아티클 • 2024. 11. 21.

가상 [키워드는](#) 비정적 클래스 멤버 함수에만 적용할 수 있습니다. 이는 함수에 대한 호출 바인딩이 런타임까지 지연됨을 의미합니다. 자세한 내용은 [Virtual Functions](#)를 참조하세요.

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

override 지정자

아티클 • 2024. 11. 21.

재정의 키워드를 사용하여 기본 클래스에서 가상 함수를 재정의하는 멤버 함수를 지정할 수 있습니다.

구문

```
function-declaration override;
```

설명

재정의는 컨텍스트를 구분하며 멤버 함수 선언 후에 사용되는 경우에만 특별한 의미를 가집니다. 그렇지 않으면 예약된 키워드가 아닙니다.

예시

재정의를 사용하여 코드에서 실수로 상속 동작을 방지할 수 있습니다. 다음 예제에서는 재정의를 사용하지 않고 파생 클래스의 멤버 함수 동작이 의도되지 않았을 수 있는 위치를 보여줍니다. 컴파일러는 이 코드의 오류를 내보내지 않습니다.

C++

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const, so it does
    not
                    // override BaseClass::funcB() const and it is a
new member function

    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a
different
```

```
// parameter type than
BaseClass::funcC(int), so
                           // DerivedClass::funcC(double) is a
new member function
};
```

재정의를 사용하면 컴파일러는 새 멤버 함수를 자동으로 만드는 대신 오류를 생성합니다.

C++

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB()
does not
                           // override BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override; // compiler error:
                           //
DerivedClass::funcC(double) does not
                           // override
BaseClass::funcC(int)

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                           // override the non-virtual BaseClass::funcD()
};
```

함수를 재정의할 수 없고 클래스를 상속할 수 없도록 지정하려면 최종 키워드를 사용합니다.

참고 항목

final 지정자
키워드

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

final 지정자

아티클 • 2024. 11. 21.

최종 키워드를 사용하여 파생 클래스에서 재정의할 수 없는 가상 함수를 지정할 수 있습니다. 상속할 수 없는 클래스를 지정하기 위해 해당 키워드를 사용할 수도 있습니다.

구문

```
function-declaration final;  
class class-name final base-classes
```

설명

final은 컨텍스트를 구분하며 함수 선언 또는 클래스 이름 다음에 사용되는 경우에만 특별한 의미를 가집니다. 그렇지 않으면 예약된 키워드가 아닙니다.

클래스 선언에서 final을 **base-classes** 사용하는 경우 선언의 선택적 부분입니다.

예시

다음 예제에서는 최종 키워드를 사용하여 가상 함수를 재정의할 수 없도록 지정합니다.

C++

```
class BaseClass  
{  
    virtual void func() final;  
};  
  
class DerivedClass: public BaseClass  
{  
    virtual void func(); // compiler error: attempting to  
                        // override a final function  
};
```

멤버 함수를 재정의할 수 있도록 지정하는 방법에 대한 자세한 내용은 재정의 지정자를 참조 [하세요](#).

다음 예제에서는 최종 키워드를 사용하여 클래스를 상속할 수 없도록 지정합니다.

C++

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                    // marked as non-inheritable
{
```

참고 항목

키워드

override 지정자

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

상속(C++)

아티클 • 2024. 11. 21.

이 단원에서는 파생 클래스를 사용하여 확장 가능한 프로그램을 생성하는 방법을 설명합니다.

개요

"상속"이라는 메커니즘을 사용하여 기존 클래스에서 새 클래스를 파생할 수 있습니다(단일 상속에서 시작하는 정보 참조). 파생에 사용되는 클래스를 특정 파생 클래스의 "기본 클래스"라고 합니다. 파생 클래스는 다음 구문을 사용하여 선언됩니다.

C++

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};

class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

클래스에 대한 태그(이름) 뒤에 콜론과 기본 사양 목록이 나타납니다. 그렇게 명명된 기본 클래스는 이전에 선언되어 있어야 합니다. 기본 사양에는 키워드 `public` `protected` `private` 중 하나인 액세스 지정자가 포함될 수 있습니다. 이러한 액세스 지정자는 기본 클래스 이름 앞에 나타나고 해당 기본 클래스에만 적용됩니다. 이러한 지정자는 기본 클래스의 멤버를 사용할 수 있는 파생 클래스의 권한을 제어합니다. 기본 클래스 멤버에 대한 액세스에 대한 자세한 내용은 Member-Access Control을 참조하세요. 액세스 지정자를 생략하면 해당 베이스에 대한 액세스가 고려됩니다 `private`. 기본 사양에는 가상 상속을 나타내는 키워드 `virtual` 가 포함될 수 있습니다. 이 키워드는 액세스 지정자(있는 경우) 앞이나 뒤에 나타날 수 있습니다. 가상 상속을 사용하는 경우 기본 클래스를 가상 기본 클래스라고 합니다.

여러 기본 클래스를 쉼표로 구분하여 지정할 수 있습니다. 단일 기본 클래스를 지정하면 상속 모델은 단일 상속입니다. 둘 이상의 기본 클래스를 지정하면 상속 모델을 다중 상속이라고 합니다.

주제는 다음과 같습니다.

- 단일 상속

- 여러 기본 클래스
- 가상 함수
- 명시적 재정의
- 추상 클래스
- 범위 규칙 요약

`_super` 및 `_interface` 키워드는 이 섹션에 설명되어 있습니다.

참고 항목

C++ 언어 참조

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

가상 함수

아티클 • 2023. 04. 03.

가상 함수는 파생 클래스에서 다시 정의할 멤버 함수입니다. 기본 클래스의 포인터나 참조를 사용하여 파생 클래스 개체를 참조할 때 해당 개체의 가상 함수를 호출하고 파생 클래스의 함수 버전을 실행할 수 있습니다.

가상 함수를 사용하면 함수 호출을 만드는데 사용한 식과 관계없이 개체에 적합한 함수가 호출됩니다.

기본 클래스에 [가상](#)으로 선언된 함수가 포함되어 있고 파생 클래스가 동일한 함수를 정의한다고 가정합니다. 기본 클래스의 포인터나 참조를 사용하여 호출되더라도 파생 클래스의 개체에 대해 파생 클래스의 함수가 호출됩니다. 다음 예제에서는 `PrintBalance` 함수와 파생 클래스 2개의 구현을 제공하는 기본 클래스를 보여 줍니다.

C++

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
    Account( double d ) { _balance = d; }
    virtual ~Account() {}
    virtual double GetBalance() { return _balance; }
    virtual void PrintBalance() { cerr << "Error. Balance not available for
base type." << endl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
    CheckingAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Checking account balance: " <<
GetBalance() << endl; }
};

class SavingsAccount : public Account {
public:
    SavingsAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Savings account balance: " <<
GetBalance(); }
};

int main() {
    // Create objects of type CheckingAccount and SavingsAccount.
```

```

CheckingAccount checking( 100.00 );
SavingsAccount savings( 1000.00 );

// Call PrintBalance using a pointer to Account.
Account *pAccount = &checking;
pAccount->PrintBalance();

// Call PrintBalance using a pointer to Account.
pAccount = &savings;
pAccount->PrintBalance();
}

```

위의 코드에서 `PrintBalance` 개체가 가리키는 경우를 제외하고 `pAccount`에 대한 호출이 동일합니다. `PrintBalance`가 `virtual`이므로 각 개체에 정의된 함수의 버전이 호출됩니다. 파생 클래스 `PrintBalance` 및 `CheckingAccount`의 `SavingsAccount` 함수가 기본 클래스 `Account`의 함수를 "재정의"합니다.

`PrintBalance` 함수의 재정의 구현을 제공하지 않는 클래스가 선언되면 기본 클래스 `Account`의 기본 구현이 사용됩니다.

형식이 같을 경우에만 파생 클래스의 함수가 기본 클래스에서 가상 함수를 재정의합니다. 파생 클래스의 함수는 기본 클래스의 가상 함수와 반환 형식만 같고 인수 목록은 달라야 합니다.

포인터나 참조를 사용하여 함수를 호출할 때 다음 규칙이 적용됩니다.

- 가상 함수 호출은 호출된 개체의 기본 형식에 따라 확인됩니다.
- 비가상 함수 호출은 포인터나 참조의 형식에 따라 확인됩니다.

다음 예제에서는 포인터를 통해 호출된 가상 함수와 비가상 함수의 동작을 보여 줍니다.

C++

```

// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
    virtual void NameOf();    // Virtual function.
    void InvokingClass();    // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
    cout << "Base::NameOf\n";
}

```

```

void Base::InvokingClass() {
    cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
    void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
    cout << "Invoked by Derived\n";
}

int main() {
    // Declare an object of type Derived.
    Derived aDerived;

    // Declare two pointers, one of type Derived * and the other
    // of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base     *pBase   = &aDerived;

    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();        // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}

```

Output

```

Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived

```

`NameOf` 함수가 `Base`의 포인터를 통해 호출되건 `Derived`의 포인터를 통해 호출되건 관계 없이 `Derived`에 대한 함수를 호출합니다. `Derived` 가 가상 함수이고 `NameOf` 및 `pBase`가 모두 `pDerived` 형식의 개체를 가리키므로 `Derived`에 대한 함수를 호출합니다.

가상 함수는 클래스 형식의 개체에 대해서만 호출되므로 전역 또는 정적 함수를로 `virtual` 선언할 수 없습니다.

키워드는 `virtual` 파생 클래스에서 재정의 함수를 선언할 때 사용할 수 있지만 불필요합니다. 가상 함수의 재정의는 항상 가상입니다.

순수 지정자를 사용하여 선언하지 않는 한 기본 클래스의 가상 함수를 정의해야 합니다. 순수 가상 함수에 대한 자세한 내용은 [추상 클래스를 참조하세요](#).

범위 결정 연산자(`::`)를 사용하여 함수 이름을 명시적으로 정규화하여 가상 함수 호출 메커니즘을 억제할 수 있습니다. `Account` 클래스가 포함된 이전의 예제를 생각해 보십시오. 기본 클래스에서 `PrintBalance`를 호출하려면 다음과 같은 코드를 사용하세요.

C++

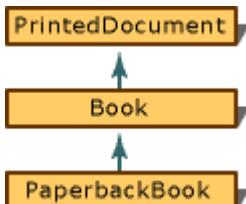
```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

위의 예제에서 `PrintBalance`에 대한 두 호출 모두 가상 함수 호출 메커니즘을 억제합니다.

단일 상속

아티클 • 2023. 04. 03.

일반적인 상속 형식인 "단일 상속"에서 클래스에는 기본 클래스가 하나만 포함됩니다. 다음 그림에 나와 있는 관계를 살펴보십시오.



간단한 단일 상속 그래프

그림에 나와 있는 일반적인 관계에서 구체적인 관계로의 진행 방식을 잘 살펴보십시오. 대부분의 클래스 계층 구조 디자인에서 확인할 수 있는 또 다른 공통적인 특성은, 파생 클래스와 기본 클래스 간에 "일종의" 관계가 있다는 것입니다. 그림에서 `Book`은 일종의 `PrintedDocument`이고 `PaperbackBook`은 일종의 `book`입니다.

그림에서 확인할 수 있는 또 한 가지 사항은, `Book`이 `PrintedDocument`에서 파생 클래스인 동시에 기본 클래스라는 것입니다(`PaperbackBook`이 `Book`에서 파생됨). 이러한 클래스 계층 구조의 기본적인 선언이 다음 예제에 나와 있습니다.

C++

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

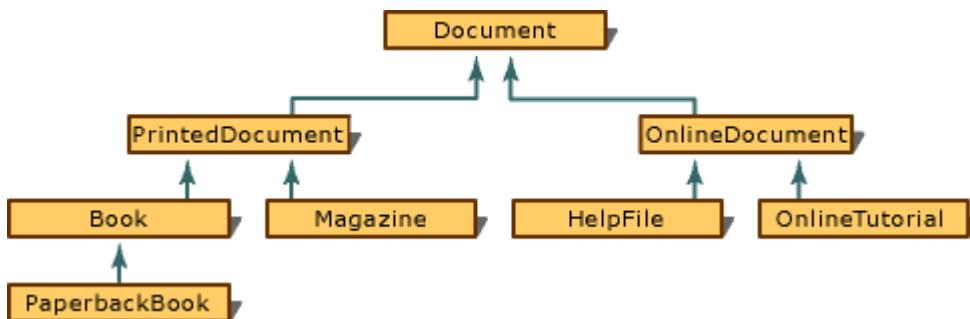
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument`는 `Book`의 "직접 기본" 클래스로 간주되며 `PaperbackBook`의 "간접 기본" 클래스입니다. 이 두 클래스 간의 차이점은 직접 기본 클래스의 경우 클래스 선언의 기본 목록에 표시되지만 간접 기본 클래스는 표시되지 않는다는 것입니다.

각 클래스가 파생되는 기본 클래스는 파생 클래스의 선언 이전에 선언됩니다. 기본 클래스에 대한 전방 참조 선언만을 제공하는 것으로는 부족하며 완전한 선언을 제공해야 합니다.

앞의 예제에서는 액세스 지정자가 `public` 사용됩니다. `public`, `protected` 및 `private` 상속의 의미는 [Member-Access Control](#) 설명되어 있습니다.

다음 그림과 같이 클래스 하나를 여러 특정 클래스의 기본 클래스로 사용할 수 있습니다.



방향이 있는 비순환 그래프 샘플

위의 다이어그램에 나와 있는 "DAG"(방향이 있는 비순환 그래프)에서 일부 클래스는 파생 클래스 두 개 이상의 기본 클래스입니다. 그러나 그 반대의 경우는 성립하지 않습니다. 즉, 지정된 파생 클래스의 직접 기본 클래스는 하나뿐입니다. 그림의 그래프는 "단일 상속" 구조체를 나타냅니다.

① 참고

방향이 있는 비순환 그래프는 단일 상속만을 고유하게 나타내지 않습니다. 즉, 다중 상속 그래프를 나타내는 데 사용되기도 합니다.

상속에서 파생 클래스는 기본 클래스의 멤버와 새로 추가하는 멤버를 포함합니다. 따라서 파생 클래스는 기본 클래스의 멤버를 참조할 수 있습니다(해당 멤버가 파생 클래스에서 다시 정의되는 경우는 제외). 이러한 멤버가 파생 클래스에서 다시 정의된 경우에는 범위 결정 연산자(::)를 사용하여 직접 또는 간접 기본 클래스의 멤버를 참조할 수 있습니다. 다음 예제를 고려해 보세요.

C++

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
    char *Name;    // Document name.
    void PrintNameOf(); // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf() {
    cout << Name << endl;
}

class Book : public Document {
public:
    Book( char *name, long pagecount );
```

```

private:
    long PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
    Name = new char[ strlen( name ) + 1 ];
    strcpy_s( Name, strlen(Name), name );
    PageCount = pagecount;
};

```

`Book`의 생성자(`Book::Book`)는 데이터 멤버 `Name`에 액세스할 수 있습니다. 프로그램에서 `Book` 형식 객체를 만들어 다음과 같이 사용할 수 있습니다.

C++

```

// Create a new object of type Book. This invokes the
// constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...

// Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();

```

위의 예제에 나와 있는 것처럼 클래스 멤버와 상속된 데이터 및 함수는 동일하게 사용됩니다. 클래스 `Book`의 구현에서 `PrintNameOf` 함수를 다시 구현해야 하는 경우 `Document` 클래스에 속하는 함수는 범위 결정(++) 연산자를 통해서만 호출할 수 있습니다.

C++

```

// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
    char *Name;           // Document name.
    void PrintNameOf() {} // Print name.
};

class Book : public Document {
    Book( char *name, long pagecount );
    void PrintNameOf();
    long PageCount;
};

void Book::PrintNameOf() {
    cout << "Name of book: ";
}

```

```
    Document::PrintNameOf();  
}
```

액세스 가능하며 명확한 기본 클래스가 있으면 파생 클래스에 대한 참조 및 포인터를 암시적으로 해당 기본 클래스에 대한 참조 및 포인터로 변환할 수 있습니다. 다음 코드에서는 포인터를 사용하여 이 개념을 보여 줍니다. 참조에도 동일한 원칙이 적용됩니다.

C++

```
// deriv_SingleInheritance4.cpp  
// compile with: /W3  
struct Document {  
    char *Name;  
    void PrintNameOf() {}  
};  
  
class PaperbackBook : public Document {};  
  
int main() {  
    Document * DocLib[10]; // Library of ten documents.  
    for (int i = 0 ; i < 5 ; i++)  
        DocLib[i] = new Document;  
    for (int i = 5 ; i < 10 ; i++)  
        DocLib[i] = new PaperbackBook;  
}
```

위의 예제에서는 여러 가지 형식이 작성됩니다. 그러나 이러한 형식은 모두 `Document` 클래스에서 파생되므로 `Document *`로의 암시적 변환이 수행됩니다. 따라서 `DocLib`는 여러 종류의 개체를 포함하는 "유형이 다른 목록", 즉 개체의 형식이 서로 다를 수 있는 목록입니다.

`Document` 클래스는 `PrintNameOf` 함수를 포함하므로 도서관에 있는 각 책의 이름을 인쇄할 수 있습니다. 그러나 `Book`의 페이지 수, `HelpFile`의 바이트 수 등 문서 형식과 관련된 일부 정보는 생략될 수 있습니다.

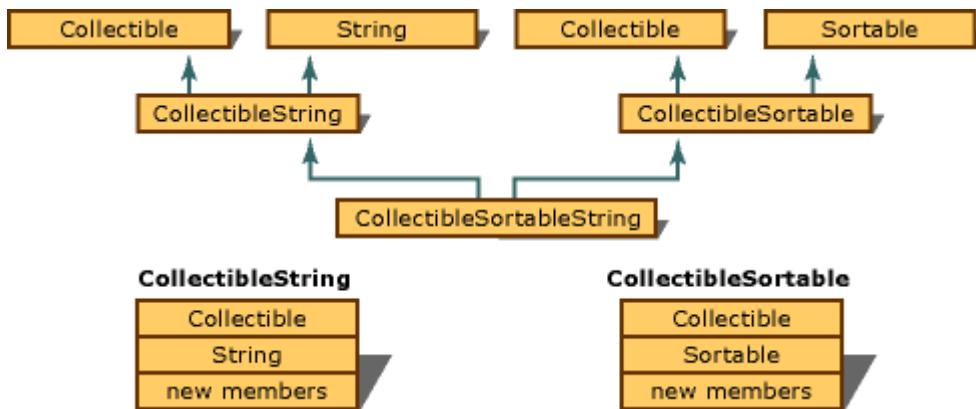
① 참고

기본 클래스를 강제로 적용하여 `PrintNameOf`와 같은 함수를 구현하는 방식은 최적의 디자인이 아닌 경우가 많습니다. **Virtual Functions**는 다른 디자인 대안을 제공합니다.

기본 클래스

아티클 • 2023. 04. 03.

상속 프로세스는 기본 클래스의 멤버와 파생된 클래스에 의해 추가된 모든 새 멤버로 구성된 새로운 파생 클래스를 만듭니다. 다중 상속에서는 동일한 기본 클래스가 둘 이상의 파생 클래스에 속하는 상속 그래프를 생성할 수 있습니다. 다음 그림에서는 이러한 그래프를 보여 줍니다.



단일 기본 클래스의 여러 인스턴스

이 그림에는 `CollectibleString` 및 `CollectibleSortable`의 구성 요소가 설명되어 있습니다. 그러나 기본 클래스인 `Collectible`은 `CollectibleSortableString` 경로 및 `CollectibleString` 경로를 통해 `CollectibleSortable`에 있습니다. 이 중복성을 없애기 위해 상속 시 이러한 클래스를 가상 기본 클래스로 선언할 수 있습니다.

다중 기본 클래스

아티클 • 2023. 04. 11.

클래스는 둘 이상의 기본 클래스에서 파생될 수 있습니다. 여러 상속 모델(클래스가 둘 이상의 기본 클래스에서 파생되는 경우)에서 기본 클래스는 기본 목록 문법 요소를 사용하여 지정됩니다. 예를 들어 `CollectionOfBook` 및 `Collection`에서 파생된 `Book`에 대한 클래스 선언을 지정할 수 있습니다.

C++

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

기본 클래스가 지정된 순서는 생성자와 소멸자가 호출되는 특정 경우를 제외하고는 중요하지 않습니다. 이러한 경우 기본 클래스가 지정되는 순서는 다음에 영향을 줍니다.

- 생성자가 호출되는 순서입니다. 코드가 `Book`의 `CollectionOfBook` 부분을 의존하여 `Collection` 파트 전에 초기화되는 경우 사양의 순서는 중요합니다. 초기화는 클래스가 기본 목록에 지정된 순서대로 이루어집니다.
- 정리하기 위해 소멸자를 호출하는 순서입니다. 다른 부품이 소멸될 때 클래스의 특정 "부품"이 있어야 하는 경우 순서가 중요합니다. 소멸자는 기본 목록에 지정된 클래스의 역순으로 호출됩니다.

① 참고

기본 클래스의 사양 순서는 클래스의 메모리 레이아웃에 영향을 줍니다. 메모리에 있는 기본 멤버의 순서에 따라 모든 프로그래밍 의사 결정을 마십시오.

기본 목록을 지정할 때 동일한 클래스 이름을 두 번 이상 지정할 수 없습니다. 그러나 클래스가 파생 클래스에 대한 간접 베이스가 두 번 이상 될 수 있습니다.

가상 기본 클래스

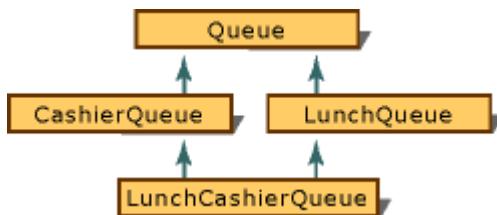
클래스는 한 번 이상 파생 클래스에 대한 간접 기본 클래스일 수 있으므로 C++는 이런 기본 클래스가 작동하는 방식을 최적화하는 방법을 제공합니다. 가상 기본 클래스는 다수의 형식 상속을 사용하는 클래스 계층에 모호성이 발생하지 않도록 공간을 절약하는 방법을 제공합니다.

각 비가상 개체에는 기본 클래스에 정의된 데이터 멤버의 복사본이 있습니다. 이 복제는 공간을 낭비하고 기본 클래스 멤버에 액세스할 때마다 기본 클래스 멤버의 어떤 사본을 원하는지 지정할 것을 요구합니다.

기본 클래스를 가상 기본으로 지정하면, 데이터 멤버가 중복되지 않으면서 한 번 이상 간접 기본으로 작동할 수 있습니다. 해당 데이터 멤버의 단일 복사본은 해당 복사본을 가상 기본 클래스로 사용하는 모든 기본 클래스에서 공유합니다.

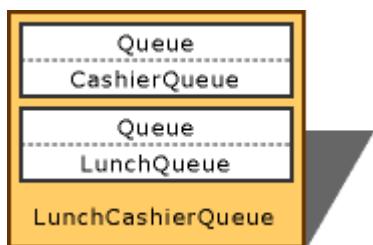
가상 기본 클래스를 선언할 때 파생 클래스 `virtual` 의 기본 목록에 키워드(keyword) 나타납니다.

시뮬레이션된 점심 줄을 보여 주는 다음 그림에서 클래스 계층 구조를 고려합니다.



시뮬레이션된 점심 회선 그래프

그림에서 `Queue`는 `CashierQueue` 및 `LunchQueue`에 대한 기본 클래스입니다. 하지만 두 클래스가 결합되어 `LunchCashierQueue`를 형성할 경우 새 클래스에 형식이 `Queue`인 두 하위 개체가 포함되는데 하나는 `CashierQueue`의 하위 개체이고 하나는 `LunchQueue`의 하위 개체인 문제가 발생할 수 있습니다. 다음 그림에서는 개념적 메모리 레이아웃(실제 메모리 레이아웃이 최적화될 수 있음)을 보여줍니다.

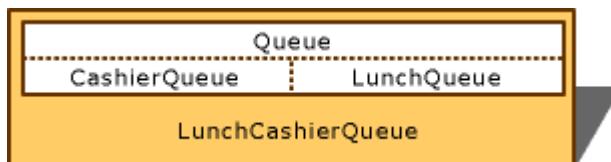


시뮬레이션된 Lunch-Line 개체

개체에는 두 개의 `Queue` 하위 개체가 `LunchCashierQueue` 있습니다. 다음 코드에서는 `Queue`를 가상 기본 클래스로 선언합니다.

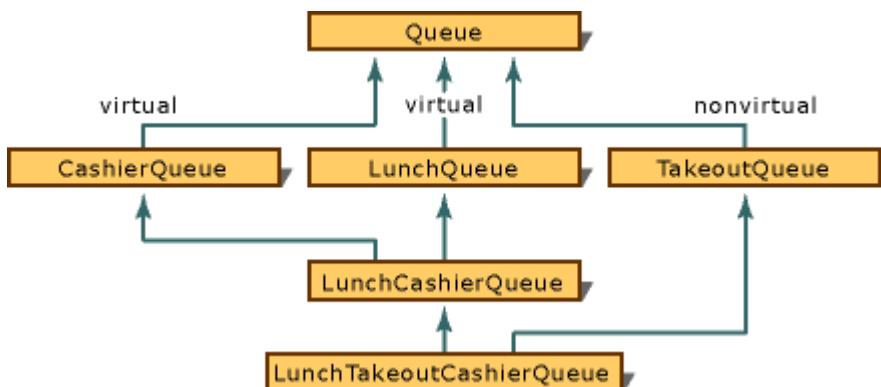
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

`virtual` 키워드(keyword) 하위 `Queue` 개체의 복사본이 하나만 포함되도록 합니다(다음 그림 참조).



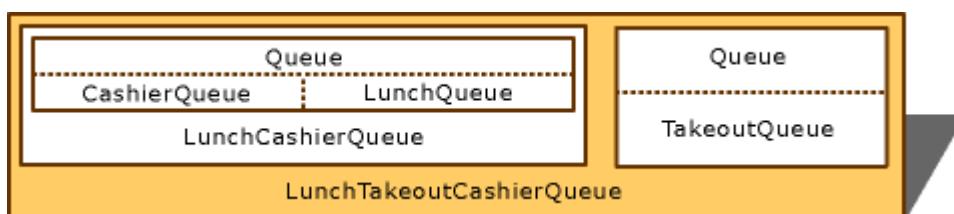
가상 기본 클래스를 사용하여 시뮬레이션된 런치 라인 개체

클래스에는 주어진 형식의 가상 구성 요소와 비가상 구성 요소가 둘 다 있을 수 있습니다. 이 문제는 다음 그림에 설명된 조건에서 발생합니다.



동일한 클래스의 가상 및 비가상 구성 요소

그림에서 `CashierQueue` 및 `LunchQueue`는 `Queue`를 가상 기본 클래스로 사용합니다. 하지만 `TakeoutQueue`는 `Queue`를 가상 기본 클래스가 아니라 기본 클래스로 지정합니다. 따라서, `LunchTakeoutCashierQueue`는 형식이 `Queue`인 두 개의 하위 개체를 가지는데, 하나는 `LunchCashierQueue`를 포함하는 상속 경로의 하위 개체이고 하나는 `TakeoutQueue`를 포함하는 경로의 하위 개체입니다. 이는 다음 그림에 설명되어 있습니다.



가상 및 비가상 상속이 있는 개체 레이아웃

① 참고

가상 상속은 비가상 상속과 비교했을 때 상당한 크기의 혜택을 제공하지만, 추가 처리 오버헤드가 발생할 수 있습니다.

파생 클래스가 가상 기본 클래스에서 상속하는 가상 함수를 재정의하고 파생 기본 클래스의 생성자 또는 소멸자가 가상 기본 클래스에 대한 포인터를 사용하여 함수를 호출하는 경우 컴파일러는 다른 숨겨진 "vtordisp" 필드를 가상 베이스가 있는 클래스에 도입할 수 있습니다. `/vd0` 컴파일러 옵션은 숨겨진 vtordisp 생성자/소멸자 변위 멤버의 추가를 표시하지 않습니다. `/vd1` 컴파일러 옵션인 기본값은 필요한 경우 사용하도록 설정합니다. 모든 클래스 생성자와 소멸자가 가상 함수를 가상으로 호출한다고 확신하는 경우에만 vtordisps를 끕니다.

`/vd` 컴파일러 옵션은 전체 컴파일 모듈에 영향을 줍니다. pragma를 `vtordisp` 사용하여 클래스별로 필드를 표시하지 않은 다음 다시 활성화 `vtordisp` 합니다.

C++

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
\#pragma vtordisp( on )
```

이름 모호성

다중 상속이 발생하면 2개 이상의 경로에서 이름이 상속될 가능성이 있습니다. 이러한 경로를 따라 클래스 멤버 이름이 반드시 고유하지는 않습니다. 이러한 이름 충돌을 "모호성"이라고 합니다.

클래스 멤버를 참조하는 식은 모호하지 않은 참조를 만들어야 합니다. 다음 예제에서는 모호성이 전개되는 방법을 보여 줍니다.

C++

```
// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
```

```

    unsigned a(); // class A also has a member "a"
    int b();     // and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};

```

위의 클래스 선언을 감안할 때 다음과 같은 코드는 에서 또는 에서 **A** 를 참조하는지 여부 **b** 가 불분명하기 **b** 때문에 모호합니다.

C++

```

C *pc = new C;
pc->b();

```

위의 예제를 살펴 보십시오. 이름은 **a** 클래스와 클래스 **A B** 모두의 멤버이므로 컴파일러는 호출할 함수를 지정하는 것을 **a** 식별할 수 없습니다. 함수, 개체, 형식 또는 열거자를 2개 이상 참조할 수 있는 경우 멤버에 대한 액세스가 모호합니다.

컴파일러는 다음 순서대로 테스트하여 모호성을 검색합니다.

1. 앞에서 설명한 대로 이름에 대한 액세스가 모호할 경우 오류 메시지가 생성됩니다.
2. 오버로드된 함수가 명확하지 않으면 해결됩니다.
3. 이름에 대한 액세스가 멤버 액세스 권한을 위반하는 경우 오류 메시지가 생성됩니다. (자세한 내용은 [Member-Access Control](#) 참조하세요.)

상속을 통해 식에 모호성이 생기면 클래스 이름으로 문제의 이름을 정규화하여 수동으로 해결할 수 있습니다. 모호성이 발생하지 않도록 앞의 예제를 적절하게 컴파일하려면 다음 코드를 사용하십시오.

C++

```

C *pc = new C;
pc->B::a();

```

① 참고

c 가 선언되면 **B** 가 **c**의 범위에서 참조될 경우 오류가 발생할 수 있습니다. 그러나 **B**의 범위에 실제로 **c**의 정규화되지 않은 참조가 생길 때까지는 오류가 발생하지 않습니다.

우위

상속 그래프를 통해 둘 이상의 이름(함수, 개체 또는 열거자)에 연결할 수 있습니다. 이러한 경우는 비가상 기본 클래스에서 모호한 것으로 간주됩니다. 이름 중 하나가 다른 이름을 "지배"하지 않는 한 가상 기본 클래스와도 모호합니다.

이름은 두 클래스에 모두 정의되고 한 클래스가 다른 클래스에서 파생된 경우 다른 이름을 지배합니다. 우위를 차지하는 이름은 파생 클래스에 있는 이름입니다. 이 이름은 모호성이 다른 방식으로 발생하지 않은 경우 다음 예제와 같이 사용됩니다.

C++

```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

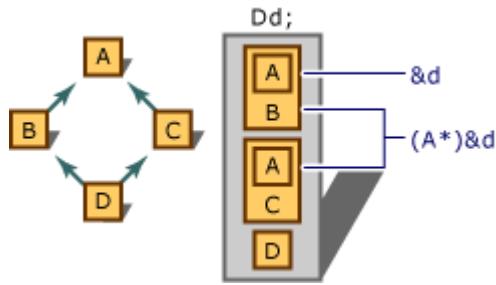
class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

모호한 변환

포인터 또는 참조 형식에서 클래스 형식으로 명시적 및 암시적으로 변환하면 모호성이 발생할 수 있습니다. 아래에 나와 있는 포인터에서 기본 클래스로의 모호한 변환 그림에서는 다음을 보여 줍니다.

- `D` 형식 개체의 선언
- 해당 개체에 address-of 연산자(&)를 적용하는 효과입니다. address-of 연산자는 항상 개체의 기본 주소를 제공합니다.
- 주소 연산자를 사용하여 얻은 포인터를 기본 클래스 형식 `A`로 명시적으로 변환하는 경우의 효과. 개체의 주소를 형식 `A*`으로 강제 변환해도 항상 컴파일러에 선택할

형식 A의 하위 개체에 대한 충분한 정보가 제공되지는 않습니다. 이 경우 두 개의 하위 개체가 있습니다.



포인터에서 기본 클래스로의 모호한 변환

형식 `A*` 으로의 변환(에 대한 포인터 `A`)은 형식의 하위 `A` 개체가 올바른 하위 개체를 식별할 수 없기 때문에 모호합니다. 사용하려는 하위 개체를 다음과 같이 명시적으로 지정하여 모호성을 방지할 수 있습니다.

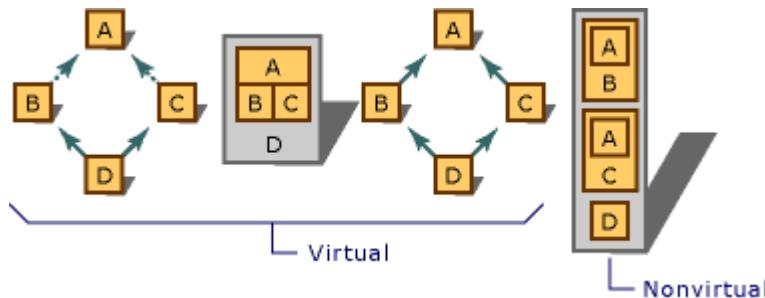
C++

```
(A*)(B*)&d      // Use B subobject.  
(A*)(C*)&d      // Use C subobject.
```

모호성 및 가상 기본 클래스

가상 기본 클래스를 사용하는 경우 다중 상속 경로를 통해 함수, 개체, 형식 및 열거자에 도달할 수 있습니다. 기본 클래스의 instance 하나만 있으므로 이러한 이름에 액세스할 때 모호성이 없습니다.

다음 그림에서는 가상 및 비가상 상속을 사용하여 개체를 구성하는 방법을 보여 줍니다.



가상 및 비가상 파생

이 그림에서 비가상 기본 클래스를 통해 `A` 클래스의 임의 멤버에 액세스하면 모호성이 발생합니다. 컴파일러는 `B`에 연결된 하위 개체를 사용할지, 아니면 `C`에 연결된 하위 개체를 사용할지를 설명하는 정보를 가지고 있지 않습니다. 그러나 가 가상 기본 클래스로 지정된 경우 `A` 어떤 하위 개체에 액세스하고 있는지는 의심의 여지가 없습니다.

추가 정보

상속

명시적 재정의 (C++)

아티클 • 2023. 10. 12.

Microsoft 전용

동일한 가상 함수가 둘 이상의 [인터페이스](#)에서 선언되고 클래스가 이러한 인터페이스에서 파생된 경우 각 가상 함수를 명시적으로 재정의할 수 있습니다.

C++/CLI를 사용하는 관리 코드의 명시적 재정의에 대한 자세한 내용은 명시적 재정의를 참조 [하세요](#).

Microsoft 전용 종료

예시

다음 코드 예제에서는 명시적 재정의를 사용하는 방법을 보여 줍니다.

C++

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

__interface IMyInt2 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
    void IMyInt1::mf1() {
        printf_s("In CMyClass::IMyInt1::mf1()\n");
    }

    void IMyInt1::mf1(int) {
        printf_s("In CMyClass::IMyInt1::mf1(int)\n");
    }

    void IMyInt1::mf2();
    void IMyInt1::mf2(int);
}
```

```

void IMyInt2::mf1() {
    printf_s("In CMyClass::IMyInt2::mf1()\n");
}

void IMyInt2::mf1(int) {
    printf_s("In CMyClass::IMyInt2::mf1(int)\n");
}

void IMyInt2::mf2();
void IMyInt2::mf2(int);
};

void CMyClass::IMyInt1::mf2() {
    printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
    printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
    printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
    printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
    IMyInt1 *pIMyInt1 = new CMyClass();
    IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

    pIMyInt1->mf1();
    pIMyInt1->mf1(1);
    pIMyInt1->mf2();
    pIMyInt1->mf2(2);
    pIMyInt2->mf1();
    pIMyInt2->mf1(3);
    pIMyInt2->mf2();
    pIMyInt2->mf2(4);

    // Cast to a CMyClass pointer so that the destructor gets called
    CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
    delete p;
}

```

Output

```

In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)

```

```
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)
```

참고 항목

[상속](#)

추상 클래스(C++)

아티클 • 2023. 10. 12.

추상 클래스는 보다 구체적인 클래스가 파생될 수 있는 일반 개념의 식 역할을 합니다. 추상 클래스 형식의 개체는 만들 수 없습니다. 그러나 포인터 및 참조를 사용하여 추상 클래스 형식을 사용할 수 있습니다.

하나 이상의 순수 가상 멤버 함수를 선언하여 추상 클래스를 만듭니다. 순수 지정자(= 0) 구문을 사용하여 선언된 가상 함수입니다. 추상 클래스에서 파생된 클래스는 순수 가상 함수를 구현해야 합니다. 이렇게 하지 않으면 파생된 클래스도 추상 클래스가 됩니다.

가상 함수에 [제시된 예제를 생각해 보세요](#). `Account` 클래스의 용도는 일반적인 기능을 제공하는 것이지만, `Account` 형식의 개체는 너무 일반적이어서 유용하게 사용하기가 어렵습니다. 즉 `Account`, 추상 클래스에 적합한 후보입니다.

C++

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
    Account( double d );    // Constructor.
    virtual double GetBalance();    // Obtain balance.
    virtual void PrintBalance() = 0;    // Pure virtual function.
private:
    double _balance;
};
```

이 선언과 이전 선언의 유일한 차이점은 `PrintBalance` 가 순수 지정자(= 0)를 사용하여 선언되었다는 것입니다.

추상 클래스에 대한 제한

추상 클래스는 다음 용도로 사용할 수 없습니다.

- 변수 또는 멤버 데이터
- 인수 형식
- 함수 반환 형식
- 명시적 변환 형식

추상 클래스의 생성자가 직접 또는 간접적으로 순수 가상 함수를 호출하는 경우 결과는 정의되지 않습니다. 하지만 추상 클래스의 생성자 및 소멸자는 다른 멤버 함수를 호출할 수 있습니다.

정의된 순수 가상 함수

추상 클래스의 순수 가상 함수를 정의하거나 구현할 수 있습니다. 정규화된 구문을 사용하여 이러한 함수를 호출할 수 있습니다.

abstract-class-name::function-name()

정의된 순수 가상 함수는 기본 클래스에 순수 가상 소멸자가 포함된 클래스 계층 구조를 디자인할 때 유용합니다. 이는 개체를 파괴하는 동안 기본 클래스 소멸자가 항상 호출되기 때문입니다. 다음 예제를 참조하세요.

C++

```
// deriv_RestrictionsOnUsingAbstractClasses.cpp
// Declare an abstract base class with a pure virtual destructor.
// It's the simplest possible abstract class.
class base
{
public:
    base() {}
    // To define the virtual destructor outside the class:
    virtual ~base() = 0;
    // Microsoft-specific extension to define it inline:
    // virtual ~base() = 0 {};
};

base::~base() {} // required if not using Microsoft extension

class derived : public base
{
public:
    derived() {}
    ~derived() {}
};

int main()
{
    derived aDerived; // destructor called when it goes out of scope
}
```

이 예제에서는 Microsoft 컴파일러 확장을 사용하여 순수 가상 `~base()`에 인라인 정의를 추가하는 방법을 보여 줍니다. 를 사용하여 `base::~base() {}` 클래스 외부에서 정의할 수도 있습니다.

개체 `aDerived` 가 범위를 벗어나면 클래스 `derived` 의 소멸자가 호출됩니다. 컴파일러는 소멸자 이후 `derived` 클래스 `base` 에 대한 소멸자를 암시적으로 호출하는 코드를 생성합니다. 순수 가상 함수에 대한 빈 구현은 함수 `~base` 에 대해 적어도 일부 구현이 있는지 확인합니다. 이 오류가 없으면 링커는 암시적 호출에 대해 해결되지 않은 외부 기호 오류를 생성합니다.

① 참고

앞의 예제에서 순수 가상 함수인 `base::~base` 는 `derived::~derived`에서 암시적으로 호출됩니다. 정규화된 멤버 함수 이름을 사용하여 순수 가상 함수를 명시적으로 호출할 수도 있습니다. 이러한 함수에는 구현이 있어야 합니다. 또는 호출로 인해 링크 시 오류가 발생합니다.

참고 항목

[상속](#)

범위 규칙의 요약

아티클 • 2023. 04. 03.

이름은 그 범위 내(과부하가 결정되는 지점까지)에서 명확하게 사용해야 합니다. 이름이 함수를 표시할 경우 함수의 매개 변수 형식 및 숫자가 명확해야 합니다. 이름이 명확하게 유지되면 [멤버 액세스](#) 규칙이 적용됩니다.

생성자 이니셜라이저

[생성자 이니셜라이저](#)는 지정된 생성자의 가장 바깥쪽 블록 범위에서 평가됩니다. 따라서 생성자의 매개 변수 이름을 사용할 수 있습니다.

전역 이름

개체, 함수 또는 열거자의 이름은 함수나 클래스 외부에 도입되었거나 전역 단항 범위 연산자(::)가 앞에 붙은 경우와 다음과 같은 이항 연산자와 함께 사용되지 않는 경우 전역입니다.

- 범위 확인(::)
- 개체 및 참조에 대한 멤버 선택(.)
- 포인터에 대한 멤버 선택(>)

정규화된 이름

이진 범위 확인 연산자(::)와 함께 사용되는 이름을 "정규화된 이름"이라고 합니다. 이진 범위 확인 연산자 다음에 지정된 이름은 연산자 왼쪽에 지정된 클래스의 멤버이거나 기본 클래스의 멤버여야 합니다.

member-selection 연산자([또는](#)->) 다음에 지정된 이름은 연산자 왼쪽에 지정된 개체의 클래스 형식 멤버이거나 기본 클래스의 멤버여야 합니다. member-selection 연산자(->)의 오른쪽에 지정된 이름은 왼쪽이 클래스 개체이고 클래스가 다른 클래스 형식에 대한 포인터로 계산되는 오버로드된 멤버 선택 연산자(->)를 정의하는 경우 다른 클래스 형식의 > 개체일 수도 있습니다. (이 프로비전은 [클래스 멤버 액세스](#)에서 자세히 설명합니다.)

컴파일러는 다음과 같은 순서로 이름을 검색하고 해당 이름이 발견되면 중지합니다.

1. 이름이 함수 내에서 사용되면 현재 블록 범위이고, 그렇지 않으면 전역 범위입니다.

2. 가장 바깥쪽 함수 범위(함수 매개 변수 포함)를 포함하여 각각의 바깥쪽 블록 범위를 통한 외부입니다.
3. 이름이 멤버 함수 내에서 사용되는 경우 클래스의 범위에서 이름을 검색합니다.
4. 클래스의 기본 클래스에서 이름을 검색합니다.
5. 바깥쪽 중첩 클래스 범위(있는 경우) 및 해당 기본이 검색됩니다. 가장 바깥쪽의 바깥쪽 클래스 범위를 검색할 때까지 검색을 계속합니다.
6. 전역 범위를 검색합니다.

하지만 이 검색 순서는 다음과 같이 수정할 수 있습니다.

1. 앞에 `::`이 있는 이름은 검색이 전역 범위에서 시작되도록 합니다.
2. , `struct` 및 키워드 앞에 오는 `class` 이름은 컴파일러에서 , `struct` 또는 `union` 이름 만 `class` 검색하도록 `union` 합니다.
3. 범위 확인 연산자(`::`)의 왼쪽 이름은 , `struct namespace` 또는 `union` 이름일 수 `class` 있습니다.

이름이 비정적 멤버를 참조하지만 정적 멤버 함수에 사용된 경우 오류 메시지가 생성됩니다. 마찬가지로 이름이 바깥쪽 클래스의 비정적 멤버를 참조하는 경우 묶은 클래스에 묶는 클래스 `this` 포인터가 없기 때문에 오류 메시지가 생성됩니다.

함수 매개 변수 이름

함수 정의의 함수 매개 변수 이름은 함수의 가장 바깥쪽 블록의 범위에 있는 것으로 간주됩니다. 따라서 이러한 이름은 로컬 이름이며 함수가 종료될 때 범위를 벗어납니다.

함수 선언(프로토타입)의 함수 매개 변수 이름은 선언의 지역 범위에 있으며 선언의 끝에서 범위를 벗어납니다.

앞의 두 단락에서 설명한 대로 기본 매개 변수는 해당 매개 변수가 기본값으로 지정된 매개 변수의 범위에 있습니다. 그러나 이러한 인수는 지역 변수 또는 비정적 클래스 멤버에 액세스할 수 없습니다. 기본 매개 변수는 함수 호출 시 계산되지만 함수 선언의 원래 범위에서 계산됩니다. 따라서 멤버 함수에 대한 기본 매개 변수는 항상 클래스 범위에서 계산됩니다.

추가 정보

[상속](#)

상속 키워드

아티클 • 2024. 07. 12.

Microsoft 전용

```
class class-name  
class __single_inheritance class-name  
class __multiple_inheritance class-name  
class __virtual_inheritance class-name
```

여기서

`class-name`
선언되는 클래스의 이름입니다.

C++를 사용하면 클래스의 정의 이전에 클래스 멤버에 대한 포인터를 선언할 수 있습니다. 예시:

C++

```
class S;  
int S::*p;
```

위 코드에서 `p`은(는) `S` 클래스의 정수 멤버에 대한 포인터로 선언되어 있지만, 이 코드에서 `class S`는 아직 정의되지 않았으며 선언만 되어 있습니다. 컴파일러는 이러한 포인터를 발견하면 포인터의 일반화된 표현을 만들어야 합니다. 표현의 크기는 지정된 상속 모델에 따라 달라집니다. 상속 모델을 컴파일러에 지정하는 방법에는 다음 세 가지가 있습니다.

- 명령줄에서 `/vmg` 스위치 사용
- `pointers_to_members` pragma 사용
- 상속 키워드 `__single_inheritance`, `__multiple_inheritance` 및 `__virtual_inheritance` 사용. 이 방법은 클래스별로 상속 모델을 제어합니다.

① 참고

항상 클래스를 정의한 후 클래스 멤버에 대한 포인터를 선언하면 이러한 옵션을 사용할 필요가 없습니다.

클래스가 정의되기 전에 클래스 멤버에 대한 포인터를 선언하면 결과 실행 파일의 크기와 속도에 부정적인 영향을 미칠 수 있습니다. 클래스에서 사용하는 상속이 복잡할수록 클래스 멤버에 대한 포인터를 표현하는 데 필요한 바이트 수가 증가합니다. 또한 포인터를 해석하는 데 필요한 코드가 커집니다. 단일(또는 없는) 상속이 복잡도가 가장 낮고 가상 상속이 가장 복잡합니다. 클래스가 정의되기 전에 선언한 멤버에 대한 포인터는 항상 가장 크고 복잡한 표현을 사용합니다.

위의 예제를 다음과 같이 변경한 경우

C++

```
class __single_inheritance S;  
int S::*p;
```

지정한 명령줄 옵션이나 pragmas에 관계없이 `class S`의 멤버에 대한 포인터는 가능한 가장 작은 표현을 사용합니다.

① 참고

클래스 멤버 포인터 표현의 동일한 정방향 선언이 해당 클래스의 멤버에 대한 포인터를 선언하는 각 변환 단위에서 발생해야 하며, 이 선언은 멤버에 대한 포인터가 선언되기 전에 발생해야 합니다.

이전 버전과의 호환성을 위해 `_single_inheritance`, `_multiple_inheritance` 및 `_virtual_inheritance`은 `_single_inheritance`, `_multiple_inheritance` 및 `_virtual_inheritance`의 동의어입니다. 단 컴파일러 옵션 `/Za(언어 확장 사용 안 함)`가 지정된 경우는 예외입니다.

Microsoft 전용 종료

참고 항목

키워드

피드백

이 페이지가 도움이 되었나요?

Yes

No

virtual (C++)

아티클 • 2024. 11. 21.

키워드는 `virtual` 가상 함수 또는 가상 기본 클래스를 선언합니다.

구문

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

매개 변수

type-specifiers

가상 멤버 함수의 반환 형식을 지정합니다.

member-function-declarator

멤버 함수를 선언합니다.

access-specifier

기본 클래스 `public` `protected` `private` 또는 에 대한 액세스 수준을 정의합니다. 키워드 앞이나 뒤를 표시할 `virtual` 수 있습니다.

base-class-name

이전에 선언된 클래스 형식을 식별합니다.

설명

자세한 내용은 Virtual Functions[를 참조하세요.](#)

클래스, [프라이빗](#), `public` 및 `protected` 키워드도 참조하세요.

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__super

아티클 • 2023. 10. 12.

Microsoft 전용

재정의하는 함수에 대한 기본 클래스 구현을 호출하고 있음을 명시적으로 나타낼 수 있도록 합니다.

구문

```
__super::member_function();
```

설명

액세스할 수 있는 모든 기본 클래스 메서드가 오버로드 확인 단계에서 고려되고 가장 일치하는 함수가 호출되는 함수입니다.

__super 는 멤버 함수의 본문 내에만 나타날 수 있습니다.

__super using 선언과 함께 사용할 수 없습니다. 자세한 내용은 선언 [사용을 참조하세요](#).

코드를 삽입하는 특성 [이](#) 도입되면 코드에 이름을 알 수 없지만 호출하려는 메서드가 포함된 하나 이상의 기본 클래스가 코드에 포함될 수 있습니다.

예시

C++

```
// deriv_super.cpp
// compile with: /c
struct B1 {
    void mf(int) {}
};

struct B2 {
    void mf(short) {}

    void mf(char) {}
};

struct D : B1, B2 {
```

```
void mf(short) {
    __super::mf(1);    // Calls B1::mf(int)
    __super::mf('s');  // Calls B2::mf(char)
}
};
```

Microsoft 전용 종료

참고 항목

[키워드](#)

__interface

아티클 • 2024. 07. 08.

Microsoft 전용

Microsoft C++ 인터페이스는 다음과 같이 정의할 수 있습니다.

- 0개 이상의 기본 인터페이스에서 상속할 수 있습니다.
- 기본 클래스에서 상속할 수 없습니다.
- 공용 순수 가상 메서드만 포함할 수 있습니다.
- 생성자, 소멸자 또는 연산자를 포함할 수 없습니다.
- 정적 메서드를 포함할 수 없습니다.
- 데이터 멤버를 포함할 수 없습니다. 속성은 허용됩니다.

구문

```
modifier __interface interface-name {interface-definition};
```

설명

C++ [클래스](#) 또는 [구조체](#)가 이러한 규칙을 사용하여 구현될 수 있지만 __interface가 이러한 규칙을 적용합니다.

예를 들어 다음은 샘플 인터페이스 정의입니다.

C++

```
__interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

관리되는 인터페이스에 대한 자세한 내용은 [인터페이스 클래스](#)를 참조하세요.

`CommitX` 및 `get_X` 함수가 순수 가상 함수임을 명시적으로 나타낼 필요가 없습니다. 첫 번째 함수에 대한 동일한 선언은 다음과 같습니다.

C++

```
virtual HRESULT CommitX() = 0;
```

`__interface`는 `novtable` `__declspec` 한정자를 의미합니다.

예시

다음 샘플에서는 인터페이스에서 선언된 속성을 사용하는 방법을 보여 줍니다.

C++

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbases.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
    [ id(0) ] int int_data;
    [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }
}
```

```

void put_int_data(int _i)
{
    m_i = _i;
}

BSTR get_bstr_data()
{
    BSTR bstr = ::SysAllocString(m_bstr);
    return bstr;
}

void put_bstr_data(BSTR bstr)
{
    if (m_bstr)
        ::SysFreeString(m_bstr);
    m_bstr = ::SysAllocString(bstr);
}
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}

```

Output

```

p->int_data = 100
bstr_data = Testing

```

Microsoft 전용 종료

참고 항목

[키워드](#)

[인터페이스 특성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

특수 멤버 함수

아티클 • 2023. 10. 12.

특수 멤버 함수는 컴파일러가 자동으로 생성하는 클래스(또는 구조체) 멤버 함수입니다. 이러한 함수는 [기본 생성자, 소멸자, 복사 생성자 및 복사 할당 연산자, 이동 생성자 및 이동 할당 연산자입니다](#). 클래스가 하나 이상의 특수 멤버 함수를 정의하지 않는 경우 컴파일러는 사용되는 함수를 암시적으로 선언하고 정의할 수 있습니다. 컴파일러에서 생성된 구현을 기본 특수 멤버 함수라고 합니다. 컴파일러는 필요하지 않은 경우 함수를 생성하지 않습니다.

= 기본 키워드(keyword) 사용하여 기본 **특수 멤버 함수를 명시적으로 선언할** 수 있습니다. 이렇게 하면 컴파일러가 함수가 전혀 선언되지 않은 것과 같은 방식으로 필요한 경우에만 함수를 정의합니다.

경우에 따라 컴파일러는 삭제된 특수 멤버 함수를 생성할 수 있습니다. 이 함수는 정의되지 않았으므로 호출할 수 없습니다. 클래스의 다른 속성을 고려할 때 클래스의 특정 특수 멤버 함수에 대한 호출이 의미가 없는 경우에 발생할 수 있습니다. 특수 멤버 함수의 자동 생성을 명시적으로 방지하려면 = delete 키워드(keyword) 사용하여 **삭제된 것으로 선언** 할 수 있습니다.

컴파일러는 다른 생성자를 선언하지 않은 경우에만 인수를 사용하지 않는 생성자인 기본 생성자를 생성합니다. 매개 변수를 사용하는 생성자만 선언한 경우 기본 생성자를 호출하려는 코드로 인해 컴파일러에서 오류 메시지가 생성됩니다. 컴파일러에서 생성된 기본 생성자는 개체의 간단한 멤버별 [기본 초기화를](#) 수행합니다. 기본 초기화는 모든 멤버 변수를 확정되지 않은 상태로 둡니다.

기본 소멸자가 개체의 멤버별 소멸을 수행합니다. 기본 클래스 소멸자가 가상인 경우에만 가상입니다.

기본 복사 및 이동 생성 및 할당 작업은 멤버별 비트 패턴 복사 또는 비정적 데이터 멤버의 이동을 수행합니다. 이동 작업은 소멸자 또는 이동 또는 복사 작업이 선언되지 않은 경우에만 생성됩니다. 기본 복사 생성자는 복사 생성자가 선언되지 않은 경우에만 생성됩니다. 이동 작업이 선언된 경우 암시적으로 삭제됩니다. 기본 복사 할당 연산자는 복사 할당 연산자가 명시적으로 선언되지 않은 경우에만 생성됩니다. 이동 작업이 선언된 경우 암시적으로 삭제됩니다.

참고 항목

[C++ 언어 참조](#)

정적 멤버(C++)

아티클 • 2024. 08. 02.

클래스는 정적 멤버 데이터와 멤버 함수를 포함할 수 있습니다. 데이터 멤버가 선언 `static` 되면 클래스의 모든 개체에 대해 데이터의 복사본 하나만 기본.

정적 데이터 멤버는 지정된 클래스 형식의 개체에 속하지 않습니다. 결과적으로, 정적 데이터 멤버 선언은 정의로 간주되지 않습니다. 데이터 멤버는 클래스 범위에서 선언되지만 정의는 파일 범위에서 수행됩니다. 이러한 정적 멤버에는 외부 링크가 있습니다. 다음 예제는 이러한 과정을 보여 줍니다.

C++

```
// static_data_members.cpp
class BufferedOutput
{
public:
    // Return number of bytes written by any object of this class.
    short BytesWritten()
    {
        return bytecount;
    }

    // Reset the counter.
    static void ResetCount()
    {
        bytecount = 0;
    }

    // Static member declaration.
    static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{}
```

앞의 코드에서 `bytecount` 멤버는 `BufferedOutput` 클래스에서 선언되었지만 클래스 선언 밖에서 정의되어야 합니다.

클래스 형식의 개체를 참조하지 않고 정적 데이터 멤버를 참조할 수 있습니다.

`BufferedOutput` 개체를 사용하여 쓴 바이트 수는 다음과 같이 확인할 수 있습니다.

C++

```
long nBytes = BufferedOutput::bytecount;
```

정적 멤버가 존재하기 위해 클래스 형식의 개체가 있어야 하는 것은 아닙니다. 정적 멤버는 멤버 선택(및 ->) 연산자를 사용하여 액세스할 수도 있습니다. 예시:

C++

```
BufferedOutput Console;  
  
long nBytes = Console.bytecount;
```

앞의 경우에서 개체(Console)에 대한 참조는 평가되지 않습니다. 반환된 값은 정적 개체 bytecount의 값입니다.

정적 데이터 멤버에는 클래스 멤버 액세스 규칙이 적용되므로 정적 데이터 멤버에 대한 전용 액세스는 클래스 멤버 함수 및 friend에만 허용됩니다. 이러한 규칙은 멤버 액세스 제어에 [설명되어 있습니다](#). 예외는 정적 데이터 멤버가 해당 액세스 제한에 관계없이 파일 범위에서 정의되어야 한다는 것입니다. 데이터 멤버를 명시적으로 초기화하는 경우 이니셜라이저에 정의를 제공해야 합니다.

정적 멤버의 형식은 클래스 이름으로 한정되지 않습니다. 따라서 형식
BufferedOutput::bytecount 은 .입니다 [long](#).

참고 항목

[클래스 및 구조체](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

사용자 정의 형식 변환(C++)

아티클 • 2024. 11. 21.

변환은 다른 형식의 값에서 일부 형식의 새 값을 생성합니다. 표준 변환은 C++ 언어로 기본 제공되고 기본 제공 형식을 지원하며 사용자 정의 형식 간 변환을 수행하는 사용자 정의 변환을 만들 수 있습니다.

표준 변환은 기본 제공 형식 간의 변환, 상속별로 관련된 형식에 대한 포인터 또는 참조 간의 변환, void 포인터와의 양방향 변환, null 포인터로의 변환을 수행합니다. 자세한 내용은 표준 변환을 참조 [하세요](#). 사용자 정의 변환은 사용자 정의 형식 간의 변환이나 사용자 정의 형식과 기본 제공 형식 간의 변환을 수행합니다. 변환 생성자 [또는](#) 변환 함수로 구현할 수 있습니다.

변환은 명시적으로(프로그래머가 형식을 다른 형식으로 변환하기 위해 호출하는 경우 캐스트 또는 직접 초기화에서) 수행하거나 암시적으로(언어나 프로그램에서 프로그래머가 지정한 형식 이외의 다른 형식에 대해 호출하는 경우) 수행할 수 있습니다.

다음과 같은 경우에 암시적 변환이 시도됩니다.

- 함수에 제공된 인수의 형식이 일치하는 매개 변수의 형식과 다릅니다.
- 함수에서 반환된 값의 형식이 함수 반환 형식과 다릅니다.
- 이니셜라이저 식의 형식이 초기화되는 개체의 형식과 다릅니다.
- 조건 문, 루프 구문 또는 스위치를 제어하는 식의 결과 형식이 이러한 제어에 필요한 결과 형식이 아닙니다.
- 연산자에 제공된 피연산자의 형식이 일치하는 피연산자 매개 변수의 형식과 다릅니다. 기본 제공 연산자의 경우 양쪽 피연산자는 같은 형식이어야 하며 양쪽 피연산자를 나타낼 수 있는 공용 형식으로 변환됩니다. 자세한 내용은 표준 변환을 참조 [하세요](#). 사용자 정의 연산자의 경우 각 피연산자는 일치하는 피연산자 매개 변수와 같은 형식이어야 합니다.

한 개의 표준 변환이 암시적 변환을 완료할 수 없는 경우 컴파일러는 사용자 정의 변환을 사용하여(필요에 따라 표준 변환을 추가적으로 수행하여) 암시적 변환을 완료할 수 있습니다.

변환 사이트에 동일한 변환을 수행하는 사용자 정의 변환이 둘 이상 있는 경우에는 변환이 모호합니다. 이러한 모호성은 컴파일러가 어떤 변환을 선택해야 할지 결정할 수 없으므로 오류입니다. 하지만 사용 가능한 변환 집합은 소스 코드의 여러 위치에서 다르게 사용될 수 있으므로(예: 소스 파일에 포함된 헤더 파일에 따라) 동일한 변환을 수행하는 여러 방법을 정의하는 경우라면 이러한 모호성은 오류는 아닙니다. 변환 사이트에 사용 가

능한 변환이 하나만 있다면 모호성이 없습니다. 모호한 변환은 여러 가지 경우에 발생될 수 있지만 가장 일반적인 경우는 다음과 같습니다.

- 다중 상속. 변환이 둘 이상의 기본 클래스에서 정의됩니다.
- 모호한 함수 호출. 변환이 대상 형식의 변환 생성자 및 소스 형식의 변환 함수로 정의됩니다. 자세한 내용은 변환 함수를 참조 [하세요](#).

일반적으로, 포함된 형식의 이름을 더욱 완벽하게 한정하거나 명시적 캐스트를 수행하여 의도를 분명하게 밝힘으로써 모호성을 해결할 수 있습니다.

변환 생성자와 변환 함수는 멤버 액세스 제어 규칙을 따르지만 변환 액세스 가능성은 모호하지 않은 변환을 확인할 수 있는 경우에만 고려됩니다. 즉, 변환은 경쟁 변환의 액세스 수준으로 인해 사용되지 못하게 되더라도 모호해질 수 있습니다. 멤버 접근성에 대한 자세한 내용은 멤버 액세스 제어를 참조 [하세요](#).

explicit 키워드 및 암시적 변환 문제

기본적으로, 사용자 정의 변환을 만들면 컴파일러가 이를 사용하여 암시적 변환을 수행할 수 있습니다. 이러한 방식을 원할 수도 있지만, 때로는 암시적 변환을 만드는 컴파일러를 안내하는 단순 규칙으로 인해 개발자가 의도하지 않은 코드가 컴파일러에 적용되는 결과가 발생할 수 있습니다.

문제를 일으킬 수 있는 암시적 변환의 잘 알려진 한 가지 예는 .로 변환하는 것입니다 `bool`. 부울 컨텍스트에서 사용할 수 있는 클래스 형식을 만들려는 이유는 여러 가지가 있습니다. 예를 들어 문이나 루프를 제어 `if` 하는 데 사용할 수 있지만 컴파일러가 사용자 정의 변환을 기본 제공 형식으로 수행하는 경우 컴파일러는 나중에 추가 표준 변환을 적용할 수 있습니다. 이 추가 표준 변환의 목적은 승격과 `short` 같은 항목을 허용하기 `int` 위한 것이지만, 클래스 형식을 의도하지 않은 정수 컨텍스트에서 `bool int` 사용할 수 있도록 하는 등 덜 명백한 변환을 위한 문도 엽니다. 이 특정 문제는 안전 부울 문제로 알려져 있습니다. 이러한 종류의 문제는 키워드가 `explicit` 도움이 될 수 있는 곳입니다.

이 키워드는 `explicit` 지정된 변환을 사용하여 암시적 변환을 수행할 수 없다는 것을 컴파일러에 알릴 수 있습니다. 키워드가 도입되기 전에 `explicit` 암시적 변환의 구문적 편의를 원하는 경우 암시적 변환이 때때로 생성되는 의도하지 않은 결과를 수락하거나 덜 편리한 명명된 변환 함수를 해결 방법으로 사용해야 했습니다. 이제 키워드를 `explicit` 사용하여 명시적 캐스트 또는 직접 초기화를 수행하는 데만 사용할 수 있는 편리한 변환을 만들 수 있으며, 이로 인해 안전 부울 문제가 예로 들 수 있는 문제가 발생하지 않습니다.

C `explicit` ++98 이후의 변환 생성자와 C++11 이후의 변환 함수에 키워드를 적용할 수 있습니다. 다음 섹션에는 키워드를 사용하는 `explicit` 방법에 대한 자세한 정보가 포함

되어 있습니다.

변환 생성자

변환 생성자는 사용자 정의 또는 기본 제공 형식에서 사용자 정의 형식으로의 변환을 정의합니다. 다음 예제에서는 기본 제공 형식에서 사용자 정의 Money 형식 double 으로 변환하는 변환 생성자를 보여 줍니다.

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}
```

display_balance 형식의 인수를 사용하는 Money 함수에 대한 첫 번째 호출의 경우 인수가 올바른 형식이므로 변환이 필요하지 않습니다. 그러나 두 번째 호출 display_balance 에서는 값 49.95 이 있는 인수 double 의 형식이 함수에 필요한 것이 아니기 때문에 변환이 필요합니다. 함수는 이 값을 직접 사용할 수 없지만 인수 double 형식에서 일치하는 매개 변수 Money 형식으로 변환되므로 형식의 Money 임시 값이 인수에서 생성되고 함수 호출을 완료하는 데 사용됩니다. 세 번째 호출 display_balance 에서 인수는 값 double 이 아니라 값 9.99 이 있는 인수이지만 float 컴파일러가 표준 변환(이 경우부터 double float) 을 수행한 다음 사용자 정의 변환을 수행하여 필요한 변환 double 을 완료할 수 있기 때문에 함수 호출을 계속 완료할 Money 수 있습니다.

변환 생성자 선언

변환 생성자 선언에는 다음의 규칙이 적용됩니다.

- 변환 대상 형식은 생성 중인 사용자 정의 형식입니다.
 - 일반적으로 변환 생성자에는 소스 형식에 대한 단 하나의 인수만 사용됩니다. 하지만 각각의 추가 매개 변수에 기본값이 있는 경우에는 변환 생성자가 추가 매개 변수를 지정할 수 있습니다. 소스 형식에는 첫 번째 매개 변수의 형식이 유지됩니다.
 - 모든 생성자와 마찬가지로 변환 생성자는 반환 형식을 지정하지 않습니다. 선언에 반환 형식을 지정하는 것은 오류입니다.
 - 변환 생성자는 명시적일 수 있습니다.

명시적 변환 생성자

변환 생성자를 **explicit** 선언하여 개체의 직접 초기화를 수행하거나 명시적 캐스트를 수행하는 데만 사용할 수 있습니다. 이를 통해, 클래스 형식 인수를 사용하는 함수가 변환 생성자의 소스 형식 인수를 암시적으로 사용할 수 없도록 하고 클래스 형식이 소스 형식 값을 통해 복사 초기화되지 않도록 할 수 있습니다. 다음 예제에서는 명시적 변환 생성자를 정의하는 방법과 올바른 형식의 코드에서의 그 효과를 보여줍니다.

```
C++  
  
#include <iostream>  
  
class Money  
{  
public:  
    Money() : amount{ 0.0 } {};  
    explicit Money(double _amount) : amount{ _amount } {};  
  
    double amount;  
};  
  
void display_balance(const Money balance)  
{  
    std::cout << "The balance is: " << balance.amount << std::endl;  
}  
  
int main(int argc, char* argv[])  
{  
    Money payable{ 79.99 };           // Legal: direct initialization is  
    // explicit.  
  
    display_balance(payable);         // Legal: no conversion required  
    display_balance(49.95);          // Error: no suitable conversion exists  
    to convert from double to Money.
```

```
    display_balance((Money)9.99f); // Legal: explicit cast to Money

    return 0;
}
```

이 예제에서는 명시적 변환 생성자를 사용하여 `payable`에 대한 직접 초기화를 수행할 수도 있습니다. 대신, `Money payable = 79.99;` 을 복사 초기화하려는 경우에는 오류가 됩니다. `display_balance`에 대한 첫 번째 호출의 경우 인수의 형식이 올바르므로 오류가 아닙니다. `display_balance`에 대한 두 번째 호출의 경우에는 암시적 변환을 수행하는 데 변환 생성자를 사용할 수 없으므로 오류입니다. 세 번째 호출 `display_balance`은 명시적 캐스팅으로 `Money` 인해 유효하지만 컴파일러가 암시적 캐스트 `float`를 삽입하여 캐스트를 완료하는 데 여전히 도움이 되었음을 알 수 `double` 있습니다.

암시적 변환을 유도할 수 있다는 편리함이 있긴 하지만 찾기 어려운 버그가 발생할 수도 있습니다. 경험에 따르면, 암시적으로 발생될 특정 변환을 원하는 것이 확실한 경우 이외에는 모든 변환 생성자를 명시적으로 사용하는 것이 좋습니다.

변환 함수

변환 함수는 사용자 정의 형식에서 다른 형식으로의 변환을 정의합니다. 변환 생성자와 함께 이러한 함수는 값이 다른 형식으로 캐스트될 때 호출되기 때문에 종종 "캐스트 연산자"라고 합니다. 다음 예제에서는 사용자 정의 형식에서 기본 제공 `double` 형식 `Money`으로 변환하는 변환 함수를 보여 줍니다.

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }

private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}
```

멤버 변수 `amount` 는 비공개로 설정되며 형식 `double` 에 대한 공용 변환 함수는 값을 `amount` 반환하기 위해 도입됩니다. `display_balance` 함수에서는 스트림 삽입 연산자 `balance` 를 통해 `<<`의 값이 표준 출력에 스트리밍될 때 암시적 변환이 발생합니다. 사용자 정의 형식에 대해 정의된 스트림 삽입 연산자는 없지만 기본 제공 형식 `Money double` 에는 스트림 삽입 연산자가 없으므로 컴파일러는 스트림 삽입 연산자를 만족시키기 위해 변환 함수 `Money` 를 `double` 사용할 수 있습니다.

변환 함수는 파생 클래스에 의해 상속됩니다. 파생 클래스의 변환 함수는 완전히 같은 형식으로 변환될 경우에만 상속된 변환 함수를 재정의합니다. 예를 들어 파생 클래스 연산자 `int`의 사용자 정의 변환 함수는 표준 변환에서 변환 관계를 `int short` 정의하더라도 기본 클래스 연산자의 사용자 정의 변환 함수를 재정의하거나 영향을 주지 않습니다.

변환 함수 선언

변환 함수 선언에는 다음의 규칙이 적용됩니다.

- 변환 대상 형식은 변환 함수 선언 전에 선언되어야 합니다. 클래스, 구조체, 열거형 및 `typedef`은 변환 함수 선언 내에서 선언할 수 없습니다.

C++

```
operator struct String { char string_storage; }() // illegal
```

- 변환 함수는 인수를 사용하지 않습니다. 선언에 매개 변수를 지정하는 것은 오류입니다.
- 변환 함수에는 변환 함수의 이름(변환 대상 형식의 이름이기도 함)에 의해 지정되는 반환 유형이 있습니다. 선언에 반환 형식을 지정하는 것은 오류입니다.
- 변환 함수는 가상일 수 있습니다.
- 변환 함수는 명시적일 수 있습니다.

명시적 변환 함수

변환 함수가 명시적 변환 함수로 선언되면 명시적 캐스트를 수행하는 데만 사용할 수 있습니다. 이를 통해, 변환 함수의 대상 형식에 대한 인수를 사용하는 함수가 클래스 형식에 대한 인수를 암시적으로 사용할 수 없도록 하고 대상 형식 인스턴스가 클래스 형식 값을 통해 복사 초기화되지 않도록 할 수 있습니다. 다음 예제에서는 명시적 변환 함수를 정의하는 방법과 올바른 형식의 코드에서의 그 효과를 보여줍니다.

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    explicit operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << (double)balance << std::endl;
}
```

여기서 변환 함수 **연산자 double**은 명시적으로 만들어졌으며 변환을 수행하기 위해 함수 `display_balance`에 형식 `double`에 대한 명시적 캐스트가 도입되었습니다. 이 캐스트가 생략되었다면 컴파일러는 `<<` 형식에 대한 적절한 스트림 삽입 연산자 `Money`를 찾지 못하고 오류가 발생됩니다.

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

변경할 수 있는 데이터 멤버 (C++)

아티클 • 2024. 03. 18.

이 키워드는 클래스의 비정적 및 비참조 데이터 멤버 중에서 `const`가 아닌 멤버에만 적용 할 수 있습니다. 데이터 멤버가 `mutable`로 선언된 경우 `const` 멤버 함수에서 이 데이터 멤버에 값을 할당할 수 있습니다.

구문

```
mutable member-variable-declaration;
```

설명

예를 들어 다음 코드는 `m_accessCount` 가 `mutable`로 선언되었으므로 `GetFlag` 가 `const` 멤버 함수이더라도 `GetFlag` 를 통해 수정될 수 있기 때문에 오류 없이 컴파일됩니다.

C++

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};
```

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

중첩 클래스 선언

아티클 • 2023. 10. 12.

클래스는 다른 클래스의 범위 내에서 선언될 수 있습니다. 이러한 클래스를 "중첩 클래스"라고 합니다. 중첩 클래스는 바깥쪽 클래스의 범위 내에 있는 것으로 간주되며 해당 범위 내에서 사용할 수 있습니다. 바로 바깥쪽 범위 이외의 범위에서 중첩 클래스를 참조하려면 정규화된 이름을 사용해야 합니다.

다음 예제에서는 중첩 클래스를 선언하는 방법을 보여 줍니다.

```
C++  
  
// nested_class_declarations.cpp  
class BufferedIO  
{  
public:  
    enum IOError { None, Access, General };  
  
    // Declare nested class BufferedInput.  
    class BufferedInput  
    {  
        public:  
            int read();  
            int good()  
            {  
                return _inputerror == None;  
            }  
        private:  
            IOError _inputerror;  
    };  
  
    // Declare nested class BufferedOutput.  
    class BufferedOutput  
    {  
        // Member list  
    };  
};  
  
int main()  
{  
}
```

`BufferedIO::BufferedInput` 및 `BufferedIO::BufferedOutput`은 `BufferedIO` 내에 선언됩니다. 이러한 클래스 이름은 `BufferedIO` 클래스의 범위 외부에서 표시되지 않습니다. 그러나 `BufferedIO` 형식의 개체에는 `BufferedInput` 또는 `BufferedOutput` 형식의 개체가 포함되지 않습니다.

중첩 클래스는 바깥쪽 클래스에서만 제공된 이름, 형식 이름, 정적 멤버의 이름 및 열거자를 직접 사용할 수 있습니다. 다른 클래스 멤버의 이름을 사용하려면 포인터, 참조 또는 개체 이름을 사용해야 합니다.

위의 `BufferedIO` 예제에서 `IOError` 함수에 표시된 것과 같이 중첩 클래스 `BufferedIO::BufferedInput` 또는 `BufferedIO::BufferedOutput`의 멤버 함수에서 `good` 열거형에 직접 액세스할 수 있습니다.

① 참고

중첩 클래스는 클래스 범위 내에서 형식만 선언하며, 중첩 클래스의 포함된 개체가 만들어지지는 않습니다. 위의 예제에서는 두 중첩 클래스를 선언하지만 이러한 클래스 형식의 개체는 선언하지 않습니다.

중첩 클래스 선언의 범위 표시 유형에 대한 예외는 형식 이름이 정방향 선언과 함께 선언된 경우입니다. 이 경우 정방향 선언에서 선언된 클래스 이름은 바깥쪽 클래스 외부에 표시되며 해당 범위는 가장 작은 바깥쪽 비클래스 범위로 정의됩니다. 예시:

C++

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

중첩 클래스에 대한 액세스 권한

클래스가 다른 클래스 안에 중첩되면 중첩된 클래스의 멤버 함수에 대해 특별한 액세스 권한이 부여되지 않습니다. 마찬가지로 바깥쪽 클래스의 멤버 함수는 중첩된 클래스의 멤버에 대해 특별한 액세스 권한이 없습니다.

중첩 클래스의 멤버 함수

중첩 클래스에서 선언된 멤버 함수는 파일 범위에서 정의될 수 있습니다. 앞의 예제는 다음과 같이 작성될 수도 있습니다.

C++

```
// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
public:
        int read(); // Declare but do not define member
        int good(); // functions read and good.
    private:
        IOError _inputerror;
    };

    class BufferedOutput
    {
        // Member list.
    };
};

// Define member functions read and good in
// file scope.
int BufferedIO::BufferedInput::read()
{
    return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}
int main()
{}
```

앞의 예제에서 정규화된 형식 이름 구문은 함수 이름을 선언하는 데 사용됩니다. 다음 선언은

C++

```
BufferedIO::BufferedInput::read()
```

는 "read 클래스 범위에 BufferedIO 있는 클래스의 BufferedInput 멤버인 함수"를 의미합니다. 이 선언은 정규화된 형식 이름 구문을 사용하므로 다음 형식의 구문을 사용할 수 있습니다.

C++

```
typedef BufferedIO::BufferedInput BIO_INPUT;  
  
int BIO_INPUT::read()
```

앞의 선언은 이전 선언과 동일하지만 클래스 이름 대신 이름을 사용합니다 `typedef`.

중첩 클래스의 Friend 함수

중첩 클래스에 선언된 friend 함수는 바깥쪽 클래스가 아닌 중첩 클래스 범위에 있다고 간주됩니다. 따라서 friend 함수는 바깥쪽 클래스의 멤버 또는 멤버 함수에 대해 특별한 액세스 권한이 없습니다. friend 함수가 파일 범위에 정의된 경우 friend 함수의 중첩 클래스에 선언된 이름을 사용하려면 다음과 같이 정규화된 형식 이름을 사용하세요.

C++

```
// friend_functions_and_nested_classes.cpp  
  
#include <string.h>  
  
enum  
{  
    sizeOfMessage = 255  
};  
  
char *rgszMessage[sizeOfMessage];  
  
class BufferedIO  
{  
public:  
    class BufferedInput  
    {  
        public:  
            friend int GetExtendedErrorStatus();  
            static char *message;  
            static int messageSize;  
            int iMsgNo;  
    };  
};
```

```
char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[iMsgNo] );

    return iMsgNo;
}

int main()
{
}
```

다음 코드에서는 friend 함수로 선언된 `GetExtendedErrorStatus` 함수를 보여 줍니다. 파일 범위에 정의된 함수에서는 메시지가 정적 배열에서 클래스 멤버로 복사됩니다. `GetExtendedErrorStatus` 를 효과적으로 구현하려면 다음과 같이 선언하십시오.

C++

```
int GetExtendedErrorStatus( char *message )
```

이전의 인터페이스를 사용하면 여러 클래스에서 오류 메시지를 복사할 메모리 위치를 전달하여 이 함수의 서비스를 사용할 수 있습니다.

참고 항목

[클래스 및 구조체](#)

익명 클래스 형식

아티클 • 2024. 11. 21.

클래스는 익명일 수 있습니다. 즉, 식별자 없이 선언할 수 있습니다. 이 기능은 다음과 같이 클래스 이름을 `typedef` 이름으로 바꿀 때 유용합니다.

C++

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

① 참고

이전 예제와 같이 익명 클래스를 사용하면 기존의 C 코드와 호환성을 유지하는 데 도움이 됩니다. 일부 C 코드에서는 익명 구조와 함께 사용하는 `typedef` 것이 널리 퍼져 있습니다.

다음과 같이 클래스 멤버를 참조하여 별도의 클래스에 포함되지 않은 것처럼 나타내려는 경우에도 익명 클래스가 유용합니다.

C++

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int iValue;
        long lValue;
    };
};

PTValue ptv;
```

앞의 코드 `iValue` 에서는 다음과 같이 개체 멤버 선택 연산자(.)를 사용하여 액세스할 수 있습니다.

C++

```
int i = ptv.iValue;
```

특정 제한이 익명 클래스에 적용됩니다. (익명 공용 구조체에 대한 자세한 내용은 다음을 참조하세요 .[공용 구조체](#).) 익명 클래스:

- 생성자나 소멸자를 가질 수 없습니다.
- 함수에 인수로 전달할 수 없습니다(줄임표를 사용하여 형식 검사를 무효화하지 않는 한).
- 함수에서 반환 값으로 반환될 수 없습니다.

익명 구조체

Microsoft 전용

Microsoft C 확장을 사용하면 이름을 지정하지 않고 다른 구조체 내에서 구조체 변수를 선언할 수 있습니다. 이러한 중첩된 구조체를 익명 구조체라고 합니다. C++에서는 익명 구조체를 허용하지 않습니다.

포함하는 구조체의 멤버인 것처럼 익명 구조체의 멤버에 액세스할 수 있습니다.

```
C++

// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int areacode;
    long number;
};

struct person
{
    char name[30];
    char gender;
    int age;
    int weight;
    struct phone;      // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
}
//Output: 1234567
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

pointers_to_members

아티클 • 2024. 07. 15.

멤버에 대한 포인터 선언은 포인터 선언의 특별한 경우입니다. 다음 시퀀스를 사용하여 선언됩니다.

```
storage-class-specifiersopt cv-qualifiersopt type-specifier ms-modifieropt qualified-name ::* cv-qualifiersopt identifier pm-initializeropt ;
```

1. 선언 지정자:

- 선택적 스토리지 클래스 지정자.
- 선택적 `const` 및 `volatile` 지정자.
- 형식 지정자: 형식의 이름 이는 클래스가 아니라 가리킬 멤버의 형식입니다.

2. 선언자:

- 선택적 Microsoft 전용 한정자. 자세한 내용은 [Microsoft 전용 한정자](#)를 참조하세요.
- 가리킬 멤버가 포함된 클래스의 정규화된 이름입니다.
- `::` 연산자
- `*` 연산자
- 선택적 `const` 및 `volatile` 지정자.
- 멤버에 대한 포인터의 이름을 지정하는 식별자

3. 선택적 멤버 포인터 이니셜라이저:

- `=` 연산자
- `&` 연산자
- 클래스의 정규화된 이름
- `::` 연산자
- 해당 형식 클래스의 비정적 멤버 이름입니다.

항상 그렇듯이 여러 선언자(및 모든 관련 이니셜라이저)가 단일 선언에서 허용됩니다. 멤버 포인터는 클래스의 정적 멤버, 참조 형식의 멤버 또는 `void`를 가리킬 수 없습니다.

클래스 멤버에 대한 포인터는 일반 포인터와 다릅니다. 즉, 멤버 유형과 해당 멤버가 속한 클래스에 대한 유형 정보를 모두 갖습니다. 일반 포인터는 메모리에 있는 단일 개체만 식별합니다(해당 개체의 주소를 포함함). 클래스의 멤버에 대한 포인터는 클래스의 모든 인스턴스에서 해당 멤버를 식별합니다. 다음 예제에서는 `Window` 클래스와 멤버 데이터에 대한 몇 가지 포인터를 선언합니다.

C++

```
// pointers_to_members1.cpp
class Window
{
public:
    Window();                                // Default constructor.
    Window( int x1, int y1,                  // Constructor specifying
            int x2, int y2 );                // Window size.
    bool SetCaption( const char *szTitle ); // Set window caption.
    const char *GetCaption();                // Get window caption.
    char *szWinCaption;                     // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
```

앞의 예에서 `pwCaption`은 `char*` 형식인 `Window` 클래스의 멤버 포인터입니다. `pwCaption`의 형식은 `char * Window::*`입니다. 다음 코드에서는 `SetCaption` 및 `GetCaption` 멤버 함수에 대한 포인터를 선언합니다.

C++

```
const char * (Window::* pfNWGC)() = &Window::GetCaption;
bool (Window::* pfNWSC)( const char * ) = &Window::SetCaption;
```

`pfNWGC` 및 `pfNWSC` 포인터는 `GetCaption` 클래스의 `SetCaption` 및 `Window`을 각각 가리킵니다. 이 코드에서는 멤버에 대한 포인터 `pwCaption`을 직접 사용하여 창 캡션에 정보를 복사합니다.

C++

```
Window wMainWindow;
Window *pwChildWindow = new Window;
char *szUntitled = "Untitled - ";
```

```

int      cUntitledLen = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1';      // same as
// wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; // same as
// pwChildWindow->szWinCaption[cUntitledLen - 1] = '2';

```

.* 연산자와 `->`* 연산자(멤버 포인터 연산자)의 차이점은 .* 연산자는 개체 또는 개체 참조에 따라 멤버를 선택하지만 `->`* 연산자는 포인터를 통해 멤버를 선택한다는 점입니다. 이러한 연산자에 대한 자세한 내용은 [멤버 포인터 연산자를 사용한 식](#)을 참조하세요.

멤버 포인터 연산자의 결과는 멤버의 형식입니다. 이 예제의 경우 `char *`입니다.

다음 코드에서는 멤버에 대한 포인터를 사용하여 `GetCaption` 및 `SetCaption` 멤버 함수를 호출합니다.

C++

```

// Allocate a buffer.
enum {
    sizeOfBuffer = 100
};
char szCaptionBase[sizeOfBuffer];

// Copy the main window caption into the buffer
// and append " [View 1]".
strcpy_s( szCaptionBase, sizeOfBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeOfBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );

```

멤버에 대한 포인터 제한

정적 멤버의 주소는 멤버 포인터가 아닙니다. 정적 멤버의 주소는 정적 멤버의 인스턴스 하나에 대한 일반적인 포인터입니다. 특정 클래스의 모든 개체에 대해 정적 멤버의 인스턴스는 하나만 존재합니다. 이는 일반적인 주소(&) 및 역참조(*) 연산자를 사용할 수 있음을 의미합니다.

멤버 및 가상 함수에 대한 포인터

멤버 포인터 함수를 통해 가상 함수를 호출하면 함수가 직접 호출된 것처럼 작동합니다. 올바른 함수가 v-table에서 조회되어 호출됩니다.

가상 함수 작업은 항상 기본 클래스 포인터를 통해 호출합니다. (가상 함수에 대한 자세한 내용은 [가상 함수를 참조하세요.](#))

다음 코드에서는 멤버 포인터 함수를 통해 가상 함수를 호출하는 방법을 보여 줍니다.

C++

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void Print();
};

void (Base::* bfnPrint)() = &Base::Print;
void Base::Print()
{
    cout << "Print function for class Base" << endl;
}

class Derived : public Base
{
public:
    void Print(); // Print is still a virtual function.
};

void Derived::Print()
{
    cout << "Print function for class Derived" << endl;
}

int main()
{
    Base    *bPtr;
    Base    bObject;
    Derived dObject;
    bPtr = &bObject;    // Set pointer to address of bObject.
    (bPtr->*bfnPrint)();
    bPtr = &dObject;    // Set pointer to address of dObject.
    (bPtr->*bfnPrint)();
}

// Output:
// Print function for class Base
// Print function for class Derived
```

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

this 포인터

아티클 • 2024. 09. 26.

`this` 포인터는 `class`, `struct` 또는 `union` 형식의 비정적 멤버 함수 내에서만 액세스할 수 있는 포인터입니다. 멤버 함수가 호출되는 개체를 가리킵니다. 정적 멤버 함수에는 `this` 포인터가 없습니다.

구문

C++

```
this  
this->member-identifier
```

설명

개체의 `this` 포인터는 개체 자체의 일부가 아닙니다. 개체에 대한 `sizeof` 문의 결과에 포함되지 않습니다. 비정적 멤버 함수가 개체에 대해 호출되는 경우 컴파일러가 개체의 주소를 숨겨진 인수로 함수에 전달합니다. 예를 들어, 다음 함수 호출은

C++

```
myDate.setMonth( 3 );
```

다음과 같이 해석할 수 있습니다.

C++

```
setMonth( &myDate, 3 );
```

개체의 주소를 멤버 함수 안에서 `this` 포인터로 사용할 수 있습니다. 대부분의 `this` 포인터 사용은 암시적입니다. `class`의 멤버를 참조할 때는 불필요하더라도 명시적 `this`를 사용하는 것이 적절합니다. 예시:

C++

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    this->month = mn;     // are equivalent
```

```
(*this).month = mn;  
}
```

멤버 함수에서 현재 개체를 반환하기 위해 `*this` 식이 주로 사용됩니다.

C++

```
return *this;
```

`this` 포인터를 사용하여 자신에 대한 참조를 막기도 합니다.

C++

```
if (&Object != this) {  
// do not execute in cases of self-reference
```

① 참고

`this` 포인터는 수정할 수 없으므로 `this` 포인터에 대한 할당은 허용되지 않습니다.
이전의 C++ 구현에서는 `this`에 할당할 수 있었습니다.

경우에 따라 `this` 포인터가 직접 사용됩니다(예: 현재 개체의 주소가 필요한 자체 참조 데이터 구조를 조작하는 경우).

예시

C++

```
// this_pointer.cpp  
// compile with: /EHsc  
  
#include <iostream>  
#include <string.h>  
  
using namespace std;  
  
class Buf  
{  
public:  
    Buf( char* szBuffer, size_t sizeOfBuffer );  
    Buf& operator=( const Buf & );  
    void Display() { cout << buffer << endl; }  
  
private:  
    char*   buffer;
```

```

        size_t sizeOfBuffer;
    };

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
    if( &otherbuf != this )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer = strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}

```

Output

```

my buffer
your buffer

```

this 포인터 형식

`this` 포인터의 형식은 함수 선언에 `const` 및/또는 `volatile` 키워드가 포함되어 있는지 여부에 따라 달라집니다. 다음 구문에서는 멤버 함수의 `this` 형식을 설명합니다.

[*cv-qualifier-list*] *class-type* * `const` `this`

멤버 함수의 선언자는 *cv-qualifier-list*를 결정합니다. `const` 또는 `volatile`(또는 둘 다)일 수 있습니다. *class-type*은 class의 이름입니다.

`this` 포인터를 다시 할당할 수 없습니다. 멤버 함수 선언에 사용되는 `const` 또는 `volatile` 한정자는 다음 표와 같이 해당 함수 범위에서 `this` 포인터가 가리키는 class 인스턴스에 적용됩니다.

[+] 테이블 확장

멤버 함수 선언	<code>myClass</code> 로 명명된 class에 대한 <code>this</code> 포인터의 형식
<code>void Func()</code>	<code>myClass *</code>
<code>void Func() const</code>	<code>const myClass *</code>
<code>void Func() volatile</code>	<code>volatile myClass *</code>
<code>void Func() const volatile</code>	<code>const volatile myClass *</code>

다음 표에서는 `const` 및 `volatile`에 대해 자세히 설명합니다.

`this` 한정자의 의미 체계

[+] 테이블 확장

한정자	의미
<code>const</code>	멤버 데이터를 변경할 수 없습니다. <code>const</code> 가 아닌 멤버 함수는 호출할 수 없습니다.
<code>volatile</code>	액세스할 때마다 메모리에서 멤버 데이터를 로드하고 특정 최적화를 비활성화합니다.

`const` 개체를 `const`가 아닌 멤버 함수로 전달할 때 발생하는 오류입니다.

마찬가지로 `volatile` 개체를 `volatile`이 아닌 멤버 함수로 전달할 때도 발생하는 오류입니다.

`const`로 선언된 멤버 함수는 멤버 데이터를 변경할 수 없습니다. `const` 함수에서 `this` 포인터는 `const` 개체에 대한 포인터입니다.

① 참고

생성자 및 소멸자는 다음과 같이 `const` 또는 `volatile`로 선언할 수 없습니다. 그러나 `const` 또는 `volatile` 개체에 대해 호출할 수 있습니다.

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

C++ 비트 필드

아티클 • 2024. 07. 08.

클래스와 구조체는 정수 형식보다 작은 스토리지 공간을 차지하는 멤버를 포함할 수 있습니다. 이러한 멤버는 비트 필드로 지정됩니다. 비트 필드 멤버-선언자 사양의 구문은 다음을 따릅니다.

구문

declarator : constant-expression

설명

(선택 사항) 선언자는 프로그램에서 멤버에 액세스하는 데 사용되는 이름입니다. 정수 형식(열거형 형식 포함)이어야 합니다. *constant-expression*은 구조체에서 멤버가 차지하는 비트 수를 지정합니다. 익명 비트 필드(식별자가 없는 비트 필드 멤버)를 안쪽 여백에 사용할 수 있습니다.

① 참고

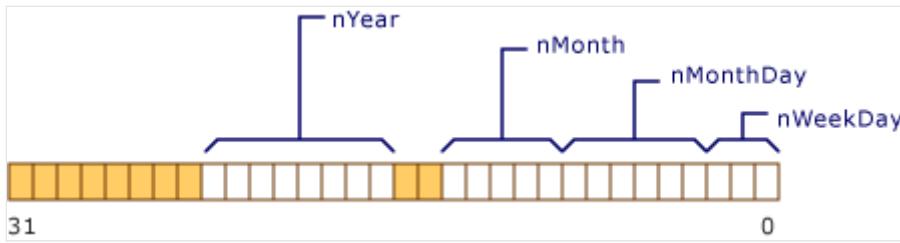
너비가 0인 명명되지 않은 비트 필드는 다음 비트 필드를 다음 **type** 경계에 강제로 맞춥니다. 여기서 **type**은 멤버의 형식입니다.

다음 예제에서는 비트 필드가 포함된 구조체를 선언합니다.

C++

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nWeekDay : 3;      // 0..7  (3 bits)
    unsigned short nMonthDay : 6;      // 0..31 (6 bits)
    unsigned short nMonth : 5;        // 0..12 (5 bits)
    unsigned short nYear : 8;         // 0..100 (8 bits)
};
```

다음 그림에서는 **Date** 형식 객체의 개념적 메모리 레이아웃을 보여 줍니다.



nYear은 길이가 8비트이므로 선언된 형식 `unsigned short`의 단어 경계를 오버플로합니다. 따라서 새 `unsigned short`의 시작 부분에서 시작됩니다. 모든 비트 필드가 기본 형식의 한 개체에 맞아야 할 필요는 없습니다. 선언에서 요청된 비트 수에 따라 새 스토리지 단위가 할당됩니다.

Microsoft 전용

비트 필드로 선언된 데이터의 순서는 이전 그림과 같이 낮은 비트에서 높은 비트로 이루어집니다.

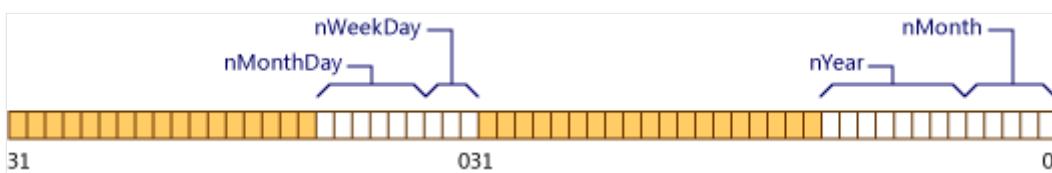
Microsoft 전용 종료

다음 예와 같이 구조체 선언에 길이가 0인 명명되지 않은 필드가 포함된 경우

C++

```
// bit_fields2.cpp
// compile with: /LD
struct Date {
    unsigned nWeekDay : 3;      // 0..7   (3 bits)
    unsigned nMonthDay : 6;     // 0..31  (6 bits)
    unsigned : 0;              // Force alignment to next boundary.
    unsigned nMonth : 5;        // 0..12  (5 bits)
    unsigned nYear : 8;         // 0..100 (8 bits)
};
```

그러면 메모리 레이아웃은 다음 그림과 같습니다.



비트 필드의 기본 형식은 [기본 제공 형식](#)에 설명된 대로 정수 형식이어야 합니다.

`const T&` 형식의 참조에 대한 이니셜라이저가 `T` 형식의 비트 필드를 참조하는 lvalue인 경우 참조는 비트 필드에 직접 바인딩되지 않습니다. 대신 참조는 비트 필드의 값을 보유하기 위해 초기화된 임시 항목에 바인딩됩니다.

비트 필드에 대한 제한

비트 필드에 대한 잘못된 연산은 다음과 같습니다.

- 비트 필드의 주소 가져오기
- 비트 필드를 사용하여 `const` 가 아닌 참조를 초기화합니다.

참고 항목

[클래스 및 구조체](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

C++ 람다 식

아티클 • 2023. 04. 03.

C++11 이상에서는 람다라고도 하는 람다 식은 함수에 인수로 호출되거나 전달되는 위치에서 바로 익명 함수 객체(클로저)를 정의하는 편리한 방법입니다. 일반적으로 람다는 알고리즘 또는 비동기 함수에 전달되는 몇 줄의 코드를 캡슐화하는 데 사용됩니다. 이 문서에서는 람다를 정의하고 다른 프로그래밍 기술과 비교합니다. 해당 장점을 설명하고 몇 가지 기본 예제를 제공합니다.

관련 문서

- [람다 식과 함수 개체](#)
- [람다 식 작업](#)
- [constexpr 람다 식](#)

람다 식의 일부

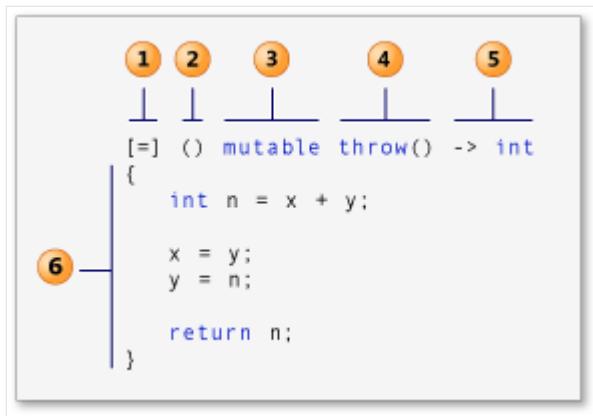
다음은 함수에 세 번째 인수 `std::sort()`로 전달되는 간단한 람다입니다.

```
C++

#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [] (float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

이 그림에서는 람다 구문의 일부를 보여 줍니다.



1. *capture* 절 (C++ 사양의 람다 소개자라고도 함)
2. *매개* 변수 목록 선택적. (람다 선언자라고도 함)
3. *변경 가능한* 사양 선택적.
4. *exception-specification* 선택적.
5. *trailing-return-type* 선택적.
6. 람다 본문.

캡처 절

람다는 본문에 새 변수(C++14)를 도입할 수 있으며 주변 범위에서 변수에 액세스하거나 캡처할 수도 있습니다. 람다는 캡처 절로 시작합니다. 캡처되는 변수와 캡처가 값에 의한 것인지 참조인지를 지정합니다. 앤퍼샌드(&) 접두사를 가진 변수는 참조 및 값으로 액세스하지 않은 변수에 의해 액세스됩니다.

빈 캡처 절인 []는 람다 식의 본문이 바깥쪽 범위의 변수에 액세스하지 않음을 나타냅니다.

캡처 기본 모드를 사용하여 람다 본문 [&]에서 참조되는 외부 변수를 캡처하는 방법을 나타낼 수 있습니다. 참조하는 모든 변수가 참조 [=]로 캡처되고 값으로 캡처됨을 의미합니다. 기본 캡처 모드를 사용하고 특정 변수에 대해서는 반대 모드를 지정할 수 있습니다. 예를 들어, 람다 본문이 외부 변수 total에 참조별로 액세스하고 외부 변수 factor에 값별로 액세스하는 경우 다음 캡처 절이 동일합니다.

C++

```

[&total, factor]
[factor, &total]
[&, factor]
[=, &total]

```

capture-default를 사용하는 경우 람다 본문에 언급된 변수만 캡처됩니다.

캡처 절에 캡처 기본값 `&`이 포함된 경우 해당 캡처 절의 캡처에 형식 `&identifier`이 있을 수 있는 식별자는 없습니다. 마찬가지로 캡처 절에 capture-default `=`가 포함된 경우 해당 캡처 절의 캡처에는 형식 `=identifier`이 있을 수 없습니다. 식별자 또는 `this`는 캡처 절에 두 번 이상 나타날 수 없습니다. 다음 코드 조각은 몇 가지 예를 보여 줍니다.

C++

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};           // OK
    [&, &i]{};          // ERROR: i preceded by & when & is the default
    [=, this]{};         // ERROR: this when = is the default
    [=, *this]{};        // OK: captures this by value. See below.
    [i, i]{};           // ERROR: i repeated
}
```

이 variadic 템플릿 예제와 같이 캡처 뒤에 줄임표가 오는 것은 팩 확장입니다.

C++

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

클래스 멤버 함수의 본문에 람다 식을 사용하려면 캡처 절에 포인터를 전달 `this` 하여 바깥쪽 클래스의 멤버 함수 및 데이터 멤버에 대한 액세스를 제공합니다.

Visual Studio 2017 버전 15.3 이상 (모드 이상에서 `/std:c++17` 사용 가능): `this` 캡처 절에서를 지정하여 값으로 포인터를 `*this` 캡처할 수 있습니다. 값으로 캡처하면 전체 클로저가 람다가 호출되는 모든 호출 사이트에 복사됩니다. (클로저는 람다 식을 캡슐화하는 익명 함수 개체입니다.) 값별 캡처는 람다가 병렬 또는 비동기 작업에서 실행되는 경우에 유용합니다. NUMA와 같은 특정 하드웨어 아키텍처에 특히 유용합니다.

클래스 멤버 함수와 함께 람다 식을 사용하는 방법을 보여 주는 예제는 [람다 식 예제의 "예제: 메서드에서 람다 식 사용"](#)을 참조하세요.

캡처 절을 사용하는 경우 특히 다중 스레딩과 함께 람다를 사용하는 경우 이러한 점을 염두에 두는 것이 좋습니다.

- 참조 캡처를 사용하여 외부에서 변수를 수정할 수 있지만 값 캡처는 수정할 수 없습니다. (`mutable` 복사본을 수정할 수 있지만 원본은 수정할 수 없습니다.)

- 참조 캡처는 외부 변수에 대한 업데이트를 반영하지만 값 캡처는 그렇지 않습니다.
- 참조 캡처는 수명 종속성을 발생시키지만 값 캡처에는 수명 종속성이 없습니다. 람다가 비동기적으로 실행되는 경우 특히 중요합니다. 비동기 람다에서 참조로 로컬을 캡처하는 경우 람다가 실행될 때까지 해당 로컬이 쉽게 사라질 수 있습니다. 코드로 인해 런타임에 액세스 위반이 발생할 수 있습니다.

일반화된 캡처(C++ 14)

C++14에서는 람다 함수의 바깥쪽 범위에 해당 변수가 존재하지 않아도 캡처 절에 새 변수를 도입하고 초기화할 수 있습니다. 이러한 초기화는 임의의 식으로 표현할 수 있습니다. 새 변수의 형식은 식에서 생성되는 형식에서 추론됩니다. 이 기능을 사용하면 주변 범위에서 이동 전용 변수(예: `std::unique_ptr`)를 캡처하고 람다에서 사용할 수 있습니다.

C++

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

매개 변수 목록

람다는 변수를 캡처하고 입력 매개 변수를 수락할 수 있습니다. 매개 변수 목록(표준 구문의 람다 선언자)은 선택 사항이며 대부분의 측면에서 함수의 매개 변수 목록과 유사합니다.

C++

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

C++14에서 매개 변수 형식이 제네릭인 경우 키워드를 `auto` 형식 지정자로 사용할 수 있습니다. 이 키워드는 컴파일러에 함수 호출 연산자를 템플릿으로 만들도록 지시합니다. 매개 변수 목록의 `auto` 각 인스턴스는 고유 형식 매개 변수와 동일합니다.

C++

```
auto y = [] (auto first, auto second)
{
```

```
    return first + second;  
};
```

람다 식은 다른 람다 식을 인수로 사용할 수 있습니다. 자세한 내용은 [람다 식 예제](#) 문서의 "상위 순서 람다 식"을 참조하세요.

매개 변수 목록은 선택 사항이므로 람다 식에 인수를 전달하지 않고 람다 선언자에 [예외 사양](#), [후행 반환 형식](#) 또는 [mutable](#)이 포함되지 않은 경우 빈 괄호를 생략할 수 있습니다.

변경 가능한 사양

일반적으로 람다의 함수 호출 연산자는 const-by-value이지만 키워드를 [mutable](#) 사용하면 이를 취소합니다. 변경 가능한 데이터 멤버를 생성하지 않습니다. 사양 [mutable](#) 을 사용하면 람다 식의 본문이 값으로 캡처되는 변수를 수정할 수 있습니다. 이 문서의 뒷부분에 있는 몇 가지 예제에서는 를 사용하는 [mutable](#) 방법을 보여 줍니다.

예외 사양

예외 사양을 [noexcept](#) 사용하여 람다 식이 예외를 throw하지 않음을 나타낼 수 있습니다. 일반 함수와 마찬가지로 람다 식이 예외 사양을 선언 [noexcept](#) 하고 람다 본문이 예외를 throw하는 경우 Microsoft C++ 컴파일러는 다음과 같이 경고 [C4297](#)을 생성합니다.

C++

```
// throw_lambda_expression.cpp  
// compile with: /W4 /EHsc  
int main() // C4297 expected  
{  
    []() noexcept { throw 5; }();  
}
```

자세한 내용은 [예외 사양\(throw\)](#)을 참조하세요.

반환 형식

람다 식의 반환 형식은 자동으로 추론됩니다. [후행 반환 형식](#)을 [auto](#) 지정하지 않는 한 키워드를 사용할 필요가 없습니다. [후행 반환 형식](#)은 일반 함수 또는 멤버 함수의 반환 형식 부분과 유사합니다. 그러나 반환 형식은 매개 변수 목록 뒤에 와야 하며 반환 형식 앞에 trailing-return-type 키워드 [->](#)를 포함해야 합니다.

람다 본문에 return 문이 하나만 포함된 경우 람다 식의 반환 형식 부분을 생략할 수 있습니다. 또는 식이 값을 반환하지 않는 경우입니다. 람다 본문에 단일 return 문이 포함되어

있으면 컴파일러는 반환 식의 형식에서 반환 형식을 추론합니다. 그렇지 않으면 컴파일러는 반환 형식을 `void` 추론합니다. 이 원칙을 설명하는 다음 예제 코드 조각을 고려합니다.

C++

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list isn't
                                valid
```

람다 식은 다른 람다 식을 반환 값으로 생성할 수 있습니다. 자세한 내용은 람다 식 예제의 "상위 순서 람다 식"을 참조하세요.

람다 본문

람다 식의 람다 본문은 복합 문입니다. 일반 함수 또는 멤버 함수의 본문에 허용되는 모든 항목을 포함할 수 있습니다. 일반 함수와 람다 식 모두의 본문은 다음과 같은 종류의 변수에 액세스할 수 있습니다.

- 앞의 설명대로 바깥쪽 범위에서 캡처된 변수
- 매개 변수.
- 로컬로 선언된 변수입니다.
- 클래스 내에서 선언되고 캡처되는 경우 클래스 `this` 데이터 멤버입니다.
- 정적 스토리지 기간이 있는 모든 변수(예: 전역 변수).

다음 예제에는 `n` 변수를 값별로 명시적으로 캡처하고 `m` 변수를 참조별로 암시적으로 캡처하는 람다 식이 포함되어 있습니다.

C++

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

Output

```
5  
0
```

n 변수는 값별로 캡처되므로 람다 식을 호출한 후 해당 값이 0으로 유지됩니다. 사양은 mutable n 람다 내에서 수정할 수 있습니다.

람다 식은 자동 스토리지 기간이 있는 변수만 캡처할 수 있습니다. 그러나 람다 식의 본문에 정적 스토리지 기간이 있는 변수를 사용할 수 있습니다. 다음 예제는 generate 함수 및 람다 식을 사용하여 vector 개체의 각 요소에 값을 할당합니다. 람다 식은 정적 변수를 수정하여 다음 요소의 값을 생성합니다.

C++

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}
```

자세한 내용은 생성을 참조하세요.

다음 코드 예제에서는 이전 예제의 함수를 사용하고 C++ 표준 라이브러리 알고리즘 generate_n을 사용하는 람다 식의 예를 추가합니다. 이 람다 식은 vector 개체의 요소를 이전의 두 요소의 합에 할당합니다. 키워드는 mutable 람다 식의 본문이 람다 식이 값으로 캡처하는 외부 변수 x 및 y의 복사본을 수정할 수 있도록 사용됩니다. 람다 식이 원래 변수 x 및 y를 값별로 캡처하기 때문에 람다가 실행된 후에도 값이 1로 유지됩니다.

C++

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
```

```

cout << s;

for (const auto& e : c) {
    cout << e << " ";
}

cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
    int y = 1;

    // Sets each element in the vector to the sum of the
    // previous two elements.
    generate_n(v.begin() + 2,
               elementCount - 2,
               [=]() mutable throw() -> int { // lambda is the 3rd parameter
                   // Generate current value.
                   int n = x + y;
                   // Update previous two values.
                   x = y;
                   y = n;
                   return n;
               });
    print("vector v after call to generate_n() with lambda: ", v);

    // Print the local variables x and y.
    // The values of x and y hold their initial values because
    // they are captured by value.
    cout << "x: " << x << " y: " << y << endl;

    // Fill the vector with a sequence of numbers
    fillVector(v);
    print("vector v after 1st call to fillVector(): ", v);
}

```

```

    // Fill the vector with the next sequence of numbers
    fillVector(v);
    print("vector v after 2nd call to fillVector(): ", v);
}

```

Output

```

vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18

```

자세한 내용은 [generate_n](#) 참조하세요.

constexpr 람다 식

Visual Studio 2017 버전 15.3 이상 (모드 이상에서 [/std:c++17](#) 사용 가능): 캡처되거나 도입된 각 데이터 멤버의 초기화가 상수 식 내에서 허용되는 경우 람다 식을 (또는 상수 식에서 사용)로 `constexpr` 선언할 수 있습니다.

C++

```

int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}

```

결과가 함수의 요구 사항을 충족하는 경우 람다는 암시적으로 `constexpr`입니다 `constexpr`.

C++

```

auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);

```

람다가 암시적 또는 명시적으로 `constexpr`이면 함수 포인터로 변환하면 함수가 `constexpr` 생성됩니다.

C++

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Microsoft 전용

람다는 CLR(공용 언어 런타임) 관리형 엔터티 `ref class`, `ref struct`, `value class` 또는 `value struct`에서 지원되지 않습니다.

와 같은 `_declspec` Microsoft 관련 한정자를 사용하는 경우 바로 뒤에 `parameter-declaration-clause` 있는 람다 식에 삽입할 수 있습니다. 예를 들면 다음과 같습니다.

C++

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;
};
```

특정 한정자가 람다에서 지원되는지 여부를 확인하려면 [Microsoft 관련](#) 한정자 섹션의 한정자에 대한 문서를 참조하세요.

Visual Studio는 C++11 표준 람다 기능 및 상태 비정상 람다를 지원합니다. 상태 비 상태 람다는 임의의 호출 규칙을 사용하는 함수 포인터로 변환할 수 있습니다.

참조

[C++ 언어 참조](#)

[C++ 표준 라이브러리의 함수 개체](#)

[함수 호출](#)

[for_each](#)

람다 식 구문

아티클 • 2023. 10. 12.

이 문서에서는 람다 식의 구문과 구성 요소에 대해 설명합니다. 람다 식에 대한 설명은 람다 식을 참조 [하세요](#).

함수 개체와 람다

코드를 작성할 때는 특히 C++ 표준 라이브러리 알고리즘을 사용하는 경우 함수 포인터와 함수 개체를 사용하여 문제를 해결하고 계산을 수행할 수 있습니다. 함수 포인터와 함수 개체는 각각 장단점이 있습니다. 예를 들어 함수 포인터는 최소한의 구문 오버헤드가 있지만 범위 내에 상태를 유지하지 않으며 함수 개체는 상태를 유지할 수 있지만 클래스 정의의 구문 오버헤드가 필요합니다.

람다는 함수 포인터와 함수 개체의 이점을 결합하여 단점을 방지합니다. 함수 개체와 마찬가지로 람다는 유연하고 상태를 기본 수 있지만 함수 개체와 달리 컴팩트 구문에는 명시적 클래스 정의가 필요하지 않습니다. 람다를 사용하면 코드를 더 쉽게 작성할 수 있고 해당 함수 개체에 대한 코드보다 오류 발생 가능성이 적습니다.

다음 예제에서는 람다 사용과 함수 개체 사용을 비교합니다. 첫 번째 예제에서는 람다를 사용하여 `vector` 개체의 각 요소가 짝수이든 홀수이든 콘솔에 인쇄합니다. 두 번째 예제에서는 함수 개체를 사용하여 같은 작업을 수행합니다.

예제 1: 람다 사용하기

다음은 람다를 `for_each` 함수에 전달하는 예제입니다. 람다는 `vector` 개체의 각 요소가 짝수인지 홀수인지 보여 주는 결과를 인쇄합니다.

코드

```
C++

// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Create a vector object that contains 9 elements.
```

```

vector<int> v;
for (int i = 1; i < 10; ++i) {
    v.push_back(i);
}

// Count the number of even numbers in the vector by
// using the for_each function and a lambda.
int evenCount = 0;
for_each(v.begin(), v.end(), [&evenCount] (int n) {
    cout << n;
    if (n % 2 == 0) {
        cout << " is even " << endl;
        ++evenCount;
    } else {
        cout << " is odd " << endl;
    }
});

// Print the count of even numbers to the console.
cout << "There are " << evenCount
     << " even numbers in the vector." << endl;
}

```

Output

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

주석

이 예제에서 `for_each` 함수에 대한 세 번째 인수는 람다입니다. `[&evenCount]` 부분은 식의 캡처 절을 지정하고 `(int n)`은 매개 변수 목록을 지정하고 나머지 부분은 식의 본문을 지정합니다.

예제 2: 함수 개체 사용하기

때로는 람다가 너무 비대해져서 이전 예제보다 훨씬 더 확장할 수 없습니다. 다음 예제에서는 `for_each` 함수와 함께 람다 대신 함수 개체를 사용하여 예제 1과 동일한 결과를 생성합니다. 두 예제 모두 `vector` 개체에 짹수의 수를 저장합니다. 연산의 상태를 유지하기 위

해 `FunctorClass` 클래스는 `m_evenCount` 변수를 멤버 변수로 참조하면서 저장합니다. 작업을 `FunctorClass` 수행하려면 함수 호출 연산자인 `operator()`를 구현합니다. Microsoft C++ 컴파일러는 예제 1의 람다 코드와 크기 및 성능이 비슷한 코드를 생성합니다. 이 문서의 문제와 같은 기본적인 문제의 경우 함수 개체 디자인보다 간단한 람다 디자인이 더 좋습니다. 그러나 이 기능에 나중에 중요한 확장이 필요할 수 있다면 코드 유지 관리를 더 수월하게 할 수 있도록 함수 개체 디자인을 사용합니다.

`operator()`에 대한 자세한 내용은 [함수 호출을 참조하세요](#). `for_each` 함수에 대한 자세한 내용은 [`for_each` 참조하세요](#).

코드

C++

```
// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
```

```

{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

Output

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

참고 항목

[람다 식](#)
[람다 식의 예](#)
[generate](#)
[generate_n](#)
[for_each](#)
[예외 사양\(throw\)](#)
[컴파일러 경고\(수준 1\) C4297](#)
[Microsoft 전용 한정자](#)

람다 식의 예

아티클 • 2024. 03. 09.

이 문서에서는 프로그램에 람다 식을 사용하는 방법을 보여 줍니다. 람다 식에 대한 개요는 [람다 식](#)을 참조하세요. 람다 식의 구조체에 대한 자세한 내용은 [람다 식 구문](#)을 참조하세요.

람다 식 선언

예 1

람다 식에 형식이 지정되었으므로 다음과 같이 `auto` 변수 또는 `function` 객체에 할당할 수 있습니다.

C++

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto
    // variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

```
5
7
```

설명

자세한 내용은 [auto, function 클래스 및 함수 호출](#)을 참조하세요.

람다 식은 함수의 본문에서 대부분 선언되지만 변수를 초기화할 수 있는 어느 곳에서나 선언할 수 있습니다.

예제 2

Microsoft C++ 컴파일러는 식이 호출되는 대신 식이 선언될 때 캡처된 변수에 람다 식을 바인딩합니다. 다음 예제에서는 변수 지역 변수 `i` 값과 참조로서 변수 `j`를 캡처하는 람다 식을 보여 줍니다. 람다 식은 값을 기준으로 `i`(을)를 캡처하기 때문에 프로그램의 뒷부분에 `i`(을)를 다시 할당해도 식의 결과에는 영향을 주지 않습니다. 그러나 람다 식을 `j`를 참조로 캡처하기 때문에 `j`를 다시 할당하면 식의 결과에 영향을 주지 않습니다.

```
C++

// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    int i = 3;
    int j = 5;

    // The following lambda expression captures i by value and
    // j by reference.
    function<int (void)> f = [i, &j] { return i + j; };

    // Change the values of i and j.
    i = 22;
    j = 44;

    // Call f and print its result.
    cout << f() << endl;
}
```

이 예에서는 다음과 같은 출력을 생성합니다.

```
Output
47
```

람다 식 호출

다음 코드 조각과 같이 람다 식을 즉시 호출할 수 있습니다. 두 번째 코드 조각은 `find_if`(와)과 같은 C++ 표준 라이브러리 알고리즘에 인수로 람다를 전달하는 방법을 보여 줍니다.

예 1

이 예제에서는 두 정수의 합을 반환하고 식 인수를 사용하여 인수 5 및 4로 식을 즉시 호출하는 람다 식을 선언합니다.

C++

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

```
9
```

예제 2

이 예제에서는 람다 식을 `find_if` 함수에 대한 인수로 전달합니다. 람다 식은 매개 변수가 짝수이면 `true`(을)를 반환합니다.

C++

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>
```

```

int main()
{
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
    numbers.push_back(46);
    numbers.push_back(99);

    // Use the find_if function and a lambda expression to find the
    // first even number in the list.
    const list<int>::const_iterator result =
        find_if(numbers.begin(), numbers.end(), [] (int n) { return (n % 2) ==
0; });

    // Print the result.
    if (result != numbers.end()) {
        cout << "The first even number in the list is " << *result << "."
        endl;
    } else {
        cout << "The list contains no even numbers." << endl;
    }
}

```

이 예제에서는 다음과 같은 출력을 생성합니다.

Output

```
The first even number in the list is 42.
```

설명

`find_if` 함수에 대한 자세한 내용은 [find_if\(을\)](#)를 참조하세요. 일반적인 알고리즘을 수행하는 C++ 표준 라이브러리 함수에 대한 자세한 내용은 [<algorithm>\(을\)](#)를 참조하세요.

[[이 문서의 내용](#)]

람다 식 중첩

예시

이 예제와 같이 람다 식을 다른 람다 식 안에 중첩할 수 있습니다. 안쪽 람다 식은 인수를 2를 곱한 후 결과를 반환합니다. 바깥쪽 람다 식은 안쪽 람다 식의 인수와 함께 호출하고

결과에 3을 더합니다.

C++

```
// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.
    int timestwoplusthree = [](int x) { return [](int y) { return y * 2; }
(x) + 3; }(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}
```

이 예제에서는 다음과 같은 출력을 생성합니다.

Output

```
13
```

설명

이 예제에서 `[](int y) { return y * 2; }`는 중첩된 람다 식입니다.

[[이 문서의 내용](#)]

고차 람다 함수

예시

대부분의 프로그래밍 언어는 고차 함수의 개념을 지원합니다. 고차 함수는 람다 식으로, 다른 람다 식을 인수로 취하거나 람다 식을 반환합니다. `function` 클래스를 사용하여 C++ 람다 식이 고차 함수처럼 동작하도록 설정할 수 있습니다. 다음 예제에서는 `function` 객체를 반환하는 람다 식과 인수로서 `function` 객체를 취하는 람다 식을 보여 줍니다.

C++

```

// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}

```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

30

[[이 문서의 내용](#)]

함수에서 람다 식 사용

예시

함수의 본문에서 람다 식을 사용할 수 있습니다. 람다 식은 바깥쪽 함수에서 액세스할 수 있는 모든 함수 또는 데이터 멤버에 액세스할 수 있습니다. 바깥쪽 클래스의 함수 및 데이터 멤버에 대한 액세스를 제공하기 위해 `this` 포인터를 명시적으로 또는 암시적으로 캡처할 수 있습니다. **Visual Studio 2017 버전 15.3 이상**(/std:c++17 이상에서 사용 가능): 원

래 개체가 범위를 벗어난 후 코드가 실행될 수 있는 비동기 또는 병렬 작업에서 람다가 사용될 때 값 (`[*this]`)으로 `this`(을)를 캡처합니다.

다음과 같이 함수에서 `this` 포인터를 명시적으로 사용할 수 있습니다.

C++

```
// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [*this](int n) { cout << n * _scale << endl; });
}
```

`this` 포인터를 암시적으로 캡처할 수도 있습니다.

C++

```
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}
```

다음 예제에서는 소수 자릿수 값을 캡슐화하는 `Scale` 클래스를 보여 줍니다.

C++

```
// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
}
```

```

// and the scale value to the console.
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale <<
endl; });
}

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. doesn't modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}

```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

```

3
6
9
12

```

설명

`ApplyScale` 함수는 람다 식을 사용하여 `vector` 개체에서 스케일 값 및 각 요소의 곱을 인쇄합니다. 람다 식은 `_scale` 멤버에 액세스할 수 있도록 `this`(을)를 암시적으로 캡처합니다.

[[이 문서의 내용](#)]

템플릿이 있는 람다 식 사용

예시

람다 식이 형식화되기 때문에 C++ 템플릿과 함께 사용할 수 있습니다. 다음 예제에서는 `negate_all` 및 `print_all` 함수를 보여 줍니다. `negate_all` 함수는 `vector` 개체의 각 요소에 단항 `operator-` (을)를 적용합니다. `print_all` 함수는 `vector` 개체의 각 요소를 콘솔에 인쇄합니다.

C++

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes signed data type.
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}

int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all(): " << endl;
    print_all(v);
}
```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

```
34
-43
56
After negate_all():
```

설명

C++ 템플릿에 대한 자세한 내용은 [템플릿](#)을 참조하세요.

[[이 문서의 내용](#)]

예외 처리

예시

람다 식의 본문은 SEH(구조적 예외 처리)와 C++ 예외 처리에 대한 규칙을 따릅니다. 람다 식의 본문에는 양각된 예외를 처리하거나 예외 처리를 포함하는 범위를 지연시킬 수 있습니다. 다음 예제에서는 `for_each` 함수와 람다 식을 사용하여 `vector` 개체를 다른 개체의 값으로 채웁니다. `try/catch` 블록을 사용하여 첫 번째 벡터에 대한 잘못된 액세스를 처리합니다.

C++

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);

    // Create another vector that contains index values.
    vector<int> indices(3);
    indices[0] = 0;
    indices[-1] = 1; // This is not a valid subscript. It will trigger an
exception.
    indices[2] = 2;

    // Use the values from the vector of index values to
    // fill the elements vector. This example uses a
    // try/catch block to handle invalid access to the
    // elements vector.
    try
    {
```

```

        for_each(indices.begin(), indices.end(), [&](int index) {
            elements.at(index) = index;
        });
    }
    catch (const out_of_range& e)
    {
        cerr << "Caught '" << e.what() << "'" << endl;
    };
}

```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

```
Caught 'invalid vector<T> subscript'.
```

설명

예외 처리에 대한 자세한 내용은 [예외 처리](#)를 참조하세요.

[\[이 문서의 내용\]](#)

관리 형식이 있는 람다 식 사용(C++/CLI)

예시

람다 식의 캡처 절에는 관리되는 형식이 있는 변수를 포함할 수 없습니다. 그러나 관리되는 형식이 포함된 인수를 람다 식의 매개 변수 목록으로 전달할 수 있습니다. 다음 예제에서는 관리되지 않는 지역 변수 `ch`를 캡처하는 람다 식을 포함하고 매개 변수로서 `System.String` 개체를 가져옵니다.

C++

```

// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    };
}

```

```
    }("Hello");  
}
```

이 예에서는 다음과 같은 출력을 생성합니다.

Output

Hello!

설명

STL/CLR 라이브러리에서 람다 식을 사용할 수도 있습니다. 자세한 내용은 [STL/CLR 라이브러리 참조](#)를 참조하세요.

ⓘ 중요

람다는 이러한 CLR(공용 언어 런타임) 관리 엔터티인 `ref class`, `ref struct`, `value class` 및 `value struct`에서 지원되지 않습니다.

[[이 문서의 내용](#)]

참고 항목

[람다 식](#)

[람다 식 구문](#)

[auto](#)

[function 클래스](#)

[find_if](#)

[<algorithm>](#)

[함수 호출](#)

[템플릿](#)

[예외 처리](#)

[STL/CLR 라이브러리 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

C++의 constexpr 람다 식

아티클 • 2023. 10. 12.

Visual Studio 2017 버전 15.3 이상 (모드 이상에서 `/std:c++17` 사용 가능): 상수 식 내에서 캡처하거나 도입하는 각 데이터 멤버의 초기화가 허용되는 경우 램다 식을 상수 식으로 `constexpr` 선언하거나 상수 식에서 사용할 수 있습니다.

C++

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

결과가 함수의 `constexpr` 요구 사항을 충족하는 경우 램다는 암시적으로 `constexpr` 발생합니다.

C++

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

람다가 암시적 또는 명시적으로 `constexpr` 램다를 함수 포인터로 변환하면 결과 함수도 `constexpr` 다음과 같습니다.

C++

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

참고 항목

[C++ 언어 참조](#)

[C++ 표준 라이브러리의 함수 개체](#)

[함수 호출](#)

[for_each](#)

배열 (C++)

아티클 • 2023. 10. 12.

배열은 인접한 메모리 영역을 차지하는 동일한 형식의 개체 시퀀스입니다. 기존 C 스타일 배열은 많은 버그의 소스이지만, 특히 이전 코드 베이스에서 여전히 일반적입니다. 최신 C++에서는 이 섹션에 설명된 C 스타일 배열 대신 사용 `std::vector` 하거나 `std::array` 사용하는 것이 좋습니다. 이러한 두 표준 라이브러리 형식은 해당 요소를 연속 메모리 블록으로 저장합니다. 그러나 더 큰 형식 안전성을 제공하고 시퀀스 내에서 유효한 위치를 가리키도록 보장되는 반복기를 지원합니다. 자세한 내용은 컨테이너를 참조 [하세요](#).

스택 선언

C++ 배열 선언에서 배열 크기는 다른 언어와 같이 형식 이름 뒤가 아니라 변수 이름 다음에 지정됩니다. 다음 예제에서는 스택에 할당할 1000 double 배열을 선언합니다. 요소 수는 정수 리터럴 또는 상수 식으로 제공되어야 합니다. 컴파일러가 할당할 스택 공간의 양을 알아야 하기 때문입니다. 런타임에 계산된 값을 사용할 수 없습니다. 배열의 각 요소에는 기본값 0이 할당됩니다. 기본값을 할당하지 않으면 각 요소에는 처음에 해당 메모리 위치에 임의의 값이 포함됩니다.

C++

```
constexpr size_t size = 1000;

// Declare an array of doubles to be allocated on the stack
double numbers[size] {0};

// Assign a new value to the first element
numbers[0] = 1;

// Assign a value to each subsequent element
// (numbers[1] is the second element in the array.)
for (size_t i = 1; i < size; i++)
{
    numbers[i] = numbers[i-1] * 1.1;

}

// Access each element
for (size_t i = 0; i < size; i++)
{
    std::cout << numbers[i] << " ";
}
```

배열의 첫 번째 요소는 0번째 요소입니다. 마지막 요소는 $(n-1)$ 요소입니다. 여기서 n 은 배열에 포함될 수 있는 요소의 수입니다. 선언의 요소 수는 정수 형식이어야 하며 0보다 커

야 합니다. 프로그램에서 보다 큰 `(size - 1)` 아래 첨자 연산자에 값을 전달하지 않도록 하는 것은 사용자의 책임입니다.

크기가 0인 배열은 배열이 마지막 필드 `struct` 이거나 `union` Microsoft 확장이 활성화되거나 `/Za /permissive-` 설정되지 않은 경우에만 유효합니다.

스택 기반 배열은 힙 기반 배열보다 더 빠르게 할당하고 액세스할 수 있습니다. 그러나 스택 공간은 제한됩니다. 배열 요소의 수는 너무 커서 스택 메모리를 너무 많이 사용할 수 없습니다. 얼마나 많은 프로그램에 따라 달라집니다. 프로파일링 도구를 사용하여 배열이 너무 큰지 여부를 확인할 수 있습니다.

힙 선언

스택에 할당하기에는 너무 크거나 컴파일 시간에 크기를 알 수 없는 배열이 필요할 수 있습니다. 식을 사용하여 힙에 이 배열을 할당할 수 있습니다 `new[]`. 연산자는 첫 번째 요소에 대한 포인터를 반환합니다. 아래 첨자 연산자는 스택 기반 배열에서와 동일한 방식으로 포인터 변수에서 작동합니다. 포인터 산술 연산을 사용하여 배열의 임의의 요소로 포인터를 이동할 수도 있습니다. 다음을 보장하는 것은 사용자의 책임입니다.

- 항상 원래 포인터 주소의 복사본을 유지하므로 배열이 더 이상 필요하지 않을 때 메모리를 삭제할 수 있습니다.
- 배열 범위를 지나 포인터 주소를 증가하거나 감소하지 않습니다.

다음 예제에서는 런타임에 힙에 배열을 정의하는 방법을 보여줍니다. 아래 첨자 연산자를 사용하고 포인터 산술 연산을 사용하여 배열 요소에 액세스하는 방법을 보여줍니다.

C++

```
void do_something(size_t size)
{
    // Declare an array of doubles to be allocated on the heap
    double* numbers = new double[size]{ 0 };

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i - 1] * 1.1;
    }

    // Access each element with subscript operator
    for (size_t i = 0; i < size; i++)
    {
        std::cout << numbers[i] << " ";
    }
}
```

```

}

// Access each element with pointer arithmetic
// Use a copy of the pointer for iterating
double* p = numbers;

for (size_t i = 0; i < size; i++)
{
    // Dereference the pointer, then increment it
    std::cout << *p++ << " ";
}

// Alternate method:
// Reset p to numbers[0]:
p = numbers;

// Use address of pointer to compute bounds.
// The compiler computes size as the number
// of elements * (bytes per element).
while (p < (numbers + size))
{
    // Dereference the pointer, then increment it
    std::cout << *p++ << " ";
}

delete[] numbers; // don't forget to do this!

}
int main()
{
    do_something(108);
}

```

배열 초기화

루프, 한 번에 하나의 요소 또는 단일 문으로 배열을 초기화할 수 있습니다. 다음 두 배열의 내용은 동일합니다.

C++

```

int a[10];
for (int i = 0; i < 10; ++i)
{
    a[i] = i + 1;
}

int b[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```

함수에 배열 전달

배열이 함수에 전달되면 스택 기반 배열이든 힙 기반 배열이든 관계없이 첫 번째 요소에 대한 포인터로 전달됩니다. 포인터는 다른 크기 또는 형식 정보를 포함하지 않습니다. 이 동작을 포인터 감쇠라고 합니다. 배열을 함수에 전달할 때는 항상 별도의 매개 변수의 요소 수를 지정해야 합니다. 또한 이 동작은 배열이 함수에 전달될 때 배열 요소가 복사되지 않음을 의미합니다. 함수가 요소를 수정하지 못하도록 하려면 매개 변수를 요소에 대한 포인터 `const` 로 지정합니다.

다음 예제에서는 배열 및 길이를 허용하는 함수를 보여 줍니다. 포인터는 복사본이 아닌 원래 배열을 가리킵니다. 매개 변수가 아니 `const` 므로 함수는 배열 요소를 수정할 수 있습니다.

C++

```
void process(double *p, const size_t len)
{
    std::cout << "process:\n";
    for (size_t i = 0; i < len; ++i)
    {
        // do something with p[i]
    }
}
```

배열 매개 변수 `p` 를 선언하고 정의하여 함수 블록 내에서 읽기 전용으로 `const` 만듭니다.

C++

```
void process(const double *p, const size_t len);
```

동작이 변경되지 않고 이러한 방식으로 동일한 함수를 선언할 수도 있습니다. 배열은 여전히 첫 번째 요소에 대한 포인터로 전달됩니다.

C++

```
// Unsized array
void process(const double p[], const size_t len);

// Fixed-size array. Length must still be specified explicitly.
void process(const double p[1000], const size_t len);
```

다차원 배열

다른 배열에서 생성된 배열은 다차원 배열입니다. 이러한 다차원 배열은 여러 개의 대괄호로 묶인 상수 식을 순서대로 배치하여 지정됩니다. 예를 들어 다음 선언을 생각해 볼 수

있습니다.

C++

```
int i2[5][7];
```

다음 그림과 같이 5개의 행과 7개의 열로 구성된 2차원 행렬에 개념적으로 정렬된 형식 `int` 배열을 지정합니다.

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

이미지는 너비가 7 셀이고 세로가 5개인 그리드입니다. 각 셀에는 셀의 인덱스가 포함됩니다. 첫 번째 셀 인덱스는 0,0으로 레이블이 지정됩니다. 해당 행의 다음 셀은 0,6인 해당 행의 마지막 셀까지 0,1입니다. 다음 행은 인덱스 1,0으로 시작합니다. 그 뒤의 셀의 인덱스는 1,1입니다. 해당 행의 마지막 셀은 1,6입니다. 이 패턴은 인덱스 4,0으로 시작하는 마지막 행까지 반복됩니다. 마지막 행의 마지막 셀의 인덱스는 4,6입니다. ::image-end

이니셜라이저에 설명된 대로 이니셜라이저 목록이 있는 다차원 배열을 [선언할 수 있습니다](#). 이러한 선언에서 첫 번째 차원의 경계를 지정하는 상수 식을 생략할 수 있습니다. 예시:

C++

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][][cMarkets] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};
```

앞의 선언은 세 개의 행과 네 개의 열로 구성된 배열을 정의합니다. 행은 공장을 나타내고 열은 공장에서 출하되는 시장을 나타냅니다. 값은 공장에서 시장까지의 운송 비용입니다. 배열의 첫 번째 차원은 생략되었지만 컴파일러가 이니셜라이저를 검사하여 해당 차원을 채웁니다.

n차원 배열 형식에서 간접 연산자(*)를 사용하면 n-1 차원 배열이 생성됩니다. n이 10이면 스칼라(또는 배열 요소)가 생성됩니다.

C++ 배열은 행 중심 순서로 저장됩니다. 행 중심 순서는 마지막 첨자가 가장 빠르게 변경됨을 의미합니다.

예시

다음과 같이 함수 선언에서 다차원 배열의 첫 번째 차원에 대한 경계 사양을 생략할 수도 있습니다.

C++

```
// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>    // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][cMkts] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
    sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int
mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
    double MinCost;

    if ( argv[1] == 0) {
        cout << "You must specify the number of markets." << endl;
        exit(0);
    }
    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
    cout << "The minimum cost to Market " << argv[1] << " is: "
        << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts)
{
    double MinCost = DBL_MAX;

    for( size_t i = 0; i < cFacts; ++i )
```

```

        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
        MinCost : TransportCosts[i][Mkt];

    return MinCost;
}

```

Output

```
The minimum cost to Market 3 is: 17.29
```

이 함수 `FindMinToMkt` 는 새 팩터리를 추가해도 코드 변경이 필요하지 않고 다시 컴파일 하기만 하면 되도록 작성됩니다.

배열 초기화

클래스 생성자가 있는 개체의 배열은 생성자에 의해 초기화됩니다. 배열의 요소보다 이니셜라이저 목록에 항목이 적으면 기본 생성자가 다시 기본 요소에 사용됩니다. 클래스에 대해 정의된 기본 생성자가 없는 경우 이니셜라이저 목록이 완료되어야 합니다. 즉, 배열의 각 요소에 대해 하나의 이니셜라이저가 있어야 합니다.

생성자 두 개를 정의하는 `Point` 클래스를 살펴보세요.

C++

```

// initializing_arrays1.cpp
class Point
{
public:
    Point()    // Default constructor.
    {
    }
    Point( int, int )    // Construct from two ints
    {
    }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
    Point( 3, 3 )    // Use int, int constructor.
};

int main()
{
}

```

`aPoint`의 첫 번째 요소는 `Point(int, int)` 생성자를 사용하여 생성되고 나머지 두 요소는 기본 생성자를 사용하여 생성됩니다.

정적 멤버 배열(여부 `const`)은 클래스 선언 외부의 정의에서 초기화할 수 있습니다. 예시:

```
C++  
  
// initializing_arrays2.cpp  
class WindowColors  
{  
public:  
    static const char *rgszWindowPartList[7];  
};  
  
const char *WindowColors::rgszWindowPartList[7] = {  
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",  
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame" };  
int main()  
{  
}
```

배열 요소 액세스

배열 첨자 연산자(`[]`)를 사용하여 배열의 개별 요소에 액세스할 수 있습니다. 아래 첨자 없이 1차원 배열의 이름을 사용하는 경우 배열의 첫 번째 요소에 대한 포인터로 평가됩니다.

```
C++  
  
// using_arrays.cpp  
int main() {  
    char chArray[10];  
    char *pch = chArray; // Evaluates to a pointer to the first element.  
    char ch = chArray[0]; // Evaluates to the value of the first element.  
    ch = chArray[3]; // Evaluates to the value of the fourth element.  
}
```

다차원 배열을 사용할 때 식에서 다양한 조합을 사용할 수 있습니다.

```
C++  
  
// using_arrays_2.cpp  
// compile with: /EHsc /W1  
#include <iostream>  
using namespace std;  
int main() {  
    double multi[4][4][3]; // Declare the array.  
    double (*p2multi)[3];  
    double (*p1multi);
```

```

cout << multi[3][2][2] << "\n"; // C4700 Use three subscripts.
p2multi = multi[3]; // Make p2multi point to
                     // fourth "plane" of multi.
p1multi = multi[3][2]; // Make p1multi point to
                     // fourth plane, third row
                     // of multi.
}

```

앞의 코드 `multi` 에서 형식 `double` 의 3차원 배열입니다. 포인터는 `p2multi` 크기 3 형식 `double` 의 배열을 가리킵니다. 이 예제에서 배열은 한 개, 두 개 및 세 개의 아래 첨자와 함께 사용됩니다. 문과 `cout` 같이 모든 아래 첨자를 지정하는 것이 더 일반적이지만 다음 문에 표시된 것처럼 배열 요소의 특정 하위 집합을 선택하는 것이 유용할 수 있습니다 `cout`.

아래 첨자 연산자 오버로드

다른 연산자처럼 사용자가 아래 첨자 연산자(`[]`)를 다시 정의할 수 있습니다. 첨자 연산자의 기본 동작은 다음 메서드를 사용하여 배열 이름과 첨자를 결합하는 것입니다.

```
*((array_name) + (subscript))
```

포인터 형식을 포함하는 모든 항목과 마찬가지로 크기 조정은 형식의 크기에 맞게 조정되도록 자동으로 수행됩니다. 결과 값은 원본의 n바이트가 아니라 배열의 *n*번째 요소입니다. `array_name` 이 변환에 대한 자세한 내용은 추가 연산자를 참조 [하세요](#).

다차원 배열에서도 다음 메서드를 사용하여 주소가 파생됩니다.

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ...
* maxn) + ... + subscriptn))
```

식의 배열

배열 형식의 식별자가 참조의 주소(&) 또는 초기화 이외의 식 `sizeof`에 나타나면 첫 번째 배열 요소에 대한 포인터로 변환됩니다. 예시:

C++

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

`psz` 포인터는 `szError1` 배열의 첫 번째 요소를 가리킵니다. 배열은 포인터와 달리 수정 가능한 I-값이 아닙니다. 따라서 다음 할당이 불법입니다.

C++

```
szError1 = psz;
```

참고 항목

[std::array](#)

참조 (C++)

아티클 • 2024. 07. 08.

참조는 포인터와 마찬가지로, 메모리의 다른 위치에 있는 개체의 주소를 저장합니다. 하지만 포인터와는 달리, 참조는 초기화되고 나면 다른 개체를 참조하도록 설정하거나 null로 설정할 수 없습니다. 명명된 변수를 참조하는 *lvalue* 참조와 임시 개체를 참조하는 *rvalue* 참조 등, 두 가지 종류의 참조가 있습니다. `&` 연산자는 lvalue 참조를 나타내고, `&&` 연산자는 컨텍스트에 따라 rvalue 참조나 범용 참조(rvalue 또는 lvalue)를 나타냅니다.

참조를 선언할 수 있는 구문은 다음과 같습니다.

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [ms-modifier] declarator [= expression];
```

참조를 지정하는 임의의 유효한 선언자를 사용할 수 있습니다. 참조가 함수 또는 배열 형식에 대한 참조가 아닌 한 다음 단순화된 구문이 적용됩니다.

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [& 또는 &&] [cv-qualifiers] identifier [= expression];
```

참조를 선언하는 시퀀스는 다음과 같습니다.

1. 선언 지정자:

- 선택적 스토리지 클래스 지정자.
- 선택적 `const` 및/또는 `volatile` 한정자입니다.
- 형식 지정자: 형식의 이름

2. 선언자:

- 선택적 Microsoft 전용 한정자. 자세한 내용은 [Microsoft 전용 한정자](#)를 참조하세요.
- `&` 연산자 또는 `&&` 연산자입니다.
- 선택적 `const` 및/또는 `volatile` 한정자입니다.
- 식별자입니다.

3. 선택적 이니셜라이저입니다.

배열 및 함수에 대한 포인터의 더 복잡한 선언자 형태도 배열 및 함수에 대한 참조에 적용됩니다. 자세한 내용은 [포인터](#)를 참조하세요.

여러 선언자와 이니셜라이저가 단일 선언 지정자 뒤에 쉼표로 구분된 목록으로 나타날 수 있습니다. 예시:

C++

```
int &i;  
int &i, &j;
```

참조, 포인터 및 개체를 다음과 같이 함께 선언할 수 있습니다.

C++

```
int &ref, *ptr, k;
```

참조는 개체의 주소를 보유하지만 구문적으로 개체처럼 동작합니다.

다음 프로그램에서 개체 이름인 `s`와 개체에 대한 참조인 `SRef`를 프로그램에서 동일하게 사용할 수 있습니다.

예시

C++

```
// references.cpp  
#include <stdio.h>  
struct S {  
    short i;  
};  
  
int main() {  
    S s;      // Declare the object.  
    S& SRef = s;    // Declare and initialize the reference.  
    s.i = 3;  
  
    printf_s("%d\n", s.i);  
    printf_s("%d\n", SRef.i);  
  
    SRef.i = 4;  
    printf_s("%d\n", s.i);  
    printf_s("%d\n", SRef.i);  
}
```

Output

```
3  
3
```

참고 항목

[참조 형식 함수 인수](#)

[참조 형식 함수 반환](#)

[포인터에 대한 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Lvalue 참조 선언자: &

아티클 • 2024. 11. 21.

개체의 주소를 보유하지만 구문상 개체처럼 동작합니다.

구문

:

type-specifier-seq & *attribute-specifier-seq* _{opt} *ptr-abstract-declarator* _{opt}

설명

lvalue 참조를 개체의 또 다른 이름으로 간주할 수 있습니다. lvalue 참조 선언은 참조 선언자가 뒤에 나오는 선택적 지정자 목록으로 구성됩니다. 참조는 초기화되어야 하고 변경될 수 없습니다.

주소가 지정된 포인터 형식으로 변환될 수 있는 개체는 유사한 참조 형식으로도 변환될 수 있습니다. 예를 들어 주소가 `char *` 형식으로 변환될 수 있는 개체는 `char &` 형식으로도 변환될 수 있습니다.

참조 선언을 [address-of 연산자](#)를 사용하는 것과 혼동하지 마세요. `&identifier` 앞에 `int` 또는 `char`와 같은 형식이 있는 경우 `identifier`는 해당 형식에 대한 참조로 선언됩니다. `&identifier` 앞에 형식이 없는 경우에는 address-of 연산자와 용도가 같습니다.

예시

다음 예제에서는 `Person` 개체 및 해당 개체에 대한 참조를 선언하여 참조 선언자를 보여 줍니다. `rFriend`가 `myFriend`에 대한 참조이기 때문에 변수 중 하나를 업데이트하면 동일한 개체가 변경됩니다.

C++

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
{
    char* Name;
    short Age;
```

```
};

int main()
{
    // Declare a Person object.
    Person myFriend;

    // Declare a reference to the Person object.
    Person& rFriend = myFriend;

    // Set the fields of the Person object.
    // Updating either variable changes the same object.
    myFriend.Name = "Bill";
    rFriend.Age = 40;

    // Print the fields of the Person object to the console.
    cout << rFriend.Name << " is " << myFriend.Age << endl;
}
```

Output

```
Bill is 40
```

참고 항목

참조

[참조 형식 함수 인수](#)

[참조 형식 함수 반환](#)

[포인터에 대한 참조](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Rvalue 참조 선언자: &&

아티클 • 2024. 07. 08.

rvalue 식에 대한 참조를 보유합니다.

구문

`rvalue-reference-type-id:`

`type-specifier-seq && attribute-specifier-seqopt ptr-abstract-declaratoropt`

설명

rvalue 참조를 사용하면 lvalue와 rvalue를 구별할 수 있습니다. lvalue 참조와 rvalue 참조는 구문 및 의미 체계가 비슷하지만 약간 다른 규칙을 따릅니다. lvalue 및 rvalue에 대한 자세한 내용은 [Lvalue 및 Rvalue](#)를 참조하십시오. lvalue 참조에 대한 자세한 내용은 [Lvalue 참조 선언자: &](#)를 참조하세요.

다음 단원에서는 Rvalue 참조가 *이동 의미 체계* 및 *완벽한 전달*의 구현을 지원하는 방법을 설명합니다.

이동 의미 체계

Rvalue 참조는 *이동 의미 체계*의 구현을 지원하여 애플리케이션의 성능을 크게 높일 수 있습니다. 의미 체계 이동을 사용하여 한 개체에서 다른 개체로 리소스(예: 동적으로 할당된 메모리)를 전송하는 코드를 작성할 수 있습니다. 이동 의미 체계는 리소스가 프로그램의 다른 곳에서 참조될 수 없는 임시 개체에서 전송될 수 있도록 하기 때문에 효과적입니다.

이동 의미 체계를 구현하려면 일반적으로 *이동 생성자*와 필요에 따라 이동 할당 연산자 (`operator=`)를 클래스에 제공합니다. 이렇게 하면 해당 소스가 rvalue인 복사 및 할당 작업에서 자동으로 의미 체계 이동을 활용합니다. 기본 복사 생성자와 달리, 컴파일러는 기본 이동 생성자를 제공하지 않습니다. 이동 생성자를 작성하고 사용하는 방법에 대한 자세한 내용은 [이동 생성자 및 이동 할당 연산자](#)를 참조하세요.

일반 함수 및 연산자를 오버로드하여 의미 체계 이동을 활용할 수도 있습니다. Visual Studio 2010에서는 C++ 표준 라이브러리에 이동 의미 체계를 도입했습니다. 예를 들어 `string` 클래스는 이동 의미 체계를 사용하는 작업을 구현합니다. 여러 문자열을 연결하고 결과를 출력하는 다음 예제를 살펴보십시오.

```

// string_concatenation.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = string("h") + "e" + "ll" + "o";
    cout << s << endl;
}

```

Visual Studio 2010 이전에는 `operator+`를 호출할 때마다 새 임시 `string` 객체(rvalue)가 할당되고 반환됩니다. `operator+`는 소스 문자열이 lvalue인지 아니면 rvalue인지를 알 수 없으므로 한 문자열을 다른 문자열에 추가할 수 없습니다. 소스 문자열이 둘 다 lvalue인 경우 프로그램의 다른 곳에서 참조될 수 있으므로 수정되지 않아야 합니다. rvalue 참조를 사용하여 프로그램의 다른 곳에서 참조될 수 없는 rvalue를 사용하도록 `operator+`를 수정할 수 있습니다. 이렇게 변경되었으므로 `operator+`는 이제 한 문자열을 다른 문자열에 추가할 수 있습니다. 이에 따라 `string` 클래스가 수행해야 하는 동적 메모리 할당 수가 크게 줄어들 수 있습니다. `string` 클래스에 대한 자세한 내용은 [basic_string 클래스](#)를 참조하세요.

이동 의미 체계는 컴파일러에서 RVO(반환 값 최적화) 또는 NRVO(명명된 반환 값 최적화)를 사용할 수 없는 경우에도 도움이 됩니다. 이러한 경우에 컴파일러는 형식에 정의된 이동 생성자를 호출합니다.

의미 체계 이동에 대해 자세히 이해하려면 요소를 `vector` 객체에 삽입하는 예를 참조하십시오. `vector` 객체의 용량이 초과될 경우 `vector` 객체는 해당 요소를 위해 충분한 메모리를 재할당한 다음, 각 요소를 다른 메모리 위치로 복사하여 삽입된 요소를 위한 공간을 만들어야 합니다. 삽입 작업은 요소를 복사할 때 먼저 새 요소를 만듭니다. 그런 다음, 복사 생성자를 호출하여 이전 요소에서 새 요소로 데이터를 복사합니다. 마지막으로 이전 요소를 삭제합니다. 이 동의미 체계를 사용하면 비용이 많이 드는 메모리 할당 및 복사 작업을 수행할 필요 없이 개체를 직접 이동할 수 있습니다.

`vector` 예에서 의미 체계 이동을 활용하기 위해 한 개체에서 다른 개체로 데이터를 이동하는 이동 생성자를 작성할 수 있습니다.

Visual Studio 2010에서 C++ 표준 라이브러리로 이동 의미 체계를 도입하는 방법에 대한 자세한 내용은 [C++ 표준 라이브러리](#)를 참조하세요.

완벽한 전달

완벽한 전달은 오버로드된 함수의 필요성을 줄이고 전달 문제를 방지하는 데 도움이 됩니다. 전달 문제는 참조를 매개 변수로 사용하는 제네릭 함수를 작성할 때 발생할 수 있습니다. 예를 들어 `const T&` 형식의 매개 변수를 사용하는 경우와 같이 이러한 매개 변수를 다른 함수에 전달하면 호출된 함수는 해당 매개 변수의 값을 수정할 수 없습니다. 제네릭 함수가 `T&` 형식의 매개 변수를 사용하는 경우에는 rvalue(예: 임시 개체 또는 정수 리터럴)를 사용하여 해당 함수를 호출할 수 없습니다.

일반적으로 이 문제를 해결하려면 각 매개 변수에 대해 `T&` 및 `const T&`를 둘 다 사용하는 제네릭 함수의 오버로드된 버전을 제공해야 합니다. 따라서 오버로드된 함수의 수가 매개 변수의 수와 함께 급격하게 증가합니다. rvalue 참조를 사용하면 임의의 인수를 허용하는 함수의 한 버전을 작성할 수 있습니다. 그러면 해당 함수는 다른 함수가 직접 호출된 경우처럼 해당 함수로 이 인수를 전달할 수 있습니다.

`W`, `X`, `Y` 및 `Z`의 4가지 형식을 선언하는 다음 예제를 살펴보십시오. 각 형식에 대한 생성자는 `const` 및 비 `const` lvalue 참조의 다른 조합을 매개 변수로 사용합니다.

```
C++  
  
struct W  
{  
    W(int&, int&) {}  
};  
  
struct X  
{  
    X(const int&, int&) {}  
};  
  
struct Y  
{  
    Y(int&, const int&) {}  
};  
  
struct Z  
{  
    Z(const int&, const int&) {}  
};
```

개체를 생성하는 제네릭 함수를 작성한다고 가정합니다. 다음 예제에서는 이 함수를 작성하는 한 가지 방법을 보여 줍니다.

```
C++  
  
template <typename T, typename A1, typename A2>  
T* factory(A1& a1, A2& a2)  
{
```

```
    return new T(a1, a2);
}
```

다음 예제에서는 `factory` 함수에 대한 유효한 호출을 보여 줍니다.

C++

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

그렇지만 다음 예제에서는 `factory` 함수에 대한 유효한 호출을 포함하지 않습니다. 이것은 `factory` 가 수정할 수 있는 lvalue 참조를 해당 매개 변수로 사용하지만 rvalue를 사용하여 호출되기 때문입니다.

C++

```
Z* pz = factory<Z>(2, 2);
```

일반적으로 이 문제를 해결하려면 `factory` 및 `A&` 매개 변수의 모든 조합에 대해 `const A&` 함수의 오버로드된 버전을 만들어야 합니다. rvalue 참조를 사용하면 다음 예제와 같이 `factory` 함수의 버전을 작성할 수 있습니다.

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

이 예제에서는 rvalue 참조를 `factory` 함수에 대한 매개 변수로 사용합니다. `std::forward` 함수는 템플릿 클래스의 생성자에 `factory` 함수의 매개 변수를 전달하기 위한 것입니다.

다음 예제에서는 `main`, `factory`, `W` 및 `X` 클래스의 인스턴스를 만들기 위해 수정된 `Y` 함수를 사용하는 `z` 함수를 보여 줍니다. 수정된 `factory` 함수는 매개 변수(lvalue 또는 rvalue)를 적절한 클래스 생성자에 전달합니다.

C++

```
int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);
    X* px = factory<X>(2, b);
    Y* py = factory<Y>(a, 2);
```

```
Z* pz = factory<Z>(2, 2);

    delete pw;
    delete px;
    delete py;
    delete pz;
}
```

rvalue 참조의 속성

Ivalue 참조 및 rvalue 참조를 사용하는 함수를 오버로드할 수 있습니다.

`const` Ivalue 참조 또는 rvalue 참조를 사용하도록 함수를 오버로드함으로써 수정할 수 없는 개체(Ivalue)와 수정할 수 있는 임시 값(rvalue)을 구별하는 코드를 작성할 수 있습니다. 개체가 `const`로 표시되지 않은 경우 개체를 rvalue 참조를 사용하는 함수에 전달할 수 있습니다. 다음 예제에서는 Ivalue 참조 및 rvalue 참조를 사용하도록 오버로드되는 `f` 함수를 보여 줍니다. `main` 함수는 Ivalue와 rvalue를 둘 다 사용하여 `f`를 호출합니다.

C++

```
// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version can't modify the
parameter." << endl;
}

void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." <<
endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}
```

이 예제는 다음과 같은 출력을 생성합니다.

Output

```
In f(const MemoryBlock&). This version can't modify the parameter.  
In f(MemoryBlock&&). This version can modify the parameter.
```

이 예제에서 첫 번째 `f` 호출은 지역 변수(lvalue)를 인수로 전달하고, 두 번째 `f` 호출은 임시 개체를 인수로 전달합니다. 임시 개체는 프로그램의 다른 곳에서 참조될 수 없기 때문에 호출은 개체를 언제든지 수정할 수 있는 rvalue 참조를 사용하는 `f`의 오버로드된 버전에 바인딩됩니다.

컴파일러는 명명된 rvalue 참조를 lvalue로 처리하고 명명되지 않은 rvalue 참조를 rvalue로 처리합니다.

rvalue 참조를 매개 변수로 사용하는 함수는 이 매개 변수를 함수 본문의 lvalue로 처리합니다. 컴파일러는 명명된 rvalue 참조를 lvalue로 처리합니다. 이것은 프로그램의 여러 부분에서 명명된 개체를 참조할 수 있기 때문입니다. 프로그램의 여러 부분이 해당 개체에서 리소스를 수정하거나 제거하도록 허용하는 것은 위험합니다. 예를 들어 프로그램의 여러 부분이 동일한 개체에서 리소스를 전송하려고 하면 첫 번째 부분만 성공합니다.

다음 예제에서는 lvalue 참조 및 rvalue 참조를 사용하도록 오버로드되는 `g` 함수를 보여 줍니다. `f` 함수는 rvalue 참조(명명된 rvalue 참조)를 매개 변수로 사용하고 rvalue 참조(명명되지 않은 rvalue 참조)를 반환합니다. `g`에서 `f`를 호출할 때 `g` 본문에서 매개 변수를 lvalue로 처리하기 때문에 오버로드 확인은 lvalue 참조를 사용하는 `f`의 버전을 선택합니다. `g`에서 `main`을 호출할 때 `g`에서 rvalue 참조를 반환하기 때문에 오버로드 확인은 rvalue 참조를 사용하는 `f`의 버전을 선택합니다.

C++

```
// named-reference.cpp  
// Compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
// A class that contains a memory resource.  
class MemoryBlock  
{  
    // TODO: Add resources for the class here.  
};  
  
void g(const MemoryBlock&)  
{  
    cout << "In g(const MemoryBlock&)." << endl;  
}  
  
void g(MemoryBlock&&)
```

```

{
    cout << "In g(MemoryBlock&&)." << endl;
}

MemoryBlock&& f(MemoryBlock&& block)
{
    g(block);
    return move(block);
}

int main()
{
    g(f(MemoryBlock()));
}

```

이 예제는 다음과 같은 출력을 생성합니다.

C++

```

In g(const MemoryBlock&).
In g(MemoryBlock&&).

```

이 예제에서 `main` 함수는 rvalue를 `f`에 전달합니다. `f`의 본문에서는 명명된 매개 변수를 lvalue로 처리합니다. `f`에서 `g`로의 호출은 매개 변수를 lvalue 참조(`g`의 첫 번째 오버로드된 버전)에 바인딩합니다.

- lvalue를 rvalue 참조로 캐스팅할 수 있습니다.

C++ 표준 라이브러리 `std::move` 함수를 사용하여 개체를 해당 개체에 대한 rvalue 참조로 변환할 수 있습니다. 또는 `static_cast` 키워드를 사용하여 다음 예제와 같이 lvalue를 rvalue 참조로 캐스팅할 수 있습니다.

C++

```

// cast-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

```

```

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
    MemoryBlock block;
    g(block);
    g(static_cast<MemoryBlock&&>(block));
}

```

이 예제는 다음과 같은 출력을 생성합니다.

C++

```

In g(const MemoryBlock&).
In g(MemoryBlock&&).

```

함수 템플릿은 해당 템플릿 인수 형식을 추론한 다음 참조 축소 규칙을 사용합니다.

해당 매개 변수를 다른 함수에 전달하는 함수 템플릿을 작성하는 것은 일반적인 패턴입니다. 템플릿 형식 추론이 rvalue 참조를 사용하는 함수 템플릿에 대해 어떻게 작용하는지를 이해해야 합니다.

함수 인수가 rvalue이면 컴파일러는 인수를 rvalue 참조로 추론합니다. 예를 들어 `T&&` 형식을 매개 변수로 사용하는 템플릿 함수에 `x` 형식의 개체에 대한 rvalue 참조를 전달한다고 가정해 보겠습니다. 템플릿 인수 추론에서는 `T`가 `x`인 것으로 추론하므로 매개 변수 형식은 `x&&`입니다. 함수 인수가 lvalue 또는 `const` lvalue인 경우 컴파일러는 인수의 형식을 해당 형식의 lvalue 참조 또는 `const` lvalue 참조로 추론합니다.

다음 예제에서는 하나의 구조체 템플릿을 선언한 다음 다양한 참조 형식에 대해 특수화합니다. `print_type_and_value` 함수는 rvalue 참조를 매개 변수로 사용하고, 이를 `s::print` 메서드의 적절한 특수화된 버전에 전달합니다. `main` 함수에서는 `s::print` 메서드를 호출하는 다양한 방법을 보여 줍니다.

C++

```

// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

template<typename T> struct S;

// The following structures specialize S by

```

```

// lvalue reference (T&), const lvalue reference (const T&),
// rvalue reference (T&&), and const rvalue reference (const T&&).
// Each structure provides a print method that prints the type of
// the structure and its parameter.

template<typename T> struct S<T&> {
    static void print(T& t)
    {
        cout << "print<T&>: " << t << endl;
    }
};

template<typename T> struct S<const T&> {
    static void print(const T& t)
    {
        cout << "print<const T&>: " << t << endl;
    }
};

template<typename T> struct S<T&&> {
    static void print(T&& t)
    {
        cout << "print<T&&>: " << t << endl;
    }
};

template<typename T> struct S<const T&&> {
    static void print(const T&& t)
    {
        cout << "print<const T&&>: " << t << endl;
    }
};

// This function forwards its parameter to a specialized
// version of the S type.
template <typename T> void print_type_and_value(T&& t)
{
    S<T&&>::print(std::forward<T>(t));
}

// This function returns the constant string "fourth".
const string fourth() { return string("fourth"); }

int main()
{
    // The following call resolves to:
    // print_type_and_value<string&>(string& && t)
    // Which collapses to:
    // print_type_and_value<string&>(string& t)
    string s1("first");
    print_type_and_value(s1);

    // The following call resolves to:
    // print_type_and_value<const string&>(const string& && t)
    // Which collapses to:

```

```

// print_type_and_value<const string&>(const string& t)
const string s2("second");
print_type_and_value(s2);

// The following call resolves to:
// print_type_and_value<string&&>(string&& t)
print_type_and_value(string("third"));

// The following call resolves to:
// print_type_and_value<const string&&>(const string&& t)
print_type_and_value(fourth());
}

```

이 예제는 다음과 같은 출력을 생성합니다.

C++

```

print<T&>: first
print<const T&>: second
print<T&&>: third
print<const T&&>: fourth

```

`print_type_and_value` 함수에 대한 각 호출을 확인하기 위해 컴파일러는 먼저 템플릿 인수 추론을 수행합니다. 그런 다음, 컴파일러는 매개 변수 형식을 추론된 템플릿 인수로 대체할 때 참조 축소 규칙을 적용합니다. 예를 들어 지역 변수 `s1`을 `print_type_and_value` 함수에 전달하면 컴파일러에서 다음과 같은 함수 시그니처를 생성합니다.

C++

```
print_type_and_value<string&>(string& && t)
```

컴파일러는 참조 축소 규칙을 사용하여 시그니처를 다음과 같이 줄입니다.

C++

```
print_type_and_value<string&>(string& t)
```

이 `print_type_and_value` 함수 버전은 매개 변수를 `s::print` 메서드의 올바른 특수화된 버전에 전달합니다.

다음 표에는 템플릿 인수 형식 추론에 대한 참조 축소 규칙이 요약되어 있습니다.

[] 테이블 확장

확장된 형식	축소된 형식
T& &	T&
T& &&	T&
T&& &	T&
T&& &&	T&&

템플릿 인수 추론은 완벽한 전달을 구현하는 중요한 요소입니다. [완벽한 전달](#) 섹션에서 완벽한 전달에 대해 좀 더 자세히 설명합니다.

요약

rvalue 참조는 rvalue와 lvalue를 구별합니다. 애플리케이션의 성능을 높이기 위해 불필요한 메모리 할당 및 복사 작업의 필요성을 없앨 수 있습니다. 또한 이를 통해 임의의 인수를 허용하는 함수를 작성할 수 있습니다. 그러면 해당 함수는 다른 함수가 직접 호출된 경우처럼 해당 함수로 이 인수를 전달할 수 있습니다.

참고 항목

[단항 연산자가 있는 식](#)

[Lvalue 참조 선언자: &](#)

[Lvalue 및 rvalue](#)

[이동 생성자 및 이동 할당 연산자\(C++\)](#)

[C++ 표준 라이브러리](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

참조 형식 함수 인수

아티클 • 2024. 01. 10.

보통은 큰 개체보다 참조를 함수에 전달하는 것이 더 효율적입니다. 이렇게 하면 개체에 액세스하는 데 사용된 구문을 유지하면서 컴파일러가 개체의 주소를 전달할 수 있습니다.

Date 구조체를 사용하는 다음 예제를 살펴보십시오.

C++

```
// reference_type_function_arguments.cpp
#include <iostream>

struct Date
{
    short Month;
    short Day;
    short Year;
};

// Create a date of the form DDDYYYY (day of year, year)
// from a Date.
long DateOfYear( Date& date )
{
    static int cDaysInMonth[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long dateOfYear = 0;

    // Add in days for months already elapsed.
    for ( int i = 0; i < date.Month - 1; ++i )
        dateOfYear += cDaysInMonth[i];

    // Add in days for this month.
    dateOfYear += date.Day;

    // Check for leap year.
    if ( date.Month > 2 &&
        ( ( date.Year % 100 != 0 || date.Year % 400 == 0 ) &&
        date.Year % 4 == 0 ) )
        dateOfYear++;

    // Add in year.
    dateOfYear *= 10000;
    dateOfYear += date.Year;

    return dateOfYear;
}

int main()
{
    Date date{ 8, 27, 2018 };
}
```

```
    long dateOfYear = DateOfYear(date);
    std::cout << dateOfYear << std::endl;
}
```

앞의 코드는 참조로 전달된 구조체의 멤버가 포인터 멤버 선택 연산자(->) 대신 멤버 선택 연산자(.)를 사용하여 액세스됨을 보여 줍니다.

참조 형식으로 전달된 인수는 포인터가 아닌 `const` 형식의 구문을 관찰하지만 포인터 형식의 한 가지 중요한 특징을 유지합니다. 이 코드에는 `date` 개체를 수정할 의도가 없기 때문에 더 적절한 함수 프로토타입은 다음과 같습니다.

C++

```
long DateOfYear( const Date& date );
```

이 프로토타입은 `DateOfYear` 함수가 인수를 변경하지 않도록 합니다.

참조 형식을 사용하는 것으로 프로토타입화된 함수는 형식 이름에서 *typename*으로 표준 변환이 있기 때문에 해당 위치에 동일한 형식의 개체를 허용할 수 있습니다.

참고 항목

[참조](#)

참조 형식 함수 반환

아티클 • 2023. 10. 12.

참조 형식을 반환하도록 함수를 선언할 수 있습니다. 이렇게 선언하는 데에는 다음과 같은 두 가지 이유가 있습니다.

- 반환되는 정보가 충분히 큰 개체여서 복사본 대신 참조를 반환하는 것이 더 효율적입니다.
- 함수의 형식이 l-value여야 합니다.
- 함수가 반환할 때 참조 개체는 범위를 벗어나지 않습니다.

참조로 함수에 큰 개체를 전달하는 것이 더 효율적일 수 있는 것처럼 참조로 함수에서 큰 개체를 반환하는 것이 더 효율적일 수도 있습니다. 참조 반환 프로토콜을 사용하면 반환하기 전에 개체를 임시 위치로 복사할 필요가 없어집니다.

함수가 l-value로 계산되어야 하는 경우에도 참조 반환 형식이 유용할 수 있습니다. 대부분의 오버로드된 연산자, 특히 대입 연산자가 이 범주에 속합니다. 오버로드된 연산자는 오버로드된 연산자[자에서](#) 다룹니다.

예시

Point 예제를 참조하세요.

C++

```
// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();
private:
    // Note that these are declared at class scope:
    unsigned obj_x;
    unsigned obj_y;
};

unsigned& Point :: x()
```

```

{
    return obj_x;
}
unsigned& Point :: y()
{
    return obj_y;
}

int main()
{
    Point ThePoint;
    // Use x() and y() as l-values.
    ThePoint.x() = 7;
    ThePoint.y() = 9;

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
    << "y = " << ThePoint.y() << "\n";
}

```

출력

Output

```
x = 7
y = 9
```

`x` 및 `y` 함수가 참조 형식을 반환하는 것으로 선언된 것을 볼 수 있습니다. 대입 문의 양쪽에 이러한 함수를 사용할 수 있습니다.

또한, `main`에서 `ThePoint` 객체가 범위 내에 있으므로 해당 참조 멤버가 여전히 활성 상태이며 이 멤버에 안전하게 액세스할 수 있습니다.

참조 형식의 선언에는 다음 경우를 제외하고 이니셜라이저를 포함해야 합니다.

- 명시적 `extern` 선언
- 클래스 멤버의 선언
- 클래스 내의 선언
- 함수의 인수 또는 함수의 반환 형식 선언

로컬의 주소 반환 시 주의 사항

로컬 범위에서 개체를 선언하는 경우 함수가 반환할 때 이 개체는 제거됩니다. 함수가 이 개체에 대한 참조를 반환하는 경우 호출자가 null 참조를 사용하려고 하면 해당 참조는 런타임에 액세스 위반을 일으킬 수 있습니다.

C++

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

컴파일러는 다음과 같은 경우 `warning C4172: returning address of local variable or temporary` 경고를 실행합니다. 간단한 프로그램에서는 메모리 위치를 덮어쓰기 전에 호출자가 참조에 액세스하는 경우 일반적으로 액세스 위반이 발생하지 않습니다. 이 경우는 순전히 운이 작용한 것입니다. 경고에 주의하세요.

참고 항목

[참조](#)

포인터에 대한 참조

아티클 • 2023. 10. 12.

포인터에 대한 참조는 개체에 대한 참조와 거의 같은 방법으로 선언할 수 있습니다. 포인터에 대한 참조는 일반 포인터처럼 사용되는 수정 가능한 값입니다.

예시

이 코드 샘플에서는 포인터에 대한 포인터와 포인터에 대한 참조를 사용하는 것 사이의 차이점을 보여줍니다.

함수는 `Add1` `Add2` 동일한 방식으로 호출되지는 않지만 함수와 기능적으로 동일합니다. 차이점은 이중 간접 참조를 사용하지만 `Add2` 포인터에 대한 참조의 편의를 사용한다는 `Add1` 것입니다.

C++

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library namespace
using namespace std;

enum {
    sizeOfBuffer = 132
};

// Define a binary tree structure.
struct BTree {
    char *szText;
    BTree *Left;
    BTree *Right;
};

// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

int main( int argc, char *argv[] ) {
    // Usage message
    if( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " [1 | 2]" << "\n";
    }
}
```

```

        cerr << "\nwhere:\n";
        cerr << "1 uses double indirection\n";
        cerr << "2 uses a reference to a pointer.\n";
        cerr << "\nInput is from stdin. Use ^Z to terminate input.\n";
        return 1;
    }

    char *szBuf = new char[sizeOfBuffer];
    if (szBuf == NULL) {
        cerr << "Out of memory!\n";
        return -1;
    }

    // Read a text file from the standard input device and
    // build a binary tree.
    while( !cin.eof() )
    {
        cin.get( szBuf, sizeOfBuffer, '\n' );
        cin.get();

        if ( strlen( szBuf ) ) {
            switch ( *argv[1] ) {
                // Method 1: Use double indirection.
                case '1':
                    Add1( &btRoot, szBuf );
                    break;
                // Method 2: Use reference to a pointer.
                case '2':
                    Add2( btRoot, szBuf );
                    break;
                default:
                    cerr << "Illegal value '"
                        << *argv[1]
                        << "' supplied for add method.\n"
                        << "Choose 1 or 2.\n";
                    return -1;
            }
        }
    }

    // Display the sorted list.
    PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTTree* MybtRoot ) {
    // Traverse the left branch of the tree recursively.
    if ( MybtRoot->Left )
        PrintTree( MybtRoot->Left );

    // Print the current node.
    cout << MybtRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if ( MybtRoot->Right )
        PrintTree( MybtRoot->Right );
}

```

```

}

// Add1: Add a node to the binary tree.
//         Uses double indirection.
int Add1( BTTree **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {
        (*Root) = new BTTree;
        (*Root)->Left = 0;
        (*Root)->Right = 0;
        (*Root)->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
            return Add1( &(*Root)->Left ), szToAdd );
        else
            return Add1( &(*Root)->Right ), szToAdd );
    }
}

// Add2: Add a node to the binary tree.
//         Uses reference to pointer
int Add2( BTTree*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTTree;
        Root->Left = 0;
        Root->Right = 0;
        Root->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( Root->szText, szToAdd ) > 0 )
            return Add2( Root->Left, szToAdd );
        else
            return Add2( Root->Right, szToAdd );
    }
}

```

Output

Usage: references_to_pointers.exe [1 | 2]

where:

- 1 uses double indirection
- 2 uses a reference to a pointer.

Input is from stdin. Use ^Z to terminate input.

참조

포인터(C++)

아티클 • 2024. 11. 21.

포인터는 개체의 메모리 주소를 저장하는 변수입니다. 포인터는 세 가지 주요 용도로 C 및 C++에서 광범위하게 사용됩니다.

- 힙에 새 개체를 할당하려면
- 함수를 다른 함수에 전달하려면
- 배열 또는 기타 데이터 구조의 요소를 반복합니다.

C 스타일 프로그래밍에서는 이러한 모든 시나리오에 원시 포인터가 사용됩니다. 그러나 원시 포인터는 많은 심각한 프로그래밍 오류의 원인입니다. 따라서 중요한 성능 이점을 제공하고 개체 삭제를 담당하는 소유 포인터인 포인터에 대한 모호성이 없는 경우를 제외하고는 사용을 강력하게 권장하지 않습니다. 최신 C++는 개체 할당을 위한 스마트 포인터, 데이터 구조를 트래버스하기 위한 반복기 및 함수 전달을 위한 람다식을 제공합니다. 원시 포인터 대신 이러한 언어 및 라이브러리 기능을 사용하면 프로그램을 더 안전하고 디버그하기 쉽고 쉽게 이해하고 유지 관리할 수 있습니다. 자세한 내용은 스마트 포인터, 반복기 및 람다식을 참조하세요.

이 섹션의 내용

- 원시 포인터
- Const 및 volatile 포인터
- new 및 delete 연산자
- 스마트 포인터
- 방법: unique_ptr 인스턴스 만들기 및 사용
- 방법: shared_ptr 인스턴스 만들기 및 사용
- 방법: weak_ptr 인스턴스 만들기 및 사용
- 방법: CComPtr 및 CComQIPtr 인스턴스 만들기 및 사용

참고 항목

[반복기](#)

[람다식](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

원시 포인터 (C++)

아티클 • 2024. 09. 26.

포인터는 변수의 형식입니다. 개체의 주소를 메모리에 저장하고 해당 개체에 액세스하는 데 사용됩니다. 원시 포인터는 [스마트 포인터](#)와 같은 캡슐화 개체에 의해 수명이 제어되지 않는 포인터입니다. 원시 포인터는 다른 비 포인터 변수의 주소를 할당하거나 `nullptr`의 값을 할당할 수 있습니다. 값이 할당되지 않은 포인터에는 임의 데이터가 포함됩니다.

포인터를 역참조하여 포인터가 가리키는 개체의 값을 검색할 수도 있습니다. [멤버 액세스 연산자](#)는 개체의 멤버에 대한 액세스를 제공합니다.

C++

```
int* p = nullptr; // declare pointer and initialize it
                  // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

포인터는 형식화된 개체 또는 `void`을 가리킬 수 있습니다. 프로그램이 메모리의 [힙](#)에 개체를 할당하면 포인터 형태로 해당 개체의 주소를 받습니다. 이러한 포인터를 [소유 포인터](#)라고 합니다. 소유 포인터(또는 해당 복사본)는 더 이상 필요하지 않은 경우 힙 할당 개체를 명시적으로 해제하는 데 사용해야 합니다. 메모리를 해제하지 못하면 [메모리 누수](#)가 발생하며 컴퓨터의 다른 프로그램에서 해당 메모리 위치를 사용할 수 없게 됩니다. `new`을 사용해서 할당된 메모리는 `delete`을 사용(또는 `delete[]`)을 사용하여 해제해야 합니다. 자세한 내용은 [new 및 delete 연산자](#)를 참조하세요.

C++

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

포인터(`const`로 선언되지 않은 경우)는 메모리의 다른 위치를 가리키도록 증가하거나 감소할 수 있습니다. 이 연산을 [포인터 산술 연산](#)이라고 합니다. 배열 또는 다른 데이터 구조의 요소를 반복하기 위해 C 스타일 프로그래밍에 사용됩니다. `const` 포인터는 다른 메모리 위치를 가리키도록 만들 수 없으며, 그런 의미에서 [참조](#)와 유사합니다. 자세한 내용은 [const 및 volatile 포인터](#)를 참조하세요.

C++

```
// declare a C-style string. Compiler adds terminating '\0'.
const char* str = "Hello world";
```

```
const int c = 1;
const int* pconst = &c; // declare a non-const pointer to const int
const int c2 = 2;
pconst = &c2; // OK pconst itself isn't const
const int* const pconst2 = &c;
// pconst2 = &c2; // Error! pconst2 is const.
```

64비트 운영 체제에서 포인터의 크기는 64비트입니다. 시스템의 포인터 크기는 주소 지정 가능한 메모리의 양을 결정합니다. 포인터의 모든 복사본은 동일한 메모리 위치를 가리킵니다. 참조와 함께 포인터는 C++에서 광범위하게 사용되어 함수에 더 큰 개체를 전달합니다. 전체 개체를 복사하는 것보다 개체의 주소를 복사하는 것이 더 효율적인 경우가 많습니다. 함수를 정의할 때는 함수가 개체를 수정하지 않는 한 포인터 매개 변수를 `const`로 지정합니다. 일반적으로 `const` 참조는 개체 값이 `nullptr`이 가능하지 않으면 함수에 개체를 전달하는 기본 방법입니다.

[함수에 대한 포인터를](#) 사용하면 함수를 다른 함수에 전달할 수 있습니다. C 스타일 프로그래밍의 "콜백"에 사용됩니다. 최신 C++는 이 용도로 [람다 식](#)을 사용합니다.

초기화 및 멤버 액세스

다음 예제에서는 원시 포인터를 선언, 초기화 및 사용하는 방법을 보여줍니다. 명시적으로 `delete`을 해야 하는 힙에 할당된 개체를 가리키도록 `new`을 사용하여 초기화됩니다. 이 예제에서는 원시 포인터와 관련된 몇 가지 위험도 보여 줍니다. (이 예제는 최신 C++가 아닌 C 스타일 프로그래밍입니다.)

C++

```
#include <iostream>
#include <string>

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

// Accepts a MyClass pointer
void func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}
```

```

// Accepts a MyClass object
void func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}

int main()
{
    // Use the * operator to declare a pointer type
    // Use new to allocate and initialize memory
    MyClass* pmc = new MyClass{ 108, "Nick" };

    // Prints the memory address. Usually not what you want.
    std::cout << pmc << std::endl;

    // Copy the pointed-to object by dereferencing the pointer
    // to access the contents of the memory location.
    // mc is a separate object, allocated here on the stack
    MyClass mc = *pmc;

    // Declare a pointer that points to mc using the addressof operator
    MyClass* pcopy = &mc;

    // Use the -> operator to access the object's public members
    pmc->print(); // "Nick, 108"

    // Copy the pointer. Now pmc and pmc2 point to same object!
    MyClass* pmc2 = pmc;

    // Use copied pointer to modify the original object
    pmc2->name = "Erika";
    pmc->print(); // "Erika, 108"
    pmc2->print(); // "Erika, 108"

    // Pass the pointer to a function.
    func_A(pmc);
    pmc->print(); // "Erika, 3"
    pmc2->print(); // "Erika, 3"

    // Dereference the pointer and pass a copy
    // of the pointed-to object to a function
    func_B(*pmc);
    pmc->print(); // "Erika, 3" (original not modified by function)

    delete(pmc); // don't forget to give memory back to operating system!
    // delete(pmc2); //crash! memory location was already deleted
}

```

포인터 산술 및 배열

포인터와 배열은 밀접하게 관련되어 있습니다. 배열이 함수에 값으로 전달되면 첫 번째 요소에 대한 포인터로 전달됩니다. 다음 예제에서는 포인터 및 배열의 다음과 같은 중요한 속성을 보여 줍니다.

- `sizeof` 연산자는 배열의 총 크기(바이트)를 반환합니다.
- 요소 수를 확인하려면 총 바이트를 한 요소의 크기로 나눕니다.
- 배열이 함수에 전달되면 포인터 형식으로 감소됩니다.
- `sizeof` 연산자가 포인터에 적용되면 포인터 크기(예: x86의 경우 4바이트 또는 x64의 8바이트)를 반환합니다.

C++

```
#include <iostream>

void func(int arr[], int length)
{
    // returns pointer size. not useful here.
    size_t test = sizeof(arr);

    for(int i = 0; i < length; ++i)
    {
        std::cout << arr[i] << " ";
    }
}

int main()
{
    int i[5]{ 1,2,3,4,5 };
    // sizeof(i) = total bytes
    int j = sizeof(i) / sizeof(i[0]);
    func(i,j);
}
```

비 `const` 포인터에서 특정 산술 연산을 사용하여 다른 메모리 위치를 가리킬 수 있습니다. 포인터는 `++`, `+=`, `-=` 및 `--` 연산자를 사용하여 증가 및 감소됩니다. 이 기술은 배열에서 사용할 수 있으며 형식화되지 않은 데이터의 버퍼에서 특히 유용합니다. `void*`는 `char`(1 바이트)의 크기로 증가합니다. 형식화된 포인터는 포인터가 가리키는 형식의 크기로 증가합니다.

다음 예제에서는 포인터 산술 연산을 사용하여 Windows의 비트맵에서 개별 픽셀에 액세스하는 방법을 보여 줍니다. `new` 및 `delete`의 사용 및 역참조 연산자를 확인합니다.

C++

```

#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{
    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    constexpr int bufferSize = 30000;
    unsigned char* buffer = new unsigned char[bufferSize];

    BITMAPFILEHEADER bf;
    bf.bfType = 0x4D42;
    bf.bfSize = header.biSize + 14 + bufferSize;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

    // Create a gray square with a 2-pixel wide outline.
    unsigned char* begin = &buffer[0];
    unsigned char* end = &buffer[0] + bufferSize;
    unsigned char* p = begin;
    constexpr int pixelWidth = 3;
    constexpr int borderWidth = 2;

    while (p < end)
    {
        // Is top or bottom edge?
        if ((p < begin + header.biWidth * pixelWidth * borderWidth)
            || (p > end - header.biWidth * pixelWidth * borderWidth))
            // Is left or right edge?
            || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth *
pixelWidth)
            || (p - begin) % (header.biWidth * pixelWidth) >
((header.biWidth - borderWidth) * pixelWidth))
        {
            *p = 0x0; // Black
        }
        else
        {
            *p = 0xC3; // Gray
        }
        p++; // Increment one byte sizeof(unsigned char).
    }

    ofstream wf(R"(box.bmp)", ios::out | ios::binary);

```

```

wf.write(reinterpret_cast<char*>(&bf), sizeof(bf));
wf.write(reinterpret_cast<char*>(&header), sizeof(header));
wf.write(reinterpret_cast<char*>(begin), bufferSize);

delete[] buffer; // Return memory to the OS.
wf.close();
}

```

void* 포인터

단순히 원시 메모리 위치를 가리키는 `void`에 대한 포인터입니다. C++ 코드와 C 함수 간에 전달하는 경우와 같이 `void*` 포인터를 사용해야 하는 경우도 있습니다.

형식화된 포인터가 `void` 포인터로 캐스팅되면 메모리 위치의 내용이 변경되지 않습니다. 그러나 충분 또는 감소 작업을 수행할 수 없도록 형식 정보가 손실됩니다. 예를 들어 `MyClass*`에서 `void*`로, 그리고 다시 `MyClass*`로 메모리 위치를 캐스팅할 수 있습니다. 이러한 작업은 본질적으로 오류가 발생하기 쉬며 오류를 방지하기 위해 세심한 주의가 필요합니다. 최신 C++는 거의 모든 상황에서 `void` 포인터 사용을 권장하지 않습니다.

C++

```

//func.c
void func(void* data, int length)
{
    char* c = (char*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    void func(void* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

```

```

int main()
{
    MyClass* mc = new MyClass{10, "Marian"};
    void* p = static_cast<void*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"
    delete(mc);

    // use operator new to allocate untyped memory block
    void* pvoid = operator new(1000);
    char* pchar = static_cast<char*>(pvoid);
    for(char* c = pchar; c < pchar + 1000; ++c)
    {
        *c = 0x00;
    }
    func(pvoid, 1000);
    char ch = static_cast<char*>(pvoid)[0];
    std::cout << ch << std::endl; // 'A'
    operator delete(pvoid);
}

```

함수에 대한 포인터

C 스타일 프로그래밍에서 함수 포인터는 주로 함수를 다른 함수에 전달하는 데 사용됩니다. 이 기술을 사용하면 호출자가 함수를 수정하지 않고 함수의 동작을 사용자 지정할 수 있습니다. 최신 C++에서 [람다 식](#)은 더 큰 형식 안전성 및 기타 장점으로 동일한 기능을 제공합니다.

함수 포인터 선언은 뾰족한 함수에 있어야 하는 서명을 지정합니다.

C++

```

// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
void (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);

```

다음 예제에서는 `std::string`를 수락하고 `std::string`를 반환하는 모든 함수를 매개 변수로 사용하는 함수 `combine`를 보여줍니다. `combine`로 전달되는 함수에 따라 문자열 앞에 추가하거나 추가합니다.

C++

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s)
{
    return base.append(" ").append(s);
}

string prepend(string s)
{
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a))
{
    return (*g)(s);
}

int main()
{
    cout << combine("from MSVC", append) << "\n";
    cout << combine("Good morning and", prepend) << "\n";
}
```

참고 항목

[스마트 포인터간접 참조 연산자: *](#)

[연산자 주소: &](#)

[C++로 시작하기](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

const 및 volatile 포인터

아티클 • 2024. 07. 08.

`const` 및 `volatile` 키워드는 포인터 처리 방법을 변경합니다. `const` 키워드는 포인터가 초기화 후 수정될 수 없도록 지정하여 이후 포인터가 수정되지 않도록 보호합니다.

`volatile` 키워드는 뒤에 오는 이름과 관련된 값이 사용자 애플리케이션의 작업 이외의 작업으로 수정될 수 있도록 지정합니다. 따라서, `volatile` 키워드는 인터럽트 서비스 루틴과의 통신에 사용되는 다중 처리나 전역 데이터 영역에서 접근할 수 있는 공유 메모리에서 개체를 선언하는 데 유용합니다.

이름을 `volatile`로 선언한 경우, 컴파일러가 프로그램에 액세스할 때마다 메모리에서 값을 다시 로딩하기 때문에 개체의 최적화 횟수가 상당히 줄어들게 됩니다. 이 키워드는 개체 상태가 예기치 않게 변경되는 경우 예상 가능한 프로그램 성능을 보장하는 유일한 방법이기도 합니다.

포인터가 가리키는 개체를 `const` 또는 `volatile`로 선언하려면 다음 형식의 선언을 사용합니다.

C++

```
const char *cpch;
volatile char *vpch;
```

포인터의 값, 즉 포인터에 저장된 실제 주소를 `const` 또는 `volatile`로 선언하려면, 다음과 같은 선언 형식을 사용합니다.

C++

```
char * const pchc;
char * volatile pchv;
```

C++ 언어에서는 `const`로 선언된 개체 또는 포인터의 수정을 허용하는 할당을 금지합니다. 그와 같은 할당은 개체 또는 포인터가 선언된 정보를 제거하기 때문에 원래의 선언 의도에 위배됩니다. 다음의 선언을 살펴보세요.

C++

```
const char cch = 'A';
char ch = 'B';
```

이전 두 개체의 선언(`const char` 형식의 `cch` 및 `char` 형식의 `ch`)이 주어진 경우, 다음의 선언/초기화가 유효합니다.

C++

```
const char *pch1 = &cch;
const char *const pch4 = &cch;
const char *pch5 = &ch;
char *pch6 = &ch;
char *const pch7 = &ch;
const char *const pch8 = &ch;
```

다음의 선언/초기화는 잘못되었습니다.

C++

```
char *pch2 = &cch;    // Error
char *const pch3 = &cch;    // Error
```

`pch2` 선언은 상수 개체가 수정될 수 있는 포인터를 선언하기 때문에 허용되지 않습니다.

`pch3` 선언은 포인터가 개체가 아닌 상수임을 지정하기 때문에 `pch2` 선언과 마찬가지 이유로 허용되지 않습니다.

다음 8개의 할당에서는 포인터를 통한 할당과 이전 선언에 대한 포인터 값의 변경을 표시합니다. 이제 `pch1`을 통해 `pch8`에 대한 초기화가 올바르다고 가정합니다.

C++

```
*pch1 = 'A';    // Error: object declared const
pch1 = &ch;    // OK: pointer not declared const
*pch2 = 'A';    // OK: normal pointer
pch2 = &ch;    // OK: normal pointer
*pch3 = 'A';    // OK: object not declared const
pch3 = &ch;    // Error: pointer declared const
*pch4 = 'A';    // Error: object declared const
pch4 = &ch;    // Error: pointer declared const
```

`volatile` 또는 `const`와 `volatile`의 혼합으로 선언된 포인터는 동일한 규칙을 따릅니다.

`const` 개체에 대한 포인터는 다음과 같이 함수 선언에 자주 사용됩니다.

C++

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char
*strSource );
```

앞의 문에서 `strcpy_s` 함수를 선언하면, 이 함수의 세 개 인수 중 두 개는 `char` 형식의 포인터가 됩니다. 인수는 값이 아닌 참조로 전달되므로 `strSource` 가 `const`로 선언되지 않은 경우, 함수는 `strDestination` 및 `strSource`를 모두 자유롭게 수정할 수 있습니다. `strSource` 를 `const`로 선언하면 호출된 함수가 `strSource` 를 변경할 수 없음을 호출자에게 보장합니다.

① 참고

`typename *`에서 `const typename *`으로의 표준 변환이 있으므로 `char *` 형식의 인수를 `strcpy_s`로 전달할 수 있습니다. 그러나 반대의 경우는 성립되지 않습니다. 개체 또는 포인터에서 `const` 특성을 제거하는 암시적 변환이 존재하지 않기 때문입니다.

지정된 형식의 `const` 포인터는 동일한 형식의 포인터에 할당될 수 있습니다. 그러나 `const`가 아닌 포인터는 `const` 포인터에 할당될 수 없습니다. 다음 코드는 올바른 할당과 잘못된 할당을 보여 줍니다.

C++

```
// const_pointer.cpp
int *const cp0bject = 0;
int *p0bject;

int main() {
    p0bject = cp0bject;
    cp0bject = p0bject;    // C3892
}
```

다음 샘플에서는 개체에 포인터에 대한 포인터가 있는 경우 개체를 `const`로 선언하는 방법을 보여 줍니다.

C++

```
// const_pointer2.cpp
struct X {
    X(int i) : m_i(i) { }
    int m_i;
};

int main() {
    // correct
    const X cx(10);
    const X * pcx = &cx;
    const X ** ppcx = &pcx;

    // also correct
    X const cx2(20);
```

```
X const * pcx2 = &cx2;
X const ** ppcx2 = &pcx2;
}
```

참고 항목

[포인터원시 포인터](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

new 및 delete 연산자

아티클 • 2024. 07. 15.

C++는 `new` 및 `delete` 연산자를 사용하는 개체의 동적 할당 및 할당 해제를 지원합니다. 이러한 연산자는 사용 가능한 저장소(힙이라고도 함)라고 하는 풀에서 개체에 대한 메모리를 할당합니다. `new` 연산자는 `operator new` 특수 함수를 호출하고, `delete` 연산자는 `operator delete` 특수 함수를 호출합니다.

C 런타임 라이브러리와 C++ 표준 라이브러리에 있는 라이브러리 파일의 목록은 [CRT 라이브러리 기능](#)을 참조하십시오.

new 연산자

컴파일러는 다음과 같은 문을 `operator new` 함수에 대한 호출로 변환합니다.

C++

```
char *pch = new char[BUFFER_SIZE];
```

요청이 0바이트의 스토리지를 위한 것이면 `operator new`는 고유한 개체에 대한 포인터를 반환합니다. 즉, `operator new`에 대한 반복적인 호출은 서로 다른 포인터를 반환합니다.

할당 요청에 대한 메모리가 불충분한 경우 `operator new`는 `std::bad_alloc` 예외를 throw 합니다. 또는 배치 형식인 `new(std::nothrow)`를 사용했거나 throw되지 않는 `operator new` 지원에서 연결한 경우 `nullptr`을 반환합니다. 자세한 내용은 [할당 오류 동작](#) 참조하세요.

다음 표에는 `operator new` 함수의 두 범위가 설명되어 있습니다.

operator new 함수의 범위

[+] 테이블 확장

연산자	범위
<code>::operator new</code>	전역
<code>class-name ::operator new</code>	클래스

`operator new`의 첫 번째 인수는 `size_t` 형식이어야 하며 반환 형식은 항상 `void*`입니다.

기본 제공 형식의 개체, 사용자 정의된 `operator new` 함수를 포함하지 않는 클래스 형식의 개체, 및 모든 형식의 배열을 할당하는 데 `new` 연산자를 사용하는 경우, 전역 `operator new` 함수가 호출됩니다. `operator new`가 정의되는 클래스 형식의 개체를 할당하는 데 `new` 연산자를 사용하는 경우, 해당 클래스의 `operator new`가 호출됩니다.

클래스에 대해 정의된 `operator new` 함수는 해당 클래스 형식의 개체에 대한 전역 `operator new` 함수를 숨기는 정적 멤버 함수입니다(가상은 될 수 없음). 메모리를 할당하고 주어진 값으로 설정하는 데 `new`를 사용하는 다음 사례를 생각해 보세요.

C++

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};

void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}

// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
    Blanks *a5 = new(0xa5) Blanks;
    return a5 != 0;
}
```

`new`의 괄호 안에 있는 인수는 `chInit` 인수로서 `Blanks::operator new`에 전달됩니다. 그러나 전역 `operator new` 함수가 숨겨져 있기 때문에 다음과 같은 오류를 생성하는 코드가 발생할 수 있습니다.

C++

```
Blanks *SomeBlanks = new Blanks;
```

컴파일러는 클래스 선언에서 멤버 배열 `new` 및 `delete` 연산자를 지원합니다. 예시:

C++

```

class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}

```

할당 오류 동작

C++ 표준 라이브러리의 `new` 함수는 C++98 이후로 C++ 표준에서 지정된 동작을 지원합니다. 할당 요청에 대한 메모리가 불충분한 경우 `operator new`는 `std::bad_alloc` 예외를 `throw`합니다.

이전 C++ 코드는 실패한 할당에 대해 `null` 포인터를 반환했습니다. `Throw`되지 않는 버전의 `new`를 예상하는 코드가 있는 경우, 프로그램을 `nothrownew.obj`와 연결합니다. 할당이 실패할 경우, `nothrownew.obj` 파일은 `nullptr`를 반환하는 버전으로 전역 `operator new`를 바꿉니다. `operator new`는 더 이상 `std::bad_alloc`을 `throw`하지 않습니다.

`nothrownew.obj` 및 기타 링커 옵션 파일에 대한 자세한 내용은 [링크 옵션](#)을 참조하세요.

전역 `operator new`의 예외를 검사하는 코드와 `null` 포인터를 검사하는 코드를 동일한 애플리케이션에서 혼합할 수 없습니다. 그러나 다르게 동작하는 클래스-로컬 `operator new`를 만들 수 있습니다. 이 가능성은 컴파일러가 기본적으로 방어적으로 작동하며 `new` 호출에서 `null` 포인터 반환에 대한 검사를 포함해야 한다는 것을 의미합니다. 이러한 컴파일러 검사를 최적화하는 방법에 대한 자세한 내용은 [/Zc:throwingnew](#)를 참조하세요.

메모리 부족 처리

`new` 식에서 실패한 할당을 테스트하는 방법은 표준 예외 메커니즘을 사용하는지 또는 `nullptr` 반환을 사용하는지에 따라 달라집니다. 표준 C++는 할당자가 `std::bad_alloc` 또는 `std::bad_alloc`에서 파생된 클래스를 `throw`할 것으로 예상합니다. 아래 샘플에 표시된 것처럼 이러한 예외를 처리할 수 있습니다.

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
    try {
        int *pI = new int[BIG_NUMBER];
    }
    catch (bad_alloc& ex) {
        cout << "Caught bad_alloc: " << ex.what() << endl;
        return -1;
    }
}
```

`nothrow` 양식의 `new`를 사용하는 경우 다음 샘플과 같이 할당 오류를 테스트할 수 있습니다.

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
    int *pI = new(nothrow) int[BIG_NUMBER];
    if ( pI == nullptr ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

다음과 같이 `nothrownew.obj` 파일을 사용하여 전역 `operator new`를 대체한 경우, 실패한 메모리 할당을 테스트할 수 있습니다.

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
    int *pI = new int[BIG_NUMBER];
    if ( !pI ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

실패한 메모리 할당 요청에 대한 처리기를 제공할 수 있습니다. 이러한 실패를 처리하는 사용자 지정 복구 루틴을 작성할 수 있습니다. 예를 들어, 일부 예약된 메모리를 해제한 다음, 할당이 다시 실행되도록 허용할 수 있습니다. 자세한 내용은 [_set_new_handler](#)를 참조하세요.

delete 연산자

new 연산자를 사용하여 동적으로 할당되는 메모리는 **delete** 연산자를 사용하여 비울 수 있습니다. **delete** 연산자는 **operator delete** 함수를 호출하며, 이 함수는 메모리를 사용 가능한 풀로 다시 비웁니다. **delete** 연산자를 사용하면 클래스 소멸자(있을 경우)도 호출됩니다.

전역 및 클래스 범위의 **operator delete** 함수가 있습니다. 하나의 **operator delete** 함수만 특정 클래스에 대해 정의할 수 있으며, 정의하는 경우 전역 **operator delete** 함수가 숨겨집니다. 전역 **operator delete** 함수는 항상 모든 형식의 배열에 대해 호출됩니다.

전역 **operator delete** 함수입니다. 전역 **operator delete** 및 클래스 멤버 **operator delete** 함수에 대해 다음의 두 가지 양식이 있습니다.

C++

```
void operator delete( void * );
void operator delete( void *, size_t );
```

앞의 두 가지 양식 중 하나만 특정 클래스에 있을 수 있습니다. 첫 번째 양식은 할당을 취소할 개체에 대한 포인터를 포함하는 **void *** 형식의 단일 인수를 사용합니다. 두 번째 양식(크기가 지정된 할당 해제)에는 두 개의 인수가 사용됩니다. 첫 번째 인수는 할당을 해제할 메모리 블록에 대한 포인터이며, 두 번째 인수는 할당을 해제할 바이트 수입니다. 이 두 가지 양식의 반환 형식은 **void**입니다(**operator delete**는 값을 반환할 수 없음).

두 번째 양식의 의도는 삭제할 개체의 올바른 크기 범주를 빠르게 검색하는 것입니다. 이러한 정보는 할당 자체 근처에 저장되지 않는 경우가 많으며, 캐시되지 않을 가능성이 큽니다. 두 번째 양식은 기본 클래스의 **operator delete** 함수를 사용하여 파생 클래스의 개체를 삭제할 때 유용합니다.

operator delete 함수는 정적이므로 가상일 수 없습니다. **operator delete** 함수는 [멤버 액세스 제어](#)에 설명된 대로 액세스 제어를 따릅니다.

다음 예제에서는 메모리의 할당 및 할당 해제를 기록하도록 설계된 사용자 정의된 **operator new** 및 **operator delete** 함수를 보여 줍니다.

C++

```

#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0; // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
    static int fInOpNew = 0; // Guard flag.

    if ( fLogMemory && !fInOpNew ) {
        fInOpNew = 1;
        clog << "Memory block " << ++cBlocksAllocated
            << " allocated for " << stAllocateBlock
            << " bytes\n";
        fInOpNew = 0;
    }
    return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
    static int fInOpDelete = 0; // Guard flag.
    if ( fLogMemory && !fInOpDelete ) {
        fInOpDelete = 1;
        clog << "Memory block " << cBlocksAllocated--
            << " deallocated\n";
        fInOpDelete = 0;
    }

    free( pvMem );
}

int main( int argc, char *argv[] ) {
    fLogMemory = 1; // Turn logging on
    if( argc > 1 )
        for( int i = 0; i < atoi( argv[1] ); ++i ) {
            char *pMem = new char[10];
            delete[] pMem;
        }
    fLogMemory = 0; // Turn logging off.
    return cBlocksAllocated;
}

```

앞의 코드를 사용하여 "메모리 누수"를 감지할 수 있습니다. 메모리 누수는 사용 가능한 저장소에 할당되었지만 비워지지 않은 메모리를 의미합니다. 누수를 감지하기 위해 전역 `new` 및 `delete` 연산자를 다시 정의해서 메모리의 할당 및 할당 해제의 횟수를 셹니다.

컴파일러는 클래스 선언에서 멤버 배열 `new` 및 `delete` 연산자를 지원합니다. 예시:

```
// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X {
public:
    void * operator new[] (size_t) {
        return 0;
    }
    void operator delete[] (void*) {}
};

void f() {
    X *pX = new X[5];
    delete [] pX;
}
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

스마트 포인터(최신 C++)

아티클 • 2023. 06. 16.

최신 C++ 프로그래밍에서 표준 라이브러리에는 프로그램에 메모리 및 리소스 누수가 없고 예외로부터 안전한지 확인하는 데 사용되는 스마트 포인터가 포함되어 있습니다.

스마트 포인터 용도

스마트 포인터는 [메모리> 헤더 파일의 네임스페이스에 정의됩니다.](#) `std::RAII` 또는 리소스 획득 초기화 프로그래밍 관용구에 매우 중요합니다. 이 관용구의 주요 목표는 개체의 모든 자원 생성이 한 줄의 코드에서 만들어지고 준비되어 그 개체가 초기화되는 동시에 자원 수집이 발생하는 것을 확인하는 것입니다. 실제로 RAII의 기본 원칙은 힙 할당 리소스(예: 동적 할당 메모리 또는 시스템 개체 핸들)의 소유권을 해당 소멸자가 리소스를 삭제하거나 비우는 코드 및 모든 관련 정리 코드가 포함된 스택 할당 개체에 제공하는 것입니다.

대부분의 경우 실제 리소스를 가리키도록 기본 포인터 또는 리소스 핸들을 초기화할 때는 포인터를 스마트 포인터로 즉시 전달합니다. 현대적인 C++에서 기본 포인터는 성능이 중요하며, 소유권 관련 혼동 여지가 없는 제한된 범위, 루프 또는 지원 함수의 작은 코드 블록에서만 사용됩니다.

다음 예제에서는 원시 포인터 선언을 스마트 포인터 선언과 비교합니다.

```
C++  
  
void UseRawPointer()  
{  
    // Using a raw pointer -- not recommended.  
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");  
  
    // Use pSong...  
  
    // Don't forget to delete!  
    delete pSong;  
}  
  
void UseSmartPointer()  
{  
    // Declare a smart pointer on stack and pass it the raw pointer.  
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));  
  
    // Use song2...  
    wstring s = song2->duration_;  
    //...
```

```
 } // song2 is deleted automatically here.
```

예제에서와 같이 스마트 포인터는 스택에서 선언되고 힙 할당 개체를 가리키는 원시 포인터를 사용하여 초기화하는 스마트 포인터입니다. 스마트 포인터가 초기화되면 원시 포인터를 소유합니다. 원시 포인터가 지정한 메모리를 스마트 포인터가 삭제해야 함을 의미합니다. 스마트 포인터 소멸자에는 삭제할 호출이 포함되며, 스마트 포인터는 스택에 선언되기 때문에 스마트 포인터가 범위를 벗어나면 스택의 다른 위치에서 예외가 throw되더라도 해당 소멸자가 호출됩니다.

스마트 포인터 클래스 오버로드가 캡슐화된 원시 포인터를 반환하는 익숙한 포인터 연산자인 `->` 및 `*`를 사용하여 캡슐화된 포인터에 액세스합니다.

C++ 스마트 포인터 관용구는 C# 같은 언어에서의 개체 생성과 유사합니다. 즉 개체를 생성한 후 시스템이 정확한 시간에 그것을 지울 수 있도록 합니다. 백그라운드에서 실행된 별도 가비지 컬렉션기는 런타임 환경을 빠르고 효율적이게 만드는 표준 C++ 범위 지정 규칙을 통해 관리되는 메모리라는 것이 차이점입니다.

① 중요

매개 변수 목록이 아닌 별도 코드 줄에서 스마트 포인터를 만들어야 특정 매개 변수 목록 할당 규칙으로 인해 아주 작은 리소스 누수도 발생하지 않습니다.

다음 예제에서는 C++ 표준 라이브러리의 스마트 포인터 형식을 사용하여 큰 개체에 대한 포인터를 캡슐화하는 방법을 `unique_ptr` 보여 줍니다.

C++

```
class LargeObject
{
public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}
void SmartPointerDemo()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);
}
```

```
} //pLarge is deleted automatically when function block goes out of scope.
```

이 예제는 다음과 같이 스마트 포인터를 사용하는 필수 단계를 보여 줍니다.

1. 스마트 포인터를 자동(지역) 변수로 선언합니다. (스마트 포인터 자체에서 `new` 또는 `malloc` 식을 사용하지 마세요.)
2. 형식 매개 변수에서 캡슐화된 포인터가 가리키는 대상의 형식을 지정합니다.
3. 스마트 포인터 `new` 생성자의 -ed 개체에 원시 포인터를 전달합니다. (일부 유ти리티 기능 또는 스마트 포인터 생성자로 이 작업을 자동으로 수행할 수 있습니다.)
4. 오버로드된 `->` 및 `*` 연산자를 사용하여 개체에 액세스합니다.
5. 스마트 포인터로 개체를 삭제할 수 있습니다.

스마트 포인터는 메모리 및 성능 관점에서 최대한 효율적으로 사용할 수 있도록 설계되었습니다. 예를 들어, `unique_ptr`의 유일한 데이터 멤버는 캡슐화된 포인터입니다. 즉, `unique_ptr`은 해당 포인터와 정확히 동일한 크기(4바이트 또는 8바이트)입니다. 오버로드된 스마트 포인터 `*` 및 `->` 연산자를 사용하여 캡슐화된 포인터에 액세스하는 것은 원시 포인터에 직접 액세스하는 것보다 훨씬 느리지 않습니다.

스마트 포인터에는 "점" 표기법을 사용하여 액세스하는 고유한 멤버 함수가 있습니다. 예를 들어 일부 C++ 표준 라이브러리 스마트 포인터에는 포인터의 소유권을 해제하는 재설정 멤버 함수가 있습니다. 다음 예제 표시된 것처럼 스마트 포인터가 범위를 벗어나기 전에 스마트 포인터에서 소유하는 메모리를 비우려는 경우에 유용합니다.

C++

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...
}
```

스마트 포인터는 일반적으로 원시 포인터에 직접 액세스하는 방법을 제공합니다. C++ 표준 라이브러리 스마트 포인터에는 `get` 이 목적을 위한 멤버 함수가 있고 `CCComPtr` 공용 `p` 클래스 멤버가 있습니다. 기본 포인터에 대한 직접 액세스를 제공하면 스마트 포인터를 사용하여 자체 코드에서 메모리를 관리하고 스마트 포인터를 지원하지 않는 코드에 원시 포인터를 전달할 수 있습니다.

```
C++  
  
void SmartPointerDemo4()  
{  
    // Create the object and pass it to a smart pointer  
    std::unique_ptr<LargeObject> pLarge(new LargeObject());  
  
    // Call a method on the object  
    pLarge->DoSomething();  
  
    // Pass raw pointer to a legacy API  
    LegacyLargeObjectFunction(pLarge.get());  
}
```

스마트 포인터 종류

다음 섹션에서는 다양한 종류의 Windows 프로그래밍 환경에서 사용할 수 있는 스마트 포인터에 대한 정보를 요약하고 사용하는 경우에 대해 설명합니다.

C++ 표준 라이브러리 스마트 포인터

POCO(Plain Old C++ 개체)에 대한 포인터를 캡슐화하는 데 가장 먼저 스마트 포인터를 사용합니다.

- `unique_ptr`

기본 포인터로 한 명의 소유자만 허용합니다. `shared_ptr`이 필요하다는 점을 확실히 알 경우 POCO의 기본 선택으로 사용합니다. 새 소유자로 이동할 수 있지만 복사하거나 공유할 수 없습니다. 사용하지 않는 `auto_ptr`를 대체합니다.

`boost::scoped_ptr`과 비교합니다. `unique_ptr` 작고 효율적입니다. 크기는 하나의 포인터이며 C++ 표준 라이브러리 컬렉션에서 빠른 삽입 및 검색을 위한 rvalue 참조를 지원합니다. 헤더 파일: `<memory>`. 자세한 내용은 방법: [unique_ptr 인스턴스 만들기 및 사용](#) 및 [unique_ptr 클래스](#)를 참조하세요.

- `shared_ptr`

참조 횟수가 계산되는 스마트 포인터입니다. 원시 포인터 하나를 여러 소유자에게 할당하려고 할 경우 사용합니다(예: 컨테이너에서 포인터 복사본을 반환할 때 원본을 유지하고 싶을 경우). 원시 포인터는 모든 `shared_ptr` 소유자가 범위를 벗어나거나

나 소유권을 포기할 때까지 삭제되지 않습니다. 크기는 2개의 포인터입니다. 하나는 개체용이고, 다른 하나는 참조 횟수가 포함된 공유 제어 블록용입니다. 헤더 파일: `<memory>`. 자세한 내용은 [방법: shared_ptr 인스턴스 만들기 및 사용 및 shared_ptr 클래스](#)를 참조하세요.

- `weak_ptr`

`shared_ptr`과 함께 사용할 수 있는 특별한 경우의 스마트 포인터입니다. `weak_ptr`은 하나 이상의 `shared_ptr` 인스턴스가 소유하는 개체에 대한 액세스를 제공하지만, 참조 수 계산에 참가하지 않습니다. 개체를 관찰하는 동시에 해당 개체를 활성 상태로 유지하지 않으려는 경우 사용합니다. `shared_ptr` 인스턴스 사이의 순환 참조를 차단하기 위해 필요한 경우도 있습니다. 헤더 파일: `<memory>`. 자세한 내용은 [방법: weak_ptr 인스턴스 만들기 및 사용 및 weak_ptr 클래스](#)를 참조하세요.

COM 개체에 대한 스마트 포인터(클래식 Windows 프로그래밍)

COM 개체를 사용하는 경우 인터페이스 포인터를 적절한 스마트 포인터 형식으로 래핑합니다. ATL(Active Template Library)은 다양한 목적을 위해 여러 스마트 포인터를 정의합니다. .tlb 파일에서 래퍼 클래스를 만들 때 컴파일러가 사용하는 `_com_ptr_t` 스마트 포인터 형식을 사용할 수도 있습니다. ATL 헤더 파일을 포함하지 않으려는 경우에 가장 좋습니다.

`CComPtr` 클래스

ATL을 사용할 수 없는 이상 이 형식을 사용합니다. `AddRef` 및 `Release` 메서드를 사용하여 참조 수 계산을 수행합니다. 자세한 내용은 [방법: CComPtr 및 CComQIPtr 인스턴스 만들기 및 사용을 참조하세요](#).

`CComQIPtr` 클래스

`CComPtr`과 유사하지만 COM 개체에서 `QueryInterface`를 호출하는 간단한 구문을 제공합니다. 자세한 내용은 [방법: CComPtr 및 CComQIPtr 인스턴스 만들기 및 사용을 참조하세요](#).

`CComHeapPtr` 클래스

`CoTaskMemFree`를 사용하여 메모리를 해제하는 개체에 대한 스마트 포인터

`CComGIPtr` 클래스

GIT(전역 인터페이스 테이블)에서 가져온 인터페이스에 대한 스마트 포인터입니다.

`_com_ptr_t` 클래스

기능 면에서 `CComQIPtr`과 유사하지만 ATL 헤더에 의존하지 않습니다.

POCO 개체에 대한 ATL 스마트 포인터

ATL은 COM 개체에 대한 스마트 포인터 외에도 POCO(일반 이전 C++ 개체)에 대한 스마트 포인터 및 스마트 포인터 컬렉션을 정의합니다. 클래식 Windows 프로그래밍에서 이러한 형식은 특히 코드 이식성이 필요하지 않거나 C++ 표준 라이브러리 및 ATL의 프로그래밍 모델을 혼합하지 않으려는 경우 C++ 표준 라이브러리 컬렉션에 대한 유용한 대안입니다.

[CAutoPtr 클래스](#)

복사 소유권을 전송하여 고유 소유권을 적용하는 스마트 포인터입니다. 사용되지 않는 `std::auto_ptr` 클래스에 비교할 수 있습니다.

[CHheapPtr 클래스](#)

C `malloc` 함수를 사용하여 할당된 개체에 대한 스마트 포인터입니다.

[CAutoVectorPtr 클래스](#)

배열을 위해 `new[]`를 사용하여 할당하는 스마트 포인터입니다.

[CAutoPtrArray 클래스](#)

`CAutoPtr` 요소 배열을 캡슐화하는 클래스입니다.

[CAutoPtrList 클래스](#)

`CAutoPtr` 노드 목록을 조작하기 위해 메서드를 캡슐화하는 클래스입니다.

추가 정보

[포인터](#)

[C++ 언어 참조](#)

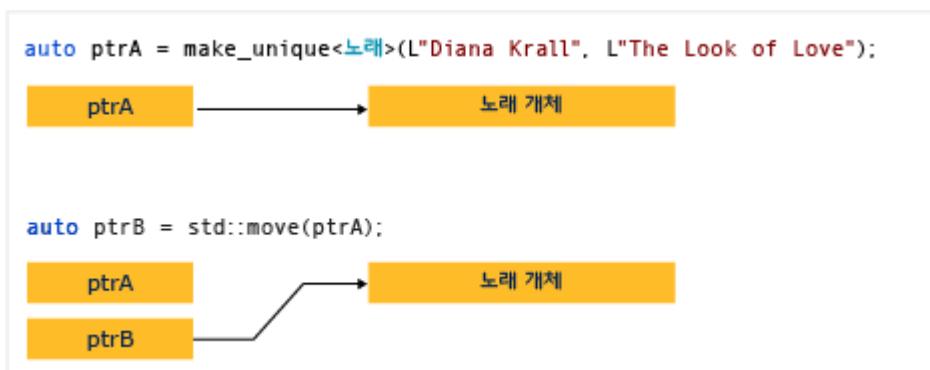
[C++ 표준 라이브러리](#)

방법: unique_ptr 인스턴스 만들기 및 사용

아티클 • 2023. 10. 12.

`unique_ptr` 포인터를 공유하지 않습니다. 다른 `unique_ptr` 함수로 복사하거나, 값으로 함수에 전달하거나, 복사본을 만들어야 하는 C++ 표준 라이브러리 알고리즘에서 사용할 수 없습니다. `unique_ptr`은 이동만 할 수 있습니다. 즉, 메모리 리소스의 소유권이 다른 `unique_ptr`로 이전되어 원래 `unique_ptr`이 더 이상 소유하지 않습니다. 소유권이 여러 개이면 프로그램 논리가 복잡해지기 때문에 개체를 하나의 소유자로 제한하는 것이 좋습니다. 따라서 일반 C++ 개체에 대한 스마트 포인터가 필요한 경우 해당 개체를 사용하고 `unique_ptr` 생성 `unique_ptr` 할 때 `make_unique` 도우미 함수를 사용합니다.

다음 다이어그램은 두 `unique_ptr` 인스턴스 사이의 소유권 이전을 보여 줍니다.



`unique_ptr`는 C++ 표준 라이브러리의 `<memory>` 헤더에 정의됩니다. 원시 포인터만큼 효율적이며 C++ 표준 라이브러리 컨테이너에서 사용할 수 있습니다. C++ 표준 라이브러리 컨테이너에 인스턴스를 추가 `unique_ptr`하면 복사 작업이 필요하지 않으므로 이동 생성자가 `unique_ptr` 효율적입니다.

예 1

다음 예제에서는 `unique_ptr` 인스턴스를 어떻게 만들고 이를 함수 사이에서 어떻게 전달하는지를 보여 줍니다.

C++

```
unique_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring& title)
{
    // Implicit move operation into the variable that stores the result.
    return make_unique<Song>(artist, title);
}
```

```

void MakeSongs()
{
    // Create a new unique_ptr with a new object.
    auto song = make_unique<Song>(L"Mr. Children", L"Namonaki Uta");

    // Use the unique_ptr.
    vector<wstring> titles = { song->title };

    // Move raw pointer from one unique_ptr to another.
    unique_ptr<Song> song2 = std::move(song);

    // Obtain unique_ptr from function that returns by value.
    auto song3 = SongFactory(L"Michael Jackson", L"Beat It");
}

```

이러한 예는 `unique_ptr`의 기본적인 특징을 보여 주며 이동은 가능하지만 복사되지 않을 수 있습니다. "이동"이 소유권을 새 `unique_ptr`로 이전하고 이전 `unique_ptr`을 다시 설정합니다.

예제 2

다음 예제에서는 `unique_ptr` 인스턴스를 만들고 이를 벡터에 사용하는 방법을 보여 줍니다.

C++

```

void SongVector()
{
    vector<unique_ptr<Song>> songs;

    // Create a few new unique_ptr<Song> instances
    // and add them to vector using implicit move semantics.
    songs.push_back(make_unique<Song>(L"B'z", L"Juice"));
    songs.push_back(make_unique<Song>(L"Namie Amuro", L"Funky Town"));
    songs.push_back(make_unique<Song>(L"Kome Kome Club", L"Kimi ga Iru Dake
de"));
    songs.push_back(make_unique<Song>(L"Ayumi Hamasaki", L"Poker Face"));

    // Pass by const reference when possible to avoid copying.
    for (const auto& song : songs)
    {
        wcout << L"Artist: " << song->artist << L"    Title: " << song->title
<< endl;
    }
}

```

루프 범위에서 `unique_ptr`이 참조로 전달됩니다. 여기에서 값으로 전달하려는 경우 `unique_ptr` 복사 생성자가 삭제되기 때문에 컴파일러는 오류를 `throw`합니다.

예 3

다음 예제에서는 클래스 멤버인 `unique_ptr`을 초기화하는 방법을 보여 줍니다.

C++

```
class MyClass
{
private:
    // MyClass owns the unique_ptr.
    unique_ptr<ClassFactory> factory;
public:

    // Initialize by using make_unique with ClassFactory default
    // constructor.
    MyClass() : factory (make_unique<ClassFactory>())
    {
    }

    void MakeClass()
    {
        factory->DoSomething();
    }
};
```

예제 4

`make_unique` 사용하여 배열을 만들 `unique_ptr` 수 있지만 배열 요소를 초기화하는 데는 사용할 `make_unique` 수 없습니다.

C++

```
// Create a unique_ptr to an array of 5 integers.
auto p = make_unique<int[]>(5);

// Initialize the array.
for (int i = 0; i < 5; ++i)
{
    p[i] = i;
    wcout << p[i] << endl;
}
```

자세한 예제는 [make_unique 참조](#)하세요.

참고 항목

스마트 포인터(모던 C++)

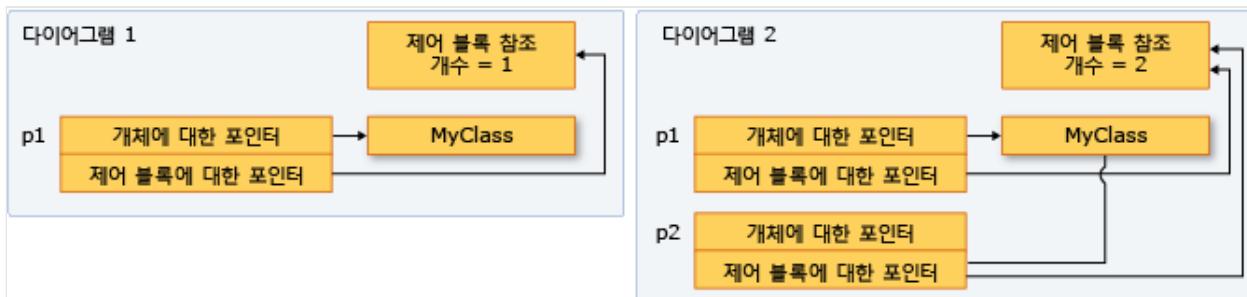
make_unique

방법: shared_ptr 인스턴스 만들기 및 사용

아티클 • 2024. 03. 22.

`shared_ptr` 형식은 둘 이상의 소유자가 개체의 수명을 관리해야 하는 시나리오를 위해 디자인된 C++ 표준 라이브러리의 스마트 포인터입니다. `shared_ptr`을 초기화한 후 복사, 함수 인수의 값으로 전달 및 다른 `shared_ptr` 인스턴스로 할당할 수 있습니다. 모든 인스턴스는 동일한 개체를 가리키고 새 `shared_ptr`이 추가되거나 범위를 벗어나거나 다시 설정될 때마다 하나의 "제어 블록"에 대한 액세스를 공유합니다. 참조 횟수가 0에 도달하면 메모리 리소스 및 제어 블록이 삭제됩니다.

다음 그림에서는 한 개의 메모리 위치를 가리키는 여러 `shared_ptr` 인스턴스를 보여 줍니다.



예제 설정

다음에 나오는 예제에서는 다음과 같이 필수 헤더를 포함하고 필요한 형식을 선언했다고 가정합니다.

C++

```
// shared_ptr-examples.cpp
// The following examples assume these declarations:
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

struct MediaAsset
{
    virtual ~MediaAsset() = default; // make it polymorphic
};

struct Song : public MediaAsset
{
```

```

    std::wstring artist;
    std::wstring title;
    Song(const std::wstring& artist_, const std::wstring& title_) :
        artist{ artist_ }, title{ title_ } {}
};

struct Photo : public MediaAsset
{
    std::wstring date;
    std::wstring location;
    std::wstring subject;
    Photo(
        const std::wstring& date_,
        const std::wstring& location_,
        const std::wstring& subject_) :
        date{ date_ }, location{ location_ }, subject{ subject_ } {}
};

using namespace std;

int main()
{
    // The examples go here, in order:
    // Example 1
    // Example 2
    // Example 3
    // Example 4
    // Example 6
}

```

예 1

가능하면 메모리 리소스를 처음으로 만들 때 `make_shared` 함수를 사용하여 `shared_ptr`을 만듭니다. `make_shared`는 예외로부터 안전합니다. 동일한 호출을 사용하여 제어 블록 및 리소스에 대한 메모리를 할당하므로 생성 오버헤드가 감소됩니다. `make_shared`를 사용하지 않는 경우 `shared_ptr` 생성자에 전달하기 전에 명시적 `new` 식을 사용하여 개체를 만들어야 합니다. 다음 예제에서는 새 개체와 함께 `shared_ptr`을 선언하고 초기화하는 다양한 방법을 보여 줍니다.

C++

```

// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"I'm Happy Just to Dance With You");

// Ok, but slightly less efficient.
// Note: Using new expression as constructor argument
// creates no named variable for other code to access.

```

```

shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

// When initialization must be separate from declaration, e.g. class
// members,
// initialize with nullptr to make your programming intent explicit.
shared_ptr<Song> sp5(nullptr);
//Equivalent to: shared_ptr<Song> sp5;
//...
sp5 = make_shared<Song>(L"Elton John", L"I'm Still Standing");

```

예제 2

다음 예제에서는 다른 `shared_ptr`로 할당된 객체의 소유권을 공유하는 `shared_ptr` 인스턴스를 선언하고 초기화하는 방법을 보여 줍니다. `sp2`가 초기화된 `shared_ptr`임을 가정 하십시오.

C++

```

//Initialize with copy constructor. Increments ref count.
auto sp3(sp2);

//Initialize via assignment. Increments ref count.
auto sp4 = sp2;

//Initialize with nullptr. sp7 is empty.
shared_ptr<Song> sp7(nullptr);

// Initialize with another shared_ptr. sp1 and sp2
// swap pointers as well as ref counts.
sp1.swap(sp2);

```

예 3

`shared_ptr`은 요소를 복사하는 알고리즘을 사용할 때 C++ 표준 템플릿 라이브러리 컨테이너에서도 유용합니다. `shared_ptr`에 요소를 래핑한 다음 내부 메모리가 필요할 때까지 유효하거나 더 이상 유효하지 않음을 인식하며 해당 요소를 다른 컨테이너에 복사할 수 있습니다. 다음 예제에서는 벡터의 `remove_copy_if` 인스턴스에서 `shared_ptr` 알고리즘을 사용하는 방법을 보여 줍니다.

C++

```

vector<shared_ptr<Song>> v {
    make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"),
    make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"),
    make_shared<Song>(L"Thalía", L"Entre El Mar y Una Estrella")
};

```

```

vector<shared_ptr<Song>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2), [] (shared_ptr<Song> s)
{
    return s->artist.compare(L"Bob Dylan") == 0;
});

for (const auto& s : v2)
{
    wcout << s->artist << L":" << s->title << endl;
}

```

예시 4

`dynamic_pointer_cast`, `static_pointer_cast` 및 `const_pointer_cast`를 사용하여 `shared_ptr`을 캐스팅할 수 있습니다. 이러한 함수는 `dynamic_cast`, `static_cast` 및 `const_cast` 연산자와 비슷합니다. 다음 예제에서는 기본 클래스에서 `shared_ptr`의 벡터에 있는 각 요소의 파생 형식을 테스트한 다음 요소를 복사하고 이에 대한 정보를 표시하는 방법을 보여 줍니다.

C++

```

vector<shared_ptr<MediaAsset>> assets {
    make_shared<Song>(L"Himesh Reshammiya", L"Tera Surroor"),
    make_shared<Song>(L"Penaz Masani", L"Tu Dil De De"),
    make_shared<Photo>(L"2011-04-06", L"Redmond, WA", L"Soccer field at Microsoft.")
};

vector<shared_ptr<MediaAsset>> photos;

copy_if(assets.begin(), assets.end(), back_inserter(photos), []
(shared_ptr<MediaAsset> p) -> bool
{
    // Use dynamic_pointer_cast to test whether
    // element is a shared_ptr<Photo>.
    shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(p);
    return temp.get() != nullptr;
});

for (const auto& p : photos)
{
    // We know that the photos vector contains only
    // shared_ptr<Photo> objects, so use static_cast.
    wcout << "Photo location: " << (static_pointer_cast<Photo>(p))->location
    << endl;
}

```

예제 5

`shared_ptr` 을 다음과 같은 방법으로 다른 함수에 전달할 수 있습니다.

- `shared_ptr` 을 값으로 전달합니다. 이 방법은 복사 생성자를 호출하고 참조 횟수를 늘리며 호출 수신자를 소유자로 변경합니다. 이 작업에는 적은 양의 오버헤드가 있으며 이는 전달하는 상당 수의 `shared_ptr` 개체 수에 따라 달라질 수 있습니다. 호출자와 호출 수신자 간의 암시적 또는 명시적 코드 계약이 호출 수신자가 소유자일 것을 요구하는 경우 이 옵션을 사용합니다.
- 참조 또는 `const` 참조로 `shared_ptr` 을 전달합니다. 이 경우 참조 횟수는 늘어나지 않고 호출 수신자는 호출자가 범위를 벗어나지 않는 한 포인터에 액세스할 수 있습니다. 또는 호출 수신자는 참조를 기반으로 `shared_ptr` 을 만들고 공유 소유자가 될 수 있습니다. 호출자가 호출 수신자를 모르는 경우 또는 `shared_ptr` 을 전달해야 하고 성능상의 이유로 복사 작업을 하지 않아야 할 경우 이 옵션을 사용합니다.
- 내부 포인터 또는 내부 개체에 대한 참조를 전달합니다. 이를 통해 호출 수신자는 개체를 사용하지만 소유권을 공유하거나 수명을 연장할 수 없습니다. 호출 수신자가 원시 포인터에서 `shared_ptr` 을 만드는 경우 새 `shared_ptr` 은 원본에 독립적이고 내부 리소스를 제어하지 않습니다. 호출자와 호출 수신자 사이의 계약이 호출자가 `shared_ptr` 수명에 대한 소유권을 보유함을 명확하게 지정하는 경우 이 옵션을 사용하십시오.
- `shared_ptr` 을 전달하는 방법을 결정할 때 호출 수신자가 내부 리소스의 소유권을 공유해야 하는지 여부를 확인합니다. "소유자"는 기본 리소스를 필요할 때까지 내부 리소스를 활성 상태로 유지할 수 있는 개체 또는 함수입니다. 호출 수신자가 포인터의 수명을 해당(함수의) 수명 이상으로 확장할 수 있음을 보장해야 하는 경우 첫 번째 옵션을 사용합니다. 호출 수신자의 수명 연장 여부를 고려하지 않는 경우 참조로 전달하고 호출 수신자는 이를 복사하거나 복사하지 않습니다.
- 내부 포인터에 도우미 함수 액세스를 제공해야 하고 도우미 함수가 포인터를 사용하고 호출 함수가 반환되기 전에 반환하는 것을 알고 있는 경우 해당 함수는 내부 포인터의 소유권을 공유하지 않아야 합니다. 호출자의 `shared_ptr` 수명 내에 포인터에 액세스해야 합니다. 이러한 경우 참조로 `shared_ptr` 을 전달하거나 원시 포인터 또는 참조를 내부 개체로 전달하는 것이 안전합니다. 이 방법으로 전달하면 약간의 성능 이점이 제공되고 프로그래밍 의도를 표시하는 데 유용할 수도 있습니다.
- 경우에 따라 `std::vector<shared_ptr<T>>` 의 예제에서 각 `shared_ptr` 을 람다 식 본문 또는 명명된 함수 개체로 전달해야 할 수도 있습니다. 람다 또는 함수가 포인터를 저장하지 않는 경우 참조로 `shared_ptr` 을 전달하여 각 요소에 대한 복사 생성자를 호출하지 않습니다.

C++

```
void use_shared_ptr_by_value(shared_ptr<int> sp);

void use_shared_ptr_by_reference(shared_ptr<int>& sp);
void use_shared_ptr_by_const_reference(const shared_ptr<int>& sp);

void use_raw_pointer(int* p);
void use_reference(int& r);

void test() {
    auto sp = make_shared<int>(5);

    // Pass the shared_ptr by value.
    // This invokes the copy constructor, increments the reference count,
    // and makes the callee an owner.
    use_shared_ptr_by_value(sp);

    // Pass the shared_ptr by reference or const reference.
    // In this case, the reference count isn't incremented.
    use_shared_ptr_by_reference(sp);
    use_shared_ptr_by_const_reference(sp);

    // Pass the underlying pointer or a reference to the underlying object.
    use_raw_pointer(sp.get());
    use_reference(*sp);

    // Pass the shared_ptr by value.
    // This invokes the move constructor, which doesn't increment the
    // reference count
    // but in fact transfers ownership to the callee.
    use_shared_ptr_by_value(move(sp));
}
```

예제 6

다음 예제에서는 `shared_ptr`이 `shared_ptr` 인스턴스가 소유하는 메모리의 포인터를 비교할 수 있도록 다양한 비교 연산자를 오버로드하는 방법을 보여 줍니다.

C++

```
// Initialize two separate raw pointers.
// Note that they contain the same values.
auto song1 = new Song(L"Village People", L"YMCA");
auto song2 = new Song(L"Village People", L"YMCA");

// Create two unrelated shared_ptrs.
shared_ptr<Song> p1(song1);
shared_ptr<Song> p2(song2);
```

```
// Unrelated shared_ptrs are never equal.  
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;  
wcout << "p1 == p2 = " << std::boolalpha << (p1 == p2) << endl;  
  
// Related shared_ptr instances are always equal.  
shared_ptr<Song> p3(p2);  
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;
```

참고 항목

[스마트 포인터\(모던 C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

방법: weak_ptr 인스턴스 만들기 및 사용

아티클 • 2023. 10. 12.

경우에 따라 개체는 참조 수를 증가하지 않고 shared_ptr 기본 개체에 액세스하는 방법을 저장해야 합니다. 일반적으로 이 상황은 인스턴스 간에 shared_ptr 순환 참조가 있는 경우에 발생합니다.

최상의 디자인은 가능하면 언제든지 포인터의 공유 소유권을 방지하는 것입니다. 그러나 인스턴스의 shared_ptr 공유 소유권이 있어야 하는 경우 인스턴스 간에 순환 참조를 사용하지 않습니다. 순환 참조가 불가피하거나 어떤 이유로든 바람직한 경우 weak_ptr 사용하여 하나 이상의 소유자에게 다른 shared_ptr 소유자에 대한 약한 참조를 제공합니다. 이를 weak_ptr 사용하여 기존 관련 인스턴스 집합에 조인하는 조인을 만들 수 있지만 기본 메모리 리소스가 여전히 유효한 경우에만 만들 shared_ptr 수 있습니다. 자체는 weak_ptr 참조 계산에 참여하지 않으므로 참조 수가 0으로 가는 것을 방지할 수 없습니다. 그러나 a를 weak_ptr 사용하여 초기화된 새 복사본을 shared_ptr 가져올 수 있습니다. 메모리가 이미 삭제 weak_ptr 된 경우 's bool 연산자는 .false 메모리가 여전히 유효한 경우 새 공유 포인터는 참조 수를 증가시키고 변수가 범위에 유지되는 한 shared_ptr 메모리가 유효할 수 있도록 보장합니다.

예시

다음 코드 예제에서는 순환 종속성이 있는 weak_ptr 개체의 적절한 삭제를 보장하는 데 사용되는 경우를 보여줍니다. 예제를 검사할 때 대체 솔루션을 고려한 후에만 만들어진 것으로 가정합니다. 개체는 Controller 컴퓨터 프로세스의 일부 측면을 나타내며 독립적으로 작동합니다. 각 컨트롤러는 언제든지 다른 컨트롤러의 상태 쿼리할 수 있어야 하며 각 컨트롤러에는 이 목적을 위한 프라이빗 vector<weak_ptr<Controller>> 이 포함되어 있습니다. 각 벡터에는 순환 참조가 포함되므로 weak_ptr 인스턴스가 대신 shared_ptr 사용됩니다.

C++

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Controller
{
public:
```

```

int Num;
wstring Status;
vector<weak_ptr<Controller>> others;
explicit Controller(int i) : Num(i), Status(L"On")
{
    wcout << L"Creating Controller" << Num << endl;
}

~Controller()
{
    wcout << L"Destroying Controller" << Num << endl;
}

// Demonstrates how to test whether the
// pointed-to memory still exists or not.
void CheckStatuses() const
{
    for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp) {
        auto p = wp.lock();
        if (p)
        {
            wcout << L"Status of " << p->Num << " = " << p->Status << endl;
        }
        else
        {
            wcout << L"Null object" << endl;
        }
    });
}

void RunTest()
{
    vector<shared_ptr<Controller>> v{
        make_shared<Controller>(0),
        make_shared<Controller>(1),
        make_shared<Controller>(2),
        make_shared<Controller>(3),
        make_shared<Controller>(4),
    };

    // Each controller depends on all others not being deleted.
    // Give each controller a pointer to all the others.
    for (int i = 0; i < v.size(); ++i)
    {
        for_each(v.begin(), v.end(), [&v, i](shared_ptr<Controller> p) {
            if (p->Num != i)
            {
                v[i]->others.push_back(weak_ptr<Controller>(p));
                wcout << L"push_back to v[" << i << "]": " << p->Num << endl;
            }
        });
    }

    for_each(v.begin(), v.end(), [](&shared_ptr<Controller> &p) {

```

```
wcout << L"use_count = " << p.use_count() << endl;
p->CheckStatuses();
});

int main()
{
    RunTest();
    wcout << L"Press any key" << endl;
    char ch;
    cin.getline(&ch, 1);
}
```

Output

```
Creating Controller0
Creating Controller1
Creating Controller2
Creating Controller3
Creating Controller4
push_back to v[0]: 1
push_back to v[0]: 2
push_back to v[0]: 3
push_back to v[0]: 4
push_back to v[1]: 0
push_back to v[1]: 2
push_back to v[1]: 3
push_back to v[1]: 4
push_back to v[2]: 0
push_back to v[2]: 1
push_back to v[2]: 3
push_back to v[2]: 4
push_back to v[3]: 0
push_back to v[3]: 1
push_back to v[3]: 2
push_back to v[3]: 4
push_back to v[4]: 0
push_back to v[4]: 1
push_back to v[4]: 2
push_back to v[4]: 3
use_count = 1
Status of 1 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
```

```
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 3 = On
Destroying Controller0
Destroying Controller1
Destroying Controller2
Destroying Controller3
Destroying Controller4
Press any key
```

실험으로 벡터를 `others` a `vector<shared_ptr<Controller>>`로 수정한 다음 출력에서 반환될 때 `RunTest` 소멸자가 호출되지 않습니다.

참고 항목

[스마트 포인터\(모던 C++\)](#)

방법: CComPtr 및 CComQIPtr 인스턴스 만들기 및 사용

아티클 • 2023. 04. 03.

클래식 Windows 프로그래밍에서 라이브러리는 종종 COM 개체(보다 정확하게는 COM 서버)로 구현됩니다. 많은 Windows 운영 체제 구성 요소가 COM 서버로 구현되므로 많은 참가자가 이 형식의 라이브러리를 제공합니다. COM의 기본 사항에 대한 자세한 내용은 [Component Object Model \(COM\)](#)을 참조하세요.

COM(구성 요소 개체 모델) 개체를 인스턴스화할 때 소멸자에서 `AddRef` 및 `Release`에 대한 호출을 사용하여 참조 계산을 수행하는 COM 스마트 포인터에 인터페이스 포인터를 저장합니다. ATL(액티브 템플릿 라이브러리) 또는 MFC 라이브러리를 사용하는 경우 `CComPtr` 스마트 포인터를 사용합니다. ATL 또는 MFC를 사용하지 않는 경우에는 `_com_ptr_t`를 사용합니다. `std::unique_ptr`에 해당하는 COM이 없기 때문에 단일 소유자 시나리오와 여러 소유자 시나리오 모두에 이러한 스마트 포인터를 사용합니다. `CComPtr` 과 `ComQIPtr` 둘 다 rvalue 참조가 있는 이동 작업을 지원합니다.

예: CComPtr

다음 예제에서는 `CComPtr` 을 사용하여 COM 개체를 인스턴스화하고 해당 인터페이스에 대한 포인터를 가져오는 방법을 보여 줍니다. `CComPtr::CoCreateInstance` 멤버 함수는 이름이 같은 Win32 함수 대신 COM 개체를 만드는 데 사용됩니다.

C++

```
void CComPtrDemo()
{
    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr<IGraphBuilder> pGraph;

    // Use its member function CoCreateInstance to
    // create the COM object and obtain the IGraphBuilder pointer.
    hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
    if(Failed(hr)){ /*... handle hr error*/ }

    // Use the overloaded -> operator to call the interface methods.
    hr = pGraph->RenderFile(L"C:\\\\Users\\\\Public\\\\Music\\\\Sample Music\\\\Sleep
    Away.mp3", NULL);
    if(Failed(hr)){ /*... handle hr error*/ }

    // Declare a second smart pointer and use it to
}
```

```

// obtain another interface from the object.
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// Obtain a third interface.
CComPtr<IMediaEvent> pEvent;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pEvent));
if(FAILED(hr)){ /*... handle hr error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }

// Use the third interface.
long evCode = 0;
pEvent->WaitForCompletion(INFINITE, &evCode);

CoUninitialize();

// Let the smart pointers do all reference counting.
}

```

`CComPtr` 및 해당 상대는 ATL의 일부이며 `atlcomcli.h`에 정의됩니다.

`_com_ptr_t comip.h`에서 선언됩니다. 컴파일러는 형식 라이브러리에 대한 래퍼 클래스를 생성할 때 `_com_ptr_t` 의 특수화를 만듭니다.

예: `CComQIPtr`

또한 ATL은 COM 개체를 쿼리하여 추가 인터페이스를 검색할 수 있도록 보다 간단한 구문이 있는 `CComQIPtr`을 제공합니다. 그러나 `CComPtr`에서 수행할 수 있는 모든 작업을 수행하고 원시 COM 인터페이스 포인터와 의미 체계가 보다 일치하므로 `CComQIPtr`을 사용하는 것이 좋습니다. `CComPtr`를 사용하여 인터페이스를 쿼리하는 경우 `out` 매개 변수에 새 인터페이스 포인터가 배치됩니다. 호출에 실패한 경우 일반적인 COM 패턴인 `HRESULT`가 반환됩니다. `CComQIPtr`을 사용하는 경우 반환 값은 포인터 자체이며, 호출에 실패한 경우 내부 `HRESULT` 반환 값에 액세스할 수 없습니다. 다음 두 줄은 `CComPtr`과 `CComQIPtr`의 오류 처리 메커니즘 차이를 보여 줍니다.

C++

```

// CComPtr with error handling:
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// CComQIPtr with error handling
CComQIPtr<IMediaEvent> pEvent = pControl;

```

```

if(!pEvent){ /*... handle NULL pointer error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }

```

예: IDispatch

`CComPtr` 은 COM 자동화 구성 요소에 대한 포인터를 저장하고 런타임에 바인딩을 사용하여 인터페이스에서 메서드를 호출할 수 있도록 하는 `IDispatch` 특수화를 제공합니다. `CComDispatchDriver` 는 암시적으로 `CComQIPtr<IDispatch, &IID_IDispatch>`로 변환할 수 있는 `CComPtr<IDispatch>`에 대한 `typedef`입니다. 따라서 이 세 가지 이름 중 하나가 코드에 표시되면 이는 `CComPtr<IDispatch>`와 같습니다. 다음 예제에서는 `CComPtr<IDispatch>`를 사용하여 Microsoft Word 개체 모델에 대한 포인터를 가져오는 방법을 보여 줍니다.

C++

```

void COMAutomationSmartPointerDemo()
{
    CComPtr<IDispatch> pWord;
    CComQIPtr<IDispatch, &IID_IDispatch> pqi = pWord;
    CComDispatchDriver pDriver = pqi;

    HRESULT hr;
    _variant_t pOutVal;

    CoInitialize(NULL);
    hr = pWord.CoCreateInstance(L"Word.Application", NULL,
        CLSCTX_LOCAL_SERVER);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Make Word visible.
    hr = pWord.PutPropertyByName(_bstr_t("Visible"), &_variant_t(1));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Get the Documents collection and store it in new CComPtr
    hr = pWord.GetPropertyByName(_bstr_t("Documents"), &pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CComPtr<IDispatch> pDocuments = pOutVal.pdispVal;

    // Use Documents to open a document
    hr = pDocuments.Invoke1 (_bstr_t("Open"),
    &_variant_t("c:\\\\users\\\\public\\\\documents\\\\sometext.txt"),&pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CoUninitialize();
}

```

추가 정보

[스마트 포인터\(최신 C++\)](#)

MSVC의 예외 처리

아티클 • 2023. 10. 12.

예외는 프로그램이 일반적인 실행 경로를 따라 계속 진행하는 것을 방해하며 프로그램의 제어를 벗어날 수 있는 오류 상태입니다. 개체 만들기, 파일 입력/출력 및 다른 모듈에서 수행된 함수 호출을 비롯한 특정 작업은 프로그램이 올바르게 실행되는 경우에도 예외의 잠재적인 원인입니다. 강력한 코드는 예외를 예상하고 처리합니다. 논리 오류를 검색하면 예외가 아닌 어설션을 사용합니다(어설션 [사용 참조](#)).

예외 종류

MSVC(Microsoft C++ 컴파일러)는 다음 세 가지 종류의 예외 처리를 지원합니다.

- [C++ 예외 처리](#)

대부분의 C++ 프로그램의 경우 C++ 예외 처리를 사용해야 합니다. 형식이 안전하며 스택 해제 중에 개체 소멸자가 호출되도록 합니다.

- [구조적 예외 처리](#)

Windows는 SEH(구조적 예외 처리)라는 자체 예외 메커니즘을 제공합니다. C++ 또는 MFC 프로그래밍은 권장되지 않습니다. MFC C가 아닌 프로그램에서만 SEH를 사용합니다.

- [MFC 예외](#)

버전 3.0부터 MFC는 C++ 예외를 사용했습니다. 형식의 C++ 예외와 유사한 이전 예외 처리 매크로를 계속 지원합니다. MFC 매크로와 C++ 예외를 혼합하는 방법에 대한 조언은 예외: MFC 매크로 및 C++ 예외를 사용하는 경우를 참조 [하세요](#).

`/EH` 컴파일러 옵션을 사용하여 C++ 프로젝트에서 사용할 예외 처리 모델을 지정합니다. 표준 C++ 예외 처리(`/EHsc`)는 Visual Studio의 새 C++ 프로젝트에서 기본값입니다.

예외 처리 메커니즘을 혼합하지 않는 것이 좋습니다. 예를 들어 구조적 예외 처리에는 C++ 예외를 사용하지 마세요. C++ 예외 처리를 단독으로 사용하면 코드의 이식성이 높아지고 모든 형식의 예외를 처리할 수 있습니다. 구조적 예외 처리의 단점에 대한 자세한 내용은 구조적 예외 처리를 참조 [하세요](#).

이 섹션의 내용

- [예외 및 오류 처리에 대한 최신 C++ 모범 사례](#)

- 예외 안전을 위해 디자인하는 방법
- 예외 코드와 예외 없는 코드 간에 인터페이스하는 방법
- try, catch 및 throw 문
- Catch 블록의 평가 방법
- 예외 및 스택 해제
- 예외 사양
- noexcept
- 처리되지 않은 C++ 예외
- C(구조적) 및 C++ 예외 혼합
- SEH(구조적 예외 처리)(C/C++)

참고 항목

C++ 언어 참조

x64 예외 처리 예외 처리
(C++/CLI 및 C++/CX)

최신 C++ 예외 및 오류 처리 모범 사례

아티클 • 2024. 03. 28.

최신 C++의 경우 대부분의 시나리오에서 논리 오류와 런타임 오류를 모두 보고하고 처리하기 위해 기본적으로 예외를 사용합니다. 스택에 오류를 감지하는 함수와 오류를 처리할 컨텍스트가 있는 함수 간에 여러 함수 호출이 포함될 수 있는 경우 특히 그렇습니다. 예외는 호출 스택에 정보를 전달하는 오류를 감지하는 코드에 대해 잘 정의된 공식적인 방법을 제공합니다.

예외 코드에 예외 사용

프로그램 오류는 다음과 같은 두 가지 범주로 나뉘는 경우가 많습니다.

- 첫 번째는 프로그래밍 실수로 인한 논리 오류입니다. 예를 들어 "범위를 벗어난 인덱스" 오류입니다.
- 두 번째는 프로그래머가 제어할 수 없는 런타임 오류입니다. 예를 들어 "네트워크 서비스를 사용할 수 없음" 오류가 있습니다.

C 스타일 프로그래밍 및 COM에서 오류 보고는 특정 함수에 대한 오류 코드 또는 상태 코드를 나타내는 값을 반환하거나, 오류가 보고되었는지 여부를 확인하기 위해 모든 함수 호출 후에 호출자가 선택적으로 검색할 수 있는 전역 변수를 설정하여 관리됩니다. 예를 들어 COM 프로그래밍은 `HRESULT` 반환 값을 사용하여 호출자에게 오류를 전달합니다. 또한 Win32 API에는 호출 스택에서 보고한 마지막 오류를 검색하는 `GetLastError` 함수가 있습니다. 두 경우 모두 코드를 인식하고 적절하게 응답하는 일은 호출자가 담당합니다. 호출자가 오류 코드를 명시적으로 처리하지 않으면 프로그램이 경고 없이 충돌할 수 있습니다. 또는 잘못된 데이터를 사용하여 계속 실행하고 잘못된 결과를 생성할 수 있습니다.

최신 C++에서 예외 사용이 선호되는 이유는 다음과 같습니다.

- 예외는 코드를 호출하여 오류 조건을 인식하고 처리하도록 강제합니다. 처리되지 않은 예외는 프로그램 실행을 중지합니다.
- 예외는 호출 스택에서 오류를 처리할 수 있는 지점으로 이동합니다. 중간 함수는 예외가 전파되도록 할 수 있으며, 다른 레이어와 조정할 필요는 없습니다.
- 예외 스택 해제 메커니즘은 잘 정의된 규칙에 따라 예외가 `throw`된 후 범위의 모든 개체를 삭제합니다.
- 예외를 사용하면 오류를 감지하는 코드와 오류를 처리하는 코드 사이를 완전히 분리할 수 있습니다.

다음 간소화된 예제에서는 C++에서 예외를 `throw`하고 `catch`하는 데 필요한 구문을 보여줍니다.

C++

```
#include <stdexcept>
#include <limits>
#include <iostream>

using namespace std;

void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
    {
        throw invalid_argument("MyFunc argument too large.");
    }
    //...
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}
```

C++의 예외는 C# 및 Java와 같은 언어의 예외와 유사합니다. `try` 블록에서 예외가 `throw` 되면 해당 형식이 예외의 형식과 일치하는 첫 번째 연결된 `catch` 블록에 의해 `catch`됩니다. 즉, 실행은 `throw` 문에서 `catch` 문으로 이동합니다. 사용할 수 있는 `catch` 블록이 없으면 `std::terminate` 가 호출되고 프로그램이 종료됩니다. C++에서는 어떤 형식이든 `throw` 될 수 있지만 `std::exception`에서 직접 또는 간접적으로 파생된 형식을 `throw`하는 것이 좋습니다. 이전 예제에서 예외 형식 `invalid_argument`는 `<stdexcept>` 헤더 파일의 표준 라이브러리에 정의되어 있습니다. C++는 예외가 `throw`될 경우 모든 리소스가 해제되도록 하기 위해 `finally` 블록을 제공하거나 요구하지 않습니다. 리소스 취득은 스마트 포인터를 사용하는 초기화(RAII) 관용구로 리소스 정리에 필요한 기능을 제공합니다. 자세한 내용은 [방법: 예외 안전 디자인](#)을 참조하세요. C++ 스택 해제 메커니즘에 대한 자세한 내용은 [예외 및 스택 해제](#)를 참조하세요.

기본 지침

어떤 프로그래밍 언어든 강력한 오류 처리는 까다로운 일입니다. 예외는 뛰어난 오류 처리를 지원하는 몇 가지 기능을 제공하지만 모든 작업을 대신할 수는 없습니다. 예외 메커니즘의 이점을 실현하려면 코드를 디자인할 때 예외를 염두에 두어야 합니다.

- 어설션을 사용하여 항상 `true`이거나 항상 `false`여야 하는 조건을 확인합니다. 예외를 사용하여 발생할 수 있는 오류(예: 공용 함수의 매개 변수에 대한 입력 유효성 검사 오류)를 확인합니다. 자세한 내용은 [예외와 어설션 비교](#) 섹션을 참조하세요.
- 오류를 처리하는 코드가 하나 이상의 중간 함수 호출로 오류를 감지하는 코드와 분리된 경우 예외를 사용합니다. 오류를 처리하는 코드가 오류를 감지하는 코드와 긴밀하게 결합된 경우 성능이 중요한 루프에서 오류 코드를 대신 사용할지 여부를 고려합니다.
- 예외를 `throw`하거나 전파할 수 있는 모든 함수의 경우 강력한 보장, 기본 보증 또는 `nothrow`(`noexcept`) 보장의 세 가지 예외 보장 중 하나를 제공합니다. 자세한 내용은 [방법: 예외 안전 디자인](#)을 참조하세요.
- 값으로 예외를 `throw`하고 참조로 `catch`합니다. 처리할 수 없는 항목은 `catch`하지 않습니다.
- C++11에서 사용되지 않는 예외 사양은 사용하지 않습니다. 자세한 내용은 [예외 사양 및 noexcept](#) 섹션을 참조하세요.
- 적용되는 경우 표준 라이브러리 예외 형식을 사용합니다. [exception 클래스](#) 계층 구조에서 사용자 지정 예외 형식을 파생합니다.
- 예외가 소멸자 또는 메모리 할당 취소 함수에서 이스케이프되는 것을 허용하지 않습니다.

예외 및 성능

예외가 `throw`되지 않는 경우 예외 메커니즘의 성능 비용은 최소화됩니다. 예외가 `throw`되면 스택 통과 및 해제 비용은 함수 호출 비용과 거의 비슷합니다. `try` 블록을 입력한 후 호출 스택을 추적하려면 다른 데이터 구조가 필요하며, 예외가 `throw`될 경우 스택을 해제하려면 더 많은 지침이 필요합니다. 그러나 대부분의 시나리오에서는 성능 및 메모리 공간의 비용이 크지 않습니다. 예외가 성능에 미치는 악영향은 메모리가 제한된 시스템에서만 중요할 수 있습니다. 또는 성능이 중요한 루프에서는 오류가 정기적으로 발생할 가능성이 높고 이를 처리하는 코드와 보고하는 코드가 긴밀하게 결합되어 있습니다. 어떤 경우에도 프로파일링 및 측정 없이는 예외의 실제 비용을 알 수 없습니다. 드물게 비용이 많이 드는 경우라도 잘 설계된 예외 정책을 통해 얻을 수 있는 정확성 향상, 유지 관리 용이성 및 기타 이점과 비교하여 비용을 따져볼 수 있습니다.

예외와 어설션 비교

예외와 어설션은 프로그램에서 런타임 오류를 검색하기 위한 두 가지 고유한 메커니즘입니다. `assert` 문을 사용하여 개발 중에 항상 `true`이거나 모든 코드가 올바른 경우 항상

`false`여야 하는 조건을 테스트합니다. 오류는 코드의 항목을 수정해야 하므로 예외를 사용하여 이러한 오류를 처리할 필요가 없습니다. 프로그램이 런타임에 복구해야 하는 조건을 나타내지 않습니다. `assert`는 디버거에서 프로그램 상태를 검사할 수 있도록 문에서 실행을 중지합니다. 예외는 첫 번째 적절한 `catch` 처리기에서 실행을 계속합니다. 예외를 사용하여 코드가 올바른 경우에도 런타임에 발생할 수 있는 오류 조건(예: "파일을 찾을 수 없음" 또는 "메모리 부족")을 확인합니다. 복구에서 로그에 메시지를 출력하고 프로그램을 종료하더라도 예외는 이러한 조건을 처리할 수 있습니다. 항상 예외를 사용하여 공용 함수에 대한 인수를 확인합니다. 함수에 오류가 없는 경우에도 사용자가 전달할 수 있는 인수를 완전히 제어하지 못할 수 있습니다.

C++ 예외와 Windows SEH 예외 비교

C 및 C++ 프로그램 모두 Windows 운영 체제에서 SEH(구조적 예외 처리) 메커니즘을 사용할 수 있습니다. SEH의 개념은 C++ 예외의 개념과 유사합니다. 단, SEH는 `try` 및 `catch` 대신 `_try`, `_except` 및 `_finally` 구문을 사용합니다. MSVC(Microsoft C++ 컴파일러)에서 C++ 예외는 SEH에 대해 구현됩니다. 그러나 C++ 코드를 작성할 때는 C++ 예외 구문을 사용합니다.

SEH에 대한 자세한 내용은 [구조적 예외 처리\(C/C++\)](#)를 참조하세요.

예외 사양 및 `noexcept`

예외 사양은 함수가 `throw`할 수 있는 예외를 지정하는 방법으로 C++에 처음 도입되었습니다. 그러나 예외 사양은 실제로 문제가 있는 것으로 판명되었으며 C++11 초안 표준에서는 더 이상 사용되지 않습니다. 함수에서 예외의 이스케이프를 허용하지 않음을 나타내는 `throw()`를 제외하고 `throw` 예외 사양은 사용하지 않는 것이 좋습니다. 사용되지 않는 양식의 예외 사양인 `throw(type-name)`를 사용해야 하는 경우 MSVC 지원이 제한됩니다. 자세한 내용은 [예외 사양\(throw\)](#)을 참조하세요. `noexcept` 지정자는 `throw()` 대신 선호되는 대안으로 C++11에 도입되었습니다.

참고 항목

[방법: 예외 코드와 예외가 아닌 코드 간의 인터페이싱](#)

[C++ 언어 참조](#)

[C++ 표준 라이브러리](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

방법: 예외 안전을 위한 디자인

아티클 • 2023. 06. 16.

예외에 대한 데이터와 함께 예외 메커니즘의 장점 중 하나는, 예외를 throw하는 문에서 이를 처리하는 첫 번째 catch 문으로 직접 이동할 수 있다는 점입니다. 처리기는 호출 스택에서 아무 상위 수준에나 있을 수 있습니다. try 문과 throw 문 사이에 호출되는 함수는 throw된 예외에 대한 정보를 알아야 할 필요가 없습니다. 하지만 예외가 위쪽으로 전파될 수 있는 어느 지점에서든 "예기치 않게" 범위 밖으로 이동할 수 있고, 부분적으로 생성된 개체, 누출된 메모리 또는 불안정한 상태의 데이터 구조를 벗어나지 않고도 그렇게 할 수 있도록 설계되어야 합니다.

기본 기술

강력한 예외 처리 정책을 위해서는 신중한 계획이 필요하며 디자인 프로세스의 일부에 포함되어야 합니다. 일반적으로 대부분의 예외는 소프트웨어 모듈의 하위 계층에서 검색되고 throw됩니다. 하지만 일반적으로 이러한 계층은 오류를 처리하거나 최종 사용자에게 메시지를 제공하기에 충분한 컨텍스트를 갖지 못합니다. 중간 계층에서 함수는 예외 개체를 검사할 때 예외를 catch하고 다시 throw할 수 있거나, 궁극적으로 예외를 catch하는 상위 계층에 대해 제공할 추가적인 유용한 정보를 포함합니다. 함수는 예외를 완전히 복구할 수 없는 경우에만 예외를 catch하고 "무시"해야 합니다. 대부분의 경우, 중간 계층에서 올바른 동작은 예외가 호출 스택으로 전파되도록 하는 것입니다. 최상의 계층의 경우에서도 정확성을 보장할 수 없는 상태로 예외가 프로그램을 벗어나는 경우 처리되지 않은 예외로 인해 프로그램이 종료되도록 두는 것이 적합할 수 있습니다.

함수가 예외를 어떻게 처리하든 간에, "예외에 대한 안전성"을 보장하기 위해서는 다음과 같은 기본 규칙에 따라 설계해야 합니다.

리소스 클래스를 단순하게 유지

클래스에서 수동 리소스 관리를 캡슐화하는 경우 단일 리소스 관리 외에는 아무 작업도 수행하지 않는 클래스를 사용합니다. 클래스를 단순하게 유지하면 리소스 누출이 발생할 위험을 줄일 수 있습니다. 다음 예제와 같이 가능하면 [스마트 포인터](#)를 사용합니다. 이 예제는 의도적으로 만들어졌으며, `shared_ptr`이 사용될 때의 차이점을 보여주기 위해 단순화한 것입니다.

C++

```
// old-style new/delete version
class NDResourceClass {
private:
    int* m_p;
```

```

    float* m_q;
public:
    NDResourceClass() : m_p(0), m_q(0) {
        m_p = new int;
        m_q = new float;
    }

    ~NDResourceClass() {
        delete m_p;
        delete m_q;
    }
    // Potential leak! When a constructor emits an exception,
    // the destructor will not be invoked.
};

// shared_ptr version
#include <memory>

using namespace std;

class SPResourceClass {
private:
    shared_ptr<int> m_p;
    shared_ptr<float> m_q;
public:
    SPResourceClass() : m_p(new int), m_q(new float) { }
    // Implicitly defined dtor is OK for these members,
    // shared_ptr will clean up and avoid leaks regardless.
};

// A more powerful case for shared_ptr

class Shape {
    // ...
};

class Circle : public Shape {
    // ...
};

class Triangle : public Shape {
    // ...
};

class SPSHapeResourceClass {
private:
    shared_ptr<Shape> m_p;
    shared_ptr<Shape> m_q;
public:
    SPSHapeResourceClass() : m_p(new Circle), m_q(new Triangle) { }
};

```

RaII 관용구를 사용하여 리소스 관리

예외로부터 안전하려면 함수는 또는 `new` 를 사용하여 `malloc` 할당한 개체가 제거되고, 예외가 `throw`되더라도 파일 핸들과 같은 모든 리소스가 닫혀 있거나 해제되도록 해야 합니다. RAII(리소스 획득 초기화) 관용구는 이러한 리소스의 관리를 자동 변수의 수명과 연결합니다. 정상적인 반환 또는 예외로 인해 함수가 범위를 벗어나면 모든 완전히 생성된 자동 변수에 대한 소멸자가 호출됩니다. 스마트 포인터와 같은 RAII 래퍼 개체는 해당 소멸자에서 적합한 `delete` 또는 `close` 함수를 호출합니다. 예외로부터 안전한 코드에서는 각 리소스의 소유권을 특정 종류의 RAII 개체로 즉시 전달하는 것이 매우 중요합니다. `vector`, `string`, `make_shared` 및 `fstream` 유사한 클래스는 리소스 획득을 처리합니다. 그러나 `unique_ptr` 기존 `shared_ptr` 생성은 개체 대신 사용자가 리소스를 획득하기 때문에 특별합니다. 따라서 리소스 릴리스는 소멸 이지만 RAII로 의심됩니다.

세 가지 예외가 보장됩니다.

일반적으로 예외 안전은 함수가 제공할 수 있는 세 가지 예외 보장(실패 없음 보장, 강력한 보장 및 기본 보장)에 대해 설명합니다.

실패 없음 보장

오류 없음(?또는 "throw 없음") 보증은 함수가 제공할 수 있는 가장 강력한 보증입니다. 이 형태의 보증에서 함수는 예외를 `throw`하지 않거나 예외가 전파되는 것을 허용하지 않습니다. 하지만 (a) 이 함수가 호출하는 모든 함수도 오류 없음 보장을 제공하는지 알고 있고, (b) `throw`되는 모든 예외가 이 함수에 도달하기 전에 `catch`된다라는 것을 알고 있고, (c) 이 함수에 도달할 수 있는 모든 예외를 `catch`하고 올바르게 처리하는 방법을 알고 있는 경우를 제외하고는 그러한 보증을 안정적으로 제공할 수 없을 것입니다.

강력한 보증 및 기본 보증은 소멸자에 대한 오류가 없다는 가정에 의존합니다. 표준 라이브러리의 모든 컨테이너 및 형식은 소멸자가 예외를 `throw`하지 않음을 보증합니다. 또한 반대되는 요구 사항도 있습니다. 표준 라이브러리의 경우 템플릿 인수와 같이 여기에 제공되는 사용자 정의된 형식은 예외를 `throw`하지 않는 소멸자를 포함해야 합니다.

강력한 보장

강력한 보증에서는 함수가 예외로 인해 범위를 벗어날 경우 메모리를 누출하지 않고 프로그램 상태가 수정되지 않아야 합니다. 강력한 보증을 제공하는 함수는 기본적으로 커밋 또는 롤백 의미를 갖는 트랜잭션입니다. 즉, 완전히 성공하거나, 아예 아무런 영향도 주지 않습니다.

기본 보장

기본 보증은 세 가지 중에서 가장 약한 보증입니다. 하지만 메모리 소비 또는 성능 면에서 강력한 보증에 대한 비용이 너무 높을 경우에는 최상의 선택일 수 있습니다. 기본 보증에서는 예외가 발생할 경우 메모리가 누출되지 않고 개체가 사용 가능한 상태로 유지되어야 합니다(데이터가 수정될 수 있다 하더라도).

예외로부터 안전한 클래스

클래스는 그 자체가 부분적으로 생성되거나 부분적으로 제거되는 것을 방지함으로써 안전하지 않은 함수에 사용되는 경우에도 자체적으로 고유한 예외 안전성을 보장할 수 있습니다. 클래스 생성자가 완료 전에 종료되는 경우, 개체가 생성되지 않고 해당 소멸자가 호출되지 않습니다. 예외 전에 초기화된 자동 변수에는 해당 소멸자가 호출되겠지만, 스마트 포인터 또는 비슷한 자동 변수에 의해 관리되지 않는 동적으로 할당되는 메모리 또는 리소스는 누출됩니다.

기본 제공 형식은 모두 오류 없음을 보증하며, 표준 라이브러리 형식에서는 최소한 기본 보증이 지원됩니다. 예외로부터 안전해야 하는 모든 사용자 정의 형식에 대해서는 다음과 같은 지침을 따르십시오.

- 스마트 포인터 또는 다른 RAII 형식의 래퍼를 사용해서 모든 리소스를 관리합니다. 생성자가 예외를 throw하면 소멸자가 호출되지 않으므로 클래스 소멸자에서 리소스 관리 기능을 사용하지 않도록 합니다. 하지만 클래스가 리소스를 하나만 제어하는 전용 리소스 관리자인 경우에는 리소스 관리를 위해 소멸자를 사용할 수 있습니다.
- 기본 클래스 생성자에서 throw된 예외는 파생된 클래스 생성자에서 무시될 수 없습니다. 파생된 생성자에서 기본 클래스 예외를 변환하고 다시 throw하도록 하려면 함수 try 블록을 사용하십시오.
- 특히 클래스에 "실패할 수 있는 초기화"라는 개념이 있는 경우 스마트 포인터로 래핑된 데이터 멤버에 모든 클래스 상태를 저장할지 여부를 고려합니다. C++는 초기화되지 않은 데이터 멤버를 허용하지만 초기화되지 않았거나 부분적으로 초기화된 클래스 인스턴스를 지원하지 않습니다. 생성자는 성공 또는 실패해야 합니다. 생성자가 완료 시점까지 실행되지 않으면 개체가 생성되지 않습니다.
- 예외가 소멸자로부터 벗어나도록 허용하지 않습니다. C++의 기본 원리에서는 소멸자가 호출 스택으로 예외를 전파하도록 허용해서는 안됩니다. 소멸자가 잠재적 예외를 throw할 수 있는 작업을 수행해야 하는 경우에는 try catch 블록에서 작업을 수행하고 예외를 무시해야 합니다. 표준 라이브러리에는 여기에서 정의하는 모든 소멸자에 대해 이 수준의 보증을 제공합니다.

추가 정보

예외 및 오류 처리에 대한 최신 C++ 모범 사례

방법: 예외 코드와 예외 코드 간 인터페이스

방법: 예외 코드와 예외가 아닌 코드 간의 인터페이스

아티클 • 2023. 10. 12.

이 문서에서는 C++ 코드에서 일관된 예외 처리를 구현하는 방법과 예외 경계에서 오류 코드와 예외를 변환하는 방법을 설명합니다.

경우에 따라 C++ 코드는 예외(예외가 아닌 코드)를 사용하지 않는 코드와 인터페이스해야 합니다. 이러한 인터페이스를 예외 경계라고 합니다. 예를 들어 C++ 프로그램에서 `CreateFile` Wind32 함수를 호출할 수 있습니다. `CreateFile` 는 예외를 throw하지 않습니다. 대신 함수에서 검색 `GetLastError` 할 수 있는 오류 코드를 설정합니다. C++ 프로그램이 사소하지 않은 경우 일관된 예외 기반 오류 처리 정책을 사용하는 것이 좋습니다. 또한 예외가 아닌 코드와 인터페이스를 사용하므로 예외를 중단하지 않을 수 있습니다. 또한 C++ 코드에서 예외 기반 및 예외 기반이 아닌 오류 정책을 혼합하지 않습니다.

C++에서 예외가 아닌 함수 호출

C++에서 비예외 함수를 호출하는 경우 오류를 검색한 다음 예외를 throw할 수 있는 C++ 함수에서 해당 함수를 래핑합니다. 이러한 래퍼 함수를 디자인할 때 먼저 제공할 예외 보장 유형(noexcept, strong 또는 basic)을 결정합니다. 두 번째로 예외가 throw된 경우 모든 리소스(예: 파일 핸들)가 올바르게 해제되도록 함수를 디자인합니다. 일반적으로 스마트 포인터 또는 유사한 리소스 관리자를 사용하여 리소스를 소유한다는 의미입니다. 디자인 고려 사항에 대한 자세한 내용은 방법: 예외 안전을 위한 설계를 참조하세요.

예시

다음 예제에서는 Win32 `CreateFile` 및 `ReadFile` 함수를 사용하여 내부적으로 두 파일을 읽고 여는 C++ 함수를 보여 줍니다. `File` 클래스는 파일 핸들에 대한 RAII(Resource Acquisition Is Initialization) 래퍼입니다. 생성자는 "파일을 찾을 수 없음" 조건을 검색하고 C++ 실행 파일의 호출 스택 위로 오류를 전파하는 예외를 throw합니다(이 예제 `main()`에서는 함수). `File` 객체가 완전히 생성된 후 예외가 throw된 경우 소멸자는 파일 핸들을 해제하기 위해 자동으로 `CloseHandle`을 호출합니다. 원하는 경우 동일한 용도로 ATL(Active Template Library) `CHandle` 클래스를 사용하거나 `unique_ptr` 사용자 지정 삭제 함수와 함께 사용할 수 있습니다. Win32 및 CRT API를 호출하는 함수는 오류를 검색한 다음 로컬로 정의된 `ThrowLastErrorIf` 함수를 사용하여 C++ 예외를 throw합니다. 그러면 클래스에서 파생된 클래스가 `runtime_error` 사용됩니다 `Win32Exception`. 이 예제의 모든 함수는 강력한 예외 보장을 제공합니다. 이러한 함수의 어느 시점에서든 예외가 throw되면 리소스가 유출되지 않고 프로그램 상태가 수정되지 않습니다.

C++

```
// compile with: /EHsc
#include <Windows.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <string>
#include <limits>
#include <stdexcept>

using namespace std;

string FormatErrorMessage(DWORD error, const string& msg)
{
    static const int BUFFERLENGTH = 1024;
    vector<char> buf(BUFFERLENGTH);
    FormatMessageA(FORMAT_MESSAGE_FROM_SYSTEM, 0, error, 0, buf.data(),
        BUFFERLENGTH - 1, 0);
    return string(buf.data()) + " (" + msg + ")";
}

class Win32Exception : public runtime_error
{
private:
    DWORD m_error;
public:
    Win32Exception(DWORD error, const string& msg)
        : runtime_error(FormatErrorMessage(error, msg)), m_error(error) { }

    DWORD GetErrorCode() const { return m_error; }
};

void ThrowLastErrorIf(bool expression, const string& msg)
{
    if (expression) {
        throw Win32Exception(GetLastError(), msg);
    }
}

class File
{
private:
    HANDLE m_handle;

    // Declared but not defined, to avoid double closing.
    File& operator=(const File&);
    File(const File&);

public:
    explicit File(const string& filename)
    {
        m_handle = CreateFileA(filename.c_str(), GENERIC_READ,
FILE_SHARE_READ,
        nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, nullptr);
    }
}
```

```

        ThrowLastErrorIf(m_handle == INVALID_HANDLE_VALUE,
                          "CreateFile call failed on file named " + filename);
    }

~File() { CloseHandle(m_handle); }

HANDLE GetHandle() { return m_handle; }

};

size_t GetFileSizeSafe(const string& filename)
{
    File fobj(filename);
    LARGE_INTEGER filesize;

    BOOL result = GetFileSizeEx(fobj.GetHandle(), &filesize);
    ThrowLastErrorIf(result == FALSE, "GetFileSizeEx failed: " + filename);

    if (filesize.QuadPart < (numeric_limits<size_t>::max)()) {
        return filesize.QuadPart;
    } else {
        throw;
    }
}

vector<char> ReadFileVector(const string& filename)
{
    File fobj(filename);
    size_t filesize = GetFileSizeSafe(filename);
    DWORD bytesRead = 0;

    vector<char> readbuffer(filesize);

    BOOL result = ReadFile(fobj.GetHandle(), readbuffer.data(),
                           readbuffer.size(),
                           &bytesRead, nullptr);
    ThrowLastErrorIf(result == FALSE, "ReadFile failed: " + filename);

    cout << filename << " file size: " << filesize << ", bytesRead: "
        << bytesRead << endl;

    return readbuffer;
}

bool IsFileDiff(const string& filename1, const string& filename2)
{
    return ReadFileVector(filename1) != ReadFileVector(filename2);
}

#include <iomanip>

int main ( int argc, char* argv[] )
{
    string filename1("file1.txt");
    string filename2("file2.txt");
}

```

```

try
{
    if(argc > 2) {
        filename1 = argv[1];
        filename2 = argv[2];
    }

    cout << "Using file names " << filename1 << " and " << filename2 <<
endl;

    if (IsFileDiff(filename1, filename2)) {
        cout << "++ Files are different." << endl;
    } else {
        cout << "== Files match." << endl;
    }
}
catch(const Win32Exception& e)
{
    ios state(nullptr);
    state.copyfmt(cout);
    cout << e.what() << endl;
    cout << "Error code: 0x" << hex << uppercase << setw(8) <<
setfill('0')
    << e.GetErrorCode() << endl;
    cout.copyfmt(state); // restore previous formatting
}
}

```

예외가 아닌 코드에서 예외적 코드 호출

C 프로그램에서 호출할 수 있는 것으로 `extern "C"` 선언된 C++ 함수입니다. C++ COM 서버는 다양한 언어로 작성된 코드에서 사용할 수 있습니다. 비예외 코드로 호출되는 C++의 공용 예외 인식 함수를 구현하는 경우 C++ 함수에서는 호출자에 예외를 다시 전파하지 않아야 합니다. 이러한 호출자는 C++ 예외를 `catch`하거나 처리할 방법이 없습니다. 프로그램이 종료되거나, 리소스가 누출되거나, 정의되지 않은 동작이 발생할 수 있습니다.

C++ 함수는 특히 처리 방법을 알고 있는 모든 예외를 `catch`하고, 적절한 경우 예외를 호출자가 이해하는 오류 코드로 변환하는 것이 좋습니다 `extern "C"`. 일부 잠재적 예외를 알지 못하는 경우 C++ 함수에는 마지막 처리기로 `catch(...)` 블록이 있어야 합니다. 이러한 경우 프로그램이 알 수 없고 복구할 수 없는 상태일 수 있으므로 호출자에게 치명적인 오류를 보고하는 것이 가장 좋습니다.

다음 예제에서는 `throw`될 수 있는 예외가 파생된 `std::exception` 예외 형식 또는 예외 형식이라고 `Win32Exception` 가정하는 함수를 보여 줍니다. 함수는 이러한 형식의 예외를 `catch`하고 호출자에게 Win32 오류 코드로 오류 정보를 전파합니다.

C++

```
BOOL DiffFiles2(const string& file1, const string& file2)
{
    try
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return FALSE;
        }
        return TRUE;
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }

    catch(std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return FALSE;
}
```

예외에서 오류 코드로 변환하는 경우 잠재적인 문제가 있습니다. 오류 코드에는 예외가 저장할 수 있는 다양한 정보가 포함되지 않는 경우가 많습니다. 이 문제를 해결하려면 throw될 수 있는 각 특정 예외 유형에 대한 블록을 제공하고 `catch` 로깅을 수행하여 오류 코드로 변환되기 전에 예외의 세부 정보를 기록할 수 있습니다. 이 방법은 여러 함수가 모두 동일한 블록 집합 `catch` 을 사용하는 경우 반복 코드를 만들 수 있습니다. 코드 반복을 방지하는 좋은 방법은 해당 블록을 구현 `try` 하고 `catch` 블록에서 `try` 호출되는 함수 자체를 허용하는 하나의 프라이빗 유틸리티 함수로 리팩터링하는 것입니다. 각 공용 함수에서 람다 식으로 유틸리티 함수에 코드를 전달합니다.

C++

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
```

```
{  
    SetLastError(MY_APPLICATION_GENERAL_ERROR);  
}  
return false;  
}
```

다음 예제에서는 함수를 정의하는 람다 식을 작성하는 방법을 보여 줍니다. 람다 식은 명명된 함수 개체를 호출하는 코드보다 인라인으로 읽기가 더 쉬운 경우가 많습니다.

C++

```
bool DiffFiles3(const string& file1, const string& file2)  
{  
    return Win32ExceptionBoundary([&]()->bool  
    {  
        File f1(file1);  
        File f2(file2);  
        if (IsTextFileDiff(f1, f2))  
        {  
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);  
            return false;  
        }  
        return true;  
    });  
}
```

람다 식에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

예외 코드에서 예외가 아닌 코드를 통해 예외적 코드 호출

예외를 인식하지 않는 코드에서 예외를 throw하는 것은 가능하지만 권장되지는 않습니다. 예를 들어 C++ 프로그램에서 제공하는 콜백 함수를 사용하는 라이브러리를 호출할 수 있습니다. 경우에 따라 원래 호출자가 처리할 수 있는 예외가 아닌 코드를 통해 콜백 함수에서 예외를 throw할 수 있습니다. 그러나 예외가 성공적으로 작동할 수 있는 상황은 엄격합니다. 스택 해제 의미 체계를 유지하는 방식으로 라이브러리 코드를 컴파일해야 합니다. 예외를 인식하지 않는 코드는 C++ 예외를 트래핑할 수 있는 작업을 수행할 수 없습니다. 또한 호출자와 콜백 간의 라이브러리 코드는 로컬 리소스를 할당할 수 없습니다. 예를 들어 예외를 인식하지 않는 코드에는 할당된 힙 메모리를 가리키는 로컬이 있을 수 없습니다. 이러한 리소스는 스택이 해제될 때 유출됩니다.

예외를 인식하지 않는 코드에서 예외를 throw하려면 다음 요구 사항을 충족해야 합니다.

- 를 사용하여 /EHs 예외 인식이 아닌 코드에서 전체 코드 경로를 빌드할 수 있습니다.
- 스택이 해제될 때 누수될 수 있는 로컬로 할당된 리소스가 없습니다.

- 코드에는 `_except` 모든 예외를 catch하는 구조적 예외 처리기가 없습니다.

예외가 아닌 코드에서 예외를 throw하는 것은 오류가 발생하기 쉽고 디버깅 문제가 발생 할 수 있으므로 권장하지 않습니다.

참고 항목

[예외 및 오류 처리에 대한 최신 C++ 모범 사례](#)

[방법: 예외 보안을 위한 디자인](#)

Try, Throw 및 Catch 문(C++)

아티클 • 2023. 10. 12.

C++에서 예외 처리를 구현하려면, `throw` 및 `catch` 식을 사용합니다 `try`.

먼저 블록을 사용하여 예외를 `try` throw할 수 있는 하나 이상의 문을 묶습니다.

식은 `throw` 예외 조건(종종 오류)이 블록에서 발생했음을 알 수 있습니다 `try`. 식의 피연산자로 모든 형식의 개체를 `throw` 사용할 수 있습니다. 일반적으로 이 개체는 일반적으로 오류 정보를 전달하는 데 사용됩니다. 대부분의 경우 표준 라이브러리에 `std::exception` 정의된 클래스 또는 파생 클래스 중 하나를 사용하는 것이 좋습니다. 해당 중 하나가 적절 `std::exception` 하지 않은 경우 .

throw될 수 있는 예외를 처리하려면 블록 바로 다음에 `try` 하나 이상의 `catch` 블록을 구현합니다. 각 `catch` 블록은 처리할 수 있는 예외 유형을 지정합니다.

이 예제에서는 `try` 블록 및 해당 처리기를 보여 줍니다. `GetNetworkResource()` 가 네트워크 연결을 통해 데이터를 받고 두 개의 예외 형식은 `std::exception`에서 파생된 사용자 정의 클래스라고 가정합니다. 예외는 문에서 참조로 `const catch` catch됩니다. 값으로 예외를 throw하고 상수 참조로 catch하는 것이 좋습니다.

예시

C++

```
MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
```

```

{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}

```

설명

절 뒤의 `try` 코드는 코드의 보호된 섹션입니다. 식 `0/ throw throw 됩니다.` 즉, 예외가 발생합니다. 절 뒤의 `catch` 코드 블록은 예외 처리기입니다. 식의 형식이 `호환되는 경우 throw 되는 예외를 catch 하는 throw catch 처리기입니다.` 블록에서 형식 일치 `catch` 를 제어하는 규칙 목록은 Catch 블록 평가 [방법을 참조하세요.](#) `catch` 문이 형식 대신 줄임표(...)를 지정하는 경우 블록은 `catch` 모든 유형의 예외를 처리합니다. 옵션을 사용하여 `/EHa` 컴파일하는 경우 C 구조적 예외와 메모리 보호, 0으로 나누기 및 부동 소수점 위반과 같은 시스템 생성 또는 애플리케이션에서 생성된 비동기 예외를 포함할 수 있습니다. `catch` 블록은 일치하는 형식을 찾기 위해 프로그램 순서로 처리되므로 줄임표 처리기는 연결된 `try` 블록의 마지막 처리기여야 합니다. 주의해서 사용 `catch(...)` 하세요. `catch` 블록이 `catch`된 특정 예외를 처리하는 방법을 알지 않는 한 프로그램을 계속하도록 허용하지 마세요. 일반적으로 `catch(...)` 블록은 오류를 기록하고 프로그램 실행을 중지하기 전에 특별한 정리 작업을 수행하는 데 사용합니다.

`throw` 피연산자 없는 식은 현재 처리 중인 예외를 다시 `throw`합니다. 원래 예외의 다형 형식 정보가 유지되므로 예외를 다시 `throw`할 때 이 양식을 사용하는 것이 좋습니다. 이러한 식은 처리기 또는 처리기에서 호출되는 함수에서 `catch`만 사용해야 `catch` 합니다. 다시 `throw`된 예외 개체는 복사본이 아니라 원래 예외 개체입니다.

C++

```

try {
    throw CSomeOtherException();
}
catch(...) {
    // Catch all exceptions - dangerous!!!
    // Respond (perhaps only partially) to the exception, then
    // re-throw to pass the exception to some other handler
    // ...
    throw;
}

```

참고 항목

예외 및 오류 처리에 대한 최신 C++ 모범 사례

키워드

처리되지 않은 C++ 예외

`_uncaught_exception`

Catch 블록 평가 방법 (C++)

아티클 • 2023. 10. 12.

일반적으로 std::exception에서 파생되는 형식을 throw할 것을 권장하지만 C++를 사용하면 모든 형식의 예외를 throw할 수 있습니다. C++ 예외는 throw된 예외와 `catch` 동일한 형식을 지정하는 처리기 또는 모든 유형의 예외를 catch할 수 있는 처리기에서 catch할 수 있습니다.

throw된 예외의 형식이 하나의 기본 클래스 또는 여러 개의 클래스를 가진 클래스인 경우 예외 형식의 기본 클래스 및 예외 형식의 기본에 대한 참조를 허용하는 처리기로 catch할 수 있습니다. 예외가 참조에 의해 catch될 때 실제 throw된 예외 개체에 바인딩됩니다. 바인딩되지 않는 예외는 복사본입니다(함수에 대한 인수와 동일).

예외가 throw되면 다음과 같은 유형의 `catch` 처리기가 예외를 catch할 수 있습니다.

- 줄임표 구문을 사용하여 모든 형식을 받아들일 수 있는 처리기.
- 예외 개체와 동일한 형식을 허용하는 처리기입니다. 복사본이므로 `const volatile` 한정자는 무시됩니다.
- 예외 개체와 동일한 형식에 대한 참조를 허용하는 처리기.
- 예외 개체와 같은 형식의 형식 또는 `volatile` 형식에 대한 참조 `const` 를 허용하는 처리기입니다.
- 예외 개체와 동일한 형식의 기본 클래스를 허용하는 처리기입니다. 복사본 `const volatile` 이므로 한정자는 무시됩니다. `catch` 기본 클래스에 대한 처리기는 파생 클래스에 `catch` 대한 처리기 앞에 있으면 안됩니다.
- 예외 개체와 동일한 형식의 기본 클래스에 대한 참조를 허용하는 처리기.
- 예외 개체와 동일한 형식의 기본 클래스 또는 `volatile` 형식에 대한 참조 `const` 를 허용하는 처리기입니다.
- throw된 포인터 개체가 표준 포인터 변환 규칙을 통해 변환될 수 있는 대상 포인터를 허용하는 처리기.

지정된 `try` 블록에 `catch` 대한 처리기가 모양 순서대로 검사되기 때문에 처리기가 표시되는 순서는 중요합니다. 예를 들어 파생 클래스의 처리기 앞에 기본 클래스의 처리기를 배치하면 오류가 발생합니다. 일치하는 `catch` 처리기를 찾은 후에는 후속 처리기가 검사되지 않습니다. 따라서 줄임표 `catch` 처리기는 해당 `try` 블록의 마지막 처리기여야 합니다. 예시:

C++

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

이 예제에서 줄임표 `catch` 처리기는 검사되는 유일한 처리기입니다.

참고 항목

[예외 및 오류 처리에 대한 최신 C++ 모범 사례](#)

C++에서 예외 및 스택 해제

아티클 • 2023. 10. 12.

C++ 예외 메커니즘에서 컨트롤은 throw 문에서 throw된 형식을 처리할 수 있는 첫 번째 catch 문으로 이동합니다. catch 문에 도달하면 throw 문과 catch 문 사이의 범위에 있는 모든 자동 변수가 스택 해제라고 하는 프로세스에서 제거됩니다. 스택 해제에서 실행은 다음과 같이 진행됩니다.

1. 컨트롤은 `try` 일반 순차적 실행을 통해 문에 도달합니다. 블록의 `try` 보호된 섹션 이 실행됩니다.
2. 보호된 섹션 `catch` 을 실행하는 동안 예외가 throw되지 않으면 블록 뒤에 `try` 있는 절이 실행되지 않습니다. 연결된 `try` 블록 뒤에 있는 마지막 `catch` 절 뒤에 있는 문에서 실행이 계속됩니다.
3. 보호된 섹션을 실행하는 동안 또는 보호된 섹션이 직접 또는 간접적으로 호출하는 루틴에서 예외가 throw되면 피연산자가 만든 개체에서 예외 개체가 만들어 `throw` 집니다. (이는 복사 생성자가 관련될 수 있음을 의미합니다.) 이 시점에서 컴파일러는 throw되는 형식의 예외를 처리할 수 있는 더 높은 실행 컨텍스트에서 또는 모든 유형의 예외를 `catch` 처리할 수 있는 처리기를 찾 `catch` 습니다. `catch` 처리기는 블록 뒤 `try` 의 모양 순서로 검사됩니다. 적절한 처리기를 찾을 수 없으면 다음 동적으로 바깥쪽 `try` 블록이 검사됩니다. 이 프로세스는 가장 바깥쪽 바깥쪽 `try` 블록을 검사할 때까지 계속됩니다.
4. 일치하는 처리기를 여전히 찾을 수 없거나 해제 프로세스 중 처리기가 컨트롤을 갖기 전에 예외가 발생하는 경우 미리 정의된 런타임 함수 `terminate` 가 호출됩니다. 예외가 throw되었지만 해제 작업을 시작하기 전에 예외가 발생하는 경우 `terminate` 가 호출됩니다.
5. 일치하는 `catch` 처리기가 발견되고 값으로 catch되는 경우 예외 개체를 복사하여 공식 매개 변수가 초기화됩니다. 참조로 catch하면 예외 개체를 참조하도록 매개 변수가 초기화됩니다. 정식 매개 변수가 초기화된 후 스택 해제 프로세스가 시작됩니다. 여기에는 처리기와 연결된 블록의 시작 `try` 부분과 예외의 throw 사이트 사이에 완전히 생성되었지만 아직 소멸되지 않은 모든 자동 개체가 소멸됩니다 `catch`. 소멸은 생성과 반대 순서로 발생합니다. `catch` 처리기가 실행되고 프로그램은 마지막 처리기(즉, 처리기가 아닌 `catch` 첫 번째 문 또는 구문)에서 실행을 다시 시작합니다. 컨트롤은 throw된 예외를 `catch` 통해서만 처리기를 입력할 수 있으며 문의 문이나 레이블 `switch` 을 `case` 통해 `goto` 서는 안됩니다.

스택 해제 예제

다음 예제에서는 예외가 throw되면 어떻게 스택이 해제되는지 보여 줍니다. 스레드에서의 실행은 방식에 따라 각 함수를 해제하면서 `c`의 throw 문에서 `main`의 catch 문으로 점프합니다. `Dummy` 개체가 만들어진 다음 범위에서 벗어날 때 제거되는 순서를 살펴보십시오. 또한 catch 문이 포함된 `main`을 제외하고는 어떤 함수도 완료할 수 없습니다. 함수 `A`는 `B()` 호출에서 반환되지 않으며 `B`도 `C()` 호출에서 반환되지 않습니다. `Dummy` 포인터의 정의 및 해당 `delete` 문에 대한 주석 처리를 제거한 다음 프로그램을 실행하면 포인터가 삭제되지 않습니다. 이는 함수가 예외 보장을 제공하지 않는 경우 발생할 수 있음을 보여 줍니다. 자세한 내용은 방법: 예외에 대한 디자인을 참조 [하세요](#). catch 문을 주석으로 처리하는 경우 처리되지 않은 예외로 인해 프로그램을 종료할 때 나타나는 현상을 확인 할 수 있습니다.

C++

```
#include <string>
#include <iostream>
using namespace std;

class MyException{};
class Dummy
{
public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created
Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s << MyName << endl; }
    string MyName;
    int level;
};

void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
```

```

{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    // Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
    // delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}

/* Output:
   Entering main
   Created Dummy: M
   Copy created Dummy: M
   Entering FunctionA
   Copy created Dummy: A
   Entering FunctionB
   Copy created Dummy: B
   Entering FunctionC
   Destroyed Dummy: C
   Destroyed Dummy: B
   Destroyed Dummy: A
   Destroyed Dummy: M
   Caught an exception of type: class MyException
   Exiting main.

*/

```

예외 사양(throw, noexcept)(C++)

아티클 • 2024. 10. 17.

예외 사양은 함수에서 전파할 수 있는 예외 형식에 대한 프로그래머의 의도를 나타내는 C++ 언어 기능입니다. 예외 사양을 사용하여 함수가 예외에 의해 종료되거나 종료되지 않도록 지정할 수 있습니다. 컴파일러는 이 정보를 사용하여 함수에 대한 호출을 최적화하고, 예기치 않은 예외가 함수를 이스케이프하는 경우 프로그램을 종료할 수 있습니다.

C++17 이전에는 두 가지 종류의 예외 사양이 있었습니다. `noexcept` 사양은 C++11에서 새로 추가되었습니다. 함수를 이스케이프할 수 있는 잠재적 예외 집합이 비어 있는지 여부를 지정합니다. `noexcept(true)`의 별칭인 `throw()`를 제외하고 동적 예외 사양 또는 `throw(optional_type_list)` 사양은 C++11에서 더 이상 사용되지 않으며 C++17에서 제거되었습니다. 이 예외 사양은 함수에서 `throw`될 수 있는 예외에 대한 요약 정보를 제공하도록 설계되었지만 실제로는 문제가 있는 것으로 밝혀졌습니다. 다소 유용한 것으로 입증된 동적 예외 사양 중 하나는 무조건 `throw()` 적인 사양이었습니다. 예를 들어 함수 선언은 다음과 같습니다.

C++

```
void MyFunction(int i) throw();
```

컴파일러에 함수가 아무 예외도 `throw`하지 않음을 알립니다. 그러나 함수가 예외를 `throw`하는 경우 모드에서 `/std:c++14` 정의되지 않은 동작이 발생할 수 있습니다. 따라서 위의 연산자 `noexcept` 대신 연산자를 사용하는 것이 좋습니다.

C++

```
void MyFunction(int i) noexcept;
```

다음 표에서는 예외 사양의 Microsoft C++ 구현을 요약합니다.

 테이블 확장

예외 사양	의미
<code>noexcept</code>	이 함수는 예외를 <code>throw</code> 하지 않습니다. <code>/std:c++14</code> 모드(기본값) <code>noexcept</code> 와 동일합니다.
<code>noexcept(true)</code>	<code>noexcept(true)</code> 동일합니다. 선언되거나 호출되는 <code>noexcept</code>
<code>throw()</code>	<code>noexcept(true)</code> <code>std::terminate</code> 함수에서 예외가 <code>throw</code> 되는 경우 모드로 <code>throw()</code> <code>/std:c++14</code> 선언된 함수에서 예외가 <code>throw</code> 되면 결과는 정의되지 않은 동작입니다. 특정 함수가 호출되지 않습니다. 이는 컴파일러가 호출해야 하는 C++14 표준의 차이입니다 <code>std::unexpected</code> .
Visual Studio 2017 버전 15.5 이상: 모드에서는 <code>/std:c++17</code> <code>noexcept</code> 모두	

예외 사양	의미
	<code>noexcept(true) throw()</code> 동일합니다. 모드에서는 <code>/std:c++17 throw()</code> 에 대한 <code>noexcept(true)</code> 별칭입니다. 모드 이상에서는 <code>/std:c++17</code> 이러한 사양 <code>std::terminate</code> 중 하나로 선언된 함수에서 예외가 throw되면 C++17 표준에서 요구하는 대로 호출됩니다.
<code>noexcept(false)</code> <code>throw(...)</code> 사양 없음	함수는 모든 형식의 예외를 throw할 수 있습니다.
<code>throw(type)</code>	(C++14 이하) 함수는 형식 <code>type</code> 의 예외를 throw할 수 있습니다. 컴파일러는 구문을 수락하지만 이를 .로 <code>noexcept(false)</code> 해석합니다. 모드 이상에서 <code>/std:c++17</code> 컴파일러는 경고 C5040을 실행합니다.

애플리케이션에서 예외 처리를 사용하는 경우 호출 스택에는 throw된 예외를 처리하는 함수가 있어야만 표시된 `noexcept noexcept(true)` 함수의 외부 범위가 종료됩니다 `throw()`. 예외를 throw하는 함수와 예외를 처리하는 함수 간에 호출된 `noexcept noexcept(true)` 함수가 (또는 `throw()` 모드에서 `/std:c++17`) 지정된 경우 `noexcept` 함수가 예외를 전파할 때 프로그램이 종료됩니다.

함수의 예외 동작은 다음 요인에 따라 달라집니다.

- 설정된 언어 표준 컴파일 모드입니다.
- C 또는 C++에서 함수를 컴파일하는지 여부
- `/EH` 사용하는 컴파일러 옵션입니다.
- 예외 사양을 명시적으로 지정하는지 여부

C 함수에서는 명시적 예외 사양이 허용되지 않습니다. C 함수는 예외를 throw하지 않는 것으로 간주되며, `/EHsa` 또는 `/EHsc`.에서 `/EHsc` / `/EHs` 구조적 예외를 throw할 수 있습니다.

다음 표에서는 C++ 함수가 잠재적으로 다양한 컴파일러 예외 처리 옵션에서 throw될 수 있는지 여부를 요약합니다.

테이블 확장

함수	<code>/EHsc</code>	<code>/EHs</code>	<code>/EHsa</code>	<code>/EHsc</code>
예외 사양이 없는 C++ 함수	예	예	예	예
또는 <code>noexcept(true) throw()</code> 예외 사양이 있는 <code>noexcept</code> C++ 함수	아니요	아니요	예	예
또는 <code>throw(...)</code> <code>throw(type)</code> 예외 사양이 있는	예	예	예	예

noexcept(false) C++ 함수

예시

C++

```
// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
    printf_s("in handler\n");
}

void f1(void) throw(int) {
    printf_s("About to throw 1\n");
    if (1)
        throw 1;
}

void f5(void) throw() {
    try {
        f1();
    }
    catch(...) {
        handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//     f1();
// }

void __declspec(nothrow) f2(void) {
    try {
        f1();
    }
    catch(int) {
        handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
    f1();
}
```

```
int main() {
    f2();

    try {
        f4();
    }
    catch(...) {
        printf_s("Caught exception from f4\n");
    }
    f5();
}
```

Output

```
About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler
```

참고 항목

[try, throw 및 catch 문\(C++\)](#)
[최신 C++ 예외 및 오류 처리 모범 사례](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

noexcept (C++)

아티클 • 2024. 11. 21.

C++11: 함수가 예외를 발생시킬 수 있는지 여부를 지정합니다.

구문

```
:  
    noexcept  
    noexcept-expression  
    throw ( )  
  
:  
    noexcept ( constant-expression )
```

매개 변수

constant-expression

잠재적인 예외 형식 집합이 비어 있는지 여부를 나타내는 `bool` 형식의 상수 식입니다. 비 조건부 버전은 `noexcept(true)` 와 같습니다.

설명

`noexcept-expression`은 예외 사양의 일종으로, 함수를 종료하는 예외에 대해 예외 처리기가 일치시킬 수 있는 형식 집합을 나타내는 함수 선언에 대한 접미사입니다.

`constant_expression`이 `true`를 생성할 때의 단항 조건 연산자

`noexcept(constant_expression)`과 해당 무조건 동의어인 `noexcept`는 함수를 종료할 수 있는 잠재적 예외 형식 집합이 비어 있음을 지정합니다. 즉, 함수는 예외를 `throw`하지 않으며 예외를 범위 외부로 전파할 수 없습니다. `constant_expression`이 `false`를 생성할 때 연산자 `noexcept(constant_expression)`를 사용하거나 예외 사양이 없는 경우(소멸자 또는 할당 해제 함수 제외) 함수를 종료할 수 있는 잠재적 예외의 집합이 모든 유형의 집합임을 나타냅니다.

직접 또는 간접적으로 호출하는 모든 함수가 `noexcept` 또는 `const` 인 경우에만 함수를 `noexcept`로 표시합니다. 컴파일러는 `noexcept` 함수에 생성될 수 있는 예외에 대해 모든 코드 경로를 검사하지는 않습니다. 예외가 `noexcept`로 표시된 함수에 도달하면 `std::terminate`가 즉시 호출되고 범위 내 개체의 소멸자가 호출된다는 보장이 없습니다. 동적 예외 지정자 `throw()` 대신 `noexcept`를 사용합니다. `noexcept(true)`의 별칭인 `throw()`를 제외하고 동적 예외 사양 또는 `throw(optional_type_list)` 사양은 C++11에서 더 이상

사용되지 않으며 C++17에서 제거되었습니다. 예외가 호출 스택으로 전파되는 것을 허용하지 않도록 하려는 함수에 `noexcept`를 적용하는 것이 좋습니다. `noexcept`로 선언된 함수는 컴파일러가 다양한 컨텍스트에서 보다 효율적인 코드를 생성할 수 있게 합니다. 자세한 내용은 [예외 사양](#)을 참조하세요.

예시

해당 인수를 복사하는 함수 템플릿이 복사 중인 개체가 POD(일반 이전 데이터 형식)인 조건에서 `noexcept`로 선언될 수 있습니다. 이러한 함수는 다음과 같이 선언될 수 있습니다.

C++

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

참고 항목

[최신 C++ 예외 및 오류 처리 모범 사례](#)

[예외 사양\(throw, noexcept\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

처리되지 않은 C++ 예외

아티클 • 2023. 10. 12.

현재 예외에 대해 일치하는 처리기(또는 줄임표 `catch` 처리기)를 찾을 수 없는 경우 미리 정의된 `terminate` 런타임 함수가 호출됩니다. (처리기에서 명시적으로 호출 `terminate` 할 수도 있습니다.) 기본 동작 `terminate` 은 .를 호출 `abort`하는 것입니다. 애플리케이션을 종료하기 전에 `terminate`로 프로그램의 몇 가지 다른 함수를 호출하려면 단일 인수로 호출되는 함수 이름을 사용하여 `set_terminate` 함수를 호출합니다. `set_terminate`는 프로그램에서 언제든지 호출할 수 있습니다. 루틴은 `terminate` 항상 인수로 지정된 마지막 함수를 호출합니다 `set_terminate`.

예시

다음 예제에서는 예외를 `char * throw`하지만 형식 `char *`의 예외를 `catch`하도록 지정된 처리기를 포함하지 않습니다. `set_terminate` 호출은 `terminate`가 `term_func`를 호출하도록 명령합니다.

C++

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
    cout << "term_func was called by terminate." << endl;
    exit( -1 );
}
int main() {
    try
    {
        set_terminate( term_func );
        throw "Out of memory!" // No catch handler for this exception
    }
    catch( int )
    {
        cout << "Integer exception raised." << endl;
    }
    return 0;
}
```

출력

Output

```
term_func was called by terminate.
```

`term_func` 함수는 `exit`를 호출하여 프로그램이나 현재 스레드를 종료합니다. 이 호출이 호출자로 반환되면 `abort`가 호출됩니다.

참고 항목

[예외 및 오류 처리에 대한 최신 C++ 모범 사례](#)

C(구조적) 및 C++ 예외 혼합

아티클 • 2023. 10. 12.

이식 가능한 코드를 작성하려는 경우 C++ 프로그램에서 SEH(구조적 예외 처리)를 사용하지 않는 것이 좋습니다. 그러나 구조화된 예외와 C++ 소스 코드를 사용하여 `/EHs` 컴파일하고 혼합하려고 할 수 있으며 두 종류의 예외를 모두 처리하기 위한 몇 가지 기능이 필요할 수 있습니다. 구조적 예외 처리기에는 개체 또는 형식화된 예외의 개념이 없으므로 C++ 코드에서 `throw`된 예외를 처리할 수 없습니다. 그러나 C++ `catch` 처리기는 구조적 예외를 처리할 수 있습니다. C++ 예외 처리 구문(`try`, `throw`, `catch`)은 C 컴파일러에서 허용되지 않지만 구조적 예외 처리 구문(`_try`, `_except`, `_finally`)은 C++ 컴파일러에서 지원됩니다.

C++ 예외로 구조적 예외를 처리하는 방법에 대한 자세한 내용은 참조 [_set_se_translator](#) 하세요.

구조적 예외와 C++ 예외를 혼합하는 경우 다음과 같은 잠재적인 문제에 유의하세요.

- C++ 예외 및 구조적 예외는 동일한 함수 내에서 혼합할 수 없습니다.
- 종료 처리기(`_finally` 블록)는 예외가 `throw`된 후 해제하는 동안에도 항상 실행됩니다.
- C++ 예외 처리는 해제 의미 체계를 사용하도록 설정하는 컴파일러 옵션으로 `/EH` 컴파일된 모든 모듈에서 해제 의미 체계를 `catch`하고 유지할 수 있습니다.
- 소멸자 함수가 모든 개체에 대해 호출되지 않는 경우가 있을 수 있습니다. 예를 들어 초기화되지 않은 함수 포인터를 통해 함수 호출을 시도하는 동안 구조적 예외가 발생할 수 있습니다. 함수 매개 변수가 호출 전에 생성된 개체인 경우 스택 해제 중에 해당 개체의 소멸자가 호출되지 않습니다.

다음 단계

- [C++ 프로그램 사용 `setjmp` 또는 `longjmp` 사용](#)

C++ 프로그램의 사용 `setjmp` 및 `longjmp` 사용에 대한 자세한 내용을 참조하세요.

- [C++에서 구조적 예외 처리](#)

C++를 사용하여 구조적 예외를 처리할 수 있는 방법의 예제를 참조하세요.

참고 항목

예외 및 오류 처리에 대한 최신 C++ 모범 사례

setjmp 및 longjmp 사용

아티클 • 2023. 10. 12.

setjmp와 longjmp를 함께 사용하면 로컬 goto 이 아닌 메서드를 실행할 수 있습니다. 일반적으로 C 코드에서 표준 호출 또는 반환 규칙을 사용하지 않고 이전에 호출한 루틴에서 실행 제어를 오류 처리 또는 복구 코드 전달하는 데 사용됩니다.

⊗ 주의

setjmp longjmp C++ 컴파일러 간에 스택 프레임 개체의 올바른 소멸을 지원하지 않으며 지역 변수에 대한 최적화를 방지하여 성능이 저하될 수 있으므로 C++ 프로그램에서 사용하지 않는 것이 좋습니다. 대신 사용하고 try catch 구성하는 것이 좋습니다.

C++ 프로그램을 사용하기 setjmp longjmp로 결정한 경우 함수와 SEH(구조적 예외 처리) 또는 C++ 예외 처리 간의 올바른 상호 작용을 보장하기 위해 setjmp.h> 또는 <setjmpex.h>도 포함합니다<.

Microsoft 전용

/EH 옵션을 사용하여 C++ 코드를 컴파일하는 경우 스택 해제 중에 로컬 개체에 대한 소멸자가 호출됩니다. 그러나 /EHs 또는 /EHsc를 사용하여 컴파일하고 noexcept 호출 longjmp을 사용하는 함수 중 하나를 사용하는 경우 최적화 프로그램 상태에 따라 해당 함수에 대한 소멸자 해제가 발생하지 않을 수 있습니다.

이식 가능한 코드에서 호출이 실행될 때 longjmp 프레임 기반 개체의 올바른 소멸은 표준에 의해 명시적으로 보장되지 않으며 다른 컴파일러에서 지원되지 않을 수 있습니다. 경고 수준 4에서 경고 C4611을 발생시키는 setjmp 호출을 알려면 '_setjmp'과 C++ 개체 소멸 간의 상호 작용은 이식할 수 없습니다.

Microsoft 전용 종료

참고 항목

[C\(구조적\) 및 C++ 예외 혼합](#)

C++에서 구조적 예외 처리

아티클 • 2023. 10. 12.

C SEH(구조적 예외 처리)와 C++ 예외 처리의 주요 차이점은 C++ 예외 처리 모델이 형식을 처리하는 반면 C 구조적 예외 처리 모델은 한 형식의 예외를 처리한다는 점입니다. 특히, `unsigned int`. 즉, C 예외는 부호 없는 정수 값으로 식별되는 반면 C++ 예외는 데이터 형식으로 식별됩니다. C에서 구조적 예외가 발생하면 가능한 각 처리기는 C 예외 컨텍스트를 검사하고 예외를 수락할지, 다른 처리기에 전달할지 또는 무시할지를 결정하는 필터를 실행합니다. C++에서 예외가 발생하는 경우 예외는 어떤 형식이든 가능합니다.

두 번째 차이점은 예외가 일반적인 제어 흐름에 보조로 발생하기 때문에 C 구조적 예외 처리 모델을 비동기라고 한다는 것입니다. C++ 예외 처리 메커니즘은 완전히 동기적입니다. 즉, 예외가 `throw`된 경우에만 발생합니다.

/EHs 또는 /EHsc [컴파일러 옵션을 사용하는 경우 C++ 예외 처리기가 구조적 예외를 처리하지 않습니다.](#) 이러한 예외는 구조적 예외 처리기 또는 `_finally` 구조적 종료 처리기에서만 `_except` 처리됩니다. 자세한 내용은 [구조적 예외 처리\(C/C++\)](#)를 참조하세요.

/EHs 컴파일러 옵션에서 C++ 프로그램에서 C 예외가 발생하는 경우 연결된 필터를 사용하는 구조적 예외 처리기 또는 예외 컨텍스트에 동적으로 더 가까운 C++ `catch` 처리기를 통해 처리할 수 있습니다. 예를 들어 이 샘플 C++ 프로그램은 C++ `try` 컨텍스트 내에서 C 예외를 발생합니다.

예제 - C++ catch 블록에서 C 예외 catch

```
C++

// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHs
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
    try {
        SEHFunc();
    }
    catch( ... ) {
        cout << "Caught a C exception." << endl;
    }
}

void SEHFunc() {
    __try {
```

```

        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        cout << "In finally." << endl;
    }
}

```

Output

```

In finally.
Caught a C exception.

```

C 예외 래퍼 클래스

위와 같은 간단한 예제에서 C 예외는 줄임표(...) **catch** 처리기에서만 catch할 수 있습니다. 예외의 형식이나 특성에 대한 정보가 처리기로 전달되지 않습니다. 이 메서드는 작동 하지만 경우에 따라 각 C 예외가 특정 클래스와 연결되도록 두 예외 처리 모델 간에 변환을 정의할 수 있습니다. 하나를 변환하려면 C 예외 "래퍼" 클래스를 정의할 수 있습니다. 이 클래스는 특정 클래스 형식을 C 예외로 특성화하기 위해 사용하거나 파생할 수 있습니다. 이렇게 하면 각 C 예외는 단일 처리기에서 모든 예외가 아닌 특정 C++ **catch** 처리기에 의해 개별적으로 처리될 수 있습니다.

래퍼 클래스에는 예외 값을 결정하는 멤버 함수로 구성되며 C 예외 모델에서 제공하는 확장된 예외 컨텍스트 정보에 액세스하는 인터페이스가 있을 수 있습니다. 기본 생성자 및 인수를 허용하는 **unsigned int** 생성자(기본 C 예외 표현을 제공하기 위해) 및 비트 복사 생성자를 정의할 수도 있습니다. C 예외 래퍼 클래스의 가능한 구현은 다음과 같습니다.

C++

```

// exceptions_Exception_Handling_Differences2.cpp
// compile with: /c
class SE_Exception {
private:
    SE_Exception() {}
    SE_Exception( SE_Exception& ) {}
    unsigned int nSE;
public:
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() {
        return nSE;
    }
};

```

이 클래스를 사용하려면 C 예외가 throw 될 때마다 내부 예외 처리 메커니즘에 의해 호출되는 사용자 지정 C 예외 변환 함수를 설치합니다. 번역 함수 내에서 적절한 일치하는 C++ `catch` 처리기에서 `SE_Exception` catch할 수 있는 형식화된 예외(형식 `SE_Exception` 또는 파생된 클래스 형식)를 throw할 수 있습니다. 대신 변환 함수는 예외를 처리하지 않았음을 나타내는 반환할 수 있습니다. 변환 함수 자체가 C 예외 [를 발생시킬 경우 종료](#) 가 호출됩니다.

사용자 지정 번역 함수를 지정하려면 번역 함수의 이름을 사용하여 `_set_se_translator` [함수를 단일 인수로 호출](#)합니다. 작성하는 변환 함수는 블록이 있는 스택의 각 함수 호출에 대해 한 번 호출됩니다 `try`. 기본 변환 함수는 없습니다. `_set_se_translator` [호출](#)하여 지정하지 않으면 줄임표 `catch` 처리기에서만 C 예외를 catch할 수 있습니다.

예제 - 사용자 지정 번역 함수 사용

예를 들어 다음 코드에서는 사용자 지정 변환 함수를 설치한 후 `SE_Exception` 클래스로 래핑된 C 예외를 발생시킵니다.

C++

```
// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHs
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
```

```
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}
```

Output

```
In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094
```

참고 항목

[C\(구조적\) 및 C++ 예외 혼합](#)

Structured Exception Handling (C/C++)

아티클 • 2024. 07. 15.

SEH(구조화된 예외 처리)는 하드웨어 오류와 같은 특정 예외 코드 상황을 적절하게 처리하기 위한 C 및 C++에 대한 Microsoft 확장입니다. Windows 및 Microsoft C++는 SEH를 지원하지만 C++ 코드에서는 ISO 표준 C++ 예외 처리를 사용하는 것이 좋습니다. 코드를 더욱 이식 가능하고 유연하게 만듭니다. 그러나 기존 코드를 유지하거나 특정 종류의 프로그램에 대해서는 SEH를 사용해야 할 수도 있습니다.

Microsoft 전용:

문법

`try-except-statement` 은:

```
_try compound-statement _except ( filter-expression ) compound-statement
```

`try-finally-statement` 은:

```
_try compound-statement _finally compound-statement
```

설명

SEH를 사용하면 실행이 예기치 않게 종료되는 경우 메모리 블록 및 파일과 같은 리소스가 올바르게 해제되도록 보장할 수 있습니다. `goto` 문을 사용하지 않는 간결한 구조의 코드나 반복 코드의 정교한 테스트를 사용하여 메모리 부족 등의 특정 문제를 처리할 수도 있습니다.

이 문서에 언급된 `try-except` 및 `try-finally` 문은 C 및 C++ 언어에 대한 Microsoft 확장입니다. 둘 다 애플리케이션이 그렇지 않을 경우 실행을 종료하는 이벤트 후에 프로그램을 제어할 수 있도록 하여 SEH를 지원합니다. SEH는 C++ 소스 파일에서 작동하지만 C++용으로 특별히 설계된 것은 아닙니다. `/EHa` 또는 `/EHsc` 옵션을 사용하여 컴파일하는 C++ 프로그램에서 SEH를 사용하는 경우 로컬 개체에 대한 소멸자가 호출되지만 다른 실행 동작은 예상과 다를 수 있습니다. 설명을 보려면 이 문서의 뒷부분에 있는 예를 참조하세요. 대부분의 경우 SEH 대신 ISO 표준 C++ 예외 처리를 사용하는 것이 좋습니다. C++ 예외 처리를 사용하면 코드 포팅 가능성이 향상되며 모든 형식의 예외를 처리할 수 있습니다.

SEH를 사용하는 C 코드가 있는 경우 이를 C++ 예외 처리를 사용하는 C++ 코드와 혼합할 수 있습니다. 자세한 내용은 [C++에서 구조화된 예외 처리](#)를 참조하세요.

SEH 메커니즘에는 다음 두 가지가 있습니다.

- 예외 처리기 또는 `_except` 블록은 `filter-expression` 값을 기반으로 예외에 응답하거나 예외를 해제할 수 있습니다. 자세한 내용은 [try-except문](#)을 참조하세요.
- 종료 처리기 또는 `_finally` 블록은 예외로 인해 종료가 발생하는지 여부에 관계없이 항상 호출됩니다. 자세한 내용은 [try-finally문](#)을 참조하세요.

이러한 두 종류의 처리기는 별개이지만 스택 해제라는 프로세스를 통해 긴밀하게 연결됩니다. 구조화된 예외가 발생하면 Windows는 현재 활성화된 가장 최근에 설치된 예외 처리기를 찾습니다. 처리기는 다음 세 가지 작업 중 하나를 수행할 수 있습니다.

- 예외를 인식하지 못하고 다른 처리기에 제어를 전달합니다 (`EXCEPTION_CONTINUE_SEARCH`).
- 예외를 인식하지만 해제합니다(`EXCEPTION_CONTINUE_EXECUTION`).
- 예외를 인식하고 처리합니다(`EXCEPTION_EXECUTE_HANDLER`).

예외가 발생할 때 실행 중이던 함수에 예외를 인식하는 예외 처리기가 없을 수 있습니다. 스택에서 훨씬 더 높은 함수에 있을 수 있습니다. 현재 실행 중인 함수 및 스택 프레임의 다른 모든 함수가 종료됩니다. 이 프로세스 동안 스택은 해제됩니다. 즉, 종료된 함수의 로컬 비정적 변수가 스택에서 지워집니다.

스택을 해제할 때 운영 체제는 각 함수에 대해 작성된 종료 처리기를 모두 호출합니다. 종료 처리기를 사용하면 그렇지 않을 경우 비정상적인 종료 때문에 열려 있을 리소스를 정리합니다. 중요한 섹션을 입력한 경우 종료 처리기에서 끝낼 수 있습니다. 프로그램이 종료되면 임시 파일 닫기 및 제거와 같은 다른 관리 작업을 수행할 수 있습니다.

다음 단계

- 예외 처리기 작성
- 종료 처리기 작성
- C++에서 구조적 예외 처리

예시

앞에서 설명한 것처럼 C++ 프로그램에서 SEH를 사용하고 `/EHs` 또는 `/EHsc` 옵션을 사용하여 컴파일하면 로컬 개체에 대한 소멸자가 호출됩니다. 그러나 C++ 예외도 사용하는 경우 실행 중의 동작은 예상과 다를 수 있습니다. 이 예에서는 이러한 동작 차이를 보여줍니다.

```

#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\n");
    }

    return 0;
}

```

`/EHsc` 를 사용하여 이 코드를 컴파일했지만 로컬 테스트 제어 매크로 `CPPEX` 가 정의되지 않은 경우 `TestClass` 소멸자는 실행되지 않습니다. 출력은 다음과 같습니다.

Output

Triggering SEH exception

Executing SEH __except block

`/EHsc`를 사용하여 코드를 컴파일하고 C++ 예외가 `throw`되도록 `CPPEX`가 `/DCPPEX`를 사용하여 정의된 경우 `TestClass` 소멸자가 실행되고 출력은 다음과 같습니다.

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

`/EHa`를 사용하여 코드를 컴파일하는 경우 표준 C++ `throw` 식을 사용하여 예외가 `throw`되든, SEH를 사용하여 `throw`되든 `TestClass` 소멸자가 실행됩니다. 즉, `CPPEX`가 정의되었는지 여부를 나타냅니다. 출력은 다음과 같습니다.

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

자세한 내용은 [/EH\(예외 처리 모델\)](#)을 참조하세요.

END Microsoft 전용

참고 항목

[예외 처리](#)

[키워드](#)

[<exception>](#)

[오류 및 예외 처리](#)

[구조화된 예외 처리\(Windows\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

예외 처리기 작성

아티클 • 2023. 10. 12.

예외 처리기는 일반적으로 특정 오류에 응답하는 데 사용됩니다. 예외 처리 구문을 사용하여 처리 방법을 알고 있는 예외를 제외한 모든 예외를 필터링할 수 있습니다. 다른 예외는 특정 예외를 찾도록 작성된 다른 처리기(런타임 라이브러리 또는 운영 체제에 있을 수 있음)에 전달되어야 합니다.

예외 처리기는 try-except 문을 사용합니다.

추가 정보

- [try-except 문](#)
- [예외 필터 작성](#)
- [소프트웨어 예외 발생](#)
- [하드웨어 예외](#)
- [예외 처리기에 대한 제한 사항](#)

참고 항목

[Structured Exception Handling \(C/C++\)](#)

try-except 문

아티클 • 2024. 07. 15.

try-except 문은 C 및 C++ 언어의 구조화된 예외 처리를 지원하는 Microsoft 전용 확장입니다.

C++

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

문법

try-except-statement:

```
__try compound-statement __except ( expression ) compound-statement
```

설명

try-except 문은 C 및 C++ 언어에 대한 Microsoft 확장입니다. 대상 애플리케이션은 이를 통해 일반적으로 응용프로그램 실행을 종료하는 이벤트가 발생하는 시기를 제어할 수 있습니다. 이러한 이벤트를 구조적 예외 또는 짧게 예외라고 합니다. 이러한 예외를 처리하는 메커니즘을 SEH(구조적 예외 처리)라고 합니다.

관련 내용은 [try-finally 문](#)을 참조하세요.

예외는 하드웨어 기반 또는 소프트웨어 기반일 수도 있습니다. 구조적 예외 처리는 애플리케이션을 하드웨어나 소프트웨어 예외로부터 완전히 복구할 수 없는 경우에도 유용합니다. SEH를 사용하면 오류 정보를 표시하고 애플리케이션의 내부 상태를 트래핑하여 문제를 진단하는 데 도움이 됩니다. 특히 재현하기 어려운, 일시적인 문제에 유용합니다.

① 참고

구조적 예외 처리는 Win32에서 C 및 C++ 소스 파일에 대해 작동하지만 하지만 특별히 C++용으로 설계되지는 않았습니다. C++ 예외 처리를 사용하여 코드의 이식성이 향상되는지 확인할 수 있습니다. 또한 C++ 예외 처리는 모든 형식의 예외를 처리할

수 있다는 점에서 보다 유연합니다. C++ 프로그램의 경우 원시 C++ 예외 처리(try, catch 및 throw 문)를 사용하는 것이 좋습니다.

절 뒤에 오는 복합 _try 문은 본문 또는 보호된 섹션입니다. 이러한 _except 식은 필터 식이라고도 합니다. 이 값에 따라 예외 처리 방법이 결정됩니다. _except 절 뒤에 오는 복합 문은 예외 처리기입니다. 처리기는 본문 섹션을 실행하는 동안 예외가 발생하는 경우 수행할 작업을 지정합니다. 다음과 같이 실행됩니다.

1. 보호된 섹션이 실행됩니다.
2. 보호된 섹션을 실행하는 동안 예외가 발생하지 않는 경우 _except 절 다음의 문에서 실행이 계속됩니다.
3. 보호된 섹션이 실행 중일 때나 보호된 섹션에서 호출하는 루틴에서 예외가 발생하는 경우 _except 식이 평가됩니다. 세 가지 값을 사용할 수 있습니다.
 - EXCEPTION_CONTINUE_EXECUTION (-1) 예외가 해제됩니다. 예외가 발생한 지점에서 계속 실행합니다.
 - EXCEPTION_CONTINUE_SEARCH (0) 예외가 인식되지 않습니다. try-except 문을 포함하는 처리기를 먼저 검색한 후, 그 다음으로 우선 순위가 높은 처리기를 검색하는 순으로 처리기 스택을 계속 검색합니다.
 - EXCEPTION_EXECUTE_HANDLER (1) 예외가 인식됩니다. _except 복합 문을 실행하여 예외 처리기로 제어를 전달하고, _except 블록 이후 실행을 계속합니다.

_except 식은 C 식으로 평가됩니다. 해당 식은 단일 값, 조건 식 연산자 또는 쉼표 연산자로 제한됩니다. 더 광범위한 처리가 필요한 경우 식은 위에 나열된 세 값 중 하나를 반환하는 루틴을 호출할 수 있습니다.

각 애플리케이션에는 자체 예외 처리기를 사용할 수 있습니다.

이는 _try 문으로 이동할 수는 없지만, 해당 문 밖으로 이동하는 것은 가능합니다. try-except 문을 실행하는 도중에 프로세스가 강제로 종료되는 경우, 예외 처리기가 호출되지 않습니다.

이전 버전과의 호환성을 위해 _try, _except 및 _leave는 _try, _except, _leave의 동의어입니다. 단 컴파일러 옵션 /Za(언어 확장 사용 안 함)가 지정된 경우는 예외입니다.

_leave 키워드

_leave 키워드는 try-except 문의 보호된 섹션 내에서만 유효하며, 보호된 섹션의 끝으로 이동하도록 합니다. 실행은 예외 처리기 뒤에 나오는 첫 번째 문에서 계속 됩니다.

`goto` 문 또한 보호된 섹션에서 외부로 이동할 수 있으며 `try-finally` 문에서처럼 성능이 저하되지 않습니다. 이는 스택 해제가 발생하지 않기 때문입니다. 그러나 `goto` 문 대신 `_leave` 키워드를 사용하는 것이 좋습니다. 보호된 섹션이 크거나 복잡한 경우, 프로그래밍 실수를 할 가능성이 적기 때문입니다.

구조적 예외 처리 내장 함수

구조화된 예외 처리는 `try-except` 문과 함께 사용할 수 있는 2개의 내장 함수인 `GetExceptionCode`와 `GetExceptionInformation`을 제공합니다.

`GetExceptionCode` 는 예외의 코드(32비트 정수)를 반환합니다.

`GetExceptionInformation` 내장 함수는 예외에 대한 추가 정보가 포함되어 있는 `EXCEPTION_POINTERS` 구조체에 포인터를 반환합니다. 이 포인터를 통하여, 하드웨어 예외 발생 시의 컴퓨터 상태에 액세스할 수 있습니다. 구조는 다음과 같습니다.

C++

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

포인터 형식 `PEXCEPTION_RECORD` 및 `PCONTEXT`의 경우, 포함 파일 `<winnt.h>`에 정의되어 있으며 `_EXCEPTION_RECORD` 및 `_CONTEXT`의 경우에는 포함 파일 `<excpt.h>`에 정의되어 있습니다.

`GetExceptionCode`의 경우 예외 처리기 내에서 사용할 수 있습니다. 하지만 `GetExceptionInformation`은 예외 필터 식 내에서만 사용할 수 있습니다. 해당 내장 함수가 가리키는 정보는 일반적으로 스택에 있으며, 예외 처리기로 제어가 전달되면 더 이상 사용할 수 없습니다.

`AbnormalTermination` 내장 함수는 종료 처리기 내에서 사용할 수 있습니다. `try-finally` 문의 본문이 순차적으로 종료되면 0을 반환합니다. 다른 모든 경우에는 1이 반환됩니다.

`<excpt.h>`는 다음과 같은 내장 함수에 대해 일부 대체 이름을 정의합니다.

`GetExceptionCode` 는 `_exception_code` 와 같습니다.

`GetExceptionInformation` 는 `_exception_info` 와 같습니다.

`AbnormalTermination` 는 `_abnormal_termination` 와 같습니다.

예시

C++

```
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    }
}

int main()
{
    int* p = 0x00000000;    // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13;      // causes an access violation exception;
        }
        __finally
        {
            puts("in finally. termination: ");
            puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
        }
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        puts("in except");
    }
    puts("world");
}
```

출력

Output

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world
```

참고 항목

[예외 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

예외 필터 작성

아티클 • 2024. 10. 16.

예외 처리기의 수준으로 이동하거나 계속 실행하여 예외를 처리할 수 있습니다. 예외 처리기 코드를 사용하여 예외를 처리하는 대신 필터 식을 사용하여 문제를 정리할 수 있습니다. 그런 다음(-1)을 반환하여 스택을 `EXCEPTION_CONTINUE_EXECUTION` 지우지 않고 정상적인 흐름을 다시 시작할 수 있습니다.

① 참고

일부 예외는 계속할 수 없습니다. 이러한 예외에 대해 필터가 -1로 평가되면 시스템에서 새 예외가 발생합니다. 호출 `RaiseException`할 때 예외가 계속될지 여부를 결정합니다.

예를 들어 다음 코드는 필터 식에서 함수 호출을 사용합니다. 이 함수는 문제를 처리한 다음 -1을 반환하여 정상적인 제어 흐름을 다시 시작합니다.

C++

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int Eval_Exception(int);
int main() {
    __try {
        ;
    }
    __except (Eval_Exception(GetExceptionCode())) {
        ;
    }
}
void HandleOverflow() {
    // Gracefully recover
}
int Eval_Exception(int n_except) {
    if (
        n_except != STATUS_INTEGER_OVERFLOW &&
        n_except != STATUS_FLOAT_OVERFLOW
    ) {
        // Pass on most exceptions
        return EXCEPTION_CONTINUE_SEARCH;
    }
    // Execute some code to clean up problem
    HandleOverflow();
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

필터가 복잡한 작업을 수행해야 할 때마다 필터 식에서 함수 호출을 사용하는 것이 좋습니다. 식을 계산하면 함수가 실행됩니다. 이 경우에는 `Eval_Exception`입니다.

예외를 확인하는 데 사용합니다 `GetExceptionCode`. 이 함수는 문의 필터 식 `_except` 내에서 호출해야 합니다. `Eval_Exception`는 호출 `GetExceptionCode`할 수 없지만 예외 코드가 전달되어야 합니다.

이 처리기는 예외가 정수 또는 부동 소수점 오버플로가 아닌 경우 제어를 다른 처리기에 전달합니다. 이 경우 처리기는 함수(`HandleOverflow` API 함수가 아닌 예제일 뿐)를 호출하여 예외에서 적절하게 복구하려고 시도합니다. 이 예제에서 비어 있는 문 블록은 `_except` (1)을 `Eval_Exception` 반환 `EXCEPTION_EXECUTE_HANDLER` 하지 않으므로 실행할 수 없습니다.

함수 호출 사용은 복잡한 필터 식을 처리하는 좋은 일반 용도의 기술입니다. 유용한 두 개의 다른 C 언어 기능은 다음과 같습니다.

- 조건 연산자
- 쉼표 연산자

조건부 연산자는 여기에서 자주 유용합니다. 특정 반환 코드를 확인한 다음 두 가지 값 중 하나를 반환하는 데 사용할 수 있습니다. 예를 들어 다음 코드의 필터는 예외가 다음과 같은 경우에만 예외 `STATUS_INTEGER_OVERFLOW`를 인식합니다.

C++

```
_except (GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0)
```

다음 코드는 동일한 결과를 생성하므로 이 경우 조건 연산자의 목적은 선명도를 제공하는 것입니다.

C++

```
_except (GetExceptionCode() == STATUS_INTEGER_OVERFLOW)
```

조건부 연산자는 필터를 -1 `EXCEPTION_CONTINUE_EXECUTION`로 평가할 수 있는 경우에 더 유용합니다.

쉼표 연산자를 사용하면 여러 식을 순서대로 실행할 수 있습니다. 그런 다음 마지막 식의 값을 반환합니다. 예를 들어 다음 코드는 예외 코드를 변수에 저장한 다음 테스트합니다.

C++

```
__except (nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW)
```

참고 항목

예외 처리기 작성

Structured Exception Handling (C/C++)

피드백

이 페이지가 도움이 되었나요?

Yes

No

제품 사용자 의견 제공 | Microsoft Q&A에서 도움말 보기

소프트웨어 예외 발생

아티클 • 2023. 10. 12.

프로그램 오류의 가장 일반적인 원인 중 몇 가지는 시스템에서 예외로 플래그가 지정되지 않습니다. 예를 들어 메모리 블록을 할당하려고 시도하지만 메모리가 부족한 경우, 런타임 또는 API 함수에서 예외를 발생시키지 않지만 오류 코드를 반환합니다.

그러나 코드에서 해당 조건을 검색한 다음 `RaiseException` 함수를 호출하여 보고하여 모든 조건을 예외로 처리할 수 있습니다. 이런 식으로 오류에 플래그를 지정함으로써 구조적 예외 처리의 장점을 모든 종류의 런타임 오류에 적용할 수 있습니다.

오류에 대해 구조적 예외 처리를 사용하려면

- 이벤트에 대해 고유한 예외 코드를 정의합니다.
- 문제를 감지하면 호출 `RaiseException` 합니다.
- 정의한 예외 코드를 테스트하려면 예외 처리 필터를 사용합니다.

`winerror.h` 파일은 <예외 코드의 형식을 보여 줍니다. 기존 예외 코드와 충돌하는 코드를 정의하지 않으려면 세 번째 최상위 비트를 1로 설정합니다. 4개의 최상위 비트는 다음 표와 같이 설정되어야 합니다.

Bits	권장하는 이진 설정	설명
31- 30	11	이러한 두 비트는 코드의 기본 상태(11 = 오류, 00 = 성공, 01 = 정보, 10 = 경고)를 나타냅니다.
29	1	클라이언트 비트. 사용자 정의 코드의 경우 1로 설정합니다.
28	0	예약된 비트. 0으로 설정된 상태로 둡니다.

"오류" 설정이 대부분의 예외에 적합하긴 하지만, 원하는 경우 처음 두 비트를 11 이외의 이진 값으로 설정할 수 있습니다. 명심해야 할 중요한 사항은 위의 표와 같이 29 및 28 비트를 설정하는 것입니다.

따라서 결과 오류 코드는 가장 높은 4비트가 16진수 E로 설정되어야 합니다. 예를 들어 다음 정의는 Windows 예외 코드와 충돌하지 않는 예외 코드를 정의합니다. 그러나 타사 DLL에서 사용하는 코드를 확인해야 할 수 있습니다.

C++

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001
```

```
#define STATUS_FILE_BAD_FORMAT 0xE0000002
```

예외 코드를 정의한 후 예외 코드를 사용하여 예외를 발생시킬 수 있습니다. 예를 들어 다음 코드는 메모리 할당 문제에 대한 응답으로 예외를 발생합니다

```
STATUS_INSUFFICIENT_MEM .
```

C++

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

예외를 발생시키기만 하려면 마지막 세 매개 변수를 0으로 설정하면 됩니다. 마지막 세 매개 변수는 추가 정보를 전달하고 처리기가 계속 실행되지 않도록 하는 플래그를 설정하는 데 유용합니다. 자세한 내용은 Windows SDK의 [RaiseException](#) 함수를 참조하세요.

이렇게 하면 예외 처리 필터에서 사용자가 정의한 코드를 테스트할 수 있습니다. 다음은 그 예입니다.

C++

```
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
          GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

참고 항목

[예외 처리기 작성](#)

[구조적 예외 처리\(C/C++\)](#)

하드웨어 예외

아티클 • 2023. 04. 03.

운영 체제에서 인식하는 대부분의 표준 예외는 하드웨어 정의 예외입니다. Windows는 몇몇 하위 수준의 소프트웨어 예외를 인식하지만 이러한 예외는 보통 운영 체제에서 잘 처리됩니다.

Windows는 다른 프로세서의 하드웨어 오류를 이 섹션의 예외 코드로 매핑합니다. 경우에 따라 프로세서는 이들 예외의 하위 집합만 생성할 수 있습니다. Windows는 예외에 대한 정보를 전처리하고 적절한 예외 코드를 발급합니다.

Windows에서 인식하는 하드웨어 예외는 다음 표에 요약되어 있습니다.

예외 코드	예외 원인
STATUS_ACCESS_VIOLATION	액세스할 수 없는 메모리 위치를 읽거나 씁니다.
STATUS_BREAKPOINT	하드웨어 정의 중단점이 발생합니다. 디버거에서만 사용됩니다.
STATUS_DATATYPE_MISALIGNMENT	올바르게 정렬되지 않은 주소에 있는 데이터를 읽거나 씁니다. 예를 들어, 16비트 엔티티는 2바이트 경계에 정렬되어야 합니다. (Intel 80x86 프로세서에는 적용되지 않습니다.)
STATUS_FLOAT_DIVIDE_BY_ZERO	부동 소수점 형식을 0.0으로 나눕니다.
STATUS_FLOAT_OVERFLOW	부동 소수점 형식의 최대 양의 지수를 초과합니다.
STATUS_FLOAT_UNDERFLOW	부동 소수점 형식의 최저 음의 지수 크기를 초과합니다.
STATUS_FLOATING_RESEVERED_OPERAND	예약된 부동 소수점 형식(잘못된 형식 사용)을 사용합니다.
STATUS_ILLEGAL_INSTRUCTION	프로세서에서 정의하지 않은 명령 코드를 실행합니다.
STATUS_PRIVILEGED_INSTRUCTION	현재 시스템 모드에서 실행될 수 없는 명령을 실행합니다.
STATUS_INTEGER_DIVIDE_BY_ZERO	정수 형식을 0으로 나눕니다.
STATUS_INTEGER_OVERFLOW	정수 범위를 초과하는 연산을 시도합니다.
STATUS_SINGLE_STEP	단일 단계 모드에서 하나의 명령을 실행합니다. 디버거에서만 사용됩니다.

앞의 표에 나온 예외 목록의 대부분은 디버거, 운영 체제 또는 다른 하위 수준 코드에서 처리됩니다. 정수 및 부동 소수점 오류를 제외하고 사용자 코드로 이러한 오류를 처리해서는 안 됩니다. 따라서 일반적으로 예외 처리 필터를 사용하여 예외를 무시하도록 하십시오(0으로 계산). 그러지 않으면 하위 수준 메커니즘이 적절하게 응답하지 않을 수 있습니다. 그러나 [종료 처리기를 작성](#)하여 이러한 하위 수준 오류의 잠재적 영향에 대해 적절한 예방 조치를 취할 수 있습니다.

추가 정보

[예외 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

예외 처리기에 대한 제한

아티클 • 2023. 10. 12.

코드에서 예외 처리기를 사용하는 주요 제한 사항은 문을 사용하여 `goto` 문 블록으로 `_try` 이동할 수 없다는 것입니다. 대신, 정상적인 제어 흐름을 통해 문 블록에 들어가야 합니다. 문 블록에서 `_try` 벗어날 수 있으며, 선택할 때 예외 처리기를 중첩할 수 있습니다.

참고 항목

[예외 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

종료 처리기 작성

아티클 • 2023. 10. 12.

예외 처리기와 달리 종료 처리기는 보호된 코드 블록이 정상적으로 종료되었는지 여부와 관계없이 항상 실행됩니다. 종료 처리기의 유일한 목적은 코드 섹션의 실행 완료 방법과 관계없이 메모리, 핸들 및 파일과 같은 리소스가 제대로 닫혔는지 확인하는 것이어야 합니다.

종료 처리기는 try-finally 문을 사용합니다.

추가 정보

- [try-finally 문](#)
- [리소스 정리](#)
- [예외 처리의 작업 타이밍](#)
- [종료 처리기에 대한 제한 사항](#)

참고 항목

[Structured Exception Handling \(C/C++\)](#)

try-finally 문

아티클 • 2024. 07. 08.

`try-finally` 문은 C 및 C++ 언어의 구조화된 예외 처리를 지원하는 Microsoft 전용 확장입니다.

구문

다음 구문에서는 `try-finally` 문을 설명합니다.

C++

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

문법

`try-finally-statement:`

```
__try compound-statement __finally compound-statement
```

`try-finally` 문은 C 및 C++ 언어의 Microsoft 확장으로서, 코드 블록 실행이 중단된 경우 대상 애플리케이션이 정리 코드를 실행하게 해줍니다. 정리는 메모리 할당 해제, 파일 닫기 및 파일 핸들 해제와 같은 작업으로 구성됩니다. `try-finally` 문은 루틴으로부터 중간에 반환되게 만들 수 있는 오류가 있는지 확인하기 위해 검사가 수행되는 위치가 많은 루틴에 특히 유용합니다.

관련 정보 및 코드 샘플은 [try-except 문](#)을 참조하세요. 구조화된 예외 처리에 대한 자세한 내용은 [구조화된 예외 처리](#)를 참조하세요. C++/CLI를 사용하여 관리되는 애플리케이션의 예외 처리에 대한 자세한 내용은 [/clr의 예외 처리](#)를 참조하세요.

① 참고

구조적 예외 처리는 Win32에서 C 및 C++ 소스 파일에 대해 작동하지만 특별히 C++용으로 설계되지는 않았습니다. C++ 예외 처리를 사용하여 코드의 이식성이 향

상되는지 확인할 수 있습니다. 또한 C++ 예외 처리는 모든 형식의 예외를 처리할 수 있다는 점에서 보다 유연합니다. C++ 프로그램의 경우 C++ 예외 처리 메커니즘(try, catch 및 throw 문)을 사용하는 것이 좋습니다.

try 절 뒤의 복합 문은 보호된 섹션입니다. finally 절 뒤의 복합 문은 종료 처리기입니다. 처리기는 보호된 섹션이 종료될 때 예외(비정상 종료)로 보호된 섹션을 종료하는지 또는 표준 풀스루(정상 종료)에 의해 종료하는지 여부에 따라 실행되는 작업 집합을 지정합니다.

제어는 단순한 순차적 실행(제어 이동)에 의해 try 문에 도달합니다. 제어가 try에 들어가면 연결된 처리기가 활성화됩니다. 제어 흐름이 try 블록 끝에 도달하면 다음과 같이 실행됩니다.

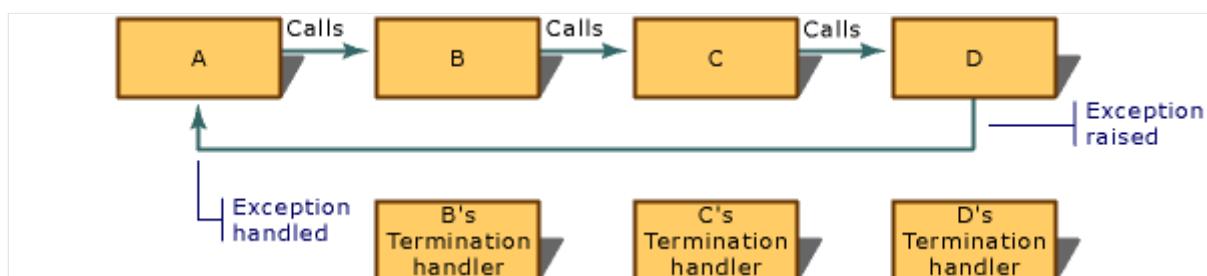
1. 종료 처리기가 호출됩니다.

2. 종료 처리기가 완료되면 실행이 finally 문 후에 계속됩니다. 그러나 보호된 섹션이 종료되면(예: 보호된 본문 외부의 goto 또는 return 문을 통해) 제어 흐름이 보호된 섹션 밖으로 이동하기 전에 종료 처리기가 실행됩니다.

finally 문은 적절한 예외 처리기 검색을 차단하지 않습니다.

try 블록에 예외가 발생할 경우 운영 체제에서 적합한 예외 처리기를 찾아야 합니다. 그렇지 않으면 프로그램이 실패합니다. 처리기를 찾으면 모든 finally 블록이 실행되고 처리기에서 실행이 다시 시작됩니다.

예를 들어, 일련의 함수 호출 링크에서는 함수 A를 D에 연결한다고 가정합니다(아래 그림 참조). 각 함수에는 종료 처리기가 하나씩 있습니다. 예외가 D 함수에서 발생하고 A에서 처리될 경우 시스템이 스택 D, C, B를 해제하면 그 순서대로 종료 처리기가 호출됩니다.



종료 처리기 실행 순서

① 참고

try-finally의 동작은 C#과 같이 finally 사용을 지원하는 일부 다른 언어와 다릅니다. 단일 try에는 finally와 except 중 하나만 있을 수 있습니다. 모두 함께

사용되는 경우 외부 try-except 문은 내부 try-finally 문을 포함해야 합니다. 또한 각 블록을 실행할 때 지정되는 규칙은 서로 다릅니다.

이전 버전과의 호환성을 위해 `_try`, `_finally` 및 `_leave`은 `_try`, `_finally` 및 `_leave`의 동의어입니다. 단 컴파일러 옵션 `/Za(언어 확장 사용 안 함)`가 지정된 경우는 예외입니다.

`_leave` 키워드

`_leave` 키워드는 `try-finally` 문의 보호된 섹션 내에서만 유효하며, 보호된 섹션의 끝으로 이동하도록 합니다. 종료 처리기의 첫 번째 문에서 계속 실행됩니다.

`goto` 문은 보호된 섹션에서 외부로 이동할 수도 있지만 스택 해제를 호출하기 때문에 성능이 저하됩니다. `_leave` 문은 스택 해제를 발생시키지 않기 때문에 더 효율적입니다.

비정상적인 종료

`longjmp` 런타임 함수를 사용하여 `try-finally` 문을 종료하는 것은 비정상 종료로 간주됩니다. `_try` 문 안으로 이동할 수 없지만 이 문 밖으로 이동할 수는 있습니다. 출발 지점 (`_try` 블록의 정상 종료)과 도착 지점(예외를 처리하는 `_except` 블록) 간에 활성화된 모든 `_finally` 문을 실행되어야 합니다. 이를 로컬 해제라고 합니다.

블록 밖으로의 점프를 포함하여 어떤 이유로든 `_try` 블록이 조기 종료되는 경우 시스템은 스택 해제 프로세스의 일부로 연결된 `_finally` 블록을 실행합니다. 이러한 경우 `_finally` 블록 내에서 호출되면 `AbnormalTermination` 함수는 `true`를 반환합니다. 그렇지 않으면 `false`를 반환합니다.

`try-finally` 문을 실행하는 중에 프로세스가 종료되면 종료 처리기가 호출되지 않습니다.

END Microsoft 전용

참고 항목

[종료 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

[키워드](#)

[종료 처리기 구문](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

리소스 정리

아티클 • 2024. 11. 21.

종료 처리기를 실행하는 동안 종료 처리기가 호출되기 전에 획득한 리소스를 알 수 없습니다. 모든 리소스를 `__try` 획득하기 전에 문 블록이 중단되어 모든 리소스가 열리지 않을 수 있습니다.

안전하려면 종료 처리 정리를 진행하기 전에 열려 있는 리소스를 확인해야 합니다. 권장 절차는 다음과 같습니다.

1. 핸들을 NULL로 초기화합니다.
2. 문 블록에서 `__try` 리소스를 가져옵니다. 핸들은 리소스를 획득할 때 양수 값으로 설정됩니다.
3. `__finally` 문 블록에서 해당 핸들 또는 플래그 변수가 0이 아니거나 NULL이 아닌 각 리소스를 해제합니다.

예시

예를 들어 다음 코드는 종료 처리기를 사용하여 세 개의 파일을 닫고 메모리 블록을 해제합니다. 이러한 리소스는 문 블록에서 `__try` 획득되었습니다. 리소스를 정리하기 전에 코드는 먼저 리소스를 획득했는지 확인합니다.

C++

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
    FILE *fp1 = NULL,
        *fp2 = NULL,
        *fp3 = NULL;
    LPVOID lpvoid = NULL;
    errno_t err;

    __try {
        lpvoid = malloc( BUFSIZ );

        err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
        err = fopen_s(&fp2, "NAMES.DAT", "w+" );
        err = fopen_s(&fp3, "CARS.DAT", "w+" );
    }
}
```

```
__finally {
    if ( fp1 )
        fclose( fp1 );
    if ( fp2 )
        fclose( fp2 );
    if ( fp3 )
        fclose( fp3 );
    if ( lpvoid )
        free( lpvoid );
}
}

int main() {
    fileOps();
}
```

참고 항목

[종료 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

예외 처리 타이밍: 요약

아티클 • 2023. 10. 12.

종료 처리기는 문 블록이 종료되는 방식에 `_try` 관계없이 실행됩니다. 원인으로는 블록 밖으로 `_try` 점프, 블록 밖으로 제어를 `longjmp` 전송하는 문, 예외 처리로 인해 스택 해제 등이 있습니다.

① 참고

Microsoft C++ 컴파일러는 두 가지 형식의 `setjmp` 문과 `longjmp` 문을 지원합니다. 빠른 버전은 종료 처리를 건너뛰지만 더 효율적입니다. 이 버전을 사용하려면 `setjmp.h` 파일을 <포함합니다. 다른 버전은 이전 단락에 설명된 대로 종료 처리를 지원합니다. 이 버전을 사용하려면 `setjmpex.h` 파일을 <포함합니다. 빠른 버전의 성능 향상 정도는 하드웨어 구성에 따라 달라집니다.

운영 체제는 다른 코드를 실행하기 전에 예외 처리기의 본문을 포함한 모든 종료 처리기를 적절한 순서대로 실행합니다.

예외로 인해 중단된 경우 시스템은 먼저 예외 처리기 하나 이상의 필터 부분을 실행한 다음 무엇을 종료할지 결정해야 합니다. 이벤트의 순서는 다음과 같습니다.

1. 예외가 발생합니다.
2. 시스템은 활성 예외 처리기의 계층 구조를 살펴보고 우선 순위가 가장 높은 처리기의 필터를 실행합니다. 가장 최근에 설치되고 가장 깊이 중첩된 예외 처리기이며 블록 및 함수 호출로 진행됩니다.
3. 이 필터가 컨트롤을 통과하는 경우(반환 0) 컨트롤을 통과하지 않는 필터가 발견될 때까지 프로세스가 계속됩니다.
4. 이 필터가 -1을 반환하면 예외가 발생한 위치에서 실행이 계속되고 종료가 발생하지 않습니다.
5. 필터가 1을 반환하는 경우 다음 이벤트가 발생합니다.
 - 시스템은 스택을 해제합니다. 예외가 발생한 위치와 예외 처리기가 포함된 스택 프레임 사이의 모든 스택 프레임을 지웁니다.
 - 스택이 해제되면서 스택의 각 종료 처리기가 실행됩니다.
 - 예외 처리기 자체가 실행됩니다.
 - 코드 줄에서 이 예외 처리기 끝 다음으로 제어가 전달됩니다.

참고 항목

종료 처리기 작성

Structured Exception Handling (C/C++)

종료 처리기에 대한 제한

아티클 • 2023. 10. 12.

문을 사용하여 `goto` 문 블록 또는 `__finally` 문 블록으로 `__try` 이동할 수 없습니다. 대신, 정상적인 제어 흐름을 통해 문 블록에 들어가야 합니다. (그러나 문 블록에서 `__try` 벗어날 수 있습니다.) 또한 예외 처리기 또는 종료 처리기를 블록 내에 중첩할 `__finally` 수 없습니다.

종료 처리기에서 허용되는 일부 종류의 코드는 의심스러운 결과를 생성하므로 주의해서 사용해야 합니다. 하나는 `goto` 문 블록에서 벗어나는 문입니다 `__finally`. 블록이 정상적인 종료의 일부로 실행되면 특별한 일이 발생하지 않습니다. 그러나 시스템이 스택을 해제하는 경우 해제가 중지됩니다. 그런 다음, 현재 함수는 비정상적인 종료가 없는 것처럼 제어를 얻습니다.

`return` 문 블록 내의 `__finally` 문은 거의 동일한 상황을 나타냅니다. 종료 처리기가 포함된 함수의 즉시 호출자에게 컨트롤이 반환됩니다. 시스템이 스택을 해제하는 경우 이 프로세스가 중지됩니다. 그런 다음 예외가 발생하지 않은 것처럼 프로그램이 진행됩니다.

참고 항목

[종료 처리기 작성](#)

[Structured Exception Handling \(C/C++\)](#)

스레드 간 예외 전송

아티클 • 2024. 07. 15.

The Microsoft C++ 컴파일러(MSVC)는 한 스레드에서 다른 스레드로 예외 전송을 지원합니다. 예외 전송으로 하나의 스레드에서 예외를 잡아내어 다른 스레드에서 예외가 throw되어 나타나도록 합니다. 예를 들어, 이 기능을 사용하여 기본 스레드가 보조 스레드에서 throw되는 모든 예외를 처리하는 위치에 있는 다중 스레드 애플리케이션을 작성할 수 있습니다. 예외 전송 병렬 프로그래밍 라이브러리를 만들거나 시스템 개발자에 주로 유용합니다. 전송 예외를 구현하기 위해, MSVC는 `exception_ptr` 형식 및 `current_exception`, `rethrow_exception` 및 `make_exception_ptr` 함수를 제공합니다.

구문

C++

```
namespace std
{
    typedef unspecified exception_ptr;
    exception_ptr current_exception();
    void rethrow_exception(exception_ptr p);
    template<class E>
        exception_ptr make_exception_ptr(E e) noexcept;
}
```

매개 변수

`unspecified`

`exception_ptr` 형식을 구현하는 데 사용되는 지정되지 않은 내부 클래스입니다.

`p`

예외를 참조하는 `exception_ptr` 개체입니다.

`E`

예외를 나타내는 클래스입니다.

`e`

매개 변수 `E` 클래스의 인스턴스입니다.

반환 값

`current_exception` 함수는 현재 진행 중인 예외를 참조하는 `exception_ptr` 개체를 반환합니다. 어떤 예외도 진행 중이 아닐 경우, 함수는 어떤 예외와도 관련되지 않은 `exception_ptr` 개체를 반환합니다.

`make_exception_ptr` 함수는 `e` 매개 변수에서 지정한, 예외를 참조하는 `exception_ptr` 개체를 반환합니다.

설명

시나리오

확장을 통해 변하는 작업량을 처리할 수 있는 애플리케이션을 만들려는 경우를 가정합니다. 이 목적을 달성하기 위해 그 작업을 수행하는 데 필요한 만큼의 2차 스레드를 초기의 1차 스레드가 생성하는 다중 스레드 애플리케이션을 설계해야 합니다. 보조 스레드는 주 스레드가 리소스를 관리하고 부하 균형을 조정하며 처리량을 향상시키도록 도와 줍니다. 다중 스레드 애플리케이션은 작업을 분산하여 단일 스레드 애플리케이션보다 성능을 높입니다.

하지만 보조 스레드에서 예외를 `throw`하는 경우 기본 스레드가 이를 처리하도록 합니다. 이것은 애플리케이션에서 2차 스레드의 수에 관계없이 일관성 있으며 통합된 방식으로 예외를 다루기를 원하기 때문입니다.

솔루션

앞의 시나리오를 처리하기 위해 스레드 간에 예외를 전송할 때 C++ 표준이 지원합니다. 보조 스레드가 예외를 `throw`하면 해당 예외는 *현재 예외*가 됩니다. 현실 세계에 비유하자면 현재 예외는 *비행* 중인 상태입니다. 현재 예외는 `throw`되는 시점부터 이 예외를 `catch`하는 예외 처리기가 반환될 때까지 *비행* 중인 것입니다.

보조 스레드는 `catch` 블록 내의 현재 예외를 `catch`할 수 있고, `exception_ptr` 개체에 예외를 저장하기 위하여 `current_exception` 함수를 호출합니다. `exception_ptr` 개체는 보조 스레드 및 기본 스레드에서 사용할 수 있어야 합니다. 예를 들어, `exception_ptr` 개체는 전역 변수일 수 있습니다. 전역 변수의 액세스는 뮤텍스로 제어됩니다. 예외 전송이라는 용어는 다른 스레드가 액세스 가능한 양식으로 변환될 수 있는 스레드의 예외를 의미합니다.

다음으로, 기본 스레드는 `rethrow_exception` 개체에서 예외를 추출하고 `throw`하는 `exception_ptr` 함수를 호출합니다. 예외가 `throw`되면 해당 주 스레드에 있는 현재 예외가 됩니다. 즉, 예외는 기본 스레드에서 발생되는 것처럼 보입니다.

마지막으로 주 스레드는 `catch` 블록에서 현재 예외를 catch한 후 처리하거나 더 높은 수준의 예외 처리기로 throw할 수 있습니다. 또는 주 스레드가 예외를 무시하고 프로세스를 끝낼 수 있습니다.

대부분의 애플리케이션은 스레드 간에 예외를 전송할 필요가 없습니다. 그러나 이 기능은 시스템에서 시스템 보조 스레드, 프로세서 또는 코어 간 작업을 분할할 수 있기 때문에 병렬 컴퓨팅 시스템에서 유용합니다. 병렬 컴퓨팅 환경에서는 단일, 전용 스레드가 보조 스레드의 모든 예외를 처리할 수 있으며 모든 애플리케이션에 대해 일관적인 예외 처리 모델을 제공할 수 있습니다.

C++ 표준 위원회 제안서에 대한 자세한 내용은 문서 번호 N2179, "스레드 간 예외 전송에 대한 언어 지원"이라는 제목의 인터넷을 검색하십시오.

예외 처리 모델 및 컴파일러 옵션

애플리케이션의 예외 처리 모델은 예외를 `catch` 및 전송할 수 있는지 여부를 결정합니다. Visual C++는 C++ 예외를 처리하기 위한 세 가지 모델인 [ISO 표준 C++ 예외 처리](#), [SEH\(구조적 예외 처리\)](#) 및 [CLR\(공용 언어 런타임\) 예외](#)를 지원합니다. `/EH` 및 `/clr` 컴파일러 옵션을 사용하여 응용 프로그램의 예외 처리 모델을 지정합니다.

다음 컴파일러 옵션 및 프로그래밍 문의 다음 조합만 예외를 전송할 수 있습니다. 다른 조합은 예외는 `catch`할 수 없거나, `catch`할 수는 있지만 예외를 전송할 수 없습니다.

- `/EHs` 컴파일러 옵션 및 `catch` 문은 SEH 및 C++ 예외를 전송할 수 있습니다.
- `/EHsc`, `/EHs` 및 `/EHsc` 컴파일러 옵션 및 `catch` 문은 C++ 예외를 전송할 수 있습니다.
- `/clr` 컴파일러 옵션 및 `catch` 문은 C++ 예외를 전송할 수 있습니다. `/clr` 컴파일러 옵션은 `/EHs` 옵션의 사양을 의미합니다. 컴파일러는 관리되는 예외의 전송을 지원하지 않습니다. 이것은 [System.Exception 클래스](#)로부터 유래한 관리되는 예외가 공용 언어 런타임의 시설을 이용함으로써 스레드 사이에서 이동할 수 있는 개체이기 때문입니다.

① 중요

`/EHsc` 컴파일러 옵션을 지정하고 C++ 예외만 `catch`하는 것이 좋습니다. `/EHs` 또는 `/clr` 컴파일러 옵션과 줄임표 예외 선언과 함께 `catch` 문을 사용한다면 보안 위협에 노출됩니다(`catch(...)`). 보통은 `catch` 문을 사용하여 몇 가지 특정 예외를 캡처하려고 할 것입니다. 하지만 `catch(...)` 문은 예기치 않은 심각한 예외를 포함하여 모든 C++ 및 SEH 예외를 캡처합니다. 예기치 않은 예외를

무시하거나 잘못 처리하는 경우 악성 코드가 이 기회를 이용하여 프로그램 보안을 해칠 수 있습니다.

사용

다음 섹션에서는 `exception_ptr` 형식과 `current_exception`, `rethrow_exception`, 및 `make_exception_ptr` 함수를 이용하여 예외 사항을 전송하는 방법을 설명합니다.

`exception_ptr` 형식

`exception_ptr` 개체를 사용하여 현재 예외 또는 사용자 지정 예외의 인스턴스를 참조합니다. Microsoft 구현에서 예외는 `EXCEPTION_RECORD` 구조체에 의해 표시됩니다. 각 `exception_ptr` 개체에는 예외를 나타내는 `EXCEPTION_RECORD` 구조체의 복사본을 가리키는 예외 참조 필드가 포함됩니다.

`exception_ptr` 변수를 선언할 때 이 변수는 예외와 연관되지 않습니다. 즉, 해당 예외 참조 필드는 NULL입니다. 이러한 `exception_ptr` 개체를 *null exception_ptr*라고 합니다.

`current_exception` 또는 `make_exception_ptr` 함수를 사용하여 `exception_ptr` 개체에 예외를 지정합니다. `exception_ptr` 변수에 예외를 할당하면, 변수 예외 참조 필드는 예외 복사본을 가리킵니다. 메모리가 부족하여 예외를 복사할 수 없는 경우 예외 참조 필드는 `std::bad_alloc` 예외의 복사본을 가리킵니다. `current_exception` 또는 `make_exception_ptr` 함수가 다른 이유로 예외를 복사할 수 없는 경우, `terminate` 함수를 호출하여 현재 프로세스를 종료합니다.

그 이름에도 불구하고 `exception_ptr` 개체 자체는 포인터가 아닙니다. 이 개체는 포인터 의미 체계를 준수하지 않으며 포인터 멤버 액세스(`->`) 또는 간접 참조(`*`) 연산자와 함께 사용될 수 없습니다. `exception_ptr` 개체에는 공용 데이터 멤버 또는 멤버 함수가 없습니다.

비교

등호(==) 및 부등호(!=) 연산자를 사용하여 두 개의 `exception_ptr` 개체를 비교할 수 있습니다. 연산자는 예외를 나타내는 `EXCEPTION_RECORD` 구조의 이진 값(비트 패턴)을 비교하지 않습니다. 대신, 연산자는 `exception_ptr` 개체의 예외 참조 필드 주소를 비교합니다. 따라서 `null exception_ptr`과 `NULL` 같은 동일한 것으로 비교됩니다.

`current_exception` 함수

`catch` 블록에서 `current_exception` 함수를 호출합니다. 예외가 비행 중이고 `catch` 블록이 해당 예외를 catch할 수 있을 경우, `current_exception` 함수는 해당 예외를 참조하는 `exception_ptr` 개체를 반환합니다. 그렇지 않으면 인수가 null `exception_ptr` 개체를 반환합니다.

세부 정보

`current_exception` 함수는 `catch` 문이 `exception-declaration` 문을 지정하는지 여부와 관계없이 발생 중인 예외를 캡처합니다.

예외를 다시 `throw`하지 않는 경우 `catch` 블록의 끝에서 현재 예외의 소멸자가 호출되지 않습니다. 그러나 소멸자에 있는 `current_exception` 함수를 호출하는 경우, 함수는 현재 예외를 참조하는 `exception_ptr` 개체를 반환합니다.

`current_exception` 수식에 대한 연속 호출은 현재 예외 건의 다른 복사본을 뜻하는 `exception_ptr` 개체를 반환합니다. 따라서 개체는 복사본에 같은 이진 값이 있더라도 다른 복사본을 참조하기에 같지 않은 것으로 비교합니다.

SEH 예외

`/EHa` 컴파일러 옵션을 사용하는 경우 C++ `catch` 블록에서 SEH 예외를 catch할 수 있습니다. `current_exception` 함수는 SEH 예외를 참조하는 `exception_ptr` 개체를 반환합니다. 또한 전송된 `exception_ptr` 개체를 인수로 사용하여 호출하는 경우 `rethrow_exception` 함수는 SEH 예외를 `throw`합니다.

`current_exception` 함수는 SEH `__finally` 종료 처리기, `__except` 예외 처리기 또는 `__except` 필터 식에서 호출하는 경우 null `exception_ptr`를 반환합니다.

전송된 예외는 중첩된 예외를 지원하지 않습니다. 예외가 처리되는 동안 다른 예외가 `throw`되는 경우 중첩된 예외가 발생합니다. 중첩된 예외를 catch하는 경우 `EXCEPTION_RECORD.ExceptionRecord` 데이터 멤버는 연결된 예외를 설명하는 `EXCEPTION_RECORD` 구조체의 체인을 가리킵니다. `current_exception` 함수는 `ExceptionRecord` 데이터 멤버가 0이 되는 `exception_ptr` 개체를 반환하기 때문에 중첩된 예외를 지원하지 않습니다.

SEH 예외를 catch하는 경우 `EXCEPTION_RECORD.ExceptionInformation` 데이터 멤버 배열의 임의의 포인터가 참조하는 메모리를 관리해야 합니다. 해당 `exception_ptr` 개체의 수명 중 메모리가 유효하며 `exception_ptr` 개체가 삭제될 때 메모리가 해제되는지 확인해야 합니다.

구조적 예외(SE) 변환기 함수를 전송 예외 기능과 함께 사용할 수 있습니다. SEH 예외가 C++ 예외로 번역되는 경우 `current_exception` 함수는 원래 SEH 예외가 아니라 번역된 예외를 참조하는 `exception_ptr`을 반환합니다. `rethrow_exception` 함수는 이후에 원래 예외가 아닌 번역된 예외를 `throw`합니다. SE translator 함수에 대한 자세한 내용은 [_set_se_translator](#)를 참조하세요.

rethrow_exception 함수

`exception_ptr` 개체에서 `catch`된 예외를 저장한 후 기본 스레드에서 개체를 처리할 수 있습니다. 기본 스레드에서 `rethrow_exception` 개체와 함께 작동하는 `exception_ptr` 함수를 호출합니다. `rethrow_exception` 함수는 `exception_ptr` 개체에서 예외를 추출하고, 주 스레드의 컨텍스트에서 예외를 `throw`합니다. `rethrow_exception` 함수의 `p` 매개 변수가 null `exception_ptr` 인 경우, 해당 함수는 `std::bad_exception`을 `throw`합니다.

이제 추출된 예외는 기본 스레드에서 현재 예외이며 이를 다른 예외와 동일한 방식으로 처리할 수 있습니다. 예외를 `catch`하는 경우 즉시 처리할 수도 있고 `throw` 문을 사용하여 더 높은 수준의 예외 처리기로 보낼 수도 있습니다. 그렇지 않으면 아무 작업도 수행하지 말고 기본 시스템 예외 처리기로 해당 프로세스를 종료합니다.

make_exception_ptr 함수

`make_exception_ptr` 함수는 클래스의 인스턴스를 인수로 사용한 다음 인스턴스를 참조하는 `exception_ptr`을 반환합니다. 모든 클래스 개체를 인수로 사용할 수 있지만, 일반적으로 [예외 클래스](#) 개체를 `make_exception_ptr` 함수의 인수로 지정합니다.

`make_exception_ptr` 함수 호출은 C++ 예외를 `throw`하고 `catch` 블록에서 `catch`한 다음 `current_exception` 함수를 호출하여 예외를 참조하는 `exception_ptr` 개체를 반환하는 것과 동일합니다. `make_exception_ptr` 함수의 Microsoft 구현은 예외를 `throw`한 후 `catch`하는 것보다 더 효율적입니다.

애플리케이션에는 일반적으로 `make_exception_ptr` 함수가 필요하지 않으며, 이 함수를 사용하지 않는 것이 좋습니다.

예시

다음 예제에서는 표준 C++ 예외와 사용자 지정 C++ 예외를 한 스레드에서 다른 스레드로 전송합니다.

C++

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;

// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int           aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );

// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};

int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;

    // Create secondary threads.
    for( int i=0; i < THREADCOUNT; i++ )
    {
        aArg[i] = i;
        aThread[i] = CreateThread(
            NULL,          // Default security attributes.
            0,             // Default stack size.
            (LPTHREAD_START_ROUTINE) ThrowExceptions,
            (LPVOID) &aArg[i], // Thread function argument.
            0,             // Default creation flags.
            &ThreadID); // Receives thread identifier.
        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return -1;
        }
    }

    // Wait for all threads to terminate.
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
    // Close thread handles.
    for( int i=0; i < THREADCOUNT; i++ ) {
        CloseHandle(aThread[i]);
    }

    // Rethrow and catch the transported exceptions.
    for ( int i = 0; i < THREADCOUNT; i++ ) {
        try {

```

```

        if (aException[i] == NULL) {
            printf("exception_ptr %d: No exception was transported.\n",
i);
        }
        else {
            rethrow_exception( aException[i] );
        }
    }
    catch( const invalid_argument & ) {
        printf("exception_ptr %d: Caught an invalid_argument
exception.\n", i);
    }
    catch( const myException & ) {
        printf("exception_ptr %d: Caught a  myException exception.\n",
i);
    }
}
}

// Each thread throws an exception depending on its thread
// function argument, and then ends.

DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.
            // In practice, your application performs an operation that
            // implicitly throws an exception.
            throw invalid_argument("A C++ exception.");
        }
        catch ( const invalid_argument & ) {
            aException[x] = current_exception();
        }
    }
    else {
        // User-defined exception.
        aException[x] = make_exception_ptr( myException() );
    }
    return TRUE;
}

```

Output

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a  myException exception.

```

요구 사항

머리글: <exception>

참고 항목

예외 처리/EH (예외 처리 모델)
/clr (공용 언어 런타임 컴파일)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

어설션 및 사용자 제공 메시지 (C++)

아티클 • 2024. 11. 21.

C++ 언어는 애플리케이션을 디버그하는 데 도움이 되는 세 가지 오류 처리 메커니즘인 `#error` 지시문, `static_assert` 키워드 및 어설션 매크로, `_assert_wassert` 매크로를 지원합니다. 세 가지 메커니즘 모두 오류 메시지를 제공하고 두 메커니즘은 소프트웨어 어설션도 테스트합니다. 소프트웨어 어설션은 프로그램의 특정 지점에서 `true`가 될 조건을 지정합니다. 컴파일 타임 어설션이 실패하는 경우 컴파일러에서 진단 메시지와 컴파일 오류를 생성합니다. 런타임 어설션이 실패하는 경우에는 운영 체제에서 진단 메시지를 제공하고 애플리케이션을 닫습니다.

설명

애플리케이션의 수명은 전처리, 컴파일 및 런타임 단계로 구성됩니다. 각 오류 처리 메커니즘은 이러한 단계 중 하나가 수행되는 동안 사용할 수 있는 디버그 정보에 액세스합니다. 효과적으로 디버깅하려면 해당 단계에 대한 적절한 정보를 제공하는 메커니즘을 선택합니다.

- `#error` 지시문은 전처리 시간에 적용됩니다. 이 지시문은 사용자 지정 메시지를 무조건 내보내고 컴파일이 오류와 함께 실패하도록 합니다. 메시지에는 전처리기 지시문으로 조작되는 텍스트가 포함될 수 있지만 모든 결과 식은 계산되지 않습니다.
- `static_assert` 선언은 컴파일 시간에 적용됩니다. 이 선언은 부울로 변환할 수 있는 사용자 지정 정수 계열 식으로 표현된 소프트웨어 어설션을 테스트합니다. 식이 0(false)으로 계산되는 경우 컴파일러는 사용자 지정 메시지를 발행하고 컴파일이 오류와 함께 실패합니다.

`static_assert` 선언은 디버깅 템플릿에 특히 유용합니다. 이는 템플릿 인수가 사용자 지정 식에 포함될 수 있기 때문입니다.

- 어설션 매크로, `_assert_wassert` 매크로는 런타임에 적용됩니다. 이 매크로는 사용자 지정 식을 계산하며, 결과가 0인 경우 진단 메시지가 제공되고 애플리케이션이 닫힙니다. 다른 많은 매크로(예: `_ASSERT` 및 `_ASERTE`)는 이 매크로와 유사하지만 시스템 정의 또는 사용자 정의 진단 메시지를 실행합니다.

참고 항목

`#error` 지시문(C/C++)

assert Macro, `_assert`, `_wassert`

`_ASSERT`, `_ASERTE`, `_ASSERT_EXPR` 매크로

static_assert

_STATIC_ASSERT 매크로

템플릿

피드백

이 페이지가 도움이 되었나요?

Yes

No

제품 사용자 의견 제공 | Microsoft Q&A에서 도움말 보기

static_assert

아티클 • 2023. 10. 12.

컴파일 시 소프트웨어 어설션을 테스트합니다. 지정된 상수 식이 `false` 있으면 컴파일러가 지정된 메시지를 표시하고, 제공된 경우 컴파일이 오류 C2338로 실패하고, 그렇지 않으면 선언이 적용되지 않습니다.

구문

```
static_assert( constant-expression, string-literal );  
  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and  
later)
```

매개 변수

constant-expression

부울로 변환할 수 있는 정수 계열 상수 식입니다. 계산된 식이 0(false) 이면 문자열 리터럴 매개 변수가 표시되고 컴파일에 오류가 발생합니다. 식이 0이 아닌 경우(true) `static_assert` 선언은 적용되지 않습니다.

string-literal

상수 식 `매개 변수가 0`이면 표시되는 메시지입니다. 메시지는 컴파일러의 기본 문자 집합에 있는 [문자 문자열입니다](#). 즉, 멀티바이트 또는 와이드 문자가 아닙니다.

설명

선언의 `static_assert` 상수 식 `매개 변수`는 소프트웨어 어설션을 나타냅니다. 소프트웨어 어설션은 프로그램의 특정 지점에서 true가 될 조건을 지정합니다. 조건이 true이면 선언에 `static_assert` 영향을 주지 않습니다. 조건이 false이면 어설션이 실패하고, 컴파일러가 [메시지를 문자열 리터럴](#) 매개 변수로 표시하고, 오류와 함께 컴파일이 실패합니다. Visual Studio 2017 이상에서는 문자열 리터럴 매개 변수가 선택 사항입니다.

이 선언은 `static_assert` 컴파일 시간에 소프트웨어 어설션을 테스트합니다. 반면, [어설션 매크로 및 _assert 및 _wassert 함수는](#) 런타임에 소프트웨어 어설션을 테스트하고 공간 또는 시간에 런타임 비용이 발생합니다. 템플릿 인수를 `static_assert` 상수 식 `매개 변수`에 포함할 수 있으므로 이 선언은 템플릿을 디버깅하는 데 특히 유용합니다.

컴파일러는 선언이 `static_assert` 발견될 때 구문 오류에 대한 선언을 검사합니다. 컴파일러는 템플릿 매개 변수에 의존하지 않는 경우 상수 식 매개 변수를 즉시 평가합니다. 그렇지 않으면 템플릿이 인스턴스화될 때 컴파일러가 상수 식 매개 변수를 평가합니다. 그 결과, 컴파일러는 선언이 발생할 때 진단 메시지를 한 번 내보내고 템플릿이 인스턴스화될 때 다시 한 번 메시지를 내보낼 수 있습니다.

네임스페이스 `static_assert` 스, 클래스 또는 블록 범위에서 키워드(keyword) 사용할 수 있습니다. (다음 `static_assert` 항목 키워드(keyword) 네임스페이스 범위에서 사용할 수 있으므로 프로그램에 새 이름을 도입하지 않더라도 기술적으로 선언입니다.)

네임스페이스 범위에 대한 `static_assert` 설명

다음 예제에서는 선언에 `static_assert` 네임스페이스 범위가 있습니다. 컴파일러가 `void`* 형식의 크기를 알고 있기 때문에 식이 즉시 계산됩니다.

예: `static_assert` 네임스페이스 범위 사용

C++

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

클래스 범위에 대한 `static_assert` 설명

다음 예제에서 선언에는 `static_assert` 클래스 범위가 있습니다. `static_assert` 템플릿 매개 변수가 POD(일반 이전 데이터) 형식인지 확인합니다. 컴파일러는 선언될 때 선언을 `static_assert` 검사하지만 클래스 템플릿이 인스턴스화 `main()` 될 때까지 상수 식 매개 변수를 `basic_string` 평가하지 않습니다.

예: `static_assert` 클래스 범위 사용

C++

```
#include <type_traits>
#include <iostream>
namespace std {
template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
    static_assert(std::is_pod<CharT>::value,
                 "Template argument CharT must be a POD type in class template basic_string");
```

```

    // ...
};

}

struct NonPOD {
    NonPOD(const NonPOD &) {}
    virtual ~NonPOD() {}
};

int main()
{
    std::basic_string<char> bs;
}

```

블록 범위에 대한 `static_assert` 설명

다음 예제에서는 선언에 `static_assert` 블록 범위가 있습니다. VMPage `static_assert` 구조체의 크기가 시스템의 가상 메모리 페이지와 같은지 확인합니다.

예: `static_assert` 블록 범위

C++

```

#include <sys/param.h> // defines PAGESIZE
class VMMClient {
public:
    struct VMPAGE { // ...
    };
    int check_pagesize() {
        static_assert(sizeof(VMPAGE) == PAGESIZE,
                     "Struct VMPAGE must be the same size as a system virtual memory
page.");
        // ...
    }
    // ...
};

```

참고 항목

[어설션 및 사용자 제공 메시지\(C++\)](#)

[#error 지시문\(C/C++\)](#)

[assert Macro, _assert, _wassert](#)

[템플릿](#)

[ASCII 문자 집합](#)

[선언 및 정의](#)

C++에서의 모듈 개요

아티클 • 2024. 02. 07.

C++20에는 모듈이 도입되었습니다. 모듈은 소스 파일과 독립적으로 컴파일되는 소스 코드 파일 집합입니다(또는 더 정확하게는 가져오는 [변환 단위](#)).

모듈은 헤더 파일 사용과 관련된 많은 문제를 제거하거나 줄입니다. 종종 컴파일 시간을 단축하며 때로는 크게 단축시킵니다. 모듈에 선언된 매크로, 전처리기 지시문 및 지원되지 않는 이름은 모듈 외부에 표시되지 않습니다. 모듈을 가져오는 번환 단위의 컴파일에는 영향을 주지 않습니다. 매크로 재정의에 대한 우려 없이 모듈을 어떤 순서로든 가져올 수 있습니다. 가져오기 번환 단위의 선언은 가져온 모듈에서 오버로드 확인 또는 이름 조회에 참여하지 않습니다. 모듈이 한 번 컴파일되면 결과는 내보낸 모든 형식, 함수 및 템플릿을 설명하는 이진 파일에 저장됩니다. 컴파일러는 헤더 파일보다 훨씬 더 빠르게 해당 파일을 처리할 수 있습니다. 또한 컴파일러는 이를 프로젝트에서 모듈을 가져오는 모든 위치에서 다시 사용할 수 있습니다.

모듈을 헤더 파일과 함께 사용할 수 있습니다. C++ 원본 파일은 `import` 모듈일 수도 있고 `#include` 헤더 파일일 수도 있습니다. 경우에 따라 헤더 파일을 모듈로 가져올 수 있으며 이는 `#include` 을(를) 사용하여 전처리기로 처리하는 것보다 빠릅니다. 새 프로젝트에서는 가급적 헤더 파일보다는 모듈을 사용하는 것을 권장합니다. 활발히 개발 중인 대규모 기존 프로젝트의 경우 레거시 헤더를 모듈로 변환하는 방법을 실험합니다. 컴파일 시간이 의미 있게 감소되는지 여부에 따라 채택합니다.

모듈을 표준 라이브러리를 가져오는 다른 방법과 대조하려면 [헤더 단위, 모듈 및 미리 컴파일된 헤더 비교](#)를 참조하세요.

Microsoft C++ 컴파일러에서 모듈 사용

Visual Studio 2022 버전 17.1을 기준으로 C++20 표준 모듈은 Microsoft C++ 컴파일러에서 완전히 구현됩니다.

C++20 표준에 의해 지정되기 전에 Microsoft는 모듈에 대한 실험적 지원을 했습니다. 컴파일러는 아래에 설명된 대로 미리 빌드된 표준 라이브러리 모듈 가져오기도 지원했습니다.

Visual Studio 2022 버전 17.5부터 표준 라이브러리를 모듈로 가져오는 것이 Microsoft C++ 컴파일러에서 표준화되었으며 완전히 구현되었습니다. 이 섹션에서는 여전히 지원되는 이전의 실험적 메서드에 대해 설명합니다. 모듈을 사용하여 표준 라이브러리를 가져오는 새로운 표준화된 방법에 대한 자세한 내용은 [모듈을 사용하여 C++ 표준 라이브러리 가져오기](#)를 참조하세요.

모듈 기능을 사용하여 단일 파티션 모듈을 만들고 Microsoft에서 제공하는 표준 라이브러리 모듈을 가져올 수 있습니다. 표준 라이브러리 모듈에 대한 지원을 사용하도록 설정하려면 [/experimental:module](#) 및 [/std:c++latest](#)을(를) 사용하여 컴파일합니다. Visual Studio 프로젝트의 솔루션 탐색기에서 프로젝트 노드를 마우스 오른쪽 단추로 클릭하고 속성을 선택합니다. 구성 드롭다운을 모든 구성으로 설정한 구성 속성>C/C++>언어>C++ 모듈 사용(실험적)을 선택합니다.

모듈과 모듈을 사용하는 코드는 동일한 컴파일러 옵션으로 컴파일되어야 합니다.

C++ 표준 라이브러리를 모듈로 사용(실험적)

이 섹션에서는 여전히 지원되는 실험적 구현에 대해 설명합니다. 모듈로 C++ 표준 라이브러리를 사용하는 새로운 표준화된 방법은 [모듈을 사용하여 C++ 표준 라이브러리 가져오기](#)에 설명되어 있습니다.

C++ 표준 라이브러리를 헤더 파일을 통해 포함하는 대신 모듈로 가져오면 프로젝트 크기에 따라 컴파일 시간을 단축할 수 있습니다. 실험적 라이브러리는 다음과 같은 명명된 모듈로 분할됩니다.

- `std.regex` 은(는) 헤더 `<regex>`의 콘텐츠를 제공합니다.
- `std.filesystem` 은(는) 헤더 `<filesystem>`의 콘텐츠를 제공합니다.
- `std.memory` 은(는) 헤더 `<memory>`의 콘텐츠를 제공합니다.
- `std.threading` 은(는) 헤더 `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` 및 `<thread>`의 콘텐츠를 제공합니다.
- `std.core` 은(는) C++ 표준 라이브러리의 다른 모든 항목을 제공합니다.

이러한 모듈을 사용하려면 소스 코드 파일의 맨 위에 가져오기 선언을 추가합니다. 예시:

C++

```
import std.core;
import std.regex;
```

Microsoft 표준 라이브러리 모듈을 사용하려면 프로그램 [/EHsc](#) 및 [/MD](#) 옵션을 사용하여 프로그램을 컴파일합니다.

예시

다음 예제에서는 `Example.ixx`라는 원본 파일의 간단한 모듈 정의를 보여줍니다. Visual Studio의 모듈 인터페이스 파일에 `.ixx` 확장이 필요합니다. 이 예제에서 인터페이스 파일

에는 함수 정의와 선언이 모두 포함됩니다. 그러나 이후 예제와 같이 하나 이상의 개별 모듈 구현 파일에 정의를 배치할 수도 있습니다.

`export module Example;` 문은 이 파일이 `Example`라는 모듈의 기본 인터페이스임을 나타냅니다. `f()`의 `export` 키워드는 다른 프로그램 또는 모듈이 `Example`을(를) 가져올 때 이 함수가 표시된다는 것을 나타냅니다.

C++

```
// Example.ixx
export module Example;

#define ANSWER 42

namespace Example_NS
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

파일 `MyProgram.cpp`(은)는 `import`(을)를 사용하여 `Example`에서 내보낸 이름에 액세스합니다. 네임스페이스 이름 `Example_NS`은(는) 여기에 표시되지만 일부 멤버는 내보내지 않으므로 표시되지 않습니다. 또한 매크로 `ANSWER`은(는) 내보내지 않으므로 표시되지 않습니다.

C++

```
// MyProgram.cpp
import Example;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Example_NS::f() << endl; // 42
    // int i = Example_NS::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

`import` 선언은 전역 범위에만 나타날 수 있습니다.

모듈 문법

```
module-name:  
  module-name-qualifier-seq opt identifier  
  
module-name-qualifier-seq:  
  identifier .  
  module-name-qualifier-seq identifier .  
  
module-partition:  
  : module-name  
  
module-declaration:  
  export opt module module-name module-partition opt attribute-specifier-seq opt ;  
  
module-import-declaration:  
  export opt import module-name attribute-specifier-seq opt ;  
  export opt import module-partition attribute-specifier-seq opt ;  
  export opt import header-name attribute-specifier-seq opt ;
```

모듈 구현

모듈 인터페이스는 모듈의 이름과 공용 인터페이스를 구성하는 모든 네임스페이스, 유형, 함수 등을 내보냅니다.

모듈 구현은 모듈에서 내보낸 항목을 정의합니다.

가장 간단한 형태로 모듈은 모듈 인터페이스와 구현을 결합하는 단일 파일일 수 있습니다. `.h` 및 `.cpp` 파일에서 수행하는 방법과 유사하게 하나 이상의 별도 모듈 구현 파일에 구현을 넣을 수도 있습니다.

더 큰 모듈의 경우 모듈의 일부를 파티션이라는 하위 모듈로 분할할 수 있습니다. 각 파티션은 모듈 파티션 이름을 내보내는 모듈 인터페이스 파일로 구성됩니다. 파티션에는 하나 이상의 파티션 구현 파일이 있을 수도 있습니다. 모듈 전체에는 모듈의 공용 인터페이스인 하나의 기본 모듈 인터페이스가 있습니다. 원하는 경우 파티션 인터페이스를 내보낼 수 있습니다.

모듈은 하나 이상의 모듈 단위로 구성됩니다. 모듈 단위는 모듈 선언을 포함하는 변환 단위(원본 파일)입니다. 모듈 단위에는 다음과 같은 몇 가지 유형이 있습니다.

- 모듈 인터페이스 단위는 모듈 이름 또는 모듈 파티션 이름을 내보냅니다. 모듈 인터페이스 단위의 모듈 선언에는 `export module` 이(가) 있습니다.

- 모듈 구현 단위는 모듈 이름 또는 모듈 파티션 이름을 내보내지 않습니다. 이름에서 알 수 있듯이 모듈을 구현합니다.
- 기본 모듈 인터페이스 단위는 모듈 이름을 내보냅니다. 모듈에는 기본 모듈 인터페이스 단위가 하나만 있어야 합니다.
- 모듈 파티션 인터페이스 단위는 모듈 파티션 이름을 내보냅니다.
- 모듈 파티션 구현 단위의 모듈 선언에는 모듈 파티션 이름이 있지만 `export` 키워드는 없습니다.

`export` 키워드는 인터페이스 파일에서만 사용됩니다. 구현 파일은 다른 모듈을 `import` 수 있지만 이름은 `export` 수 없습니다. 구현 파일에는 확장명이 있을 수 있습니다.

모듈, 네임스페이스 및 인수 종속 조회

모듈의 네임스페이스에 대한 규칙은 다른 코드와 동일합니다. 네임스페이스 내의 선언을 내보내면 바깥쪽 네임스페이스(해당 네임스페이스에서 명시적으로 내보내지 않은 멤버 제외)도 암시적으로 내보내집니다. 네임스페이스를 명시적으로 내보내면 해당 네임스페이스 정의 내의 모든 선언이 내보내집니다.

컴파일러는 가져오는 변환 단위에서 오버로드 해결을 위해 인수 종속 조회를 수행하는 경우 함수의 인수 유형이 정의된 위치와 동일한 변환 단위(모듈 인터페이스 포함)에 선언된 함수를 고려합니다.

모듈 파티션

모듈 파티션은 다음을 제외하고 모듈과 유사합니다.

- 전체 모듈에서 모든 선언의 소유권을 공유합니다.
- 파티션 인터페이스 파일에서 내보낸 모든 이름은 기본 인터페이스 파일에서 가져오고 내보냅니다.
- 파티션의 이름은 모듈 이름으로 시작하고 그 뒤에 콜론(:)이 와야 합니다.
- 파티션의 선언은 전체 모듈 내에 볼 수 있습니다.\
- ODR(단일 정의 규칙) 오류를 방지하기 위해 특별한 예방 조치가 필요하지 않습니다. 한 파티션에서 이름(함수, 클래스 등)을 선언하고 다른 파티션에서 정의할 수 있습니다.

파티션 구현 파일은 다음과 같이 시작되며 C++ 표준 관점에서는 내부 파티션입니다.

C++

```
module Example:part1;
```

파티션 인터페이스 파일은 다음과 같이 시작됩니다.

C++

```
export module Example:part1;
```

다른 파티션의 선언에 액세스하려면 파티션에서 해당 선언을 가져와야 합니다. 그러나 모듈 이름이 아닌 파티션 이름만 사용할 수 있습니다.

C++

```
module Example:part2;
import :part1;
```

기본 인터페이스 단위는 다음과 같이 모듈의 모든 인터페이스 파티션 파일을 가져오고 다시 내보내야 합니다.

C++

```
export import :part1;
export import :part2;
```

기본 인터페이스 단위는 파티션 구현 파일을 가져올 수 있지만 내보낼 수는 없습니다. 이러한 파일은 이름을 내보낼 수 없습니다. 이 제한으로 인해 모듈에서 구현 세부 정보를 모듈 내부에 유지할 수 있습니다.

모듈 및 헤더 파일

모듈 선언 앞에 `#include` 지시문을 넣어 모듈 원본 파일에 헤더 파일을 포함할 수 있습니다. 이러한 파일은 전역 모듈 조각에 있는 것으로 간주됩니다. 모듈은 명시적으로 포함하는 헤더에 있는 전역 모듈 조각의 이름만 볼 수 있습니다. 전역 모듈 조각에는 사용되는 기호만 포함됩니다.

C++

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

기존 헤더 파일을 사용하여 가져올 모듈을 제어할 수 있습니다.

C++

```
// MyProgram.h
import std.core;
#ifndef DEBUG_LOGGING
import std.filesystem;
#endif
```

가져온 헤더 파일

일부 헤더는 충분히 자체 포함되므로 `import` 키워드를 사용하여 가져올 수 있습니다. 가져온 헤더와 가져온 모듈 간의 주요 차이점은 헤더의 전처리기 정의가 가져오기 프로그램에서 `import` 문 바로 다음에 표시된다는 점입니다.

C++

```
import <vector>;
import "myheader.h";
```

참고 항목

[module, import, export](#)

[명명된 모듈 자습서](#)

[헤더 단위, 모듈 및 미리 컴파일된 헤더 비교](#)

module, , import export

아티클 • 2024. 11. 21.

, 및 선언은 `module` C++20에서 사용할 수 있으며 컴파일러 스위치와 함께 `/std:c++20` 또는 이후 버전(예: `/std:c++latest`)이 필요합니다 `/experimental:module`. `export` `import` 자세한 내용은 [C++의 모듈 개요를 참조하세요](#).

module

`module` 모듈 구현 파일의 시작 부분에 선언을 배치하여 파일 내용이 명명된 모듈에 속하도록 지정합니다.

C++

```
module ModuleA;
```

export

확장 `.ixx` 명이 `export module` 있어야 하는 모듈의 기본 인터페이스 파일에 대한 선언을 사용합니다.

C++

```
export module ModuleA;
```

인터페이스 파일에서 공용 인터페이스의 `export` 일부인 이름에 한정자를 사용합니다.

C++

```
// ModuleA.ixx

export module ModuleA;

namespace ModuleA_NS
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

내보내지 않은 이름은 모듈을 가져오는 코드에 표시되지 않습니다.

C++

```
//MyProgram.cpp

import ModuleA;

int main() {
    ModuleA_NS::f(); // OK
    ModuleA_NS::d(); // OK
    ModuleA_NS::internal_f(); // Ill-formed: error C2065: 'internal_f':
                                undeclared identifier
}
```

export 모듈 구현 파일에 키워드가 표시되지 않을 수 있습니다. 네임스페이스 이름에 적용되면 **export** 네임스페이스의 모든 이름이 내보내집니다.

import

선언을 **import** 사용하여 프로그램에 모듈의 이름을 표시합니다. 선언은 **import** 선언 뒤 **module** 와 지시 **#include** 문 뒤가 아니라 파일의 선언 앞에 나타나야 합니다.

C++

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

설명

module 둘 다 **import** 논리 줄의 시작 부분에 나타나는 경우에만 키워드로 처리됩니다.

C++

```
// OK:
module ;
module module-name
import :
import <
import "
```

```
import module-name
export module ;
export module module-name
export import :
export import <
export import "
export import module-name

// Error:
int i; module ;
```

Microsoft 전용

Microsoft C++에서 토큰 `import` 은 `module` 매크로에 대한 인수로 사용될 때 항상 식별자이며 키워드가 되지 않습니다.

예시

C++

```
#define foo(...) __VA_ARGS__
foo(
    import // Always an identifier, never a keyword
)
```

End Microsoft Specific

참고 항목

[C++에서의 모듈 개요](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

자습서: 명령줄에서 모듈을 사용하여 C++ 표준 라이브러리 가져오기

아티클 • 2024. 02. 01.

C++ 라이브러리 모듈을 사용하여 C++ 표준 라이브러리를 가져오는 방법을 알아봅니다. 이로 인해 헤더 파일이나 헤더 단위 또는 PCH(미리 컴파일된 헤더)를 사용하는 것보다 더 빠르게 컴파일하고 더 강력해집니다.

이 자습서에서는 다음에 대해 알아봅니다.

- 명령줄에서 표준 라이브러리를 모듈로 가져오는 방법.
- 모듈의 성능 및 유용성 이점.
- 두 가지 표준 라이브러리 모듈(`std` 및 `std.compat`)과 상호 간의 차이점.

필수 조건

이 자습서에는 Visual Studio 2022 17.5 이상이 필요합니다.

표준 라이브러리 모듈 소개

헤더 파일은 매크로 정의 및 포함하는 순서에 따라 변경될 수 있는 의미 체계로 인해 문제가 발생하며 컴파일 속도가 느려집니다. 모듈에서 이러한 문제를 해결합니다.

이제 표준 라이브러리를 헤더 파일의 복잡한 모음 대신 모듈로 가져올 수 있습니다. 이는 헤더 파일이나 헤더 단위 또는 PCH(미리 컴파일된 헤더)를 포함하는 것보다 훨씬 빠르고 강력합니다.

C++23 표준 라이브러리에는 두 개의 명명된 모듈, 즉 `std` 및 `std.compat` 가 도입되었습니다.

- `std` 는 C++ 표준 라이브러리 네임스페이스 `std`(예: `std::vector`)에 정의된 선언 및 이름을 내보냅니다. 또한 `std::printf()` 와 같은 함수를 제공하는 `<cstdio>` 및 `<cstdlib>` 와 같은 C 래퍼 헤더의 콘텐츠를 내보냅니다. `::printf()` 와 같은 전역 네임스페이스에 정의된 C 함수는 내보내지지 않습니다. 이는 `<cstdio>` also 같은 C 래퍼 헤더를 포함하면 C 전역 네임스페이스 버전을 가져오는 `stdio.h` 같은 C 헤더 파일이 포함되는 상황을 개선합니다. `std` 를 가져오면 문제가 되지 않습니다.
- `std.compat` 는 `std` 의 모든 항목을 내보내고 `::printf`, `::fopen`, `::size_t`, `::strlen` 등과 같은 C 런타임 전역 네임스페이스를 추가합니다. `std.compat` 모듈을 사용하면

전역 네임스페이스의 많은 C 런타임 함수/형식을 참조하는 코드베이스 작업을 더 쉽게 할 수 있습니다.

컴파일러는 `import std;` 또는 `import std.compat;`를 사용할 때 전체 표준 라이브러리를 가져오며, 단일 헤더 파일을 가져오는 것보다 더 빠르게 수행합니다. 예를 들어, `#include <vector>` 보다 `import std;` (또는 `import std.compat`)를 사용하여 전체 표준 라이브러리를 가져오는 것이 더 빠릅니다.

명명된 모듈은 매크로를 노출하지 않기 때문에 `std` 또는 `std.compat`를 가져올 때 `assert`, `errno`, `offsetof`, `va_arg` 등과 같은 매크로를 사용할 수 없습니다. 해결 방법은 표준 라이브러리 명명된 모듈 고려 사항을 참조하세요.

C++ 모듈 정보

헤더 파일을 사용하여 C++의 원본 파일 간에 선언과 정의를 공유합니다. 표준 라이브러리 모듈 이전에 `#include <vector>`와 같은 지시문을 사용하여 필요한 표준 라이브러리 부분을 포함합니다. 헤더 파일은 포함 순서나 특정 매크로 정의 여부에 따라 의미 체계가 변경될 수 있으므로 손상될 수 있으며 구성하기 어렵습니다. 또한 이를 포함하는 모든 원본 파일에서 재처리되기 때문에 컴파일 속도가 느려집니다.

C++20에는 모듈이라는 최신 대안이 도입되었습니다. C++23에서는 모듈 지원을 활용하여 표준 라이브러리를 나타내는 명명된 모듈을 도입할 수 있었습니다.

헤더 파일과 마찬가지로 모듈을 사용하면 원본 파일 전체에서 선언과 정의를 공유할 수 있습니다. 그러나 헤더 파일과 달리 모듈은 매크로 정의나 가져오는 순서로 인해 의미 체계가 변경되지 않기 때문에 손상될 수 없으며 구성하기가 쉽습니다. 컴파일러는 `#include` 파일을 처리하는 것보다 훨씬 빠르게 모듈을 처리할 수 있으며 컴파일 시간에도 더 적은 메모리를 사용합니다. 명명된 모듈은 매크로 정의나 프라이빗 구현 세부 정보를 노출하지 않습니다.

모듈에 대한 자세한 내용은 [C++ 모듈 개요](#)를 참조하세요. 이 문서에서는 C++ 표준 라이브러리를 모듈로 사용하는 방법도 설명하지만 이전의 실험적인 방법을 사용합니다.

이 문서에서는 표준 라이브러리를 사용하는 최선의 새로운 방법을 보여 줍니다. 표준 라이브러리를 사용하는 다른 방법에 대한 자세한 내용은 [헤더 단위, 모듈 및 미리 컴파일된 헤더 비교](#)를 참조하세요.

std를 사용하여 표준 라이브러리 가져오기

다음 예에서는 명령줄 컴파일러를 사용하여 표준 라이브러리를 모듈로 사용하는 방법을 보여 줍니다. Visual Studio IDE에서 이 작업을 수행하는 방법에 대한 자세한 내용은 [ISO](#)

C++23 표준 라이브러리 모듈 빌드를 참조하세요.

`import std;` 또는 `import std.compat;` 문은 표준 라이브러리를 애플리케이션으로 가져옵니다. 하지만 먼저 표준 라이브러리 명명된 모듈을 이진 파일 형식으로 컴파일해야 합니다. 다음 단계에서는 방법을 보여 줍니다.

예: `std` 를 빌드하고 가져오는 방법

1. VS용 x86 Native Tools 명령 프롬프트를 엽니다. Windows 시작 메뉴에서 *x86 native* 를 입력하면 앱 목록에 프롬프트가 나타납니다. Visual Studio 2022 버전 17.5 이상의 프롬프트인지 확인합니다. 잘못된 버전의 프롬프트를 사용하면 오류가 발생합니다. 이 자습서에 사용된 예는 CMD 셸에 대한 것입니다.
2. `%USERPROFILE%\source\repos\STLModules` 와 같은 디렉터리를 만들고 이를 현재 디렉터리로 만듭니다. 쓰기 권한이 없는 디렉터리를 선택하면 컴파일 중에 오류가 발생합니다.
3. 다음 명령을 사용하여 `std` 명명된 모듈을 컴파일합니다.

Windows 명령 프롬프트

```
c1 /std:c++latest /EHsc /nologo /W4 /c  
"%VCToolsInstallDir%\modules\std..hxx"
```

오류가 발생하면 올바른 버전의 명령 프롬프트를 사용하고 있는지 확인합니다.

빌드된 모듈을 가져오는 코드와 함께 사용하려는 동일한 컴파일러 설정을 사용하여 `std` 명명된 모듈을 컴파일합니다. 다중 프로젝트 솔루션이 있는 경우 표준 라이브러리 명명된 모듈을 한 번 컴파일한 다음 [/reference](#) 컴파일러 옵션을 사용하여 모든 프로젝트에서 이를 참조할 수 있습니다.

이전 컴파일러 명령을 사용하여 컴파일러는 두 개의 파일을 출력합니다.

- `std.ifc` 는 컴파일러가 `import std;` 문을 처리하기 위해 참조하는 명명된 모듈 인터페이스의 컴파일된 이진 파일 표현입니다. 이는 컴파일 시간 전용 아티팩트입니다. 사용자의 애플리케이션과 함께 제공되지 않습니다.
- `std.obj` 에는 명명된 모듈의 구현이 포함되어 있습니다. 표준 라이브러리에서 사용하는 기능을 애플리케이션에 정적으로 연결하려면 샘플 앱을 컴파일할 때 명령줄에 `std.obj` 를 추가합니다.

이 예의 주요 명령줄 스위치는 다음과 같습니다.

스위치	의미
/std:c++:latest	최신 버전의 C++ 언어 표준 및 라이브러리를 사용합니다. 모듈 지원은 /std:c++20에서 사용할 수 있지만 표준 라이브러리 명명된 모듈에 대한 지원을 가져오려면 최신 표준 라이브러리가 필요합니다.
/EHsc	extern "C"로 표시된 함수를 제외하고 C++ 예외 처리를 사용합니다.
/W4	모든 수준 1, 수준 2, 수준 3 및 대부분의 수준 4(정보) 경고를 사용하도록 설정하여 잠재적인 문제를 조기에 파악하는 데 도움이 되기 때문에 일반적으로 /W4를 사용하는 것이 좋습니다(특히 새 프로젝트의 경우). 이를 사용하면 찾기 어려운 코드 결함을 최소화할 수 있는 매우 사소한 경고가 기본적으로 제공됩니다.
/c	이 시점에서 명명된 모듈 인터페이스의 이진 파일을 빌드하고 있으므로 링크 없이 컴파일합니다.

다음 스위치를 사용하여 개체 파일 이름과 명명된 모듈 인터페이스 파일 이름을 제어할 수 있습니다.

- /Fo는 개체 파일의 이름을 설정합니다. 예: /Fo:"somethingelse". 기본적으로 컴파일러는 컴파일 중인 모듈 원본 파일(.icxx)과 동일한 개체 파일 이름을 사용합니다. 이 예에서는 모듈 파일 std.ixxx를 컴파일하고 있으므로 개체 파일 이름은 기본적으로 std.obj입니다.
- /ifcOutput은 명명된 모듈 인터페이스 파일(.ifc)의 이름을 설정합니다. 예: /ifcOutput "somethingelse.ifc". 기본적으로 컴파일러는 모듈 인터페이스 파일(.ifc)에 대해 컴파일 중인 모듈 원본 파일(.icxx)과 동일한 이름을 사용합니다. 이 예에서는 모듈 파일 std.ixxx를 컴파일하고 있으므로 생성된 ifc 파일은 기본적으로 std.ifc입니다.

4. 먼저 다음 콘텐츠로 importExample.cpp라는 파일을 만들어 빌드한 std 라이브러리를 가져옵니다.

```
C++

// requires /std:c++latest

import std;

int main()
{
    std::cout << "Import the STL library for best performance\n";
    std::vector<int> v{5, 5, 5};
    for (const auto& e : v)
    {
        std::cout << e;
    }
}
```

```
}
```

앞의 코드에서 `import std;` 는 `#include <vector>` 및 `#include <iostream>` 을 바꿉니다. `import std;` 문은 하나의 문으로 모든 표준 라이브러리를 사용할 수 있게 만듭니다. 전체 표준 라이브러리를 가져오는 것이 `#include <vector>` 와 같은 단일 표준 라이브러리 헤더 파일을 처리하는 것보다 훨씬 빠른 경우가 많습니다.

5. 이전 단계와 동일한 디렉터리에서 다음 명령을 사용하여 예를 컴파일합니다.

Windows 명령 프롬프트

```
c1 /c /std:c++latest /EHsc /nologo /W4 /reference "std=std.ifc"  
importExample.cpp  
link importExample.obj std.lib
```

컴파일러는 `import` 문에서 지정한 모듈 이름과 일치하는 `.ifc` 파일을 자동으로 찾기 때문에 이 예에서는 명령줄에 `/reference "std=std.ifc"` 를 지정할 필요가 없습니다. 컴파일러가 `import std;` 를 발견하면 소스 코드와 동일한 디렉터리에 있으면 `std.ifc` 를 찾을 수 있습니다. `.ifc` 파일이 소스 코드와 다른 디렉터리에 있는 경우 `/reference` 컴파일러 스위치를 사용하여 참조하세요.

이 예에서는 소스 코드를 컴파일하는 것과 모듈 구현을 애플리케이션에 연결하는 단계는 별도로 수행되지만, 그럴 필요는 없습니다. `c1 /std:c++latest /EHsc /nologo /W4 /reference "std=std.ifc" importExample.cpp std.lib` 를 사용하면 한 단계로 컴파일하고 링크할 수 있습니다. 그러나 별도로 빌드하고 링크하는 것이 편리할 수 있습니다. 그러면 표준 라이브러리 명명된 모듈을 한 번만 빌드한 다음 빌드의 링크 단계에서 특정 프로젝트 또는 여러 프로젝트에서 이를 참조할 수 있기 때문입니다.

단일 프로젝트를 빌드하는 경우 `std` 표준 라이브러리 명명된 모듈을 빌드하는 단계와 명령줄에 `"%VCToolsInstallDir%\modules\std.ixx"` 를 추가하여 애플리케이션을 빌드하는 단계를 결합할 수 있습니다. `std` 모듈을 사용하는 `.cpp` 파일 앞에 배치합니다.

기본적으로 출력 실행 파일의 이름은 첫 번째 입력 파일에서 가져옵니다. 원하는 실행 파일 이름을 지정하려면 `/Fe` 컴파일러 옵션을 사용합니다. 이 자습서에서는 표준 라이브러리 명명된 모듈을 한 번만 빌드하면 프로젝트 또는 여러 프로젝트에서 이를 참조할 수 있으므로 `std` 라는 모듈을 별도의 단계로 컴파일하는 방법을 보여줍니다. 그러나 다음 명령줄에서 볼 수 있듯이 모든 것을 함께 빌드하는 것이 편리할 수 있습니다.

Windows 명령 프롬프트

```
c1 /FeimportExample /std:c++latest /EHsc /nologo /W4  
"%VCToolsInstallDir%\modules\std.ixx" importExample.cpp
```

이전 명령줄이 주어지면 컴파일러는 `importExample.exe`라는 실행 파일을 생성합니다. 실행하면 다음과 같은 출력이 생성됩니다.

출력

```
Import the STL library for best performance  
555
```

`std.compat`를 사용하여 표준 라이브러리 및 전역 C 함수 가져오기

C++ 표준 라이브러리에는 ISO C 표준 라이브러리가 포함되어 있습니다. `std.compat` 모듈은 `std::vector`, `std::cout`, `std::printf`, `std::scanf` 등과 같은 `std` 모듈의 모든 기능을 제공합니다. 그러나 `::printf`, `::scanf`, `::fopen`, `::size_t` 등과 같은 함수의 전역 네임스페이스 버전도 제공합니다.

`std.compat` 명명된 모듈은 전역 네임스페이스에서 C 런타임 함수를 참조하는 기존 코드를 쉽게 마이그레이션하기 위해 제공되는 호환성 계층입니다. 전역 네임스페이스에 이름을 추가하지 않으려면 `import std;`를 사용합니다. 한정되지 않은(전역 네임스페이스) C 런타임 함수를 많이 사용하는 코드베이스를 쉽게 마이그레이션해야 하는 경우 `import std.compat;`를 사용합니다. 이는 전역 네임스페이스 C 런타임 이름을 제공하므로 모든 전역 이름을 `std::`로 한정할 필요가 없습니다. 전역 네임스페이스 C 런타임 함수를 사용하는 기존 코드가 없으면 `import std.compat;`를 사용할 필요가 없습니다. 코드에서 몇 가지 C 런타임 함수만 호출하는 경우 `import std;`를 사용하고 이를 필요로 하는 몇 가지 전역 네임스페이스 C 런타임 이름을 `std::`로 한정하는 것이 더 나을 수 있습니다. 예:

`std::printf()` 코드를 컴파일하려고 할 때 `error C3861: 'printf': identifier not found`와 같은 오류가 표시되면 `import std.compat;`를 사용하여 전역 네임스페이스 C 런타임 함수를 가져오는 것이 좋습니다.

예: `std.compat`를 빌드하고 가져오는 방법

`import std.compat;`를 사용하려면 먼저 `std.compat.ixx`의 소스 코드 형식에 있는 모듈 인터페이스 파일을 컴파일해야 합니다. Visual Studio에서는 프로젝트와 일치하는 컴파일러 설정을 사용하여 모듈을 컴파일할 수 있도록 모듈의 소스 코드를 제공합니다. 이러한

단계는 `std` 명명된 모듈을 빌드하는 단계와 유사합니다. `std` 명명된 모듈이 먼저 빌드되고 이에 따라 달라지는 `std.compat` 가 빌드됩니다.

1. VS용 Native Tools 명령 프롬프트를 엽니다. Windows 시작 메뉴에서 x86 `N|I/E|B` 을 입력하면 앱 목록에 프롬프트가 나타납니다. Visual Studio 2022 버전 17.5 이상의 프롬프트인지 확인합니다. 잘못된 버전의 프롬프트를 사용하면 컴파일러 오류가 발생합니다.
2. 이 예를 시도하기 위해 `%USERPROFILE%\source\repos\STLModules` 와 같은 디렉터리를 만들고 이를 현재 디렉터리로 만듭니다. 쓰기 권한이 없는 디렉터리를 선택하면 오류가 발생합니다.
3. 다음 명령을 사용하여 `std` 및 `std.compat` 명명된 모듈을 컴파일합니다.

Windows 명령 프롬프트

```
c1 /std:c++latest /EHsc /nologo /W4 /c  
"%VCToolsInstallDir%\modules\std.ixx"  
"%VCToolsInstallDir%\modules\std.compat.ixx"
```

가져올 코드와 함께 사용하려는 것과 동일한 컴파일러 설정을 사용하여 `std` 및 `std.compat` 를 컴파일해야 합니다. 다중 프로젝트 솔루션이 있는 경우 이를 한 번 컴파일한 다음 [/reference](#) 컴파일러 옵션을 사용하여 모든 프로젝트에서 참조할 수 있습니다.

오류가 발생하면 올바른 버전의 명령 프롬프트를 사용하고 있는지 확인합니다.

컴파일러는 이전 두 단계에서 네 개의 파일을 출력합니다.

- `std.ifc` 는 컴파일러가 `import std;` 문을 처리하기 위해 참조하는 명명된 모듈 인터페이스의 컴파일된 이진 파일입니다. 또한 `std.compat` 는 `std` 를 기반으로 빌드되므로 컴파일러는 `std.ifc` 를 참조하여 `import std.compat;` 를 처리합니다. 이는 컴파일 시간 전용 아티팩트입니다. 사용자의 애플리케이션과 함께 제공되지 않습니다.
- `std.obj` 에는 표준 라이브러리 구현이 포함되어 있습니다.
- `std.compat.ifc` 는 컴파일러가 `import std.compat;` 문을 처리하기 위해 참조하는 명명된 모듈 인터페이스의 컴파일된 이진 파일입니다. 이는 컴파일 시간 전용 아티팩트입니다. 사용자의 애플리케이션과 함께 제공되지 않습니다.
- `std.compat.obj` 에는 구현이 포함되어 있습니다. 그러나 대부분의 구현은 `std.obj` 에서 제공됩니다. 표준 라이브러리에서 사용하는 기능을 애플리케이션에 정적으로 연결하려면 샘플 앱을 컴파일할 때 명령줄에 `std.obj` 를 추가합니다.

다음 스위치를 사용하여 개체 파일 이름과 명명된 모듈 인터페이스 파일 이름을 제어할 수 있습니다.

- `/Fo`는 개체 파일의 이름을 설정합니다. 예: `/Fo:"somethingelse"`. 기본적으로 컴파일러는 컴파일 중인 모듈 원본 파일(`.ixx`)과 동일한 개체 파일 이름을 사용합니다. 이 예에서는 모듈 파일 `std.ixx` 및 `std.compat.obj`를 컴파일하고 있으므로 개체 파일 이름은 기본적으로 `std.obj` 및 `std.compat.obj`입니다.
- `/ifcOutput`은 명명된 모듈 인터페이스 파일(`.ifc`)의 이름을 설정합니다. 예: `/ifcOutput "somethingelse.ifc"`. 기본적으로 컴파일러는 모듈 인터페이스 파일(`.ifc`)에 대해 컴파일 중인 모듈 원본 파일(`.ixx`)과 동일한 이름을 사용합니다. 이 예에서는 모듈 파일 `std.ixx` 및 `std.compat.ixx`를 컴파일하고 있으므로 생성된 `.ifc` 파일은 기본적으로 `std.ifc` 및 `std.compat.ifc`입니다.

4. 먼저 다음 콘텐츠로 `stdCompatExample.cpp`라는 파일을 만들어 `std.compat` 라이브러리를 가져옵니다.

```
C++  
  
import std.compat;  
  
int main()  
{  
    printf("Import std.compat to get global names like printf()\n");  
  
    std::vector<int> v{5, 5, 5};  
    for (const auto& e : v)  
    {  
        printf("%i", e);  
    }  
}
```

앞의 코드에서 `import std.compat;` 는 `#include <cstdio>` 및 `#include <vector>`를 바꿉니다. `import std.compat;` 문은 하나의 문으로 표준 라이브러리와 C 런타임 함수를 사용할 수 있게 만듭니다. C++ 표준 라이브러리 및 C 런타임 라이브러리 전역 네임스페이스 함수를 포함하는 이 명명된 모듈을 가져오는 것이 `#include <vector>` 와 같은 단일 `#include`를 처리하는 것보다 빠릅니다.

5. 다음 명령을 사용하여 예를 컴파일합니다.

Windows 명령 프롬프트

```
c1 /std:c++latest /EHsc /nologo /W4 stdCompatExample.cpp  
link stdCompatExample.obj std.obj std.compat.obj
```

컴파일러가 `import` 문에서 모듈 이름과 일치하는 `.ifc` 파일을 자동으로 찾기 때문에 명령줄에서 `std.compat.ifc`를 지정할 필요가 없었습니다. 컴파일러가 `import std.compat;` 를 발견하면 소스 코드와 동일한 디렉터리에 넣기 때문에 `std.compat.ifc`를 찾습니다. 이를 통해 명령줄에서 이를 지정할 필요가 없습니다. `.ifc` 파일이 소스 코드와 다른 디렉터리에 있거나 이름이 다른 경우 [/reference](#) 컴파일러 스위치를 사용하여 참조합니다.

`std.compat` 를 가져올 때 `std.compat` 가 `std.obj` 의 코드를 사용하므로 `std.compat` 및 `std.obj` 모두에 대해 링크해야 합니다.

단일 프로젝트를 빌드하는 경우 명령줄에 `"%VCToolsInstallDir%\modules\std.ixx"` 및 `"%VCToolsInstallDir%\modules\std.compat.ixx"`(순서대로)를 추가하여 `std` 및 `std.compat` 표준 라이브러리 명명된 모듈의 빌드 단계를 결합할 수 있습니다. 이 자습서에서는 표준 라이브러리 명명된 모듈을 한 번만 빌드하면 프로젝트 또는 여러 프로젝트에서 참조할 수 있으므로 표준 라이브러리 모듈을 별도의 단계로 빌드하는 방법을 보여 줍니다. 그러나 한 번에 모두 빌드하는 것이 편리한 경우 이를 사용하는 `.cpp` 파일 앞에 배치하고 이 예에 표시된 대로 `/Fe` 를 지정하여 빌드된 `exe` 의 이름을 지정합니다.

Windows 명령 프롬프트

```
c1 /c /FestdCompatExample /std:c++latest /EHsc /nologo /W4  
"%VCToolsInstallDir%\modules\std.ixx"  
"%VCToolsInstallDir%\modules\std.compat.ixx" stdCompatExample.cpp  
link stdCompatExample.obj std.obj std.compat.obj
```

이 예에서는 소스 코드를 컴파일하는 것과 모듈 구현을 애플리케이션에 연결하는 단계는 별도로 수행되지만, 그럴 필요는 없습니다. `c1 /std:c++latest /EHsc /nologo /W4 stdCompatExample.cpp std.obj std.compat.obj` 를 사용하면 한 단계로 컴파일하고 링크할 수 있습니다. 그러나 별도로 빌드하고 링크하는 것이 편리할 수 있습니다. 그러면 표준 라이브러리 명명된 모듈을 한 번만 빌드한 다음 빌드의 링크 단계에서 특정 프로젝트 또는 여러 프로젝트에서 이를 참조할 수 있기 때문입니다.

이전 컴파일러 명령은 `stdCompatExample.exe` 라는 실행 파일을 생성합니다. 실행하면 다음과 같은 출력이 생성됩니다.

출력

```
Import std.compat to get global names like printf()  
555
```

표준 라이브러리 명명된 모듈 고려 사항

명명된 모듈의 버전 관리는 헤더의 버전 관리와 동일합니다. `.ixx` 명명된 모듈 파일은 헤더와 함께 설치됩니다. 예를 들어, `"%VCToolsInstallDir%\modules\std.ixx`는 이 글을 작성할 때 사용된 도구 버전에서 `C:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.38.33130\modules\std.ixx`로 확인됩니다. 사용할 헤더 파일의 버전을 선택하는 것과 같은 방법으로 명명된 모듈의 버전을 선택합니다. 즉, 참조하는 디렉터리를 기준으로 합니다.

가져오기 헤더 단위와 명명된 모듈을 혼합하여 일치시키지 마세요. 예를 들어, 동일한 파일에 `import <vector>`; 와 `import std;`를 사용하지 마세요.

C++ 표준 라이브러리 헤더 파일 가져오기와 명명된 모듈 `std` 또는 `std.compat`를 혼합하여 일치시키지 마세요. 예를 들어, 동일한 파일에 `#include <vector>`와 `import std;`를 사용하지 마세요. 그러나 C 헤더를 포함하고 동일한 파일에 명명된 모듈을 가져올 수 있습니다. 예를 들어, 동일한 파일에서 `import std;` 및 `#include <math.h>`를 사용할 수 있습니다. C++ 표준 라이브러리 버전 `<cmath>`를 포함하지 마세요.

모듈을 여러 번 가져오는 것을 방어할 필요는 없습니다. 즉, 모듈에는 `#ifndef` 스타일 헤더 가드가 필요하지 않습니다. 컴파일러는 명명된 모듈을 이미 가져온 시기를 알고 이를 수행하려는 중복 시도를 무시합니다.

`assert()` 매크로를 사용해야 하는 경우 `#include <assert.h>`를 사용하며,

`errno` 매크로를 사용해야 하는 경우 `#include <errno.h>`를 사용합니다. 예를 들어, 명명된 모듈은 매크로를 노출하지 않기 때문에 이는 `<math.h>`에서 오류를 확인해야 하는 경우의 해결 방법입니다.

`NAN`, `INFINITY` 및 `INT_MIN`과 같은 매크로는 포함할 수 있는 `<limits.h>`에 의해 정의됩니다. 그러나 `import std;`에서 `NAN` 및 `INFINITY` 대신 `numeric_limits<double>::quiet_NaN()` 및 `numeric_limits<double>::infinity()`를 사용할 수 있고, `INT_MIN` 대신 `std::numeric_limits<int>::min()`을 사용할 수 있습니다.

요약

이 자습서에서는 모듈을 사용하여 표준 라이브러리를 가져왔습니다. 다음으로, [C++의 명명된 모듈 자습서](#)에서 고유의 모듈을 만들고 가져오는 방법을 알아봅니다.

참고 항목

[헤더 단위, 모듈 및 미리 컴파일된 헤더 비교](#)

[C++에서의 모듈 개요](#)

[Visual Studio의 C++ 모듈 둘러보기 ↗](#)

[C++ 명명된 모듈로 프로젝트 이동 ↗](#)

명명된 모듈 자습서(C++)

아티클 • 2023. 04. 03.

이 자습서에서는 C++20 모듈을 만드는 방법에 대해 설명합니다. 모듈은 헤더 파일을 대체합니다. 모듈이 헤더 파일을 개선하는 방법을 알아봅니다.

이 자습서에서는 다음 방법을 알아봅니다.

- 모듈 만들기 및 가져오기
- 기본 모듈 인터페이스 단위 만들기
- 모듈 파티션 파일 만들기
- 모듈 단위 구현 파일 만들기

필수 구성 요소

이 자습서에는 Visual Studio 2022 17.1.0 이상이 필요합니다.

이 자습서의 코드 예제에서 작업하는 동안 IntelliSense 오류가 발생할 수 있습니다. IntelliSense 엔진에서의 작업이 컴파일러를 따라잡고 있습니다. IntelliSense 오류는 무시할 수 있으며 코드 예제가 빌드되지 않도록 방지할 수 있습니다. IntelliSense 작업의 진행률을 추적하려면 이 [문제를 참조하세요](#).

C++ 모듈이란?

헤더 파일은 C++의 원본 파일 간에 선언과 정의를 공유하는 방법입니다. 헤더 파일은 깨지기 쉽고 작성하기 어렵습니다. 포함된 순서 또는 정의되지 않은 매크로에 따라 다르게 컴파일할 수 있습니다. 포함된 각 원본 파일에 대해 다시 처리되므로 컴파일 시간이 느려질 수 있습니다.

C++20에서는 C++ 프로그램을 구성하기 위한 최신 접근 방식인 모듈을 소개합니다.

헤더 파일과 마찬가지로 모듈을 사용하면 소스 파일 간에 선언 및 정의를 공유할 수 있습니다. 그러나 헤더 파일과 달리 모듈은 매크로 정의 또는 프라이빗 구현 세부 정보를 누출하지 않습니다.

매크로 정의나 가져온 항목, 가져오기 순서 등으로 인해 의미 체계가 변경되지 않으므로 모듈을 더 쉽게 작성할 수 있습니다. 또한 소비자에게 표시되는 항목을 더 쉽게 제어할 수 있습니다.

모듈은 헤더 파일이 보호되지 않도록 추가 안전 보장을 제공합니다. 컴파일러와 링커는 함께 작동하여 가능한 이름 충돌 문제를 방지하고 더 강력한 ODR (정의 규칙) 보장을 제

공합니다.

강력한 소유권 모델은 링커가 내보낸 이름을 내보내는 모듈에 연결하기 때문에 링크 타임에 이름 간의 충돌을 방지합니다. 이 모델을 사용하면 Microsoft Visual C++ 컴파일러가 동일한 프로그램에서 비슷한 이름을 보고하는 다른 모듈을 연결하여 발생하는 정의되지 않은 동작을 방지할 수 있습니다. 자세한 내용은 [강력한 소유권](#)을 참조하세요.

모듈은 이진 파일로 컴파일된 하나 이상의 소스 코드 파일로 구성됩니다. 이진 파일은 모듈에서 내보낸 모든 형식, 함수 및 템플릿을 설명합니다. 원본 파일이 모듈을 가져오면 컴파일러는 모듈의 내용이 포함된 이진 파일을 읽습니다. 이진 파일을 읽는 것은 헤더 파일을 처리하는 것보다 훨씬 빠릅니다. 또한 모듈을 가져올 때마다 컴파일러에서 이진 파일을 다시 사용하므로 시간이 훨씬 더 절약됩니다. 모듈은 가져올 때마다가 아니라 한 번 빌드되므로 빌드 시간을 단축할 수 있으며 경우에 따라 크게 줄일 수 있습니다.

더 중요한 것은 모듈에는 헤더 파일이 수행하는 취약성 문제가 없다는 것입니다. 모듈을 가져오면 모듈의 의미 체계 또는 가져온 다른 모듈의 의미 체계가 변경되지 않습니다. 모듈에 선언된 매크로, 전처리기 지시문 및 내보내지 않은 이름은 가져오는 원본 파일에 표시되지 않습니다. 모듈을 순서대로 가져올 수 있으며 모듈의 의미는 변경되지 않습니다.

모듈은 헤더 파일과 나란히 사용할 수 있습니다. 이 기능은 모듈을 사용하도록 코드 베이스를 마이그레이션하는 경우 단계별로 수행할 수 있으므로 편리합니다.

경우에 따라 헤더 파일이 아닌 헤더 단위로 `#include` 가져올 수 있습니다. 헤더 단위는 [PCH\(미리 컴파일된 헤더\)](#) 파일 대신 사용하는 것이 좋습니다. [공유 PCH](#) 파일보다 쉽게 설정하고 사용할 수 있지만 비슷한 성능 이점을 제공합니다. 자세한 내용은 [연습: Microsoft Visual C++ 헤더 단위 빌드 및 가져오기를 참조하세요.](#)

코드는 정적 라이브러리 프로젝트에 대한 프로젝트-프로젝트 참조를 사용하여 동일한 프로젝트 또는 참조된 프로젝트의 모듈을 자동으로 사용할 수 있습니다.

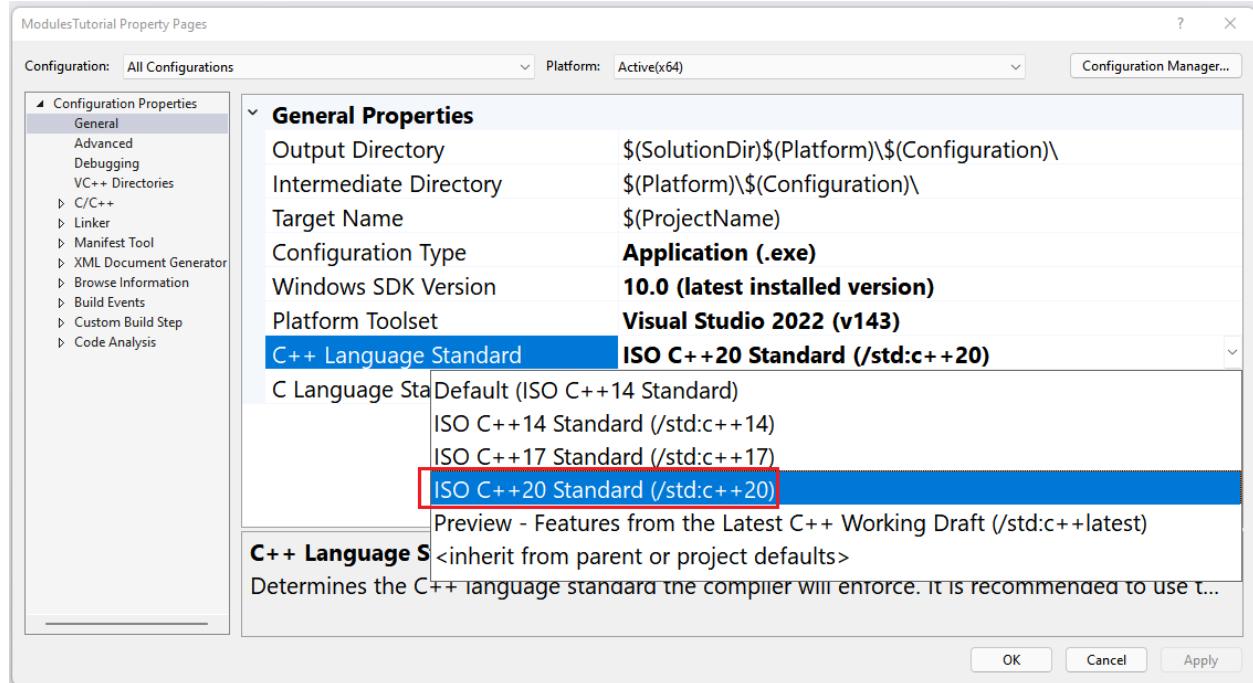
프로젝트를 만듭니다.

간단한 프로젝트를 빌드할 때 모듈의 다양한 측면을 살펴보겠습니다. 프로젝트는 헤더 파일 대신 모듈을 사용하여 API를 구현합니다.

Visual Studio 2022 이상에서 [새 프로젝트 만들기](#)를 선택한 다음 [콘솔 앱](#) (C++의 경우) 프로젝트 유형을 선택합니다. 이 프로젝트 유형을 사용할 수 없는 경우 Visual Studio를 설치할 때 [C++를 사용한 데스크톱 개발](#) 워크로드를 선택하지 않았을 수 있습니다. Visual Studio 설치 관리자 사용하여 C++ 워크로드를 추가할 수 있습니다.

새 프로젝트에 이름을 `ModulesTutorial` 지정하고 프로젝트를 만듭니다.

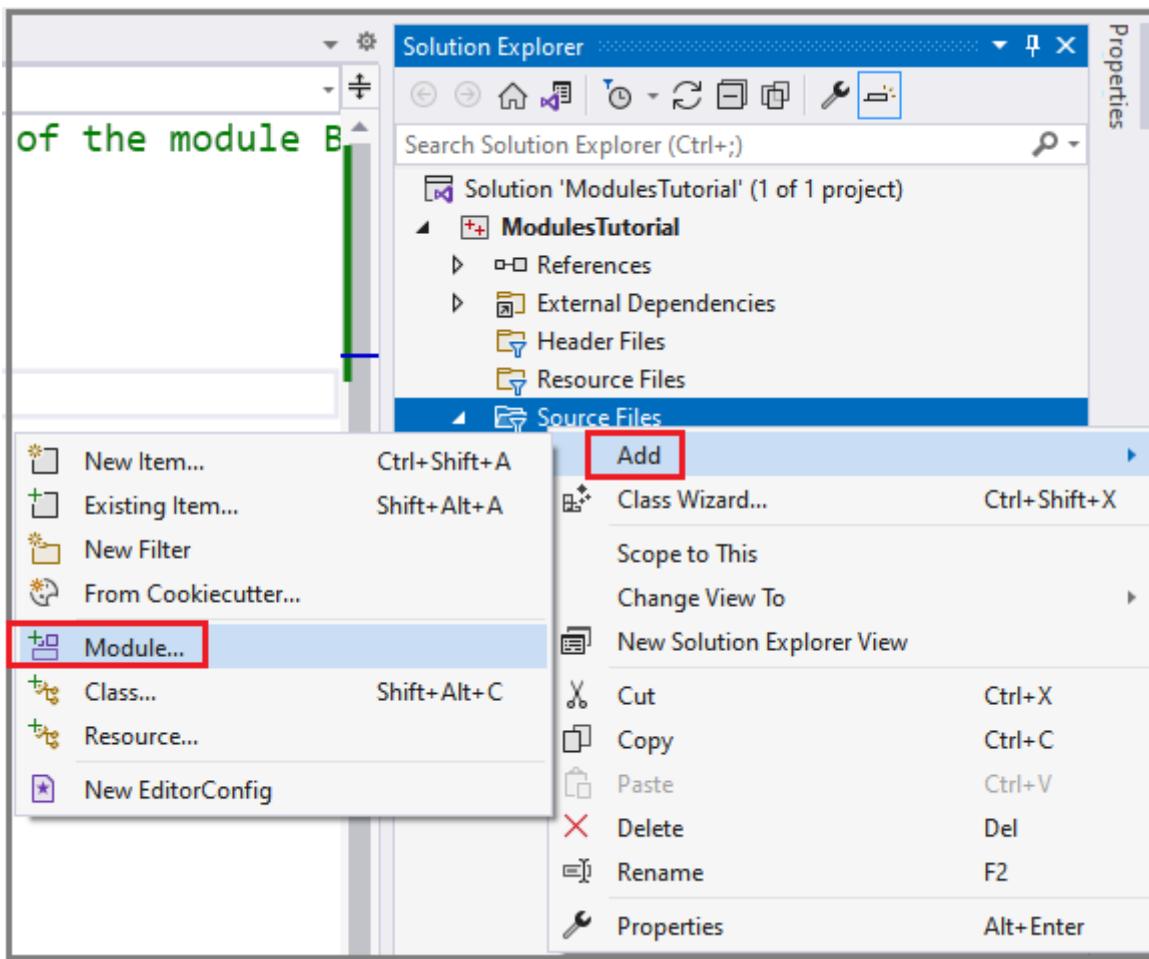
모듈은 C++20 기능이므로 또는 /std:c++latest 컴파일러 옵션을 사용합니다./std:c++20. 솔루션 탐색기 프로젝트 이름을 ModulesTutorial 마우스 오른쪽 단추로 클릭한 다음 속성을 선택합니다. 프로젝트 속성 페이지 대화 상자에서 구성 을 모든 구성으로 , 플랫폼을 모든 플랫폼으로 변경합니다. 왼쪽의 트리 뷰 창에서 구성 속성>일반 을 선택합니다. C++ 언어 표준 속성을 선택합니다. 드롭다운을 사용하여 속성 값을 ISO C++20 Standard(/std:c++20)로 변경합니다. 확인을 선택하여 변경 내용을 적용합니다.



기본 모듈 인터페이스 단위 만들기

모듈은 하나 이상의 파일로 구성됩니다. 이러한 파일 중 하나는 기본 모듈 인터페이스 단위라고 합니다. 모듈이 내보내는 항목을 정의합니다. 즉, 모듈의 가져오기에 표시되는 내용입니다. 모듈당 하나의 기본 모듈 인터페이스 단위만 있을 수 있습니다.

기본 모듈 인터페이스 단위를 추가하려면 솔루션 탐색기 원본 파일을 마우스 오른쪽 단추로 클릭한 다음 **모듈추가**>를 선택합니다.



표시되는 새 항목 추가 대화 상자에서 새 모듈에 이름을 `BasicPlane.Figures.ixx` 지정하고 추가를 선택합니다.

만든 모듈 파일의 기본 콘텐츠에는 다음 두 줄이 있습니다.

```
C++  
  
export module BasicPlane;  
  
export void MyFunc();
```

첫 번째 줄의 키워드는 `export module` 이 파일이 모듈 인터페이스 단위임을 선언합니다. 여기에는 미묘한 점이 있습니다. 명명된 모든 모듈에 대해 모듈 파티션이 지정되지 않은 모듈 인터페이스 단위가 정확히 하나 있어야 합니다. 해당 모듈 단위를 **기본 모듈 인터페이스 단위**라고 합니다.

기본 모듈 인터페이스 단위는 소스 파일이 모듈을 가져올 때 노출할 함수, 형식, 템플릿, 기타 모듈 및 모듈 파티션을 선언하는 위치입니다. 모듈은 여러 파일로 구성 될 수 있지만 기본 모듈 인터페이스 파일만 노출할 내용을 식별합니다.

파일의 `BasicPlane.Figures.ixx` 내용을 다음으로 바꿉니다.

```
C++  
  
export module BasicPlane;
```

```
export module BasicPlane.Figures; // the export module keywords mark this file as a primary module interface unit
```

이 줄은 이 파일을 기본 모듈 인터페이스로 식별하고 모듈에 이름을 `BasicPlane.Figures` 지정합니다. 모듈 이름의 마침표는 컴파일러에 특별한 의미가 없습니다. 마침표는 모듈의 구성 방식을 전달하는 데 사용할 수 있습니다. 함께 작동하는 모듈 파일이 여러 개 있는 경우 기간을 사용하여 문제 분리를 나타낼 수 있습니다. 이 자습서에서는 기간을 사용하여 API의 다양한 기능 영역을 나타냅니다.

이 이름은 "명명된 모듈"의 "명명된"의 출처이기도 합니다. 이 모듈의 일부인 파일은 이 이름을 사용하여 명명된 모듈의 일부로 자신을 식별합니다. 명명된 모듈은 동일한 모듈 이름을 가진 모듈 단위의 컬렉션입니다.

더 나아가기 전에 잠시 구현할 API에 대해 이야기해야 합니다. 그것은 우리가 다음에 내리는 선택에 영향을 미칩니다. API는 서로 다른 셰이프를 나타냅니다. 이 예제 `Point`에서는 및 `Rectangle`의 몇 가지 셰이프만 제공하려고 합니다. `Point` 는 와 같은 `Rectangle` 더 복잡한 셰이프의 일부로 사용됩니다.

모듈의 몇 가지 기능을 설명하기 위해 이 API를 조각으로 고려합니다. 한 조각은 API가 `Point` 됩니다. 다른 부분은 입니다 `Rectangle`. 이 API가 더 복잡한 것으로 확장될 것이라고 상상해 보세요. 나누기는 문제를 분리하거나 코드 유지 관리를 완화하는 데 유용합니다.

지금까지 이 API를 노출하는 기본 모듈 인터페이스를 만들었습니다. 이제 API를 빌드해 `Point` 보겠습니다. 이 모듈의 일부가 되기를 바랍니다. 논리적 조직의 이유와 잠재적인 빌드 효율성을 위해 API의 이 부분을 자체적으로 쉽게 이해할 수 있도록 하고자 합니다. 이렇게 하려면 **모듈 파티션 파일**을 만듭니다.

모듈 파티션 파일은 모듈의 조각 또는 구성 요소입니다. 고유한 이유는 모듈의 개별 조각으로 처리할 수 있지만 모듈 내에서만 처리할 수 있다는 것입니다. 모듈 파티션은 모듈 외부에서 사용할 수 없습니다. 모듈 파티션은 모듈 구현을 관리 가능한 부분으로 나누는 데 유용합니다.

파티션을 주 모듈로 가져오면 내보내는지 여부에 관계없이 모든 선언이 주 모듈에 표시됩니다. 파티션을 명명된 모듈에 속하는 파티션 인터페이스, 기본 모듈 인터페이스 또는 모듈 단위로 가져올 수 있습니다.

모듈 파티션 파일 만들기

`Point` 모듈 파티션

모듈 파티션 파일을 만들려면 솔루션 탐색기소스 파일을 마우스 오른쪽 단추로 클릭한 다음 모듈추가>를 선택합니다. 파일 `BasicPlane.Figures-Point.ixx` 이름을 지정하고 추가를 선택합니다.

모듈 파티션 파일이므로 하이픈과 파티션 이름을 모듈 이름에 추가했습니다. 이 규칙은 컴파일러가 모듈 이름에 따라 이름 조회 규칙을 사용하여 파티션에 대해 컴파일된 `.ifc` 파일을 찾기 때문에 명령줄 사례에서 컴파일러를 지원합니다. 이렇게 하면 모듈에 속하는 파티션을 찾기 위해 명시적 `/reference` 명령줄 인수를 제공할 필요가 없습니다. 모듈에 속하는 파일을 쉽게 확인할 수 있으므로 모듈에 속한 파일을 이름으로 구성하는 데도 유용합니다.

`BasicPlane.Figures-Point.ixx`의 내용을 다음으로 바꿉니다.

C++

```
export module BasicPlane.Figures:Point; // defines a module partition,
Point, that's part of the module BasicPlane.Figures

export struct Point
{
    int x, y;
};
```

파일이 `export module`로 시작합니다. 이러한 키워드는 기본 모듈 인터페이스가 시작되는 방식이기도 합니다. 이 파일을 다르게 만드는 것은 모듈 이름 뒤에 파티션 이름이 오는 콜론(:)입니다. 이 명명 규칙은 파일을 모듈 파티션으로 식별합니다. 파티션에 대한 모듈 인터페이스를 정의하므로 기본 모듈 인터페이스로 간주되지 않습니다.

이름은 `BasicPlane.Figures:Point` 이 파티션을 모듈 `BasicPlane.Figures`의 일부로 식별합니다. (이름에 있는 마침표는 컴파일러에 특별한 의미가 없습니다.) 콜론은 이 파일에 모듈에 속하는라는 `Point` 모듈 `BasicPlane.Figures` 파티션이 포함되어 있음을 나타냅니다. 이 파티션을 이 명명된 모듈의 일부인 다른 파일로 가져올 수 있습니다.

이 파일에서 키워드는 `export struct Point` 소비자에게 표시됩니다.

Rectangle 모듈 파티션

정의할 다음 파티션은 입니다 `Rectangle`. 이전과 동일한 단계를 사용하여 다른 모듈 파일을 만듭니다. 솔루션 탐색기원본 파일을 마우스 오른쪽 단추로 클릭한 다음 모듈추가>를 선택합니다. 파일 이름을 `BasicPlane.Figures-Rectangle.ixx`으로 지정하고 추가를 선택합니다.

`BasicPlane.Figures-Rectangle.ixx`의 내용을 다음으로 바꿉니다.

C++

```
export module BasicPlane.Figures:Rectangle; // defines the module partition
Rectangle

import :Point;

export struct Rectangle // make this struct visible to importers
{
    Point ul, lr;
};

// These functions are declared, but will
// be defined in a module implementation file
export int area(const Rectangle& r);
export int height(const Rectangle& r);
export int width(const Rectangle& r);
```

파일은 모듈의 `export module BasicPlane.Figures:Rectangle;` 일부인 모듈 파티션을 선언하는 로 `BasicPlane.Figures` 시작됩니다. 모듈 이름에 추가된 는 `:Rectangle` 모듈 `BasicPlane.Figures`의 파티션으로 정의합니다. 이 명명된 모듈의 일부인 모듈 파일로 개별적으로 가져올 수 있습니다.

다음으로, `import :Point;` 모듈 파티션을 가져오는 방법을 보여줍니다. 문은 `import` 모듈 파티션에서 내보낸 모든 형식, 함수 및 템플릿을 모듈에 표시합니다. 모듈 이름을 지정할 필요가 없습니다. 컴파일러는 파일 맨 위에 있는 로 인해 `export module BasicPlane.Figures:Rectangle;` 이 파일이 모듈에 속 `BasicPlane.Figures` 한다는 것을 알고 있습니다.

다음으로, 코드는 사각형의 `struct Rectangle` 다양한 속성을 반환하는 일부 함수에 대한 및 선언의 정의를 내보냅니다. 키워드는 `export` 모듈의 소비자에게 앞에 표시되는 내용을 표시할지 여부를 나타냅니다. 함수, `height` 및 `width` 함수를 `area` 모듈 외부에서 볼 수 있도록 하는 데 사용됩니다.

모듈 파티션의 모든 정의 및 선언은 키워드가 있는지 여부에 관계없이 가져오기 모듈 단위에 `export` 표시됩니다. 키워드는 `export` 기본 모듈 인터페이스에서 파티션을 내보낼 때 정의, 선언 또는 `typedef`가 모듈 외부에 표시되는지 여부를 제어합니다.

이름은 다음과 같은 여러 가지 방법으로 모듈 소비자에게 표시됩니다.

- 내보낼 각 형식, 함수 등에 키워드 `export` 를 배치합니다.
- 예를 들어 `export namespace N { ... }` 네임스페이스 앞에 배치 `export` 하면 중괄호 내에 정의된 모든 항목이 내보내집니다. 그러나 모듈의 다른 곳에서 를 `struct S` 정의하는 `namespace N { struct S {...}; }` 경우 모듈의 소비자가 사용할 수 없습니다.

이름이 같은 다른 네임스페이스가 있더라도 네임스페이스 선언 앞에는 이 네임스페이스 선언을 사용할 수 없습니다 `export`.

- 형식, 함수 등을 내보내지 않아야 하는 경우 키워드를 `export` 생략합니다. 모듈의 일부이지만 모듈의 가져오기에는 표시되지 않는 다른 파일에 표시됩니다.
- 를 사용하여 `module :private;` 프라이빗 모듈 파티션의 시작을 표시합니다. 프라이빗 모듈 파티션은 선언이 해당 파일에만 표시되는 모듈의 섹션입니다. 이 모듈을 가져오는 파일이나 이 모듈의 일부인 다른 파일에는 표시되지 않습니다. 파일에 대한 정적 로컬 섹션으로 간주합니다. 이 섹션은 파일 내에서만 볼 수 있습니다.
- 가져온 모듈 또는 모듈 파티션을 표시하려면 를 사용합니다 `export import`. 예제는 다음 섹션에 나와 있습니다.

모듈 파티션 작성

이제 API의 두 부분이 정의되었으므로 이 모듈을 가져오는 파일이 전체적으로 액세스할 수 있도록 API를 함께 만들어 보겠습니다.

모든 모듈 파티션은 자신이 속한 모듈 정의의 일부로 노출되어야 합니다. 파티션은 기본 모듈 인터페이스에 노출됩니다. `BasicPlane.Figures.ixx` 기본 모듈 인터페이스를 정의하는 파일을 엽니다. 해당 내용을 다음으로 대체합니다.

C++

```
export module BasicPlane.Figures; // keywords export module marks this as a
primary module interface unit

export import :Point; // bring in the Point partition, and export it to
consumers of this module
export import :Rectangle; // bring in the Rectangle partition, and export it
to consumers of this module
```

로 시작하는 `export import` 두 줄은 여기에 새로워집니다. 이와 같이 결합된 경우 이러한 두 키워드는 컴파일러에 지정된 모듈을 가져와서 이 모듈의 소비자에게 표시하도록 지시합니다. 이 경우 모듈 이름의 콜론(:)은 모듈 파티션을 가져오고 있음을 나타냅니다.

가져온 이름에는 전체 모듈 이름이 포함되지 않습니다. 예를 들어 파티션은 `:Point`로 `export module BasicPlane.Figures:Point` 선언되었습니다. 그러나 여기서는 를 가져오고 있습니다 `:Point`. 모듈의 기본 모듈 인터페이스 파일에 있으므로 모듈 `BasicPlane.Figures` 이름이 암시되고 파티션 이름만 지정됩니다.

지금까지 사용할 수 있도록 하려는 API 표면을 노출하는 기본 모듈 인터페이스를 정의했습니다. 그러나 정의 `area()` 되지 않은,, `height()` 또는 `width()` 만 선언했습니다. 다음으로 모듈 구현 파일을 만들어 이 작업을 수행합니다.

모듈 단위 구현 파일 만들기

모듈 단위 구현 파일은 확장으로 `.iexx` 끝나지 않습니다. 일반 `.cpp` 파일입니다. 원본 파일의 **솔루션 탐색기** 마우스 오른쪽 단추를 클릭하여 원본 파일을 만들어 모듈 단위 구현 파일을 추가하고 **새 항목 추가**를 선택한 다음, **C++ 파일(.cpp)**을 선택합니다. 새 파일에 이름을 `BasicPlane.Figures-Rectangle.cpp` 지정한 다음, **추가**를 선택합니다.

모듈 파티션의 구현 파일에 대한 명명 규칙은 파티션에 대한 명명 규칙을 따릅니다. `.cpp` 그러나 구현 파일이므로 확장명도 있습니다.

파일의 `BasicPlane.Figures-Rectangle.cpp` 내용을 다음으로 바꿉니다.

```
C++  
  
module;  
  
// global module fragment area. Put #include directives here  
  
module BasicPlane.Figures:Rectangle;  
  
int area(const Rectangle& r) { return width(r) * height(r); }  
int height(const Rectangle& r) { return r.ul.y - r.lr.y; }  
int width(const Rectangle& r) { return r.lr.x - r.ul.x; }
```

이 파일은 `module;` 전역 모듈 조각이라는 모듈의 특수 영역을 도입하는 것으로 시작됩니다. 명명된 모듈에 대한 코드 앞에 있으며 와 같은 `#include` 전처리기 지시문을 사용할 수 있습니다. 전역 모듈 조각의 코드는 모듈 인터페이스에서 소유하거나 내보내지 않습니다.

헤더 파일을 포함하는 경우 일반적으로 모듈의 내보낸 부분으로 처리되지 않도록 합니다. 일반적으로 모듈 인터페이스의 일부가 되어서는 안 되는 구현 세부 정보로 헤더 파일을 포함합니다. 이 작업을 수행하려는 고급 사례가 있을 수 있지만 일반적으로는 그렇지 않습니다. 전역 모듈 조각의 지시문에 대해 `#include` 별도의 메타데이터(`.ifc` 파일)가 생성되지 않습니다. 전역 모듈 조각은 와 같은 `windows.h` 헤더 파일을 포함하거나 Linux, `unistd.h`에 포함할 수 있는 좋은 위치를 제공합니다.

빌드 중인 모듈 구현 파일에는 구현의 일부로 라이브러리가 필요하지 않으므로 라이브러리가 포함되지 않습니다. 그러나 그렇게 한다면, 이 영역은 지시문이 `#include` 갈 곳입니다.

줄 `module BasicPlane.Figures:Rectangle;` 은 이 파일이 명명된 모듈 `BasicPlane.Figures`의 일부임을 나타냅니다. 컴파일러는 기본 모듈 인터페이스에서 노출하는 형식과 함수를 이 파일에 자동으로 가져옵니다. 모듈 구현 단위에는 모듈 선언의 `export` 키워드 앞에 키워드가 `module` 없습니다.

다음은 함수 `height()` 및 `width()`의 정의입니다 `area()`. 의 파티션 `BasicPlane.Figures-Rectangle.ixx`에서 `Rectangle` 선언되었습니다. 이 모듈의 기본 모듈 인터페이스가 `Point` 및 `Rectangle` 모듈 파티션을 가져왔기 때문에 이러한 형식은 모듈 단위 구현 파일에 표시됩니다. 모듈 구현 단위의 흥미로운 기능: 컴파일러는 해당 모듈 기본 인터페이스의 모든 항목을 파일에 자동으로 표시합니다. 아니요 `imports <module-name>` 는 필요하지 않습니다.

구현 단위 내에서 선언하는 모든 항목은 해당 모듈이 속한 모듈에만 표시됩니다.

모듈 가져오기

이제 정의한 모듈을 사용합니다. `ModulesTutorial.cpp` 파일을 엽니다. 프로젝트의 일부로 자동으로 만들어졌습니다. 현재 함수 `main()`를 포함합니다. 해당 내용을 다음으로 대체 합니다.

C++

```
#include <iostream>

import BasicPlane.Figures;

int main()
{
    Rectangle r{ {1,8}, {11,3} };

    std::cout << "area: " << area(r) << '\n';
    std::cout << "width: " << width(r) << '\n';

    return 0;
}
```

문 `import BasicPlane.Figures;` 은 모듈에서 내보낸 모든 함수와 형식을 `BasicPlane.Figures` 이 파일에 표시합니다. 지시 `#include` 문 앞이나 뒤에 올 수 있습니다.

그런 다음, 앱은 모듈의 형식과 함수를 사용하여 정의된 사각형의 영역과 너비를 출력합니다.

출력

```
area: 50
width: 10
```

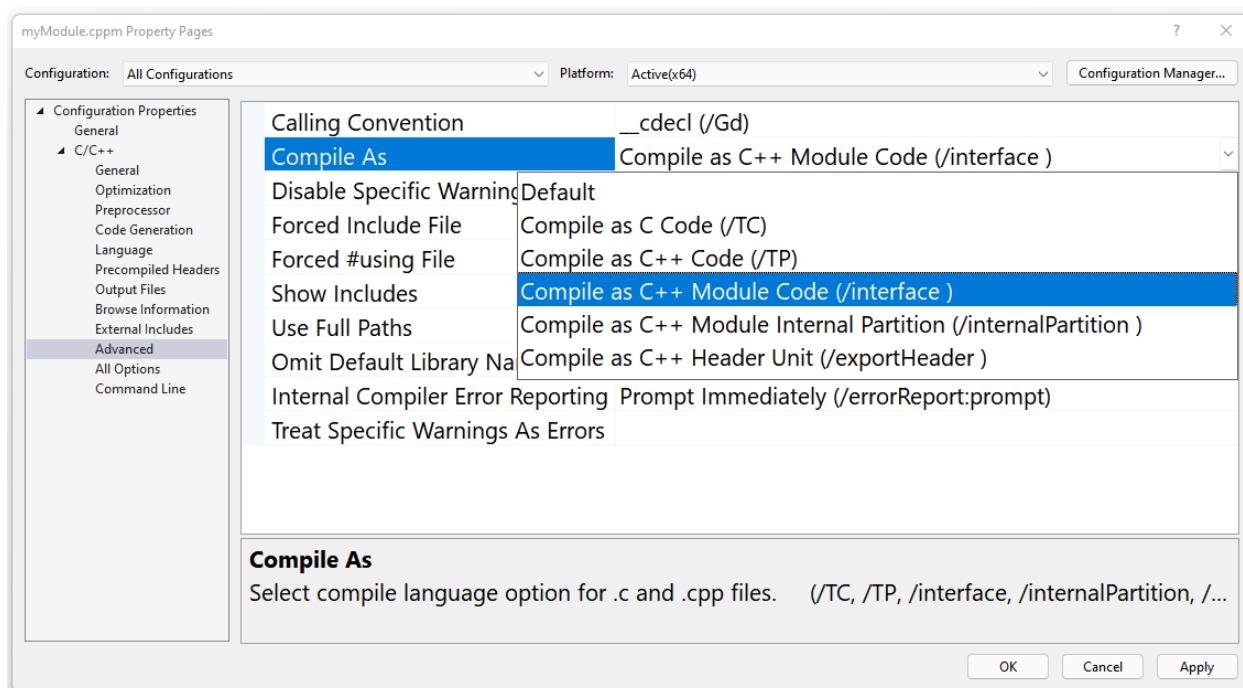
모듈 분석

이제 다양한 모듈 파일을 자세히 살펴보겠습니다.

기본 모듈 인터페이스

모듈은 하나 이상의 파일로 구성됩니다. 그 중 하나는 가져오기자가 볼 수 있는 인터페이스를 정의합니다. 이 파일에는 기본 모듈 인터페이스가 포함되어 있습니다. 모듈당 하나의 기본 모듈 인터페이스만 있을 수 있습니다. 앞에서 설명한 것처럼 내보낸 모듈 인터페이스 단위는 모듈 파티션을 지정하지 않습니다.

기본적으로 확장이 `.ixx` 있습니다. 그러나 모든 확장명에서 원본 파일을 모듈 인터페이스 파일로 처리할 수 있습니다. 이렇게 하려면 원본 파일의 속성 페이지에 대한 고급 탭의 **Compile As** 속성을 **모듈로 컴파일(/인터페이스)**로 설정합니다.



모듈 인터페이스 정의 파일의 기본 개요는 다음과 같습니다.

```
C++  
  
module; // optional. Defines the beginning of the global module fragment  
  
// #include directives go here but only apply to this file and  
// aren't shared with other module implementation files.  
// Macro definitions aren't visible outside this file, or to importers.  
// import statements aren't allowed here. They go in the module preamble,  
// below.  
  
export module [module-name]; // Required. Marks the beginning of the module  
preamble
```

```
// import statements go here. They're available to all files that belong to  
the named module  
// Put #includes in in the global module fragment, above  
  
// After any import statements, the module purview begins here  
// Put exported functions, types, and templates here  
  
module :private; // optional. The start of the private module partition.  
  
// Everything after this point is visible only within this file, and isn't  
// visible to any of the other files that belong to the named module.
```

이 파일은 전역 모듈 조각의 시작을 나타내거나 `export module [module-name];` 모듈 *purview*의 시작을 나타내기 위해 로 시작해야 `module;` 합니다.

모듈 *purview*는 함수, 형식, 템플릿 등을 모듈에서 노출하려는 위치입니다.

또한 파일에 표시된 것처럼 키워드를 통해 `export import` 다른 모듈 또는 모듈 파티션을 노출할 수 있습니다 `BasicPlane.Figures.ixx`.

기본 인터페이스 파일은 모듈에 대해 정의된 모든 인터페이스 파티션을 직접 또는 간접적으로 내보내거나 프로그램이 잘못된 형식이어야 합니다.

프라이빗 모듈 파티션은 이 파일에만 표시하려는 항목을 배치할 수 있는 위치입니다.

모듈 인터페이스 단위는 키워드 앞에 키워드 `module` 를 추가합니다 `export`.

모듈 구문에 대한 자세한 내용은 [모듈을 참조하세요](#).

모듈 구현 단위

모듈 구현 단위는 명명된 모듈에 속합니다. 해당 모듈이 속한 명명된 모듈은 파일의 `module [module-name]` 문으로 표시됩니다. 모듈 구현 단위는 코드 위생 또는 기타 이유로 기본 모듈 인터페이스 또는 모듈 파티션 파일에 배치하지 않으려는 구현 세부 정보를 제공합니다.

모듈 구현 단위는 큰 모듈을 더 작은 조각으로 분할하는 데 유용하므로 빌드 시간이 빨라질 수 있습니다. 이 기술은 [모범 사례](#) 섹션에서 간략하게 설명합니다.

모듈 구현 단위 파일에는 확장명이 있습니다 `.cpp`. 모듈 구현 단위 파일의 기본 개요는 다음과 같습니다.

C++

```
// optional #include or import statements. These only apply to this file  
// imports in the associated module's interface are automatically available  
to this file
```

```
module [module-name]; // required. Identifies which named module this
implementation unit belongs to

// implementation
```

모듈 파티션 파일

모듈 파티션은 모듈을 다른 조각 또는 *파티션*으로 구성 요소화하는 방법을 제공합니다. 모듈 파티션은 명명된 모듈의 일부인 파일에서만 가져오도록 되어 있습니다. 명명된 모듈 외부에서 가져올 수 없습니다.

파티션에는 인터페이스 파일과 구현 파일이 0개 이상 있습니다. 모듈 파티션은 전체 모듈에 있는 모든 선언의 소유권을 공유합니다.

파티션 인터페이스 파일로 내보낸 모든 이름은 기본 인터페이스 파일에서 가져오고 다시 내보내야 합니다(`export import`). 파티션의 이름은 모듈 이름, 물론, 파티션 이름으로 시작해야 합니다.

파티션 인터페이스 파일의 기본 개요는 다음과 같습니다.

C++

```
module; // optional. Defines the beginning of the global module fragment

// This is where #include directives go. They only apply to this file and
aren't shared
// with other module implementation files.
// Macro definitions aren't visible outside of this file or to importers
// import statements aren't allowed here. They go in the module preamble,
below

export module [Module-name]:[Partition name]; // Required. Marks the
beginning of the module preamble

// import statements go here.
// To access declarations in another partition, import the partition. Only
use the partition name, not the module name.
// For example, import :Point;
// #include directives don't go here. The recommended place is in the global
module fragment, above

// export imports statements go here

// after import, export import statements, the module purview begins
// put exported functions, types, and templates for the partition here

module :private; // optional. Everything after this point is visible only
within this file, and isn't
```

```
// visible to any of the other files that belong to  
the named module.
```

```
...
```

모듈 모범 사례

모듈 및 모듈을 가져오는 코드는 동일한 컴파일러 옵션으로 컴파일되어야 합니다.

모듈 이름 지정

- 모듈 이름에 마침표('.')를 사용할 수 있지만 컴파일러에는 특별한 의미가 없습니다. 이를 사용하여 모듈 사용자에게 의미를 전달합니다. 예를 들어 라이브러리 또는 프로젝트 상위 네임스페이스로 시작합니다. 모듈의 기능을 설명하는 이름으로 마칩니다. `BasicPlane.Figures` 는 기하학적 평면에 대한 API 및 평면에 나타낼 수 있는 특히 그림을 전달하기 위한 것입니다.
- 모듈 기본 인터페이스를 포함하는 파일의 이름은 일반적으로 모듈의 이름입니다. 예를 들어 모듈 이름 `BasicPlane.Figures` 을 지정하면 기본 인터페이스를 포함하는 파일의 이름은 로 지정 `BasicPlane.Figures.ixx` 됩니다.
- 모듈 파티션 파일의 이름은 일반적으로 `<primary-module-name>-<module-partition-name>` 모듈 이름 뒤에 하이픈('-')과 파티션 이름이 있는 위치입니다. 예를 들어 `BasicPlane.Figures-Rectangle.ixx`

명령줄에서 빌드하고 모듈 파티션에 이 명명 규칙을 사용하는 경우 각 모듈 파티션 파일에 대해 명시적으로 추가할 `/reference` 필요가 없습니다. 컴파일러는 모듈의 이름에 따라 자동으로 찾습니다. 컴파일된 파티션 파일의 이름(확장명으로 `.ifc` 종료)은 모듈 이름에서 생성됩니다. 모듈 이름을 `BasicPlane.Figures:Rectangle` 고려합니다. 컴파일러는 해당 컴파일된 해당 파티션 파일 `Rectangle` 의 이름이 `BasicPlane.Figures-Rectangle.ifc`임을 예상합니다. 컴파일러는 이 명명 체계를 사용하여 파티션에 대한 인터페이스 단위 파일을 자동으로 찾아 모듈 파티션을 더 쉽게 사용할 수 있도록 합니다.

고유한 규칙을 사용하여 이름을 지정할 수 있습니다. 하지만 명령줄 컴파일러에 해당하는 `/reference` 인수를 지정해야 합니다.

요소 모듈

모듈 구현 파일 및 파티션을 사용하여 더 쉬운 코드 유지 관리 및 잠재적으로 더 빠른 컴파일 시간을 위해 모듈을 고려합니다.

예를 들어 모듈의 구현을 모듈 인터페이스 정의 파일에서 모듈 구현 파일로 이동하면 구현을 변경해도 모듈을 가져오는 모든 파일이 반드시 다시 컴파일되는 것은 아닙니다(구

현이 없는 한 `inline`).

모듈 파티션을 사용하면 큰 모듈을 논리적으로 더 쉽게 인수할 수 있습니다. 구현의 일부를 변경해도 모든 모듈의 파일이 다시 컴파일되지 않도록 컴파일 시간을 개선하는 데 사용할 수 있습니다.

요약

이 자습서에서는 C++20 모듈의 기본 사항을 소개했습니다. 기본 모듈 인터페이스를 만들고, 모듈 파티션을 정의하고, 모듈 구현 파일을 빌드했습니다.

추가 정보

[C++에서의 모듈 개요](#)

[module, import, export 키워드](#)

[Visual Studio에서 C++ 모듈 둘러보기 ↗](#)

[실용적인 C++20 모듈 및 C++ 모듈 관련 도구의 미래 ↗](#)

[프로젝트 이름을 모듈로 C++로 이동 ↗](#)

[연습: Microsoft Visual C++에서 헤더 단위 빌드 및 가져오기](#)

템플릿 (C++)

아티클 • 2024. 07. 15.

템플릿은 C++에서 제네릭 프로그래밍의 기초입니다. 강력한 형식의 언어인 C++에서는 모든 변수에 프로그래머가 명시적으로 선언하거나 컴파일러에서 추론한 특정 형식이 있어야 합니다. 그러나 많은 데이터 구조와 알고리즘이 어떤 형식에서 작동하든 동일하게 보입니다. 템플릿을 사용하면 클래스 또는 함수의 작업을 정의하고, 그러한 작업이 어떤 구체적인 형식에서 작동해야 하는지를 사용자가 지정하도록 할 수 있습니다.

템플릿 정의 및 사용

템플릿은 사용자가 템플릿 매개 변수에 대해 제공하는 인수를 기반으로 컴파일 시간에 일반 형식 또는 함수를 생성하는 구문입니다. 예를 들어 다음과 같이 함수 템플릿을 정의할 수 있습니다.

```
C++

template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

위의 코드는 반환 값과 호출 매개 변수(lhs 및 rhs)가 모두 이 형식인 단일 형식 매개 변수 *T*가 있는 제네릭 함수에 대한 템플릿을 설명합니다. 원하는 형식 매개 변수의 이름을 지정할 수 있지만 관례상 단일 대문자가 가장 일반적으로 사용됩니다. *T*는 템플릿 매개 변수이며, `typename` 키워드는 이 매개 변수가 형식의 자리 표시자임을 나타냅니다. 함수가 호출되면 컴파일러는 *T*의 모든 인스턴스를 사용자가 지정하거나 컴파일러에서 추론하는 구체적인 형식 인수로 바꿉니다. 컴파일러가 템플릿에서 클래스 또는 함수를 생성하는 프로세스를 `템플릿 인스턴스화`라고 하며, `minimum<int>`는 템플릿 `minimum<T>`의 인스턴스화입니다.

다른 곳에서는 사용자가 `int`에 특수화된 템플릿의 인스턴스를 선언할 수 있습니다. `get_a()` 및 `get_b()`가 `int`를 반환하는 함수라고 가정합니다.

```
C++

int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

하지만 이는 함수 템플릿이고 컴파일러는 *a* 및 *b* 인수에서 *T* 형식을 추론할 수 있으므로 일반적인 함수처럼 호출할 수 있습니다.

C++

```
int i = minimum(a, b);
```

컴파일러가 마지막 문을 발견하면 템플릿의 모든 *T*가 `int`로 대체되는 새 함수가 생성됩니다.

C++

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

컴파일러가 함수 템플릿에서 형식 추론을 수행하는 방법에 대한 규칙은 일반 함수 규칙을 기반으로 합니다. 자세한 내용은 [함수 템플릿 호출의 오버로드 확인](#)을 참조하세요.

형식 매개 변수

위의 `minimum` 템플릿에서 *T* 형식 매개 변수는 `const` 및 참조 한정자를 추가하는 함수 호출 매개 변수에서 사용될 때까지 어떤 방식으로든 정규화되지 않습니다.

형식 매개 변수의 수에는 실질적인 제한이 없습니다. 여러 매개 변수를 쉼표로 구분합니다.

C++

```
template <typename T, typename U, typename V> class Foo{};
```

`class` 키워드는 이 컨텍스트에서 `typename`과 동일합니다. 이전 예를 다음과 같이 표현할 수 있습니다.

C++

```
template <class T, class U, class V> class Foo{};
```

줄임표 연산자(...)를 사용하여 형식 매개 변수의 0개 이상의 임의의 수를 사용하는 템플릿을 정의할 수 있습니다.

C++

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

모든 기본 제공 형식 또는 사용자 정의 형식을 형식 인수로 사용할 수 있습니다. 예를 들어 표준 라이브러리의 `std::vector`를 사용하여 `int`, `double`, `std::string`, `MyClass`, `const MyClass*`, `MyClass&` 등의 변수를 저장할 수 있습니다. 템플릿을 사용할 때의 주요 제한 사항은 형식 인수가 형식 매개 변수에 적용되는 모든 작업을 지원해야 한다는 것입니다. 예를 들어 이 예와 같이 `MyClass`를 사용하여 `minimum`을 호출하는 경우:

C++

```
class MyClass
{
public:
    int num;
    std::wstring description;
};

int main()
{
    MyClass mc1 {1, L"hello"};
    MyClass mc2 {2, L"goodbye"};
    auto result = minimum(mc1, mc2); // Error! C2678
}
```

`MyClass`가 < 연산자에 대한 오버로드를 제공하지 않으므로 컴파일러 오류가 생성됩니다.

특정 템플릿의 형식 인수가 모두 동일한 개체 계층 구조에 속해야 한다는 내재된 요구 사항은 없습니다. 하지만 이러한 제한을 적용하는 템플릿을 정의할 수는 있습니다. 개체 지향 기술을 템플릿과 결합할 수 있습니다. 예를 들어 `Derived*`를 벡터 `<Base*>`에 저장할 수 있습니다. 인수는 포인터여야 함

C++

```
vector<MyClass*> vec;
MyDerived d(3, L"back again", time(0));
vec.push_back(&d);

// or more realistically:
vector<shared_ptr<MyClass>> vec2;
vec2.push_back(make_shared<MyDerived>());
```

`std::vector` 및 기타 표준 라이브러리 컨테이너가 `T` 요소에 적용하는 기본 요구 사항은 `T`가 복사 할당과 복사 생성이 가능해야 한다는 것입니다.

비형식 매개 변수

C# 및 Java와 같은 다른 언어의 제네릭 형식과 달리 C++ 템플릿은 값 매개 변수라고도 하는 비형식 매개 변수를 지원합니다. 예를 들어 이 예의 표준 라이브러리의 `std::array` 클래스와 유사하게 배열의 길이를 지정하는 상수 적분 값을 제공할 수 있습니다.

```
C++

template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

템플릿 선언의 구문을 참고하세요. `size_t` 같은 컴파일 시간에 템플릿 인수로 전달되며 `const` 또는 `constexpr` 식이어야 합니다. 다음과 같이 사용합니다.

```
C++

MyArray<MyClass*, 10> arr;
```

포인터 및 참조를 포함한 다른 종류의 같은 비형식 매개 변수로 전달할 수 있습니다. 예를 들어 함수 또는 함수 개체에 대한 포인터를 전달하여 템플릿 코드 내에서 일부 작업을 사용자 지정할 수 있습니다.

비형식 템플릿 매개 변수에 대한 형식 추론

Visual Studio 2017 이상 및 `/std:c++17` 모드 이상에서 컴파일러는 `auto`로 선언된 비형식 템플릿 인수의 형식을 추론합니다.

```
C++

template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;           // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;         // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;          // v3 == 'a', decltype(v3) is char
```

템플릿 매개 변수로서의 템플릿

템플릿은 템플릿 매개 변수가 될 수 있습니다. 이 예에서 MyClass2에는 typename 매개 변수 T와 템플릿 매개 변수 Arr이라는 두 개의 템플릿 매개 변수가 있습니다.

C++

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

Arr 매개 변수 자체에는 본문이 없으므로 매개 변수 이름이 필요하지 않습니다. 실제로 MyClass2 본문 내에서 Arr의 typename 또는 클래스 매개 변수 이름을 참조하는 것은 오류입니다. 따라서 이 예와 같이 Arr의 형식 매개 변수 이름을 생략할 수 있습니다.

C++

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

기본 템플릿 인수

클래스 및 함수 템플릿은 기본 인수를 가질 수 있습니다. 템플릿에 기본 인수가 있는 경우 이를 사용할 때 지정하지 않은 채로 둘 수 있습니다. 예를 들어 std::vector 템플릿에 할당자에 대한 기본 인수가 있습니다.

C++

```
template <class T, class Allocator = allocator<T>> class vector;
```

대부분의 경우 기본 std::allocator 클래스가 허용되므로 다음과 같은 벡터를 사용합니다.

C++

```
vector<int> myInts;
```

하지만 필요한 경우 다음처럼 사용자 지정 할당자를 지정할 수 있습니다.

C++

```
vector<int, MyAllocator> ints;
```

템플릿 인수가 여러 개인 경우 첫 번째 기본 인수 다음의 모든 인수에는 기본 인수가 있어야 합니다.

매개 변수가 모두 기본값인 템플릿을 사용하는 경우 빈 꺠쇠괄호를 사용합니다.

C++

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};

...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

템플릿 특수화

어떤 경우에는 템플릿이 모든 형식에 대해 정확히 동일한 코드를 정의하는 것은 불가능하거나 바람직하지 않습니다. 예를 들어 형식 인수가 포인터, std::wstring 또는 특정 기본 클래스에서 파생된 형식인 경우에만 실행되도록 코드 경로를 정의하려 할 수 있습니다. 이러한 경우 해당 특정 형식에 대한 템플릿의 **특수화**를 정의할 수 있습니다. 사용자가 해당 형식으로 템플릿을 인스턴스화하면 컴파일러는 특수화를 사용하여 클래스를 생성하고 다른 모든 형식에 대해서는 일반적인 템플릿을 선택합니다. 모든 매개 변수가 특수화되는 특수화는 **전체 특수화**입니다. 일부 매개 변수만 특수화하면 **부분 특수화**라고 부릅니다.

C++

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
```

```
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

특수화된 각 형식 매개 변수가 고유한 한 템플릿은 여러 특수화를 가질 수 있습니다. 클래스 템플릿만 부분적으로 특수화될 수 있습니다. 템플릿의 모든 전체 특수화와 부분 특수화는 원래 템플릿과 동일한 네임스페이스에서 선언되어야 합니다.

자세한 내용은 [템플릿 특수화](#)를 참조하세요.

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

typename

아티클 • 2024. 07. 15.

템플릿 정의에서 `typename`은 알 수 없는 식별자가 형식이라는 힌트를 컴파일러에 제공합니다. 템플릿 매개 변수 목록에서 형식 매개 변수를 지정하는 데 사용됩니다.

구문

```
typename identifier ;
```

설명

템플릿 정의의 이름이 템플릿 인수에 종속된 정규화된 이름인 경우 `typename` 키워드를 사용해야 합니다. 정규화된 이름이 종속되지 않는 경우에는 선택적으로 사용합니다. 자세한 내용은 [템플릿 및 이름 확인](#)을 참조하세요.

`typename`은 템플릿 선언 또는 정의의 모든 위치에서 모든 형식에 의해 사용될 수 있습니다. 템플릿 기본 클래스의 템플릿 인수로 사용되지 않는 한 기본 클래스 목록에서는 허용되지 않습니다.

C++

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{}
```

`typename` 키워드는 템플릿 매개 변수 목록의 `class` 대신 사용할 수도 있습니다. 예를 들어 다음 문은 의미상 동일합니다.

C++

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

예시

C++

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y; // treat Y as a type
};

int main()
{
}
```

참고 항목

[템플릿](#)
[키워드](#)

피드백

이 페이지가 도움이 되었나요? [!\[\]\(05533de71cf71e0fdd9e45894dd99158_img.jpg\) Yes](#) [!\[\]\(ff15cdbb535ef4ef5286665f0dec3e2a_img.jpg\) No](#)

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

클래스 템플릿

아티클 • 2023. 10. 12.

이 문서에서는 C++ 클래스 템플릿과 관련된 규칙을 설명합니다.

클래스 템플릿의 멤버 함수

멤버 함수는 클래스 템플릿의 내부 또는 외부에서 정의할 수 있습니다. 클래스 템플릿 외부에서 정의된 경우 함수 템플릿처럼 정의됩니다.

C++

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{

};

template< class T, int i > void MyStack< T, i >::push( const T item )
{

};

template< class T, int i > T& MyStack< T, i >::pop( void )
{

};

int main()
{
}
```

템플릿 클래스 멤버 함수와 마찬가지로 클래스의 생성자 멤버 함수 정의에는 템플릿 인수 목록이 두 번 포함됩니다.

멤버 함수 자체는 함수 템플릿일 수 있으며 다음 예제와 같이 추가 매개 변수를 지정할 수 있습니다.

C++

```

// member_templates.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{

}

int main()
{
}

```

중첩 클래스 템플릿

템플릿은 클래스 또는 클래스 템플릿 내에서 정의할 수 있으며, 이 경우 멤버 템플릿이라고 합니다. 클래스인 멤버 템플릿은 중첩된 클래스 템플릿이라고 합니다. 함수인 멤버 템플릿은 멤버 함수 템플릿에서 [설명합니다](#).

중첩된 클래스 템플릿은 바깥쪽 클래스의 범위 안에 클래스 템플릿으로 선언되며 바깥쪽 클래스 안이나 밖에 정의할 수 있습니다.

다음 코드에서는 일반 클래스 안에 중첩된 클래스 템플릿을 보여 줍니다.

C++

```

// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{

    template <class T>
    struct Y
    {
        T m_t;
        Y(T t): m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;
}
```

```

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()
{
    X x(1, 'a');
    x.print();
}

```

다음 코드는 중첩된 템플릿 형식 매개 변수를 사용하여 중첩 클래스 템플릿을 만듭니다.

C++

```

// nested_class_template2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
    template <class U> class Y
    {
        U* u;
        public:
            Y();
            U& Value();
            void print();
            ~Y();
    };

    Y<int> y;
    public:
        X(T t) { y.Value() = t; }
        void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
    cout << "X<T>::Y<U>::Y()" << endl;
    u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()

```

```

{
    return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
    cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
    cout << "X<T>::Y<U>::~Y()" << endl;
    delete u;
}

int main()
{
    X<int>* xi = new X<int>(10);
    X<char>* xc = new X<char>('c');
    xi->print();
    xc->print();
    delete xi;
    delete xc;
}

/* Output:
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()
*/

```

로컬 클래스는 멤버 템플릿을 가질 수 없습니다.

템플릿 친구

수업 템플릿에는 친구가 [있을](#) 수 있습니다. 클래스 또는 클래스 템플릿, 함수 또는 함수 템플릿은 템플릿 클래스에 대한 friend일 수 있습니다. friends는 클래스 템플릿 또는 함수 템플릿의 특수화가 될 수도 있지만 부분 특수화는 될 수 없습니다.

다음 예제에서 friend 함수는 클래스 템플릿 내에서 함수 템플릿으로 정의됩니다. 이 코드는 모든 템플릿 인스턴스화에 대한 friend 함수 버전을 만듭니다. 이 구문은 사용자의 friend 함수가 클래스와 같은 템플릿 매개 변수로 결정될 경우 유용합니다.

C++

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
    T* array;
    int size;

public:
    Array(int sz): size(sz) {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }

    Array(const Array& a) {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }

    T& operator[](int i) {
        return *(array + i);
    }

    int Length() { return size; }

    void print() {
        for (int i = 0; i < size; i++)
            cout << *(array + i) << " ";

        cout << endl;
    }

    template<class T>
    friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
    Array<T>* a = new Array<T>(a1.size + a2.size);
    for (int i = 0; i < a1.size; i++)
        (*a)[i] = *(a1.array + i);

    for (int i = 0; i < a2.size; i++)
        (*a)[i + a1.size] = *(a2.array + i);

    return a;
}

int main() {
```

```

Array<char> alpha1(26);
for (int i = 0 ; i < alpha1.Length() ; i++)
    alpha1[i] = 'A' + i;

alpha1.print();

Array<char> alpha2(26);
for (int i = 0 ; i < alpha2.Length() ; i++)
    alpha2[i] = 'a' + i;

alpha2.print();
Array<char>*alpha3 = combine(alpha1, alpha2);
alpha3->print();
delete alpha3;
}

/* Output:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l
m n o p q r s t u v w x y z
*/

```

다음 예제에서는 템플릿 특수화를 가진 friend를 다룹니다. 원본 함수 템플릿이 friend인 경우 함수 템플릿 특수화는 자동으로 friend입니다.

다음 코드의 friend 선언 앞의 주석에서 나타낸 것처럼 템플릿의 특수 버전만 친구로 선언 할 수도 있습니다. 특수화를 친구로 선언하는 경우 friend 템플릿 특수화의 정의를 템플릿 클래스 외부에 배치해야 합니다.

C++

```

// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
}
```

```

    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {
        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}

```

```
/* Output:  
10 generic  
10 int  
*/
```

다음 예제에서는 클래스 템플릿 내에 선언된 friend 클래스 템플릿을 보여 줍니다. 클래스 템플릿은 friend 클래스의 템플릿 인수로 사용됩니다. Friend 클래스 템플릿은 선언된 클래스 템플릿 외부에서 정의되어야 합니다. 또한 friend 템플릿의 모든 특수화 또는 부분 특수화는 원본 클래스 템플릿의 friend입니다.

C++

```
// template_friend3.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
template <class T>  
class X  
{  
private:  
    T* data;  
    void InitData(int seed) { data = new T(seed); }  
public:  
    void print() { cout << *data << endl; }  
    template <class U> friend class Factory;  
};  
  
template <class U>  
class Factory  
{  
public:  
    U* GetNewObject(int seed)  
    {  
        U* pu = new U;  
        pu->InitData(seed);  
        return pu;  
    }  
};  
  
int main()  
{  
    Factory< X<int> > XintFactory;  
    X<int>* x1 = XintFactory.GetNewObject(65);  
    X<int>* x2 = XintFactory.GetNewObject(97);  
  
    Factory< X<char> > XcharFactory;  
    X<char>* x3 = XcharFactory.GetNewObject(65);  
    X<char>* x4 = XcharFactory.GetNewObject(97);  
    x1->print();  
    x2->print();  
    x3->print();
```

```
    x4->print();
}
/* Output:
65
97
A
a
*/
```

템플릿 매개 변수 다시 사용

템플릿 매개 변수는 템플릿 매개 변수 목록에서 다시 사용할 수 있습니다. 예를 들어 다음 코드는 허용됩니다.

C++

```
// template_specifications2.cpp

class Y
{
};

template<class T, T* pT> class X1
{
};

template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{}
```

참고 항목

[템플릿](#)

함수 템플릿

아티클 • 2023. 10. 12.

클래스 템플릿은 인스턴스화할 때 클래스에 전달되는 형식 인수를 기반으로 하는 관련 클래스 패밀리를 정의합니다. 함수 템플릿은 클래스 템플릿과 유사하지만 함수 패밀리를 정의합니다. 함수 템플릿을 사용하면 동일한 코드를 기반으로 하지만 서로 다른 형식이나 클래스에서 작동하는 함수 집합을 지정할 수 있습니다. 다음 함수 템플릿은 두 항목을 바꿉니다.

C++

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() {
```

이 코드는 인수 값을 바꾸는 함수 패밀리를 정의합니다. 이 템플릿에서 교환 `int` 및 형식 및 `long` 사용자 정의 형식을 생성하는 함수를 생성할 수 있습니다. 클래스의 복사 생성자와 대입 연산자가 제대로 정의된 경우 `MySwap`으로 클래스도 바꿀 수 있습니다.

또한 컴파일러가 컴파일 시간에 `a` 및 `b` 매개 변수의 형식을 알고 있으므로 함수 템플릿은 서로 다른 형식의 개체를 교환하지 못하게 합니다.

이 함수는 `void` 포인터를 사용하여 템플릿이 아닌 함수로 실행할 수 있지만 템플릿 버전은 형식 안정적입니다. 다음 호출을 참조하십시오.

C++

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );       //error
```

컴파일러가 형식이 다른 매개 변수를 사용하여 `MySwap` 함수를 생성할 수 없으므로 두 번째 `MySwap` 호출은 컴파일 시간 오류를 트리거합니다. `void` 포인터가 사용된 경우 두 함수 호출 모두 올바르게 컴파일되지만 런타임에는 함수가 제대로 작동하지 않습니다.

함수 템플릿에 대한 템플릿 인수는 명시적으로 지정할 수 있습니다. 예시:

C++

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
    f<char>(j);    // Generate the specialization f(char).
    // If not explicitly specified, f(int) would be deduced.
}
```

템플릿 인수가 명시적으로 지정되면 함수 인수를 해당 함수 템플릿 매개 변수의 형식으로 변환하기 위해 표준 암시적 변환이 수행됩니다. 위의 예제에서 컴파일러는 형식 `char`으로 변환 `j` 됩니다.

참고 항목

[템플릿](#)

[함수 템플릿 인스턴스화](#)

[명시적 인스턴스화](#)

[함수 템플릿의 명시적 특수화](#)

함수 템플릿 인스턴스화

아티클 • 2023. 10. 12.

함수 템플릿이 각 형식에 대해 처음 호출될 때 컴파일러는 인스턴스를 만듭니다(인스턴스화). 각 인스턴스는 형식에 대해 특수화된 템플릿 함수의 버전입니다. 이 인스턴스는 함수가 형식에 사용될 때마다 호출됩니다. 동일한 인스턴스가 여러 개 있는 경우 서로 다른 모듈에 있더라도 인스턴스의 복사본이 하나만 실행 파일에 존재하게 됩니다.

함수 인수의 변환은 매개 변수가 템플릿 인수에 종속되지 않은 인수 및 매개 변수 쌍에 대해 함수 템플릿에서 허용됩니다.

함수 템플릿은 특정 형식을 인수로 사용하는 템플릿을 선언하여 명시적으로 인스턴스화 할 수 있습니다. 예를 들어 다음 코드는 허용됩니다.

C++

```
// function_template_instantiation.cpp
template<class T> void f(T) { }

// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);

// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
}
```

참고 항목

[함수 템플릿](#)

명시적 인스턴스화

아티클 • 2024. 11. 21.

명시적 인스턴스화를 통해 코드에서 실제로 사용하지 않고 템플릿 기반 클래스 또는 함수의 인스턴스를 만들 수 있습니다. 배포에 템플릿을 사용하는 라이브러리(`.lib`) 파일을 만들 때 유용하기 때문에 입증되지 않은 템플릿 정의는 개체(`.obj`) 파일에 포함되지 않습니다.

예제

이 코드는 변수 및 6개 항목에 대해 `int` 명시적으로 인스턴스화 `MyStack` 합니다.

C++

```
template class MyStack<int, 6>;
```

이 명령문은 개체에 대한 스토리지 없이 `MyStack`의 인스턴스를 만듭니다. 모든 멤버에 대한 코드가 생성됩니다.

다음 줄은 생성자 멤버 함수만 명시적으로 인스턴스화합니다.

C++

```
template MyStack<int, 6>::MyStack( void );
```

함수 템플릿 인스턴스화의 예제 [와 같이](#) 특정 형식 인수를 사용하여 함수 템플릿을 명시적으로 인스턴스화하여 다시 뮤을 수 있습니다.

키워드를 `extern` 사용하여 멤버의 자동 인스턴스화를 방지할 수 있습니다. 예시:

C++

```
extern template class MyStack<int, 6>;
```

마찬가지로 특정 멤버를 외부 및 인스턴스화되지 않음으로 표시할 수 있습니다.

C++

```
extern template MyStack<int, 6>::MyStack( void );
```

키워드를 `extern` 사용하여 둘 이상의 개체 모듈에서 컴파일러가 동일한 인스턴스화 코드를 생성하지 못하게 할 수 있습니다. 함수가 호출되는 경우 하나 이상의 연결된 모듈에서 지정된 명시적 템플릿 매개 변수를 사용하여 함수 템플릿을 인스턴스화해야 합니다. 그렇지 않으면 프로그램을 빌드할 때 링커 오류가 발생합니다.

① 참고

특수화의 키워드는 `extern` 클래스 본문 외부에 정의된 멤버 함수에만 적용됩니다. 클래스 선언 내에 정의된 함수는 인라인 함수로 간주되며 항상 인스턴스화됩니다.

참고 항목

[함수 템플릿](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

함수 템플릿의 명시적 특수화

아티클 • 2023. 10. 12.

함수 템플릿을 사용하면 특정 형식에 대한 함수 템플릿의 명시적 특수화(재정의)를 제공하여 해당 형식에 대한 특별한 동작을 정의할 수 있습니다. 예시:

C++

```
template<> void MySwap(double a, double b);
```

이 선언을 사용하면 변수에 대해 `double` 다른 함수를 정의할 수 있습니다. 템플릿이 아닌 함수와 마찬가지로 표준 형식 변환(예: 형식 `float double` 변수 승격)이 적용됩니다.

예시

C++

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
}

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
}
int main()
{
}
```

참고 항목

[함수 템플릿](#)

함수 템플릿의 부분 순서 지정(C++)

아티클 • 2023. 10. 12.

함수 호출의 인수 목록과 일치하는 다양한 함수 템플릿을 사용할 수 있습니다. C++에서 는 호출해야 하는 함수를 지정하는 함수 템플릿의 부분 정렬을 정의합니다. 정렬이 부분 적인 이유는 동일하게 특수화된 것으로 간주되는 일부 템플릿이 있을 수 있기 때문입니다.

컴파일러는 가능한 일치 항목에서 사용할 수 있는 가장 특수한 함수 템플릿을 선택합니다. 예를 들어 함수 템플릿이 형식 T 을 사용하고 다른 함수 템플릿을 T^* 사용할 수 있는 T^* 경우 버전이 더 특수화되었다고 합니다. 인수가 포인터 형식일 때마다 제네릭 T 버전 보다 선호되지만 둘 다 일치를 허용할 수 있습니다.

더욱 특수화된 함수 템플릿 후보를 확인하려면 다음 프로세스를 사용합니다.

1. 두 함수 템플릿을 T_2 고려합니다. T_1
2. 매개 변수 T_1 를 가상의 고유 형식 x 으로 바꿉니다.
3. 매개 변수 목록을 사용하여 해당 매개 변수 목록에 T_1 유효한 템플릿인지 T_2 확인합니다. 암시적 변환을 무시합니다.
4. 동일한 프로세스를 T_1 반복하고 T_2 반대로 반복합니다.
5. 한 템플릿이 다른 템플릿에 대한 유효한 템플릿 인수 목록이지만 반대가 true가 아닌 경우 해당 템플릿은 다른 템플릿보다 덜 특수한 것으로 간주됩니다. 이 전 단계를 사용하여 두 템플릿이 서로에 대해 유효한 인수를 형성하는 경우 동일하게 특수화된 것으로 간주되며 이를 사용하려고 할 때 모호한 호출 결과가 발생합니다.
6. 다음의 규칙을 사용합니다.
 - a. 특정 형식으로 특수화된 템플릿은 제네릭 형식 인수를 사용한 템플릿보다 특수화됩니다.
 - b. 가상 형식 x^* 은 템플릿 인수에 대한 유효한 인수이지만 x 템플릿 인수에 유효한 인수 $T^* T$ 가 아니므로 템플릿을 사용하는 것은 하나의 용도로만 $T^* T$ 사용하는 것보다 더 특수합니다.
 - c. `const T` 는 템플릿 인수에 대한 유효한 인수이지만 x 템플릿 인수에 대한 T 유효한 인수 `const T` 가 아니기 때문에 `const x` 보다 특수합니다 T .

- d. `const T*`는 템플릿 인수에 대한 유효한 인수이지만 `x*` 템플릿 인수에 대한 `T*`은 유효한 인수 `const T*`가 아니기 때문에 `const X*` 보다 특수합니다 `T*`.

예시

다음 샘플은 표준에 지정된 대로 작동합니다.

C++

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>

template <class T> void f(T) {
    printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
    printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
    printf_s("Even more specialized function for const T*\n");
}

int main() {
    int i = 0;
    const int j = 0;
    int *pi = &i;
    const int *cpi = &j;

    f(i);    // Calls less specialized function.
    f(pi);   // Calls more specialized function.
    f(cpi); // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
}
```

출력

Output

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

참고 항목

함수 템플릿

멤버 함수 템플릿

아티클 • 2023. 10. 12.

멤버 템플릿이라는 용어는 멤버 함수 템플릿과 중첩된 클래스 템플릿을 둘 다 나타냅니다. 멤버 함수 템플릿은 클래스 또는 클래스 템플릿의 멤버인 함수 템플릿입니다.

멤버 함수는 여러 컨텍스트에서 함수 템플릿이 될 수 있습니다. 클래스 템플릿의 모든 함수는 제네릭이지만 멤버 템플릿 또는 멤버 함수 템플릿이라고 하는 것은 아닙니다. 이러한 멤버 함수가 자체 템플릿 인수를 사용하는 경우 멤버 함수 템플릿으로 간주됩니다.

예: 멤버 함수 템플릿 선언

템플릿이 아닌 클래스 또는 클래스 템플릿의 멤버 함수 템플릿은 해당 템플릿 매개 변수를 사용하여 함수 템플릿으로 선언됩니다.

C++

```
// member_function_templates.cpp
struct X
{
    template <class T> void mf(T* t) {}
};

int main()
{
    int i;
    X* x = new X();
    x->mf(&i);
}
```

예: 클래스 템플릿의 멤버 함수 템플릿

다음 예제에서는 클래스 템플릿의 멤버 함수 템플릿을 보여줍니다.

C++

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
```

```
};

int main()
{
}
```

예: 클래스 외부의 멤버 템플릿 정의

C++

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{}
```

예: 템플릿 기반 사용자 정의 변환

로컬 클래스는 멤버 템플릿을 가질 수 없습니다.

멤버 함수 템플릿은 가상 함수일 수 없습니다. 또한 기본 클래스 가상 함수와 동일한 이름으로 선언된 경우 기본 클래스에서 가상 함수를 재정의할 수 없습니다.

다음 예제에서는 템플릿 기반 사용자 정의 변환을 보여줍니다.

C++

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
    template <class U> operator S<U>()
    {
        return S<U>();
    }
};
```

```
int main()
{
    S<int> s1;
    S<long> s2 = s1; // Convert s1 using UDC and copy constructs S<long>.
}
```

참고 항목

[함수 템플릿](#)

템플릿 특수화(C++)

아티클 • 2023. 10. 12.

클래스 템플릿은 부분적으로 특수화될 수 있으며 결과 클래스는 여전히 템플릿입니다. 부분 특수화를 사용하면 다음과 같은 상황에서 특정 형식에 대해 템플릿 코드를 부분적으로 사용자 지정할 수 있습니다.

- 템플릿에 여러 형식이 있으며 그 중 일부만 특수화되어야 하는 경우. 결과는 나머지 형식에 대해 매개 변수화된 템플릿입니다.
- 템플릿에 형식이 하나만 있지만 포인터, 참조, 멤버에 대한 포인터 또는 함수 포인터 형식에 대한 특수화가 필요한 경우. 특수화 자체는 여전히 가리켜지거나 참조된 형식에 대한 템플릿입니다.

예: 클래스 템플릿의 부분 특수화

C++

```
// partial_specialization_of_class_templates.cpp
#include <stdio.h>

template <class T> struct PTS {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 0
    };
};

template <class T> struct PTS<T*> {
    enum {
        IsPointer = 1,
        IsPointerToDataMember = 0
    };
};

template <class T, class U> struct PTS<T U::*> {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 1
    };
};

struct S{};

int main() {
    printf_s("PTS<S>::IsPointer == %d \nPTS<S>::IsPointerToDataMember == %d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
```

```

    printf_s("PTS<S*>::IsPointer == %d \nPTS<S*>::IsPointerToDataMember == %d\n"
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d \nPTS"
            "<int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>:::
    IsPointerToDataMember);
}

```

Output

```

PTS<S>::IsPointer == 0
PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1
PTS<S*>::IsPointerToDataMember == 0
PTS<int S::*>::IsPointer == 0
PTS<int S::*>::IsPointerToDataMember == 1

```

예: 포인터 형식에 대한 부분 특수화

모든 형식을 사용하는 템플릿 컬렉션 클래스가 있는 경우 포인터 형식 `T T*` 을 사용하는 부분 특수화를 만들 수 있습니다. 다음 코드에서는 컬렉션 클래스 템플릿 `Bag` 와 컬렉션이 포인터 형식을 배열에 복사하기 전에 역참조하는 포인터 형식에 대한 부분 특수화를 보여 줍니다. 그런 다음 컬렉션은 가리켜진 값을 저장합니다. 원래 템플릿을 사용했다면 포인터 자체만 컬렉션에 저장되고 데이터는 삭제나 수정에 취약한 상태가 되었을 것입니다. 이 컬렉션의 특수 포인터 버전에서는 `add` 메서드에서 null 포인터를 검사하는 코드가 추가되었습니다.

C++

```

// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t) {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];

```

```

        for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
        tmp[size++] = t;
        delete[] elem;
        elem = tmp;
    }
    else
        elem[size++] = t;
}

void print() {
    for (int i = 0; i < size; i++)
        cout << elem[i] << " ";
    cout << endl;
}
};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T* t) {
        T* tmp;
        if (t == NULL) { // Check for NULL
            cout << "Null pointer!" << endl;
            return;
        }

        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = *t; // Dereference
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = *t; // Dereference
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};

int main() {

```

```

Bag<int> xi;
Bag<char> xc;
Bag<int*> xp; // Uses partial specialization for pointer types.

xi.add(10);
xi.add(9);
xi.add(8);
xi.print();

xc.add('a');
xc.add('b');
xc.add('c');
xc.print();

int i = 3, j = 87, *p = new int[2];
*p = 8;
*(p + 1) = 100;
xp.add(&i);
xp.add(&j);
xp.add(p);
xp.add(p + 1);
delete[] p;
p = NULL;
xp.add(p);
xp.print();
}

```

Output

```

10 9 8
a b c
Null pointer!
3 87 8 100

```

예: 한 가지 형식이 되도록 부분 특수화 정의 int

다음 예제에서는 두 형식의 쌍을 사용하는 템플릿 클래스를 정의한 다음 형식 중 하나가 되도록 특수화된 템플릿 클래스의 부분 특수화를 정의합니다 `int`. 특수화는 정수에 따라 간단한 거품형 정렬을 구현하는 추가 정렬 메서드를 정의합니다.

C++

```

// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
    Key* keys;

```

```

Value* values;
int size;
int max_size;
public:
Dictionary(int initial_size) : size(0) {
    max_size = 1;
    while (initial_size >= max_size)
        max_size *= 2;
    keys = new Key[max_size];
    values = new Value[max_size];
}
void add(Key key, Value value) {
    Key* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
        max_size *= 2;
        tmpKey = new Key [max_size];
        tmpVal = new Value [max_size];
        for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
        }
        tmpKey[size] = key;
        tmpVal[size] = value;
        delete[] keys;
        delete[] values;
        keys = tmpKey;
        values = tmpVal;
    }
    else {
        keys[size] = key;
        values[size] = value;
    }
    size++;
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}
};

// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
int* keys;
Value* values;
int size;
int max_size;
public:
Dictionary(int initial_size) : size(0) {
    max_size = 1;
    while (initial_size >= max_size)
        max_size *= 2;
    keys = new int[max_size];
    values = new Value[max_size];
}

```

```

}

void add(int key, Value value) {
    int* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
        max_size *= 2;
        tmpKey = new int [max_size];
        tmpVal = new Value [max_size];
        for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
        }
        tmpKey[size] = key;
        tmpVal[size] = value;
        delete[] keys;
        delete[] values;
        keys = tmpKey;
        values = tmpVal;
    }
    else {
        keys[size] = key;
        values[size] = value;
    }
    size++;
}

void sort() {
    // Sort method is defined.
    int smallest = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
                smallest = j;
        }
        swap(keys[i], keys[smallest]);
        swap(values[i], values[smallest]);
    }
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}

int main() {
    Dictionary<const char*, const char*> dict(10);
    dict.print();
    dict.add("apple", "fruit");
    dict.add("banana", "fruit");
    dict.add("dog", "animal");
    dict.print();

    Dictionary<int, const char*> dict_specialized(10);
    dict_specialized.print();
}

```

```
dict_specialized.add(100, "apple");
dict_specialized.add(101, "banana");
dict_specialized.add(103, "dog");
dict_specialized.add(89, "cat");
dict_specialized.print();
dict_specialized.sort();
cout << endl << "Sorted list:" << endl;
dict_specialized.print();
}
```

Output

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```

템플릿 및 이름 확인

아티클 • 2023. 10. 12.

템플릿 정의에는 3가지 형식의 이름이 있습니다.

- 지역으로 선언된 이름(템플릿 자체의 이름과 템플릿 정의 내에 선언된 모든 이름 포함)
- 템플릿 정의 바깥쪽 범위의 이름
- 어떤 방식으로든 템플릿 인수에 종속되는 이름(종속 이름)

처음 두 이름은 클래스 및 함수 범위와 관련된 반면 종속 이름의 경우 복잡성을 처리하기 위해 템플릿 정의에 이름 확인을 위한 특별한 규칙이 요구됩니다. 이는 템플릿이 인스턴스화될 때까지 컴파일러가 이러한 이름에 대해 거의 알지 못하기 때문입니다. 이름의 형식은 사용되는 템플릿 인수에 따라 전혀 달라질 수 있습니다. 독립적 이름은 일반적인 규칙에 따라 템플릿 정의 시점에 조회됩니다. 템플릿 인수와는 별개인 이러한 이름은 모든 템플릿 특수화를 위해 한 번 조회됩니다. 종속 이름은 템플릿이 인스턴스화될 때까지 조회되지 않으며 각 특수화에 대해 개별적으로 조회됩니다.

형식은 템플릿 인수에 종속된 경우 종속적입니다. 특히 형식은 다음과 같은 경우 종속적입니다.

- 템플릿 인수 자체임

C++

T

- 종속 형식을 포함하며 한정자를 사용하는 정규화된 이름임

C++

T::myType

- 정규화되지 않은 부분이 종속 형식을 식별하는 경우 정규화된 이름임

C++

N::T

- 기본 형식이 종속 형식인 const 또는 volatile 형식임

```
C++
```

```
const T
```

- 종속 형식 기반의 포인터, 참조, 배열 또는 함수 포인터 형식임

```
C++
```

```
T *, T &, T [10], T (*)()
```

- 크기가 템플릿 매개 변수를 기반으로 하는 배열임

```
C++
```

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- 템플릿 매개 변수에서 생성된 템플릿 형식임

```
C++
```

```
T<int>, MyTemplate<T>
```

형식 종속성 및 값 종속성

템플릿 매개 변수에 종속된 이름 및 식은 템플릿 매개 변수가 형식 매개 변수인지, 아니면 값 매개 변수인지에 따라 형식 종속 항목 또는 값 종속 항목으로 분류됩니다. 또한 템플릿 인수에 종속된 형식으로 템플릿에 선언된 모든 식별자는 값 종속 식으로 초기화된 정수 계열 또는 열거형 형식과 마찬가지로 값 종속 항목으로 간주됩니다.

형식 종속 및 값 종속 식은 형식 또는 값에 종속되는 변수가 포함된 식입니다. 이러한 식의 의미 체계는 템플릿에 사용되는 매개 변수에 따라 달라질 수 있습니다.

참고 항목

[템플릿](#)

종속적인 형식에 대한 이름 확인

아티클 • 2024. 11. 21.

템플릿 정의에서 정규화된 이름을 사용하여 `typename` 지정된 정규화된 이름이 형식을 식별하도록 컴파일러에 알릴 수 있습니다. 자세한 내용은 `typename`을 참조 [하세요](#).

C++

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
    void f(typename T::myType* mt) {}
};

class Yarg
{
public:
    struct myType { };
};

int main()
{
    X<Yarg> x;
    x.f(new Yarg::myType());
    printf("Name resolved by using typename keyword.");
}
```

Output

```
Name resolved by using typename keyword.
```

종속 이름에 대한 이름 조회는 템플릿 정의의 컨텍스트와 다음 예제에서 이 컨텍스트에서 찾을 수 `myFunction(char)` 있는 이름과 템플릿 인스턴스화의 컨텍스트를 모두 검사합니다. 다음 예제에서는 템플릿이 `main`에서 인스턴스화됩니다. 따라서

`MyNamespace::myFunction` 인스턴스화 지점에서 볼 수 있으며 더 나은 일치 항목으로 선택됩니다. `MyNamespace::myFunction`의 이름이 바뀐 경우 `myFunction(char)`이 대신 호출됩니다.

모든 이름은 종속 이름인 것처럼 확인됩니다. 그러나 충돌 발생 가능성이 있을 경우 철저하게 정규화된 이름을 사용하는 것이 좋습니다.

C++

```

// template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
    cout << "Char myFunction" << endl;
}

template <class T> class Class1
{
public:
    Class1(T i)
    {
        // If replaced with myFunction(1), myFunction(char)
        // will be called
        myFunction(i);
    }
};

namespace MyNamespace
{
    void myFunction(int)
    {
        cout << "Int MyNamespace::myFunction" << endl;
    }
};

using namespace MyNamespace;

int main()
{
    Class1<int>* c1 = new Class1<int>(100);
}

```

출력

Output

Int MyNamespace::myFunction

템플릿 명확성

Visual Studio 2012는 "template" 키워드를 사용하여 명확하게 구분하기 위해 C++98/03/11 표준 규칙을 적용합니다. 다음 예제에서 Visual Studio 2010은 일치하지 않는 줄과 규격 줄을 모두 허용합니다. Visual Studio 2012는 규격 줄만 허용합니다.

C++

```
#include <iostream>
#include <ostream>
#include <typeinfo>
using namespace std;

template <typename T> struct Allocator {
    template <typename U> struct Rebind {
        typedef Allocator<U> Other;
    };
};

template <typename X, typename AY> struct Container {
    #if defined(NONCONFORMANT)
        typedef typename AY::Rebind<X>::Other AX; // nonconformant
    #elif defined(CONFORMANT)
        typedef typename AY::template Rebind<X>::Other AX; // conformant
    #else
        #error Define NONCONFORMANT or CONFORMANT.
    #endif
};

int main() {
    cout << typeid(Container<int, Allocator<float>>::AX).name() << endl;
}
```

기본적으로 C++에서는 `AY::Rebind`가 템플릿이 아니며 컴파일러에서 다음 "⟨"을 less-than으로 해석한다고 가정하기 때문에 명확성 규칙 준수가 필요합니다. C++에 `Rebind`가 템플릿이라는 것을 인지시켜 "⟨"을 꺼쇠로 올바르게 구문 분석할 수 있도록 해야 합니다.

참고 항목

[이름 확인](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

로컬로 선언된 이름에 대한 이름 확인

아티클 • 2023. 04. 03.

템플릿 인수를 사용하거나 사용하지 않고 템플릿 이름 자체를 참조할 수 있습니다. 클래스 템플릿의 범위에서 이름 자체는 템플릿을 나타냅니다. 템플릿 특수화 또는 부분 특수화의 범위에서 이름만 특수화 또는 부분 특수화를 참조합니다. 템플릿의 다른 특수화나 부분 특수화도 해당 템플릿 인수와 함께 참조될 수 있습니다.

예: 특수화 및 부분 특수화

다음 코드는 클래스 템플릿의 이름이 A 특수화 또는 부분 특수화의 범위에서 다르게 해석되는 것을 보여 주는 코드입니다.

C++

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
    A* a1;      // A refers to A<T>
    A<int>* a2; // A<int> refers to a specialization of A.
    A<T*>* a3; // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
    A* a4; // A refers to A<T*>.
};

template<> class A<int> {
    A* a5; // A refers to A<int>.
};
```

예: 템플릿 매개 변수와 개체 간의 이름 충돌

템플릿 매개 변수와 다른 개체 간에 이름이 충돌하는 경우 템플릿 매개 변수를 숨길 수도 있고 숨길 수 없습니다. 다음 규칙은 우선 순위를 결정하는 데 도움이 됩니다.

템플릿 매개 변수의 범위는 매개 변수가 처음 나타난 지점부터 클래스 또는 함수 템플릿의 끝까지입니다. 이름이 템플릿 인수 목록 또는 기본 클래스 목록에 다시 나타나는 경우 동일한 형식을 참조합니다. 표준 C++에서는 이 이외에 템플릿 매개 변수와 동일한 이름을 같은 범위에서 선언할 수 없습니다. Microsoft 확장은 템플릿 매개 변수가 템플릿 범위에서 다시 정의될 수 있도록 합니다. 다음 예제는 클래스 템플릿의 기본 지정에서의 템플릿 매개 변수 사용을 보여 줍니다.

C++

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
    // Derived1<int> d;
}
```

예제: 클래스 템플릿 외부의 멤버 함수 정의

멤버 함수가 클래스 템플릿 외부에서 정의되면 다른 템플릿 매개 변수 이름을 사용할 수 있습니다. 클래스 템플릿 멤버 함수의 정의가 선언과 다른 템플릿 매개 변수 이름을 사용하고 정의에 사용된 이름이 선언의 다른 멤버와 충돌하는 경우 템플릿 선언의 멤버가 우선합니다.

C++

```
// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
    struct Z {
        Z() { cout << "Z::Z()" << endl; }
    };
    void f();
};

template <class Z>
void C<Z>::f() {
    // Z refers to the struct Z, not to the template arg;
    // Therefore, the constructor for struct Z will be called.
    Z z;
}

int main() {
    C<int> c;
    c.f();
}
```

Output

```
Z::Z()
```

예: 네임스페이스 외부의 템플릿 또는 멤버 함수 정의

템플릿이 선언된 네임스페이스 외부에서 함수 템플릿 또는 멤버 함수를 정의할 때 템플릿 인수가 네임스페이스의 다른 멤버 이름보다 우선합니다.

C++

```
// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
    void g() { cout << "NS::g" << endl; }

    template <class T> struct C {
        void f();
        void g() { cout << "C<T>::g" << endl; }
    };
}

template <class T>
void NS::C<T>::f() {
    g(); // C<T>::g, not NS::g
}

int main() {
    NS::C<int> c;
    c.f();
}
```

Output

```
C<T>::g
```

예: 기본 클래스 또는 멤버 이름은 템플릿 인수를 숨깁니다.

템플릿 클래스 선언 외부에 있는 정의에서 템플릿 클래스에 템플릿 인수에 종속되지 않는 기본 클래스가 있고 기본 클래스 또는 해당 멤버 중 하나가 템플릿 인수와 동일한 이름을 갖는 경우 기본 클래스 또는 멤버 이름은 템플릿 인수를 숨깁니다.

C++

```
// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
    int i;
    void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
    void f();
};

template <class B, int i>
void C<B, i>::f() {
    B b; // Base class b, not template argument.
    b.print();
    i = 1; // Set base class's i to 1.
}

int main() {
    C<int, 1> c;
    c.f();
    cout << c.i << endl;
}
```

Output

```
Base
1
```

참고 항목

[이름 확인](#)

함수 템플릿 호출의 오버로드 확인

아티클 • 2023. 10. 12.

함수 템플릿은 같은 이름의 비 템플릿 함수를 오버로드할 수 있습니다. 이 시나리오에서 컴파일러는 먼저 템플릿 인수 추론을 사용하여 함수 템플릿을 고유한 특수화로 인스턴스화하여 함수 호출을 확인하려고 시도합니다. 템플릿 인수 추론이 실패하면 컴파일러는 인스턴스화된 함수 템플릿 오버로드와 비 템플릿 함수 오버로드를 모두 고려하여 호출을 해결합니다. 이러한 다른 오버로드는 후보 집합이라고 합니다. 템플릿 인수 추론에 성공하면 생성된 함수를 후보 집합의 다른 함수와 비교하여 오버로드 확인 규칙에 따라 가장 일치하는 항목을 결정합니다. 자세한 내용은 함수 오버로드를 참조하세요.

예: 템플릿이 아닌 함수 선택

비 템플릿 함수가 함수 템플릿과 똑같이 일치하는 경우 다음 예제의 호출 `f(1, 1)` 과 같이 템플릿이 아닌 함수가 선택됩니다(템플릿 인수가 명시적으로 지정되지 않은 경우).

C++

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
}

int main()
{
    f(1, 1);    // Equally good match; choose the non-template function.
    f('a', 1); // Chooses the function template.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

Output

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

예: 정확한 일치 함수 템플릿 기본 설정

다음 예제에서는 템플릿이 아닌 함수에 변환이 필요한 경우 정확히 일치하는 함수 템플릿이 선호된다는 것을 보여 줍니다.

C++

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    long l = 0;
    int i = 0;
    // Call the function template f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}
```

Output

```
void f(T1, T2)
```

참고 항목

[이름 확인](#)

[typename](#)

소스 코드 조직(C++ 템플릿)

아티클 • 2024. 11. 21.

클래스 템플릿을 정의할 때 멤버 정의가 컴파일러에 필요할 경우 표시되도록 소스 코드를 구성해야 합니다. 포함 모델이나 명시적 인스턴스화 모델을 사용할 수 있습니다. 포함 모델에서는 템플릿을 사용하는 모든 파일에 멤버 정의를 포함합니다. 이 방식은 가장 단순하며 템플릿에 구체적인 형식을 사용할 수 있다는 면에서 가장 유연합니다. 단점은 컴파일 시간이 늘어날 수 있는 것입니다. 프로젝트 또는 포함된 파일 자체가 큰 경우 시간이 중요할 수 있습니다. 명시적 인스턴스화 방식을 사용하면 템플릿 자체에서 구체적인 클래스 또는 클래스 멤버를 특정 형식에 맞게 인스턴스화합니다. 이 방식은 컴파일 시간을 단축할 수 있지만, 템플릿 구현자를 미리 사용하도록 설정한 클래스에만 사용할 수 있습니다. 일반적으로 컴파일 시간이 문제가 되는 경우를 제외하고는 포함 모델을 사용하는 것이 좋습니다.

배경

템플릿은 컴파일러가 템플릿 또는 해당 멤버에 대한 개체 코드를 생성하지 않는다는 점에서 일반 클래스와 같지 않습니다. 템플릿이 구체적인 형식으로 인스턴스화될 때까지 생성할 것이 없습니다. 컴파일러에서 `MyClass<int> mc;` 과 같은 템플릿 인스턴스화가 발생하는 데 해당 시그니처가 있는 클래스가 아직 없으면 새 클래스를 생성합니다. 또한 사용되는 멤버 함수의 코드를 생성하려고 합니다. 이러한 정의가 컴파일되는 .cpp 파일에서 직접 또는 간접적으로 #included 않은 파일에 있는 경우 컴파일러는 해당 정의를 볼 수 없습니다. 컴파일러의 관점에서 볼 때 반드시 오류가 아닌 것은 아닙니다. 함수는 링커가 찾을 다른 번역 단위에 정의될 수 있습니다. 링커에서 해당 코드를 찾지 못하면 해결되지 않은 외부 오류가 발생합니다.

포함 모델

템플릿 정의를 변환 단위 전체에 표시하도록 설정하는 가장 일반적이고 간단한 방법은 헤더 파일 자체에 정의를 넣는 것입니다. 템플릿을 사용하는 모든 .cpp 파일은 헤더에 `#include` 만 있습니다. 이 방법은 표준 라이브러리에서 사용됩니다.

C++

```
#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
```

```

T arr[N];
public:
// Full definitions:
MyArray(){}  

void Print()
{
    for (const auto v : arr)
    {
        std::cout << v << " , ";
    }
}

T& operator[](int i)
{
    return arr[i];
}
};

#endif

```

이 방식을 사용하면 컴파일러가 전체 템플릿 정의에 액세스할 수 있고 임의 형식의 템플릿을 필요에 따라 인스턴스화할 수 있습니다. 간단하고 비교적 쉽게 유지 관리할 수 있습니다. 그러나 포함 모델은 컴파일 시간이 길다는 단점이 있습니다. 큰 프로그램에서, 특히 템플릿 헤더 자체에 다른 헤더가 #포함된 경우 이 단점이 상당할 수 있습니다. 헤더를 #includes 모든 .cpp 파일은 함수 템플릿 및 모든 정의의 자체 복사본을 가져옵니다. 링커는 일반적으로 함수에 대한 여러 정의로 끝나지 않도록 항목을 정렬할 수 있지만 이 작업을 수행하는 데 시간이 걸립니다. 작은 프로그램에서는 이런 추가 컴파일 시간이 길지 않을 수 있습니다.

명시적 인스턴스화 모델

프로젝트에 포함 모델이 실행 가능하지 않고 템플릿을 인스턴스화하는 데 사용할 형식 집합을 확실하게 알고 있는 경우 템플릿 코드를 .h 파일로 분리하고 .cpp 파일에서 .cpp 템플릿을 명시적으로 인스턴스화할 수 있습니다. 이 방법은 컴파일러가 사용자 인스턴스화를 발견할 때 볼 수 있는 개체 코드를 생성합니다.

키워드 **template** 와 인스턴스화하려는 엔터티의 서명을 사용하여 명시적 인스턴스화를 만듭니다. 이 엔터티는 형식 또는 멤버일 수 있습니다. 형식을 명시적으로 인스턴스화하는 경우 모든 멤버가 인스턴스화됩니다.

헤더 파일 `MyArray.h` 은 템플릿 클래스 `MyArray` 를 선언합니다.

C++

```

//MyArray.h
#ifndef MYARRAY
#define MYARRAY

```

```

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};

#endif

```

소스 파일 `MyArray.cpp` 은 명시적으로 인스턴스화하고 `template MyArray<string, 5>` 다음을 수행합니다 `template MyArray<double, 5>`.

C++

```

//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << " ";
    }
    cout << endl;
}

template MyArray<double, 5>;
template MyArray<string, 5>;

```

이전 예제에서 명시적 인스턴스화는 파일의 맨 아래에 있습니다 `.cpp`. A는 `MyArray` 형식에 `String` 대해서만 `double` 사용할 수 있습니다.

① 참고

C++11에서 `export` 키워드는 템플릿 정의의 컨텍스트에서 더 이상 사용되지 않습니다. 대부분의 컴파일러에서 이 키워드를 지원하지 않았으므로 실제로 영향은 거의 없습니다.

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

이벤트 처리

아티클 • 2023. 10. 12.

이벤트 처리는 주로 COM 클래스(일반적으로 ATL 클래스 또는 [coclass](#) 특성을 사용하여 COM 개체를 구현하는 C++ 클래스)에서 지원됩니다. 자세한 내용은 COM의 이벤트 처리를 참조 [하세요](#).

네이티브 C++ 클래스(COM 개체를 구현하지 않는 C++ 클래스)에도 이벤트 처리가 지원됩니다. 네이티브 C++ 이벤트 처리 지원은 더 이상 사용되지 않으며 향후 릴리스에서 제거될 예정입니다. 자세한 내용은 네이티브 C++의 이벤트 처리를 참조 [하세요](#).

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. 규칙 모드를 지정할 `/permissive-` 때 컴파일되지 않습니다.

이벤트 처리는 단일 및 다중 스레드 사용을 모두 지원합니다. 동시 다중 스레드 액세스로부터 데이터를 보호합니다. 이벤트 원본 또는 수신기 클래스에서 서브클래스를 파생시킬 수 있습니다. 이러한 하위 클래스는 확장 이벤트 소싱 및 수신을 지원합니다.

Microsoft C++ 컴파일러에는 이벤트 및 이벤트 처리기를 선언하기 위한 특성 및 키워드(keyword) 포함되어 있습니다. CLR 프로그램과 네이티브 C++ 프로그램에서 이벤트 특성과 키워드를 사용할 수 있습니다.

아티클	설명
event_source	이벤트 소스를 만듭니다.
event_receiver	이벤트 수신기(싱크)를 만듭니다.
_event	이벤트를 선언합니다.
_raise	이벤트의 호출 사이트를 강조합니다.
_hook	처리기 메서드를 이벤트와 연결합니다.
_unhook	이벤트에서 처리기 메서드를 연결 해제합니다.

참고 항목

[C++ 언어 참조](#)

[키워드](#)

`_event` 키워드

아티클 • 2024. 07. 12.

이벤트를 선언합니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. `/permissive-` 규칙 모드를 지정하면 컴파일되지 않습니다.

구문

```
_event member-function-declarator ;
 _event __interface interface-specifier ;
 _event data-member-declarator ;
```

설명

Microsoft 관련 키워드 `_event`은 멤버 함수 선언, 인터페이스 선언 또는 데이터 멤버 선언에 적용할 수 있습니다. 하지만 `_event` 키워드를 사용하여 중첩된 클래스의 멤버를 한정할 수는 없습니다.

이벤트 소스 및 수신기가 네이티브 C++ 또는 COM이거나 관리되는지(.NET Framework)에 따라 다음 구문을 이벤트로 사용할 수 있습니다.

[+] 테이블 확장

네이티브 C++	COM:	관리됨(.NET Framework)
멤버 함수	-	메서드(method)
-	인터페이스	-
-	-	데이터 멤버

이벤트 수신기에서 `_hook`을(를) 사용하여 처리기 멤버 함수를 이벤트 멤버 함수와 연결합니다. 키워드를 사용하여 이벤트를 `_event` 만든 후에는 이벤트가 호출될 때 해당 이벤트에 연결된 모든 이벤트 처리기가 호출됩니다.

`_event` 멤버 함수 선언에는 정의가 있을 수 없습니다. 정의는 암시적으로 생성되므로 이벤트 멤버 함수는 일반 멤버 함수인 것처럼 호출할 수 있습니다.

① 참고

템플릿 기반 클래스 또는 구조체에 `event`를 포함시킬 수 없습니다.

네이티브 이벤트

네이티브 이벤트는 멤버 함수입니다. 반환 형식은 일반적으로 `HRESULT` 또는 `void`이지만 `enum`을 비롯한 모든 정수 계열 형식일 수 있습니다. 이벤트가 정수 반환 형식을 사용하는 경우 이벤트 처리기가 0이 아닌 값을 반환할 때 오류 조건이 정의됩니다. 이 경우 발생하는 이벤트는 다른 대리자를 호출합니다.

C++

```
// Examples of native C++ events:  
__event void OnDblClick();  
__event HRESULT OnClick(int* b, char* s);
```

샘플 코드는 [네이티브 C++에서 이벤트 처리](#)를 참조하세요.

COM 이벤트

COM 이벤트는 인터페이스입니다. 이벤트 소스 인터페이스에서 멤버 함수의 매개 변수는 *in* 매개 변수여야 하지만 엄격하게 적용되지는 않습니다. *out* 매개 변수는 멀티캐스팅할 때 유용하지 않기 때문입니다. *out* 매개 변수를 사용하는 경우 수준 1 경고가 발생합니다.

반환 형식은 일반적으로 `HRESULT` 또는 `void`이지만 `enum`을 비롯한 모든 정수 계열 형식일 수 있습니다. 이벤트가 정수 반환 형식을 사용하고 이벤트 처리기가 0이 아닌 값을 반환하는 경우 오류 조건입니다. 발생하는 이벤트는 다른 대리자 호출을 중단합니다. 컴파일러는 생성된 IDL에서 이벤트 원본 인터페이스를 `source`(으)로 자동으로 표시합니다.

COM 이벤트 소스의 경우 `_interface` 키워드가 `_event` 뒤에 항상 필요합니다.

C++

```
// Example of a COM event:  
__event __interface IEvent1;
```

샘플 코드는 [COM에서 이벤트 처리](#)를 참조하세요.

관리되는 이벤트

새 구문의 이벤트 코딩에 대한 자세한 내용은 [이벤트](#)를 참조하세요.

관리되는 이벤트는 데이터 멤버 또는 멤버 함수입니다. 이벤트와 함께 사용하는 경우 대리자의 반환 형식은 [공용 언어 사양](#)을 준수해야 합니다. 이벤트 처리기의 반환 형식은 대리자의 반환 형식과 일치해야 합니다. 대리자에 대한 자세한 내용은 [대리자 및 이벤트](#)를 참조하세요. 관리되는 이벤트가 데이터 멤버인 경우 그 형식은 대리자에 대한 포인터여야 합니다.

.NET Framework에서 데이터 멤버를 메서드(즉, 해당 대리자의 `Invoke` 메서드) 자체인 것처럼 취급할 수 있습니다. 이렇게 하려면 관리되는 이벤트 데이터 멤버를 선언하기 위한 대리자 형식을 미리 정의합니다. 반면에 관리되는 이벤트 메서드는 해당 관리되는 대리자(이미 정의되지 않은 경우)를 암시적으로 정의합니다. 예를 들어 `onClick`과 같은 이벤트 값을 다음과 같이 이벤트로 선언할 수 있습니다.

C++

```
// Examples of managed events:  
__event ClickEventHandler* OnClick; // data member as event  
__event void OnClick(String* s); // method as event
```

관리되는 이벤트를 암시적으로 선언할 때 이벤트 처리기가 추가되거나 제거되는 경우 호출될 `add` 및 `remove` 접근자를 지정할 수 있습니다. 클래스 외부에서 이벤트를 호출하는(발생시키는) 멤버 함수를 정의할 수도 있습니다.

예제: 네이티브 이벤트

C++

```
// EventHandling_Native_Event.cpp  
// compile with: /c  
[event_source(native)]  
class CSource {  
public:  
    __event void MyEvent(int nValue);  
};
```

예제: COM 이벤트

C++

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbases.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-
B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT MyEvent();
};

[ coclass, uuid("00000000-0000-0000-0000-000000000003"), event_source(com)
]
class CSource : public IEventSource {
public:
    __event __interface IEventSource;
    HRESULT FireEvent() {
        __raise MyEvent();
        return S_OK;
    }
};
```

참고 항목

키워드

이벤트 처리

event_source

event_receiver

_hook

_unhook

_raise

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

hook 키워드

아티클 • 2024. 07. 15.

처리기 메서드를 이벤트와 연결합니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. /permissive- 규칙 모드를 지정하면 컴파일되지 않습니다.

구문

C++

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

매개 변수

`&SourceClass::EventMethod`

이벤트 처리기 메서드를 후크할 이벤트 메서드에 대한 포인터입니다.

- 네이티브 C++ 이벤트: `SourceClass`는 이벤트 원본 클래스이며 `EventMethod`는 이벤트입니다.
- COM 이벤트: `SourceClass`는 이벤트 원본 인터페이스이며 `EventMethod`는 해당 메서드 중 하나입니다.
- 관리되는 이벤트: `SourceClass`는 이벤트 원본 클래스이며 `EventMethod`는 이벤트입니다.

`interface`

`receiver`에 후크되는 인터페이스 이름이며, `event_receiver` 특성의 `Layout_dependent` 매

개 변수가 `true`인 COM 이벤트 수신기에만 해당합니다.

`source`

이벤트 소스의 인스턴스에 대한 포인터입니다. `event_receiver`에서 지정된 코드 `type`에 따라 `source`는 다음 형식 중 하나가 될 수 있습니다.

- 네이티브 이벤트 소스 개체 포인터입니다.
- `IUnknown` 기반 포인터(COM 원본).
- 관리되는 이벤트의 관리되는 개체 포인터입니다.

`&ReceiverClass::HandlerMethod`

이벤트에 후크될 이벤트 처리기 메서드에 대한 포인터입니다. 처리기는 클래스의 메서드 또는 동일한 클래스에 대한 참조로 지정됩니다. 클래스 이름을 지정하지 않으면 `_hook`는 자신이 호출된 클래스가 해당 클래스와 같다고 가정합니다.

- 네이티브 C++ 이벤트: `ReceiverClass`는 이벤트 수신기 클래스이며 `HandlerMethod`는 처리기입니다.
- COM 이벤트: `ReceiverClass`는 이벤트 수신기 인터페이스이며 `HandlerMethod`는 해당 처리기 중 하나입니다.
- 관리되는 이벤트: `ReceiverClass`는 이벤트 수신기 클래스이며 `HandlerMethod`는 처리기입니다.

`receiver`

(선택 사항) 이벤트 수신기 클래스의 인스턴스에 대한 포인터입니다. 수신기를 지정하지 않으면 기본값은 `_hook`가 호출되는 수신기 클래스 또는 구조체입니다.

사용

이벤트 수신기 클래스 외부의 `main`을 포함하여 모든 함수 범위에서 사용할 수 있습니다.

설명

이벤트 수신기에서 내장 함수 `_hook`를 사용하여 처리기 메서드를 이벤트 메서드와 연결하거나 후크할 수 있습니다. 이렇게 하면 소스에서 지정된 이벤트를 발생시킬 때 지정된 처리기가 호출됩니다. 여러 처리기를 단일 이벤트에 후크하거나 여러 이벤트를 단일 처리기에 후크할 수 있습니다.

두 가지 형태의 `_hook` 가 있습니다. 대부분의 경우에 첫 번째 형태(인수 4개)를 사용할 수 있으며, 특히 `event_receiver` 특성의 `layout_dependent` 매개 변수가 `false` 인 COM 이벤트 수신기에 사용할 수 있습니다.

이러한 경우 메서드 중 하나에서 이벤트를 발생시키기 전에 인터페이스에서 모든 메서드를 후크할 필요는 없습니다. 이벤트를 처리하는 메서드만 후크하면 됩니다.

`Layout_dependent = true` 인 COM 이벤트 수신기에만 두 번째 형태(인수 2개)의 `_hook` 를 사용할 수 있습니다.

`_hook` 는 long 값을 반환합니다. 0이 아닌 반환 값은 오류가 발생했음을 나타냅니다(관리되는 이벤트가 예외를 throw함).

컴파일러는 이벤트가 있는지 확인하고 이벤트 시그니처가 대리자 시그니처와 일치하는지 확인합니다.

COM 이벤트를 제외하고 이벤트 수신기 외부에서 `_hook` 및 `_unhook` 를 호출할 수 있습니다.

`_hook` 를 사용하는 대신 `+ =` 연산자를 사용할 수 있습니다.

새 구문에서 관리되는 이벤트를 코딩하는 방법에 대한 자세한 내용은 [event](#)를 참조하세요.

① 참고

템플릿 기반 클래스 또는 구조체에 `event` 를 포함시킬 수 없습니다.

예시

샘플은 [네이티브 C++](#)에서 이벤트 처리 및 COM에서 이벤트 처리를 참조하세요.

참고 항목

키워드

이벤트 처리

`event_source`

`event_receiver`

`_event`

`_unhook`

`_raise`

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_raise` 키워드

아티클 • 2024. 07. 15.

이벤트의 호출 사이트를 강조합니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. `/permissive-` 적합성 모드를 지정하면 컴파일되지 않습니다.

구문

```
_raise method-declarator ;
```

설명

관리 코드에서 이벤트가 정의된 클래스 내에서만 이벤트를 발생시킬 수 있습니다. 자세한 내용은 [event](#)를 참조하세요.

비이벤트를 호출할 경우 `_raise` 키워드를 사용하면 오류가 생략됩니다.

① 참고

템플릿 기반 클래스 또는 구조체에 event를 포함시킬 수 없습니다.

예시

C++

```
// EventHandlingRef_raise.cpp
struct E {
    __event void func1();
    void func1(int) {}

    void func2() {}

    void b() {
        __raise func1();
        __raise func1(1); // C3745: 'int Event::bar(int)':           // only an event can be 'raised'
                          // only an event can be 'raised'
        __raise func2(); // C3745
```

```
}

int main() {
    E e;
    __raise e.func1();
    __raise e.func1(1); // C3745
    __raise e.func2(); // C3745
}
```

참고 항목

[키워드](#)

[이벤트 처리](#)

[_event](#)

[_hook](#)

[_unhook](#)

[.NET 및 UWP용 구성 요소 확장](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__unhook 키워드

아티클 • 2024. 11. 21.

이벤트에서 처리기 메서드를 연결 해제합니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. /permissive- 규칙 모드를 지정하면 컴파일되지 않습니다.

구문

C++

```
long __unhook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __unhook(
    interface,
    source
);

long __unhook(
    source
);
```

매개 변수

`&SourceClass::EventMethod`

이벤트 처리기 메서드를 해제한 이벤트 메서드에 대한 포인터입니다.

- 네이티브 C++ 이벤트: `SourceClass`는 이벤트 원본 클래스이며 `EventMethod`는 이벤트입니다.
- COM 이벤트: `SourceClass`는 이벤트 원본 인터페이스이며 `EventMethod`는 해당 메서드 중 하나입니다.
- 관리되는 이벤트: `SourceClass`는 이벤트 원본 클래스이며 `EventMethod`는 이벤트입니다.

`interface`

수신기에서 해제되는 인터페이스 이름이며, 특성의 `layout_dependent` 매개 변수가 있는 COM 이벤트 수신기에만 해당 `event_receiver` 합니다 `true`.

`source`

이벤트 소스의 인스턴스에 대한 포인터입니다. 지정된 코드 `type` `event_receiver`에 따라 원본은 다음 형식 중 하나일 수 있습니다.

- 네이티브 이벤트 소스 개체 포인터입니다.
- `IUnknown` 기반 포인터(COM 원본).
- 관리되는 이벤트의 관리되는 개체 포인터입니다.

`&ReceiverClass::HandlerMethod` 이벤트에서 해제할 이벤트 처리기 메서드에 대한 포인터입니다. 처리기는 클래스의 메서드 또는 동일한 참조로 지정됩니다. 클래스 이름을 `_unhook` 지정하지 않으면 클래스가 호출되는 클래스로 가정합니다.

- 네이티브 C++ 이벤트: `ReceiverClass`는 이벤트 수신기 클래스이며 `HandlerMethod`는 처리기입니다.
- COM 이벤트: `ReceiverClass`는 이벤트 수신기 인터페이스이며 `HandlerMethod`는 해당 처리기 중 하나입니다.
- 관리되는 이벤트: `ReceiverClass`는 이벤트 수신기 클래스이며 `HandlerMethod`는 처리기입니다.

`receiver`(선택 사항) 이벤트 수신기 클래스의 인스턴스에 대한 포인터입니다. 수신기를 지정하지 않으면 기본값은 `_unhook` 가 호출되는 수신기 클래스 또는 구조체입니다.

사용

이벤트 수신기 클래스 외부를 포함하여 `main` 모든 함수 범위에서 사용할 수 있습니다.

설명

이벤트 수신기의 내장 함수 `_unhook` 를 사용하여 이벤트 메서드에서 처리기 메서드를 연결 해제하거나 "해제"합니다.

의 세 가지 형태가 있습니다 `_unhook`. 대부분 첫 번째(인수 4개) 형식을 사용할 수 있습니다. COM 이벤트 수신기에 대해서만 두 번째(두 인수) 형식 `_unhook` 을 사용할 수 있습니다.

다. 전체 이벤트 인터페이스를 해제합니다. 세 번째(인수 1개) 형식을 사용하여 지정된 소스에서 모든 대리자를 언후크할 수 있습니다.

0이 아닌 반환 값은 예외가 발생했음을 나타냅니다(관리되는 이벤트가 예외를 throw함).

아직 연결되지 않은 이벤트 및 이벤트 처리기를 호출 `_unhook` 하는 경우 아무런 영향을 주지 않습니다.

컴파일할 때 컴파일러는 이벤트가 있는지 확인하고 지정된 처리기를 사용하여 매개 변수 형식을 검사합니다.

COM 이벤트를 제외하고 이벤트 수신기 외부에서 `_hook` 및 `_unhook`를 호출할 수 있습니다.

사용에 `_unhook` 대한 대안은 -= 연산자를 사용하는 것입니다.

새 구문에서 관리되는 이벤트를 코딩하는 방법에 대한 자세한 내용은 이벤트를 [참조하세요](#).

① 참고

템플릿 기반 클래스 또는 구조체에 `event`를 포함시킬 수 없습니다.

예시

샘플은 네이티브 C++의 이벤트 처리 및 COM의 이벤트 처리를 참조하세요.

참고 항목

키워드

`event_source`

`event_receiver`

`_event`

`_hook`

`_raise`

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

네이티브 C++에서 이벤트 처리

아티클 • 2023. 10. 12.

네이티브 C++ 이벤트 처리에서 `event_source` 및 `event_receiver` 특성을 각각 사용하여 이벤트 원본 및 이벤트 수신기를 `native type` 설정합니다. 이러한 특성을 사용하면 이벤트를 발생시키고 COM이 아닌 네이티브 컨텍스트에서 이벤트를 처리하는 데 적용되는 클래스를 허용합니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. 규칙 모드를 지정할 `/permissive-` 때 컴파일되지 않습니다.

이벤트 선언

이벤트 소스 클래스에서 메서드 선언의 `_event` 키워드(keyword) 사용하여 메서드를 이벤트로 선언합니다. 메서드를 선언해야 하지만 정의하지 마세요. 이렇게 하면 컴파일러가 이벤트로 만들 때 메서드를 암시적으로 정의하므로 컴파일러 오류가 생성됩니다. 네이티브 이벤트는 매개 변수가 0개 이상인 메서드일 수 있습니다. 반환 형식 또는 정수 계열 형식일 `void` 수 있습니다.

이벤트 처리기 정의

이벤트 수신기 클래스에서 이벤트 처리기를 정의합니다. 이벤트 처리기는 처리할 이벤트와 일치하는 서명(반환 형식, 호출 규칙 및 인수)이 있는 메서드입니다.

이벤트 처리기를 이벤트에 연결

또한 이벤트 수신기 클래스에서는 내장 함수 `_hook` 를 사용하여 이벤트를 이벤트 처리기와 연결하고 `_unhook` 이벤트 처리기에서 이벤트를 연결 해제합니다. 한 이벤트 처리기에 여러 이벤트를 후크하거나 한 이벤트에 여러 이벤트 처리기를 후크할 수 있습니다.

이벤트 발생

이벤트를 발생하려면 이벤트 원본 클래스에서 이벤트로 선언된 메서드를 호출합니다. 처리기가 이벤트에 후크된 경우 처리기가 호출됩니다.

네이티브 C++ 이벤트 코드

다음 예제에서는 네이티브 C++에서 이벤트를 발생시키는 방법을 보여 줍니다. 예제를 컴파일 및 실행하려면 코드의 주석을 참조하십시오. Visual Studio IDE에서 코드를 빌드하려면 옵션이 꺼져 있는지 `/permissive-` 확인합니다.

예시

코드

C++

```
// evh_native.cpp
// compile by using: cl /EHsc /W3 evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
    void MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
    }

    void MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
    }

    void hookEvent(CSource* pSource) {
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void unhookEvent(CSource* pSource) {
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    CSource source;
    CReceiver receiver;

    receiver.hookEvent(&source);
    __raise source.MyEvent(123);
```

```
    receiver.unhookEvent(&source);  
}
```

출력

Output

```
MyHandler2 was called with value 123.  
MyHandler1 was called with value 123.
```

참고 항목

[이벤트 처리](#)

COM에서 이벤트 처리

아티클 • 2024. 11. 21.

COM 이벤트 처리에서 각각 지정된 특성과 특성을 사용하여 `event_source` 이벤트 원본 및 `event_receiver` 이벤트 수신기를 `type = com` 설정합니다. 이러한 특성은 사용자 지정, 디스패치 및 이중 인터페이스에 적절한 코드를 삽입합니다. 삽입된 코드를 사용하면 특성이 지정된 클래스가 COM 연결 지점을 통해 이벤트를 발생시키고 이벤트를 처리할 수 있습니다.

① 참고

네이티브 C++의 이벤트 특성은 표준 C++와 호환되지 않습니다. `/permissive-` 규칙 모드를 지정하면 컴파일되지 않습니다.

이벤트 선언하기

이벤트 소스 클래스에서 인터페이스 선언의 `_event` 키워드를 사용하여 해당 인터페이스의 메서드를 이벤트로 선언합니다. 해당 인터페이스의 이벤트는 이를 인터페이스 메서드로 호출할 때 발생합니다. 이벤트 인터페이스의 메서드에는 매개 변수가 0개 이상 있을 수 있습니다(모두 매개 변수에 있어야 합니다). 반환 형식은 `void` 또는 모든 정수 계열 형식이 될 수 있습니다.

이벤트 처리기 정의

이벤트 수신기 클래스에서 이벤트 처리기를 정의합니다. 이벤트 처리기는 처리할 이벤트와 일치하는 서명(반환 형식, 호출 규칙 및 인수)이 있는 메서드입니다. COM 이벤트의 경우 호출 규칙이 일치하지 않아도 됩니다. 자세한 내용은 아래의 레이아웃 종속 COM 이벤트를 [참조하세요](#).

이벤트 처리기를 이벤트에 연결

또한 이벤트 수신기 클래스에서는 내장 함수 `_hook`를 사용하여 이벤트를 이벤트 처리기와 연결하고 `_unhook` 이벤트 처리기에서 이벤트를 연결 해제합니다. 한 이벤트 처리기에 여러 이벤트를 후크하거나 한 이벤트에 여러 이벤트 처리기를 후크할 수 있습니다.

① 참고

일반적으로 COM 이벤트 수신기가 이벤트 소스 인터페이스 정의에 액세스할 수 있게 만드는 두 가지 방법이 있습니다. 첫 번째 방법은 아래와 같이 공용 헤더 파일을 공유하는 것입니다. 두 번째는 가져오기 한정자에서 `embedded_idl #import "사용하여"` 이벤트 원본 형식 라이브러리가 특성 생성 코드가 유지된 .tlh 파일에 기록되도록 하는 것입니다.

이벤트 발생

이벤트를 발생하려면 이벤트 소스 클래스의 키워드로 `_event` 선언된 인터페이스에서 메서드를 호출합니다. 처리기가 이벤트에 후크된 경우 처리기가 호출됩니다.

COM 이벤트 코드

다음 예제에서는 COM 클래스에서 이벤트를 발생시키는 방법을 보여 줍니다. 예제를 컴파일 및 실행하려면 코드의 주석을 참조하십시오.

C++

```
// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
    [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;
```

서버는 다음과 같습니다.

C++

```
// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-
```

```

B8A1CEC98830" ];

[ coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        __raise MyEvent(123);
        return S_OK;
    }
};

```

클라이언트는 다음과 같습니다.

C++

```

// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
    HRESULT MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
        return S_OK;
    }

    HRESULT MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
        return S_OK;
    }

    void HookEvent(IEventSource* pSource) {
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void UnhookEvent(IEventSource* pSource) {
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    // Create COM object

```

```

CoInitialize(NULL);
{
    IEventSource* pSource = 0;
    HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL,
        CLSCTX_ALL, __uuidof(IEventSource), (void **) &pSource);
    if (FAILED(hr)) {
        return -1;
    }

    // Create receiver and fire event
    CReceiver receiver;
    receiver.HookEvent(pSource);
    pSource->FireEvent();
    receiver.UnhookEvent(pSource);
}
CoUninitialize();
return 0;
}

```

출력

Output

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

레이아웃 종속 COM 이벤트

레이아웃 종속성은 COM 프로그래밍에만 문제가 됩니다. 네이티브 및 관리되는 이벤트 처리에서 처리기의 서명(반환 형식, 호출 규칙 및 인수)은 해당 이벤트와 일치해야 하지만 처리기 이름은 해당 이벤트와 일치할 필요가 없습니다.

그러나 COM 이벤트 처리에서 매개 변수 `event_receiver` 를 `true layout_dependent />`로 설정하면 이름과 서명 일치가 적용됩니다. 이벤트 수신기 및 후크 이벤트에 있는 처리기의 이름과 서명은 정확히 일치해야 합니다.

호출 규칙 및 스토리지 클래스(가상, 정적 등)로 설정 `false` 되면 `Layout_dependent` 발생 이벤트 메서드와 후킹 메서드(대리자) 간에 혼합 및 일치시킬 수 있습니다. 을 갖는 `Layout_dependent = true` 것이 약간 더 효율적입니다.

예를 들어, `IEventSource` 가 다음 메서드를 사용하도록 정의되어 있다고 가정합니다.

C++

```
[id(1)] HRESULT MyEvent1([in] int value);
```

```
[id(2)] HRESULT MyEvent2([in] int value);
```

이벤트 소스의 형태가 다음과 같다고 가정합니다.

C++

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

그런 다음 이벤트 수신기에서 `IEventSource`의 메서드에 후크된 처리기가 다음과 같이 해당 이름과 시그니처를 일치시켜야 합니다.

C++

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
    HRESULT MyEvent1(int nValue) { // name and signature matches MyEvent1
        ...
    }
    HRESULT MyEvent2(E c, char* pc) { // signature doesn't match MyEvent2
        ...
    }
    HRESULT MyHandler1(int nValue) { // name doesn't match MyEvent1 (or 2)
        ...
    }
    void HookEvent(IEventSource* pSource) {
        __hook(IFace, pSource); // Hooks up all name-matched events
                               // under layout_dependent = true
        __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1); // valid
        __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2); // not valid
        __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
    }
};
```

참고 항목

[이벤트 처리](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

Microsoft 전용 한정자

아티클 • 2024. 11. 21.

이 단원에서는 다음 영역의 Microsoft 전용 C++ 확장에 대해 설명합니다.

- [기반 주소 지정](#), 다른 포인터를 오프셋할 수 있는 기준으로 포인터를 사용하는 방법
- [함수 호출 규칙](#)
- [_declspec 키워드로 선언된 확장 스토리지 클래스 특성](#)
- [_w64 키워드](#)

Microsoft 관련 키워드

Microsoft 전용 키워드 중 대다수는 파생 형식을 형성하는 선언자를 수정하는 데 사용될 수 있습니다. 선언자에 대한 자세한 내용은 선언자를 참조 [하세요](#).

테이블 확장

키워드	의미	파생 형식을 만드는 데 사용됩니까?
<code>_based</code>	뒤에 오는 이름이 32비트 오프셋을 선언에 포함된 32비트 기준으로 선언합니다.	예
<code>_cdecl</code>	뒤에 오는 이름이 C 명명 및 호출 규칙을 사용합니다.	예
<code>_declspec</code>	뒤에 오는 이름이 Microsoft 전용 스토리지 클래스 특성을 지정합니다.	아니요
<code>_fastcall</code>	뒤에 오는 이름이 인수 전달 시 가능하면 스택 대신 레지스터를 사용하는 함수를 선언합니다.	예
<code>_restrict</code>	<code>_declspec(제한)</code> 은 비슷하지만 변수에 사용합니다.	아니요
<code>_stdcall</code>	뒤에 오는 이름이 표준 호출 규칙을 준수하는 함수를 지정합니다.	예
<code>_w64</code>	64비트 컴파일러에서 더 큰 형식으로 데이터 형식을 표시합니다.	아니요
<code>_unaligned</code>	형식 또는 다른 데이터의 포인터가 정렬되지 않음을 지정합니다.	아니요
<code>_vectorcall</code>	뒤에 오는 이름이 인수 전달 시 가능하면 스택 대신 SSE 레지스터 등의 레지스터를 사용하는 함수를 선언합니다.	예

참고 항목

C++ 언어 참조

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

기반 주소

아티클 • 2023. 10. 12.

이 단원에 포함된 항목은 다음과 같습니다.

- [_based 문법](#)
- [기반 포인터](#)

참고 항목

[Microsoft 전용 한정자](#)

`__based` 문법

아티클 • 2024. 11. 21.

Microsoft 전용

기반 주소 지정은 개체가 할당된(정적 및 동적 기반 데이터) 세그먼트를 정밀하게 제어해야 할 때 유용합니다.

32비트와 64비트 컴파일이 호환되는 유일한 주소 지정 방식은 32비트/64비트 기반에 대해 32비트/64비트 치환을 포함하거나 `void`를 기반으로 하는 형식을 정의하는 "포인터 기반"입니다.

문법

:

`__based(base-expression)`

:

`based-variable based-abstract-declarator segment-name segment-cast`

:

`identifier`

:

`abstract-declarator`

:

`type-name`

Microsoft 전용 종료

참고 항목

[기반 포인터](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

기반 포인터 (C++)

아티클 • 2024. 11. 21.

키 `_based` 워드를 사용하면 포인터(기준 포인터의 오프셋인 포인터)를 기반으로 포인터를 선언할 수 있습니다. 키워드는 `_based` Microsoft 전용입니다.

구문

```
type _based( base ) declarator
```

설명

포인터 주소를 기반으로 하는 포인터는 32비트 또는 64비트 컴파일에서 유효한 키워드의 `_based` 유일한 형태입니다. Microsoft 32비트 C/C++ 컴파일러의 기반 포인터는 32비트 포인터 기반에서 오프셋된 32비트입니다. 이 제한과 유사하게 64비트 환경에서 기반 포인터는 64비트 기반에서 오프셋된 64비트입니다.

포인터 기반의 포인터는 포인터가 포함된 영구 식별자에 사용됩니다. 포인터 기반의 포인터로 구성된 연결 목록은 디스크에 저장한 다음, 유효한 포인터를 이용하여 메모리의 다른 장소로 다시 로드할 수 있습니다. 예시:

```
C++

// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void _based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

`vpBuffer` 포인터는 나중에 프로그램에 할당된 메모리 주소로 할당됩니다. 연결 목록은 `vpBuffer` 값을 기준으로 재배치됩니다.

① 참고

메모리 매핑된 파일을 사용하여 포인터를 포함하는 식별자 유지를 수행할 수도 있습니다.

기반 포인터를 역참조하는 경우 기반은 반드시 명시적으로 지정하거나 선언을 통해 암시적으로 알려야 합니다.

이전 버전과의 호환성을 위해 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)를 지정하지 않는 한 `_based` 동의어 `__based` 입니다.

예시

다음 코드는 기반 변경을 통한 기반 포인터 변경을 설명합니다.

```
C++  
  
// based_pointers2.cpp  
// compile with: /EHsc  
#include <iostream>  
  
int a1[] = { 1,2,3 };  
int a2[] = { 10,11,12 };  
int *pBased;  
  
typedef int __based(pBased) * pBasedPtr;  
  
using namespace std;  
int main() {  
    pBased = &a1[0];  
    pBasedPtr pb = 0;  
  
    cout << *pb << endl;  
    cout << *(pb+1) << endl;  
  
    pBased = &a2[0];  
  
    cout << *pb << endl;  
    cout << *(pb+1) << endl;  
}
```

Output

```
1  
2  
10  
11
```

참고 항목

[키워드](#)

[alloc_text](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

호출 규칙

아티클 • 2023. 10. 12.

Visual C/C++ 컴파일러는 내부 및 외부 함수 호출에 몇 가지 다양한 규칙을 제공합니다. 이러한 다양한 접근 방법에 대한 이해는 프로그램을 디버깅하고 어셈블리 언어 루틴에 코드를 연결하는 데 도움이 됩니다.

이 주제의 항목에서는 호출 규칙간의 차이점, 인수를 전달하는 방식, 함수가 값을 반환하는 방식에 대해 설명합니다. 또한 고급 기능인 naked 함수 호출에 대한 설명을 통해 사용자 지정 프롤로그 및 에필로그 코드를 작성할 수 있도록 합니다.

x64 프로세서에 대한 호출 규칙에 대한 자세한 내용은 호출 규칙을 [참조하세요](#).

이 섹션의 토픽

- [인수 전달 및 명명 규칙](#) (`__cdecl`, `__stdcall`, `__fastcall` 및 기타)
- [호출 예제: 함수 프로토타입 및 호출](#)
- [naked 함수 호출을 사용하여 사용자 지정 프롤로그/에필로그 코드 작성](#)
- [부동 소수점 보조 프로세서 및 호출 규칙](#)
- [사용되지 않는 호출 규칙](#)

참고 항목

[Microsoft 전용 한정자](#)

인수 전달 및 명명 규칙

아티클 • 2023. 10. 12.

Microsoft 전용

Microsoft C++ 컴파일러를 사용하면 함수와 호출자 간에 인수를 전달하고 값을 반환하기 위한 규칙을 지정할 수 있습니다. 지원되는 모든 플랫폼에서 모든 규칙을 사용할 수 있는 것은 아니며, 일부 규칙은 플랫폼별 구현을 사용합니다. 대부분의 경우 특정 플랫폼에서 지원되지 않는 규칙을 지정하는 키워드 또는 컴파일러 스위치는 무시되며 플랫폼 기본 규칙이 사용됩니다.

x86 플랫폼에서는 모든 인수가 전달되면 32비트로 확장됩니다. 반환 값도 32비트로 확장되며 EDX:EAX 레지스터 쌍에서 반환되는 8바이트 구조체를 제외하고 EAX 레지스터에서 반환됩니다. 더 큰 구조체는 숨겨진 반환 구조체에 대한 포인터로 EAX 레지스터에서 반환됩니다. 매개 변수는 오른쪽에서 왼쪽으로 스택에 푸시됩니다. POD가 아닌 구조체는 레지스터에서 반환되지 않습니다.

컴파일러는 ESI, EDI, EBX 및 EBP 레지스터가 함수에서 사용되는 경우 이러한 레지스터를 저장하고 복원하는 프롤로그 및 에필로그 코드를 생성합니다.

① 참고

구조체, 공용 구조체 또는 클래스가 값으로 함수에서 반환되는 경우 형식의 모든 정의가 동일해야 하며, 그렇지 않으면 프로그램이 런타임에 실패할 수 있습니다.

고유한 함수 프롤로그 및 에필로그 코드를 정의하는 방법에 대한 자세한 내용은 Naked 함수 호출을 [참조하세요](#).

x64 플랫폼을 대상으로 하는 코드의 기본 호출 규칙에 대한 자세한 내용은 x64 호출 규칙을 [참조하세요](#). ARM 플랫폼을 대상으로 하는 코드에서 규칙 문제를 호출하는 방법에 대한 자세한 내용은 일반적인 Visual C++ ARM 마이그레이션 문제를 [참조하세요](#).

다음과 같은 호출 규칙이 Visual C/C++ 컴파일러에서 지원됩니다.

키워드	스택 정리	매개 변수 전달
_cdecl	호출자	매개 변수를 스택에 역순으로(오른쪽에서 왼쪽으로) 푸시합니다.
_clrcall	해당 없음	CLR 식 스택에 매개 변수를 순서대로(왼쪽에서 오른쪽으로) 로드합니다.
_stdcall	호출 수신자	매개 변수를 스택에 역순으로(오른쪽에서 왼쪽으로) 푸시합니다.

키워드	스택 정리	매개 변수 전달
<code>_fastcall</code>	호출 수신자	레지스터에 저장된 다음 스택에 푸시됩니다.
<code>_thiscall</code>	호출 수신자	스택에 푸시됨; <code>this</code> ECX에 저장된 포인터
<code>_vectorcall</code>	호출 수신자	레지스터에 저장된 다음 스택에 역순으로(오른쪽에서 왼쪽으로) 푸시 됩니다.

관련 정보는 사용되지 않는 호출 규칙을 참조 [하세요](#).

Microsoft 전용 종료

참고 항목

[호출 규칙](#)

_cdecl

아티클 • 2024. 11. 21.

_cdecl 는 C 및 C++ 프로그램에 대한 기본 호출 규칙입니다. 스택은 호출자에 의해 정리되므로 함수를 수행할 `vararg` 수 있습니다. 호출 규칙은 **_cdecl** 스택 정리 코드를 포함하기 위해 각 함수 호출이 필요하기 때문에 **_stdcall** 보다 큰 실행 파일을 만듭니다. 다음 목록에서는 이러한 호출 규칙의 구현을 보여 줍니다. **_cdecl** 한정자는 Microsoft 전용입니다.

테이블 확장

요소	구현
인수 전달 순서	오른쪽에서 왼쪽
스택 유지 관리 책임	호출하는 함수가 스택에서 인수를 꺼냅니다.
이름 데코레이션 규칙	C 링크를 사용하는 _cdecl 함수를 내보낼 경우를 제외하고 밑줄 문자(_)는 이름 앞에 접두사로 지정됩니다.
대/소문자 변환 규칙	대/소문자 변환은 수행되지 않습니다.

① 참고

관련 정보는 데코레이팅된 이름을 참조 [하세요](#).

_cdecl 변수 또는 함수 이름 앞에 한정자를 배치합니다. C 명명 및 호출 규칙이 기본값이므로 x86 코드에서 사용해야 **_cdecl** 하는 유일한 시간은 (vectorcall), (stdcall) `/Gz` 또는 `/Gr` (fastcall) 컴파일러 옵션을 지정한 `/Gv` 경우입니다. `/Gd` 컴파일러 옵션은 호출 규칙을 강제로 실행합니다 **_cdecl**.

ARM 및 x64 프로세서 **_cdecl** 에서는 허용되지만 일반적으로 컴파일러에서 무시됩니다. ARM 및 x64에서는 규칙에 따라 인수는 가능한 경우 레지스터로 전달되고 이후 인수는 스택에 전달됩니다. x64 코드 **_cdecl** 에서 `/Gv` 컴파일러 옵션을 재정의하고 기본 x64 호출 규칙을 사용합니다.

비정적 클래스 함수의 경우 함수가 아웃오브 라인으로 정의되면 호출 규칙 한정자를 아웃오브 라인 정의에서 지정하지 않아도 됩니다. 즉, 클래스 비정적 멤버 메서드의 경우 선언하는 동안 지정된 호출 규칙이 정의 시 가정됩니다. 다음의 클래스 정의를 가정해 봅니다.

C++

```
struct CMyClass {  
    void __cdecl mymethod();  
};
```

다음 코드는

C++

```
void CMyClass::mymethod() { return; }
```

다음 코드 조각과 일치합니다.

C++

```
void __cdecl CMyClass::mymethod() { return; }
```

이전 버전과의 호환성을 위해 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)를 지정하지 않는 한 `cdecl` 및 `_cdecl` 동의어 `__cdecl`입니다.

예시

다음 예제에서 컴파일러는 `system` 함수에 대해 C 명명 규칙 및 호출 규칙을 사용하도록 지시되었습니다.

C++

```
// Example of the __cdecl keyword on function  
int __cdecl system(const char *);  
// Example of the __cdecl keyword on function pointer  
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD  
flags, ...);
```

참고 항목

[인수 전달 및 명명 규칙](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

_clrcall

아티클 • 2024. 11. 21.

관리 코드에서만 함수를 호출할 수 있도록 지정합니다. **관리 코드에서만 호출되는 모든 가상 함수에 `_clrcall` 사용합니다.** 그러나 네이티브 코드에서 호출되는 함수에는 이 호출 규칙을 사용할 수 없습니다. `_clrcall` 한정자는 Microsoft 전용입니다.

`_clrcall` 사용하여 관리되는 함수에서 가상 관리 함수로 호출하거나 관리되는 함수에서 포인터를 통해 관리되는 함수로 호출할 때 성능을 향상시킵니다.

진입점은 컴파일러에서 생성된 별도의 함수입니다. 함수에 네이티브 진입점과 관리되는 진입점이 둘 다 있을 경우 둘 중 하나는 함수 구현이 포함된 실제 함수입니다. 다른 함수는 실제 함수를 호출하고 공용 언어 런타임에서 PInvoke를 수행하게 하는 별도의 함수(펑크)입니다. 함수를 `_clrcall` 표시할 때 함수 구현이 MSIL이어야 하며 네이티브 진입점 함수가 생성되지 않음을 나타냅니다.

`_clrcall` 지정되지 않은 경우 네이티브 함수의 주소를 사용하는 경우 컴파일러는 네이티브 진입점을 사용합니다. `_clrcall` 함수가 관리되며 관리되는 함수에서 네이티브로 전환 할 필요가 없음을 나타냅니다. 이 경우 컴파일러가 관리되는 진입점을 사용합니다.

(사용 안 `/clr:pure` 됨 또는 `/clr:safe`)이 **사용되고** `_clrcall` 사용되지 않는 경우 `/clr` 함수의 주소를 사용하면 항상 네이티브 진입점 함수의 주소가 반환됩니다. `_clrcall` 사용하면 네이티브 진입점 함수가 만들어지지 않으므로 진입점 펑크 함수가 아닌 관리되는 함수의 주소를 가져옵니다. 자세한 내용은 Double Thunking을 참조하세요. `/clr:pure` 및 `/clr:safe` 컴파일러 옵션은 Visual Studio 2015에서 더 이상 사용되지 않으며 Visual Studio 2017에서는 지원되지 않습니다.

`/clr(공용 언어 런타임 컴파일)`은 모든 함수 및 함수 포인터가 `_clrcall` 컴파일러에서 컴파일러 내의 함수가 `_clrcall` 이외의 것으로 표시되도록 허용하지 않음을 의미합니다.

`/clr:pure`을 사용하는 경우 함수 포인터 및 외부 선언에서만 `_clrcall` 지정할 수 있습니다.

해당 함수에 MSIL 구현이 있는 한 **`/clr`을 사용하여** 컴파일된 기존 C++ 코드에서 `_clrcall` 함수를 직접 호출 할 수 있습니다. `_clrcall` 함수는 인라인 asm이 있고 CPU 관련 내장 함수를 호출하는 함수에서 직접 호출할 수 없습니다(예: 해당 함수가 컴파일된 `/clr` 경우에도).

`_clrcall` 함수 포인터는 생성된 애플리케이션 도메인에서만 사용됩니다. 애플리케이션 도메인 [CrossAppDomainDelegate](#)에 `_clrcall` 함수 포인터를 전달하는 대신 사용합니다. 자세한 내용은 [애플리케이션 도메인 및 Visual C++를 참조하세요](#).

예제

함수가 `_clrcall` 사용하여 선언되면 필요할 때 코드가 생성됩니다(예: 함수가 호출될 때).

C++

```
// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
    Console::WriteLine("in Func1");
    return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the address of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf(); // comment this line and comparison will pass

int main() {
    if (&Func1 == pf)
        Console::WriteLine("&Func1 == pf, comparison succeeds");
    else
        Console::WriteLine("&Func1 != pf, comparison fails");

    // even though comparison fails, stub and function call are correct
    pf();
    Func1();
}
```

Output

```
in Func1
&Func1 != pf, comparison fails
in Func1
in Func1
```

다음 샘플에서는 함수 포인터를 정의하고 관리 코드에서만 함수 포인터를 호출할 수 있도록 선언합니다. 여기서는 컴파일러가 관리되는 함수를 직접 호출하고 네이티브 진입점(이중 쟁크 문제)을 방지할 수 있습니다.

C++

```
// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
```

```
void (*pTest)() = &Test;  
(*pTest)();  
  
void (__cdecl *pTest2)() = &Test;  
(*pTest2)();  
}
```

참고 항목

[인수 전달 및 명명 규칙](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__stdcall

아티클 • 2024. 07. 08.

__stdcall 호출 규칙은 Win32 API 함수를 호출하는 데 사용됩니다. 호출 수신자가 스택을 정리하므로 컴파일러는 `vararg` 함수를 `__cdecl`로 만듭니다. 이 호출 규칙을 사용하는 함수에는 함수 프로토타입이 필요합니다. `__stdcall` 한정자는 Microsoft 전용입니다.

구문

`return-type __stdcall function-name[(argument-list)]`

설명

다음 목록에서는 이러한 호출 규칙의 구현을 보여 줍니다.

[+] 테이블 확장

요소	구현
인수 전달 순서	오른쪽에서 왼쪽
인수 전달 규칙	포인터 또는 참조 형식이 전달되지 않는 경우 값으로 전달
스택 유지 관리 책임	호출된 함수가 스택에서 자신의 인수를 꺼냅니다.
이름 데코레이션 규칙	밑줄(_)이 이름 앞에 붙습니다. 이름 뒤에는 기호(@)가 오고 그 위에 인수 목록의 바이트 수(10진수)가 옵니다. 따라서 <code>int func(int a, double b)</code> 로 선언된 함수는 <code>_func@12</code> 로 데코레이팅됩니다.
대/소문자 변환 규칙	None

[/Gz](#) 컴파일러 옵션은 다른 호출 규칙으로 명시적으로 선언되지 않은 모든 함수에 대해 `__stdcall`을 지정합니다.

이전 버전과의 호환성을 위해 `__stdcall`은 `__stdcall`의 동의어입니다. 단 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

`__stdcall` 한정자를 사용하여 선언된 함수는 `__cdecl`을 사용하여 선언된 함수와 동일한 방식으로 값을 반환합니다.

ARM 및 x64 프로세서에서는 `__stdcall`이 컴파일러에 의해 허용되고 무시됩니다. ARM 및 x64 아키텍처에서는 규칙에 따라 가능한 경우 인수가 레지스터로 전달되고 이후 인수는 스택에 전달됩니다.

비정적 클래스 함수의 경우 함수가 아웃오브 라인으로 정의되면 호출 규칙 한정자를 아웃오브 라인 정의에서 지정하지 않아도 됩니다. 즉, 클래스 비정적 멤버 메서드의 경우 선언하는 동안 지정된 호출 규칙이 정의 시 가정됩니다. 다음과 같은 클래스 정의가 주어진 경우

C++

```
struct CMyClass {
    void __stdcall mymethod();
```

this

C++

```
void CMyClass::mymethod() { return; }
```

다음 코드와 동일합니다.

C++

```
void __stdcall CMyClass::mymethod() { return; }
```

예시

다음 예에서는 `__stdcall`을 사용한 결과 모든 `WINAPI` 함수 형식이 표준 호출로 처리됩니다.

C++

```
// Example of the __stdcall keyword
#define WINAPI __stdcall
// Example of the __stdcall keyword on function pointer
typedef BOOL (__stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD
flags, ...);
```

참고 항목

인수 전달 및 명명 규칙

키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

fastcall

아티클 • 2024. 11. 21.

Microsoft 전용

호출 규칙은 `_fastcall` 가능한 경우 함수에 대한 인수를 레지스터에 전달하도록 지정합니다. 이 호출 규칙은 x86 아키텍처에만 적용됩니다. 다음 목록에서는 이러한 호출 규칙의 구현을 보여 줍니다.

[+] 테이블 확장

요소	구현
인수 전달 순서	왼쪽에서 오른쪽으로 인수 목록에 있는 처음 두 <code>DWORD</code> 개 또는 더 작은 인수는 ECX 및 EDX 레지스터에 전달됩니다. 다른 모든 인수는 오른쪽에서 왼쪽으로 스택에 전달됩니다.
스택 유지 관리 책임	호출된 함수가 스택에서 인수를 꺼냅니다.
이름 데코레이션 규칙	기호(@)에 이름 앞에 접두사로 지정됩니다. 매개 변수 목록의 바이트 수(10진수)가 뒤에 붙는 at 기호는 이름에 접미사가 붙습니다.
대/소문자 변환 규칙	대/소문자 변환은 수행되지 않습니다.
클래스, 구조체 및 공용 구조체	크기에 관계없이 "멀티바이트" 형식으로 처리되고 스택에 전달됩니다.
열거형 및 열거형 클래스	레지스터에 의해 기본 형식이 전달되는 경우 레지스터에 의해 전달됩니다. 예를 들어 기본 형식이 <code>int</code> <code>unsigned int</code> 크기가 8, 16 또는 32비트인 경우입니다.

① 참고

이후 컴파일러 버전은 다른 레지스터를 사용하여 매개 변수를 저장할 수도 있습니다.

/Gr 컴파일러 옵션을 사용하면 함수가 충돌하는 특성을 사용하여 선언되거나 함수 이름이 선언되지 않는 한 모듈의 각 함수가 컴파일 `_fastcall` 됩니다 `main`.

`_fastcall` ARM 및 x64 아키텍처를 대상으로 하는 컴파일러에서 키워드를 허용하고 무시합니다. x64 칩에서는 규칙에 따라 가능한 경우 처음 4개의 인수가 레지스터에 전달되고 추가 인수가 스택에 전달됩니다. 자세한 내용은 x64 호출 규칙을 참조 [하세요](#). ARM 칩

의 경우 최대 4개의 정수 인수와 8개의 부동 소수점 인수가 레지스터로 전달될 수 있으며 추가 인수는 스택에 전달됩니다.

비정적 클래스 함수의 경우 함수가 오프라인으로 정의된 경우 호출 규칙 한정자를 아웃 오브 라인 정의에 지정할 필요가 없습니다. 즉, 클래스 비정적 멤버 메서드의 경우 선언하는 동안 지정된 호출 규칙이 정의 시 가정됩니다. 다음의 클래스 정의를 가정해 봅니다.

C++

```
struct CMyClass {
    void __fastcall mymethod();
```

```
};
```

다음 코드는

C++

```
void CMyClass::mymethod() { return; }
```

다음 코드 조각과 일치합니다.

C++

```
void __fastcall CMyClass::mymethod() { return; }
```

이전 버전과의 호환성을 위해 `_fastcall`은(는) `__fastcall`의 동의어입니다. 단, 컴파일러 옵션 [/Za \(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

예시

다음 예제에서 `DeleteAggrWrapper` 함수에는 레지스터로 인수가 전달됩니다.

C++

```
// Example of the __fastcall keyword
#define FASTCALL __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __fastcall keyword on function pointer
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2,
DWORD flags, ...);
```

참고 항목

인수 전달 및 명명 규칙

키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__thiscall

아티클 • 2024. 07. 12.

Microsoft 관련 **__thiscall** 호출 규칙은 x86 아키텍처의 C++ 클래스 멤버 함수에서 사용됩니다. 변수 인수(**vararg** 함수)를 사용하지 않는 멤버 함수에서 사용하는 기본 호출 규칙입니다.

__thiscall에서 호출 수신자는 스택을 정리하므로 **vararg** 함수에는 불가능합니다. 인수는 오른쪽에서 왼쪽으로 스택에 푸시됩니다. **this** 포인터는 스택이 아닌 등록 ECX를 통해 전달됩니다.

ARM, ARM64 및 x64 컴퓨터에서 **__thiscall** 컴파일러에서 허용되고 무시됩니다. 기본적으로 레지스터 기반 호출 규칙을 사용하기 때문입니다.

__thiscall을(를) 사용하는 한 가지 이유는 멤버 함수가 기본적으로 **_clrcall**을(를) 사용하는 클래스에 있기 때문입니다. 이 경우 **__thiscall**을(를) 사용하여 네이티브 코드에서 개별 멤버 함수를 호출할 수 있도록 할 수 있습니다.

/clr:pure을(를) 사용하여 컴파일할 때 달리 지정하지 않으면 모든 함수 및 함수 포인터가 **_clrcall**입니다. **/clr:pure** 및 **/clr:safe** 컴파일러 옵션은 Visual Studio 2015에서 더 이상 사용되지 않으며 Visual Studio 2017에서는 지원되지 않습니다.

vararg 멤버 함수는 **_cdecl** 호출 규칙을 사용합니다. 모든 함수 인수는 스택에 푸시되고 **this** 포인터는 스택에 마지막으로 배치됩니다.

이 호출 규칙은 C++에만 적용되므로 C 이름 장식 체계가 없습니다.

비정적 클래스 멤버 함수를 아웃 오브 라인으로 정의하는 경우 선언에서만 호출 규칙 한정자를 지정합니다. 아웃 오브 라인 정의에서 다시 지정할 필요가 없습니다. 컴파일러는 정의 지점에서 선언하는 동안 지정된 호출 규칙을 사용합니다.

참고 항목

[인수 전달 및 명명 규칙](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

vectorcall

아티클 • 2023. 04. 03.

Microsoft 전용

호출 규칙은 `_vectorcall` 가능한 경우 함수에 대한 인수를 레지스터에 전달하도록 지정합니다. `_vectorcall`는 인수에 대해 기본 [x64 호출 규칙](#) 사용보다 `_fastcall` 더 많은 레지스터를 사용합니다. `_vectorcall` 호출 규칙은 SSE2(스트리밍 SIMD 확장명 2) 이상을 포함하는 x86 및 x64 프로세서의 네이티브 코드에서만 지원됩니다. 여러 부동 소수점 또는 SIMD 벡터 인수를 전달하는 함수의 속도를 조정하고 레지스터에 로드된 인수를 활용하는 작업을 수행하는 데 사용합니다 `_vectorcall`. 다음 목록에서는 x86 및 x64 구현에 공통적인 기능을 보여 줍니다 `_vectorcall`. 차이점은 이 문서의 뒷부분에 설명되어 있습니다.

요소	구현
C 이름 데코레이션 규칙	함수 이름은 두 개의 "at" 기호(@@)가 접미사로 붙고 그 뒤에 매개 변수 목록의 바이트 수(10진수)가 이어집니다.
대/소문자 변환 규칙	대/소문자 변환은 수행되지 않습니다.

`/Gv` 컴파일러 옵션을 사용하면 함수가 멤버 함수가 아니면 모듈의 각 함수가 컴파일되거나, 충돌하는 호출 규칙 특성으로 `_vectorcall` 선언되거나, 변수 인수 목록을 사용 `vararg`하거나, 이름이 `main` 있는 경우가 아니면 컴파일됩니다.

함수에 `_vectorcall` 등록하여 정수 형식 값, 벡터형식 값 및 HVA(동종 벡터 집계) 값의 세 가지 종류의 인수를 전달할 수 있습니다.

정수 형식은 프로세서의 네이티브 레지스터 크기(예: x86 컴퓨터의 4바이트 또는 x64 컴퓨터의 경우 8바이트)에 적합하며 비트 표현을 변경하지 않고 레지스터 길이의 정수로 다시 변환할 수 있습니다. 예를 들어 x86(`long long` x64)에서 승격 `int`하거나(예 `char short`: x64에서) 원래 형식으로 캐스팅 `int long long` 할 수 있는 모든 형식은 정수 형식입니다. 정수 형식에는 포인터, 참조 및 `struct union` 4바이트(x64의 경우 8바이트) 이하의 형식이 포함됩니다. x64 플랫폼에서는 더 크고 `struct union` 형식이 호출자가 할당한 메모리에 대한 참조로 전달됩니다. x86 플랫폼에서는 스택의 값으로 전달됩니다.

벡터 형식은 부동 소수점 형식(예: 또는 `float double` SIMD 벡터 형식) `_m128 _m256`입니다.

HVA 형식은 동일한 벡터 형식을 갖는 최대 4개의 데이터 멤버로 구성된 복합 형식입니다. HVA 형식은 멤버의 벡터 형식과 맞춤 요구 사항이 동일합니다. 세 가지 동일한 벡터 형식

을 포함하고 32바이트 맞춤이 있는 HVA `struct` 정의의 예입니다.

C++

```
typedef struct {
    __m256 x;
    __m256 y;
    __m256 z;
} hva3;      // 3 element HVA type on __m256
```

별도로 컴파일된 코드가 오류 없이 연결할 수 있도록 헤더 파일의 키워드를 사용하여 함수를 명시적으로 `_vectorcall` 선언합니다. 함수를 사용 `_vectorcall`하려면 프로토 타입이어야 하며 가변 길이 인수 목록을 사용할 `vararg` 수 없습니다.

멤버 함수는 지정자를 사용하여 `_vectorcall` 선언할 수 있습니다. 숨겨진 `this` 포인터는 레지스터에 의해 첫 번째 정수 형식 인수로 전달됩니다.

ARM 컴퓨터에서 `_vectorcall` 컴파일러에서 허용되고 무시됩니다. ARM64EC에서 `_vectorcall` 컴파일러에서 지원되지 않고 거부됩니다.

비정적 클래스 멤버 함수의 경우 함수가 아웃오브 라인으로 정의되면 호출 규칙 한정자 를 아웃오브 라인 정의에서 지정하지 않아도 됩니다. 즉, 클래스 비정적 멤버의 경우 선언하는 동안 지정된 호출 규칙이 정의 시에 가정됩니다. 다음의 클래스 정의를 가정해 봅니다.

C++

```
struct MyClass {
    void _vectorcall mymethod();
};
```

다음 코드는

C++

```
void MyClass::mymethod() { return; }
```

다음 코드 조각과 일치합니다.

C++

```
void _vectorcall MyClass::mymethod() { return; }
```

함수에 `__vectorcall` 대한 포인터 `__vectorcall` 를 만들 때 호출 규칙 한정자를 지정해야 합니다. 다음 예제에서는 네 `double` 개의 인수를 사용하고 값을 반환하는 함수에 `__vectorcall` 대한 포인터를 `_m256` 만듭니다 `typedef`.

C++

```
typedef _m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

이전 버전 `__vectorcall` 과의 호환성을 위해 `__vectorcall` 컴파일러 옵션 [/Za](#) (언어 확장 사용 안 함) 을 지정하지 않는 한 동의어입니다.

x64의 `__vectorcall` 규칙

`__vectorcall` x64의 호출 규칙은 표준 x64 호출 규칙을 확장하여 추가 레지스터를 활용합니다. 정수 형식 인수와 벡터 형식 인수 모두 인수 목록에 있는 위치를 기반으로 레지스터에 매핑됩니다. HVA 인수는 사용되지 않는 벡터 레지스터에 할당됩니다.

왼쪽에서 오른쪽 순서로 처음 네 개의 인수가 정수 형식 인수인 경우 해당 위치 RCX, RDX, R8 또는 R9에 해당하는 레지스터에서 전달됩니다. 숨겨진 `this` 포인터는 첫 번째 정수 형식 인수로 처리됩니다. 처음 네 인수 중 하나의 HVA 인수를 사용 가능한 레지스터에 전달 할 수 없는 경우 호출자 할당 메모리에 대한 참조가 해당 정수 형식 레지스터에 전달됩니다. 네 번째 매개 변수 위치 뒤에 오는 정수 형식 인수는 스택에서 전달됩니다.

왼쪽에서 오른쪽 순서로 처음 여섯 개의 인수가 벡터 형식 인수인 경우 인수 위치에 따라 0에서 5의 SSE 벡터 레지스터 값에 의해 전달됩니다. 부동 소수점 및 `_m128` 형식은 XMM 레지스터에 전달되고 `_m256` 형식은 YMM 레지스터에 전달됩니다. 벡터 형식이 참조가 아닌 값에 의해 전달되고 추가 레지스터가 사용되므로 이는 표준 x64 호출 규칙과 다릅니다. 벡터 형식 인수에 할당된 새도 스택 공간은 8바이트 단위로 고정되며 [/homeparams](#) 옵션이 적용되지 않습니다. 일곱 번째 이후의 매개 변수 자리에 오는 벡터 형식 인수는 호출자가 할당한 메모리에 대한 참조에 의해 스택에서 전달됩니다.

벡터 인수에 대해 레지스터가 할당된 후 HVA 인수의 데이터 멤버는 전체 HVA에 사용할 수 있는 충분한 레지스터가 있는 한 사용하지 않는 벡터에 XMM0을 XMM5(또는 형식의 경우 YMM0에서 YMM5로) 등록하기 위해 `_m256` 오름차순으로 할당됩니다. 사용할 수 있는 레지스터가 충분하지 않은 경우 HVA 인수는 호출자가 할당한 메모리에 대한 참조에 의해 전달됩니다. HVA 인수에 대한 스택 새도 공간은 정의되지 않은 내용을 포함하여 8바이트로 고정됩니다. HVA 인수는 매개 변수 목록의 왼쪽에서 오른쪽 순서로 레지스터에 할당되며 어떤 위치에나 있을 수 있습니다. 벡터 레지스터에 할당되지 않은 처음 네 개의 인수 위치 중 하나에 있는 HVA 인수는 해당 위치에 대응하는 정수 레지스터에서 참조에 의해 전달됩니다. 네 번째 매개 변수 위치 뒤로 참조에 의해 전달된 HVA 인수는 스택에서 푸시됩니다.

함수의 `__vectorcall` 결과는 가능한 경우 레지스터의 값으로 반환됩니다. 8바이트 이하의 정수 형식 구조체 또는 공용 구조체를 포함한 정수 형식의 결과는 RAX에 값으로 반환됩니다. 벡터 형식 결과는 크기에 따라 XMM0 또는 YMM0에 값으로 반환됩니다. HVA 결과에는 요소 크기에 따라 레지스터 XMM0:XMM3 또는 YMM0:YMM3에 값으로 반환되는 각 데이터 요소가 있습니다. 해당 레지스터에 맞지 않는 결과 형식은 호출자가 할당한 메모리에 대한 참조로 반환됩니다.

스택은 호출자가 x64 구현 `__vectorcall`에서 유지 관리합니다. 호출자 프롤로그 및 에필로그 코드는 호출된 함수의 스택을 할당하고 호출합니다. 인수는 오른쪽에서 왼쪽으로 스택에서 푸시되며 새도 스택 공간에는 레지스터에 의해 전달된 인수가 할당됩니다.

예:

C++

```
// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
// This example creates an annotated assembly listing in
// crt_vc64.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;      // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;      // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
// Passes b by element in [XMM0:XMM1];
```

```

// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0,YMM2,YMM4,YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    f = example4(1, 2.0f, h4, d, 5);
    i = example5(1, h2, 3, h4, 5);
    h4 = example6(h2, h4, c, h2);
}

```

x86의 `_vectorcall` 규칙

호출 규칙은 `_vectorcall` 32비트 정수 형식 인수에 대한 규칙을 따르고 `_fastcall` 벡터 형식 및 HVA 인수에 대한 SSE 벡터 레지스터를 활용합니다.

매개 변수 목록에서 왼쪽에서 오른쪽 순서로 처음 두 개 정수 형식 인수는 각각 ECX 및 EDX에 배치됩니다. 숨겨진 `this` 포인터는 첫 번째 정수 형식 인수로 처리되고 ECX로 전달됩니다. 처음 여섯 개 벡터 형식 인수는 인수 크기에 따라 0에서 5의 SSE 벡터 레지스터 값에 의해 XMM 또는 YMM 레지스터에서 전달됩니다.

왼쪽에서 오른쪽 순서로 처음 여섯 개 벡터 형식 인수는 0에서 5의 SSE 벡터 레지스터 값에 의해 전달됩니다. 부동 소수점 및 `_m128` 형식은 XMM 레지스터에 전달되고 `_m256` 형식은 YMM 레지스터에 전달됩니다. 레지스터에 의해 전달된 벡터 형식 인수에는 쇄도 스택 공간이 할당되지 않습니다. 일곱 번째 이후의 벡터 형식 인수는 호출자가 할당한 메모리에 대한 참조에 의해 스택에서 전달됩니다. 컴파일러 오류 [C2719](#)의 제한 사항은 이러한 인수에 적용되지 않습니다.

벡터 인수에 대해 레지스터가 할당된 후 HVA 인수의 데이터 멤버는 전체 HVA에 사용할 수 있는 충분한 레지스터가 있는 한 사용하지 않는 벡터 레지스터 XMM0에서 XMM5(또는 형식의 경우 `_m256` YMM0에서 YMM5로)에 오름차순으로 할당됩니다. 사용할 수 있는 레지스터가 충분하지 않은 경우 HVA 인수는 호출자가 할당한 메모리에 대한 참조에 의해 스택에서 전달됩니다. HVA 인수에는 스택 쇄도 공간이 할당되지 않습니다. HVA 인수는 매개 변수 목록의 왼쪽에서 오른쪽 순서로 레지스터에 할당되며 어떤 위치에나 있을 수 있습니다.

함수의 `_vectorcall` 결과는 가능한 경우 레지스터의 값으로 반환됩니다. 4바이트 이하의 정수 형식 구조체 또는 공용 구조체를 포함한 정수 형식의 결과는 EAX에 값으로 반환됩니다. 8바이트 이하의 정수 형식 구조체 또는 공용 구조체는 EDX:EAX에 값으로 반환됩니다. 벡터 형식 결과는 크기에 따라 XMM0 또는 YMM0에 값으로 반환됩니다. HVA 결과에는 요소 크기에 따라 레지스터 XMM0:XMM3 또는 YMM0:YMM3에 값으로 반환되는 각 데이터 요소가 있습니다. 다른 결과 형식은 호출자가 할당한 메모리에 대한 참조로 반환됩니다.

x86 구현 `_vectorcall`은 호출자가 오른쪽에서 왼쪽으로 스택에 푸시하는 인수의 규칙을 따르며 호출된 함수는 스택이 반환되기 직전에 스택을 지웁니다. 레지스터에 배치되지 않은 인수만 스택으로 푸시됩니다.

예:

```

// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;      // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;      // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {

```

```

    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    f = example4(1, 2.0f, h4, d, 5);
    i = example5(1, h2, 3, h4, 5);
    h4 = example6(h2, h4, c, h2);
}

```

Microsoft 특정 종료

참고 항목

[인수 전달 및 명명 규칙](#)

[C++ 키워드](#)

호출 예제: 함수 프로토타입 및 호출

아티클 • 2023. 10. 12.

Microsoft 전용

다음 예제에서는 다양한 호출 규칙을 사용하여 함수를 호출한 결과를 보여 줍니다.

이 예제는 다음과 같은 함수 구조를 기반으로 합니다. `calltype`을 적절한 호출 규칙으로 바꾸십시오.

C++

```
void    calltype MyFunc( char c, short s, int i, double f );
.
.
.

void    MyFunc( char c, short s, int i, double f )
{
.
.
.
}

MyFunc ( 'x', 12, 8192, 2.7183);
```

자세한 내용은 호출 예제의 결과를 참조 [하세요](#).

Microsoft 전용 종료

참고 항목

[호출 규칙](#)

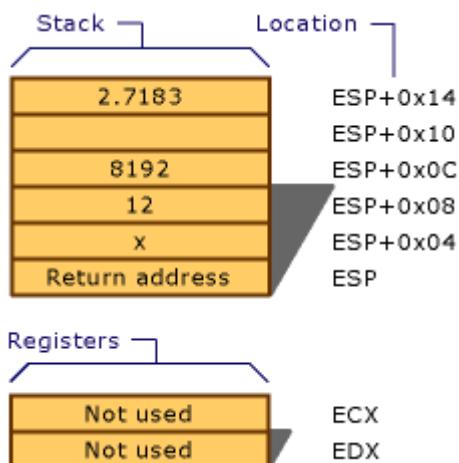
호출 예제 결과

아티클 • 2023. 10. 12.

Microsoft 전용

__cdecl

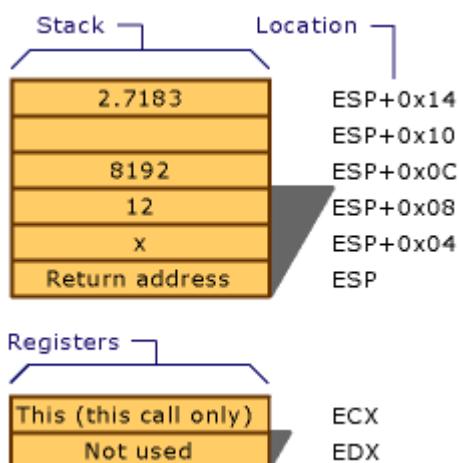
C 데코레이팅된 함수 이름은 .입니다 `_MyFunc`.



`__cdecl` 호출 규칙

__stdcall 및 thiscall

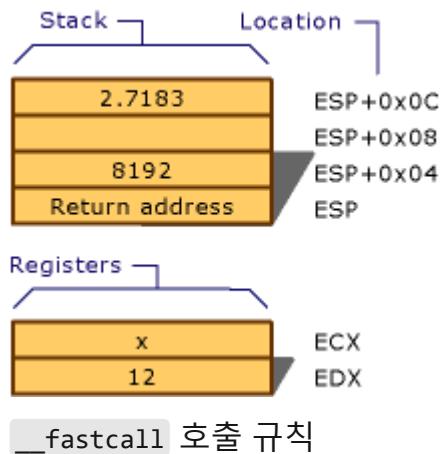
C 데코레이팅된 이름(`__stdcall`)은 `_MyFunc@20`. C++ 데코레이팅된 이름은 구현에 따라 다릅니다.



`__stdcall` 및 `thiscall` 호출 규칙

__fastcall

C 데코레이팅된 이름(`__fastcall`)은 `@MyFunc@20`. C++ 데코레이팅된 이름은 구현에 따라 다릅니다.



Microsoft 전용 종료

참고 항목

[호출 예제: 함수 프로토타입 및 호출](#)

Naked 함수 호출

아티클 • 2023. 10. 12.

Microsoft 전용

특성으로 **naked** 선언된 함수는 프롤로그 또는 에필로그 코드 없이 내보내기되므로 인라인 어셈블러를 사용하여 사용자 지정 프롤로그/에필로그 시퀀스를 작성할 수 있습니다. Naked 함수는 고급 기능으로 제공됩니다. 이 함수를 통해 C/C++ 이외의 컨텍스트에서 호출되는 함수를 선언할 수 있으므로 매개 변수의 위치와 유지되고 있는 레지스터에 대한 다양한 가정을 만들 수 있습니다. 예시에는 인터럽트 처리기와 같은 루틴이 포함됩니다. 이러한 기능은 VxD(가상 디바이스 드라이버) 작성에 특히 유용합니다.

추가 정보

- [naked](#)
- [Naked 함수의 규칙 및 제한](#)
- [프롤로그-에필로그 코드 작성 시 고려 사항](#)

Microsoft 전용 종료

참고 항목

[호출 규칙](#)

Naked 함수의 규칙 및 제한

아티클 • 2024. 11. 21.

Microsoft 전용

naked 함수에는 다음과 같은 규칙과 제한이 적용됩니다.

- 문은 `return` 허용되지 않습니다.
- 구조적 예외 처리 및 C++ 예외 처리 구문은 스택 프레임에서 해제되어야 하기 때문에 허용되지 않습니다.
- 같은 이유로, 모든 형태의 `setjmp` 가 금지됩니다.
- `_alloca` 함수의 사용이 금지됩니다.
- 프롤로그 시퀀스 전에 지역 변수에 대한 초기화 코드가 나타나지 않도록 하기 위해 초기화된 지역 변수가 함수 범위에서 허용되지 않습니다. 특히 C++ 개체의 선언이 함수 범위에서 허용되지 않습니다. 그러나 중첩된 범위에서 초기화된 데이터가 있을 수 있습니다.
- 프레임 포인터 최적화(/Oy 컴파일러 옵션)는 권장되지 않지만 naked 함수에 대해 자동으로 무시됩니다.
- C++ 클래스 개체를 함수 어휘 범위에서 선언할 수 없습니다. 그러나 중첩된 블록에서 개체를 선언할 수 있습니다.
- `naked/clr`로 컴파일할 때 키워드는 무시됩니다.
- `_fastcall` naked 함수의 경우 C/C++ 코드에서 레지스터 인수 중 하나에 대한 참조가 있을 때마다 프롤로그 코드는 해당 변수의 스택 위치에 해당 레지스터의 값을 저장해야 합니다. 예시:

C++

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall power(int i, int j) {
    // calculates i^j, assumes that j >= 0

    // prolog
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
        // store ECX and EDX into stack locations allocated for i and j
    }
```

```
    mov i, ecx
    mov j, edx
}

{
    int k = 1;    // return value
    while (j-- > 0)
        k *= i;
    __asm {
        mov eax, k
    };
}

// epilog
__asm {
    mov esp, ebp
    pop ebp
    ret
}
}
```

Microsoft 전용 종료

참고 항목

[Naked 함수 호출](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

프롤로그/에필로그 코드 작성 시 고려 사항

아티클 • 2023. 10. 12.

Microsoft 전용

고유한 프롤로그 및 에필로그 코드 시퀀스를 작성하기 전에 스택 프레임이 배치되는 방식을 이해하는 것이 중요합니다. 기호를 사용하는 `_LOCAL_SIZE` 방법을 아는 것도 유용합니다.

스택 프레임 레이아웃

이 예제에서는 32비트 함수에 표시될 수 있는 표준 prolog 코드를 보여 줍니다.

```
push    ebp          ; Save ebp
mov     ebp, esp    ; Set stack frame pointer
sub     esp, localbytes ; Allocate space for locals
push    <registers> ; Save registers
```

`localbytes` 변수는 지역 변수를 위한 스택에 필요한 바이트 수를 나타내고, `<registers>` 변수는 스택에 저장될 레지스터의 목록을 나타내는 자리 표시자입니다. 레지스터를 푸시한 후 스택에 다른 적절한 데이터를 배치할 수 있습니다. 다음은 해당 epilog 코드입니다.

```
pop    <registers> ; Restore registers
mov     esp, ebp    ; Restore stack pointer
pop    ebp          ; Restore ebp
ret                 ; Return from function
```

스택은 항상 높은 메모리 주소에서 낮은 메모리 주소로 감소합니다. 기본 포인터(`ebp`)는 `ebp`의 푸시된 값을 가리킵니다. 지역 변수 영역은 `ebp-4`에서 시작합니다. 지역 변수에 액세스하려면 `ebp`에서 적절한 값을 빼는 방법으로 `ebp`에서 오프셋을 계산합니다.

`_LOCAL_SIZE`

컴파일러는 함수 프롤로그 코드의 인라인 어셈블러 블록에 사용할 기호 `_LOCAL_SIZE`를 제공합니다. 이 기호는 사용자 지정 프롤로그 코드에서 스택 프레임에 지역 변수를 위한

공간을 할당하는 데 사용됩니다.

컴파일러가 .의 `__LOCAL_SIZE` 값을 결정합니다. 이 값은 모든 사용자 정의 지역 변수와 컴파일러에서 생성된 임시 변수의 총 바이트 수입니다. `__LOCAL_SIZE` 는 즉시 피연산자로만 사용할 수 있습니다. 식에서 사용할 수 없습니다. 이 기호의 값을 변경하거나 다시 정의하면 안 됩니다. 예시:

```
mov      eax, __LOCAL_SIZE          ;Immediate operand--Okay
mov      eax, [ebp - __LOCAL_SIZE]  ;Error
```

사용자 지정 프롤로그 및 에필로그 시퀀스를 포함하는 naked 함수의 다음 예제에서는 프롤로그 시퀀스에서 기호를 사용합니다 `__LOCAL_SIZE`.

C++

```
// the__local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
    int i;
    int j;

    __asm { /* prolog */
        push    ebp
        mov     ebp, esp
        sub     esp, __LOCAL_SIZE
    }

    /* Function body */
    __asm { /* epilog */
        mov     esp, ebp
        pop     ebp
        ret
    }
}
```

Microsoft 전용 종료

참고 항목

[Naked 함수 호출](#)

부동 소수점 보조 프로세서 및 호출 규칙

아티클 • 2024. 11. 21.

부동 소수점 공동 프로세서에 대한 어셈블리 루틴을 작성하는 경우 부동 소수점 제어 단어를 유지하고 값(함수가 ST(0)에서 반환되어야 하는) 값을 반환 `float double` 하지 않는 한 부동 소수점 제어 단어를 유지하고 공동 처리기 스택을 정리해야 합니다.

참고 항목

[호출 규칙](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

사용되지 않는 호출 규칙

아티클 • 2023. 10. 12.

Microsoft 전용

`_pascal`, `_fortran` 및 `_syscall` 호출 규칙은 더 이상 지원되지 않습니다. 지원되는 호출 규칙 및 적절한 링커 옵션 중 하나를 사용하여 해당 기능을 에뮬레이트할 수 있습니다.

<이제 windows.h> 는 대상에 대한 적절한 호출 규칙으로 변환되는 WINAPI 매크로를 지원합니다. 이전에 PASCAL 또는 `_far _pascal` 사용한 WINAPI를 사용합니다.

Microsoft 전용 종료

참고 항목

[인수 전달 및 명명 규칙](#)

restrict (C++ AMP)

아티클 • 2024. 11. 21.

제한 지정자는 함수 및 람다 선언에 적용할 수 있습니다. 제한 지정자는 C++ AMP(C++ Accelerated Massive Parallelism) 런타임을 사용하는 애플리케이션의 함수 동작 및 함수의 코드에 제한을 적용합니다.

① 참고

스토리지 클래스 특성의 `restrict` 일부인 키워드에 `_declspec` 대한 자세한 내용은 [restrict을 참조하세요](#).

절은 `restrict` 다음 형식을 사용합니다.

[+] 테이블 확장

절	설명
<code>restrict(cpu)</code>	이 함수는 전체 C++ 언어를 사용할 수 있습니다. <code>restrict(cpu)</code> 함수를 사용하여 선언된 다른 함수만 이 함수를 호출할 수 있습니다.
<code>restrict(amp)</code>	이 함수는 C++ AMP를 통해 가속할 수 있는 C++ 언어의 하위 집합만 사용할 수 있습니다.
<code>restrict(cpu)</code> 및 <code>restrict(amp)</code> 의 시퀀스 입니다.	이 함수는 <code>restrict(cpu)</code> 및 <code>restrict(amp)</code> 둘 다의 제한을 따라야 합니다. 이 함수는 <code>restrict(cpu)</code> , <code>restrict(amp)</code> , <code>restrict(cpu, amp)</code> 또는 <code>restrict(amp, cpu)</code> 를 사용하여 선언된 함수를 통해 호출할 수 있습니다.
<code>restrict(A) restrict(B)</code> 형식을 <code>restrict(A,B)</code> 로 작성할 수 있습니다.	

설명

`restrict` 키워드는 상황에 맞는 키워드입니다. 제한 지정자인 `cpu` 및 `amp`는 예약어가 아닙니다. 지정자 목록은 확장할 수 없습니다. 절이 없는 `restrict` 함수는 절이 있는 `restrict(cpu)` 함수와 동일합니다.

`restrict(amp)` 절이 있는 함수에는 다음 제한이 적용됩니다.

- 이 함수는 `restrict(amp)` 절이 포함된 함수만 호출할 수 있습니다.
- 함수 인라이닝 처리 가능해야 합니다.

- 함수는 이러한 형식만 포함하는 클래스 `float unsigned int` 및 `double` 구조체와, 변수만 `int` 선언할 수 있습니다. `bool` 은(는) 허용되지만 복합 형식으로 사용하는 경우 4 바이트 정렬이어야 합니다.
- 람다 함수는 참조로 캡처할 수 없으며 포인터를 캡처할 수 없습니다.
- 참조 및 단일 간접 포인터는 로컬 변수, 함수 인수 및 반환 형식으로만 지원됩니다.
- 다음은 허용되지 않습니다.
 - 재귀
 - volatile 키워드로 [선언된 변수입니다.](#)
 - 가상 함수
 - 함수에 대한 포인터
 - 멤버 함수에 대한 포인터
 - 구조체의 포인터
 - 포인터에 대한 포인터
 - `goto` 문.
 - 레이블 문
 - `try`, `catch` 또는 `throw` 문입니다.
 - 전역 변수
 - 정적 변수 대신 `tile_static` 키워드를 사용합니다.
 - `dynamic_cast` 캐스트.
 - `typeid` 연산자
 - `asm` 선언
 - `vararg`

함수 제한에 대한 자세한 내용은 `restrict(amp)` 제한을 참조 [하세요](#).

예시

다음 예제에서는 절을 사용하는 `restrict(amp)` 방법을 보여줍니다.

C++

```
void functionAmp() restrict(amp) {}
void functionNonAmp() {}

void callFunctions() restrict(amp)
{
    // int is allowed.
    int x;
    // long long int is not allowed in an amp-restricted function. This
    generates a compiler error.
    // long long int y;

    // Calling an amp-restricted function is allowed.
    functionAmp();

    // Calling a non-amp-restricted function is not allowed.
    // functionNonAmp();
}
```

참고 항목

[C++ AMP\(C++ Accelerated Massive Parallelism\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

tile_static 키워드

아티클 • 2024. 11. 21.

`tile_static` 키워드는 스레드 타일의 모든 스레드에서 액세스할 수 있는 변수를 선언하는 데 사용됩니다. 변수의 수명은 실행이 선언 지점에 도달할 때 시작되고 커널 함수가 반환될 때 종료됩니다. 타일 사용에 대한 자세한 내용은 타일 사용을 참조 [하세요](#).

`tile_static` 키워드에는 다음과 같은 제한 사항이 있습니다.

- `restrict(amp)` 한정자가 있는 함수 내의 변수에만 사용할 수 있습니다.
- 포인터 또는 참조 형식인 변수에 사용할 수 없습니다.
- `tile_static` 변수에는 이니셜라이저가 있을 수 없습니다. 기본 생성자 및 소멸자가 자동으로 호출되지 않습니다.
- 초기화 되지 않은 `tile_static` 변수의 값이 정의되지 않았습니다.
- 타일이 지정되지 않은 `parallel_for_each` 호출에 의해 루팅된 호출 그래프에서 `tile_static` 변수가 선언되면 경고가 생성되고 변수의 동작이 정의되지 않습니다.

예시

다음 예제에서는 `tile_static` 변수를 사용하여 타일의 여러 스레드에 걸쳐 데이터를 누적하는 방법을 보여 줍니다.

C++

```
// Sample data:  
int sampledata[] = {  
    2, 2, 9, 7, 1, 4,  
    4, 4, 8, 8, 3, 4,  
    1, 5, 1, 2, 5, 2,  
    6, 8, 3, 2, 7, 2};  
  
// The tiles:  
// 2 2      9 7      1 4  
// 4 4      8 8      3 4  
//  
// 1 5      1 2      5 2  
// 6 8      3 2      7 2  
  
// Averages:  
int averagedata[] = {  
    0, 0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0, 0,
```

```

    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.extent and divide the extent into 2 x 2
    tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
{
    // Create a 2 x 2 array to hold the values in this tile.
    tile_static int nums[2][2];
    // Copy the values for the tile into the 2 x 2 array.
    nums[idx.local[1]][idx.local[0]] = sample[idx.global];
    // When all the threads have executed and the 2 x 2 array is
    complete, find the average.
    idx.barrier.wait();
    int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
    // Copy the average into the array_view.
    average[idx.global] = sum / 4;
}
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
// Sample data.
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles are:
// 2 2      9 7      1 4
// 4 4      8 8      3 4
//
// 1 5      1 2      5 2
// 6 8      3 2      7 2

// Averages.
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,

```

```

    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.grid and divide the grid into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
{
    // Create a 2 x 2 array to hold the values in this tile.
    tile_static int nums[2][2];
    // Copy the values for the tile into the 2 x 2 array.
    nums[idx.local[1]][idx.local[0]] = sample[idx.global];
    // When all the threads have executed and the 2 x 2 array is
    complete, find the average.
    idx.barrier.wait();
    int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
    // Copy the average into the array_view.
    average[idx.global] = sum / 4;
}
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output.
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4

```

참고 항목

- [Microsoft 전용 한정자](#)
- [C++ AMP 개요](#)
- [parallel_for_each 함수\(C++ AMP\)](#)
- [연습: 매트릭스 곱](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__declspec

아티클 • 2024. 07. 15.

Microsoft 전용

스토리지 클래스 정보를 지정하기 위한 확장 특성 구문은 주어진 형식의 인스턴스가 아래에 나열된 Microsoft 전용 스토리지 클래스 특성으로 저장되도록 지정하는 `__declspec` 키워드를 사용합니다. 다른 스토리지 클래스 한정자의 예로는 `static` 및 `extern` 키워드가 있습니다. 그러나 이러한 키워드는 C 및 C++ 언어의 ANSI 사양에 포함되므로 확장 특성 구문에서 다루지 않습니다. 확장명 특성 구문은 C 및 C++ 언어에 대한 Microsoft 전용 확장을 간소화하고 표준화합니다.

문법

`decl-specifier:`

`__declspec (extended-decl-modifier-seq)`

`extended-decl-modifier-seq:`

`extended-decl-modifier` opt

`extended-decl-modifier` `extended-decl-modifier-seq`

`extended-decl-modifier:`

`align(number)`

`allocate(" segname ")`

`allocator`

`appdomain`

`code_seg(" segname ")`

`deprecated`

`dllimport`

`dllexport`

`empty_bases`

`jit intrinsic`

`naked`

`noalias`

`noinline`

`noreturn`

`nothrow`

`novtable`

```
no_SANITIZE_ADDRESS  
process  
property( { get=Get-Func-Name | ,put=Put-Func-Name } )  
restrict  
safebuffers  
selectany  
spectre(nomitigation)  
thread  
uuid(" ComObjectGUID ")
```

공백은 선언 한정자 시퀀스를 구분합니다. 뒤에 나오는 단원에 예제가 있습니다.

확장 특성 문법은 Microsoft 전용 스토리지 클래스 특성인 align, allocate, allocator, appdomain, code_seg, deprecated, dllexport, dllimport, empty_bases, jitintrinsic, naked, noalias, noinline, noreturn, nothrow, novtable, no_sanitize_address, process, restrict, safebuffers, selectany, spectre, 및 thread를 지원합니다. 또한 COM 개체 특성인 property 및 uuid도 지원합니다.

`code_seg`, `dllexport`, `dllimport`, `empty_bases`, `naked`, `noalias`, `nothrow`, `no_SANITIZE_ADDRESS`, `property`, `restrict`, `selectany`, `thread`, 및 `uuid` 스토리지 클래스 특성은 적용되는 개체나 함수의 선언에만 해당되는 속성입니다. `thread` 특성은 데이터 및 개체에만 영향을 줍니다. `naked` 및 `spectre` 특성은 함수에만 영향을 줍니다. `dllimport` 및 `dllexport` 특성은 함수, 데이터, 및 개체에 영향을 줍니다. `property`, `selectany`, 및 `uuid` 특성은 COM 개체에 영향을 줍니다.

이전 버전과의 호환성을 위해 `_declspec`은 `_declspec`의 동의어입니다. 단, 컴파일러 옵션 `/Za`(언어 확장 사용 안 함)가 지정된 경우는 예외입니다.

_declspec 키워드는 간단한 선언의 시작 부분에 배치해야 합니다. 컴파일러는 선언에서 * 또는 & 뒤와 변수 식별자 앞에 오는 **declspec** 키워드를 경고 없이 무시합니다.

사용자 정의 형식 선언의 시작 부분에 지정된 `_declspec` 특성은 해당 형식의 변수에 적용됩니다. 예시:

C++

```
_declspec(dllexport) class X {} varX;
```

이 경우 특성이 `varX`에 적용됩니다. `class` 또는 `struct` 키워드 뒤에 오는 `_declspec` 특성은 사용자 정의 형식에 적용됩니다. 예시:

C++

```
class __declspec(dllexport) X {};
```

이 경우 특성이 `x`에 적용됩니다.

간단한 선언에 `__declspec` 특성을 사용하는 데 대한 일반적인 지침은 다음과 같습니다.

```
decl-specifier-seq init-declarator-list ;
```

`decl-specifier-seq`에는 무엇보다도 기본 형식(예: `int`, `float`, `typedef`, 또는 클래스 이름), 스토리지 클래스(예: `static`, `extern`) 또는 `__declspec` 확장명이 포함되어야 합니다. `init-declarator-list`에는 무엇보다도 선언의 포인터 부분이 포함되어야 합니다. 예시:

C++

```
__declspec(selectany) int * pi1 = 0;    //Recommended, selectany & int both
part of decl-specifier
int __declspec(selectany) * pi2 = 0;    //OK, selectany & int both part of
decl-specifier
int * __declspec(selectany) pi3 = 0;    //ERROR, selectany is not part of a
declarator
```

다음 코드는 정수 스레드 지역 변수를 선언하고 값을 사용하여 초기화합니다.

C++

```
// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

Microsoft 전용 종료

참고 항목

[키워드](#)

[C 확장 스토리지 클래스 특성](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

align (C++)

아티클 • 2024. 07. 08.

Visual Studio 2015 이상에서는 [alignas](#) 지정자(C++11)를 사용하여 맞춤을 제어합니다. 자세한 내용은 [맞춤을 참조하세요](#).

Microsoft 전용

`__declspec(align(#))` 를 사용하여 사용자 정의 데이터(예: 함수의 정적 할당 또는 자동 데이터)를 정확하게 제어합니다.

구문

`__declspec(align(#)) 선언자`

설명

최신 프로세서 명령을 사용하는 애플리케이션을 작성할 경우 몇 가지 새로운 제약 조건과 문제가 생깁니다. 많은 새로운 명령에서는 데이터를 16바이트 경계로 정렬해야 합니다. 자주 사용하는 데이터를 프로세서의 캐시 라인 크기에 맞춰 정렬하면 캐시 성능이 개선됩니다. 예를 들어, 크기가 32바이트 미만인 구조를 정의하는 경우 해당 구조체 형식의 개체가 효율적으로 캐시되도록 32바이트 맞춤이 필요할 수 있습니다.

#은 맞춤 값입니다. 유효한 항목은 2, 4, 8, 16, 32 또는 64와 같이 1에서 8192(바이트) 사이에 속하는 2의 정수 제곱입니다. `declarator`는 정렬된 것으로 선언하는 데이터입니다.

형식의 맞춤 요구 사항인 `size_t` 형식의 값을 반환하는 방법에 대한 자세한 내용은 [alignof](#)를 참조하세요. 64비트 프로세서를 대상으로 할 때 정렬되지 않은 포인터를 선언하는 방법에 대한 자세한 내용은 [_unaligned](#)를 참조하세요.

`struct`, `union` 또는 `class`를 정의하거나 변수를 선언할 때 `__declspec(align(#))` 을 사용할 수 있습니다.

컴파일러는 복사 또는 데이터 변형 작업 중에 데이터의 맞춤 특성 보존을 보장하거나 시도하지 않습니다. 예를 들어, `memcpy`는 `__declspec(align(#))` 으로 선언된 구조체를 어느 위치에나 복사할 수 있습니다. 일반 할당자(예: `malloc`, C++ `operator new` 및 Win32 할당자)는 일반적으로 `__declspec(align(#))` 구조체 또는 구조체 배열에 대해 정렬되지 않은 메모리를 반환합니다. 복사 또는 데이터 변환 작업의 대상이 올바르게 정렬되도록 하려면 [_aligned_malloc](#)를 사용합니다. 또는 고유의 할당자를 작성합니다.

함수 매개 변수에 대한 맞춤을 지정할 수 없습니다. 맞춤 특성이 있는 데이터를 스택의 값으로 전달하면 호출 규칙에 따라 맞춤이 제어됩니다. 호출된 함수에서 데이터 맞춤이 중요한 경우에는 사용 전에 매개 변수를 올바르게 맞춰진 메모리로 복사합니다.

`__declspec(align(#))`을 사용하지 않는 경우 컴파일러는 일반적으로 대상 프로세서와 데이터 크기를 기반으로 자연 경계에 데이터를 맞춥니다(32비트 프로세서에서는 최대 4바이트 경계, 64비트 프로세서에서는 최대 8바이트 경계). 클래스 또는 구조체의 데이터는 최소한의 자연 맞춤 및 현재의 압축 설정(`#pragma pack` 또는 `/Zp` 컴파일러 옵션에서)으로 클래스나 구조체에 맞춰집니다.

이 예제에서는 `__declspec(align(#))`의 사용을 보여 줍니다.

C++

```
__declspec(align(32)) struct Str1{  
    int a, b, c, d, e;  
};
```

현재 이 형식에는 32비트 맞춤 특성이 포함되어 있습니다. 즉, 모든 정적 및 자동 인스턴스가 32바이트 경계에서 시작됩니다. 이 형식을 멤버로 선언한 다른 구조체 형식은 이 형식의 맞춤 특성을 유지합니다. 즉, `Str1`을 요소로 포함하는 모든 구조체의 맞춤 특성은 32 이상입니다.

여기서 `sizeof(struct Str1)`은 32와 같습니다. 즉, `Str1` 개체 배열을 만드는 경우 배열의 기준을 32바이트로 맞추면 배열의 각 멤버도 32바이트로 맞춰집니다. 동적 메모리에서 기본이 올바르게 정렬된 배열을 만들려면 `_aligned_malloc`를 사용합니다. 또는 고유의 할당자를 작성합니다.

구조체의 `sizeof` 값은 최종 멤버의 오프셋에 해당 멤버의 크기를 더하여 최대 멤버 맞춤 값 또는 전체 구조체 맞춤 값 중 더 큰 값의 가장 근사한 배수로 반올림한 값입니다.

컴파일러는 구조체 맞춤에 다음과 같은 규칙을 사용합니다.

- `__declspec(align(#))`로 재정의하지 않으면 스칼라 구조체 멤버의 맞춤은 최소 크기와 현재 압축입니다.
- `__declspec(align(#))`로 재정의하지 않으면 구조체의 멤버는 멤버의 최대 개별 맞춤입니다.
- 구조체 멤버는 이전 멤버 끝의 오프셋보다 크거나 같은 맞춤의 최소 배수인 부모 구조체의 시작 부분부터 오프셋에 배치됩니다.
- 구조체의 크기는 마지막 멤버의 오프셋 끝보다 크거나 같은 맞춤의 최소 배수입니다.

`__declspec(align(#))`는 맞춤 제한만 늘릴 수 있습니다.

자세한 내용은 다음을 참조하세요.

- align 예
- `__declspec(align(#))`를 사용하여 새 형식 정의
- 스레드 로컬 스토리지에서 데이터 맞춤
- align이 데이터 압축과 함께 작동하는 방식
- x64 구조체 맞춤 예

align 예제

다음 예제에서는 `__declspec(align(#))`가 데이터 구조체의 크기 및 맞춤에 영향을 주는 방식을 보여 줍니다. 예제에서는 다음과 같은 정의를 가정합니다.

C++

```
#define CACHE_LINE 32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

이 예제에서는 `s1`를 사용하여 `__declspec(align(32))` 구조체를 정의합니다. 변수 정의에 대해 또는 기타 형식 선언에서 사용되는 모든 `s1`은 32바이트로 맞춰집니다.

`sizeof(struct S1)`은 32를 반환하고 `s1`은 16바이트 뒤에 정수 4개에 필요한 16 패딩 바이트를 둡니다. 각 `int` 멤버는 4바이트로 맞춰야 하지만 구조체 자체의 맞춤은 32로 선언됩니다. 그런 다음 전체 맞춤은 32입니다.

C++

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
    int a, b, c, d;
};
struct S1 s1; // s1 is 32-byte cache aligned
```

이 예제에서는 최대 맞춤 요구 사항의 배수가 8의 배수인 16이므로 `sizeof(struct S2)`는 멤버 크기의 합계와 똑같은 16을 반환합니다.

C++

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

다음 예제에서 `sizeof(struct S3)`는 64를 반환합니다.

C++

```
struct S3 {
    struct S1 s1;    // S3 inherits cache alignment requirement
                      // from S1 declaration
    int a;          // a is now cache aligned because of s1
                    // 28 bytes of trailing padding
};
```

이 예제에서 `a`에는 자연 형식 맞춤(여기서는 4바이트)이 사용됩니다. 그러나 `s1`은 32바이트 맞춤이어야 합니다. 28바이트 패딩이 `a` 뒤에 오므로 `s1`은 오프셋 32에서 시작됩니다. 그런 다음 `s4`가 구조체의 최대 맞춤 요구 사항인 `s1`의 맞춤 요구 사항을 상속합니다. `sizeof(struct S4)` 가 64를 반환합니다.

C++

```
struct S4 {
    int a;
    // 28 bytes padding
    struct S1 s1;      // S4 inherits cache alignment requirement of S1
};
```

다음 3개의 변수 선언에도 `_declspec(align(#))` 가 사용됩니다. 각 선언에서 변수가 32바이트 맞춤이어야 합니다. 배열에서는 각 배열 멤버가 아닌 배열의 기준 주소가 32바이트로 정렬됩니다. 각 배열 멤버의 `sizeof` 값은 `_declspec(align(#))` 을 사용해도 영향을 받지 않습니다.

C++

```
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

배열의 각 멤버를 맞추려면 다음과 같은 코드를 사용해야 합니다.

C++

```
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

이 예제에서는 구조체 자체를 맞추는 경우와 첫 번째 요소를 맞추는 경우의 결과는 같습니다.

C++

```
CACHE_ALIGN struct S6 {
    int a;
    int b;
};

struct S7 {
    CACHE_ALIGN int a;
    int b;
};
```

`S6` 및 `S7`의 맞춤, 할당 및 크기 특성이 동일합니다.

이 예에서 `a`, `b`, `c` 및 `d`의 시작 주소 맞춤은 각각 4, 1, 4 및 1입니다.

C++

```
void fn() {
    int a;
    char b;
    long c;
    char d[10]
}
```

메모리가 힙에 할당된 경우 호출되는 할당 함수에 따라 맞춤이 결정됩니다. 예를 들어, `malloc`을 사용할 경우 피연산자 크기에 따라 결과가 결정됩니다. $arg \geq 8$ 이면 반환되는 메모리는 8바이트에 맞춰집니다. $arg < 8$ 이면 반환되는 메모리의 맞춤은 arg 보다 작은 2의 첫 번째 거듭제곱이 됩니다. 예를 들어, `malloc(7)`을 사용하는 경우 맞춤은 4바이트입니다.

`__declspec(align(#))`을 사용하여 새 형식 정의

맞춤 특성을 사용하여 형식을 정의할 수 있습니다.

예를 들어 다음과 같이 맞춤 값을 사용하여 `struct`를 정의할 수 있습니다.

C++

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

이제 `aType`과 `bType`은 동일한 크기(8바이트)이지만 `bType` 형식의 변수는 32바이트로 정렬됩니다.

스레드 로컬 스토리지에서 데이터 맞춤

`__declspec(thread)` 특성으로 만들어 이미지의 TLS 섹션에 넣은 정적 TLS(스레드 로컬 스토리지)는 보통의 정적 데이터와 똑같은 맞춤으로 작동합니다. 운영 체제에서는 TLS 데이터를 만들기 위해 메모리에 TLS 섹션의 크기를 할당하고 TLS 섹션 맞춤 특성을 고려합니다.

다음 예제에서는 맞춰진 데이터를 스레드 로컬 스토리지에 배치하는 여러 가지 방법을 보여 줍니다.

C++

```
// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;

// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
    int a;
    int b;
};
// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;
```

align이 데이터 압축과 함께 작동하는 방식

`/Zp` 컴파일러 옵션과 `pack` pragma는 구조체 및 공용 구조체 멤버의 압축 데이터 역할을 합니다. 이 예제에서는 `/Zp` 와 `__declspec(align(#))` 이 함께 작동하는 방법을 보여 줍니다.

C++

```
struct S {
    char a;
    short b;
    double c;
    CACHE_ALIGN double d;
    char e;
    double f;
};
```

다음 표에는 서로 다른 `/Zp`(또는 `#pragma pack`) 값 아래의 각 멤버의 오프셋이 나열되어 있으며 두 멤버가 어떻게 상호 작용하는지 보여 줍니다.

변수	/Zp1	/Zp2	/Zp4	/Zp8
a	0	0	0	0
b	1	2	2	2
c	3	4	4	8
d	32	32	32	32
e	40	40	40	40
f	41	42	44	48
sizeof(S)	64	64	64	64

자세한 내용은 [/Zp\(구조체 멤버 맞춤\)](#)을 참조하세요.

개체의 오프셋은 이전 개체의 오프셋과 현재 압축 설정을 기반으로 합니다. 단, 개체에 `__declspec(align(#))` 특성이 있는 경우 맞춤은 이전 개체의 오프셋과 개체의 `__declspec(align(#))` 값을 기반으로 합니다.

Microsoft 전용 종료

참고 항목

[__declspec](#)

[ARM ABI 규칙 개요](#)

[x64 소프트웨어 규칙](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

할당

아티클 • 2024. 07. 15.

Microsoft 전용

`allocate` 선언 지정자는 데이터 항목이 할당될 데이터 세그먼트의 이름을 지정합니다.

구문

```
__declspec(allocate(" segname )) 선언자
```

설명

*segname*이라는 이름은 다음 pragma 중 하나를 사용하여 선언해야 합니다.

- `code_seg`
- `const_seg`
- `data_seg`
- `init_seg`
- `section`

예시

```
C++  
  
// allocate.cpp  
#pragma section("mycode", read)  
__declspec(allocate("mycode")) int i = 0;  
  
int main() {  
}
```

Microsoft 전용 종료

참고 항목

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

allocator

아티클 • 2023. 10. 12.

Microsoft 전용

allocator 선언 지정자를 사용자 지정 메모리 할당 함수에 적용하여 ETW(Windows용 이벤트 추적)를 통해 할당을 표시할 수 있습니다.

구문

`__declspec(allocator)`

설명

Visual Studio의 네이티브 메모리 프로파일러에서는 런타임 중에 내보낸 할당 ETW 이벤트 데이터를 수집하여 작동합니다. CRT 및 Windows SDK의 할당자가 해당 할당 데이터를 캡처할 수 있도록 원본 수준에서 주석이 추가되었습니다. 고유한 할당자를 작성하는 경우 myMalloc에 대한 이 예제와 같이 새로 할당된 힙 메모리에 대한 포인터를 반환하는 모든 함수를 `__declspec(allocator)` 데코레이트할 수 있습니다.

C++

```
__declspec(allocator) void* myMalloc(size_t size)
```

자세한 내용은 Visual Studio 및 사용자 지정 네이티브 ETW 힙 이벤트의 메모리 사용량 측정을 [참조하세요](#).

Microsoft 전용 종료

appdomain

아티클 • 2024. 11. 21.

관리되는 애플리케이션의 각 애플리케이션 도메인에 특정 전역 변수 또는 정적 멤버 변수의 자체 복사본이 포함되어야 하도록 지정합니다. 자세한 내용은 애플리케이션 도메인 및 Visual C++ [를 참조하세요](#).

모든 애플리케이션 도메인에는 appdomain별 변수의 자체 복사본이 있습니다. 어셈블리를 애플리케이션 도메인으로 로드하면 appdomain 변수의 생성자가 실행되고 애플리케이션 도메인을 언로드하면 소멸자가 실행됩니다.

공용 언어 런타임에서 프로세스 내의 모든 애플리케이션 도메인이 전역 변수를 공유하도록 하려면 `_declspec(process)` 한정자를 사용합니다. `_declspec(process)`는 기본적으로 /clr에서 적용됩니다. /clr:pure 및 /clr:safe 컴파일러 옵션은 Visual Studio 2015에서 더 이상 사용되지 않으며 Visual Studio 2017에서는 지원되지 않습니다.

`_declspec(appdomain)`는 /clr 컴파일러 옵션 중 하나가 사용되는 경우에만 유효합니다. 전역 변수, 정적 멤버 변수 또는 정적 로컬 변수만 `_declspec(appdomain)`로 표시할 수 있습니다. 관리되는 형식의 정적 멤버는 항상 이 동작을 수행하므로 `_declspec(appdomain)`를 적용하면 오류가 발생합니다.

사용 `_declspec(appdomain)`은 TLS(스레드 로컬 스토리지) [를 사용하는](#) 것과 유사합니다. 스레드에는 애플리케이션 도메인과 마찬가지로 자체 스토리지가 있습니다.

`_declspec(appdomain)`를 사용하면 전역 변수가 이 애플리케이션용으로 작성된 각 애플리케이션 도메인에 자체 스토리지를 포함할 수 있습니다.

프로세스당 및 appdomain 변수당 사용을 혼합하는 데는 제한이 있습니다. 자세한 내용은 [프로세스를 참조하세요](#).

예를 들어 프로그램 시작 시에는 모든 프로세스별 변수가 초기화된 후에 모든 appdomain 변수가 초기화됩니다. 따라서 프로세스별 변수는 초기화 중에 애플리케이션 도메인별 변수의 값을 사용할 수 없습니다. 따라서 appdomain별 변수와 프로세스별 변수를 조합하여 사용(할당)하는 것은 적절하지 않습니다.

특정 애플리케이션 도메인에서 함수를 호출하는 방법에 대한 자세한 내용은 `call_in_appdomain` 함수 [를 참조하세요](#).

예시

C++

```

// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
    CGlobal(bool bProcess) {
        Counter = 10;
        m_bProcess = bProcess;
        Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor",
m_bProcess ? (String^)"process" : (String^)"appdomain");
    }

    ~CGlobal() {
        Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor",
m_bProcess ? (String^)"process" : (String^)"appdomain");
    }

    int Counter;

private:
    bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
    static void change() {
        ++appdomain_global.Counter;
    }

    static void display() {
        Console::WriteLine("process_global value in appdomain '{0}': {1}",
            AppDomain::CurrentDomain->FriendlyName,
            process_global.Counter);

        Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
            AppDomain::CurrentDomain->FriendlyName,
            appdomain_global.Counter);
    }
};

int main() {
    AppDomain^ defaultDomain = AppDomain::GetCurrentDomain();
    AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
    AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
    CrossAppDomainDelegate^ changeDelegate = gcnew
CrossAppDomainDelegate(&Functions::change);
    CrossAppDomainDelegate^ displayDelegate = gcnew
CrossAppDomainDelegate(&Functions::display);
}

```

```

// Print the initial values of appdomain_global in all appdomains.
Console::WriteLine("Initial value");
defaultDomain->DoCallBack(displayDelegate);
domain->DoCallBack(displayDelegate);
domain2->DoCallBack(displayDelegate);

// Changing the value of appdomain_global in the domain and domain2
// appdomain_global value in "default" appdomain remain unchanged
process_global.Counter = 20;
domain->DoCallBack(changeDelegate);
domain2->DoCallBack(changeDelegate);
domain2->DoCallBack(changeDelegate);

// Print values again
Console::WriteLine("Changed value");
defaultDomain->DoCallBack(displayDelegate);
domain->DoCallBack(displayDelegate);
domain2->DoCallBack(displayDelegate);

AppDomain::Unload(domain);
AppDomain::Unload(domain2);
}

```

Output

```

__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor

```

참고 항목

_declspec

키워드

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공  | Microsoft Q&A에서 도움말 보기

`__declspec(code_seg)`

아티클 • 2024. 07. 15.

Microsoft 전용

`code_seg` 선언 속성은 함수 또는 클래스 멤버 함수의 개체 코드가 저장될 `.obj` 파일의 실행 텍스트 세그먼트의 이름을 지정합니다.

구문

```
__declspec(code_seg(" segname ")) declarator
```

설명

`__declspec(code_seg(...))` 특성을 사용하면 메모리에 개별적으로 페이지 또는 잠글 수 있는 별도의 명명된 세그먼트에 코드를 배치할 수 있습니다. 이 특성을 사용하여 인스턴스화한 템플릿과 컴파일러에서 생성된 코드의 배치를 제어할 수 있습니다.

세그먼트는 메모리에 하나의 단위로 로드된 `.obj` 파일의 명명된 데이터 블록입니다. 텍스트 세그먼트는 실행 코드가 포함된 세그먼트입니다. 섹션이라는 용어는 종종 세그먼트와 같은 의미로 사용됩니다.

`declarator` 가 지정될 때 생성되는 개체 코드는 좁은 문자열 리터럴인 `segname`에서 지정한 텍스트 세그먼트에 배치됩니다. `segname`이라는 이름은 [섹션](#) pragma에 지정되지 않아도 선언에 사용할 수 있습니다. 기본적으로 `code_seg` 가 지정되지 않으면 개체 코드가 `.text`라는 세그먼트에 배치됩니다. `code_seg` 특성은 기준의 모든 `#pragma code_seg` 지시문을 재정의합니다. 멤버 함수에 적용된 `code_seg` 특성은 바깥쪽 클래스에 적용되는 모든 `code_seg` 특성을 재정의합니다.

엔터티에 `code_seg` 특성이 있는 경우 같은 엔터티의 모든 선언 및 정의가 `code_seg` 특성과 동일해야 합니다. 기본 클래스에 `code_seg` 특성이 있는 경우 파생된 클래스는 특성이 동일해야 합니다.

`code_seg` 특성이 네임스페이스 범위 함수 또는 멤버 함수에 적용되면 해당 함수의 개체 코드가 지정된 텍스트 세그먼트에 배치됩니다. 이 특성이 클래스에 적용되면 컴파일러 생성 특수 멤버 함수를 포함하는 클래스와 중첩된 클래스의 모든 멤버 함수가 지정된 세그먼트에 배치됩니다. 멤버 함수 본문에 정의된 클래스 등 로컬 정의 클래스는 바깥쪽 범위의 `code_seg` 특성을 상속하지 않습니다.

`code_seg` 특성이 클래스 템플릿 또는 함수 템플릿에 적용되면 템플릿의 모든 암시적 특수화가 지정된 세그먼트에 배치됩니다. 명시적 또는 부분적 특수화는 기본 템플릿에서 `code_seg` 특성을 상속하지 않습니다. 특수화에 같거나 다른 `code_seg` 특성을 지정할 수 있습니다. `code_seg` 특성은 명시적 템플릿 인스턴스화에 적용할 수 없습니다.

기본적으로 특수 멤버 함수 등 컴파일러에서 생성된 코드는 `.text` 세그먼트에 배치됩니다. `#pragma code_seg` 지시문은 이 기본값을 재정의하지 않습니다. 컴파일러에서 생성된 코드가 배치되는 제어에 클래스, 클래스 템플릿 또는 함수 템플릿의 `code_seg` 특성을 사용합니다.

람다는 바깥쪽 범위에서 `code_seg` 특성을 상속합니다. 람다에 대한 세그먼트를 지정하려면 매개 변수 선언 절 뒤와 변경 가능한 변수 또는 예외, 모든 후행 반환 형식 사양 및 람다 본문 앞에 `code_seg` 특성을 적용합니다. 자세한 내용은 [람다 식 구문](#)을 참조하세요. 이 예제에서는 `PagedMem`이라는 세그먼트에 람다를 정의합니다.

C++

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;  
};
```

여러 세그먼트에 특정 멤버 함수, 특히 가상 멤버 함수를 배치할 때 주의해야 합니다. 기본 클래스 메서드가 페이지징되지 않은 세그먼트에 있을 때 페이지징된 세그먼트에 상주하는 파생 클래스에서 가상 함수를 정의한다고 가정해 보겠습니다. 다른 기본 클래스 메서드 또는 사용자 코드에서는 가상 메서드를 호출해도 페이지 풀트가 트리거되지 않는다고 가정 할 수 있습니다.

예시

이 예제에서는 암시적 및 명시적 템플릿 특수화를 사용할 때 `code_seg` 특성이 세그먼트 배치를 제어하는 방법을 보여 줍니다.

C++

```
// code_seg.cpp  
// Compile: cl /EHsc /W4 code_seg.cpp  
  
// Base template places object code in Segment_1 segment  
template<class T>  
class __declspec(code_seg("Segment_1")) Example  
{  
public:  
    virtual void VirtualMemberFunction(T /*arg*/) {}  
};
```

```
// bool specialization places code in default .text segment
template<>
class Example<bool>
{
public:
    virtual void VirtualMemberFunction(bool /*arg*/) {}
};

// int specialization places code in Segment_2 segment
template<>
class __declspec(code_seg("Segment_2")) Example<int>
{
public:
    virtual void VirtualMemberFunction(int /*arg*/) {}
};

// Compiler warns and ignores __declspec(code_seg("Segment_3"))
// in this explicit specialization
__declspec(code_seg("Segment_3")) Example<short>; // C4071

int main()
{
    // implicit double specialization uses base template's
    // __declspec(code_seg("Segment_1")) to place object code
    Example<double> doubleExample{};
    doubleExample.VirtualMemberFunction(3.14L);

    // bool specialization places object code in default .text segment
    Example<bool> boolExample{};
    boolExample.VirtualMemberFunction(true);

    // int specialization uses __declspec(code_seg("Segment_2"))
    // to place object code
    Example<int> intExample{};
    intExample.VirtualMemberFunction(42);
}
```

Microsoft 전용 종료

참고 항목

[__declspec](#)
[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

사용되지 않음 (C++)

아티클 • 2023. 10. 12.

이 항목에서는 더 이상 사용되지 않는 Microsoft 전용 declspec 선언에 대해 설명합니다. C++14 `[[deprecated]]` 특성에 대한 정보와 해당 특성을 사용하는 시기와 Microsoft 관련 declspec 또는 pragma에 대한 지침은 C++ 표준 특성을 [참조하세요](#).

아래에 설명된 예외를 제외하고 선언은 `deprecated` 사용되지 않는 `pragma`와 동일한 기능을 제공합니다.

- 선언을 `deprecated` 사용하면 특정 형식의 함수 오버로드를 더 이상 사용되지 않는 것으로 지정할 수 있지만 pragma 형식은 함수 이름의 모든 오버로드된 형식에 적용됩니다.
- 선언을 `deprecated` 사용하면 컴파일 시간에 표시할 메시지를 지정할 수 있습니다. 메시지의 텍스트는 매크로에서 제공될 수 있습니다.
- 매크로는 pragma로 더 이상 사용되지 `deprecated` 않는 것으로 표시될 수 있습니다.

컴파일러에서 사용되지 않는 식별자 또는 표준 `[[deprecated]]` 특성을 [사용하는 경우 C4996](#) 경고가 throw됩니다.

예제

다음 샘플에서는 함수를 사용되지 않는 것으로 표시하는 방법과 사용되지 않는 함수가 사용되는 경우 컴파일 타임에 표시될 메시지를 지정하는 방법을 보여 줍니다.

C++

```
// deprecated.cpp
// compile with: /W3
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void
func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
    func1();
    func1(1);    // C4996
    func2(1);    // C4996
    func3(1);    // C4996
}
```

다음 샘플에서는 클래스를 사용되지 않는 것으로 표시하는 방법과 사용되지 않는 클래스가 사용되는 경우 컴파일 타임에 표시될 메시지를 지정하는 방법을 보여 줍니다.

C++

```
// deprecate_class.cpp
// compile with: /W3
struct __declspec(deprecated) X {
    void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
    void f(){}
};

int main() {
    X x;      // C4996
    X2 x2;    // C4996
}
```

참고 항목

[__declspec](#)

[키워드](#)

dllexport, dllimport

아티클 • 2024. 04. 01.

Microsoft 전용

`dllexport` 및 `dllimport` 저장소 클래스 특성은 C 및 C++ 언어에 대한 Microsoft 고유의 확장입니다. 이러한 특성을 사용하여 함수, 데이터 및 개체를 DLL에 내보내거나 DLL에서 가져올 수 있습니다.

구문

```
__declspec( dllimport ) declarator  
__declspec( dllexport ) declarator
```

설명

이러한 특성은 실행 파일이나 다른 DLL일 수 있는 클라이언트에 대한 DLL 인터페이스를 명시적으로 정의합니다. 함수를 `dllexport`(으)로 선언하면 적어도 내보낸 함수의 지정과 관련하여 모듈 정의(`.def`) 파일이 필요하지 않게 됩니다. `dllexport` 특성은 `_export` 키워드를 대체합니다.

클래스가 `__declspec(dllexport)`(으)로 표시된 경우 클래스 계층 구조에 있는 클래스 템플릿의 특수화는 암시적으로 `__declspec(dllexport)`(으)로 표시됩니다. 이는 클래스 템플릿이 명시적으로 인스턴스화되었으며 클래스의 멤버가 정의되어야 함을 의미합니다.

함수의 `dllexport`은(는) "name mangling"이라고도 하는 데코레이팅된 이름으로 함수를 노출합니다. C++ 함수의 경우 데코레이팅된 이름에는 형식 및 매개 변수 정보를 인코딩하는 추가 문자가 포함됩니다. C 함수 또는 `extern "C"`(으)로 선언된 함수는 C 이름 데코레이션 규칙을 따릅니다. C/C++ 코드의 이름 데코레이션에 대한 자세한 내용은 [데코레이트된 이름](#)을 참조하세요.

데코레이트되지 않은 이름을 내보내려면 `EXPORTS` 섹션에서 데코레이트되지 않은 이름을 정의하는 모듈 정의(`.def`) 파일을 사용하여 연결할 수 있습니다. 자세한 내용은 [EXPORTS](#)를 참조하세요. 데코레이트되지 않은 이름을 내보내는 또 다른 방법은 소스 코드에 `#pragma comment(linker, "/export:alias=decorated_name")` 지시문을 사용하는 것입니다.

`dllexport` 또는 `dllimport`을(를) 선언하는 경우 [확장 특성 구문](#)과 `__declspec` 키워드를 사용해야 합니다.

예시

C++

```
// Example of the __declspec( dllimport ) and __declspec( dllexport ) class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

또는 매크로 정의를 사용하여 코드를 보다 읽기 쉽게 만들 수 있습니다.

C++

```
#define DllImport    __declspec( dllimport )
#define Dllexport    __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
Dllexport int n;
```

자세한 내용은 다음을 참조하세요.

- 정의 및 선언
- dllexport 및 dllimport(으)로 인라인 C++ 함수 정의
- 일반 규칙 및 제한 사항
- C++ 클래스에서 dllimport 및 dllexport 사용

Microsoft 전용 종료

참고 항목

[__declspec](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

선언 및 정의 (C++)

아티클 • 2023. 10. 12.

Microsoft 전용

DLL 인터페이스는 시스템의 일부 프로그램에서 내보내지는 것으로 알려진 모든 항목(함수 및 데이터), 즉 `dllimport` 또는 `dllexport`로 선언되는 모든 항목을 참조합니다. DLL 인터페이스에 포함된 모든 선언은 `dllimport` 또는 `dllexport` 특성을 지정해야 합니다. 그러나 정의는 특성만 `dllexport` 지정해야 합니다. 예를 들어, 다음 함수 정의는 컴파일러 오류를 생성합니다.

```
__declspec( dllimport ) int func() {    // Error; dllimport
                                         // prohibited on definition.
    return 1;
}
```

다음 코드도 오류를 생성합니다.

```
__declspec( dllimport ) int i = 10; // Error; this is a definition.
```

그러나 다음은 올바른 구문입니다.

```
__declspec( dllexport ) int i = 10; // Okay--export definition
```

`dllimport` 가 선언을 암시하는 반면, `dllexport`는 정의를 암시합니다. `extern`에 `dllexport` 키워드를 사용하여 선언을 강제해야 합니다. 그렇지 않으면 정의가 암시됩니다. 따라서 다음 예제는 올바릅니다.

```
#define DllImport __declspec( dllimport )
#define DllExport __declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
                      // declaration.
```

다음 예제를 보면 위의 예제를 쉽게 이해할 수 있습니다.

```
static __declspec( dllexport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s; // Error; not declared
                                         // extern.
    __declspec( dllexport ) int m;      // Okay; this is a
                                         // declaration.
    __declspec( dllexport ) int n;      // Error; implies external
                                         // definition in local scope.
    extern __declspec( dllimport ) int i; // Okay; this is a
                                         // declaration.
    extern __declspec( dllexport ) int k; // Okay; extern implies
                                         // declaration.
    __declspec( dllexport ) int x = 5;   // Error; implies external
                                         // definition in local scope.
}
```

Microsoft 전용 종료

참고 항목

[dllexport, dllimport](#)

dllexport 및 dllimport로 인라인 C++ 함수 정의

아티클 • 2023. 10. 12.

Microsoft 전용

dllexport 특성으로 함수를 인라인으로 정의할 수 있습니다. 이 경우 프로그램의 임의의 모듈이 함수를 참조하는지 여부와 상관없이 함수가 항상 인스턴스화되어 내보내집니다. 함수는 다른 프로그램에서 가져올 수 있는 것으로 간주됩니다.

dllimport 특성으로 선언된 함수를 인라인으로 정의할 수도 있습니다. 이 경우 함수는 /Ob 지정에 따라 확장될 수 있으나 인스턴스화되지는 않습니다. 특히 인라인으로 가져온 함수의 주소가 사용된 경우 DLL에 상주하는 함수의 주소가 반환됩니다. 이 동작은 인라인으로 가져온 함수가 아닌 함수의 주소를 사용할 때와 동일합니다.

이러한 규칙은 정의가 클래스 정의 내에서 나타나는 인라인 함수에 적용됩니다. 또한, 인라인 함수의 정적 로컬 데이터 및 문자열은 단일 프로그램(즉, DLL 인터페이스가 없는 실행 파일)과 마찬가지로 DLL과 클라이언트 사이에 같은 ID를 유지합니다.

가져온 인라인 함수를 제공할 때 특별히 주의해야 합니다. 예를 들어 DLL을 업데이트하는 경우 클라이언트에서 DLL의 변경된 버전을 사용하는 것으로 단정하지 마십시오. 적합한 버전의 DLL을 로드하려면 DLL의 클라이언트를 다시 빌드하십시오.

Microsoft 전용 종료

참고 항목

[dllexport, dllimport](#)

일반 규칙 및 제한 사항

아티클 • 2023. 10. 12.

Microsoft 전용

- 또는 특성 없이 함수 또는 개체를 `dllimport dllexport` 선언하는 경우 함수 또는 개체는 DLL 인터페이스의 일부로 간주되지 않습니다. 따라서 해당 모듈 또는 같은 프로그램의 다른 모듈에 함수 또는 개체의 정의가 있어야 합니다. 함수 또는 개체를 DLL 인터페이스의 일부로 만들려면 다른 모듈에서 함수 또는 개체의 정의를 다음과 같이 `dllexport` 선언해야 합니다. 그렇게 하지 않으면 링커 오류가 생성됩니다.

특성을 사용하여 함수 또는 개체를 `dllexport` 선언하는 경우 해당 정의가 동일한 프로그램의 일부 모듈에 표시되어야 합니다. 그렇게 하지 않으면 링커 오류가 생성됩니다.

- 프로그램의 단일 모듈에 동일한 함수 또는 개체 `dllexport`에 대한 선언과 `dllexport` 둘 다 `dllimport` 포함된 경우 특성이 특성보다 `dllimport` 우선합니다. 그러나 이 경우 컴파일러 경고가 생성됩니다. 예시:

C++

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i; // Warning; inconsistent;
                             // dllexport takes precedence.
```

- C++에서는 전역적으로 선언되거나 정적 로컬 데이터 포인터를 초기화하거나 특성을 사용하여 선언된 `dllimport` 데이터 개체의 주소를 사용하여 초기화하여 C에서 오류를 생성할 수 있습니다. 또한 특성으로 선언된 함수의 주소를 사용하여 정적 로컬 함수 포인터를 초기화할 `dllimport` 수 있습니다. C에서는 이러한 대입으로 인해 포인터가 함수의 주소 대신 DLL 가져오기 쟁크(함수로 제어를 전송하는 코드 스텝)의 주소로 설정됩니다. C++에서는 포인터가 함수의 주소로 설정됩니다. 예시:

C++

```
__declspec( dllimport ) void func1( void );
__declspec( dllimport ) int i;

int *pi = &i; // Error in C
static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                     // function in C++

void func2()
{
    static int *pi = &i; // Error in C
```

```

    static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                         // function in C++
}

```

그러나 개체 선언에 특성을 포함하는 `__declspec(dllexport)` 프로그램은 프로그램 어디가에 해당 개체에 대한 정의를 제공해야 하므로 함수의 주소를 사용하여 전역 또는 로컬 정적 함수 포인터를 `__declspec(dllexport)` 초기화할 수 있습니다. 마찬가지로 `__declspec(dllexport)` 데이터 개체의 주소로 전역 또는 로컬 정적 데이터 포인터를 초기화할 수 있습니다. 예를 들어, 다음 코드는 C 또는 C++에서 오류를 발생시키지 않습니다.

```

C++

__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                                // Okay
static void ( *pf )( void ) = &func1;          // Okay

void func2()
{
    static int *pi = &i;                      // Okay
    static void ( *pf )( void ) = &func1;      // Okay
}

```

- 기본 클래스로 표시되지 `__declspec(dllexport)` 않은 일반 클래스에 적용 `__declspec(dllexport)` 하는 경우 컴파일러는 C4275를 생성합니다.

컴파일러는 기본 클래스가 클래스 템플릿의 특수화인 경우 동일한 경고를 생성합니다. 이 문제를 해결하려면 기본 클래스를 .로 `__declspec(dllexport)` 표시합니다. 클래스 템플릿의 특수화 문제는 클래스 템플릿을 표시할 수 없는 위치를 지정

`__declspec(dllexport)` 하는 것입니다. 대신 클래스 템플릿을 명시적으로 인스턴스화하고 이 명시적 인스턴스화를 .로 `__declspec(dllexport)` 표시합니다. 예시:

```

C++

template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...

```

템플릿 인수가 파생 클래스일 경우 이 해결 방법은 실패합니다. 예시:

```

C++

class __declspec(dllexport) D : public B<D> {
// ...

```

템플릿의 일반적인 패턴이므로 컴파일러는 하나 이상의 기본 클래스가 있는 클래스에 적용되는 경우와 하나 이상의 기본 클래스가 클래스 템플릿의 특수화인 경우의 의미 체계 `__declspec(dllexport)` 를 변경했습니다. 이 경우 컴파일러는 클래스 템플릿의 특수화에 암시적으로 적용 `__declspec(dllexport)` 됩니다. 다음을 수행하고 경고를 받을 수 없습니다.

C++

```
class __declspec(dllexport) D : public B<D> {  
// ...
```

Microsoft 전용 종료

참고 항목

[__declspec\(dllexport\)](#), [__declspec\(dllimport\)](#)

C++ 클래스에서 `DllImport` 및 `DllExport` 사용

아티클 • 2023. 04. 03.

Microsoft 전용

또는 `DllExport` 특성을 사용하여 C++ 클래스를 선언할 `DllImport` 수 있습니다. 이 품은 전체 클래스를 가져오거나 내보냄을 의미합니다. 이 방법으로 내보낸 클래스를 내보낼 수 있는 클래스라고 합니다.

다음 예제에서는 내보낼 수 있는 클래스를 정의합니다. 모든 멤버 함수 및 정적 데이터를 내보냅니다.

C++

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

내보내기 가능한 클래스의 `DllImport` 멤버에서 및 `DllExport` 특성을 명시적으로 사용하는 것은 금지됩니다.

`DllExport` 클래스

클래스 `DllExport`를 선언하면 모든 멤버 함수 및 정적 데이터 멤버가 내보내집니다. 모든 해당 멤버의 정의를 동일한 프로그램에 제공해야 합니다. 그렇게 하지 않으면 링커 오류가 생성됩니다. 명시적 정의를 제공할 필요가 없는 순수 가상 함수에 이 규칙에 대한 예외가 적용됩니다. 그러나 추상 클래스에 대한 소멸자가 항상 기본 클래스의 소멸자에 의해 호출되므로 순수 가상 소멸자는 항상 정의를 제공해야 합니다. 이 규칙은 내보내기 불가능한 클래스에 대해서도 동일합니다.

클래스를 반환하는 함수 또는 클래스 형식의 데이터를 내보낼 경우 클래스를 내보내야 합니다.

`DllImport` 클래스

클래스 `DllImport`를 선언하면 모든 멤버 함수와 정적 데이터 멤버를 가져옵니다. 비클래스 형식의 `DllImport` 및 동작과 `DllExport` 달리 정적 데이터 멤버는 클래스가 정의된 동

일한 프로그램에서 `dllimport` 정의를 지정할 수 없습니다.

상속 및 내보내기 가능 클래스

내보낼 수 있는 클래스의 모든 기본 클래스는 내보낼 수 있어야 합니다. 그렇지 않으면 컴파일러 경고가 생성됩니다. 또한 클래스이기도 한 액세스 가능 멤버를 모두 내보낼 수 있어야 합니다. 이 규칙은 클래스에서 `dllexport dllimport` 상속할 클래스와 `dllimport` 클래스에서 상속할 클래스를 `dllexport` 허용합니다(후자는 권장되지 않음). 따라서 C++ 액세스 규칙에 따라 DLL의 클라이언트에 액세스할 수 있는 모든 항목은 내보낼 수 있는 인터페이스의 일부여야 합니다. 인라인 함수에서 참조되는 전용 데이터 멤버가 여기에 포함됩니다.

선택적 멤버 가져오기/내보내기

클래스 내의 멤버 함수 및 정적 데이터에는 암시적으로 외부 링크가 있으므로 전체 클래스를 `dllimport` 내보내지 않는 한 또는 `dllexport` 특성으로 선언할 수 있습니다. 전체 클래스를 가져오거나 내보낼 경우 멤버 함수 및 데이터의 명시적 선언은 또는 `dllexport`로 `dllimport` 금지됩니다. 클래스 정의 내에서 정적 데이터 멤버를 `dllexport` 선언하는 경우 정의는 클래스가 아닌 외부 링크와 마찬가지로 동일한 프로그램 내 어딘가에서 발생해야 합니다.

마찬가지로 또는 `dllexport` 특성을 사용하여 멤버 함수를 선언할 `dllimport` 수 있습니다. 이 경우 동일한 프로그램 내 어딘가에 `dllexport` 정의를 제공해야 합니다.

선택적 멤버 가져오기 및 내보내기와 관련하여 몇 가지 중요한 사항을 기억하십시오.

- 선택적 멤버 가져오기/내보내기는 더 제한적으로 내보낸 클래스 인터페이스의 버전을 제공하는 데 주로 사용됩니다. 즉, 언어가 허용하는 것보다 적게 공개 및 전용 기능을 노출하는 DLL을 디자인할 수 있습니다. 내보낼 수 있는 인터페이스를 자세히 조정하는 데에도 유용합니다. 정의에 따라 클라이언트가 일부 전용 데이터에 액세스할 수 없는 경우 전체 클래스를 내보낼 필요가 없습니다.
- 클래스의 가상 함수 한 개를 내보내는 경우 가상 함수를 모두 내보내거나 적어도 클라이언트가 직접 사용할 수 있는 버전을 제공해야 합니다.
- 선택적 멤버 가져오기/내보내기가 가상 함수와 함께 사용되는 클래스가 있을 경우 함수가 내보내기 가능 인터페이스에 있거나 인라인으로 정의되어야 합니다(클라이언트가 볼 수 있음).
- 멤버를 `dllexport` 정의하지만 클래스 정의에 포함하지 않으면 컴파일러 오류가 생성됩니다. 클래스 헤더에 멤버를 정의해야 합니다.

- 또는 로 클래스 멤버 `dllimport` `dllexport` 의 정의가 허용되지만 클래스 정의에 지정된 인터페이스를 재정의할 수는 없습니다.
- 선언한 클래스 정의의 본문이 아닌 다른 위치에서 멤버 함수를 정의하는 경우 함수가 또는 `dllimport` (`dllexport`이 정의가 클래스 선언에 지정된 것과 다른 경우) 경고가 생성됩니다.

Microsoft 전용 종료

참조

[dllexport, dllimport](#)

empty_bases

아티클 • 2023. 10. 12.

Microsoft 전용

C++ 표준에서는 가장 많이 파생된 개체의 크기가 0이 아니어야 하며 1바이트 이상의 스토리지를 차지해야 합니다. 요구 사항은 가장 파생된 개체로만 확장되므로 기본 클래스 하위 개체에는 이 제약 조건이 적용되지 않습니다. EBCO(빈 기본 클래스 최적화)는 이러한 자유를 활용합니다. 이로 인해 메모리 사용량이 감소하여 성능이 향상될 수 있습니다. Microsoft Visual C++ 컴파일러는 지금까지 EBCO에 대한 지원이 제한되었습니다. Visual Studio 2015 업데이트 3 이상 버전에서는 이 최적화를 최대한 활용하는 클래스 형식에 대한 새 `__declspec(empty_bases)` 특성을 추가했습니다.

① 중요

`__declspec(empty_bases)` 이 기능을 사용하면 ABI가 적용되는 구조 및 클래스 레이아웃이 변경될 수 있습니다. 이 스토리지 클래스 특성을 사용할 때 모든 클라이언트 코드가 코드와 구조 및 클래스에 대해 동일한 정의를 사용하는지 확인합니다.

구문

```
__declspec( empty_bases )
```

설명

Visual Studio에서 사양 `__declspec(align())`이나 `alignas()` 사양이 없는 빈 클래스의 크기는 1 바이트입니다.

C++

```
struct Empty1 {};
static_assert(sizeof(Empty1) == 1, "Empty1 should be 1 byte");
```

단일 비정적 데이터 멤버가 형식 `char` 인 클래스의 크기도 1 바이트입니다.

C++

```
struct Struct1
{
    char c;
```

```
};

static_assert(sizeof(Struct1) == 1, "Struct1 should be 1 byte");
```

클래스 계층 구조에서 이러한 클래스를 결합하면 크기가 1바이트인 클래스도 생성됩니다.

C++

```
struct Derived1 : Empty1
{
    char c;
};

static_assert(sizeof(Derived1) == 1, "Derived1 should be 1 byte");
```

이 결과는 2바이트 크기(1바이트 및 1 `Empty1` `Derived1::c` 바이트)가 없으면 `Derived1` 작업 중인 빈 기본 클래스 최적화입니다. 빈 클래스 체인이 있는 경우에도 클래스 레이아웃이 최적입니다.

C++

```
struct Empty2 : Empty1 {};
struct Derived2 : Empty2
{
    char c;
};

static_assert(sizeof(Derived2) == 1, "Derived2 should be 1 byte");
```

그러나 Visual Studio의 기본 클래스 레이아웃은 여러 상속 시나리오에서 EBCO를 활용하지 않습니다.

C++

```
struct Empty3 {};
struct Derived3 : Empty2, Empty3
{
    char c;
};

static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // Error
```

크기는 1바이트일 수 있지만 `Derived3` 기본 클래스 레이아웃의 크기는 2바이트입니다. 클래스 레이아웃 알고리즘은 두 개의 연속 빈 기본 클래스 사이에 1바이트 안쪽 여백을 추가하여 결과적으로 `Empty2` 다음 내에서 `Derived3` 추가 바이트를 사용합니다.

```
class Derived3 size(2):
    +---  
0 | +--- (base class Empty2)
0 | | +--- (base class Empty1)
| | +---  
| +---  
1 | +--- (base class Empty3)
| +---  
1 | c
+---
```

이 최적이외 레이아웃의 효과는 이후 기본 클래스 또는 멤버 하위 개체의 맞춤 요구 사항이 추가 안쪽 여백을 강제로 적용할 때 복합됩니다.

C++

```
struct Derived4 : Empty2, Empty3
{
    int i;
};

static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // Error
```

형식 `int` 의 개체에 대한 자연 맞춤은 4바이트이므로 다음을 올바르게 정렬 `Derived4::i` 하려면 3바이트의 추가 안쪽 여백을 추가 `Empty3` 해야 합니다.

```
class Derived4 size(8):
    +---  
0 | +--- (base class Empty2)
0 | | +--- (base class Empty1)
| | +---  
| +---  
1 | +--- (base class Empty3)
| +---  
| <alignment member> (size=3)
4 | i
+---
```

기본 클래스 레이아웃의 또 다른 문제는 빈 기본 클래스가 클래스의 끝을 지나 오프셋에 배치될 수 있다는 것입니다.

C++

```
struct Struct2 : Struct1, Empty1
{
```

```
};

static_assert(sizeof(Struct2) == 1, "Struct2 should be 1 byte");
```

```
class Struct2 size(1):
+---  
0 | +--- (base class Struct1)  
0 | | c  
| +---  
1 | +--- (base class Empty1)  
| +---  
+---
```

Struct2 최적 크기 Empty1 이지만 오프셋 1 내에 Struct2 배치되지만 Struct2 크기를 고려하여 증가하지는 않습니다. 따라서 개체 배열 A의 Struct2 경우 하위 개체 A[0]의 Empty1 주소는 대/소문자를 구분하지 않아야 하는 주소 A[1] 와 동일합니다. 이 문제는 내 오프셋 0 Struct2에 배치되어 하위 개체와 겹치는 Struct1 경우 Empty1 발생하지 않습니다.

이러한 제한 사항을 해결하고 EBCO를 완전히 활용하도록 기본 레이아웃 알고리즘이 수정되지 않았습니다. 이러한 변경으로 이진 호환성이 손상됩니다. EBCO의 결과로 클래스의 기본 레이아웃이 변경된 경우 클래스 정의를 포함하는 모든 개체 파일 및 라이브러리를 다시 컴파일해야 클래스 레이아웃에 모두 동의할 수 있습니다. 또한 이 요구 사항은 외부 원본에서 가져온 라이브러리로 확장됩니다. 이러한 라이브러리의 개발자는 다른 버전의 컴파일러를 사용하는 고객을 지원하기 위해 EBCO 레이아웃과 함께 컴파일된 독립 버전을 제공해야 합니다. 기본 레이아웃을 변경할 수는 없지만 클래스 특성을 추가하여 __declspec(empty_bases) 클래스별로 레이아웃을 변경하는 방법을 제공할 수 있습니다. 이 특성으로 정의된 클래스는 EBCO를 최대한 활용할 수 있습니다.

C++

```
struct __declspec(empty_bases) Derived3 : Empty2, Empty3
{
    char c;
};

static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // No Error
```

```
class Derived3 size(1):
+---  
0 | +--- (base class Empty2)  
0 | | +--- (base class Empty1)  
| | +---
```

```
    | +---  
0 | +--- (base class Empty3)  
| +---  
0 | c  
+---
```

모든 하위 개체 `Derived3` 는 오프셋 0에 배치되고 크기는 최적 1바이트입니다. 기억 `_declspec(empty_bases)` 해야 할 한 가지 중요한 점은 적용된 클래스의 레이아웃에만 영향을 줍니다. 기본 클래스에는 재귀적으로 적용되지 않습니다.

C++

```
struct __declspec(empty_bases) Derived5 : Derived4  
{  
};  
static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // Error
```

```
class Derived5 size(8):  
    +---  
0 | +--- (base class Derived4)  
0 | | +--- (base class Empty2)  
0 | | | +--- (base class Empty1)  
| | +---  
| +---  
1 | | +--- (base class Empty3)  
| +---  
| | <alignment member> (size=3)  
4 | | i  
| +---  
+---
```

적용 `_declspec(empty_bases)` 되었지만 `Derived5` 직접 빈 기본 클래스가 없으므로 EBCO에 적합하지 않으므로 아무런 영향도 주지 않습니다. 그러나 대신 EBCO에 적합한 기본 클래스에 `Derived4` 적용되는 경우 둘 다 `Derived4 Derived5` 최적의 레이아웃을 갖습니다.

C++

```
struct __declspec(empty_bases) Derived4 : Empty2, Empty3  
{  
    int i;  
};  
static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // No  
Error  
  
struct Derived5 : Derived4  
{
```

```
};

static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // No
Error
```

```
class Derived5 size(4):
+---
0 | +--- (base class Derived4)
0 | | +--- (base class Empty2)
0 | | | +--- (base class Empty1)
| | |
| |
0 | | +--- (base class Empty3)
| |
0 | | i
| |
+---
```

모든 개체 파일 및 라이브러리가 클래스 레이아웃 `_declspec(empty_bases)`에 동의해야 하기 때문에 제어하는 클래스에만 적용할 수 있습니다. 표준 라이브러리의 클래스 또는 EBCO 레이아웃으로 다시 컴파일되지 않은 라이브러리에 포함된 클래스에는 적용할 수 없습니다.

Microsoft 전용 종료

참고 항목

[_declspec](#)

[키워드](#)

jitintrinsic

아티클 • 2023. 10. 12.

64비트 공용 언어 런타임에 중요한 항목으로 함수를 표시합니다. 이는 Microsoft에서 제공하는 라이브러리의 특정 함수에서 사용됩니다.

구문

```
__declspec(jitintrinsic)
```

설명

jitintrinsic 는 함수 서명에 MODOPT([IsJitIntrinsic](#))를 추가합니다.

예기치 않은 결과가 발생할 수 있으므로 사용자는 이 **__declspec** 한정자를 사용하지 않는 것이 좋습니다.

참고 항목

[__declspec](#)

[키워드](#)

naked(C++)

아티클 • 2024. 11. 21.

Microsoft 전용

특성으로 선언된 함수의 **naked** 경우 컴파일러는 프롤로그 및 에필로그 코드 없이 코드를 생성합니다. 이 기능을 이용하여 인라인 어셈블러 코드로 사용자 정의 프롤로그/에필로그 코드 시퀀스를 작성할 수 있습니다. naked 함수는 가상 디바이스 드라이버 작성에 특히 유용합니다. **naked** 이 특성은 x86 및 ARM에서만 유효하며 x64에서는 사용할 수 없습니다.

구문

```
_declspec(naked) declarator
```

설명

naked 특성은 함수 정의와만 관련이 있고 형식 한정자가 아니므로 naked 함수는 확장 특성 구문과 **_declspec 키워드**를 사용해야 합니다.

함수가 **_forceinline** 키워드로도 표시된 경우에도 컴파일러는 naked 특성으로 표시된 함수에 대한 [인라인 함수를](#) 생성할 수 없습니다.

특성이 멤버가 아닌 메서드의 **naked** 정의 이외의 항목에 적용되는 경우 컴파일러에서 오류를 발생합니다.

예제

이 코드는 특성을 사용하여 함수를 **naked** 정의합니다.

```
_declspec( naked ) int func( formal_parameters ) {}
```

또는 다음을 수행합니다.

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

naked 특성은 함수의 프롤로그 및 에필로그 시퀀스에 사용되는 컴파일러 코드 생성에만 영향을 줍니다. 이러한 함수를 호출하기 위해 생성되는 코드에는 영향을 주지 않습니다. 따라서 **naked** 특성을 함수 형식의 일부로 간주하지 않으며, 함수 포인터는 **naked** 특성을 가질 수 없습니다. 또한 **naked** 특성은 데이터 정의에 적용될 수 없습니다. 예를 들어 이 코드 샘플은 오류를 생성합니다.

```
_declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

naked 특성은 함수 정의에만 관련되며, 함수의 프로토타입에 지정될 수 없습니다. 예를 들어 이 선언은 컴파일러 오류를 생성합니다.

```
_declspec( naked ) int func(); // Error--naked attribute not permitted on
function declarations
```

Microsoft 전용 종료

참고 항목

[_declspec](#)

[키워드](#)

[Naked 함수 호출](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

noalias

아티클 • 2023. 10. 12.

Microsoft 전용

`noalias`는 함수 호출이 표시되는 전역 상태를 수정하거나 참조하지 않으며 포인터 매개 변수(첫 번째 수준 간접 참조)로 직접 가리키는 메모리만 수정한다는 것을 의미합니다.

함수에 주석이 추가 `noalias`된 경우 최적화 프로그램은 매개 변수 자체와 포인터 매개 변수의 첫 번째 수준 간접 참조만 함수 내에서 참조되거나 수정된다고 가정할 수 있습니다.

주석은 `noalias` 주석이 추가된 함수의 본문 내에서만 적용됩니다. 함수를 표시 `_declspec(noalias)` 해도 함수에서 반환하는 포인터의 별칭에는 영향을 주지 않습니다.

별칭에 영향을 미칠 수 있는 다른 주석은 다음을 참조하세요 [_declspec\(restrict\)](#).

예시

다음 샘플에서는 .의 `_declspec(noalias)` 사용을 보여 줍니다.

메모리에 액세스하는 함수 `multiply`에 주석이 추가되면 `_declspec(noalias)`이 함수는 매개 변수 목록의 포인터를 제외하고는 전역 상태를 수정하지 않는다는 것을 컴파일러에 알릴 수 있습니다.

```
C

// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

float * init(int m, int n)
{
```

```

float * a;
int i, j;
int k=1;

a = ma(m * n);
if (!a) exit(1);
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        a[i*n+j] = 0.1/k++;
return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}

```

참고 항목

_declspec

키워드

_declspec(restrict)

noinline

아티클 • 2023. 10. 12.

Microsoft 전용

`__declspec(noinline)` 는 특정 멤버 함수(클래스의 함수)를 인라인하지 않도록 컴파일러에 지시합니다.

함수가 작고 코드 성능에 심각한 영향을 주지 않는 경우 함수를 인라인 처리하지 않는 것이 적합할 수 있습니다. 즉, 오류 조건을 처리하는 함수처럼 함수가 작고 자주 호출될 가능성이 적은 경우가 여기에 해당합니다.

함수가 표시된 `noinline` 경우 호출 함수는 더 작아지므로 컴파일러 인라인 처리의 후보가 됩니다.

C++

```
class X {
    __declspec(noinline) int mbrfunc() {
        return 0;
    } // will not inline
};
```

Microsoft 전용 종료

참고 항목

[__declspec](#)

키워드

[inline](#), [_inline](#), [_forceinline](#)

noreturn

아티클 • 2023. 11. 16.

Microsoft 전용

이 특성은 `_declspec` 함수가 반환하지 않는다는 것을 컴파일러에 알려줍니다. 그런 다음 컴파일러는 함수를 호출한 다음 코드에 연결할 수 없다는 것을 `_declspec(noreturn)` 알고 있습니다.

컴파일러가 값을 반환하지 않는 제어 경로를 가진 함수를 발견할 경우 경고(C4715) 또는 오류 메시지(C2202)가 생성됩니다. 반환되지 않는 함수로 인해 컨트롤 경로에 연결할 수 없는 경우 이 경고 또는 오류를 방지하는 데 사용합니다 `_declspec(noreturn)`.

① 참고

반환할 것으로 예상되는 함수에 추가 `_declspec(noreturn)` 하면 정의되지 않은 동작이 발생할 수 있습니다.

예시

다음 예제에서는 인수 `isZeroOrPositive` 가 음 `fatal` 수이면 호출됩니다. 해당 컨트롤 경로에는 반환 문이 없으므로 모든 컨트롤 경로가 값을 반환하지 않는다는 경고 C4715가 발생합니다. 로 `_declspec(noreturn)` 선언하면 `fatal` 해당 경고가 완화됩니다. 이 경고는 프로그램을 종료한 이후 `fatal()` 아무 소용이 없기 때문에 바람직합니다.

C++

```
// noreturn2.cpp
#include <exception>

__declspec(noreturn) void fatal()
{
    std::terminate();
}

int isZeroOrPositive(int val)
{
    if (val == 0)
    {
        return 0;
    }
    else if (val > 0)
    {
        return 1;
```

```
    }
    // this function terminates if val is negative
    fatal();
}

int main()
{
    isZeroOrPositive(123);
}
```

Microsoft 전용 종료

참고 항목

[_declspec](#)

[키워드](#)

no_sanitize_address

아티클 • 2023. 10. 12.

Microsoft 전용

`__declspec(no_sanitize_address)` 지정자는 함수, 지역 변수 또는 전역 변수에서 주소 삐제기를 사용하지 않도록 컴파일러에 지시합니다. 이 지정자는 AddressSanitizer와 함께 사용됩니다.

① 참고

`__declspec(no_sanitize_address)`는 런타임 동작이 아니라 컴파일러 동작을 사용하지 않도록 설정합니다.

예시

예제는 [AddressSanitizer 빌드 참조](#)를 참조하세요.

Microsoft 전용 종료

참고 항목

[__declspec](#)

[키워드](#)

[AddressSanitizer](#)

nothrow (C++)

아티클 • 2024. 07. 08.

Microsoft 전용

함수 선언에서 사용할 수 있는 `_declspec` 확장 특성입니다.

구문

```
return-type _declspec(nothrow) [call-convention] function-name ([argument-list])
```

설명

모든 새 코드에서는 `_declspec(nothrow)` 대신 `noexcept` 연산자를 사용하는 것이 좋습니다.

이 특성은 컴파일러에게 선언한 함수와 이 함수가 요청한 함수들이 예외를 `throw`하지 않도록 명령합니다. 그러나 지시문을 적용하지는 않습니다. 즉, `noexcept` 또는 `std:c++17` 모드(Visual Studio 2017 버전 15.5 이상), `throw()` 와 달리 `std::terminate`가 호출되지 않습니다.

이제 기본값인, 동기 예외 처리 모델을 이용하여 컴파일러는 해제할 수 있는 특정 개체의 수명 추적 메커니즘을 제거할 수 있으며, 코드 크기를 크게 줄일 수 있습니다. 다음 전처리기 지시문을 제공할 경우 아래 세 가지 함수 선언은 `/std:c++14` 모드에서 동일합니다.

C++

```
#define WINAPI _declspec(nothrow) _stdcall

void WINAPI f1();
void _declspec(nothrow) _stdcall f2();
void _stdcall f3() throw();
```

`/std:c++17` 모드에서 `throw()` 는 함수에서 예외가 `throw`되면 `std::terminate`가 호출되도록 하기 때문에 `_declspec(nothrow)` 를 사용하는 다른 모드와 동등하지 않습니다.

`void _stdcall f3() throw();` 선언은 C++ 표준에 정의된 구문을 사용합니다. C++17에서는 `throw()` 키워드가 더 이상 사용되지 않습니다.

Microsoft 전용 종료

참고 항목

[_declspec](#)

[noexcept](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

novtable

아티클 • 2023. 10. 12.

Microsoft 전용

확장된 특성입니다 `__declspec`.

이 형식은 `__declspec` 모든 클래스 선언에 적용할 수 있지만 순수 인터페이스 클래스, 즉 자체적으로 인스턴스화되지 않는 클래스에만 적용해야 합니다. `__declspec` 컴파일러가 코드를 생성하지 못하게 하여 클래스의 생성자 및 소멸자에서 vptr을 초기화합니다. 대부분의 경우 이렇게 하면 클래스와 연결된 vtable에 대한 참조만 제거되므로 링커가 이를 제거합니다. 이 형식을 `__declspec` 사용하면 코드 크기가 크게 감소할 수 있습니다.

표시된 `novtable` 클래스를 인스턴스화한 다음 클래스 멤버에 액세스하려고 하면 AV(액세스 위반)가 수신됩니다.

예시

C++

```
// novtable.cpp
#include <stdio.h>

struct __declspec(novtable) X {
    virtual void mf();
};

struct Y : public X {
    void mf() {
        printf_s("In Y\n");
    }
};

int main() {
    // X *pX = new X();
    // pX->mf();    // Causes a runtime access violation.

    Y *pY = new Y();
    pY->mf();
}
```

Output

```
In Y
```

참고 항목

[_declspec](#)

[키워드](#)

process

아티클 • 2024. 11. 21.

프로세스의 모든 애플리케이션 도메인에서 공유되는 특정 전역 변수, 정적 멤버 변수 또는 정적 지역 변수의 단일 복사본이 관리되는 애플리케이션 프로세스에 있어야 함을 지정합니다. 이는 주로 Visual Studio 2015에서 사용되지 않고 Visual Studio 2017에서 지원되지 않는 컴파일 시 `/clr:pure` 사용됩니다. 전역 및 정적 변수를 사용하여 `/clr` 컴파일하는 경우 기본적으로 프로세스당 변수이며 사용할 `_declspec(process)` 필요가 없습니다.

전역 변수, 정적 멤버 변수 또는 네이티브 형식의 정적 지역 변수만으로 `_declspec(process)` 표시할 수 있습니다.

`process` 는 .을 사용하여 `/clr` 컴파일할 때만 유효합니다.

각 애플리케이션 도메인에 전역 변수의 자체 복사본이 있도록 하려면 `appdomain`을 사용합니다.

자세한 내용은 애플리케이션 도메인 및 Visual C++ [를 참조하세요](#).

참고 항목

[_declspec](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

속성 (C++)

아티클 • 2023. 10. 12.

Microsoft 전용

이 특성은 클래스 또는 구조체 정의에서 비정적 "가상 데이터 멤버"에 적용할 수 있습니다. 컴파일러는 해당 참조를 함수 호출로 변경하여 이러한 "가상 데이터 멤버"를 데이터 멤버로 처리합니다.

구문

```
__declspec( property( get=get_func_name ) ) declarator  
__declspec( property( put=put_func_name ) ) declarator  
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

설명

컴파일러에서 멤버 선택 연산자("." 또는 "->")의 오른쪽에 이 특성으로 선언된 데이터 멤버를 볼 때 해당 식이 l-value인지 r-value인지에 따라 작업을 `get` `put` `a` 또는 함수로 변환합니다. "`+=`"와 같은 더 복잡한 컨텍스트에서 다시 쓰기는 둘 다 `get` `put` 수행하여 수행됩니다.

이 특성은 클래스 또는 구조체 정의에 있는 빈 배열의 선언에서도 사용할 수 있습니다. 예시:

C++

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

위의 문은 `x[]`를 하나 이상의 배열 인덱스와 함께 사용할 수 있음을 나타냅니다. 이 경우 `i=p->x[a][b]` 가 `i=p->GetX(a, b)`로 바뀌고 `p->x[a][b] = i` 가 `p->PutX(a, b, i);`로 바뀝니다.

Microsoft 전용 종료

예시

C++

```
// declspec_property.cpp
struct S {
    int i;
    void putprop(int j) {
        i = j;
    }

    int getprop() {
        return i;
    }
};

__declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
    S s;
    s.the_prop = 5;
    return s.the_prop;
}
```

참고 항목

[_declspec](#)

[키워드](#)

restrict

아티클 • 2024. 07. 20.

Microsoft 전용

포인터 형식을 반환하는 함수 선언 또는 정의에 적용될 때 `restrict` 은(는) 함수가 앤리어싱되지 않은 개체, 즉 다른 포인터에 의해 참조되는 개체를 반환한다는 것을 컴파일러에 알려줍니다. 이렇게 하면 컴파일러가 추가 최적화를 수행할 수 있습니다.

구문

```
| _declspec(restrict) pointer_return_type 함수();
```

설명

컴파일러가 `_declspec(restrict)` 을(를) 전파합니다. 예를 들어 CRT `malloc` 함수에는 `_declspec(restrict)` 장식이 있으므로 컴파일러는 `malloc`에 의해 메모리 위치로 초기화된 포인터도 기존 포인터에 의해 별칭이 지정되지 않는다고 가정합니다.

컴파일러는 반환된 포인터가 실제로 별칭이 지정되지 않았는지 확인하지 않습니다. 프로그램이 `restrict` `_declspec` 한정자로 표시된 포인터에 별칭을 지정하지 않도록 하는 것은 개발자의 책임입니다.

변수에 대한 유사한 의미 체계를 보려면 [_restrict](#)를 참조하세요.

함수 내의 앤리어싱에 적용되는 다른 주석은 [_declspec\(noalias\)](#)를 참조하세요.

C++ AMP의 일부인 `restrict` 키워드에 대한 자세한 내용은 [restrict\(C++ AMP\)](#)를 참조하세요.

예시

다음 샘플에서는 `_declspec(restrict)` 을(를) 사용하는 방법을 보여 줍니다.

포인터를 반환하는 함수에 `_declspec(restrict)` 을(를) 적용하면 반환 값이 가리키는 메모리에 별칭이 지정되지 않음을 컴파일러에 알릴 수 있습니다. 이 예제에서 `mempool` 및 `memptr` 포인터는 전역이므로 컴파일러는 참조하는 메모리에 별칭이 지정되지 않았는지 확인할 수 없습니다. 그러나 프로그램에서 참조하지 않는 메모리를 반환하는 방식으로 `ma` 및 해당 호출자 `init` 이(가) 사용되므로 최적화 프로그램을 돋기 위해 `_declspec(restrict)`이 사용됩니다. 이는 CRT 헤더가 `_declspec(restrict)` 을(를) 사용하

여기 기존 포인터로 별칭을 지정할 수 없는 메모리를 항상 반환함을 나타내기 위해 `malloc` 와(과) 같은 할당 함수를 데코레이트하는 방법과 비슷합니다.

C

```
// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1f/k++;
    return a;
}

void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;
```

```
mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));  
  
if (!mempool)  
{  
    puts("ERROR: Malloc returned null");  
    exit(1);  
}  
  
memptr = mempool;  
a = init(M, N);  
b = init(N, P);  
c = init(M, P);  
  
multiply(a, b, c);  
}
```

Microsoft 전용 종료

참고 항목

키워드

[_declspec](#)

[_declspec\(noalias\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

safebuffer

아티클 • 2024. 11. 21.

Microsoft 전용

함수에 대한 버퍼 오버런 보안 검사를 삽입하지 않도록 컴파일러에 지시합니다.

구문

```
_declspec( safebuffers )
```

설명

/GS 컴파일러 옵션을 사용하면 컴파일러가 스택에 보안 검사를 삽입하여 버퍼 오버런을 테스트합니다. 보안 검사에 적합한 데이터 구조 형식은 /GS(버퍼 보안 검사)에 설명되어 있습니다. 버퍼 오버런 검색에 대한 자세한 내용은 MSVC의 [보안 기능을 참조](#)하세요.

전문가 수동 코드 검토 또는 외부 분석이 함수가 버퍼 오버런으로부터 안전한지 확인할 수 있습니다. 이 경우 함수 선언에 키워드를 적용하여 `_declspec(safebuffers)` 함수에 대한 보안 검사를 표시하지 않을 수 있습니다.

⊗ 주의

그러나 버퍼 보안 검사는 중요한 보안 보호를 제공하고 성능에 별다른 영향을 미치지 않으므로, 함수의 성능이 매우 중요하고 해당 함수의 안전성이 파악되는 드문 경우를 제외하고, 버퍼 보안 검사를 억제하지 않도록 권장합니다.

인라인 함수

주 함수는 [인라인 키워드를 사용하여](#) 보조 함수의 복사본을 삽입할 수 있습니다. 키워드가 `_declspec(safebuffers)` 함수에 적용되면 해당 함수에 대해 버퍼 오버런 검색이 표시되지 않습니다. 그러나 인라인 처리는 `_declspec(safebuffers)` 다음과 같은 방법으로 키워드에 영향을 줍니다.

/GS 컴파일러 옵션이 두 함수 모두에 대해 지정되었지만 주 함수가 키워드를 `_declspec(safebuffers)` 지정한다고 가정합니다. 보조 함수의 데이터 구조는 보안 검사

를 가능하게 하기 때문에 이 함수는 보안 검사를 억제하지 않습니다. 이 경우 다음과 같습니다.

- 컴파일러 최적화에 관계없이 컴파일러가 해당 함수를 인라인으로 강제 적용하도록 보조 함수에 `_forceinline` 키워드를 지정합니다.
- 보조 함수는 보안 검사에 적합하므로 키워드를 지정하더라도 보안 검사가 기본 함수에도 `_declspec(safebuffers)` 적용됩니다.

예시

다음 코드에서는 키워드를 사용하는 `_declspec(safebuffers)` 방법을 보여줍니다.

```
C++  
  
// compile with: /c /GS  
typedef struct {  
    int x[20];  
} BUFFER;  
static int checkBuffers() {  
    BUFFER cb;  
    // Use the buffer...  
    return 0;  
};  
static __declspec(safebuffers)  
int noCheckBuffers() {  
    BUFFER ncb;  
    // Use the buffer...  
    return 0;  
}  
int wmain() {  
    checkBuffers();  
    noCheckBuffers();  
    return 0;  
}
```

Microsoft 전용 종료

참고 항목

[__declspec](#)

키워드

[inline](#), [_inline](#), [_forceinline](#)

[strict_gs_check](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

selectany

아티클 • 2024. 07. 15.

Microsoft 전용

선언된 전역 데이터 항목(변수 또는 개체)이 pick-any COMDAT(패키지된 함수)임을 컴파일러에 알립니다.

구문

`_declspec(selectany) 선언자`

설명

링크 타임에 COMDAT의 정의가 여러 개 표시되면 링커가 그 중 하나를 선택하고 나머지는 무시합니다. 링커 옵션 `/OPT:REF`(최적화)가 선택되면 링커 출력에서 참조하지 않는 데터 항목을 모두 제거하기 위해 COMDAT가 제거됩니다.

선언에서 전역 함수 또는 정적 메서드에 의한 생성자 및 할당은 참조를 만들지 않으며 `/OPT:REF` 제거를 막지 않습니다. 데이터에 대한 다른 참조가 없을 경우 그러한 코드로 인해 의도하지 않은 결과가 발생해서는 안 됩니다.

동적으로 초기화되는 전역 개체의 경우, `selectany`는 참조하지 않는 개체의 초기화 코드도 삭제합니다.

보통 EXE 또는 DLL 프로젝트에서 한 번만 전역 데이터 항목을 초기화할 수 있습니다. 둘 이상의 소스 파일에 같은 헤더가 나타날 경우, 헤더에 정의된 전역 데이터를 초기화하는 데 `selectany`를 사용할 수 있습니다. C 및 C++ 컴파일러 모두에서 `selectany`를 사용할 수 있습니다.

① 참고

외부에 표시되는 전역 데이터 항목의 실제 초기화에 한해 `selectany`를 적용할 수 있습니다.

예제: `selectany` 특성

이 코드에서는 `selectany` 특성을 사용하는 방법을 보여 줍니다.

C++

```
//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;};
int i;
};

__declspec(selectany) X x(1);
```

예제: `selectany` 특성을 사용한 데이터 COMDAT 정리

이 코드에서는 `/OPT:ICF` 링커 옵션도 사용하는 경우 `selectany` 특성을 사용하여 데이터 COMDAT 정리를 수행하는 방법을 보여 줍니다. 데이터는 `selectany`로 표시되고 `const`(읽기 전용) 섹션에 있어야 한다는 점에 유의하세요. 읽기 전용 섹션을 명시적으로 지정해야 합니다.

C++

```
// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
    int ij;
    ij = ix + jx;
}
```

참고 항목

[_declspec](#)

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

spectre

아티클 • 2024. 11. 21.

Microsoft 전용

함수에 대한 Spectre variant 1 추측 실행 장벽 명령을 삽입하지 않도록 컴파일러에 지시합니다.

구문

```
__declspec( spectre(nomitigation) )
```

설명

/Qspectre 컴파일러 옵션을 사용하면 컴파일러가 추측 실행 장벽 명령을 삽입합니다. 스펙터 변형 1 보안 취약성이 존재한다는 분석이 표시되는 위치에 삽입됩니다. 내보내는 특정 지침은 프로세서에 따라 달라집니다. 이러한 지침은 코드 크기 또는 성능에 최소한의 영향을 주어야 하지만 코드가 취약성의 영향을 받지 않고 최대 성능이 필요한 경우가 있을 수 있습니다.

전문가 분석을 통해 스펙터 변형 1 경계에서 함수가 안전한지 검사 바이패스 결함을 확인할 수 있습니다. 이 경우 함수 선언에 적용하여 함수 내에서 완화 코드 생성을 `__declspec(spectre(nomitigation))` 억제할 수 있습니다.

⊗ 주의

/Qspectre 투기적 실행 장벽 지침은 중요한 보안 보호를 제공하며 성능에 미미한 영향을 줍니다. 함수의 성능이 매우 중요하고 해당 함수의 안전성이 파악되는 드문 경우를 제외하고, 버퍼 보안 검사를 억제하지 않도록 권장합니다.

예시

다음 코드에서는 `__declspec(spectre(nomitigation))` 사용 방법을 보여 줍니다.

C++

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
```

```
// ...
return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

Microsoft 전용 종료

참고 항목

[_declspec](#)

[키워드](#)

[/Qspectre](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

스레드

아티클 • 2024. 07. 08.

Microsoft 전용

`thread` 확장 저장소 클래스 한정자는 스레드 로컬 변수를 선언하는 데 사용됩니다.

C++11 이상의 이식 가능한 형식의 경우, 이식 가능 코드에 대해 `thread_local` 스토리지 클래스 지정자를 사용합니다. Windows에서 `thread_local`은 `_declspec(thread)`로 구현됩니다.

구문

`_declspec(thread)` 선언자

설명

TLS(스레드 로컬 스토리지)는 다중 스레드 프로세스의 각 스레드가 스레드 데이터를 위한 스토리지를 할당하는 메커니즘입니다. 표준 다중 스레드 프로그램에서 데이터는 지정된 프로세스의 모든 스레드에서 공유되지만 스레드 로컬 스토리지는 스레드별 데이터를 할당하기 위한 메커니즘입니다. 스레드에 대한 자세한 내용은 [다중 스레딩](#)을 참조하세요.

스레드 로컬 변수의 선언은 [확장된 특성 구문](#)과 `thread` 키워드가 있는 `_declspec` 키워드를 사용해야 합니다. 예를 들어, 다음 코드는 정수 스레드 로컬 변수를 선언한 다음 값으로 초기화합니다.

C++

```
_declspec( thread ) int tls_i = 1;
```

동적으로 로드된 라이브러리에서 스레드 로컬 변수를 사용하는 경우 스레드-로컬 변수가 올바르게 초기화되지 않을 수 있는 요인을 알고 있어야 합니다.

- 함수 호출(생성자 포함)을 사용하여 변수를 초기화하는 경우 이 함수는 이진/DLL을 프로세스로 로드한 스레드와 이진/DLL이 로드된 후에 시작된 스레드에 대해서만 호출됩니다. 초기화 함수는 DLL이 로드될 때 이미 실행 중인 다른 스레드에 대해 호출되지 않습니다. 동적 초기화는 DLL_THREAD_ATTACH에 대한 DllMain 호출에서 발생하지만 스레드가 시작될 때 DLL이 프로세스에 없는 경우 DLL은 해당 메시지를 가져오지 않습니다.

2. 상수 값을 사용하여 정적으로 초기화되는 스레드 로컬 변수는 일반적으로 모든 스레드에서 제대로 초기화됩니다. 그러나 2017년 12월 현재, Microsoft C++ 컴파일러에는 `constexpr` 변수가 정적 초기화 대신 동적 초기화를 수신하는 알려진 규칙 문제가 있습니다.

참고: 이러한 두 문제는 컴파일러의 향후 업데이트에서 해결될 것으로 예상됩니다.

또한 스레드 로컬 개체 및 변수를 선언하는 경우 다음과 같은 지침을 준수해야 합니다.

- 데이터 선언 및 정의에 대해서만 `thread` 특성을 적용할 수 있습니다. `thread`는 함수 선언 또는 정의에 사용될 수 없습니다.
- 정적 스토리지 기간이 있는 데이터 항목에만 `thread` 특성을 지정할 수 있습니다. 여기에는 전역 데이터 개체(`static` 및 `extern`), 지역 정적 개체 및 클래스의 정적 데이터 멤버가 포함됩니다. 자동 데이터 개체는 `thread` 특성을 사용하여 선언할 수 없습니다.
- 스레드 로컬 개체의 선언과 정의가 같은 파일에서 발생하는지, 아니면 별도의 파일에서 발생하는지와 관계없이 해당 선언과 정의에 대해 `thread` 특성을 사용해야 합니다.
- `thread` 특성은 형식 한정자로 사용할 수 없습니다.
- `thread` 특성을 사용하는 개체를 선언할 수 있으므로 다음 두 예제는 의미 체계 면에서 같습니다.

C++

```
// declspec_thread_2.cpp
// compile with: /LD
__declspec( thread ) class B {
public:
    int data;
} BObject; // BObject declared thread local.

class B2 {
public:
    int data;
};
__declspec( thread ) B2 BObject2; // BObject2 declared thread local.
```

- 표준 C에서는 비정적 개체에 대해 자신에 대한 참조를 포함하는 식으로 개체 또는 변수를 초기화할 수 있습니다. C++에서는 일반적으로 자신에 대한 참조를 포함하는 식으로 개체를 동적으로 초기화할 수 있지만 스레드 로컬 개체에 대해서는 이렇게 초기화할 수 없습니다. 예시:

C++

```
// declspec_thread_3.cpp
// compile with: /LD
#define Thread __declspec( thread )
int j = j;    // Okay in C++; C error
Thread int tls_i = sizeof( tls_i );    // Okay in C and C++
```

초기화되는 개체를 포함하는 `sizeof` 식은 자신에 대한 참조로 간주되지 않으므로 C 와 C++에서 허용됩니다.

Microsoft 전용 종료

참고 항목

[__declspec](#)

[키워드](#)

[TLS\(스레드 로컬 스토리지\)](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

uuid (C++)

아티클 • 2023. 10. 12.

Microsoft 전용

컴파일러는 GUID를 선언되거나 정의된 클래스 또는 구조체(전체 COM 개체 정의에만 해당)에 특성과 `uuid` 연결합니다.

구문

```
__declspec( uuid("ComObjectGUID") ) declarator
```

설명

특성은 `uuid` 문자열을 인수로 사용합니다. 이 문자열은 {} 구분 기호를 사용하거나 사용하지 않는 일반 레지스트리 형식의 GUID 이름을 지정합니다. 예시:

C++

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

재선언에서 이 특성을 적용할 수 있습니다. 이렇게 하면 시스템 헤더가 GUID를 제공하는 다른 헤더(예<: comdef.h>)의 다시 선언과 같은 `IUnknown` 인터페이스의 정의를 제공할 수 있습니다.

키워드(keyword) `_uuidof` 적용하여 사용자 정의 형식에 연결된 상수 GUID를 검색할 수 있습니다.

Microsoft 전용 종료

참고 항목

[__declspec](#)

[키워드](#)

restrict

아티클 • 2024. 11. 21.

`_declspec (restrict)` 한정자와 `_restrict` 마찬가지로 키워드(두 개의 선행 밑줄 '_')는 기호가 현재 범위에서 별칭이 지정되지 않음을 나타냅니다. 키워드는 `_restrict` `_declspec (restrict)` 다음과 같은 방법으로 한정자와 다릅니다.

- `_restrict` 키워드는 변수에서만 유효하며 `_declspec (restrict)` 함수 선언 및 정의에서만 유효합니다.
- `_restrict`은 C99부터 시작하여 모드에서 [/std:c17 사용할 수 /std:c11](#) 있는 C와 유사 `restrict` 하지만 `_restrict` C++ 및 C 프로그램 모두에서 사용할 수 있습니다.
- 사용하는 경우 `_restrict` 컴파일러는 변수의 별칭 없음 속성을 전파하지 않습니다. 즉, 변수를 `_restrict` 변수 `_restrict` 가 아닌 변수에 할당하는 경우 컴파일러는 `_restrict` 아닌 변수의 별칭을 계속 허용합니다. 이는 C99 C 언어 `restrict` 키워드의 동작과 다릅니다.

일반적으로 전체 함수의 동작에 영향을 주려면 키워드 대신 사용합니다 `_declspec (restrict)`.

이전 버전과의 호환성을 위해 `_restrict`은 `_restrict`의 동의어입니다. 단, 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

Visual Studio 2015 이상 `_restrict` 에서는 C++ 참조에서 사용할 수 있습니다.

① 참고

키워드 `volatile` 가 `volatile` 있는 변수에서 사용하는 경우 우선합니다.

예시

C++

```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
    int i;
    for (i = 0; i < n; i++) {
```

```
a[i] = b[i] + c[i];
c[i] = b[i] + d[i];
}
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
    int * __restrict x;
    double * __restrict y;
};
```

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

__sptr, __uptr

아티클 • 2023. 10. 12.

Microsoft 전용

컴파일러가 **__sptr** 32비트 포인터를 64비트 포인터로 변환하는 방법을 지정하려면 32비트 포인터 선언의 한 **__uptr** 정자를 사용합니다. 예를 들어 32비트 포인터는 64비트 포인터 변수에 할당되거나 64비트 플랫폼에서 역참조될 때 변환됩니다.

64비트 플랫폼 지원에 대한 Microsoft 설명서에서는 경우에 따라 32비트 포인터의 최상위 비트를 부호 비트로 지칭합니다. 기본적으로 컴파일러는 부호 확장을 사용하여 32비트 포인터를 64비트 포인터로 변환합니다. 즉, 64비트 포인터의 최하위 32비트는 32비트 포인터의 값으로 설정되며 최상위 32비트는 32비트 포인터의 부호 비트 값으로 설정됩니다. 이 변환은 부호 비트가 0인 경우 올바른 결과를 생성하지만 부호 비트가 1인 경우에는 올바른 결과를 생성하지 않습니다. 예를 들어 32비트 주소 0x7FFFFFFF는 동일한 64비트 주소 0x000000007FFFFFFF를 생성하지만 32비트 주소 0x80000000은 0xFFFFFFFF80000000으로 올바르지 않게 변경됩니다.

__sptr 또는 부호 있는 포인터 한정자는 포인터 변환이 64비트 포인터의 가장 중요한 비트를 32비트 포인터의 부호 비트로 설정하도록 지정합니다. **__uptr** 부호 없는 포인터인 한정자는 변환이 가장 중요한 비트를 0으로 설정하도록 지정합니다. 다음 선언에서는 정규화되지 않은 포인터 2개, **__uptr __ptr32** 형식으로 정규화된 포인터 2개 및 함수 매개변수와 함께 사용되는 한정자를 보여 **__sptr** 줍니다.

C++

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

__sptr 포인터 선언과 함께 한 **__uptr** 정자를 사용합니다. 포인터 형식 [한정자의 위치](#)에 한정자를 사용합니다. 즉, 한정자가 별표 뒤에 있어야 합니다. 멤버 [에 대한 포인터와 함께](#) 한정자를 사용할 수 없습니다. 한정자는 포인터 선언이 아닌 선언에 영향을 주지 않습니다.

이전 버전과의 호환성을 위해 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)를 지정하지 않는 한 **_sptr** 및 **__uptr __uptr** 동의어 **__sptr**입니다.

예시

다음 예제에서는 한정자를 사용하는 `_sptr` 32비트 포인터를 선언하고 `_uptr` 각 32비트 포인터를 64비트 포인터 변수에 할당한 다음 각 64비트 포인터의 16진수 값을 표시합니다. 이 예제는 네이티브 64비트 컴파일러로 컴파일되며 64비트 플랫폼에서 실행됩니다.

C++

```
// sptr_uptr.cpp
// processor: x64
#include "stdio.h"

int main()
{
    void *      __ptr64 p64;
    void *      __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

    // Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

    // The printf() function automatically displays leading zeroes with each 32-bit
    // pointer. These are unrelated
    // to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit
pointer):\n");
    printf("p32d:      %p\n", p32d);
    printf("p32s:      %p\n", p32s);
    printf("p32u:      %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit
pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}
```

Output

```
Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:      0000000087654321
p32s:      0000000087654321
p32u:      0000000087654321
```

```
Display the 64-bit pointer created from each 32-bit pointer:
```

```
p32d: p64 = FFFFFFFF87654321  
p32s: p64 = FFFFFFFF87654321  
p32u: p64 = 0000000087654321
```

Microsoft 전용 종료

참고 항목

[Microsoft 전용 한정자](#)

`_unaligned`

아티클 • 2024. 11. 21.

Microsoft 관련. 한정자를 사용하여 포인터 `_unaligned` 를 선언하면 컴파일러는 포인터가 정렬되지 않은 데이터에 주소를 지정한다고 가정합니다. 따라서 플랫폼에 적합한 코드는 포인터를 통해 정렬되지 않은 읽기 및 쓰기를 처리하기 위해 생성됩니다.

설명

이 한정자는 포인터로 주소가 지정된 데이터의 맞춤을 설명합니다. 포인터 자체가 정렬된 것으로 가정합니다.

키워드의 `_unaligned` 필요성은 플랫폼 및 환경에 따라 다릅니다. 데이터를 적절하게 표시하지 못하면 성능 저하에서 하드웨어 오류에 이르기까지 다양한 문제가 발생할 수 있습니다. `_unaligned` x86 플랫폼에는 한정자가 유효하지 않습니다.

이전 버전과의 호환성을 위해 `_unaligned`은 `_unaligned`의 동의어입니다. 단, 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

정렬에 대한 자세한 내용은 다음을 참조하십시오.

- [align](#)
- [alignof 연산자](#)
- [pack](#)
- [/Zp \(구조체 멤버 맞춤\)](#)
- [x64 구조체 맞춤 예](#)

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

w64

아티클 • 2024. 07. 15.

Microsoft 전용인 이 키워드는 더 이상 사용되지 않습니다. Visual Studio 2013 이전의 Visual Studio 버전에서는 이 키워드를 통해 변수를 표시할 수 있으므로, [/Wp64](#)를 사용하여 컴파일하는 경우 컴파일러는 64비트 컴파일러를 통해 컴파일할 때 보고되는 모든 경고를 보고합니다.

구문

형식 `_w64` 식별자

매개 변수

type

32비트에서 64비트 컴파일러로 이식되는 코드에서 문제를 일으킬 수 있는 세 가지 형식 (`int`, `long`, 또는 포인터) 중 하나.

identifier

만들고 있는 변수에 대한 식별자입니다.

설명

① 중요

[/Wp64](#) 컴파일러 옵션 및 `_w64` 키워드는 Visual Studio 2010 및 Visual Studio 2013에서 더 이상 사용되지 않으며 Visual Studio 2013부터는 제거되었습니다. 명령줄에서 `/Wp64` 컴파일러 옵션을 사용하면 컴파일러에서 명령줄 경고 D9002를 표시합니다. `_w64` 키워드는 자동으로 무시됩니다. 64비트 이식성 문제를 감지하기 위해 이 옵션과 키워드를 사용하는 대신, 64비트 플랫폼을 대상으로 하는 Microsoft C++ 컴파일러를 사용합니다. 자세한 내용은 [64비트, x64 대상을 위한 Visual C++ 구성](#)을 참조하세요.

`_w64` 가 포함된 `typedef`은 x86에서는 32비트이고 x64에서는 64비트가 되어야 합니다.

Visual Studio 2010 이전의 Microsoft C++ 컴파일러 버전을 사용하여 이식성 문제를 감지하려면, 32비트 플랫폼과 64비트 플랫폼 간에 크기를 변경하는 모든 `typedef`에서 `_w64`

키워드를 지정해야 합니다. 해당 형식의 경우 `_w64`는 `typedef`의 32비트 정의에서만 나타나야 합니다.

이전 버전과의 호환성을 위해 `_w64`은 `_w64`의 동의어입니다. 단, 컴파일러 옵션 [/Za\(언어 확장 사용 안 함\)](#)가 지정된 경우는 예외입니다.

컴파일에 `/Wp64`가 사용되지 않으면 `_w64` 키워드가 무시됩니다.

64비트로 이식하는 방법에 대한 자세한 내용은 다음 항목을 참조하세요.

- [MSVC 컴파일러 옵션](#)
- [32비트 코드를 64비트 코드로 이식](#)
- [64비트, x64 대상을 위한 Visual C++ 구성](#)

예시

```
C++

// __w64.cpp
// compile with: /W3 /Wp64
typedef int Int_32;
#ifndef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
    Int_32 i0 = 5;
    Int_Native i1 = 10;
    i0 = i1;    // C4244 64-bit int assigned to 32-bit int

    // char __w64 c; error, cannot use __w64 on char
}
```

참고 항목

[키워드](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

func

아티클 • 2023. 10. 12.

(C++11) 미리 정의된 식별자는 `_func_` 바깥쪽 함수의 정규화되지 않은 이름 및 장식되지 않은 이름을 포함하는 문자열로 암시적으로 정의됩니다. `_func_` 는 C++ 표준에 따라 위임되며 Microsoft 확장이 아닙니다.

구문

C++

`_func_`

Return Value

함수 이름을 포함하는 문자의 null로 끝나는 const char 배열을 반환합니다.

예시

C++

```
#include <string>
#include <iostream>

namespace Test
{
    struct Foo
    {
        static void DoSomething(int i, std::string s)
        {
            std::cout << __func__ << std::endl; // Output: DoSomething
        }
    };
}

int main()
{
    Test::Foo::DoSomething(42, "Hello");

    return 0;
}
```

요구 사항

C++11

컴파일러 COM 지원

아티클 • 2023. 10. 12.

Microsoft 전용

Microsoft C++ 컴파일러는 COM(구성 요소 개체 모델) 형식 라이브러리를 직접 읽고 내용을 컴파일에 포함할 수 있는 C++ 소스 코드로 변환할 수 있습니다. 언어 확장은 데스크톱 앱에 대한 클라이언트 쪽에서 COM 프로그래밍을 용이하게 하기 위해 사용할 수 있습니다.

컴파일러는 #import 전처리기 지시문을 사용하여 형식 라이브러리를 읽고 COM 인터페이스를 클래스로 설명하는 C++ 헤더 파일로 변환할 수 있습니다. #import 특성 집합은 결과 형식 라이브러리 헤더 파일에 대한 콘텐츠를 사용자가 제어하는 데 사용할 수 있습니다.

_declspec 특성 uuid를 사용하여 COM 개체에 GUID(Globally Unique Identifier)를 할당할 수 있습니다. 키워드(keyword) _uuidof 사용하여 COM 개체와 연결된 GUID를 추출할 수 있습니다. 또 다른 _declspec 특성인 속성을 사용하여 COM 개체의 get 데이터 멤버 및 set 메서드를 지정할 수 있습니다.

COM 지원 전역 함수 및 클래스 집합은 다음에서 throw _com_raise_error 된 오류 개체를 캡슐화하고, 스마트 포인터를 구현하고, 형식 및 BSTR 형식을 지원 VARIANT하도록 제공됩니다.

- 컴파일러 COM 전역 함수
- _bstr_t
- _com_error
- _com_ptr_t
- _variant_t

Microsoft 전용 종료

참고 항목

- 컴파일러 COM 지원 클래스
- 컴파일러 COM 전역 함수

컴파일러 COM 전역 함수

아티클 • 2023. 10. 12.

Microsoft 전용

다음 루틴을 사용할 수 있습니다.

함수	설명
_com_raise_error	오류에 대한 응답으로 <code>_com_error</code> throw합니다.
_set_com_error_handler	COM 오류 처리에 사용되는 기본 함수를 대체합니다.
ConvertBSTRToString	값을 <code>char *</code> 로 <code>BSTR</code> 변환합니다.
ConvertStringToBSTR	값을 <code>BSTR</code> 로 <code>char *</code> 변환합니다.

Microsoft 전용 종료

참고 항목

[컴파일러 COM 지원 클래스](#)

[컴파일러 COM 지원](#)

_com_raise_error

아티클 • 2023. 10. 12.

Microsoft 전용

오류에 대한 응답으로 _com_error throw합니다.

구문

C++

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

매개 변수

hr

HRESULT 정보입니다.

perrinfo

IErrorInfo 개체

설명

comdef.h>에 <정의된 _com_raise_error 동일한 이름과 프로토타입의 사용자가 작성한 버전으로 바꿀 수 있습니다. 이는 #import 를 사용하고 C++ 예외 처리는 사용하지 않으려는 경우 실행할 수 있습니다. 이 경우 사용자 버전의 _com_raise_error 메시지 상자를 수행 longjmp 하거나 표시하고 중지하도록 결정할 수 있습니다. 컴파일러 COM 지원 코드가 반환을 예상하고 있지 않기 때문에 사용자 버전은 반환할 수 없습니다.

_set_com_error_handler 사용하여 기본 오류 처리 함수를 바꿀 수도 있습니다.

기본적으로 _com_raise_error 다음과 같이 정의됩니다.

C++

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
    throw _com_error(hr, perrinfo);
}
```

요구 사항

헤더:<comdef.h>

Lib: wchar_t 네이티브 형식 컴파일러 옵션이 있는 경우 comsuppw.lib 또는 comsuppwd.lib를 사용합니다. wchar_t 네이티브 형식이 꺼져 있으면 comsupp.lib를 사용합니다. 자세한 내용은 [/Zc:wchar_t\(wchar_t는 네이티브 형식임\)](#)을 참조하세요.

참고 항목

컴파일러 COM 전역 함수

[_set_com_error_handler](#)

ConvertStringToBSTR

아티클 • 2024. 11. 21.

Microsoft 전용

값을 `BSTR`로 `char *` 변환합니다.

구문

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

매개 변수

pSrc

변수입니다 `char *`.

예시

C++

```
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
    char* lpszText = "Test";
    printf_s("char * text: %s\n", lpszText);

    BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
    wprintf_s(L"BSTR text: %s\n", bstrText);

    SysFreeString(bstrText);
}
```

Output

```
char * text: Test
```

BSTR text: Test

Microsoft 전용 종료

요구 사항

헤더:<comutil.h>

Lib: comsuppw.lib 또는 comsuppwd.lib(자세한 내용은 /Zc:wchar_t(wchar_t 네이티브 형식) 참조)

참고 항목

[컴파일러 COM 전역 함수](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

ConvertBSTRToString

아티클 • 2024. 11. 21.

Microsoft 전용

값을 `char *`로 `BSTR` 변환합니다.

구문

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

매개 변수

pSrc

`BSTR` 변수입니다.

설명

`ConvertBSTRToString` 은 삭제해야 하는 문자열을 할당합니다.

예시

C++

```
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
    BSTR bstrText = ::SysAllocString(L"Test");
    wprintf_s(L"BSTR text: %s\n", bstrText);

    char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
    printf_s("char * text: %s\n", lpszText2);

    SysFreeString(bstrText);
    delete[] lpszText2;
}
```

Output

```
BSTR text: Test  
char * text: Test
```

Microsoft 전용 종료

요구 사항

헤더:<comutil.h>

Lib: comsuppw.lib 또는 comsuppwd.lib(자세한 내용은 [/Zc:wchar_t\(wchar_t 네이티브 형식\) 참조](#))

참고 항목

[컴파일러 COM 전역 함수](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_set_com_error_handler

아티클 • 2023. 10. 12.

COM 오류 처리에 사용되는 기본 함수를 대체합니다. _set_com_error_handler Microsoft 전용입니다.

구문

C++

```
void __stdcall _set_com_error_handler(
    void (__stdcall *pHandler)(
        HRESULT hr,
        IErrorInfo* perrinfo
    )
);
```

매개 변수

pHandler

대체 함수에 대한 포인터입니다.

Hr

HRESULT 정보입니다.

perrinfo

IErrorInfo 개체

설명

기본적으로 [_com_raise_error](#) 모든 COM 오류를 처리합니다. _set_com_error_handler 사용하여 고유한 오류 처리 함수를 호출하여 이 동작을 변경할 수 있습니다.

대체 함수에는 [_com_raise_error](#)의 시그니처에 해당하는 시그니처가 있어야 합니다.

예시

C++

```
// _set_com_error_handler.cpp
// compile with /EHsc
```

```

#include <stdio.h>
#include <comdef.h>
#include <comutil.h>

// Importing ado dll to attempt to establish an ado connection.
// Not related to _set_com_error_handler
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace
rename("EOF", "adoEOF")

void __stdcall _My_com_raise_error(HRESULT hr, IErrorInfo* perrinfo)
{
    throw "Unable to establish the connection!";
}

int main()
{
    _set_com_error_handler(_My_com_raise_error);
    _bstr_t bstrEmpty(L"");
    _ConnectionPtr Connection = NULL;
    try
    {
        Connection.CreateInstance(__uuidof(Connection));
        Connection->Open(bstrEmpty, bstrEmpty, bstrEmpty, 0);
    }
    catch(char* errorMessage)
    {
        printf("Exception raised: %s\n", errorMessage);
    }

    return 0;
}

```

Output

Exception raised: Unable to establish the connection!

요구 사항

헤더:<comdef.h>

Lib: /Zc:wchar_t 컴파일러 옵션이 지정된 경우(기본값) comsuppw.lib 또는
comsuppwd.lib를 사용합니다. /Zc:wchar_t- 컴파일러 옵션이 지정된 경우 comsupp.lib를
사용합니다. IDE에서 이 옵션을 설정하는 방법을 비롯한 자세한 내용은
[/Zc:wchar_t\(wchar_t 네이티브 형식\)](#)을 참조하세요.

참고 항목

컴파일러 COM 전역 함수

컴파일러 COM 지원 클래스

아티클 • 2023. 10. 12.

Microsoft 전용

표준 클래스는 COM 형식 중 일부를 지원하는 데 사용됩니다. 클래스는 `comdef.h` 및 형식 라이브러리에서 생성된 헤더 파일에 정의됩니다.

클래스	목적
<code>_bstr_t</code>	<code>BSTR</code> 형식을 래핑하여 유용한 연산자와 메서드를 제공합니다.
<code>_com_error</code>	대부분의 오류에서 <code>_com_raise_error</code> <code>throw</code> 된 오류 개체를 정의합니다.
<code>_com_ptr_t</code>	COM 인터페이스 포인터를 캡슐화하고, <code>Release</code> 및 <code>QueryInterface</code> .에 필요한 호출을 <code>AddRef</code> 자동화합니다.
<code>_variant_t</code>	<code>VARIANT</code> 형식을 래핑하여 유용한 연산자와 메서드를 제공합니다.

Microsoft 전용 종료

참고 항목

[컴파일러 COM 지원](#)

[컴파일러 COM 전역 함수](#)

[C++ 언어 참조](#)

_bstr_t 클래스

아티클 • 2024. 07. 08.

Microsoft 전용

_bstr_t 개체는 BSTR 데이터 형식을 캡슐화합니다. 클래스는 해당하는 경우 [SysAllocString](#) 및 [SysFreeString](#)에 대한 함수 호출과 다른 BSTR API를 통해 리소스 할당 및 할당 해제를 관리합니다. _bstr_t 클래스는 과도한 오버헤드를 방지하기 위해 참조 가산을 사용합니다.

멤버

공사

[+] 테이블 확장

생성자	Description
_bstr_t	_bstr_t 개체를 생성합니다.

작업

[+] 테이블 확장

함수	설명
Assign	BSTR 을 BSTR로 래핑된 _bstr_t로 복사합니다.
Attach	_bstr_t 래퍼를 BSTR에 연결합니다.
copy	캡슐화된 BSTR의 복사본을 구성합니다.
Detach	BSTR로 래핑된 _bstr_t를 반환하고 BSTR을 _bstr_t에서 분리합니다.
GetAddress	BSTR로 래핑된 _bstr_t를 가리킵니다.
GetBSTR	BSTR에 의해 래핑되는 _bstr_t의 시작 부분을 가리킵니다.
length	_bstr_t에 있는 문자의 수를 반환합니다.

연산자

연산자	설명
operator =	기존 <code>_bstr_t</code> 개체에 새 값을 할당합니다.
operator +=	<code>_bstr_t</code> 개체의 끝 부분에 문자를 추가합니다.
operator +	두 문자열을 연결합니다.
operator !	캡슐화된 <code>BSTR</code> 이 NULL 문자열인지 확인합니다.
operator ==	두 <code>_bstr_t</code> 개체를 비교합니다.
operator !=	
operator <	
operator >	
operator <=	
operator >=	
operator wchar_t*	캡슐화된 유니코드 또는 멀티바이트 <code>BSTR</code> 개체에 대한 포인터를 추출합니다.
operator char*	

Microsoft 전용 종료

요구 사항

헤더:<comutil.h>

라이브러리: `comsuppw.Lib` 또는 `comsuppwd.Lib`(자세한 내용은 [/Zc:wchar_t\(wchar_t는 네 이티브 형식임\) 참조](#))

참고 항목

[컴파일러 COM 지원 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_bstr_t::_bstr_t

아티클 • 2023. 10. 12.

Microsoft 전용

_bstr_t 개체를 생성합니다.

구문

C++

```
_bstr_t( ) throw( );
_bstr_t(
    const _bstr_t& s1
) throw( );
_bstr_t(
    const char* s2
);
_bstr_t(
    const wchar_t* s3
);
_bstr_t(
    const _variant_t& var
);
_bstr_t(
    BSTR bstr,
    bool fCopy
);
```

매개 변수

s1

복사할 **_bstr_t** 개체입니다.

s2

멀티바이트 문자열입니다.

s3

유니코드 문자열입니다.

var

_variant_t 개체입니다.

`bstr`

기존 `BSTR` 개체입니다.

`fCopy`

이면 `false` 인수를 `bstr` 호출 `SysAllocString`하여 복사본을 만들지 않고 새 개체에 연결 됩니다.

설명

이 클래스는 `_bstr_t` 다음과 같은 여러 생성자를 제공합니다.

`_bstr_t()`

null `BSTR` 개체를 캡슐화하는 기본 `_bstr_t` 개체를 생성합니다.

`_bstr_t(_bstr_t& s1)`

`_bstr_t` 개체를 다른 개체의 복사본으로 생성합니다. 이 생성자는 새 개체를 만드는 대신 캡슐화된 `BSTR` 개체의 참조 수를 증가시키는 단순 복사본을 만듭니다.

`_bstr_t(char* s2)`

새 `_bstr_t` 개체를 만드는 `SysAllocString`을 호출하여 `BSTR` 개체를 생성한 다음 캡슐화 합니다. 이 생성자는 먼저 멀티바이트를 유니코드로 변환합니다.

`_bstr_t(wchar_t* s3)`

새 `_bstr_t` 개체를 만드는 `SysAllocString`을 호출하여 `BSTR` 개체를 생성한 다음 캡슐화 합니다.

`_bstr_t(_variant_t& var)`

`_bstr_t` 먼저 캡슐화된 `VARIANT` 개체에서 `_variant_t` 개체를 검색하여 `BSTR` 개체에서 개체를 생성합니다.

`_bstr_t(BSTR bstr, bool fCopy)`

기존 `_bstr_t(BSTR` 문자열의 반대)에서 `wchar_t*` 개체를 생성합니다. 이 `false` 경우 `fCopy` 제공된 `BSTR` 개체는 을 사용하여 `SysAllocString` 새 복사본을 만들지 않고 새 개체에 연결됩니다. 이 생성자는 형식 라이브러리 헤더의 래퍼 함수에서 인터페이스 메서드에서 반환된 형식의 `BSTR` 소유권을 캡슐화하고 가져오는 데 사용됩니다.

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

[_variant_t 클래스](#)

_bstr_t::Assign

아티클 • 2023. 04. 03.

Microsoft 전용

BSTR 을 BSTR 로 래핑된 _bstr_t 로 복사합니다.

구문

C++

```
void Assign(  
    BSTR s  
) ;
```

매개 변수

s

BSTR 로 래핑된 BSTR 로 복사되는 _bstr_t 입니다.

설명

Assign 는 콘텐츠에 관계없이 전체 길이의 BSTR 이진 복사본을 수행합니다.

예

C++

```
// _bstr_t_Assign.cpp  
  
#include <comdef.h>  
#include <stdio.h>  
  
int main()  
{  
    // creates a _bstr_t wrapper  
    _bstr_t bstrWrapper;  
  
    // creates BSTR and attaches to it  
    bstrWrapper = "some text";  
    wprintf_s(L"bstrWrapper = %s\n",  
            static_cast<wchar_t*>(bstrWrapper));
```

```

// bstrWrapper releases its BSTR
BSTR bstr = bstrWrapper.Detach();
wprintf_s(L"bstrWrapper = %s\n",
          static_cast<wchar_t*>(bstrWrapper));
// "some text"
wprintf_s(L"bstr = %s\n", bstr);

bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocatedString")));
wprintf_s(L"bstrWrapper = %s\n",
          static_cast<wchar_t*>(bstrWrapper));

// assign a BSTR to our _bstr_t
bstrWrapper.Assign(bstr);
wprintf_s(L"bstrWrapper = %s\n",
          static_cast<wchar_t*>(bstrWrapper));

// done with BSTR, do manual cleanup
SysFreeString(bstr);

// reuse bstr
bstr= SysAllocString(OLESTR("Yet another string"));
// two wrappers, one BSTR
_bstr_t bstrWrapper2 = bstrWrapper;

*bstrWrapper.GetAddress() = bstr;

// bstrWrapper and bstrWrapper2 do still point to BSTR
bstr = 0;
wprintf_s(L"bstrWrapper = %s\n",
          static_cast<wchar_t*>(bstrWrapper));
wprintf_s(L"bstrWrapper2 = %s\n",
          static_cast<wchar_t*>(bstrWrapper2));

// new value into BSTR
_snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
             L"changing BSTR");
wprintf_s(L"bstrWrapper = %s\n",
          static_cast<wchar_t*>(bstrWrapper));
wprintf_s(L"bstrWrapper2 = %s\n",
          static_cast<wchar_t*>(bstrWrapper2));
}

```

Output

```

bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocatedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text

```

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::Attach`

아티클 • 2023. 10. 12.

Microsoft 전용

`_bstr_t` 래퍼를 `BSTR`에 연결합니다.

구문

C++

```
void Attach(  
    BSTR s  
)
```

매개 변수

`s`

`BSTR` 변수와 연결되거나 이 변수에 할당될 `_bstr_t`입니다.

설명

이전에 `_bstr_t`가 다른 `BSTR`에 연결된 경우 다른 `_bstr_t` 변수가 `BSTR`을 사용하고 있지 않으면 `_bstr_t`가 `BSTR` 리소스를 정리합니다.

예시

를 사용하는 예제를 참조하세요 [_bstr_t::Assign Attach](#).

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::copy`

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 `BSTR`의 복사본을 구성합니다.

구문

C++

```
BSTR copy( bool fCopy = true ) const;
```

매개 변수

`fCopy`

이 `copy` 포함된 `BSTR copy` 복사본을 반환하면 `true` 실제 `BSTR` 복사본이 반환됩니다.

설명

매개 변수에 따라 캡슐화된 개체 또는 캡슐화된 `BSTR` 개체 자체의 새로 할당된 복사본을 반환합니다.

예시

C++

```
STDMETHODIMP CAalertMsg::get_ConnectionStr(BSTR *pVal){ // m_bsConStr is  
_bstr_t  
*pVal = m_bsConStr.copy();  
}
```

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::Detach`

아티클 • 2023. 10. 12.

Microsoft 전용

`BSTR`로 래핑된 `_bstr_t`을 반환하고 `BSTR`을 `_bstr_t`에서 분리합니다.

구문

C++

```
BSTR Detach( ) throw;
```

Return Value

에 `BSTR` 의해 캡슐화된 값을 반환합니다 `_bstr_t`.

예시

를 사용하는 `Detach` 예제를 참조하세요 [_bstr_t::Assign](#).

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::GetAddress`

아티클 • 2023. 10. 12.

Microsoft 전용

기존 문자열을 해제하고 새로 할당된 문자열의 주소를 반환합니다.

구문

C++

```
BSTR* GetAddress( );
```

Return Value

`BSTR`로 래핑되는 `_bstr_t`에 대한 포인터입니다.

설명

`GetAddress`를 공유하는 모든 `_bstr_t` 개체에 영향을 줍니다 `BSTR`. 둘 이상의 `_bstr_t BSTR` 복사 생성자 및 `operator=`.

예시

를 사용하는 `GetAddress` 예제를 참조하세요 [_bstr_t::Assign](#).

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::GetBSTR`

아티클 • 2023. 10. 12.

Microsoft 전용

`BSTR`에 의해 래핑되는 `_bstr_t`의 시작 부분을 가리킵니다.

구문

C++

```
BSTR& GetBSTR( );
```

Return Value

`BSTR`에 의해 래핑되는 `_bstr_t`의 시작 부분입니다.

설명

`GetBSTR`를 공유하는 모든 `_bstr_t` 개체에 영향을 줍니다 `BSTR`. 둘 이상의 `_bstr_t` `BSTR` 복사 생성자 및 `operator=`.

예시

를 사용하는 `GetBSTR` 예제를 참조하세요 [_bstr_t::Assign](#).

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

_bstr_t::length

아티클 • 2023. 04. 03.

Microsoft 전용

종결 null이 포함되지 않는 캡슐화된 `_bstr_t`의 `BSTR`에서 문자 수가 반환됩니다.

구문

C++

```
unsigned int length( ) const throw( );
```

설명

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

_bstr_t::operator =

아티클 • 2023. 06. 16.

Microsoft 전용

기존 `_bstr_t` 개체에 새 값을 할당합니다.

구문

C++

```
_bstr_t& operator=(const _bstr_t& s1) throw ( );
_bstr_t& operator=(const char* s2);
_bstr_t& operator=(const wchar_t* s3);
_bstr_t& operator=(const _variant_t& var);
```

매개 변수

`s1`

기존 `_bstr_t` 개체에 할당될 `_bstr_t` 개체입니다.

`s2`

기존 `_bstr_t` 개체에 할당될 멀티바이트 문자열입니다.

`s3`

기존 `_bstr_t` 개체에 할당될 유니코드 문자열입니다.

`var`

기존 `_variant_t` 개체에 할당될 `_bstr_t` 개체입니다.

Microsoft 전용 종료

예제

를 사용하는 `operator=` 예제는 를 참조하세요 [_bstr_t::Assign](#).

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::operator +=, _bstr_t::operator`

+

아티클 • 2023. 10. 12.

Microsoft 전용

개체의 `_bstr_t` 끝에 문자를 추가하거나 두 문자열을 연결합니다.

구문

C++

```
_bstr_t& operator+=( const _bstr_t& s1 );
_bstr_t operator+( const _bstr_t& s1 );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1);
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1);
```

매개 변수

`s1`

`_bstr_t` 개체입니다.

`s2`

멀티바이트 문자열입니다.

`s3`

유니코드 문자열입니다.

설명

이러한 연산자는 다음과 같이 문자열 연결을 수행합니다.

- `operator+=(s1)` 이 개체의 캡슐화된 끝에 캡슐화된 `BSTR s1` 문자를 추가합니다 `BSTR`.
- `operator+(s1)` 이 개체와 개체를 연결하여 형성된 새 `_bstr_t` 개체 `BSTR` 를 `s1` 반환합니다.
- `operator+(s2, s1)` 멀티바이트 문자열을 연결하고 유니코드로 변환한 `BSTR` 다음에 캡슐화된 `s1` 새 `_bstr_t` 문자열 `s2` 을 반환합니다.

- `operator+(s3, s1)` 유니코드 문자열과 캡슐화된 문자열을 연결하여 형성된 `s1` 새 `_bstr_t` 형식 `s3` 을 `BSTR` 반환합니다.

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

`_bstr_t::operator !`

아티클 • 2024. 11. 21.

Microsoft 전용

캡슐화된 `BSTR`이 NULL 문자열인지 확인합니다.

구문

C++

```
bool operator!( ) const throw( );
```

Return Value

캡슐화된 `BSTR` 문자열이 NULL 문자열 `false` 이 아니면 반환 `true` 됩니다.

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_bstr_t` 관계형 연산자

아티클 • 2023. 10. 12.

Microsoft 전용

두 `_bstr_t` 개체를 비교합니다.

구문

C++

```
bool operator==(const _bstr_t& str) const throw();
bool operator!=(const _bstr_t& str) const throw();
bool operator<(const _bstr_t& str) const throw();
bool operator>(const _bstr_t& str) const throw();
bool operator<=(const _bstr_t& str) const throw();
bool operator>=(const _bstr_t& str) const throw();
```

설명

이러한 연산자는 두 `_bstr_t` 개체를 사전순으로 비교합니다. 비교가 유지되면 연산자가 반환 `true` 되며, 그렇지 않으면 반환됩니다 `false`.

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

_bstr_t::wchar_t*, _bstr_t::char*

아티클 • 2023. 10. 12.

Microsoft 전용

BSTR 문자를 좁거나 넓은 문자 배열로 반환합니다.

구문

C++

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

설명

이러한 연산자를 사용하여 개체에 의해 캡슐화된 문자 데이터를 추출할 BSTR 수 있습니다. 반환된 포인터에 새 값을 할당해도 원래 BSTR 데이터는 수정되지 않습니다.

Microsoft 전용 종료

참고 항목

[_bstr_t 클래스](#)

_com_error 클래스

아티클 • 2024. 07. 12.

Microsoft 전용

_com_error 개체는 형식 라이브러리 또는 COM 지원 클래스 중 하나에서 생성된 헤더 파일의 오류 처리 래퍼 함수에서 검색된 예외 조건을 나타냅니다. _com_error 클래스는 HRESULT 오류 코드 및 연결된 모든 IErrorInfo Interface 개체를 캡슐화합니다.

공사

[+] 테이블 확장

속성	설명
_com_error	_com_error 개체를 생성합니다.

연산자

[+] 테이블 확장

속성	설명
operator =	기존 _com_error 개체를 다른 개체에 할당합니다.

추출기 함수

[+] 테이블 확장

속성	설명
Error	생성자에 전달된 HRESULT를 검색합니다.
ErrorInfo	생성자에 전달된 IErrorInfo 개체를 검색합니다.
WCode	캡슐화된 HRESULT에 매핑되는 16비트 오류 코드를 검색합니다.

IErrorInfo 함수

[+] 테이블 확장

속성	설명
Description	<code>IErrorInfo::GetDescription</code> 함수를 호출합니다.
HelpContext	<code>IErrorInfo::GetHelpContext</code> 함수를 호출합니다.
HelpFile	<code>IErrorInfo::GetHelpFile</code> 함수 호출
Source	<code>IErrorInfo::GetSource</code> 함수를 호출합니다.
GUID	<code>IErrorInfo::GetGUID</code> 함수를 호출합니다.

메시지 추출기 서식 지정

[+] 테이블 확장

속성	설명
ErrorMessage	<code>_com_error</code> 개체에 저장된 <code>HRESULT</code> 에 대한 문자열 메시지를 검색합니다.

ExepInfo.wCode - HRESULT 맵 편집기

[+] 테이블 확장

속성	설명
HRESULTToWCode	32비트 <code>HRESULT</code> 를 16비트 <code>wCode</code> 로 매핑합니다.
WCodeToHRESULT	16비트 <code>wCode</code> 를 32비트 <code>HRESULT</code> 로 매핑합니다.

Microsoft 전용 종료

요구 사항

헤더:<comdef.h>

라이브러리: `comsuppw.lib` 또는 `comsuppwd.lib`(자세한 내용은 [/Zc:wchar_t\(wchar_t는 네 이티브 형식임\) 참조](#))

참고 항목

[컴파일러 COM 지원 클래스](#)

[IErrorInfo 인터페이스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_error` 멤버 함수

아티클 • 2023. 04. 03.

멤버 함수에 `_com_error` 대한 자세한 내용은 클래스를 참조 [_com_error 하세요.](#)

참고 항목

[_com_error 클래스](#)

`_com_error::_com_error`

아티클 • 2024. 08. 02.

Microsoft 전용

`_com_error` 개체를 생성합니다.

구문

C++

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef = false) throw();

_com_error( const _com_error& that ) throw();
```

매개 변수

`hr`

`HRESULT` 정보.

`perrinfo`

`IErrorInfo` 개체

`fAddRef`

기본값은 생성자가 null `IErrorInfo` 이 아닌 인터페이스에서 AddRef를 호출하지 않도록 합니다. 이 동작은 다음과 같이 인터페이스의 소유권이 개체에 전달되는 일반적인 경우 올바른 참조 계산을 `_com_error` 제공합니다.

C++

```
throw _com_error(hr, perrinfo);
```

코드에서 소유권을 개체 `AddRef` 로 `_com_error` 이전하지 않고 소멸자의 오프셋 `Release` `_com_error`에 필요한 경우 다음과 같이 개체를 생성합니다.

C++

```
_com_error err(hr, perrinfo, true);
```

that

기존 `_com_error` 개체입니다.

설명

첫 번째 생성자는 선택적 `IErrorInfo` 개체가 지정된 새 개체를 `HRESULT` 만듭니다. 두 번째는 기존 개체의 복사본을 `_com_error` 만듭니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_error::Description`

아티클 • 2024. 11. 21.

Microsoft 전용

`IErrorInfo::GetDescription` 함수를 호출합니다.

구문

C++

```
_bstr_t Description() const;
```

반환 값

개체 내에 `_com_error` 기록된 개체의 `IErrorInfo IErrorInfo::GetDescription` 결과를 반환합니다. 결과 `BSTR`은 `_bstr_t` 개체에 캡슐화됩니다. 레코드가 없 `IErrorInfo` 으면 빈 `_bstr_t`.

설명

함수를 `IErrorInfo::GetDescription` 호출하고 개체 내에 기록된 함수를 `_com_error` 검색합니다 `IErrorInfo`. 메서드를 호출하는 `IErrorInfo::GetDescription` 동안 오류가 발생하면 무시됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

_com_error::Error

아티클 • 2024. 11. 21.

Microsoft 전용

생성자에 전달된 `HRESULT`를 검색합니다.

구문

C++

```
HRESULT Error() const throw();
```

반환 값

생성자에 전달된 원시 `HRESULT` 항목입니다.

설명

개체의 캡슐화된 `HRESULT` 항목을 검색 `_com_error` 합니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_error::ErrorInfo

아티클 • 2024. 11. 21.

Microsoft 전용

생성자에 전달된 `IErrorInfo` 개체를 검색합니다.

구문

C++

```
IErrorInfo * ErrorInfo( ) const throw( );
```

반환 값

생성자에 전달된 원시 `IErrorInfo` 항목입니다.

설명

개체에서 캡슐화된 `IErrorInfo` 항목을 검색하거나 `NULL` 기록된 항목이 없는 `IErrorInfo _com_error` 경우 검색합니다. 호출자는 사용이 완료되면 반환된 개체를 호출 `Release` 해야 합니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_error::ErrorMessage`

아티클 • 2024. 11. 21.

Microsoft 전용

`_com_error` 개체에 저장된 `HRESULT`에 대한 문자열 메시지를 검색합니다.

구문

C++

```
const TCHAR * ErrorMessage() const throw();
```

반환 값

개체 내에 `_com_error` 기록된 문자열 메시지를 `HRESULT` 반환합니다. 매핑된 `HRESULT` 16 비 `wCode`트인 경우 제네릭 메시지 "`IDispatch error #<wCode>`"가 반환됩니다. 메시지가 발견되지 않으면 일반 메시지 "`Unknown error #<HRESULT>`"가 반환됩니다. 반환된 문자열은 매크로의 상태에 `_UNICODE` 따라 유니코드 또는 멀티바이트 문자열입니다.

설명

개체 내에 `_com_error` 기록할 적절한 시스템 메시지 텍스트를 `HRESULT` 검색합니다. Win32 `FormatMessage` 함수를 호출하여 시스템 메시지 텍스트를 가져옵니다. 반환된 문자열은 API에 `FormatMessage` 의해 할당되며 개체가 제거되면 해제 `_com_error` 됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

_com_error::GUID

아티클 • 2024. 11. 21.

Microsoft 전용

IErrorInfo::GetGUID 함수를 호출합니다.

구문

C++

```
GUID GUID() const throw();
```

반환 값

개체 내에 _com_error 기록된 개체의 IErrorInfo IErrorInfo::GetGUID 결과를 반환합니다. 기록된 개체가 없 IErrorInfo 으면 반환됩니다 GUID_NULL.

설명

메서드를 호출하는 IErrorInfo::GetGUID 동안 오류가 발생하면 무시됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_error::HelpContext`

아티클 • 2024. 11. 21.

Microsoft 전용

`IErrorInfo::GetHelpContext` 함수를 호출합니다.

구문

C++

```
DWORD HelpContext() const throw();
```

반환 값

개체 내에 `_com_error` 기록된 개체의 `IErrorInfo IErrorInfo::GetHelpContext` 결과를 반환합니다. 기록된 개체가 없 `IErrorInfo` 으면 0을 반환합니다.

설명

메서드를 호출하는 `IErrorInfo::GetHelpContext` 등안 오류가 발생하면 무시됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_error::HelpFile

아티클 • 2024. 11. 21.

Microsoft 전용

`IErrorInfo::GetHelpFile` 함수를 호출합니다.

구문

C++

```
_bstr_t HelpFile() const;
```

반환 값

개체 내에 `_com_error` 기록된 개체의 `IErrorInfo IErrorInfo::GetHelpFile` 결과를 반환합니다. 결과 BSTR은 `_bstr_t` 개체에 캡슐화됩니다. 레코드가 없 `IErrorInfo` 으면 빈 `_bstr_t`.

설명

메서드를 호출하는 `IErrorInfo::GetHelpFile` 동안 오류가 발생하면 무시됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_error::HRESULTToWCode`

아티클 • 2024. 11. 21.

Microsoft 전용

32비트 `HRESULT` 를 16비트 `wCode` 로 매핑합니다.

구문

C++

```
static WORD HRESULTToWCode(
    HRESULT hr
) throw();
```

매개 변수

`hr`

16비트 `HRESULT` 트로 매핑할 32비트 `wCode` 트입니다.

반환 값

32비트 `wCode` 트에서 매핑된 16비트 `HRESULT` 트.

설명

자세한 내용은 [_com_error::WCode](#)를 참조하세요.

Microsoft 전용 종료

참고 항목

[_com_error::WCode](#)
[_com_error::WCodeToHRESULT](#)
[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_error::Source

아티클 • 2024. 11. 21.

Microsoft 전용

`IErrorInfo::GetSource` 함수를 호출합니다.

구문

C++

```
_bstr_t Source() const;
```

반환 값

개체 내에 `_com_error` 기록된 개체의 `IErrorInfo IErrorInfo::GetSource` 결과를 반환합니다. 결과 `BSTR`은 `_bstr_t` 개체에 캡슐화됩니다. 레코드가 없 `IErrorInfo` 으면 빈 `_bstr_t`.

설명

메서드를 호출하는 `IErrorInfo::GetSource` 동안 오류가 발생하면 무시됩니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_error::WCode

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 `HRESULT` 16비트 오류 코드를 검색합니다.

구문

C++

```
WORD WCode ( ) const throw();
```

반환 값

`HRESULT` 0x8004FFFF 0x80040200 범위 내에 있으면 메서드는 `WCode` 빼기 0x80040200 반환 `HRESULT` 하고, 그렇지 않으면 0을 반환합니다.

설명

이 `WCode` 메서드는 COM 지원 코드에서 발생하는 매팅을 실행 취소하는 데 사용됩니다. 속성 또는 메서드의 `dispinterface` 래퍼는 인수를 패키지하고 호출하는 지원 루틴을 호출 `IDispatch::Invoke` 합니다. 반환 시 오류가 `HRESULT DISP_E_EXCEPTION` 반환되면 전달된 구조에서 `EXCEPINFO` 오류 정보가 검색됩니다 `IDispatch::Invoke`. 오류 코드는 구조체의 멤버에 `wCode` 저장된 16비트 값이거나 구조체의 `EXCEPINFO` 멤버 `EXCEPINFO`에 있는 `scode` 전체 32비트 값일 수 있습니다. 16비 `wCode` 트가 반환되면 먼저 32비트 오류 `HRESULT`에 매팅되어야 합니다.

Microsoft 전용 종료

[_com_error::HRESULTToWCode](#)
[_com_error::WCodeToHRESULT](#)
[_com_error 클래스](#)

참고 항목

_com_error::WCodeToHRESULT

아티클 • 2024. 11. 21.

Microsoft 전용

16비트 `wCode` 를 32비트 `HRESULT` 로 매핑합니다.

구문

C++

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw();
```

매개 변수

`wCode`

32비 `wCode` 트로 매핑할 16비 `HRESULT` 트입니다.

반환 값

16비 `HRESULT` 트에서 매핑된 32비 `wCode` 트 .

설명

멤버 함수를 [WCode](#) 참조하세요.

Microsoft 전용 종료

참고 항목

[_com_error::WCode](#)

[_com_error::HRESULTToWCode](#)

[_com_error 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_error 연산자

아티클 • 2023. 10. 12.

연산자에 대한 자세한 내용은 클래스를 [_com_error](#) 참조 [_com_error](#) 하세요.

참고 항목

[_com_error 클래스](#)

`_com_error::operator=`

아티클 • 2023. 04. 03.

Microsoft 전용

기존 `_com_error` 개체를 다른 개체에 할당합니다.

구문

C++

```
_com_error& operator=(  
    const _com_error& that  
) throw ();
```

매개 변수

that

`_com_error` 개체입니다.

Microsoft 전용 종료

참고 항목

[_com_error 클래스](#)

_com_ptr_t 클래스

아티클 • 2024. 11. 21.

Microsoft 전용

_com_ptr_t 개체는 COM 인터페이스 포인터를 캡슐화하고 "스마트" 포인터라고 합니다. 이 템플릿 클래스는 멤버 함수에 대한 함수 호출을 통해 리소스 할당 및 할당 취소를 IUnknown 관리합니다. AddRef QueryInterface Release

스마트 포인터는 일반적으로 _COM_SMARTPTR_TYPEDEF 매크로에서 제공하는 typedef 정의에서 참조됩니다. 이 매크로는 인터페이스 이름과 IID를 사용하고 인터페이스 이름과 접미사가 Ptr 있는 _com_ptr_t 특수화를 선언합니다. 예시:

C++

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

는 _com_ptr_t 특수화를 선언합니다. IMyInterfacePtr

이 템플릿 클래스의 멤버가 아닌 함수 템플릿 집합은 비교 연산자의 오른쪽에 있는 스마트 포인터와의 비교를 지원합니다.

공사

[+] 테이블 확장

속성	설명
_com_ptr_t	_com_ptr_t 개체를 생성합니다.

하위 수준 연산

[+] 테이블 확장

속성	설명
AddRef	AddRef 캡슐화된 인터페이스 포인터의 IUnknown 멤버 함수를 호출합니다.
Attach	이 스마트 포인터 형식의 원시 인터페이스 포인터를 캡슐화합니다.
CreateInstance	CLSID 또는 ProgID 가 지정된 개체의 새 인스턴스를 만듭니다.
Detach	캡슐화된 인터페이스 포인터를 추출하고 반환합니다.

속성	설명
GetActiveObject	<code>CLSID</code> 또는 <code>ProgID</code> 가 지정된 개체의 기존 인스턴스에 연결됩니다.
GetInterfacePtr	캡슐화된 인터페이스 포인터가 반환됩니다.
QueryInterface	<code>QueryInterface</code> 캡슐화된 인터페이스 포인터의 <code>IUnknown</code> 멤버 함수를 호출합니다.
엔지니어링	<code>Release</code> 캡슐화된 인터페이스 포인터의 <code>IUnknown</code> 멤버 함수를 호출합니다.

연산자

[+] 테이블 확장

속성	설명
연산자 =	기존 <code>_com_ptr_t</code> 개체에 새 값을 할당합니다.
<code>operator ==, !=, <, >, <=, >=</code>	스마트 포인터 개체를 다른 스마트 포인터, 원시 인터페이스 포인터 또는 NULL과 비교합니다.
추출기	캡슐화된 COM 인터페이스 포인터를 추출합니다.

Microsoft 전용 종료

요구 사항

헤더:<comip.h>

Lib: comsuppw.lib 또는 comsuppwd.lib(자세한 내용은 /Zc:wchar_t(wchar_t 네이티브 형식) 참조)

참고 항목

[컴파일러 COM 지원 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

_com_ptr_t Member 함수

아티클 • 2023. 10. 12.

_com_ptr_t 멤버 함수에 대한 자세한 내용은 _com_ptr_t 클래스를 참조[하세요](#).

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t::_com_ptr_t

아티클 • 2024. 11. 21.

Microsoft 전용

_com_ptr_t 개체를 생성합니다.

구문

C++

```
// Default constructor.  
// Constructs a NULL smart pointer.  
_com_ptr_t() throw();  
  
// Constructs a NULL smart pointer. The NULL argument must be zero.  
_com_ptr_t(  
    int null  
)  
  
// Constructs a smart pointer as a copy of another instance of the  
// same smart pointer. AddRef is called to increment the reference  
// count for the encapsulated interface pointer.  
_com_ptr_t(  
    const _com_ptr_t& cp  
) throw();  
  
// Move constructor (Visual Studio 2015 Update 3 and later)  
_com_ptr_t(_com_ptr_t&& cp) throw();  
  
// Constructs a smart pointer from a raw interface pointer of this  
// smart pointer's type. If fAddRef is true, AddRef is called  
// to increment the reference count for the encapsulated  
// interface pointer. If fAddRef is false, this constructor  
// takes ownership of the raw interface pointer without calling AddRef.  
_com_ptr_t(  
    Interface* pInterface,  
    bool fAddRef  
) throw();  
  
// Construct pointer for a _variant_t object.  
// Constructs a smart pointer from a _variant_t object. The  
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or  
// it can be converted into one of these two types. If QueryInterface  
// fails with an E_NOINTERFACE error, a NULL smart pointer is  
// constructed.  
_com_ptr_t(  
    const _variant_t& varSrc  
)
```

```
// Constructs a smart pointer given the CLSID of a coclass. This
// function calls CoCreateInstance, by the member function
// CreateInstance, to create a new COM object and then queries for
// this smart pointer's interface type. If QueryInterface fails with
// an E_NOINTERFACE error, a NULL smart pointer is constructed.
explicit _com_ptr_t(
    const CLSID& clsid,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Calls CoCreateClass with provided CLSID retrieved from string.
explicit _com_ptr_t(
    LPCWSTR str,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
    LPCSTR str,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
    Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
```

```
const _com_ptr_t<_OtherIID>& p  
);  
  
// Constructs a smart-pointer from any IUnknown-based interface pointer.  
template<typename _InterfaceType>  
_com_ptr_t(  
    _InterfaceType* p  
);  
  
// Disable conversion using _com_ptr_t* specialization of  
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)  
template<>  
explicit _com_ptr_t(  
    _com_ptr_t* p  
);
```

매개 변수

pInterface

원시 인터페이스 포인터입니다.

fAddRef

`AddRef` 이면 `true` 캡슐화된 인터페이스 포인터의 참조 수를 증분하기 위해 호출됩니다.

cp

`_com_ptr_t` 개체입니다.

p

원시 인터페이스 포인터로, 해당 형식이 이 `_com_ptr_t` 개체의 스마트 포인터 형식과 다른 것입니다.

varSrc

`_variant_t` 개체입니다.

clsid

`CLSID` coclass의

dwClsContext

실행 코드를 실행하는 컨텍스트입니다.

lpcStr

("{"로 시작) 또는 .을 포함하는 `CLSID` 멀티바이트 문자열입니다 `ProgID`.

pOuter

집합체에서 알 수 없는 외부 형식입니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

`_com_ptr_t::AddRef`

아티클 • 2023. 10. 12.

Microsoft 전용

`AddRef` 캡슐화된 인터페이스 포인터의 `IUnknown` 멤버 함수를 호출합니다.

구문

C++

```
void AddRef( );
```

설명

캡슐화된 인터페이스 포인터를 호출 `IUnknown::AddRef` 하여 포인터가 `E_POINTER` NULL이면 오류가 발생합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t::Attach

아티클 • 2024. 07. 15.

Microsoft 전용

이 스마트 포인터 형식의 원시 인터페이스 포인터를 캡슐화합니다.

구문

C++

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

매개 변수

pInterface

원시 인터페이스 포인터입니다.

fAddRef

`true` 이면 `AddRef` 가 호출됩니다. `false` 인 경우 `_com_ptr_t` 개체는 `AddRef` 를 호출하지 않고 원시 인터페이스 포인터의 소유권을 가져옵니다.

설명

- `Attach(pInterface)` `AddRef` 가 호출되지 않았습니다. 인터페이스 소유권이 이 `_com_ptr_t` 개체에 전달됩니다. 이전에 캡슐화된 포인터의 참조 카운트를 줄이기 위해 `Release` 가 호출됩니다.
- `Attach(pInterface , fAddRef)` `fAddRef` 가 `true` 인 경우, `AddRef` 는 캡슐화된 인터페이스 포인터의 참조 카운트를 증가시키기 위해 호출됩니다. `fAddRef` 가 `false` 인 경우 이 `_com_ptr_t` 개체는 `AddRef` 를 호출하지 않고 원시 인터페이스 포인터의 소유권을 갖습니다. 이전에 캡슐화된 포인터의 참조 카운트를 줄이기 위해 `Release` 가 호출됩니다.

Microsoft 전용 종료

참고 항목

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_ptr_t::CreateInstance

아티클 • 2024. 07. 08.

Microsoft 전용

`CLSID` 또는 `ProgID`가 지정된 개체의 새 인스턴스를 만듭니다.

구문

```
HRESULT CreateInstance(
    const CLSID& rclsid,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCWSTR clsidString,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCSTR clsidStringA,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

매개 변수

rclsid

개체의 `CLSID`입니다.

clsidString

`CLSID`("{"로 시작) 또는 `ProgID`를 보유하는 유니코드 문자열입니다.

clsidStringA

ANSI 코드 페이지를 사용하고 `CLSID`("{"로 시작) 또는 `ProgID`를 포함하는 멀티바이트 문자열입니다.

dwClsContext

실행 코드를 실행하는 컨텍스트입니다.

pOuter

집합체에서 알 수 없는 외부 형식입니다.

설명

이러한 멤버 함수는 새로운 COM 개체를 만들고 이 스마트 포인터의 인터페이스 형식을 쿼리하기 위해 `CoCreateInstance`를 호출합니다. 그러면 결과 포인터는 이 `_com_ptr_t` 개체 안에 캡슐화됩니다. 이전에 캡슐화된 포인터의 참조 카운트를 줄이기 위해 `Release`가 호출됩니다. 이 루틴은 `HRESULT`를 반환하여 성공 또는 실패를 나타냅니다.

- `CreateInstance(rclsid , dwClsContext)` `CLSID` 가 지정된 개체의 실행 중인 새 인스턴스를 만듭니다.
- `CreateInstance(clsidString , dwClsContext)` `CLSID` ("{"로 시작) 또는 `ProgID`를 포함하는 유니코드 문자열이 제공된 개체의 새 실행 인스턴스를 만듭니다.
- `CreateInstance(clsidStringA , dwClsContext)` `CLSID` ("{"로 시작) 또는 `ProgID`를 포함하는 멀티바이트 문자열이 제공된 개체의 새 실행 인스턴스를 만듭니다. 문자열이 OEM 코드 페이지가 아닌 ANSI 코드 페이지에 있다고 가정하는 `MultiByteToWideChar`를 호출합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_ptr_t::Detach

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 인터페이스 포인터를 추출하고 반환합니다.

구문

```
Interface* Detach( ) throw( );
```

설명

캡슐화된 인터페이스 포인터를 추출하고 반환한 다음, 캡슐화된 포인터 스토리지를 NULL로 지웁니다. 이 작업을 통해 인터페이스 포인터의 캡슐화를 제거합니다. 반환된 인터페이스 포인터를 호출 `Release` 하는 것은 사용자에게 달려 있습니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t::GetActiveObject

아티클 • 2024. 07. 08.

Microsoft 전용

`CLSID` 또는 `ProgID`가 지정된 개체의 기존 인스턴스에 연결됩니다.

구문

```
HRESULT GetActiveObject(
    const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
    LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
    LPCSTR clsidStringA
) throw( );
```

매개 변수

rclsid

개체의 `CLSID`입니다.

clsidString

`CLSID`("{"로 시작) 또는 `ProgID`를 보유하는 유니코드 문자열입니다.

clsidStringA

ANSI 코드 페이지를 사용하고 `CLSID`("{"로 시작) 또는 `ProgID`를 포함하는 멀티바이트 문자열입니다.

설명

이러한 멤버 함수는 `GetActiveObject`를 호출하여 OLE에 등록된 실행 개체에 대한 포인터를 검색한 다음 이 스마트 포인터의 인터페이스 형식을 쿼리합니다. 그러면 결과 포인터는 이 `_com_ptr_t` 개체 안에 캡슐화됩니다. 이전에 캡슐화된 포인터의 참조 카운트를 줄이기 위해 `Release`가 호출됩니다. 이 루틴은 `HRESULT`를 반환하여 성공 또는 실패를 나타냅니다.

- **GetActiveObject(`rclsid`)** `CLSID`가 지정된 개체의 기존 인스턴스에 연결됩니다.
- **GetActiveObject(`clsidString`)** `CLSID`("{"로 시작) 또는 `ProgID` 중 하나를 포함하는 유니코드 문자열이 지정된 개체의 기존 인스턴스에 연결합니다.
- **GetActiveObject(`clsidStringA`)** `CLSID`("{"로 시작) 또는 `ProgID` 중 하나를 포함하는 멀티바이트 문자열이 지정된 개체의 기존 인스턴스에 연결합니다. 문자열이 OEM 코드 페이지가 아닌 ANSI 코드 페이지에 있다고 가정하는 [MultiByteToWideChar](#)를 호출합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_com_ptr_t::GetInterfacePtr

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 인터페이스 포인터가 반환됩니다.

구문

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

설명

NULL일 수 있는 캡슐화된 인터페이스 포인터를 반환합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t::QueryInterface

아티클 • 2024. 11. 21.

Microsoft 전용

캡슐화된 인터페이스 포인터에서 `IUnknown` `QueryInterface` 멤버 함수를 호출합니다.

구문

```
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType*& p
) throw ( );
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType** p
) throw( );
```

매개 변수

iid

`IID` 인터페이스 포인터의

p

원시 인터페이스 포인터입니다.

설명

지정된 `IID` 인터페이스 포인터를 사용하여 캡슐화된 인터페이스 포인터를 호출 `IUnknown::QueryInterface`하고 결과 원시 인터페이스 포인터를 *p*로 반환합니다. 이 루틴은 HRESULT를 반환하여 성공 또는 실패를 나타냅니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#) | Microsoft Q&A에서 도움말 보기

`_com_ptr_t::Release`

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 인터페이스 포인터의 `IUnknown` `Release` 멤버 함수를 호출합니다.

구문

C++

```
void Release( );
```

설명

캡슐화된 인터페이스 포인터를 호출 `IUnknown::Release` 하여 이 인터페이스 포인터가 `E_POINTER` NULL이면 오류가 발생합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t 연산자

아티클 • 2023. 10. 12.

연산자 `_com_ptr_t` 에 대한 자세한 내용은 `_com_ptr_t` 클래스를 참조 [하세요](#).

참고 항목

[_com_ptr_t 클래스](#)

`_com_ptr_t::operator =`

아티클 • 2023. 10. 12.

Microsoft 전용

기존 `_com_ptr_t` 개체에 새 값을 할당합니다.

구문

```
template<typename _OtherIID>
_com_ptr_t& operator=( const _com_ptr_t<_OtherIID>& p );

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=(_InterfaceType* p );

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=( Interface* pInterface ) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=( const _com_ptr_t& cp ) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=( int null );

// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=( const _variant_t& varSrc );
```

설명

이 `_com_ptr_t` 개체에 인터페이스 포인터를 할당합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t 관계형 연산자

아티클 • 2023. 10. 12.

Microsoft 전용

스마트 포인터 객체를 다른 스마트 포인터, 원시 인터페이스 포인터 또는 NULL과 비교합니다.

구문

```
template<typename _OtherIID>
bool operator==( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator==( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator==( _InterfaceType* p );

template<>
bool operator==( Interface* p );

template<>
bool operator==( const _com_ptr_t& p ) throw();

template<>
bool operator==( _com_ptr_t& p ) throw();

bool operator==( Int null );

template<typename _OtherIID>
bool operator!=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator!=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator!=( _InterfaceType* p );

bool operator!=( Int null );

template<typename _OtherIID>
bool operator<( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<( _com_ptr_t<_OtherIID>& p );
```

```
template<typename _InterfaceType>
bool operator<( _InterfaceType* p );

template<typename _OtherIID>
bool operator>( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>(_com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>( _InterfaceType* p );

template<typename _OtherIID>
bool operator<=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<=( _InterfaceType* p );

template<typename _OtherIID>
bool operator>=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>=( _InterfaceType* p );
```

설명

스마트 포인터 객체를 다른 스마트 포인터, 원시 인터페이스 포인터 또는 NULL과 비교합니다. NULL 포인터 테스트를 제외하고, 이러한 연산자는 먼저 두 포인터를 `IUnknown` 쿼리하고 결과를 비교합니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_com_ptr_t 추출기

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 COM 인터페이스 포인터를 추출합니다.

구문

C++

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

설명

- **operator Interface*** NULL일 수 있는 캡슐화된 인터페이스 포인터를 반환합니다.
- **operator Interface&** 캡슐화된 인터페이스 포인터에 대한 참조를 반환하고 포인터가 NULL인 경우 오류를 발생합니다.
- **operator*** 스마트 포인터 개체가 역참조될 때 실제 캡슐화된 인터페이스처럼 작동할 수 있습니다.
- **operator->** 스마트 포인터 개체가 역참조될 때 실제 캡슐화된 인터페이스처럼 작동할 수 있습니다.
- **operator&** 캡슐화된 인터페이스 포인터를 NULL로 바꿔서 캡슐화된 포인터의 주소를 반환합니다. 이 연산자를 사용하면 주소별로 스마트 포인터를 인터페이스 포인터를 반환하는 **out 매개 변수**가 있는 함수에 전달할 수 있습니다.
- **operator bool** 조건식에서 스마트 포인터 개체를 사용할 수 있습니다. 포인터가 NULL이 아니면 이 연산자가 반환 **true** 됩니다.

① 참고

로 `operator bool` 선언되지 `explicit _com_ptr_t` 않으므로 암시적으로 모든 스칼라 형식으로 `bool` 변환할 수 있는 로 변환할 수 있습니다. 이렇게 하면 코드에 예기치 않은 결과가 발생할 수 있습니다. 이 변환을 의도하지 않게 사용하지 않도록 컴파일러 경고(수준 4) C4800을 사용하도록 설정합니다.

참고 항목

[_com_ptr_t 클래스](#)

관계형 함수 템플릿

아티클 • 2023. 10. 12.

Microsoft 전용

구문

```
template<typename _InterfaceType> bool operator==(  
    int NULL,  
    _com_ptr_t<_InterfaceType>& p  
);  
template<typename _Interface,  
        typename _InterfacePtr> bool operator==(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
);  
template<typename _Interface> bool operator!=(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
);  
template<typename _Interface,  
        typename _InterfacePtr> bool operator!=(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
);  
template<typename _Interface> bool operator<(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
);  
template<typename _Interface,  
        typename _InterfacePtr> bool operator<(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
);  
template<typename _Interface> bool operator>(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
);  
template<typename _Interface,  
        typename _InterfacePtr> bool operator>(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
);  
template<typename _Interface> bool operator<=(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
);  
template<typename _Interface,
```

```
typename _InterfacePtr> bool operator<=(
    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>=(
    int NULL,
    _com_ptr_t<_Interface>& p
);
template<typename _Interface,
         typename _InterfacePtr> bool operator>=(
    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);
```

매개 변수

i

원시 인터페이스 포인터입니다.

p

스마트 포인터입니다.

설명

이러한 함수 템플릿을 사용하면 비교 연산자 오른쪽에 있는 스마트 포인터와 비교할 수 있습니다. 이러한 템플릿은 `_com_ptr_t`의 멤버 함수가 아닙니다.

Microsoft 전용 종료

참고 항목

[_com_ptr_t 클래스](#)

_variant_t 클래스

아티클 • 2024. 11. 21.

Microsoft 전용

_variant_t 개체는 데이터 형식을 VARIANT 캡슐화합니다. 클래스는 리소스 할당 및 할당 취소를 관리하고 적절하게 함수를 VariantInit 호출합니다 VariantClear .

공사

[+] 테이블 확장

속성	설명
_variant_t	_variant_t 개체를 생성합니다.

작업

[+] 테이블 확장

속성	설명
Attach	개체를 VARIANT _variant_t 개체에 연결합니다.
지우기	캡슐화된 VARIANT 개체를 지웁니다.
ChangeType	_variant_t 개체의 형식을 표시된 개체로 변경합니다 VARTYPE .
Detach	캡슐화된 VARIANT 개체를 이 _variant_t 개체에서 분리합니다.
SetString	이 _variant_t 개체에 문자열을 할당합니다.

연산자

[+] 테이블 확장

속성	설명
연산자 =	기존 _variant_t 개체에 새 값을 할당합니다.
operator ==, !=	두 _variant_t 개체를 같음 또는 같지 않은지 비교합니다.
추출기	캡슐화된 VARIANT 개체에서 데이터를 추출합니다.

요구 사항

헤더:<comutil.h>

Lib: comsuppw.lib 또는 comsuppwd.lib(자세한 내용은 /Zc:wchar_t(wchar_t 네이티브 형식) 참조)

참고 항목

[컴파일러 COM 지원 클래스](#)

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

_variant_t 멤버 함수

아티클 • 2023. 10. 12.

[_variant_t 멤버 함수에 대한 자세한 내용은 _variant_t 클래스를 참조하세요.](#)

참고 항목

[_variant_t 클래스](#)

_variant_t::_variant_t

아티클 • 2023. 10. 12.

Microsoft 전용

_variant_t 개체를 생성합니다.

구문

C++

```
_variant_t( ) throw( );

_variant_t(
    const VARIANT& varSrc
);

_variant_t(
    const VARIANT* pVarSrc
);

_variant_t(
    const _variant_t& var_t_src
);

_variant_t(
    VARIANT& varSrc,
    bool fCopy
);

_variant_t(
    short sSrc,
    VARTYPE vtSrc = VT_I2
);

_variant_t(
    long lSrc,
    VARTYPE vtSrc = VT_I4
);

_variant_t(
    float fltSrc
) throw( );

_variant_t(
    double dblSrc,
    VARTYPE vtSrc = VT_R8
);

_variant_t(
```

```
    const CY& cySrc
) throw( );

_variant_t(
    const _bstr_t& bstrSrc
);

_variant_t(
    const wchar_t *wstrSrc
);

_variant_t(
    const char* strSrc
);

_variant_t(
    IDispatch* pDispSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    bool bSrc
) throw( );

_variant_t(
    IUnknown* pIUnknownSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    const DECIMAL& decSrc
) throw( );

_variant_t(
    BYTE bSrc
) throw( );

variant_t(
    char cSrc
) throw();

_variant_t(
    unsigned short usSrc
) throw();

_variant_t(
    unsigned long ulSrc
) throw();

_variant_t(
    int iSrc
) throw();

_variant_t(
    unsigned int uiSrc
)
```

```
) throw();

__variant_t(
    __int64 i8Src
) throw();

__variant_t(
    unsigned __int64 ui8Src
) throw();
```

매개 변수

`varSrc`

새 `VARIANT` 개체로 복사할 `_variant_t` 개체입니다.

`pVarSrc`

새 `_variant_t` 개체에 `VARIANT` 복사할 개체에 대한 포인터입니다.

`var_t_Src`

새 `_variant_t` 개체로 복사할 `_variant_t` 개체입니다.

`fCopy`

이면 `false` 제공된 `VARIANT` 개체가 새 복사본을 만들지 않고 새 `_variant_t` 개체에 연결됩니다 `VariantCopy`.

`ISrc, sSrc`

새 `_variant_t` 개체로 복사할 정수 값입니다.

`vtSrc`

새 `_variant_t` 개체의 경우입니다 `VARTYPE`.

`fltSrc, dblSrc`

새 `_variant_t` 개체에 복사될 숫자 값입니다.

`cySrc`

새 `CY` 개체로 복사할 `_variant_t` 개체입니다.

`bstrSrc`

새 `_bstr_t` 개체로 복사할 `_variant_t` 개체입니다.

`strSrc, wstrSrc`

새 `_variant_t` 개체로 복사할 문자열입니다.

bSrc

새 `bool` 개체로 복사할 `_variant_t` 값입니다.

pIUnknownSrc

새 `_variant_t` 개체에 캡슐화할 VT_UNKNOWN 개체에 대한 COM 인터페이스 포인터입니다.

pDispSrc

새 `_variant_t` 개체에 캡슐화할 VT_DISPATCH 개체에 대한 COM 인터페이스 포인터입니다.

decSrc

새 `DECIMAL` 개체로 복사할 `_variant_t` 값입니다.

bSrc

새 `BYTE` 개체로 복사할 `_variant_t` 값입니다.

cSrc

새 `char` 개체로 복사할 `_variant_t` 값입니다.

usSrc

새 `unsigned short` 개체로 복사할 `_variant_t` 값입니다.

ulSrc

새 `unsigned long` 개체로 복사할 `_variant_t` 값입니다.

iSrc

`int` 새 `_variant_t` 개체에 복사할 값입니다.

uiSrc

`unsigned int` 새 `_variant_t` 개체에 복사할 값입니다.

i8Src

`_int64` 새 `_variant_t` 개체에 복사할 값입니다.

ui8Src

`unsigned _int64` 새 `_variant_t` 개체에 복사할 값입니다.

설명

- `_variant_t()` 빈 `_variant_t` 개체를 생성합니다 `VT_EMPTY`.

- `_variant_t(VARIANT& varSrc)` 개체의 `_variant_t` 복사본에서 개체를 `VARIANT` 생성합니다. 변형 형식이 유지됩니다.
- `_variant_t(VARIANT* pVarSrc)` 개체의 `_variant_t` 복사본에서 개체를 `VARIANT` 생성합니다. 변형 형식이 유지됩니다.
- `_variant_t(_variant_t& var_t_Src)` 다른 `_variant_t` 개체에서 개체를 `_variant_t` 생성합니다. 변형 형식이 유지됩니다.
- `_variant_t(VARIANT& varSrc, bool fCopy)` `_variant_t` 기존 `VARIANT` 개체에서 개체를 생성합니다. 이 `false` 경우 `fCopy` 개체는 `VARIANT` 복사본을 만들지 않고 새 개체에 연결됩니다.
- `_variant_t(short sSrc, VARTYPE vtSrc = VT_I2)` `_variant_t` 형식 `VT_I2` 또는 `VT_BOOL` 정수 값에서 개체를 `short` 생성합니다. 다른 `VARTYPE` 모든 결과는 오류가 발생합니다 `E_INVALIDARG`.
- `_variant_t(long lSrc, VARTYPE vtSrc = VT_I4)` `_variant_t` 형식 `VT_I4` `VT_BOOL` 또는 정수 값에서 개체를 `long` 생성합니다. 다른 `VARTYPE` 모든 결과는 오류가 발생합니다 `E_INVALIDARG`.
- `_variant_t(float fltSrc)` `_variant_t` 숫자 값에서 형식 `VT_R4`의 개체를 `float` 생성합니다.
- `_variant_t(double dblSrc, VARTYPE vtSrc = VT_R8)` `_variant_t` 형식 `VT_R8` 또는 `VT_DATE` 숫자 값에서 개체를 `double` 생성합니다. 다른 `VARTYPE` 모든 결과는 오류가 발생합니다 `E_INVALIDARG`.
- `_variant_t(CY& cySrc)` 개체에서 `_variant_t` 형식 `VT_CY`의 개체를 `CY` 생성합니다.
- `*_variant_t(_bstr_t& bstrSrc)`* 개체에서 형식 `VT_BSTR`의 개체를 `_bstr_t` 생성 `_variant_t` 합니다. 새 `BSTR`이 할당됩니다.
- `_variant_t(wchar_t* wstrSrc)` `_variant_t` 유니코드 문자열에서 형식 `VT_BSTR`의 개체를 생성합니다. 새 `BSTR`이 할당됩니다.
- `_variant_t(char* strSrc)` 문자열에서 `_variant_t` 형식 `VT_BSTR`의 개체를 생성 합니다. 새 `BSTR`이 할당됩니다.
- `_variant_t(bool bSrc)` 값에서 `_variant_t` 형식 `VT_BOOL`의 개체를 `bool` 생성합니다.

- `_variant_t(IUnknown* pIUnknownSrc, bool fAddRef = true)` `_variant_t` COM 인터페이스 포인터에서 형식 `VT_UNKNOWN` 의 개체를 생성합니다. 이 경우 `fAddRef` 제공된 인터페이스 포인터에서 개체가 제거될 때 `_variant_t` 발생하는 호출과 일치하도록 `Release` 호출됩니다. `true` `AddRef` 제공된 인터페이스 포인터를 호출 `Release` 하는 것은 사용자에게 달려 있습니다. 이 `false` 경우 `fAddRef` 이 생성자는 제공된 인터페이스 포인터의 소유권을 사용합니다. 제공된 인터페이스 포인터를 호출 `Release` 하지 마세요.
- `_variant_t(IDispatch* pDispSrc, bool fAddRef = true)` `_variant_t` COM 인터페이스 포인터에서 형식 `VT_DISPATCH` 의 개체를 생성합니다. 이 경우 `fAddRef` 제공된 인터페이스 포인터에서 개체가 제거될 때 `_variant_t` 발생하는 호출과 일치하도록 `Release` 호출됩니다. `true` `AddRef` 제공된 인터페이스 포인터를 호출 `Release` 하는 것은 사용자에게 달려 있습니다. 이 `false` 경우 `fAddRef` 이 생성자는 제공된 인터페이스 포인터의 소유권을 사용합니다. 제공된 인터페이스 포인터를 호출 `Release` 하지 마세요.
- `_variant_t(DECIMAL& decSrc)` 값에서 `_variant_t` 형식 `VT_DECIMAL` 의 개체를 `DECIMAL` 생성합니다.
- `_variant_t(BYTE bSrc)` 값에서 `_variant_t` 형식 `VT_UI1` 의 개체를 `BYTE` 생성합니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

_variant_t::Attach

아티클 • 2023. 10. 12.

Microsoft 전용

개체를 VARIANT _variant_t 개체에 연결합니다.

구문

C++

```
void Attach(VARIANT& varSrc);
```

매개 변수

varSrc

VARIANT 이 _variant_t 개체에 연결할 개체입니다.

설명

캡슐화하여 소유권 VARIANT 을 가져옵니다. 이 멤버 함수는 캡슐화된 기존 함수를 해제한 VARIANT 다음 제공된 VARIANT 리소스를 복사하고 VT_EMPTY 설정 VARTYPE 하여 리소스를 _variant_t 소멸자만 해제할 수 있도록 합니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

`_variant_t::Clear`

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 `VARIANT` 개체를 지웁니다.

구문

C++

```
void Clear( );
```

설명

캡슐화된 `VARIANT` 개체를 호출 `VariantClear` 합니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

_variant_t::ChangeType

아티클 • 2023. 10. 12.

Microsoft 전용

개체의 형식을 `_variant_t` 표시된 형식으로 변경합니다 `VARTYPE`.

구문

C++

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

매개 변수

Vartype

이 `_variant_t` 개체의 경우입니다 `VARTYPE`.

pSrc

변환할 `_variant_t` 개체의 포인터입니다. 이 값이 `NULL`이면 변환이 수행됩니다.

설명

이 멤버 함수는 개체를 `_variant_t` 지정된 `VARTYPE` 개체로 변환합니다. `pSrc`가 `NULL`이면 변환이 수행되고, 그렇지 않으면 이 `_variant_t` 개체가 `pSrc`에서 복사된 다음 변환됩니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

`_variant_t::Detach`

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 `VARIANT` 개체를 이 `_variant_t` 개체에서 분리합니다.

구문

```
VARIANT Detach( );
```

Return Value

캡슐화된 `VARIANT`.

설명

캡슐화된 `VARIANT` 개체를 추출하고 반환한 다음 삭제하지 않고 이 `_variant_t` 개체를 줍니다. 이 멤버 함수는 `VARIANT` 캡슐화에서 제거하고 이 `_variant_t` 개체의 `VT_EMPTY` 설정합니다 `VARTYPE`. `VariantClear` 함수를 호출하여 반환 `VARIANT` 된 값을 해제하는 것은 사용자에게 달려 있습니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

`_variant_t::SetString`

아티클 • 2023. 10. 12.

Microsoft 전용

이 `_variant_t` 개체에 문자열을 할당합니다.

구문

C++

```
void SetString(const char* pSrc);
```

매개 변수

pSrc

문자열에 대한 포인터입니다.

설명

ANSI 문자열을 유니코드 `BSTR` 문자열로 변환하고 이 `_variant_t` 개체에 할당합니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

_variant_t 연산자

아티클 • 2023. 10. 12.

_variant_t 연산자에 대한 자세한 내용은 _variant_t 클래스를 참조[하세요](#).

참고 항목

[_variant_t 클래스](#)

_variant_t::operator=

아티클 • 2023. 10. 12.

인스턴스에 새 값을 `_variant_t` 할당합니다.

`_variant_t` 클래스와 해당 `operator=` 멤버는 Microsoft 전용입니다.

구문

C++

```
_variant_t& operator=( const VARIANT& varSrc );
_variant_t& operator=( const VARIANT* pVarSrc );
_variant_t& operator=( const _variant_t& var_t_src );
_variant_t& operator=( short sSrc );
_variant_t& operator=( long lSrc );
_variant_t& operator=( float fltSrc );
_variant_t& operator=( double dblSrc );
_variant_t& operator=( const CY& cySrc );
_variant_t& operator=( const _bstr_t& bstrSrc );
_variant_t& operator=( const wchar_t* wstrSrc );
_variant_t& operator=( const char* strSrc );
_variant_t& operator=( IDispatch* pDispSrc );
_variant_t& operator=( bool bSrc );
_variant_t& operator=( IUnknown* pSrc );
_variant_t& operator=( const DECIMAL& decSrc );
_variant_t& operator=( BYTE byteSrc );
_variant_t& operator=( char cSrc );
_variant_t& operator=( unsigned short usSrc );
_variant_t& operator=( unsigned long ulSrc );
_variant_t& operator=( int iSrc );
_variant_t& operator=( unsigned int uiSrc );
_variant_t& operator=( __int64 i8Src );
_variant_t& operator=( unsigned __int64 ui8Src );
```

매개 변수

`varSrc`

콘텐츠 및 `VT_*` 형식을 `VARIANT` 복사할 원본에 대한 참조입니다.

`pVarSrc`

콘텐츠 및 `VT_*` 형식을 `VARIANT` 복사할 포인터입니다.

`var_t_src`

콘텐츠 및 `VT_*` 형식을 `_variant_t` 복사할 원본에 대한 참조입니다.

`sSrc`

`short` 복사할 정수 값입니다. 지정된 형식 `VT_BOOL` 이 형식인 경우 `*this .VT_BOOL` 그렇지 않으면 지정된 형식 `VT_I2`입니다.

`lSrc`

`long` 복사할 정수 값입니다. 지정된 형식 `VT_BOOL` 이 형식인 경우 `*this .VT_BOOL` 지정된 형식 `VT_ERROR` 이 형식인 경우 `*this .VT_ERROR` 그렇지 않으면 지정된 형식 `VT_I4`입니다.

`fltSrc`

`float` 복사할 숫자 값입니다. 지정된 형식 `VT_R4`.

`dblSrc`

`double` 복사할 숫자 값입니다. 지정된 형식 `VT_DATE` 이 형식인 경우 `this .VT_DATE` 그렇지 않으면 지정된 형식 `VT_R8`입니다.

`cySrc`

복사할 `CY` 개체입니다. 지정된 형식 `VT_CY`.

`bstrSrc`

복사할 `BSTR` 개체입니다. 지정된 형식 `VT_BSTR`.

`wstrSrc`

복사할 유니코드 문자열로, 지정된 형식 `VT_BSTR`으로 `BSTR` 저장됩니다.

`strSrc`

복사할 멀티바이트 문자열로, 지정된 형식 `VT_BSTR`으로 `BSTR` 저장됩니다.

`pDispSrc`

`IDispatch` 호출을 사용하여 복사할 `AddRef` 포인터입니다. 지정된 형식 `VT_DISPATCH`.

`bSrc`

`bool` 복사할 값입니다. 지정된 형식 `VT_BOOL`.

`pSrc`

`IUnknown` 호출을 사용하여 복사할 `AddRef` 포인터입니다. 지정된 형식 `VT_UNKNOWN`.

`decSrc`

복사할 `DECIMAL` 개체입니다. 지정된 형식 `VT_DECIMAL`.

`byteSrc`

`BYTE` 복사할 값입니다. 지정된 형식 `VT_UI1`.

`cSrc`

`char` 복사할 값입니다. 지정된 형식 `VT_I1`.

`usSrc`

`unsigned short` 복사할 값입니다. 지정된 형식 `VT_UI2`.

`ulSrc`

`unsigned long` 복사할 값입니다. 지정된 형식 `VT_UI4`.

`iSrc`

`int` 복사할 값입니다. 지정된 형식 `VT_INT`.

`uiSrc`

`unsigned int` 복사할 값입니다. 지정된 형식 `VT_UINT`.

`i8Src`

`_int64` 복사할 값 또는 `long long` 값입니다. 지정된 형식 `VT_I8`.

`ui8Src`

`unsigned __int64` 복사할 값 또는 `unsigned long long` 값입니다. 지정된 형식 `VT_UI8`.

설명

`operator=` 대입 연산자는 개체 형식을 삭제하거나 형식을 호출

`Release IDispatch* IUnknown*` 하는 기존 값을 지웁니다. 그런 다음 개체에 새 값을 `_variant_t` 복사합니다. 지정된 값과 일치하도록 형식을 변경 `_variant_t` 합니다(예: <`long a0/>` 및 `double` 인수에 대해 `short` 언급된 경우 제외). 값 형식은 직접 복사됩니다. A `VARIANT` 또는 포인터 또는 `_variant_t` 참조 인수는 할당된 개체의 내용과 형식을 복사합니다. 다른 포인터 또는 참조 형식 인수는 할당된 개체의 복사본을 만듭니다. 대입 연산자는 인수를 `IDispatch*` 호출 `AddRef` 합니다 `IUnknown*`.

`operator=` 는 `_com_raise_error` 오류가 발생하면 호출합니다.

`operator=` 는 업데이트 `_variant_t` 된 개체에 대한 참조를 반환합니다.

참고 항목

[_variant_t 클래스](#)

`_variant_t` 관계형 연산자

아티클 • 2023. 10. 12.

Microsoft 전용

두 `_variant_t` 개체가 같음 또는 같지 않음을 비교합니다.

구문

```
bool operator==(  
    const VARIANT& varSrc) const;  
bool operator==(  
    const VARIANT* pSrc) const;  
bool operator!=(  
    const VARIANT& varSrc) const;  
bool operator!=(  
    const VARIANT* pSrc) const;
```

매개 변수

`varSrc`

개체와 `_variant_t` 비교할 A `VARIANT`입니다.

`pSrc`

`VARIANT` 개체와 비교할 포인터입니다 `_variant_t`.

Return Value

비교가 유지되면 반환 `true` 합니다(없는 경우) `false`.

설명

개체를 `_variant_t` 같음 또는 같지 않은지 테스트하는 `VARIANT` 개체와 비교합니다.

Microsoft 전용 종료

참고 항목

variant_t 클래스

_variant_t Extractors

아티클 • 2023. 10. 12.

Microsoft 전용

캡슐화된 VARIANT 개체에서 데이터를 추출합니다.

구문

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

설명

캡슐화된 VARIANT에서 원시 데이터를 추출합니다. VARIANT 올바른 형식 VariantChangeType 이 아닌 경우 변환을 시도하는 데 사용되며 실패 시 오류가 발생합니다.

- operator short() 정수 값을 추출합니다 short .
- operator long() 정수 값을 추출합니다 long .
- operator float() 숫자 값을 추출합니다 float .
- operator double() 정수 값을 추출합니다 double .

- **operator CY()** 개체를 추출합니다 `CY`.
- **operator bool()** 값을 추출합니다 `bool`.
- **operator DECIMAL()** 값을 추출합니다 `DECIMAL`.
- **operator BYTE()** 값을 추출합니다 `BYTE`.
- 연산자 `_bstr_t()` 개체에 `_bstr_t` 캡슐화된 문자열을 추출합니다.
- **operator IDispatch*()** 캡슐화된 `VARIANT`에서 dispinterface 포인터를 추출합니다. `AddRef` 는 결과 포인터에서 호출되므로 해제를 호출 `Release` 하는 것은 사용자에게 달려 있습니다.
- 연산자 `IUnknown*()` 캡슐화된 `VARIANT`에서 COM 인터페이스 포인터를 추출합니다. `AddRef` 는 결과 포인터에서 호출되므로 해제를 호출 `Release` 하는 것은 사용자에게 달려 있습니다.

Microsoft 전용 종료

참고 항목

[_variant_t 클래스](#)

Microsoft 확장명

아티클 • 2024. 11. 21.

:

```
__asm assembly-instruction ; opt  
__asm { assembly-instruction-list } ; opt
```

:

```
assembly-instruction ; opt  
assembly-instruction ; assembly-instruction-list ; opt
```

:

```
ms-modifier ms-modifier-list opt
```

:

```
__cdecl  
__fastcall  
__stdcall  
__syscall(향후 구현을 위해 예약됨)  
__oldcall(향후 구현을 위해 예약됨)  
__unaligned(향후 구현을 위해 예약됨)  
based-modifier
```

:

```
__based ( based-type )
```

:

```
name
```

피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) | [Microsoft Q&A에서 도움말 보기](#)

비표준 동작

아티클 • 2024. 01. 05.

다음 섹션에서는 Microsoft C++ 구현이 C++ 표준을 준수하지 않는 일부 위치를 나열합니다. 아래에 제공된 섹션 번호는 C++11 표준(ISO/IEC 14882:2011(E))의 섹션 번호를 참조합니다.

C++ 표준에 정의된 것과 다른 컴파일러 제한 목록은 컴파일러 제한에 [제공됩니다](#).

공변 반환 형식

가상 함수에 개수가 가변적인 인수가 있을 때 가상 기본 클래스는 공변(covariant) 반환 형식으로 지원되지 않습니다. 이는 C++11 ISO 사양의 섹션 10.3, 단락 7을 따르지 않습니다. 다음 샘플은 컴파일되지 않습니다. 컴파일러 오류 [C2688](#)을 생성합니다.

```
C++

// CovariantReturn.cpp
class A
{
    virtual A* f(int c, ...); // remove ...
};

class B : virtual A
{
    B* f(int c, ...); // C2688 remove ...
};
```

템플릿에서 독립적인 이름 바인딩

Microsoft C++ 컴파일러는 템플릿을 처음 구문 분석할 때 현재 종속되지 않는 이름 바인딩을 지원하지 않습니다. 이는 C++11 ISO 사양의 섹션 14.6.3을 따르지 않습니다. 이로 인해 템플릿이 확인되고 인스턴스화되기 전에 오버로드가 선언될 수 있습니다.

```
C++

#include <iostream>
using namespace std;

namespace N {
    void f(int) { cout << "f(int)" << endl; }
}

template <class T> void g(T) {
```

```

N::f('a');    // calls f(char), should call f(int)
}

namespace N {
    void f(char) { cout << "f(char)" << endl;}
}

int main() {
    g('c');
}
// Output: f(char)

```

함수 예외 지정자

`throw()` 이외의 함수 예외 지정자는 구문 분석되지만 사용되지 않습니다. ISO C++11 사양의 섹션 15.4를 따르지 않습니다. 예시:

C++

```

void f() throw(int); // parsed but not used
void g() throw();    // parsed and used

```

예외 사양에 대한 자세한 내용은 예외 사양을 참조 [하세요](#).

char_traits::eof()

C++ 표준은 `char_traits::eof`가 유효한 `char_type` 값에 해당해서는 안 됨을 명시합니다. Microsoft C++ 컴파일러는 형식에 대해 이 제약 조건을 적용하지만 형식 `char wchar_t`에는 적용되지 않습니다. 이는 C++11 ISO 사양의 12.1.1 섹션에 있는 표 62의 요구 사항을 준수하지 않습니다. 아래 예제에서는 이 동작을 보여 줍니다.

C++

```

#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}

```

개체 스토리지 위치

C++ 표준(단원 1.8, 6항)에서는 전체 C++ 개체에 고유한 스토리지 위치가 있어야 합니다. 그러나 Microsoft C++에서는 데이터 멤버가 없는 형식이 개체의 수명 동안 다른 형식과 스토리지 위치를 공유하는 경우가 있습니다.

컴파일러 한계

아티클 • 2023. 10. 12.

C++ 표준에서는 다양한 언어 구문에 대한 한도를 권장합니다. 다음은 Microsoft C++ 컴파일러가 권장 제한을 구현하지 않는 경우의 목록입니다. 첫 번째 숫자는 ISO C++ 11 표준(INCITS/ISO/IEC 14882-2011[2012], 부록 B)에 설정된 제한이며 두 번째 숫자는 Microsoft C++ 컴파일러에서 구현하는 제한입니다.

- 복합 문, 반복 제어 구조 및 선택 컨트롤 구조 중첩 수준 - C++ 표준: 256, Microsoft C++ 컴파일러: 중첩된 문의 조합에 따라 달라지지만 일반적으로 100에서 110 사이입니다.
- 한 매크로 정의의 매개 변수 - C++ 표준: 256, :127을 사용 `/zc:preprocessor-`하거나 :32767을 사용하는 `/zc:preprocessor` Microsoft C++ 컴파일러.
- 한 매크로 호출의 인수 - C++ 표준: 256, :127을 사용 `/zc:preprocessor-`하거나 :32767을 사용하는 `/zc:preprocessor` Microsoft C++ 컴파일러.
- 문자열 리터럴 또는 와이드 문자열 리터럴의 문자(연결 후) - C++ 표준: 65536, Microsoft C++ 컴파일러: NULL 종결자를 포함한 65535 싱글 바이트 문자 및 NULL 종결자를 포함한 32767 더블 바이트 문자.
- 단일 `struct-declaration-list` C++ 표준의 중첩 클래스, 구조체 또는 공용 구조체 정의 수준: 256, Microsoft C++ 컴파일러: 16.
- 생성자 정의의 멤버 이니셜라이저 - C++ 표준: 6144, Microsoft C++ 컴파일러: 6144 이상.
- 한 식별자의 범위 자격 - C++ 표준: 256, Microsoft C++ 컴파일러: 127.
- 중첩 사양 `extern` - C++ 표준: 1024, Microsoft C++ 컴파일러: 9(전역 범위에서 암시적 `extern` 사양을 계산하지 않음, 전역 범위에서 암시적 `extern` 사양을 계산하는 경우 10)
- 템플릿 선언의 템플릿 인수 - C++ 표준: 1024, Microsoft C++ 컴파일러: 2046.

참고 항목

[비표준 동작](#)

C/C++ 전처리기 참조

아티클 • 2023. 06. 16.

C/C++ 전처리기 참조는 Microsoft C/C++에서 구현되는 전처리기를 설명합니다. 전처리기는 C 및 C++ 파일이 컴파일러로 전달되기 전에 해당 파일에 대한 준비 작업을 수행합니다. 전처리기를 사용하여 조건에 따라 코드를 컴파일하고, 파일을 삽입하고, 컴파일 시간 오류 메시지를 지정하고, 코드 섹션에 시스템별 규칙을 적용할 수 있습니다.

Visual Studio 2019에서 [/Zc:preprocessor](#) 컴파일러 옵션은 완전히 호환되는 C11 및 C17 전처리기를 제공합니다. 컴파일러 플래그 `/std:c11` 또는 `/std:c17`를 사용하는 경우 기본값입니다.

섹션 내용

전처리기

기존 및 새로운 준수 전처리기에 대한 개요를 제공합니다.

전처리기 지시문

여러 실행 환경에서 소스 프로그램을 변경하기 쉽고 컴파일하기 용이하게 만들기 위해 일반적으로 사용되는 지시문에 대해 설명합니다.

전처리기 연산자

`#define` 지시문의 컨텍스트에서 사용되는 네 가지 전처리기 관련 연산자에 대해 설명합니다.

미리 정의된 매크로

C 및 C++ 표준 및 Microsoft C++에서 지정한 미리 정의된 매크로에 대해 설명합니다.

pragma

각 컴파일러가 C 및 C++ 언어와 전반적인 호환성을 유지하면서 시스템별 기능과 운영 체제별 기능을 제공하는 데 사용되는 pragma에 대해 설명합니다.

관련 단원

C++ 언어 참조

C++ 언어의 Microsoft 구현에 대한 참조 자료를 제공합니다.

C 언어 참조

C 언어의 Microsoft 구현에 대한 참조 자료를 제공합니다.

C/C++ 빌드 참조

컴파일러 및 링커 옵션에 대해 설명하는 항목의 링크를 제공합니다.

[Visual Studio 프로젝트 - C++](#)

프로젝트 시스템에서 C++ 프로젝트용 파일을 찾기 위해 검색할 디렉터리를 지정할 수 있도록 하는 Visual Studio 사용자 인터페이스에 대해 설명합니다.

C++ 표준 라이브러리 참조(STL)

아티클 • 2023. 10. 12.

C++ 프로그램은 C++ 표준 라이브러리에 따른 이러한 구현에서 여러 함수를 호출할 수 있습니다. 이러한 함수는 입력 및 출력과 같은 서비스를 수행하고 자주 사용되는 작업의 효율적인 구현을 제공합니다.

적절한 Visual C++ 런타임 파일과 연결하는 방법에 대한 자세한 내용은 C 런타임 .lib(CRT) 및 C++ STL(표준 라이브러리) .lib 파일을 참조[하세요](#).

① 참고

Microsoft의 C++ 표준 라이브러리 구현을 STL 또는 표준 템플릿 라이브러리라고도 합니다. C++ 표준 라이브러리는 ISO 14882에 정의된 대로 라이브러리의 공식 이름이지만 검색 엔진에서 "STL" 및 "표준 템플릿 라이브러리"가 널리 사용되기 때문에 이러한 이름을 사용하여 설명서를 더 쉽게 찾을 수 있습니다.

역사적 관점에서, "STL" 원래 알렉산더 스테파노프에 의해 작성된 표준 템플릿 라이브러리를 참조. 해당 라이브러리의 일부는 ISO C 런타임 라이브러리, Boost 라이브러리의 일부 및 기타 기능과 함께 C++ 표준 라이브러리에서 표준화되었습니다. 때때로 "STL"은 Stepanov의 STL에서 조정된 C++ 표준 라이브러리의 컨테이너 및 알고리즘 부분을 참조하는 데 사용됩니다. 이 설명서에서 STL(표준 템플릿 라이브러리)은 C++ 표준 라이브러리 전체를 참조합니다.

이 섹션의 내용

[C++ 표준 라이브러리 개요](#)는 C++ 표준 라이브러리의 Microsoft 구현에 대한 개요를 제공합니다.

[iostream 프로그래밍 프로그래밍](#)에 대한 개요를 `iostream` 제공합니다.

[헤더 파일 참조](#) 코드 예제와 함께 C++ 표준 라이브러리 헤더 파일에 대한 참조 항목에 대한 링크를 제공합니다.